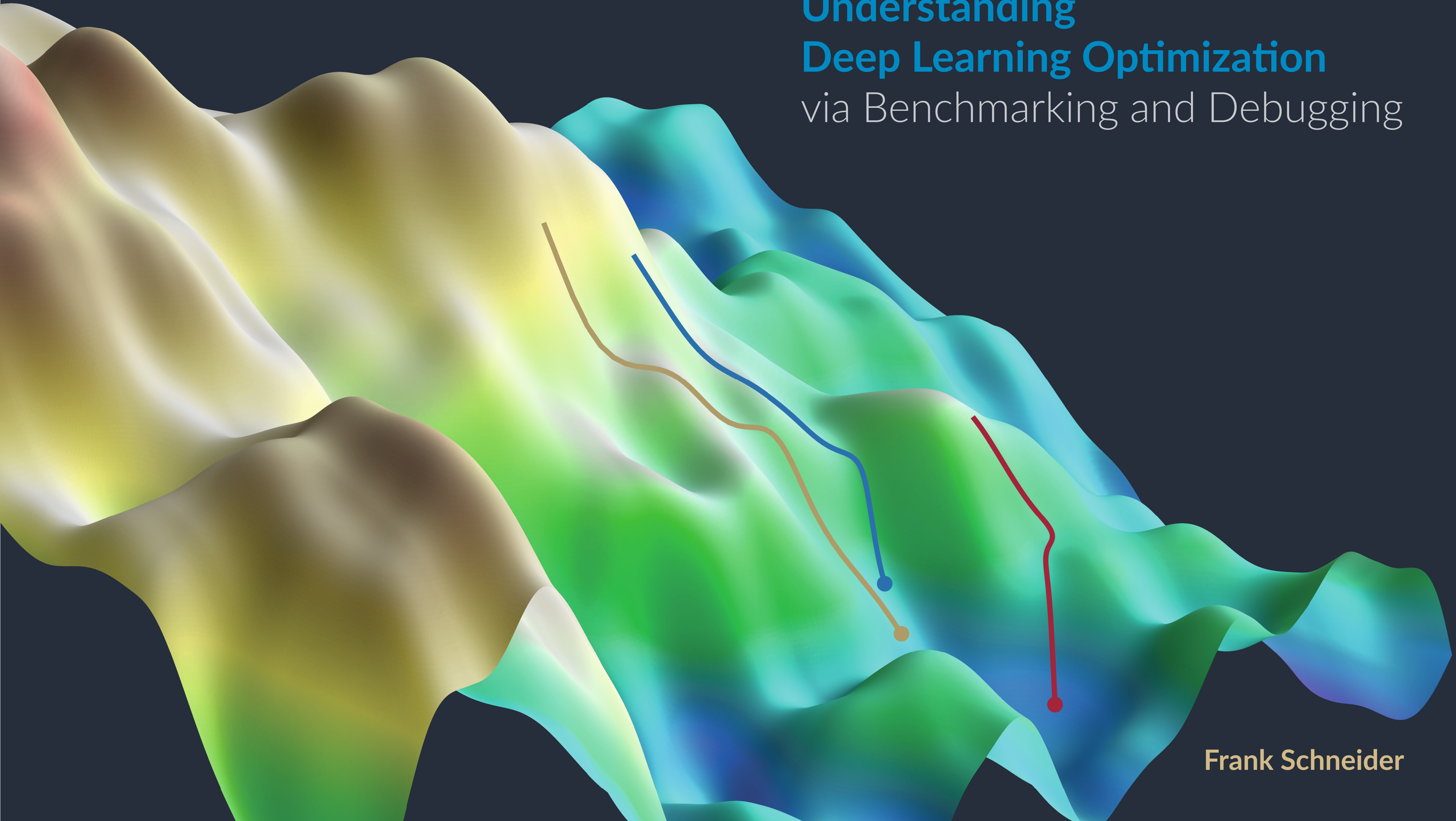
Understanding Deep Learning Optimization
via Benchmarking and Debugging



Frank Schneider

Understanding Deep Learning Optimization via Benchmarking and Debugging

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Frank Schneider
aus Stuttgart

Tübingen
2022

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	29.07.2022
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Philipp Hennig
2. Berichterstatter:	Prof. Dr. Hendrik Lensch

Abstract

The central paradigm of machine learning (ML) is the idea that computers can *learn* the strategies needed to solve a task without being explicitly programmed to do so. The hope is that given data, computers can recognize underlying patterns and figure out how to perform tasks without extensive human oversight. To achieve this, many machine learning problems are framed as minimizing a loss function, which makes optimization methods a core part of training ML models.

Machine learning and in particular deep learning is often perceived as a cutting-edge technology, the underlying optimization algorithms, however, tend to resemble rather simplistic, even archaic methods. Crucially, they rely on extensive human intervention to successfully train modern neural networks. One reason for this tedious, finicky, and lengthy training process lies in our insufficient understanding of optimization methods in the challenging deep learning setting. As a result, training neural nets, to this day, has the reputation of being more of an art form than a science and requires a level of human assistance that runs counter to the core principle of ML.

Although hundreds of optimization algorithms for deep learning have been proposed, there is no widely agreed-upon protocol for evaluating their performance. Without a standardized and independent evaluation protocol, it is difficult to reliably demonstrate the usefulness of novel methods. In this thesis, we present strategies for quantitatively and reproducibly comparing deep learning optimizers in a meaningful way. This protocol considers the unique challenges of deep learning such as the inherent stochasticity or the crucial distinction between *learning* and pure *optimization*. It is formalized and automatized in the Python package DeepOBS and allows fairer, faster, and more convincing empirical comparisons of deep learning optimizers.

Based on this benchmarking protocol, we compare fifteen popular deep learning optimizers to gain insight into the field's current state. To provide evidence-backed heuristics for choosing among the growing list of optimization methods, we extensively evaluate them with roughly 50,000 training runs. Our benchmark indicates that the comparably traditional Adam optimizer remains a strong but not dominating contender and that newer methods fail to consistently outperform it.

In addition to the optimizer, other causes can impede neural network training, such as inefficient model architectures or hyperparameters. Traditional performance metrics, such as training loss or validation accuracy, can show *if* a model is learning or not, but not *why*. To provide this understanding and a glimpse into the black box of neural networks, we developed Cockpit, a debugging tool specifically for deep learning. It combines novel and proven observables into a live monitoring tool for practitioners. Among other findings, Cockpit reveals that well-tuned training runs consistently overshoot the local minimum, at least for significant portions of the training.

The use of thorough benchmarking experiments and tailored debugging tools improves our understanding of neural network training. In the absence of theoretical insights, these empirical results and practical tools are essential for guiding practitioners. More importantly, our results show that there is a need and a clear path for fundamentally different optimization methods to make deep learning more accessible, robust, and resource-efficient.

Zusammenfassung

Das zentrale Prinzip des maschinellen Lernens (ML) ist die Vorstellung, dass Computer die notwendigen Strategien zur Lösung einer Aufgabe *erlernen* können, ohne explizit dafür programmiert worden zu sein. Die Hoffnung ist, dass Computer anhand von Daten die zugrunde liegenden Muster erkennen und selbst feststellen, wie sie Aufgaben erledigen können, ohne dass sie dabei von Menschen geleitet werden müssen. Um diese Aufgabe zu erfüllen, werden viele Probleme des maschinellen Lernens als Minimierung einer Verlustfunktion formuliert. Daher sind Optimierungsverfahren ein zentraler Bestandteil des Trainings von ML-Modellen.

Obwohl das maschinelle Lernen und insbesondere das tiefe Lernen oft als innovative Spitzentechnologie wahrgenommen wird, basieren viele der zugrunde liegenden Optimierungsalgorithmen eher auf simplen, fast archaischen Verfahren. Um moderne neuronale Netze erfolgreich zu trainieren, bedarf es daher häufig umfangreicher menschlicher Unterstützung. Ein Grund für diesen mühsamen, umständlichen und langwierigen Trainingsprozess ist unser mangelndes Verständnis der Optimierungsmethoden im anspruchsvollen Rahmen des tiefen Lernens. Auch deshalb hat das Training neuronaler Netze bis heute den Ruf, eher eine Kunstform als eine echte Wissenschaft zu sein und erfordert ein Maß an menschlicher Beteiligung, welche dem Kernprinzip des maschinellen Lernens widerspricht.

Obwohl bereits Hunderte Optimierungsverfahren für das tiefe Lernen vorgeschlagen wurden, gibt es noch kein allgemein anerkanntes Protokoll zur Beurteilung ihrer Qualität. Ohne ein standardisiertes und unabhängiges Bewertungsprotokoll ist es jedoch schwierig, die Nützlichkeit neuartiger Methoden zuverlässig nachzuweisen. In dieser Arbeit werden Strategien vorgestellt, mit denen sich Optimierer für das tiefe Lernen quantitativ, reproduzierbar und aussagekräftig vergleichen lassen. Dieses Protokoll berücksichtigt die einzigartigen Herausforderungen des tiefen Lernens, wie etwa die inhärente Stochastizität oder die wichtige Unterscheidung zwischen *Lernen* und reiner *Optimierung*. Die Erkenntnisse sind im Python-Paket DeepOBS formalisiert und automatisiert, wodurch gerechtere, schnellere und überzeugendere empirische Vergleiche von Optimierern ermöglicht werden.

Auf der Grundlage dieses Benchmarking-Protokolls werden anschließend fünfzehn populäre Deep-Learning-Optimierer verglichen, um einen Überblick über den aktuellen Entwicklungsstand in diesem Bereich zu gewinnen. Um fundierte Entscheidungshilfen für die Auswahl einer Optimierungsmethode aus der wachsenden Liste zu erhalten, evaluiert der Benchmark sie umfassend anhand von fast 50 000 Trainingsprozessen. Unser Benchmark zeigt, dass der vergleichsweise traditionelle Adam-Optimierer eine gute, aber nicht dominierende Methode ist und dass neuere Algorithmen ihn nicht kontinuierlich übertreffen können.

Neben dem verwendeten Optimierer können auch andere Ursachen das Training neuronaler Netze erschweren, etwa ineffiziente Modellarchitekturen oder Hyperparameter. Herkömmliche Leistungsindikatoren, wie etwa die Verlustfunktion auf den Trainingsdaten oder die erreichte Genauigkeit auf einem separaten Validierungsdatensatz, können zwar zeigen, *ob* das Modell lernt oder nicht, aber nicht *warum*. Um dieses Verständnis und gleichzeitig einen Blick in die Blackbox der neuronalen Netze zu liefern, wird in dieser Arbeit Cockpit präsentiert, ein Debugging-Tool speziell für das tiefe Lernen. Es kombiniert neuartige und bewährte Observablen zu einem Echtzeit-Überwachungswerkzeug für das Training neuronaler Netze. Cockpit macht unter anderem deutlich, dass gut getunte Trainingsprozesse konsequent über das lokale Minimum hinausgehen, zumindest für wesentliche Phasen des Trainings.

Der Einsatz von sorgfältigen Benchmarking-Experimenten und maßgeschneiderten Debugging-Tools verbessert unser Verständnis des Trainings neuronaler Netze. Angesichts des Mangels an theoretischen Erkenntnissen sind diese empirischen Ergebnisse und praktischen Instrumente unerlässlich für die Unterstützung in der Praxis. Vor allem aber zeigen sie auf, dass es einen Bedarf und einen klaren Weg für grundlegend neuartigen Optimierungsmethoden gibt, um das tiefe Lernen zugänglicher, robuster und ressourcenschonender zu machen.

Acknowledgments

This thesis would not have been possible without the direct and indirect support of many people.

First and foremost, I am extremely grateful to have been advised and mentored by Prof. Dr. Philipp Hennig. You provided me with a fantastic scientific home and guided me through the challenging time of pursuing a Ph.D. Thank you for trusting in and being patient with me. I will be forever grateful to have such an inspiring role model as a Ph.D. supervisor, who combined an exemplary level of scientific knowledge with the necessary dose of humility, integrity, and well-placed humor.

I am also thankful for Prof. Dr. Hendrik Lensch and Prof. Dr. Andreas Geiger who supported my Ph.D. as supervisors and examiners. I really appreciate the time and effort spent evaluating this thesis as well as providing me with new perspectives during our regular TAC meetings.

I am grateful for having been surrounded by so many people smarter than me, I have learned a tremendous amount from you simply through osmosis. I'd like to express my deepest thanks to my former and current office mates Maren Mahsereci, Lukas Balles, Hans Kersting, Felix Dangel, Frederik Künstner, and Lukas Tatzel. It was always a joy to be around you and pester you with questions. This extends to the entire Methods of Machine Learning (MoML) group of the University of Tübingen (formerly known as Probabilistic Numerics group of the MPI for Intelligent Systems): Simon Bartels, Alexandra Gessner, Filip DeRoos, Motonobu Kanagawa, Matthias Werner, Susanne Zabel, Filip Tronarp, Agustinus Kristiadi, Jonathan Wenger, Nicholas Krämer, Thomas Gläßle, Katharina Ott, Nathanael Bosch, Julia Grosse, Marius Hobbhahn, Emilia Magnani, and Franziska Weiler. I am very fortunate to have met and worked with so many inspiring and kind people. Research is more fun if done together.

Thank you to Aaron Bahde, Robin Schmidt, and Vanessa Tsingunidis who I was fortunate to supervise through their master's thesis, research project, and bachelor's thesis. Thanks for being patient with me, while I figured out how to do this thing called supervising. Our collaboration certainly had an impact not only on this thesis but also on me personally.

I would also like to thank my friends Georg, Malte, Anna, Lukas, Annika, Alex, Lena, Stefanie, Raphael, Daniel, Stephanie, and everyone I didn't list (such as the entire theater group: Volles Rohr Theater) who supported me in countless scientific and non-scientific ways during this journey. Your advice and friendship guided me not only through my bachelor's, master's, and now my Ph.D. but also the pandemic that occurred during the second half of my Ph.D. You provided moral support and distraction whenever I needed it. None of it would have been possible without your help.

I cannot begin to express my thanks to my family for providing me with everything one could hope for. Your unconditional support and patience provided me with the curiosity and confidence to tackle a Ph.D. The fact that you still listen to me explaining what I work on is unbelievable.

Finally, I cannot thank Katrin enough for everything. Your selfless support, especially when this Ph.D. adventure was exhausting, frustrating, and tough, is simply amazing.

From the bottom of my heart: Thank you!

Frank Schneider
Tübingen, December 2021

Table of Contents

Notation

This section provides a reference for the notation used throughout the thesis. We first list placeholder symbols to indicate how we format symbols of the same type and mathematical operators. Under Specific Symbols we then list particular objects that carry a consistent meaning. We, for example, use upper case bold formatting for matrices, *i.e.* $\boldsymbol{A}$, (which is shown under Numbers and Arrays) while the matrix $\boldsymbol{I}$ specifically refers to the identity matrix (see Specific Symbols).

Numbers and Arrays

a	A **scalar** number.
$\boldsymbol{a}$	A **vector**.
$\boldsymbol{A}$	A **matrix**.

a_i	The i-th **element of vector** $\boldsymbol{a}$. We also write $[\cdot]_i$ to denote selecting the i-th component of an object.
$A_{i,j}$	**Element** i, j **of matrix** $\boldsymbol{A}$.
$\boldsymbol{A}_{i,:}$	Entire **row** i **of matrix** $\boldsymbol{A}$. The notation for column is analogously.
$\boldsymbol{a}^{(i)}$	The i-th **vector from some set**, *e.g.* the i-th example in the training data set or the parameters after i iterations. We add the parenthesis to distinguish it from exponents.

Linear Algebra

$\boldsymbol{A}^\top$	**Transpose** of the matrix $\boldsymbol{A}$.
$\boldsymbol{A}^{-1}$	**Inverse** of the matrix $\boldsymbol{A}$.
$\mathrm{Tr}(\boldsymbol{A})$	**Trace** of the matrix $\boldsymbol{A}$.
$\det(\boldsymbol{A})$	**Determinant** of the matrix $\boldsymbol{A}$.
$\mathrm{diag}(\boldsymbol{a})$	A square and **diagonal** *matrix* with the diagonal elements given by the vector $\boldsymbol{a}$.
$\mathrm{diag}(\boldsymbol{A})$	A ***vector*** whose elements are given by the diagonal elements of the matrix $\boldsymbol{A}$.
$\boldsymbol{A} \odot \boldsymbol{B}$	**Hadamard**, *i.e.* element-wise, **product** of matrix $\boldsymbol{A}$ with $\boldsymbol{B}$.
$\boldsymbol{A}^{\odot 2}$	**Element-wise square** of matrix $\boldsymbol{A}$.
$\lVert\boldsymbol{a}\rVert_p$	L^p **norm** of $\boldsymbol{a}$.

Calculus

$\mathbb{A}$	A **set**. Note, the notation for the expectation, *i.e.* $\mathbb{E}$, would be the same as for a set called "E". To avoid confusion we refrain from naming any set "E" in this thesis.
$\lvert\mathbb{A}\rvert$	**Cardinality** of set $\mathbb{A}$, *i.e.* the number elements in this set.
$\{0, 1, \dots, n\}$	A set containing all **natural numbers** until n.
$f : \mathbb{A} \longrightarrow \mathbb{B}$	A **function** f from domain $\mathbb{A}$ to codomain $\mathbb{B}$.
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A **parameterized function** of $\boldsymbol{x}$ parameterized by $\boldsymbol{\theta}$. To lighten the notation, we sometimes use $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ or omit $\boldsymbol{\theta}$ as a shorthand.
$f \circ g$	**Composition** of the functions f and g, *i.e.* $f(g(\cdot))$.
$\frac{\mathrm{d}f}{\mathrm{d}x}$	**Derivative** of f with respect to x.
$\frac{\partial f}{\partial x}$	**Partial derivative** of f with respect to x.
$\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$	**Gradient** of f with respect to $\boldsymbol{x}$.
$\frac{\partial f}{\partial \boldsymbol{x}}$ or $\boldsymbol{J}_{\boldsymbol{x}}^{f}$	**Jacobian** matrix of the form $\frac{\partial f}{\partial \boldsymbol{x}} \in \mathbb{R}^{m \times n}$ for $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$.
$\nabla_{\boldsymbol{x}}^2 f(\boldsymbol{x})$ or $\boldsymbol{H}_{\boldsymbol{x}}^{f}$	The **Hessian** matrix of f at point $\boldsymbol{x}$.
$\int f(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$	Definite **integral** over the entire domain of $\boldsymbol{x}$.
$[a, b]$	The (real) **interval** between and *including* a and b.
$(a, b]$	The (real) **half-open interval** *excluding* a but including b.

Probability Theory

$P(\boldsymbol{x})$	Probability **distribution** over the (vector-valued) random variable $\boldsymbol{x}$.
$\boldsymbol{x} \sim P$	The random variable $\boldsymbol{x}$ is **distributed according** to P.
$P(\boldsymbol{x} \vert \boldsymbol{y})$	The **conditional probability** of $\boldsymbol{x}$ given $\boldsymbol{y}$.
$\mathbb{E}_{\boldsymbol{x} \sim P}\left[f(\boldsymbol{x})\right]$	**Expectation** of $f(\boldsymbol{x})$ if $\boldsymbol{x}$ is distributed according to P. If the distribution is clear, we might drop it from the subscript and write $\mathbb{E}_{\boldsymbol{x}}\left[f(\boldsymbol{x})\right]$. If it is clear from the context which random variable the expectation is over, we omit the subscript entirely, *i.e.* $\mathbb{E}\left[f(\boldsymbol{x})\right]$.
$\mathrm{Var}\left[f(\boldsymbol{x})\right]$	**Variance** of $f(\boldsymbol{x})$.
$\mathrm{Cov}\left[f(\boldsymbol{x}), g(\boldsymbol{x})\right]$	**Covariance** of $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$.
$D_{\mathrm{KL}}(P \Vert Q)$	**Kullback-Leibler divergence** of P and Q.
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	**Gaussian distribution** over the vector-valued random variable $\boldsymbol{x}$ with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$.

Specific Symbols

$\boldsymbol{I}$	The square **identity matrix**. The dimensionality is implied by the context (or noted explicitly). The identity matrix has ones on the main diagonal elements and zeros elsewhere.
$\lambda_i(\boldsymbol{A})$	The i-th **eigenvalue** of matrix $\boldsymbol{A}$. Commonly the eigenvalues are ordered in ascending order, *i.e.* $\{\lambda_1, \dots, \lambda_{\max}\}$.
$\mathbb{R}$	The set of all **real numbers**. With $\mathbb{R}^+$ we indicate the set of all non-negative real numbers.
$\mathbb{R}^n$	The set of all n-dimensional **vectors of real numbers**.
$\mathbb{R}^{d \times n}$	The set of all **real valued matrices** with d rows and n columns.
$\mathbb{N}$	The set of all **natural numbers**, *i.e.* $\{0, 1, 2, \dots\}$. To indicate only the **positive** natural numbers, *i.e.* $\{1, 2, \dots\}$ we use $\mathbb{N}^+$.
$\mathbb{D}_{\text{train}}$	The **training set**. We drop the subscript if a distinction from the validation or test set is irrelevant or it is clear from context.
$\mathbb{D}_{\text{valid}}$	The **validation set**.
$\mathbb{D}_{\text{test}}$	The **test set**.
N	**Size of the training set**, *i.e.* $N = \lvert\mathbb{D}_{\text{train}}\rvert$.
$\mathbb{B}$	A **(mini-)batch** of samples drawn from some larger set, *e.g.* i.i.d. samples from the training data set. $\mathbb{B}^{(t)}$ refers to the mini-batch that was drawn at training step t.
$f(\boldsymbol{x}; \boldsymbol{\theta})$ or $f_{\boldsymbol{\theta}}(\boldsymbol{x})$	Machine learning **model** mapping inputs $\boldsymbol{x}$ via its parameters $\boldsymbol{\theta}$ to predictions $\hat{y}$.
$\boldsymbol{\theta}$	**Model parameters** $\boldsymbol{\theta} \in \mathbb{R}^D$.
$\boldsymbol{x}$	**Input** to the model, *i.e.* the features used for the model prediction, $\boldsymbol{x} \in \mathbb{X} \subseteq \mathbb{R}^I$.
$\hat{y}$	**Model predictions**, *e.g.* the predicted class probabilities, $\hat{y} \in \mathbb{Y} \subseteq \mathbb{R}^C$.
y	**True labels** or target vectors.
η	The **learning rate** of the iterative optimization method, *e.g.* SGD.
B	The (mini-)**batch size** used in the training algorithm.
$\sigma(x)$	Logistic **sigmoid function**, *i.e.* $\frac{1}{1+e^{-x}}$.
$\log(x)$	**Natural logarithm** (base e) of x. Logarithms with different bases, *e.g.* the **binary logarithm** with base 2, are denoted $\log_2(x)$.
$\mathcal{O}(\cdot)$	Landau **big O** notation.

Introduction 1

Deep learning has seen a significant increase in popularity in recent years, causing a steady rise in both scientific publications and public headlines. Often fueled by the ambitious goal of developing machines that can think like humans or even provide super-human intelligence, deep learning and the use of artificial neural networks has crystallized as a technique that can tackle a wide range of problems in several diverse domains. There are several areas and applications where approaches based on deep learning have led to serious performance increase, compared to traditional approaches: The generation of photo-realistic images of fictional people [*e.g.*, 61, 102, 158], automatic speech recognition software [*e.g.*, 9, 123] for digital assistants, computer programs for playing games such as Go [*e.g.*, 268], chess [*e.g.*, 269] or Atari [*e.g.*, 209, 265], machine translation systems [*e.g.*, 151, 323], or models for the prediction of protein structure [154] have all benefited from the inclusion artificial neural networks.

[61] Choi et al. (2020), "StarGAN v2: Diverse Image Synthesis for Multiple Domains"
[102] Goodfellow et al. (2014), "Generative Adversarial Nets"
[158] Karras et al. (2018), "Progressive Growing of GANs for Improved Quality, Stability, and Variation"
[9] Amodei et al. (2016), "Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin"
[123] Hinton et al. (2012), "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups"
[268] Silver et al. (2016), "Mastering the game of Go with deep neural networks and tree search"
[269] Silver et al. (2018), "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play"
[209] Mnih et al. (2015), "Human-level control through deep reinforcement learning"
[265] Schrittwieser et al. (2020), "Mastering Atari, Go, chess and shogi by planning with a learned model"
[151] Johnson et al. (2017), "Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation"
[323] Wu et al. (2016), "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation"
[154] Jumper et al. (2021), "Highly accurate protein structure prediction with AlphaFold"
[70] Deng et al. (2009), "ImageNet: A Large-Scale Hierarchical Image Database"

There are several reasons, why deep learning has emerged as one of the most promising branches of science when it comes to developing artificial intelligence (AI) specifically in the first decades of the 21st century:

- **Large data sets:** The advent of big data, *e.g.* massive data sets, facilitated by the ubiquitous use of computers and the spread of the internet, has enabled machine learning models to learn from vast amounts of data. Compared to other machine learning models, deep neural networks are particularly adept at leveraging these large data sets to infer patterns. Publicly available data sets such as the popular ImageNet [70] database for image classification and object recognition, provided a popular playground for researchers around the globe to train their models and evaluate newly developed algorithms.
- **More compute power:** Processing these vast data sets naturally also required large amounts of compute resources. Following Moore's law[1] [210], compute power increased rapidly in recent years, but it was primarily the shift to graphics processing units (GPUs) that provided the necessary compute power to apply neural networks to real-world applications. GPUs, originally developed for graphics applications such as consumer video games, allow parallel processing of many comparatively simple independent operations, such as the matrix-vector products in neural networks.

1: Moore's law is the observation that the number of transistors in an integrated circuit doubles approximately every two years. This "law" has predicted the available compute power surprisingly accurately.

- **Flexible software frameworks:** Lower-level libraries, such as CUDA, provided a relatively easy way to perform general-purpose computation on GPUs. But it was the introduction of higher-level frameworks such as Caffe [147], Chainer [296], Lasagne [73], Theano [241], TensorFlow [1], PyTorch [228], and JAX [39] that allowed the broader scientific community to use neural networks. These libraries provided automatic differentiation and abstracted lower-level operations, such as specific deep learning layers, into easy-to-use and almost plug-and-play modules. This allowed researchers and practitioners to build more complicated machine learning pipelines by combining the individual ingredients, such as model architectures, loss function, or data pre-processing similar to modular building blocks. These deep learning frameworks also simplified the sharing and incorporation of pre-trained models.
- **Algorithmic improvements:** Several algorithmic improvements have also contributed to the success of deep learning in recent years. The invention of the backpropagation algorithm (often simply called backprop) [160, 183, 248, 317], enabled the efficient computation of the gradient of a loss function with respect to the network's parameters. The development of *stochastic* optimization methods such as SGD [242] or later Adam [166] allowed these gradients to be used as learning signals for efficient training in many applications. Advances in neural network architectures, such as convolutional layers [90, 176, 310], ReLU activations [89, 98, 217], and more recent, residual networks [117], or transformer models [307] further increased the performance and expressiveness of deep learning approaches.

This renewed attention to deep learning naturally also led to increased interest in their training methods. The training or optimization methods of deep neural networks are a major part of the deep learning pipeline and crucially responsible for the learning process. Compared to the rather elaborate model architectures, these training algorithms are comparably simple.[2] Yet, these methods come with a relatively unsatisfying user experience. In practice, these methods have hyperparameters, such as the learning rate, that need to be manually set, which is usually done either by expert intuition or by expensive parameter searches which basically amounts to trial and error.

The fact, that these hyperparameters exist is most likely evidence of our lack of understanding of neural network training. In fact, optimization for deep learning might still be a very significant bottleneck that hinders the performance of modern neural networks.[3] Instead of having autonomous optimization methods that pro-

[210] Moore (1965), "Cramming more components onto integrated circuits"

[147] Jia et al. (2014), "Caffe: Convolutional Architecture for Fast Feature Embedding"

[296] Tokui et al. (2015), "Chainer: a Next-Generation Open Source Framework for Deep Learning"

[73] Dieleman et al. (2015), "Lasagne: First release."

[241] Al-Rfou et al. (2016), "Theano: A Python framework for fast computation of mathematical expressions"

[1] Abadi et al. (2015), "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems"

[228] Paszke et al. (2019), "PyTorch: An Imperative Style, High-Performance Deep Learning Library"

[39] Bradbury et al. (2018), "JAX: composable transformations of Python+NumPy programs"

[160] Kelley (1960), "Gradient Theory of Optimal Flight Paths"

[183] Linnainmaa (1970), "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors"

[248] Rumelhart et al. (1986), "Learning representations by back-propagating errors"

[317] Werbos (1982), "Applications of advances in nonlinear sensitivity analysis"

[242] Robbins et al. (1951), "A Stochastic Approximation Method"

[166] Kingma et al. (2015), "Adam: A Method for Stochastic Optimization"

[90] Fukushima (1980), "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position"

[176] LeCun et al. (1989), "Backpropagation Applied to Handwritten Zip Code Recognition"

[310] Waibel et al. (1989), "Phoneme recognition using time-delay neural networks"

vide efficient training *in a single training run*, we rely on extensive hyperparameter tuning and training heuristics.

Hundreds of methods have been proposed to make neural network training more efficient or more convenient. In this thesis, we aim to bring some structure to this crowded field. Firstly, we want to clearly identify what constitutes a better deep learning optimizer. Secondly, we empirically compare a selection of currently popular methods. Finally, we propose a novel type of debugging tool specifically designed to better understand the training process.

The goal of these efforts is to improve our understanding of neural network training through rigorous benchmarking and targeted debugging to open the black box of deep learning a little bit more. The hope is that increased understanding will naturally lead to ways of automating the training process and to more resource-efficient, robust, and accessible neural network training.

[89] Fukushima (1969), "Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements"
[98] Glorot et al. (2011), "Deep Sparse Rectifier Neural Networks"
[217] Nair et al. (2010), "Rectified Linear Units Improve Restricted Boltzmann Machines"
[117] He et al. (2016), "Deep Residual Learning for Image Recognition"
[307] Vaswani et al. (2017), "Attention Is All You Need"

2: SGD, one of the most popular optimizers for training deep neural networks, can comfortably be described in a single line, see Update Rule 3.3.2.

3: A specific example of this could be model architectures that cannot be trained using our current methods. However, these models might provide additional performance gains over the contemporary models that have co-evolved with the used optimization methods. To put it more bluntly, we might narrow the set of machine learning models to the ones our current models can train efficiently.

1.1 Detailed Outline

Part I provides a summary of the key concepts and methods necessary for the understanding of this thesis. Specific elements that are used repeatedly in this work are explained in this background part. These background chapters assume basic knowledge of *e.g.* linear algebra, multivariate calculus, and statistics.

- Chapter 2 introduces the paradigm of machine learning. Crucially it makes a clear statement that although optimization is a critical component of machine learning, there are important differences between *optimization* and *learning*.
- Chapter 3 will more formally introduce mathematical optimization, specifically focusing on *empirical risk minimization* a central principle underlying many machine learning tasks. Furthermore, the chapter includes a description of popular optimization methods for machine learning and deep learning, most of which will be empirically evaluated in this work.
- Finally, Chapter 4 concludes this part by introducing the essential concepts of deep learning such as typical layers or model architectures.

Part II represents the main contributions of this thesis and is largely based on peer-reviewed publications, see Section 1.2.

- In Chapter 5, we explore how to fairly and meaningfully compare optimization methods for deep learning. The chapter aims to understand and formalize how optimizers should be properly evaluated to test their usefulness as training algorithms for deep learning tasks. The three main challenges for devising such a standardized benchmarking protocol for deep learning are: (i) The stochasticity of deep learning, which not only requires optimization methods to be *tuned* but also necessitates multiple repetitions of training runs, since the results of a single run can be critically affected by noise. (ii) Many realistic deep learning problems can take days or even weeks to run. The test problems for a benchmark must therefore be carefully selected to cover a wide range of problems, without increasing the runtime to the point of being infeasible. The combination of the long training time and the need for tuning, also means that optimization for deep learning is a multi-objective problem: Optimization methods can be "better" either by improving the final performance, by reducing the training time needed for an acceptable result, or by requiring fewer tuning runs. (iii) The optimization methods in deep learning are used as *training* algorithms. Although they operate on a training loss, the most relevant measure for practitioners is usually a different performance metric on the test set, such as the accuracy. This crucial difference between *optimizing* and *learning* must be considered when designing a benchmarking protocol for deep learning optimizers.

- Chapter 6 leverages the benchmarking protocol devised in Chapter 5 and uses it to empirically compare fifteen popular optimization methods for deep learning. With currently more than a hundred optimization methods proposed for deep learning, see Table 3.1, there is a need for an independent third-party evaluation of deep learning optimizers. The major challenge for such an empirical comparison is to identify and reduce the possible dimensions that could be included in such a benchmark. It is simply impossible to test all methods, especially when considering different batch sizes, learning rate schedules, tuning methods, and search spaces. The most important task for this benchmark was to make a sensible selection of each individual dimension in order to still achieve a meaningful comparison.
- In Chapter 7, we propose a new type of debugger specifically designed for training neural networks. We take the idea of a cockpit, as a useful aggregation of meaningful and well-designed instruments, meant to aid a pilot in the complicated process of flying a plane and apply it to the perhaps comparably complicated process of training a deep neural network. Here, the instruments of our Cockpit are novel observables based on higher-order information about the gradient distribution. By complementing the usual examination of learning

curves, *e.g.* monitoring the train and test loss, with these novel instruments, we can obtain a more meaningful status report for practitioners. Identifying relevant observables, their efficient computation to enable real-time monitoring, and showcasing their capabilities are central aspects of the work presented in this chapter.

Part III summarizes the findings and conclusions from this thesis and provides ideas for future work in this area. Some of the future work mentioned in Section 8.2 represent current work in progress, such as the benchmark outlined in Section 8.2.1, while others describe possible natural extensions of the work described in this thesis.

The appendix contains additional results, experiments, or code examples of Chapters 5 to 7.

1.2 Publications

Parts of the contents of this thesis are based on publications done in close collaboration with colleagues. Listed below are the peer-reviewed publications that this thesis builds upon together with a listing of the individual co-author contributions. All involved co-authors agreed to this listing.

Chapter 5 is based on the following peer-reviewed conference publication:

DeepOBS: A Deep Learning Optimizer Benchmark Suite

Frank Schneider, Lukas Balles, and Philipp Hennig. "DeepOBS: A Deep Learning Optimizer Benchmark Suite". *7th International Conference on Learning Representations, ICLR*. 2019. [262]

The co-author contributions were as follows:

	Ideas	**Experiments**	**Analysis**	**Writing**
F. Schneider	50 %	75 %	80 %	60 %
L. Balles	30 %	15 %	10 %	15 %
P. Hennig	20 %	10 %	10 %	25 %

The following peer-reviewed conference publication forms the basis for Chapter 6:

Descending through a Crowded Valley — Benchmarking Deep Learning Optimizers

Robin M. Schmidt, Frank Schneider, and Philipp Hennig. "Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers". *38th International Conference on Machine Learning, ICML*. 2021. [261]

The co-author contributions were as follows:

	Ideas	**Experiments**	**Analysis**	**Writing**
R. Schmidt	15 %	65 %	15 %	30 %
F. Schneider	70 %	30 %	70 %	55 %
P. Hennig	15 %	5 %	15 %	15 %

Chapter 7 draws on the following peer-reviewed conference paper:

Cockpit: A Practical Debugging Tool for the Training of Deep Neural Networks

Frank Schneider, Felix Dangel, and Philipp Hennig. "Cockpit: A Practical Debugging Tool for the Training of Deep Neural Networks". *Advances in Neural Information Processing Systems 34, NeurIPS*. 2021. [263]

The co-author contributions were as follows:

	Ideas	**Experiments**	**Analysis**	**Writing**
F. Schneider	45 %	40 %	40 %	45 %
F. Dangel	40 %	50 %	40 %	40 %
P. Hennig	15 %	10 %	20 %	15 %

Part I

Preliminaries & Background

Machine Learning 2

In machine learning, a subfield of artificial intelligence (Figure 2.1), knowledge is not explicitly programmed or hard-coded into the software by the code's author. Instead, the strategies needed to perform a task are **learned** from data. A traditional translation system, for example, might use hard-coded word-to-word translation tables that were created by domain experts. In contrast, translation systems based on machine learning would instead leverage continuous texts that are available in multiple languages, such as documents from the European Parliament,[1] to learn a function that is able to transform one language into another.

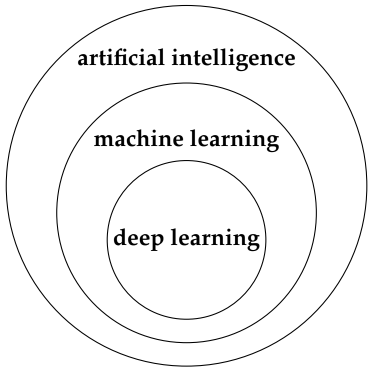


Figure 2.1: Relationship between artificial intelligence, machine learning, and deep learning. Artificial intelligence describes any technique which enables computers to demonstrate intelligent behavior, similar to humans or other animals. Machine learning is a subfield of artificial intelligence that gives machines the ability to perform tasks without being explicitly programmed for them. Rule-based systems are an example of artificial intelligence without machine learning. Deep learning is a subfield of machine learning that uses neural networks as machine learning models. Using support vector machines, decision trees or Gaussian processes are examples of machine learning systems without deep learning.

1: A part of the popular WMT 2016 data set [31] for machine translation consist of the multilingual parallel corpus of the proceedings of the European Parliament.

This seemingly small shift of replacing hard-coded strategies with learned ones has tremendous consequences. It allows computers to solve tasks that are hard to formalize as rules but can instead be solved with relatively little difficulty by pattern matching. Describing what characterizes a handwritten "two", for example, is quite complicated. It seems impossible to define fixed rules describing what pixel values an image of a handwritten "two" must possess. Especially, when considering the wide range of different handwritings or awkward angles of the photograph (see Figure 2.2 for examples of handwritten digits from the MNIST data set [177]). However, recognizing a handwritten digit is an easy task for a human. Just like a human, who developed this skill by seeing many examples of handwriting, artificial systems can learn to classify handwritten digits by learning from a large corpus of labeled examples.

Additionally, the same *learning approach* can be applied to domains where even domain experts have trouble seeing meaningful patterns. An example of this is protein folding. The three-dimensional structure of a protein is crucial to understanding its biological function. Determining a protein's structure is currently mostly determined experimentally, a costly and tedious process. Machine learning approaches, such as AlphaFold [154], use data to train a model that predicts the three-dimensional protein structure from its amino acid sequence. Estimating the structure of a single protein can thereby be achieved in days compared to the previous experimental approaches that can take years.

In comparison with more traditional software, the quality of a machine learning system crucially depends not only on the underlying algorithms, but the data it was trained on. For any real-world application there is only a limited amount of training

[31] Bojar et al. (2016), "Findings of the 2016 Conference on Machine Translation"

[177] LeCun et al. (1998), "Gradient-Based Learning Applied to Document Recognition"

[154] Jumper et al. (2021), "Highly accurate protein structure prediction with AlphaFold"

data available. Yet, the learned system should ideally generalize to new and unseen data as well. The fact that a machine learning system is trained on a specific data set but expected to perform well for general data, is what separates *learning* from pure *optimization* (see Section 2.2). This requires new methods to ensure that the learned system is useful even when applied outside of its training data. We discuss those methods in Section 2.3 and summarize them under the term *regularization*.

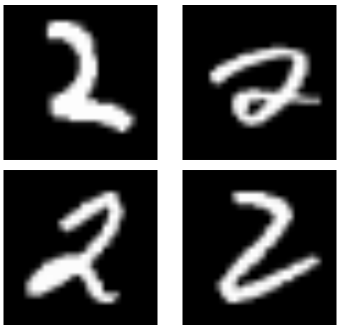


Figure 2.2: **Handwritten versions of the digit "two"** can come in different shapes and forms. Encoding rules that would categorize a handwritten digit with its correct digit is a challenging problem. One of the earliest success stories of deep learning came from learning a classifier from labeled data instead [177].

Machine learning algorithms are often categorized according to the type of task they aim to solve and the characteristics of the available data or feedback. The most common machine learning tasks include the following:

- **Supervised learning:** The data set in supervised learning consists of (input) features and associated (output) labels or targets. The goal of the machine learning system is to learn a function that can predict the correct target, given an input. For example, an image classification data set such as MNIST contains images of handwritten digits with their corresponding labels. Here, the label is the digit that the handwritten line is representing, *i.e.* all images in Figure 2.2 would have the label `"2"`. A supervised learning algorithm can then be trained to predict the correct label given an input image and thus classify handwritten digits, for example, zip codes on letters for post offices. We will have a more detailed look into supervised learning and its ingredients in Section 2.1.
- **Unsupervised learning:** In unsupervised learning, an algorithm is not provided with any labels. Instead, the goal of the machine learning model is to learn useful properties and occurring patterns of the given data. Given unlabeled data, *e.g.* the metadata of a song or its properties such as the average volume or tempo, unsupervised methods can cluster them into classes of similar examples to detect music genres or suggest similar music.[2] A major advantage of unsupervised learning is that unlabeled data is usually much easier to collect than labeled data, which often needs to be annotated by human experts. Naturally, the boundaries between supervised and unsupervised learning are fuzzy. **Semi-supervised learning**, for example, uses a large amount of unlabeled data with a small amount of labeled data.

2: In a *supervised learning* approach to the same task, each song would be classified with a label describing the genre it belongs to.

- **Reinforcement learning:** In reinforcement learning, an autonomous agent must learn to perform a task and is provided with feedback in terms of a reward or a penalty. Instead of having labeled data to learn from, it learns by experience gained from trial and error. Reinforcement learning can, for example, be used to create systems that can successfully play Atari video games. Here, the machine learning model receives the pixels of the game as an input, decides on an action, and uses the game's score as a reward to learn a successful strategy [*e.g.*, 209].

[209] Mnih et al. (2015), "Human-level control through deep reinforcement learning"

In summary, machine learning can be described informally as using **data** or **experience** to learn a **model** to perform a **task** as measured by a **performance metric**. In the following sections, we will describe and provide intuitive descriptions and examples for these four important building blocks of any machine learning system. For this, we will be using the concrete example of *supervised machine learning*, which will also be the primary focus of this work.

2.1 Supervised Learning

The characteristic property of supervised learning is that we are given both inputs x and corresponding labels y. Figure 2.3 illustrates input-output-pairs for some exemplary supervised learning tasks, such as image classification, where the inputs x are images and the associated labels y indicate the type of object in the image. The goal is to learn a function f_θ that accurately models the relationship between those inputs x and labels y. Given unseen data, *e.g.* new images, the hope is that this learned function f_θ can then accurately map them to the correct labels as well, *e.g.* classify those unfamiliar images.

In the following sections, we will take a look at the specific tasks that can be solved by supervised learning (Section 2.1.1). Every machine learning task requires a (family) of models (Section 2.1.2) that is trained on data (Section 2.1.3). To describe how well the model is currently describing the relationship between the inputs and the labels, we use a loss function or a performance metric (Section 2.1.4). Although we will focus on intuitions and examples from supervised learning, many of the properties and described challenges extend to general machine learning applications. In this chapter, we will mainly focus on the practical perspective and re-visit many aspects in Chapter 3 from the more theoretical point-of-view of optimization.

2.1.1 Tasks

The **task** that the machine learning model learns, is critically determined by the content of the input-output pairs. Given the same inputs, *e.g.* text paragraphs, the model could either learn to infer its author (which would be categorized as a *text classification* task), sum up its content (called *text summarization*), or describe the polarity

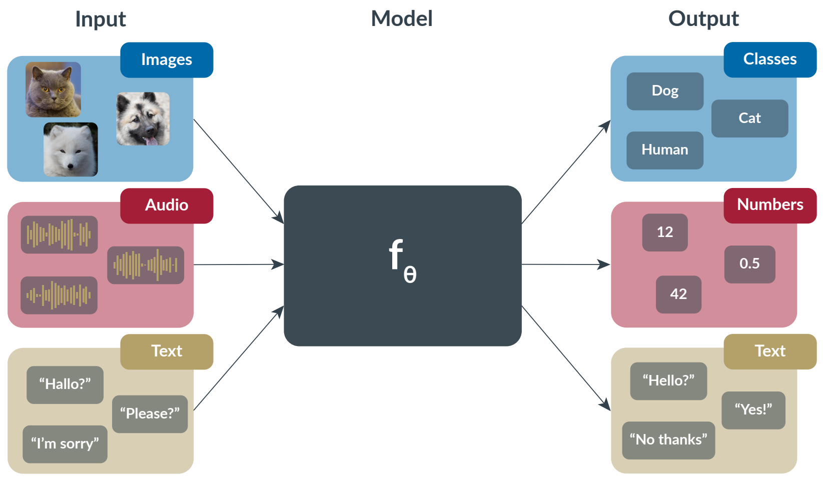


Figure 2.3: Illustration of supervised learning. The task that the model learns is largely influenced by the input and output data. The same input data, *e.g.*, images of animals, can lead to different tasks depending on the output labels. Classes such as "dog" or "cat" enables the model to classify the animal present in the picture. Text outputs, on the other hand, could be used to automatically caption the images. Similarly, the nature of the task also depends on the input. Using audio snippets of spoken words as inputs along with a string representation of the spoken words would result in an automatic speech recognition task. Machine translation would instead require the translated text as output labels.

and emotions of the text (known as *sentiment analysis*). Which of these it will learn depends on the type of labels provided.

Supervised learning tasks are often categorized according to the type of the output labels:

Binary classification refers to tasks, where the correct outputs can only be one of two choices, *i.e.* $y \in \{0, 1\}$. They are usually labeled by *one* and *zero* to signal the membership to each group or the presence or absence of a property. Typical examples of binary classification problems include:

- **Quality control:** A machine learning model could be used to automatically detect, whether a manufactured item meets the specified quality standards. Inputs could for example be images of the product and the outputs indicate whether it meets the specifications ($y = 1$) or not ($y = 0$).
- **Anti-spam filters:** To detect unwanted emails, data of both regular mails ($y = 0$) and spam mails ($y = 1$) can be collected to learn a model. The inputs could contain metadata of the email, such as the sender, the sending time, the inclusion of URLs, or even the entire content of the email. Based on these inputs, a machine learning model could learn to automatically detect unwanted or nefarious messages.
- **Image classification:** Binary classification can, for example, be used to flag images that contain offensive contents. Here, the model produces a prediction of whether an image should be flagged ($y = 1$) based on the pixel values of said image.

Multiclass classification extends the idea of binary classification to three or more classes, *i.e.* $y \in \{1, \ldots, C\}$. For computational convenience, the labels are usually represented by a vector-valued label $\boldsymbol{y}$ that has a 1 at the correct class and is zero otherwise. For example, the label vector $\boldsymbol{y} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \end{pmatrix}^\top$ would indicate the correct label to be class 2, which could *e.g.* stand for `"apples"` in the context of image classification. This *one-hot* encoding circumvents the problem that an integer encoding, *i.e.* $y \in \{1, \dots, C\}$ implies a natural ordered relationship between the classes, which may not exist for categorical data, such as possible objects in image classification. Multiclass classification includes the following quintessential examples:

- **Image classification:** Multiclass image classification could categorize images by their content. Possible classes given by the labels would be `"cats"`, `"dogs"`, or `"human"`.
- **Speech recognition:** Given audio clips of spoken words or sentences, the goal of *Automatic Speech Recognition* (ASR) is to infer the text transcript of what was spoken.
- **Video classification:** Machine learning models could be used to classify movies by their genres. The input data could contain metadata such as the titles, directors, the running time, or even (a subset) of the movie's frame. The labels would then give the most fitting genre, such as `"film noir"`, `"adventure"`, or `"fantasy"`.

Regression tasks predict continuous variables, *i.e.* $y \in \mathbb{R}$. The specific application often restricts the output domain further, *e.g.* restricting the predictions to lie within the positive real numbers. Examples for regression tasks are:

- **Predicting house prices:** Based on features describing the house, such as the number of rooms or its size, one might predict the price of the building.
- **Weather forecasting:** The weather of the next few days can be predicted based on atmospheric features such as humidity, pressure, or past temperatures.
- **Energy consumption:** Predicting future energy consumption can help manage the distribution of electricity in the power grid. Based on the date, time, and other factors such as the weather, one could predict the expected energy consumption.

2.1.2 Model

The family of machine learning models that is tasked with learning a relationship between the given inputs and labels is usually parameterized. This allows an optimization method (see Chapter 3) to efficiently search over possible mappings and find the one that provides the best matching. Defining this family of models is a crucial step, as it defines the space of possible hypotheses. Only if the hypothesis space includes a function that is capable of, at least approximately, describing the relationship between inputs and outputs, the machine learning system has the chance to learn the task satisfactorily.

A variety of machine learning models have been proposed, all with individual strengths and weaknesses. Among them are support vector machines [34, 64], naive Bayes classifiers, decision trees [41], or neural networks [118, 204]. In this work, we will focus on artificial neural networks (see Chapter 4), as their structure implies interesting properties for optimization and deep neural nets were able to achieve astounding performance in multiple areas in recent years.

[34] Boser et al. (1992), "A training algorithm for optimal margin classifiers"
[64] Cortes et al. (1995), "Support-vector networks"
[41] Breiman et al. (1984), "Classification and Regression Trees"
[118] Hebb (1949), "Organization of Behavior"
[204] McCulloch et al. (1943), "A logical calculus of the ideas immanent in nervous activity"

Naturally, there is no single machine learning model that works best on all types of supervised learning tasks, a notion that is also described by the *no free lunch theorem* [319].[3] Instead, the tasks, the available training data, or the dimensionality of the input space, among other things, have to be carefully considered when selecting a model. A large model capacity, for example, allows to describe complicated relationships but might be prone to overfitting (see Section 2.2). Neural networks, for example, usually provide superior performance in the regime of large training data sets.

[319] Wolpert et al. (1997), "No free lunch theorems for optimization"

3: Colloquially, the *no free lunch theorem* states that (under certain constraints) every optimization technique will perform as well as every other method if averaged over *all possible problems*. Due to the close relationship between optimization and machine learning, this result also extends to machine learning. Roughly summarized, it means that there is no single universally best machine learning model [*e.g.*, 101, 213].

[101] Goodfellow et al. (2016), "Deep Learning"
[213] Murphy (2012), "Machine Learning A Probabilistic Perspective"

2.1.3 Data

Central to the learning process is the data. In supervised learning, we are given a set of input-output pairs that can be used to train the machine learning model. Ultimately, however, we are not interested in the model's performance on this known training data set, but on new, previously unseen data. To this end, it is common to split the available data into a *training data set*, a *validation data set*, and a *test set*.

The **training data** is used to fit the parameters of the model, *e.g.* the weights and biases of a neural network classifier. What separates pure optimization from machine learning, however, is that we are not interested in finding the solution that best describes the *training* data, but *general* data of the same type. The benefit from a machine learning model does not come from the ability to classify exactly the images in a given data set but instead *learn* what constitutes an image of a cat, for example. Even during the process of model training, which mostly consists of finding the parameters of a function f_θ that fits the given data well, we have to consider this goal of generalization (see Section 2.2 for a more detailed view on the differences between optimization and machine learning and

Section 2.3 for a discussion of techniques to achieve generalization capabilities during training).

To check whether our learned model works on unseen data and therefore generalizes, we can investigate the model's performance on the **test set**. This independent set of data, from the same distribution as the *training* set, represents novel data to the model. As long as the test set has truly only been used to assess the performance of the learned model, it provides an unbiased evaluation of the final model and a prediction for how well the model is able to infer labels on novel data.

One might be tempted to use the model's performance on this *test* set to select from multiple models, *e.g.* trained models of polynomials of varying degrees. However, this would invalidate the *test* set's ability to predict the performance on unseen data, as the selection process itself can be seen as part of the training. Instead, model selection or hyperparameter tuning, such as tuning the number of layers in a neural net, or the learning rate of the optimization method, are done on yet another separate data set, the **validation data set**.[4]

4: The *validation set* is sometimes also called *development set* or simply *dev set*.

2.1.4 Loss Functions and Performance Metrics

In the previous sections, we have used the somewhat vague notion that the machine learning model should be able to "fit" the data accurately. A **loss function**[5] formalizes what a "good fit" means by condensing the behavior of the machine learning model into a single scalar number. It penalizes "bad" or unwanted behavior of the model and rewards good predictions. In other words, it quantitatively measures how well the model f_θ can predict the data. This reduction into a single scalar allows to compare and rank different hypothesis models directly.

5: Depending on the context, the loss function is also called the *objective function*, *criterion*, *cost function*, or *error function*.

The choice of loss function is generally task-specific. For classification, for example, the **accuracy** of a model is a natural measure for the quality of the model. The accuracy of a model is described by the proportion of correctly classified examples.[6] However, not all machine learning scenarios have a straightforward and objective choice of loss function. Should a system be penalized more for frequent, but smaller-sized mistakes, or if it makes grave mistakes but only rarely?

6: Another often-used performance measure is the **error rate**, which is simply defined as $(1 - \mathit{accuracy})$ and describes the proportion of examples classified incorrectly. This error rate is also called the *0-1 loss*.

These design choices depend on the specific application and often require human judgment. In medical domains, larger mistakes could mean severe health consequences. In contrast, a manufacturer that uses machine learning in its intermediate process might be more satisfied having only a few but grave mistakes that can be easily sorted out in quality control. A loss function measures whether or not a model performs "better", but what is "better" might not always be clear or at least dependent on the application.

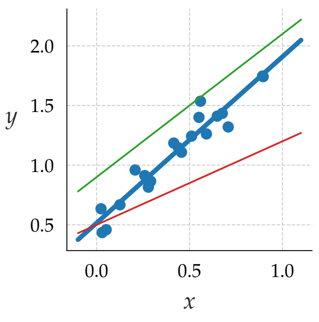
In other cases, it is conceptually clear what "better" means, but it is impractical to compute. GENERATIVE ADVERSARIAL NETWORKS (GANs) [102], for example, have been used to create photorealistic images of non-existing people [*e.g.*, 61, 158, 159]. Here, it is clear that "better" models produce images that look more realistically, *i.e.* that are indistinguishable from real images when judged by humans. This, however, is not computable and therefore impractical to use as a loss function. Instead, computable approximations such as the FRÉCHET INCEPTION DISTANCE (FID) are used.

[102] Goodfellow et al. (2014), "Generative Adversarial Nets"
[61] Choi et al. (2020), "StarGAN v2: Diverse Image Synthesis for Multiple Domains"
[158] Karras et al. (2018), "Progressive Growing of GANs for Improved Quality, Stability, and Variation"
[159] Karras et al. (2020), "Analyzing and Improving the Image Quality of StyleGAN"

Even if a measure is computable, it might not be feasible as a loss function for training. Oftentimes the loss function needed for efficient training carries restrictions, such as the requirement for it to be differentiable. If the gradient of the loss with respect to the model's parameter is easily computable, gradient-based optimization methods can be used to train large machine learning models efficiently (see Section 3.2.1). As a result, the loss function used for training is often only a surrogate function and an approximation of the quality measure we actually care about. However, non-differentiable measures such as the accuracy or the FID can still be used as **performance metrics**. These performance metrics can be used to measure the performance of the model for model comparison, hyperparameter tuning on a validation set, or quality control. The fact that an optimization algorithm works uses a loss function such as the cross-entropy loss, while the actual performance metric of interest is something else, *e.g.* the classification accuracy, is a crucial difference between machine learning and pure optimization which will be discussed further in the next section, Section 2.2.

In Section 4.2.3, we will take a closer look at two specific and often-used loss functions in deep learning, the cross-entropy loss for classification problems and the mean squared error loss for regression tasks.

(a) Regression functions



(b) Loss landscape

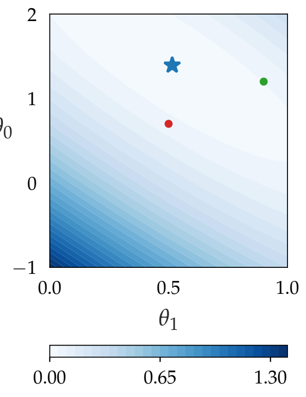
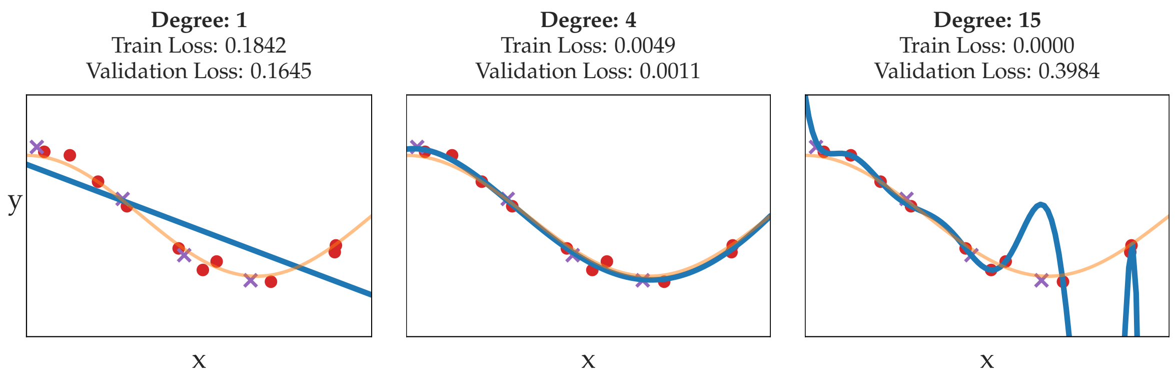


Figure 2.4: Function and weight view of a regression problem. (a) shows the regression data (●), the optimal fit (—), and two other possible hypothesis (—, —). (b) Each hypothesis (● or ●) is defined by its parameters θ_0 and θ_1 and has an associated loss value (shown by the background shading). The totality of all possible hypotheses and their loss values forms the *loss landscape*, which in this case forms a quadratic bowl or valley shaped landscape. Its minimum (★) provides the optimal fit to the data.

2.2 Learning Is More than Optimization

Learning and *optimization* are closely linked. Finding the specific parameters of a parameterized machine learning model that fit the given training data the best, can essentially be solved by optimization. Each specific set of parameters can be associated with a specific modeling hypothesis and given some training data, also be associated with a training loss describing how well this hypothesis describes the data. We can now take a look at all



(a) Underfitting **(b)** Appropriate fit **(c)** Overfitting

Figure 2.5: Under- and overfitting models. Given noisy training samples (●) from some ground truth function (—), we can fit different polynomials of varying degrees (—). We can see, that (a) a linear function is not expressive enough to accurately describe the data and suffers from *underfitting*. (c) Using a polynomial of degree 15 on the other hand can fit the data virtually perfectly but *overfits* to the training data. While it accurately describes the training data points, it fails to learn the structure of the ground truth solution. Extra data points, that were excluded from training but used as *validation data* (✖) can identify the inability of both polynomials to generalize. (b) Using a polynomial of degree 4 describes not only the training data but the validation data as well and thus provides an appropriate fit.

possible hypotheses, *i.e.* all possible parameter combinations, and their associated loss values. While illustrating this **loss landscape** is simple to do for a model with two parameters, see Figure 2.4, it is infeasible for practical machine learning models with often millions of parameters. Nevertheless, finding the best parameter set always amounts to searching for the global minimum of the loss landscape.

2.2.1 Generalization to Unseen Data

However, the central challenge of machine learning is to find models and algorithms that perform well on *new* and *previously unseen* data. The ability of a machine learning model to perform well on unobserved inputs of the same type is called **generalization** and constitutes perhaps the central problem of machine learning. Figure 2.5 shows a fitting of three polynomials of different degrees. The linear polynomial shown in Figure 2.5a is not expressive enough to describe the given data and even its optimal set of parameters results in a comparable large loss on the training data. This phenomenon, known as **underfitting**, is often the result of small **model capacity**[7] and can be identified by the model being unable to obtain a low training loss.

7: A colloquial definition of *model capacity* is given by the model's ability to describe a wide variety of functions. A polynomial of degree n, for example, will have a higher model capacity than a polynomial of degree $n - 1$, since it is able to fit more functions accurately. More technically rigorous formulations of model capacity, such as VC DIMENSION [306], exist, but have so far proven difficult to extend meaningfully to the deep learning setting [*e.g.*, 343].

[306] Vapnik et al. (1971), "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities"

[343] Zhang et al. (2017), "Understanding deep learning requires rethinking generalization"

In comparison, both a polynomial of degree 4 (shown in Figure 2.5b) and degree 15 (shown in Figure 2.5c) are able to correctly fit the training data points. However, only the former really *learned* the correct pattern of the ground truth. The polynomial of higher degree, on the other side, **overfits** to the training data. While it can accurately describe the training data, both its interpolation and extrapolation behavior for unseen data from the ground truth is poor. Models with a large capacity can overfit since they can essentially *memorize* the training set instead of learning the important patterns and structures of it.[8]

8: Zhang et al. [343] showed that state-of-the-art convolutional neural networks can memorize entire training sets of contemporary sizes. The intriguing and so far unsolved question of neural networks is why common strategies lead to the network *learning* the relevant patterns of the data, even if it could *memorize* the data instead.

[343] Zhang et al. (2017), "Understanding deep learning requires rethinking generalization"

We can measure the generalization performance by measuring the performance of the model on a *test set* that the model previously had no access to.[9] The difference between the performance on the *training data* and the *test data* is called the **generalization gap** or the **generalization error**. *Overfitting* is present in models that have a low training loss but a large test loss. In the example shown in Figure 2.5, we can see that the polynomial of higher degree has a significantly larger loss on this unseen data.

9: This can, for example, be done by reserving a part of the available data solely for testing purposes. This *heldout* data must not be used for training, but only for checking the model's ability to generalize, see Section 2.1.3.

The focus on the generalization performance is a crucial difference that separates machine learning from optimization. In our example, both the polynomial of degree 4 and degree 15 were able to fit the given data. From a pure optimization perspective, both models performed approximately the same, and there is little reason to prefer one over the other. Taking a more practical example, we can compare a model that simply *memorized* the training data and a second model that truly *learned*, like a human, what constitutes an image of a cat. From the point of view of pure optimization, both models have the same performance (on the training set, which is the object the optimization algorithm interacts with). But both models are certainly not equally useful as a machine learning model.[10]

10: The idea of selecting between multiple models that approximately fit the data is often guided by the principle of **Occam's razor**. Loosely speaking, the principle states that when faced with multiple hypotheses resulting in the same prediction quality, one should prefer the one with the fewest assumptions.

As we will see in the next section, multiple techniques can help to prevent models from overfitting and even the utilized optimization method itself can influence how well a learned solution is able to generalize (see Section 2.3.3). This difference between *learning* and *optimizing* is a crucial characteristic that we have to take into account, for example, when comparing different optimization methods for their use in deep learning problems (Chapters 5 and 6).

2.2.2 The Bias-Variance Tradeoff

To better understand the occurrence of under- and overfitting, we can study the relationship between the model capacity and its training and test performance. Figure 2.6 illustrates a typical relationship, where training and test performance behave differently when increasing the model's capacity. The characteristic U-shape is a result of the **bias-variance tradeoff** and can be understood by decomposing the error of the machine learning model.

Let us assume a ground-truth function with noise $y = f(x) + \varepsilon$, where the noise ε has zero mean and variance σ^2. Our goal is to learn a function $\hat{f}_\theta$ that approximates the true function f as best as possible as measured by the squared error $(y - \hat{f}_\theta)^2$. We are also given a training data set $\mathbb{D}_{\text{train}} = \{(x^{(1)}, y^{(1)}), \dots, (x^{(N)}, y^{(N)})\}$ which are assumed to be i.i.d. samples of some true data distribution P_{true}. We can decompose the expected error as:

$$\begin{aligned}\mathbb{E}_{\mathbb{D}_{\text{train}} \sim P_{\text{true}}}\left[\left(y - \hat{f}_\theta\right)^2\right] &= \underbrace{\left(\mathbb{E}\left[\hat{f}_\theta\right] - f\right)^2}_{\text{Bias}^2} \\ &+ \underbrace{\mathbb{E}\left[\left(\mathbb{E}\left[\hat{f}_\theta\right] - \hat{f}_\theta\right)^2\right]}_{\text{Variance}} \\ &+ \underbrace{\mathbb{E}\left[\varepsilon^2\right]}_{\text{Irreducible error}}, \end{aligned} \tag{2.1}$$

where all expectations are taken over different choices of the training set $\mathbb{D}_{\text{train}}$. The occurring three terms can be described as:

- **Bias:** The bias measures the difference between the expected prediction and the true value. Models with high bias incorrectly describe the data, *e.g.* when approximating a non-linear function using linear models. The bias describes errors caused by incorrect assumptions of the machine learning model, missing the relevant relationship between the inputs and outputs and thus underfitting.
- **Variance:** The variance describes how much the learned model fluctuates when using different training data points. This error can be seen as a result of modeling the noise rather than the underlying structure and thus overfitting to the noise
- **Irreducible error:** Since the ground-truth function contained noise ε, we cannot expect to model the function perfectly. Instead, we have to accept an irreducible error that is given by the variance σ^2 of the noise. Intuitively, this means that the larger the noise level of the data, the larger the expected error on an unseen example.

Figure 2.8 illustrates the different terms of the expected error, by fitting polynomials of varying degrees to different subsets of the training data. Fitting a polynomial of degree 2 (shown in Figure 2.8b) is not expressive enough to describe the ground-truth function. The quadratic function fails to approximate the function accurately, resulting in a high bias, but the variance between different fits is relatively small. Using a polynomial of an appropriate degree (Figure 2.8c shows degree 6, the same as the ground-truth function) approximates the function well. Both the bias and the variance are comparatively small. Increasing the degree further (*e.g.* to degree 10, illustrated in Figure 2.8d)

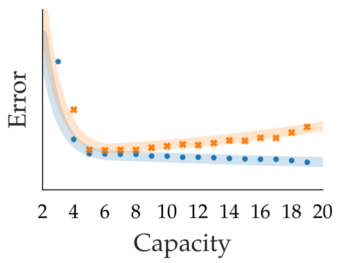


Figure 2.6: Illustration of a typical relationship between the capacity and the errors. The *training* error (▮) continuously decreases with larger model capacity. The *test* error initially (▮) initially decreases as well, but at some point, it increases again. The shaded areas show a typical progression, while the colored dots and crosses show specific training (●) and test errors (✖) when fitting a polynomial of the given degree (a measure of the capacity).

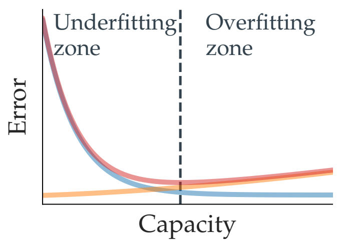


Figure 2.7: Illustration of the bias-variance tradeoff. As the model capacity increases, bias (▮) tends to decrease. Simultaneously, variance (▮) increases with increased capacity. The resulting generalization error (▮), which is the sum of both effects (plus an irreducible error term) forms a characteristic U-shape, which is also visible in Figure 2.6.

also increases the variance. The predicted values fluctuate widely depending on which data points were used (most notably in this example for small x-values). We can see that models with higher capacity, *i.e.* higher polynomial degree, show smaller bias, but higher variance. Models with smaller capacity, *i.e.* polynomials of smaller degree, exhibit the opposite, *i.e.* higher bias, but lower variance. This bias-variance tradeoff is summarized in Figure 2.7 as a function of the model capacity. [11]

11: This "traditional" bias-variance tradeoff seems to be at odds with the observed performance of contemporary large-scale machine learning models such as deep neural networks with millions of parameters. Belkin et al. [23] augment the classical view of the bias-variance tradeoff with the *double descent* phenomenon. In this *double descent* model, increasing the model capacity further, beyond a point they coin the *interpolation threshold*, reduces the test error once again. This *double descent* phenomenon can be seen as an extension of the bias-variance tradeoff and has been observed in a variety of deep learning and other machine learning models [*e.g.*, 218].

[23] Belkin et al. (2019), "Reconciling modern machine-learning practice and the classical bias–variance tradeoff"

[218] Nakkiran et al. (2020), "Deep Double Descent: Where Bigger Models and More Data Hurt"

2.2.3 Generalization to Performance Metrics

Beyond the generalization to previously unseen data, machine learning models often have to deal with the generalization to a different performance metric. As mentioned in Section 2.1.4, the types of functions that can be used as loss functions for efficient optimization methods can be limited. Deep learning, for example, requires the loss functions to be differentiable, so that gradients can be computed efficiently which in term allows first-order optimization methods to find good solutions quickly. The performance metric of interest, however, does not necessarily follow all these restrictions. A prominent example is the *accuracy* used for image classification. It describes the number of correctly classified images and is thus a natural measurement of model quality. This *0-1 loss*, however, is not differentiable, and many machine learning models for image classifications are trained with cross-entropy instead (see Section 2.1.4).

This describes another example of how learning differs from optimization. Although the model is trained to minimize the *training loss*, we care about the *test accuracy*. In other words, although we perform *optimization*, we care about the model's ability to *learn*. In Chapter 5, we encounter an example of this non-trivial relationship between the model's performance on the loss and its performance in terms of the accuracy. Looking at the column of P3 Fashion-MNIST CNN in Figure 5.2 we can observe that although the training loss consistently decreases over the course of the training, the test loss quickly increases again. Usually, this is seen as an indicator of overfitting and training should be stopped before the test loss increases again. However, when looking at the test accuracy, we can see that it continues to increase as well, although the test loss would indicate a worse performance. This phenomenon is discussed in more detail in Section 5.4 and shows that generalizes does not only involve the step to unseen data (from *train* to *test*) but also to the relevant performance metric (from *cross-entropy loss* to *accuracy*).

2.3 Regularization

Regularization methods aim to address the challenges mentioned in the previous sections. A common theme for regularization approaches is that they express a preference for specific solutions, either explicitly or implicitly. These methods all share the goal of preferably selecting models that are particularly good at generalizing to unseen data.

A simple approach to address overfitting is to control the model capacity. This can, for example, be done by *only* including "simple" polynomials such as linear or quadratic functions in the hypothesis space of possible models. Similar to other regularization methods, this can be seen as expressing an infinitely strong preference for linear and quadratic models.

Just like there is no universally "best" machine learning algorithm, there is no "best" regularization method. Instead, regularization often requires some domain expertise and understanding of which type of solutions should be preferred for a specific task. Some strategies constrain the model (Section 2.3.1), some change the training data (Section 2.3.2), or involve the optimization process (Section 2.3.3). We will now look at often-used regularization methods focusing on ones that are popular for deep learning. A more detailed overview of regularization strategies can be found in Goodfellow et al. [101].

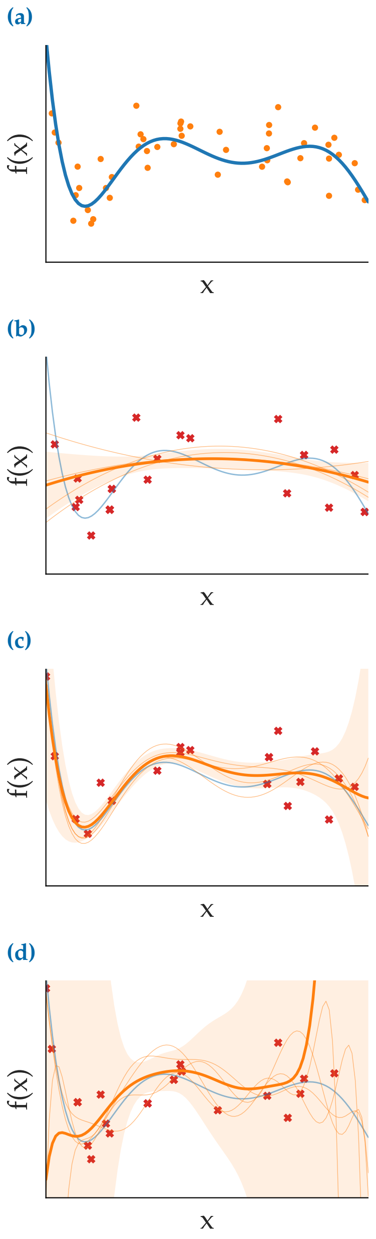


Figure 2.8: Illustration of bias and variance at different model capacities. (a) shows the ground-truth function (—) and the available noisy data points (●). Subfigures (b) to (d) use models of increasing capacity (degrees 2, 6, 10) fitted on random subsets of the data (one random subset is shown with red crosses (✖)). For each degree, we show three fits based on different random subsets (—) and the mean of 1000 fits with a thicker line (—). One standard deviation is shown as a shaded area (▮).

[101] Goodfellow et al. (2016), "Deep Learning"

[101] Goodfellow et al. (2016), "Deep Learning"

2.3.1 Norm Penalty

A straightforward way to express a preference for specific solutions is to penalize unwanted solutions, *e.g.* ones with large parameter norms. Adding a norm penalty term to the loss that is proportional to the norm of the parameters enforces a soft constrain on the model itself:

$$\tilde{L}(f_{\boldsymbol{\theta}}) = L(f_{\boldsymbol{\theta}}) + \lambda \|\boldsymbol{\theta}\| \,, \qquad (2.2)$$

where L is the loss function, $\tilde{L}$ describes the regularized loss function and $\lambda \in [0, \infty)$ is the regularization strength. λ is a hyperparameter that determines the amount of regularization (see Figure 2.9) and setting it to zero is equivalent to no regularization. The regularization hyperparameter is usually tuned for a specific problem with regularization typically only being applied to the *weights* of a neural network layer, but not its *biases* [101].

Looking at Equation (2.2), we can see that optimizing $\tilde{L}$ simultaneously minimizes L and $\|\boldsymbol{\theta}\|$. Using a norm penalty means, given the same loss, we prefer solutions with small parameters.

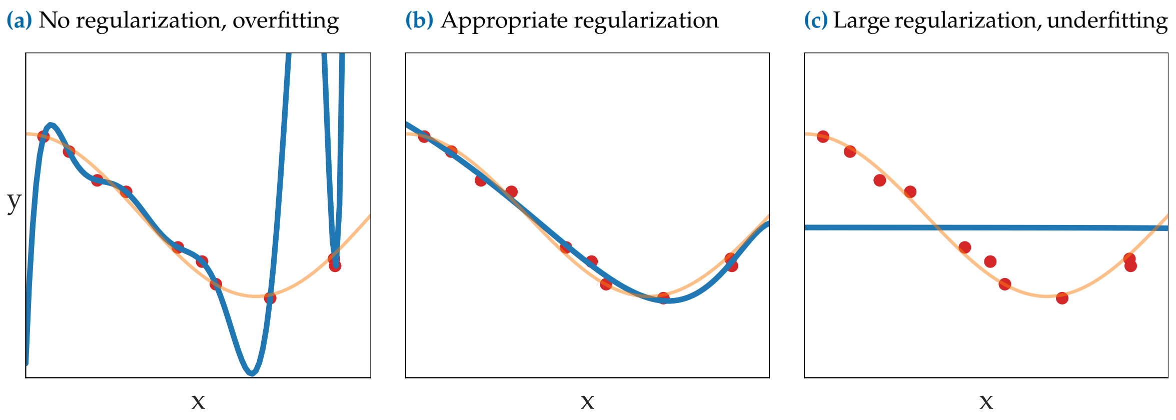


Figure 2.9: Fitting a polynomial with different regularization strengths. (a) When fitting a high-degree polynomial (here polynomial of degree 15) without regularization we overfit to the training data. (b) With an appropriate regularization strength, we can get a learned model that not only fits the training data correctly but also truthfully describes the ground truth function. Although the given polynomial is capable of describing more expressive functions, the added norm penalty encouraged the model to select a simpler function. (c) If the norm penalty is too strong, the model underfits and fails to describe even the training data.

Conversely, larger parameters are allowed, if they also further decrease the original loss function.

The choice of norm used in Equation (2.2) crucially determines the regularization effect. The most common choice is to use the squared L^2 parameter norm, $\|\boldsymbol{\theta}\|_2^2 = \sum_{i=1}^{D} \theta_i^2$. Adding the L^2 penalty is motivated by the fact that models with larger parameters tend to describe more flexible and in some sense more complicated functions. However, there is no clear measure for what describes a *simple* function and the L^2 norm is just one of many possible choices.

Another popular choice is the L^1 parameter norm, $\|\boldsymbol{\theta}\|_1 = \sum_{i=1}^{D} |\theta_i|$. While regularization with both norms pulls the optimal parameters closer to zero, their exact effects vary, depending on the norm, see Figure 2.10. In comparison to the L^2 penalty, L^1 regularization enforces sparsity.

The L^1 and L^2 regularization are simply special cases of using L^p parameter norms for regularization. However, they are by far the most popular choices. Depending on the domain, **$\mathbf{L^1}$ regularization** is also known as *Lasso regression*[12] [293] and **$\mathbf{L^2}$ regularization** is known as *ridge regression* [128] or *Tikhonov regularization* [295]. L^2 regularization is sometimes also referred to as **weight decay**. It is easy to see, where the name is coming from if one considers the update rule of Gradient Descent (GD) when using the regularized loss.[13] The gradient of the regularized loss is

$$\nabla_{\boldsymbol{\theta}} \tilde{L}(f_{\boldsymbol{\theta}}) = \nabla_{\boldsymbol{\theta}} L(f_{\boldsymbol{\theta}}) + 2\lambda\boldsymbol{\theta}\,, \tag{2.3}$$

and the update step of GD thus

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta(\nabla_{\boldsymbol{\theta}} L(f_{\boldsymbol{\theta}}) + 2\lambda\boldsymbol{\theta}^{(t)})\,, \tag{2.4}$$

which we can re-write as

$$\boldsymbol{\theta}^{(t+1)} = (1 - 2\eta\lambda)\boldsymbol{\theta}^{(t)} - \eta\nabla_{\boldsymbol{\theta}} L(f_{\boldsymbol{\theta}})\,. \tag{2.5}$$

Compared to an update step with GD on the unregularized loss (see Equation (3.19)), the update now includes a shrinkage step of the parameters. The L^2 regularization and the *weight decay* step as described in Equation (2.5) or by Hanson and Pratt [112] are, however, only equivalent when using GD or Stochastic Gradient Descent. Loshchilov and Hutter [194] proposed a new optimization method called AdamW that uses weight decay regularization for the popular Adam optimizer.

2.3.2 Data Augmentation

A simple way to prevent overfitting is to use more training data. Intuitively, the more diverse data available for training, the easier it is to find a model that will generalize to new data. The more the training data set represents the entire distribution of possible samples, the fewer *truly* unseen examples it has to process. However, collecting more data can be a time-consuming — and sometimes even infeasible — procedure.

Data augmentation can be a method to leverage the existing data as much as possible, by creating new synthetic data. Figure 2.11 shows typical data augmentation techniques used for images, such as random cropping, rotations, changes to the contrast, or adding noise. If the perturbations are small enough, they do not alter the corresponding label, *i.e.* a cat is still a cat, even when flipped horizontally. These new synthetic images can extend the existing training set and thus act as a regularizer.

The data augmentation highly depends on the data type. Augmenting the training data, by including rotated versions of the input generally helps for images but does not work for language inputs. But even within the same data type, one has to be careful to not affect the correct class. Flipping an image of a hot dog vertically, for example, results in another image showing a hot dog. Applying the same augmentation to the digit “six”, however, would create a “nine”. Data augmentation thus always requires a level of domain expertise in order to define data augmentation techniques that help extend the training set in a meaningful way, without changing the associated labels.

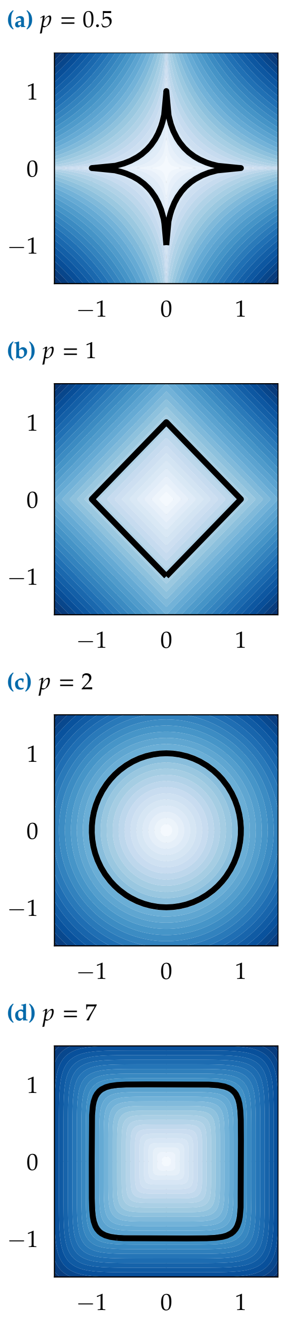


Figure 2.10: Visualization of different norms. Unit circles and heatmaps of different p-values for the L^p norm. Unit vectors for norms with smaller p-values lie closer to an axis, thus these norms tend to enforce sparsity.

12: Lasso stands for *least absolute shrinkage and selection operator*.

[293] Tibshirani (1996), “Regression Shrinkage and Selection via the Lasso”

[128] Hoerl et al. (1970), “Ridge Regression: Biased Estimation for Nonorthogonal Problems”

2.3.3 Optimization's Role in Regularization

The optimization procedure used to train the machine learning model itself can have regularization effects. A simple but popular approach is to stop training right before overfitting occurs. The stopping criterion for this process, known as **early stopping**, is usually based on the performance of the machine learning model on some holdout data, the *validation set*.[14] We often empirically observe that the training loss decreases consistently, but the performance metric on the validation set starts to increase at some point (see Figure 2.12). If we stop training early at this point, we receive a model with better validation set performance, and thus hopefully a better performance on truly unseen test data.

Early stopping is a popular regularization technique in deep learning since it is easy to implement and effective. The effects of early stopping could be similar to the regularization effects of using norm penalties (see Section 2.3.1) since the model's parameters are restricted in how far they can move from their initializing, given the limited training time [*e.g.*, 101].

It has long been theorized and discussed that the shape of the local loss landscape around the model's parameters influences their generalization performance. The intuitive idea is that **flat minima** generalize better since these solutions are relatively insensitive to small variations. *Flat minima* are characterized as having a comparably large neighborhood of similar low loss (see Figure 2.13). In contrast, sharp minima are more likely to be the artifacts of the limited finite training set, than the result of a general property of the true risk landscape. The relationship between the flatness of a minimum and its generalization capability has seen significant research in recent years [*e.g.*, 75, 125, 126, 148, 162, 212].

SGD with smaller batch sizes or larger learning rates appears to find flatter minima with better generalization properties [*e.g.*, 146]. Using smaller batch sizes can be viewed as injecting more noise into the learning process and thus regularizing the optimization problem. Similarly, larger learning rates, intuitively, have a harder time "falling into" sharper minima. Thus learning rates, that

[295] Tikhonov (1943), "On the stability of inverse problems"

13: The update rule of GD is simply to update the parameters in the direction of the negative gradient. See Section 3.3.1 for a more detailed discussion of GD and related optimization methods.

[112] Hanson et al. (1988), "Comparing Biases for Minimal Network Construction with Back-Propagation"

[194] Loshchilov et al. (2019), "Decoupled weight decay regularization"

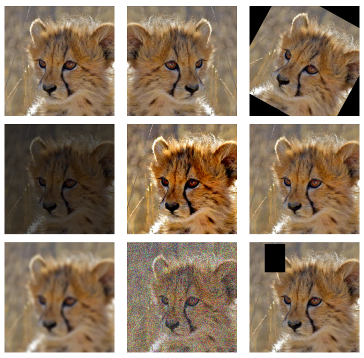
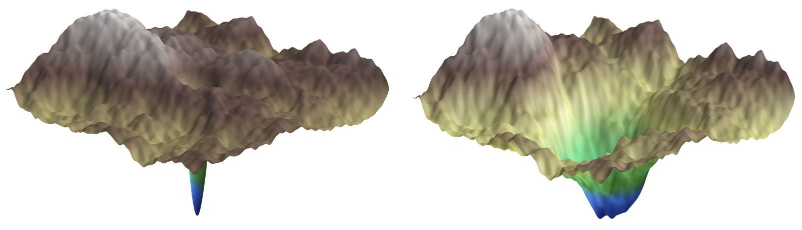


Figure 2.11: Different data augmentations applied to the image of an animal from the AFHQ data set [61]. The first row shows positional augmentations, with the top left image showing the original, top center a horizontal flip, and the top right a rotated version. The second row uses color augmentation, with the images showing (*from left to right*) brightness, contrast, and saturation augmentations. The bottom row introduces noise to the images by applying a Gaussian blur (*left*), Gaussian noise (*center*), and erasing random parts of the image (*right*).

[61] Choi et al. (2020), "StarGAN v2: Diverse Image Synthesis for Multiple Domains"



(a) Sharp minimum (b) Wide minimum

Figure 2.13: Illustration of a sharp versus a flat minimum. The figure illustrates an (artificial) loss landscape of a machine learning model with two parameters. Both landscapes contain a local minimum with a considerably low loss value. The difference between the sharp and the flat minimum is the wideness of the area of low loss.

maximize the test set performances are commonly larger than ones that minimize training loss. This indicates that small batch sizes or larger learning rates have beneficial effects beyond requiring less compute per step or reducing the number of total iterations. This implicit bias of SGD to prefer certain solutions is thus a form of regularization [275]. Some optimization methods, such as SAM (Sharpness-Aware Minimization) [86] or Entropy-SGD [50] make this preference more explicit.

14: Mahsereci et al. [199] presented a stopping criterion for early stopping without the need for a validation set.

[199] Mahsereci et al. (2017), "Early Stopping without a Validation Set"

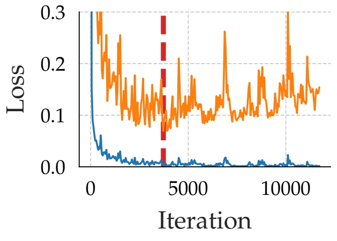


Figure 2.12: **Early stopping can be used to prevent overfitting.** The training loss (━) of a simple convolutional neural network on MNIST decreases as training progresses. The test loss (━) initially decreases as well but at some point plateaus and even begins to increase again. Stopping training earlier, at the point of minimal test loss (- -), can provide a solution with better generalization capabilities. Note that due to the stochastic nature of the training process, both loss curves are noisy.

[101] Goodfellow et al. (2016), "Deep Learning"

[75] Dinh et al. (2017), "Sharp Minima Can Generalize For Deep Nets"

[125] Hochreiter et al. (1994), "Simplifying Neural Nets by Discovering Flat Minima"

[126] Hochreiter et al. (1997), "Flat Minima"

[148] Jiang et al. (2019), "Fantastic Generalization Measures and Where to Find Them"

[162] Keskar et al. (2017), "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima"

[212] Mulayoff et al. (2020), "Unique Properties of Flat Minima in Deep Networks"

[146] Jastrzębski et al. (2018), "Three Factors Influencing Minima in SGD"

[275] Smith et al. (2021), "On the Origin of Implicit Regularization in Stochastic Gradient Descent"

[86] Foret et al. (2021), "Sharpness-aware Minimization for Efficiently Improving Generalization"

[50] Chaudhari et al. (2017), "Entropy-SGD: Biasing gradient descent into wide valleys"

Stochastic Optimization | 3

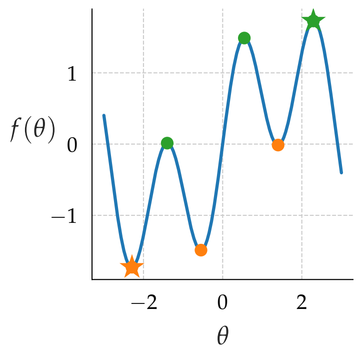
A central component of any machine learning pipeline is its optimization procedure. In machine learning, computers *learn* to solve a given task based on available data instead of explicitly being programmed for it (see Chapter 2). This effectively means that some strategy needs to be selected out of a large set of possible hypotheses. Selecting this strategy is done by an optimization method, also called an **optimizer**, which aims at selecting the "best" element with respect to some criterion among a set of possibilities. In order to understand the optimization methods at the heart of many machine learning applications, we can turn to the field of *mathematical optimization* (Section 3.1).

In this work, we study *empirical risk minimization* problems (Section 3.2), which include the scenario of supervised learning using deep neural networks. Training neural networks in a supervised fashion comes with some characteristic properties. Modern deep learning systems have millions or even billions of parameters, turning the underlying optimization problem into an extremely high-dimensional problem. The use of large data sets that themselves have millions of data points requires the *sub-sampling* of the data to allow efficient computation. This turns a regular optimization problem into a *stochastic* one, where both the function evaluations and the gradients are only observed with noise.

These properties cause many classical methods, originally developed for the noise-free setting, to become inefficient or unsuccessful. Recent research has resulted in an overwhelming and ever-growing list of new optimization methods for deep learning (see Section 3.3 and specifically Table 3.1). Navigating this large set of optimizers, each with their own set of hyperparameters, is currently a main challenge for many deep learning practitioners. Evaluating, comparing, and understanding these methods will be a core element of this work.

Many of the optimization methods for deep learning have hyperparameters that must be set by the user. These hyperparameters, most notoriously the learning rate, are often tuned and sometimes even dynamically changed during the training process. The tuning method, as well as self-tuning methods, are discussed in Section 3.4.

(a) One-dimensional loss function



(b) Two-dimensional loss function

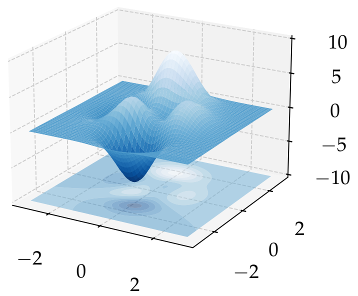


Figure 3.1: Illustration of loss functions for mathematical optimization. (a) A one-dimensional loss function (—) with multiple local minima (●) and local maxima (●). The global minimum and maximum of this periodic function are marked with a star (★,★). (b) A two-dimensional loss function with multiple local minima, local maxima, and saddle points.

3.1 Mathematical Optimization

In mathematical optimization, we are tasked with finding an extremum, *i.e.* either a minimum or a maximum, of some function $L(\boldsymbol{\theta})$. We can formalize an **optimization problem** in the following way:

Definition 3.1.1 [Optimization Problem]
Given a function $L(\boldsymbol{\theta}) : \mathbb{A} \to \mathbb{R}$, find the element $\boldsymbol{\theta}^* \in \mathbb{A}$ such that $L(\boldsymbol{\theta}^*) \leq L(\boldsymbol{\theta}) \,\forall\, \boldsymbol{\theta} \in \mathbb{A}$ (for **minimization**) or $L(\boldsymbol{\theta}^*) \geq L(\boldsymbol{\theta}) \,\forall\, \boldsymbol{\theta} \in \mathbb{A}$ (for **maximization**). The element $\boldsymbol{\theta}^*$ is known as the **minimizer** or **maximizer** of the function.

The function L that should be optimized is called **loss function**, **objective function**, **cost function**, or **risk** depending on the context. It is common, to focus on minimization since maximization of $L(\boldsymbol{\theta})$ is equivalent to minimization of $-L(\boldsymbol{\theta})$. We can write the minimization problem more concisely by

$$\min_{\boldsymbol{\theta} \in \mathbb{A}} L(\boldsymbol{\theta})\,. \tag{3.1}$$

In the following, we will assume that L is bounded, *i.e.* that the minimum of the function is within $(-\infty, \infty)$.

3.1.1 Global and Local Minima

A relaxation of Equation (3.1) describes *local minima*.[1] Here, all the function values are greater than or equal to the local minimum only in some region around it, *i.e.* [2]

1: All *global* extrema are also *local* extrema. Following Definition 3.1.2, we can just extend $\delta \to \infty$.

2: *Strict* local minima are defined similarly, but enforcing a strict inequality between the function values, *i.e.* $L(\boldsymbol{\theta}^*) < L(\boldsymbol{\theta})$ instead.

Definition 3.1.2 [Local Minimum]
For a local minimum $\boldsymbol{\theta}^*$, there exists some $\delta > 0$ such that $\forall\, \boldsymbol{\theta} \in \mathbb{A}$ with $\|\boldsymbol{\theta} - \boldsymbol{\theta}^*\| \leq \delta$ it holds that $L(\boldsymbol{\theta}^*) \leq L(\boldsymbol{\theta})$.

A local maximum is defined analogously. Figure 3.1 illustrates the crucial difference between a *local* and a *global* minimum.

If our objective function L has a first derivative, we can state a *necessary* condition for a minimum in terms of its gradient:[3]

3: To keep the presentation, we refrain from presenting the proofs of the theorems. They can be found in any standard textbook on optimization, *e.g.* [38, 222, 224].

[38] Boyd et al. (2004), "Convex Optimization"

[222] Nesterov (2018), "Lectures on Convex Optimization"

[224] Nocedal et al. (2006), "Numerical Optimization"

Theorem 3.1.3 [Necessary Condition for Optimality]
If $\boldsymbol{\theta}^* \in \mathbb{R}^D$ is a local minimum of a differentiable function $L : \mathbb{R}^D \to \mathbb{R}$, then

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*) = 0\,. \tag{3.2}$$

A point where the gradient vanishes is called a **stationary point**.

This, however, is only a *necessary* condition for a minimum. Stationary points of vanishing gradients could also be maxima, or **saddle points**. At a saddle point, the gradient is zero but it is neither a local minimum nor a local maximum. A point, which is a local minimum along one axis, but a local maximum along another axis, would be an example of such a saddle point, see Figure 3.2.

We can use the second derivative to distinguish between local minima, local maxima, and saddle points. For this, we use the *definiteness* of the Hessian:

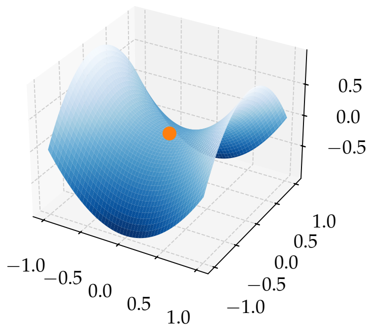


Figure 3.2: Illustration of a saddle point. At the saddle point (●), the gradient is zero. However, it is neither a local maximum nor a local minimum. Instead, it is a local minimum in one direction and a local maximum in the other direction.

Definition 3.1.4 [Definiteness of a Matrix]
A symmetric real matrix $\boldsymbol{A} \in \mathbb{R}^{d \times d}$ is called **positive definite**, if $\forall$ non-zero $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{x}^\top \boldsymbol{A} \boldsymbol{x} > 0$.

If only $\boldsymbol{x}^\top \boldsymbol{A} \boldsymbol{x} \geq 0$ holds, we call the matrix **positive semi-definite**. We denote these properties as $\boldsymbol{A} \succ 0$ and $\boldsymbol{A} \succeq 0$.

Analogously, we call a matrix **negative definite**, if $\boldsymbol{x}^\top \boldsymbol{A} \boldsymbol{x} < 0$ holds and denote it with $\boldsymbol{A} \prec 0$. A matrix, which is neither positive (semi-)definite nor negative (semi-)definite is called **indefinite**.

Theorem 3.1.5 [Sufficient Condition for Optimality]
If $L : \mathbb{R}^D \longrightarrow \mathbb{R}$ is twice-differentiable and $\boldsymbol{\theta}^* \in \mathbb{R}^D$ satisfies

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*) \quad \text{and} \quad \nabla^2_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*) \succ 0\,, \tag{3.3}$$

then $\boldsymbol{\theta}^*$ is a strict local minimum of L.

Conversely, a sufficient condition for a strict local maximum enforces $\nabla^2_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*) \prec 0$. If the Hessian $\nabla^2_{\boldsymbol{\theta}} L$ is indefinite, then the stationary point is a saddle-point.[4]

4: If the matrix is either positive semi-definite or negative semi-definite, this *second-derivative test* is inconclusive.

The definiteness of a matrix is connected to its eigenvalues. More specifically, if a matrix is positive definite, all of its eigenvalues are positive and if all eigenvalues are only non-negative, the matrix is positive semi-definite. The eigenvalues themselves have a geometric meaning. They describe the curvature of the local landscape in the direction of the associated eigenvector. The larger an eigenvalue, the more the function "bends upwards" in this direction and the higher the curvature. Similarly, a negative eigenvalue implies a "downward bend", *i.e.* a locally concave function in the direction of the eigenvector. This gives a geometric interpretation, why a positive definite matrix, *i.e.* one where all eigenvalues are positive, describes a local minimum, while a stationary point with both positive and negative eigenvalues describes a saddle-point.

3.1.2 Classification of Optimization Problems

Typically, optimization problems are classified based on properties of the objective function:

- **Convex vs. non-convex optimization (Figure 3.3):** For convex functions, any line segment between any two points on the graph of the function lies on or above the graph, *i.e.* $\forall t \in [0,1]$ and $\forall \boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \in \mathbb{A}$ it holds that $L(t\boldsymbol{\theta}_1 + (1-t)\boldsymbol{\theta}_2) \leq tL(\boldsymbol{\theta}_1) + (1-t)L(\boldsymbol{\theta}_2)$. Optimizing convex functions is much easier than non-convex objectives since in that case, it can be proven that any stationary point is also a global minimum.
- **Smooth vs. non-smooth problems (Figure 3.4):** In smooth problems, the objective function L is differentiable. Since the gradient is defined everywhere and the (negative) gradient always points in the direction of the steepest ascent (descent), it can be used to guide an optimization method. Depending on the community, smooth problems can also require further restrictions beyond being once-differentiable such as having a bounded Hessian.
- **Deterministic vs. stochastic functions (Figure 3.5):** For stochastic optimization functions, we can only observe the function (and if applicable, the gradient) with noise. This means that an optimization algorithm must deal with the uncertainty introduced by the noisy observations.

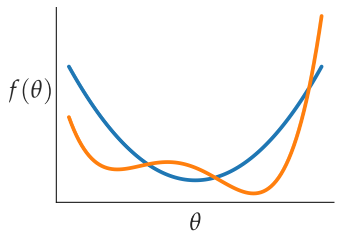


Figure 3.3: A convex vs. a non-convex function. Loosely speaking, a convex function (—) constantly "bends upward". In contrast, a non-convex function (—) can have multiple "ups and downs".

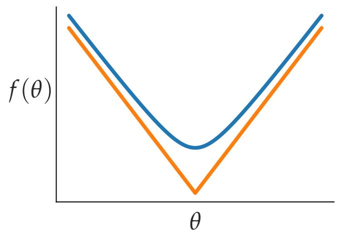


Figure 3.4: A smooth vs. a non-smooth function. The derivative of a smooth function (—) is defined everywhere. In contrast, a non-smooth function (—) can have kinks or steps.

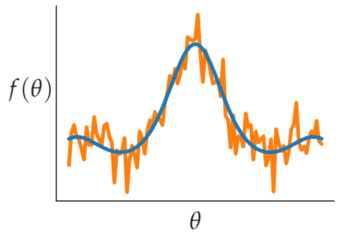


Figure 3.5: A deterministic vs. a stochastic function. Observing a deterministic function (—) with noise results in a stochastic function (one possible sample illustrated by —).

There are additional ways to categorize an optimization problem such as continuous vs. discrete optimization (in continuous optimization, the optimization variable is continuous as opposed to integers, for example), constrained vs. unconstrained domains (in unconstrained optimization, the domain is $\mathbb{R}^n$ for some n as opposed to a strict subset of $\mathbb{R}^n$ which is often formulated in terms of inequalities, *e.g.* $\min L(\boldsymbol{\theta})$ subject to $g(\boldsymbol{\theta}) \leq 0$), or linear vs. non-linear objectives.

Optimization for deep learning is typically in the domain of *non-convex*, *stochastic*, *continuous*, *unconstrained*, and *non-linear* optimization.

3.2 Empirical Risk Minimization

Let's consider the task of supervised learning for either regression or classification where we want to learn a function $f^*(\boldsymbol{x}) : \mathbb{X} \to \mathbb{Y}$ which is able to correctly predict the output $y \in \mathbb{Y}$ given some features $\boldsymbol{x} \in \mathbb{X}$. [5] We are also given a non-negative and real-valued **loss function** $\ell(\hat{y}, y) : \mathbb{Y} \times \mathbb{Y} \to \mathbb{R}^+$ that quantifies how different the predictions of a model $\hat{y}$ are from the true targets y.[6]

5: A popular example for this would be image classification, where the inputs $\boldsymbol{x}$ would be the raw pixel values of an image and the outputs y would describe the occurrence of a certain class in this image, *e.g.* $y = 1$ when the picture shows a cat and $y = 0$ if no cat is present in the image. f^* is in this example a perfect cat-detector that can tell, with 100 % accuracy, whether there is a cat in a given picture or not.

Turning this into a machine learning problem, we use a family of **model (or hypothesis) functions** $f_{\boldsymbol{\theta}}$ from some parameterized hypothesis space $\mathcal{F} = \left\{ f_{\boldsymbol{\theta}} = f(\boldsymbol{x}; \boldsymbol{\theta}) \,|\, \boldsymbol{\theta} \in \mathbb{R}^D \right\}$. Our goal is now to find the particular model or hypothesis from this fixed class of functions $\mathcal{F}$ that best describes the relationship between the inputs $\boldsymbol{x}$ and outputs y. Ideally, f^* should be part of the model space $\mathcal{F}$. However, this assumption is often unrealistic in practice. Nevertheless, for many practical applications, it is sufficient if $\mathcal{F}$ contains functions that serve as reasonable approximations to f^*.

6: In this chapter, we only consider loss functions that take $\hat{y}$ and y as arguments. It is rather trivial to extend the formulation to regularized losses that include, for example, the model parameters $\boldsymbol{\theta}$ as an argument.

If we assume a joint probability distribution $P_{\text{true}}(\boldsymbol{x}, y)$ that describes the true underlying data distribution, we can formulate the **true risk** or **expected loss** [7] of a specific model as:

7: The true risk is often just called *risk* or sometimes *expected risk*, *expected loss*, or *population risk*.

Definition 3.2.1 [True Risk or Expected Loss]
Consider a supervised learning problem. We define the **true risk** $L_{P_{\text{true}}}(f_{\boldsymbol{\theta}})$ of a hypothesis or model $f_{\boldsymbol{\theta}}$ as

$$\begin{aligned} L_{P_{\text{true}}}(f_{\boldsymbol{\theta}}) &= \mathbb{E}_{(\boldsymbol{x},y)\sim P_{\text{true}}(\boldsymbol{x},y)} \left[\ell\left(f(\boldsymbol{x}; \boldsymbol{\theta}), y \right) \right] \\ &= \int \ell\left(f(\boldsymbol{x}; \boldsymbol{\theta}), y \right) \mathrm{d}P_{\text{true}}(\boldsymbol{x}, y) \,. \end{aligned} \tag{3.4}$$

The goal of the machine learning problem is to find the function that minimizes this true risk: [8]

$$f^*_{\boldsymbol{\theta}} = \underset{f_{\boldsymbol{\theta}} \in \mathcal{F}}{\arg\min}\, L_{P_{\text{true}}}(f_{\boldsymbol{\theta}}) \,. \tag{3.5}$$

8: Instead of defining the risk of a function, *i.e.* $L_{P_{\text{true}}}(f_{\boldsymbol{\theta}})$, we could also express it as the risk of the *parameters*, *i.e.* $L_{P_{\text{true}}}(\boldsymbol{\theta})$. Since the function is parameterized by $\boldsymbol{\theta}$, this is an analogous view.

Obviously, this true underlying **population distribution** P_{true} is inaccessible. Loosely speaking, the expectation in Equation (3.4) is taken over the infinite set of "all possible input-output pairs", *e.g.* all possible images of handwritten digits and their associated digit. If P_{true} was known, minimizing Equation (3.4) would be a computable optimization problem.

While the true distribution is unknown, we often have access to a finite **training set** $\mathbb{D}_{\text{train}} = \left\{ (\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(N)}, y^{(N)}) \right\}$, where $\boldsymbol{x}^{(i)}$ is the input of the i-th example in the set and $y^{(i)}$ is the corresponding label. We assume the pairs in the training set are i.i.d. samples from the true population distribution. This allows us to convert the *machine learning* problem of finding a minimizer to the true risk in Equation (3.4) into an *optimization* problem by minimizing the **empirical risk** or empirical loss:

Definition 3.2.2 [Empirical Risk or Empirical Loss]
For a supervised learning problem, we define the **empirical risk**

$L_{\mathbb{D}_{\text{train}}}(f_{\boldsymbol{\theta}})$ on the training set $\mathbb{D}_{\text{train}}$ of a model $f_{\boldsymbol{\theta}}$ as

$$\begin{aligned} L_{\mathbb{D}_{\text{train}}}(f_{\boldsymbol{\theta}}) &= \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{P}_{\text{train}}(\boldsymbol{x},y)}\left[\ell\left(f(\boldsymbol{x};\boldsymbol{\theta}),y\right)\right] \\ &= \frac{1}{N}\sum_{i=1}^{N}\underbrace{\ell(f(\boldsymbol{x}^{(i)},\boldsymbol{\theta}),y^{(i)})}_{=:\ell^{(i)}}\,. \end{aligned} \tag{3.6}$$

Compared to the expected loss, we replaced the population distribution P_{true} with the **empirical distribution** $\hat{P}_{\text{train}}$ defined by the training set $\mathbb{D}_{\text{train}}$. Instead of Equation (3.5) we now optimize the model parameters $\boldsymbol{\theta}$ to find a minimizer of the empirical risk:

$$\hat{f}^{*}_{\boldsymbol{\theta}} = \arg\min_{f_{\boldsymbol{\theta}}\in\mathcal{F}} L_{\mathbb{D}_{\text{train}}}(f_{\boldsymbol{\theta}})\,. \tag{3.7}$$

The true goal, however, is still to find a minimizer of the true risk. Ultimately, we are interested in finding models that work on general samples from the true underlying distribution P_{true}. An image classifier for detecting vermin on crop plants, for example, should be able to correctly classify images "in the wild", not only the ones that are in the training data. The assumption is that the empirical risk $L_{\mathbb{D}_{\text{train}}}$ will approximate the true risk $L_{P_{\text{true}}}$ sufficiently well, such that a model minimizing the former will also perform reasonably well on the latter. Whether or not this is the case, is discussed under the question of **generalization**.

This description of empirical risk minimization leaves us with several open questions:

- **How do we select the hypothesis space $\mathcal{F}$?** In Section 4.1 we will see that artificial neural networks represent a particularly powerful and numerically appealing class of functions. For the majority of this work, we will focus on algorithms for training neural networks.
- **How do we find a minimizer of the risk?** Similar to the empirical risk being used as an approximation, we can use the empirical gradient of the loss as an estimation of the true gradient (see Section 3.2.1). Section 3.3 will then describe popular methods which use this stochastic gradient as a learning signal to train neural networks in particular.
- **How can we ensure that the selected model generalizes?** Which decisions influence whether a learned function can accurately predict the output not only for examples in the training data but also for unseen data? Questions like these were already raised and partially answered in Sections 2.2 and 2.3. In Section 3.2.2 we will look at the generalization performance again, this time through the lens of empirical risk minimization.

3.2.1 Gradient-Based Learning

For many machine learning models, including neural networks as we will see in Section 4.1, we can efficiently compute the gradient in addition to the loss value. While the loss value of the current parameters can give us an estimation for how well the model is performing, gradients provide us with a *direction* for how to improve the model. This crucial information allows the use of iterative gradient-based optimization methods.

Definition 3.2.3 [True Gradient]
Given a supervised learning problem, we define the **true** or **expected gradient** $g_{P_{\text{true}}}$ as the gradient of the true risk with respect to the model parameters, *i.e.*

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} L_{P_{\text{true}}}(f_{\boldsymbol{\theta}}) &= \mathbb{E}_{(\boldsymbol{x},y)\sim P_{\text{true}}(\boldsymbol{x},y)}\left[\nabla_{\boldsymbol{\theta}} \ell(f(\boldsymbol{x};\boldsymbol{\theta}),y)\right] \\ &=: g_{P_{\text{true}}} \,.\end{aligned} \tag{3.8}$$

Mirroring the statements above, this true gradient is inaccessible, since the true population distribution P_{true} is unknown. Instead, we can again make use of the empirical data set $\mathbb{D}_{\text{train}}$ and its empirical distribution $\hat{P}_{\text{train}}$, to compute an estimate of the true gradient, which we will call the **empirical gradient**.

Definition 3.2.4 [Empirical Gradient]
We can approximate the true gradient using an empirical data set with the **empirical gradient** $g_{\mathbb{D}_{\text{train}}}$ defined as

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} L_{\mathbb{D}_{\text{train}}}(f_{\boldsymbol{\theta}}) &= \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \ell(f(\boldsymbol{x}^{(i)},\boldsymbol{\theta}),y^{(i)}) \\ &=: g_{\mathbb{D}_{\text{train}}} \,.\end{aligned} \tag{3.9}$$

Data Sub-Sampling via Mini-Batches

Contemporary data sets can easily have millions of data points and in those cases computing the empirical gradient can be expensive.[9] The empirical gradient itself, however, was only an approximation of the quantity we actually care about, the true gradient. Instead of approximating the true gradient with the full training data set, we might as well use a smaller set of data points to compute an approximation. For this, we define a **mini-batch**, which is a sub-sampled set of the training data with $\mathbb{B} = \left\{(\boldsymbol{x}^{(1)},y^{(1)}),\dots,(\boldsymbol{x}^{(B)},y^{(B)})\right\} \subseteq \mathbb{D}_{\text{train}}$.[10] The number of elements in $\mathbb{B}$ is referred to as the batch size and we denote it as $|\mathbb{B}| = B$.

9: The following analysis is true not only for the gradient but the loss, as well. In practice, both the loss and the gradient are computed on a mini-batch.

10: Note, that we overload the notation here. $\boldsymbol{x}^{(i)}$ can both mean the i-th element in the entire data set or just within a single batch. Which one it refers to is usually clear from the context, *e.g.* the sum in Equation (3.9) is over elements from the data set while Equation (3.10) uses just elements within a batch.

Using this mini-batch, we can now define the **mini-batch gradient**.

Definition 3.2.5 [Mini-Batch Gradient]
Given a mini-batch $\mathbb{B}$ of training samples, we can estimate the empirical gradient via the **mini-batch gradient** $g_{\mathbb{B}}$, given by

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} L_{\mathbb{B}}(f_{\boldsymbol{\theta}}) &= \frac{1}{B}\sum_{i=1}^{B} \underbrace{\nabla_{\boldsymbol{\theta}} \ell(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), y^{(i)})}_{=: g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})} \\ &=: g_{\mathbb{B}} \,.\end{aligned} \tag{3.10}$$

Denoted by $g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})$ are the **individual gradients**, *i.e.* the gradient vector resulting from a simple datum.

Comparing Definitions 3.2.4 and 3.2.5, it is easy to see that conceptually, there is no real difference between approximating the true gradient via the expected gradient or via the mini-batch gradient. Both use an empirical set of data points sampled from the true distribution. With the mini-batch gradient, however, we can smoothly control the trade-off between approximation quality and computational cost via the batch size: Larger batch sizes provide a better approximation but require the more costly computation of more examples.

The batch size is therefore often chosen to maximally utilize the available memory of the hardware, but additional effects such as the relationship between the batch size and the generalization capabilities might be taken into account. The mini-batches are often selected randomly by shuffling all the available training data and separating it into $\lfloor N/B \rfloor$ batches of size B that will be each used for one iteration (see Section 3.3).[11]

11: The remaining data points could either be used in a smaller final batch or be dropped for this division.

Beyond the Gradient: Higher-Order Derivatives

We can, of course, use the same strategy that we have used for the gradient, the first-order derivative, and apply it to higher-order derivatives, such as the Hessian. The Hessian provides information about the curvature that characterizes the local loss landscape which can be used to more informatively steer the optimization process. Defining the **mini-batch Hessian** works analogously to the mini-batch loss or the mini-batch gradient:

Definition 3.2.6 [Mini-Batch Hessian]
Similar to the mini-batch gradient, we can define the **mini-batch**

Hessian for a supervised learning problem as

$$\begin{aligned}\nabla^2_{\boldsymbol{\theta}} L_{\mathbb{B}}(f_{\boldsymbol{\theta}}) &= \frac{1}{B}\sum_{i=1}^{B} \nabla^2_{\boldsymbol{\theta}} \ell(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), y^{(i)}) \\ &=: \boldsymbol{H}_{\mathbb{B}} \,.\end{aligned} \tag{3.11}$$

The Hessian is a matrix of size $\boldsymbol{H} \in \mathbb{R}^{D\times D}$. For many practical machine learning problems, the number of parameters can easily be in the millions, thus computing or even storing the Hessian is infeasible. To make practical use of the Hessian, we require approximations, which are briefly described in Section 3.3.2.

Beyond the Mean: Higher-Order Statistics

Similar to computing higher-order derivatives, we can also consider higher-order statistical moments. Looking at the variance, instead of the expectation as in Definition 3.2.1, provides us with the variance of the true risk,

Definition 3.2.7 [True Risk Variance or True Loss Variance]
Given Definition 3.2.1, the **variance of the true risk** $\Lambda_{P_{\text{true}}}(f_{\boldsymbol{\theta}}) \in \mathbb{R}$ is given by

$$\begin{aligned}\Lambda_{P_{\text{true}}}(f_{\boldsymbol{\theta}}) &= \operatorname{Var}_{(\boldsymbol{x},y)\sim P_{\text{true}}(\boldsymbol{x},y)}\left[\ell(f(\boldsymbol{x};\boldsymbol{\theta}), y)\right] \\ &= \int \left(\ell\left(f(\boldsymbol{x};\boldsymbol{\theta}), y\right) - L_{P_{\text{true}}}(f_{\boldsymbol{\theta}})\right)^2 \mathrm{d}P_{\text{true}}(\boldsymbol{x}, y)\,.\end{aligned} \tag{3.12}$$

as well as an empirical approximation thereof:

Definition 3.2.8 [Empirical Risk Variance or Empirical Loss Variance]
We can approximate the true loss variance, as defined by Definition 3.2.7, by the sample variance using some data set $\mathbb{D}$:

$$\Lambda_{P_{\text{true}}}(f_{\boldsymbol{\theta}}) \approx \frac{1}{|\mathbb{D}|-1}\sum_{i=1}^{|\mathbb{D}|}\left(\ell(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), y^{(i)}) - L_{\mathbb{D}}(f_{\boldsymbol{\theta}})\right)^2 =: \hat{\Lambda}_{\mathbb{D}}\,. \tag{3.13}$$

Just like before, conceptually there is no real difference, whether we use the full training data set $\mathbb{D} = \mathbb{D}_{\text{train}}$ or just a single mini-batch $\mathbb{D} = \mathbb{B}$ for this approximation.

Of course, the variance can also be computed not only for the losses but also for the gradients:

Definition 3.2.9 [True Gradient Variance]

Given the true gradient of Definition 3.2.3, its variance is defined as

$$\begin{aligned}\Sigma_{P_{\text{true}}}(f_{\boldsymbol{\theta}}) &= \operatorname{Var}_{(x,y)\sim P_{\text{true}}(x,y)}\left[\nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{x};\boldsymbol{\theta}),y)\right] \\ &= \int \left(\nabla_{\boldsymbol{\theta}}\ell\left(f(\boldsymbol{x};\boldsymbol{\theta}),y\right) - \nabla_{\boldsymbol{\theta}}L_{P_{\text{true}}}(f_{\boldsymbol{\theta}})\right) \\ &\qquad \left(\nabla_{\boldsymbol{\theta}}\ell\left(f(\boldsymbol{x};\boldsymbol{\theta}),y\right) - \nabla_{\boldsymbol{\theta}}L_{P_{\text{true}}}(f_{\boldsymbol{\theta}})\right)^{\top} \mathrm{d}P_{\text{true}}(\boldsymbol{x},y)\,.\end{aligned} \tag{3.14}$$

Definition 3.2.10 [Empirical Gradient Variance]
We can approximate the true gradient variance, as defined by Definition 3.2.9, by the sample variance using some data set $\mathbb{D}$:

$$\Sigma_{P_{\text{true}}}(f_{\boldsymbol{\theta}}) \approx \frac{1}{|\mathbb{D}|-1}\sum_{i=1}^{|\mathbb{D}|}\left(g_{\mathbb{D}}^{(i)} - g_{\mathbb{D}}\right)\left(g_{\mathbb{D}}^{(i)} - g_{\mathbb{D}}\right)^{\top} =: \hat{\Sigma}_{\mathbb{D}}\,. \tag{3.15}$$

Leveraging these higher-order statistical quantities will be a central aspect of Chapter 7. A straightforward use-case of the empirical variance is to approximate the quality of our mini-batch gradient estimate via the covariance matrix of the mini-batch gradient:

$$\sqrt{\operatorname{Var}\left[g_{\mathbb{B}}\right]} = \frac{\Sigma_{P_{\text{true}}}^{1/2}}{\sqrt{B}} \approx \frac{\hat{\Sigma}_{\mathbb{B}}^{1/2}}{\sqrt{B}} \tag{3.16}$$

This also shows that increasing the batch size has diminishing returns. Increasing the batch size by a factor of 100 increases the required computations by the same factor but reduces the standard error of the gradient estimate only by a factor of $\sqrt{100} = 10$.

3.2.2 Optimization's View on Generalization

We can use the principle of empirical risk minimization to take another look at generalization (see Section 2.2). Defining the following objects:

Symbol	Definition	Description
	$\min L_{P_{\text{true}}}$	The minimal achievable risk by any predictor
$f_{\boldsymbol{\theta}}^{*}$	$= \arg\min_{f_{\boldsymbol{\theta}}\in\mathcal{F}} L_{P_{\text{true}}}(f_{\boldsymbol{\theta}})$	Minimizer of the true risk
$\hat{f}_{\boldsymbol{\theta}}^{*}$	$= \arg\min_{f_{\boldsymbol{\theta}}\in\mathcal{F}} L_{\mathbb{D}_{\text{train}}}(f_{\boldsymbol{\theta}})$	Minimizer of the empirical risk
$\hat{f}_{\boldsymbol{\theta}^{(t)}}$		Current model function

We can decompose the difference between the risk of our current estimate and the lowest achievable risk as

$$L_{P_{\text{true}}}(\hat{f}_{\boldsymbol{\theta}^{(t)}}) - \min L_{P_{\text{true}}} = \underbrace{L_{P_{\text{true}}}(\hat{f}_{\boldsymbol{\theta}^{(t)}}) - L_{P_{\text{true}}}(\hat{f}^*_{\boldsymbol{\theta}})}_{\text{optimization error}} + \underbrace{L_{P_{\text{true}}}(\hat{f}^*_{\boldsymbol{\theta}}) - L_{P_{\text{true}}}(f^*_{\boldsymbol{\theta}})}_{\text{estimation error}} + \underbrace{L_{P_{\text{true}}}(f^*_{\boldsymbol{\theta}}) - \min L_{P_{\text{true}}}}_{\text{approximation error}} . \tag{3.17}$$

The difference thus consists of three error terms:

- **Approximation error:** The **approximation error** is a result of only considering a restricted class of model functions $\mathcal{F}$ which does not necessarily include the "true minimizer". Increasing the model capacity tends to decrease the approximation error, as better approximations of the true minimizer are included in the hypothesis space $\mathcal{F}$. This is related to the *bias* term mentioned in the bias-variance tradeoff (Section 2.2.2).
- **Estimation error:** Increasing the model capacity often comes at the cost of also increasing the **estimation error**. The estimation error is a result of optimizing the *empirical* risk instead of the *true* risk. Since a high-capacity model is more expressive, it is more likely to overfit to the given data, reducing the empirical risk but often-times increasing the true risk. Increasing the amount of training data, or using data augmentation techniques (see Section 2.3.2), generally reduces the estimation error as it reduces the gap between the empirical data set and the true population.
- **Optimization error:** The remaining term is the **optimization error**. It describes the discrepancy between the globally optimal solution in $\mathcal{F}$ and the (current) solution found by the optimization method.

Balancing these three error terms is a central challenge of machine learning in practice. Changing the model capacity, the number of examples in the training set, or the runtime of the optimization method all affect the total computation time, the three error terms, and therefore the generalization capability of the machine learning system [*e.g.*, 37].

[37] Bottou et al. (2007), "The Trade-offs of Large Scale Learning"

If we look at overfitting from the optimizer's perspective, one can observe that overfitting occurs over the course of the training process, as seen, for example, in Figure 2.12. Since the optimizer operates on the empirical risk, we can expect it to continuously decrease the empirical risk throughout the optimization. But the same must not be true for the *true* risk. We often observe that initially the true risk decreases as well, but at some point increases again, even when the empirical risk continues to decrease. At this point, the learned solution overfits to the empirical data set which harms the performance on the true population.

This reiterates the point that learning is crucially different from optimization (see Section 2.2). Even more pointedly, the best optimization method may not necessarily be the best *training* method. An optimizer that quickly arrives at a reasonable solution might in practice be preferable to an optimization algorithm that can achieve a lower empirical risk if trained for a long time. Additional qualities such as which local minimum the optimizer ends up in (see Section 2.3.3), have to be considered when studying optimization methods for deep learning.

3.3 Optimization Methods

An optimization method is any algorithm that aims to solve an optimization problem, *i.e.* to find $\boldsymbol{\theta}^*$ such that $\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}\in\mathbb{A}} L(\boldsymbol{\theta})$ (see Equation (3.1)). Which type of optimization method is used, usually depends on the categorization of the optimization problem (see Section 3.1.2). In deep learning, where we typically deal with non-convex, stochastic functions, we commonly use first-order **iterative** optimization methods.

Given an initial starting point $\boldsymbol{\theta}^{(0)} \in \mathbb{R}^D$, *iterative* optimization methods produce a sequence of approximate solutions $\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \dots,$ where the next iterate $\boldsymbol{\theta}^{(t+1)}$ is computed by taking a step of **step size** $\eta^{(t)}$ into an **update direction**[12] $\boldsymbol{s}^{(t)}$ starting from the current estimate $\boldsymbol{\theta}^{(t)}$, *i.e.*

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta^{(t)}\boldsymbol{s}^{(t)} . \tag{3.18}$$

In the context of machine learning and deep learning, the *step size* $\eta^{(t)}$ is also called the **learning rate** of the optimization algorithm.[13] Equation (3.18) is known as the **update rule** of the optimizer and one update of the parameters following this equation is generally considered one **iteration** of the algorithm.[14] It is common to use a different batch in each iteration, *i.e.* that each evaluation of the loss function and its gradient is performed on a different batch, to avoid overfitting. Once each available batch is used, this marks the end of a single training **epoch**, signifying that every example in the training data set was used once to inform the optimization methods. Depending on, among other things, the size of the training data set, contemporary neural networks are trained for hundreds of epochs.

12: The *update direction* of an optimizer is sometimes also called the **search direction**, especially, when the algorithm itself determines an appropriate step size, *e.g.* by testing multiple learning rates.

13: In Equation (3.18), we used a scalar-valued step size, but many optimization methods, including some described in Section 3.3.1, define *vector-valued* step sizes that are multiplied element-wise with the update direction. This is an abstract change rather than a practical one. Using an element-wise learning rate is equivalent to changing the update direction and using a scalar step size. Whether the update rule is presented as having a scalar learning rate or a vector-wise learning rate (with different update vectors) is mostly a stylistic choice that can emphasize certain aspects.

14: In some cases, an *iteration* can be defined differently. Line searches, for example, probe different step sizes to find a suitable one, before performing a parameter update. This means that the loss function (and the gradient) are evaluated multiple times before the next iterate is computed. In deep learning, a loss function evaluation is a costly process and therefore it sometimes is more helpful to compare the optimization process per loss evaluation instead of per parameter update.

The use of gradient (or even Hessian) information allows an efficient search for the optimal model parameters without having to explore the entire domain, which for modern neural networks with tens or hundreds of millions of parameters is plainly impossible. The drawback of using this derivative information to steer the optimization process is that they tend to only find *local* minima. Empirically, however, these local minima often perform well enough. Finding a global minimizer may in fact not be desirable in cases where the optimization problem is part of a machine learning problem as it could provide worse generalization.

Because optimization methods play such a central role in machine learning (and many other scientific domains), it is not surprising that countless methods have been developed. Focusing only on optimizers that have been suggested for or applied in deep learning, we identify more than 150 optimization methods (see Table 3.1).

Algorithm 3.1 illustrates a sketch of a typical iterative optimizer for deep learning. The crucial question, that will define an optimization method, is how the learning rate and the update direction will be determined. Although general quantities might be used to compute both the learning rate and the update direction, it is common to distinguish between methods that only use the gradient (*first-order* methods, described in Section 3.3.1) and algorithms that include Hessian information as well (*second-order* methods, described in Section 3.3.2).

```python
def IterativeOptimizer(evaluate_func, init_params, max_epochs):
    params = init_params()
    for epoch in range(max_epochs):
        loss, gradients,... = evaluate_func()
        s = compute_update_direction(gradients,...)
        lr = compute_learning_rate(...)
        params = params + lr * s
    return params
```

Algorithm 3.1: Algorithmic sketch of a typical iterative optimization method for deep learning.

3.3.1 First-order Methods

First-order iterative methods are the most common optimization algorithms in deep learning. In this section, we will provide a description of the most popular optimizers, including all fifteen methods that are part of the optimizer benchmark in Chapter 6.

Stochastic Gradient Descent

As described in Section 3.2.1, we can compute the gradient of the loss function to efficiently obtain a signal of how to (locally) decrease the loss. The most straightforward method to leverage this gradient information is the method of ***steepest descent*** also

Table 3.1: **Table of deep learning optimizers.** Note, that this extensive list is still far from being complete and shows only a subset of all existing methods for deep learning.

Name	Ref.	Name	Ref.	Name	Ref.
AcceleGrad	[179]	C-ADAM	[303]	PAGE	[181]
ACClip	[348]	CADA	[55]	PAL	[214]
AdaAlter	[327]	Cool Momentum	[33]	PolyAdam	[226]
AdaBatch	[71]	CProp	[231]	Polyak	[230]
AdaBayes/AdaBayes-SS	[7]	Curveball	[120]	PowerSGD/PowerSGDM	[309]
AdaBelief	[361]	Dadam	[220]	Probabilistic Polyak	[244]
AdaBlock	[338]	DeepMemory	[320]	ProbLS	[200]
AdaBound	[196]	DGNOpt	[185]	PStorm	[331]
AdaComp	[51]	DiffGrad	[77]	QHAdam/QHM	[197]
Adadelta	[341]	EAdam	[336]	RAdam	[189]
Adafactor	[267]	EKFAC	[94]	Ranger	[321]
AdaFix	[14]	Eve	[114]	RangerLars	[106]
AdaFom	[56]	Expectigrad	[65]	RMSProp	[294]
AdaFTRL	[225]	FastAdaBelief	[356]	RMSterov	[60]
Adagrad	[78]	FRSGD	[312]	S-SGD	[280]
ADAHESSIAN	[333]	G-AdaGrad	[47]	SAdam	[315]
Adai	[329]	GADAM	[347]	Sadam/SAMSGrad	[298]
AdaLoss	[290]	Gadam	[107]	SALR	[337]
Adam	[166]	GOALS	[46]	SAM	[86]
Adam$^+$	[190]	GOLS-I	[155]	SC-Adagrad/SC-RMSProp	[211]
AdamAL	[289]	Grad-Avg	[232]	SDProp	[139]
AdaMax	[166]	GRAPES	[69]	SGD	[242]
AdamBS	[191]	Gravilon	[161]	SGD-BB	[287]
AdamNC	[238]	Gravity	[17]	SGD-G2	[12]
AdaMod	[74]	HAdam	[149]	SGDEM	[236]
AdamP/SGDP	[121]	HyperAdam	[315]	SGDHess	[299]
AdamT	[353]	K-BFGS/K-BFGS(L)	[100]	SGDM	[188]
AdamW	[194]	KF-QN-CNN	[240]	SGDR	[193]
AdamX	[300]	KFAC	[203]	SHAdagrad	[136]
ADAS	[81]	KFLR/KFRA	[35]	Shampoo	[10, 110]
AdaS	[130]	L4Adam/L4Momentum	[243]	SignAdam++	[313]
AdaScale	[152]	LAMB	[335]	SignSGD	[28]
AdaSGD	[314]	LaProp	[362]	SKQN/S4QN	[332]
AdaShift	[358]	LARS	[334]	SM3	[11]
AdaSqrt	[134]	LHOPT	[8]	SMG	[301]
Adathm	[279]	LookAhead	[349]	SNGM	[352]
AdaX/AdaX-W	[180]	M-SVAG	[20]	SoftAdam	[85]
AEGD	[186]	MADGRAD	[68]	SRSGD	[311]
ALI-G	[29]	MAS	[173]	Step-Tuned SGD	[44]
AMSBound	[196]	MEKA	[54]	SWATS	[163]
AMSGrad	[238]	MTAdam	[201]	SWNTS	[57]
AngularGrad	[247]	MVRC-1/MVRC-2	[58]	TAdam	[140]
ArmijoLS	[308]	Nadam	[76]	TEKFAC	[91]
ARSG	[57]	NAMSB/NAMSG	[57]	VAdam	[164]
ASAM	[171]	ND-Adam	[350]	VR-SGD	[266]
AutoLRS	[150]	Nero	[192]	vSGD-b/vSGD-g/vSGD-l	[259]
AvaGrad	[256]	Nesterov	[221]	vSGD-fd	[258]
BAdam	[252]	Noisy Adam/Noisy K-FAC	[344]	WNGrad	[322]
BGAdam	[18]	NosAdam	[135]	YellowFin	[346]
BPGrad	[351]	Novograd	[97]	Yogi	[340]
BRMSProp	[7]	NT-SGD	[357]		
BSGD	[133]	Padam	[53]		

called **Gradient Descent (GD)**.[15] This iterative first-order method computes a new parameter estimate via

Update Rule 3.3.1 [Gradient Descent (GD)]

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta g_{\mathbb{D}_{\text{train}}}, \tag{3.19}$$

15: Gradient Descent is often attributed to Cauchy [45] but was most likely independently invented multiple times.

[45] Cauchy (1847), "Méthode générale pour la résolution des systemes d'équations simultanées"

where $\eta \in \mathbb{R}^+$ is the *step size* or *learning rate* determining the update size of each iteration and $g_{\mathbb{D}_{\text{train}}}$ is the gradient evaluated on all training samples from $\mathbb{D}_{\text{train}}$. Determining an appropriate learning rate presents a major challenge for deep learning, which we will discuss in more detail in Section 3.4.

Using GD requires computing the gradient over the *entire* data set for a single iteration of the optimization method. For large data sets, this is computationally expensive and intractably slow. As discussed in Section 3.2.1 one can also use an approximation of the empirical gradient only using a single batch. This results in an algorithm known as **Stochastic Gradient Descent (SGD)**,[16] where the update is performed by

16: SGD can be traced back to Robbins and Monro [242] with Kiefer and Wolfowitz [165] describing the method in its contemporary form.

[242] Robbins et al. (1951), "A Stochastic Approximation Method"

[165] Kiefer et al. (1952), "Stochastic estimation of the maximum of a regression function"

Update Rule 3.3.2 [Stochastic Gradient Descent (SGD)]

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta g_{\mathbb{B}^{(t)}}. \tag{3.20}$$

Here $g_{\mathbb{B}^{(t)}}$ denotes the gradient computed on the mini-batch drawn at iteration t of the optimization routine.[17] If the batch size is chosen appropriately one can obtain a reasonable approximation of the true empirical gradient at a much cheaper cost of $\mathcal{O}(B)$ instead of $\mathcal{O}(N)$. It is not unusual for contemporary data sets to have millions or even billions of examples, yet successfully training a model is possible using batch sizes on the order of fewer than a thousand samples, *i.e.* $B \ll N$. This very effective reduction in cost makes SGD one of the most popular optimization method for large-scale machine learning and especially deep learning.

17: Some works in the literature distinguish between Stochastic Gradient Descent and *mini-batch gradient descent* where the former uses a single training example to compute the gradient estimate. In this work, we use SGD and *mini-batch gradient descent* synonymously, considering the former a special case with $B = 1$.

Momentum Methods

A known problem of SGD is that it oscillates when navigating tight valleys or ravines [282].[18] These landscapes are characterized by having drastically different curvatures in different directions. Figure 3.6 illustrates a two-dimensional loss function with one high-curvature direction (→) and one low-curvature direction (→). Considering only the component of the gradient pointing in the direction of high curvature, we can observe that subsequent gradients repeatedly point in the opposite direction. For the direction of low curvature, the situation is reversed. Here, successive gradients point consistently in the same direction.

The observed learning signal in the low-curvature direction is therefore very clear, while it is much more inconclusive in the high-curvature direction. The magnitude of each Hessian eigenvalue indicates the rate of change of the slope of the loss landscape in the direction of its eigenvector, which makes it natural to trust the gradient more in the low-curvature direction and thus take a larger step. We can accomplish this by adding a "short-term memory" to our update rule, which changes the update in each direction based on the observation of past gradients: [19]

Update Rule 3.3.3 [(Heavy Ball) Momentum]

$$v^{(t)} = \rho v^{(t-1)} + \eta g_{\mathbb{B}^{(t)}} \tag{3.21}$$
$$\theta^{(t+1)} = \theta^{(t)} - v^{(t)} . \tag{3.22}$$

The introduced parameter $\rho \in [0, 1)$ is called the *momentum factor* that determines how much weight is given to older gradients and how fast the information from older gradients decays. Common values for ρ are $0.5, 0.9, 0.99$ or even 0.999 [*e.g.*, 99, 101], but it can also be tuned similar to other optimization hyperparameters (see Section 3.4.1) or even be scheduled [*e.g.*, 273, 281, 311].

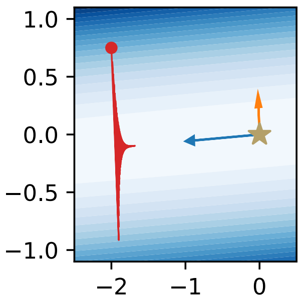
This optimization method was proposed by Polyak [230] and is called (classical) **Momentum** or *heavy ball method* due to relating the update equation to momentum in physics.[20] Here, the newly introduced variable v plays the role of the velocity with the negative gradients at each iteration acting as a force moving the object - the eponymous *heavy ball* - through the loss landscape. The accumulation of successive gradients means that the Momentum optimizer will tend to keep traveling in the same direction. This physics analogy can be helpful to gain an intuitive understanding of how adding momentum affects the optimization trajectory. [21]

Nesterov [221] introduced a slight modification to the classical momentum method described above. The resulting update rule is now known as **Nesterov Accelerated Gradient** (NAG) [22] and is given by

Update Rule 3.3.4 [Nesterov Accelerated Gradient (NAG)]

$$v^{(t)} = \rho v^{(t-1)} + \eta \frac{1}{B} \sum_{i=1}^{B} \nabla_{\theta} \ell(f(x^{(i)}, \theta - \rho v^{(t-1)}), y^{(i)}) \tag{3.23}$$
$$\theta^{(t+1)} = \theta^{(t)} - v^{(t)} . \tag{3.24}$$

(a) SGD



(b) Momentum

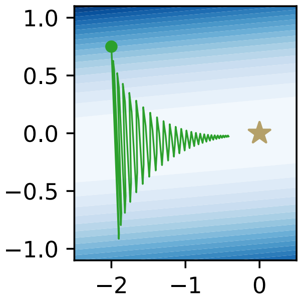


Figure 3.6: Illustration of the difference between SGD and Momentum. (a) On a two-dimensional deterministic quadratic loss function with one high-curvature (→) and one low-curvature direction (→) SGD (—) oscillates heavily without much progress towards the minimum (★). (b) Momentum (—) accumulates "velocity" in the low-curvature direction and thus progresses much more towards the minimum compared to SGD.

[282] Sutton (1986), "Two problems with back propagation and other steepest descent learning procedures for networks"

18: Valleys or ravines are also called *trenches*, *canyons*, or *canals* in the literature.

19: There exist subtly different versions of the update rule for the Momentum optimization method. PyTorch, for example, uses a variant where the learning rate is applied to the momentum update $v^{(t)}$ instead of applying it to the gradient which can slightly alter the behavior when learning rate schedules are used. A common alternative description of momentum is given by $\theta^{(t+1)} = \theta^{(t)} - \eta g_{\mathbb{B}^{(t)}} + \rho(\theta^{(t)} - \theta^{(t-1)})$.

Remembering that $g_{\mathbb{B}^{(t)}} := \frac{1}{B}\sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} \ell(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), y^{(i)})$ we can see that the only difference between Update Rules 3.3.3 and 3.3.4 is where the gradient is evaluated (compare Equations (3.21) and (3.23)). Since we know that we will use the velocity term $\boldsymbol{v}^{(t)}$ to update the parameters, we can do this step first, and evaluate the gradient at this point (see Figure 3.7).

In practice, the Momentum and NAG optimizers have been successfully applied to deep learning. They tend to greatly improve the training stability and speed of neural network training in particular for deep models using image data [*e.g.*, 9, 115, 117, 239, 281, 284]. The observed improved performance over SGD on some deep learning problems is often attributed to their ability to better utilize small but consistent gradients, handle noisy gradients, and navigate through ill-conditioned local loss landscapes.

[99] Goh (2017), "Why Momentum Really Works"
[101] Goodfellow et al. (2016), "Deep Learning"
[273] Smith (2018), "A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay"
[281] Sutskever et al. (2013), "On the importance of initialization and momentum in deep learning"
[311] Wang et al. (2020), "Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent"
[230] Polyak (1964), "Some methods of speeding up the convergence of iteration methods"

20: *(Classical) momentum*, *heavy ball method*, and *Polyak's momentum* are all names used for the method described by Equation (3.22).

21: An alternative interpretation for Momentum is based on assuming that the step sizes are comparably small. In this view, Momentum accumulates multiple batches of gradients and thus smoothes the gradient updates, effectively using information from a larger batch of gradients for each parameter update.

AdaGrad

AdaGrad (short for *adaptive gradient method*[23]) introduced the approach of re-scaling the learning rate element-wise. [24] Duchi et al. [78] proposed to scale it inversely proportional to the square root of the sum of the past squared gradient values:

Update Rule 3.3.5 [AdaGrad]

$$\boldsymbol{s}^{(t)} = \boldsymbol{s}^{(t-1)} + g_{\mathbb{B}^{(t)}} \odot g_{\mathbb{B}^{(t)}} \tag{3.25}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{s}^{(t)}} + \varepsilon} \odot g_{\mathbb{B}^{(t)}}\,, \tag{3.26}$$

where the square root and division are applied element-wise. Here, the additional parameter $\varepsilon > 0$ is a small constant that was originally introduced to avoid a division by zero.[25] This inverse scaling results in a considerable decrease of the learning rate for directions with large historic gradients and conversely relatively moderate decline of the step size for directions with infrequent or smaller gradients.

Originally, AdaGrad was developed for convex optimization problems but has found application in the non-convex setting of deep learning, for example for large-scale recommendation systems [219]. However, the accumulated squared gradients are always positive which makes the denominator of the scaling continuously increase during the training. Empirical results show that this results in a too aggressive shrinking of the learning rate [101].

RMSProp

The **RMSProp** algorithm (abbreviation of *Root Mean Square Propagation*) proposed by Tieleman and Hinton [294][26] tries to fix the aggressive learning rate decay of AdaGrad by introducing an exponentially decaying average over the past squared gradients:

Update Rule 3.3.6 [RMSProp]

$$s^{(t)} = \rho s^{(t-1)} + (1-\rho) g_{\mathbb{B}^{(t)}} \odot g_{\mathbb{B}^{(t)}} \tag{3.27}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{s^{(t)}} + \varepsilon} \odot g_{\mathbb{B}^{(t)}} \,. \tag{3.28}$$

This newly introduced decay rate ρ is usually set to 0.9 or 0.99. An interesting variant of RMSProp was described by Graves [108] which combines RMSProp with momentum. The RMSProp optimizer has been applied to many deep learning problems, most notably vision tasks [*e.g.*, 131, 283, 288].

Adadelta

Adadelta [341] represents another method that was invented to correct the aggressive learning rate schedule of AdaGrad. It was independently developed from RMSProp and includes the same exponential moving average of the past squared gradients. Additionally, it aims to replace the need for a manually defined learning rate by using another exponential moving average, this time of the squared parameter updates:

Update Rule 3.3.7 [Adadelta]

$$s^{(t)} = \rho s^{(t-1)} + (1-\rho) g_{\mathbb{B}^{(t)}} \odot g_{\mathbb{B}^{(t)}} \tag{3.29}$$

$$d^{(t)} = \rho d^{(t-1)} + (1-\rho) \Delta_\theta^2 \tag{3.30}$$

$$\Delta_\theta = \frac{\sqrt{d + \varepsilon}}{\sqrt{s + \varepsilon}} \odot g_{\mathbb{B}^{(t)}} \tag{3.31}$$

$$\theta^{(t+1)} = \theta^{(t)} - \eta \Delta_\theta \,. \tag{3.32}$$

Here, we re-introduced the learning rate η that can be set to 1.0 to recapture the original formulation. Empirically, it turned out that Adadelta was not able to completely replace the need for tuning learning rates and many popular deep learning frameworks offer a learning rate for Adadelta as well.

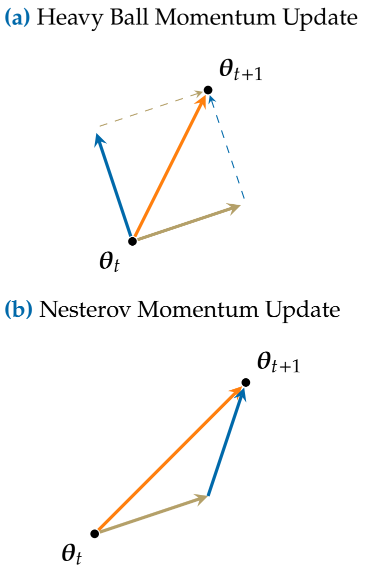


Figure 3.7: Illustration of the different momentum updates. (a) In the standard Momentum method, the gradient (→) is computed at the current parameter position (θ_t). The final update step (→) is a sum of this gradient and the velocity step (→). (b) NAG computes the gradient (→) *after* applying the velocity step (→). This results in a slightly different update step (→) compared to the classical Momentum since the gradients are evaluated at two different locations.

[221] Nesterov (1983), "A method for solving the convex programming problem with convergence rate $O(1/k^2)$"

22: *Nesterov accelerated gradient* is sometimes also called *Nesterov momentum*.

[9] Amodei et al. (2016), "Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin"
[115] He et al. (2017), "Mask R-CNN"
[117] He et al. (2016), "Deep Residual Learning for Image Recognition"
[239] Redmon et al. (2016), "You Only Look Once: Unified, Real-Time Object Detection"
[281] Sutskever et al. (2013), "On the importance of initialization and momentum in deep learning"
[284] Szegedy et al. (2015), "Going Deeper With Convolutions"

23: The term *adaptive gradient method* now usually subsumes all methods that use an element-wise learning rate, such as Adam or RMSProp.

Adam

The **Adam** optimizer (derived from *adaptive moment estimation*) uses estimates of the first and second moments of the gradients and can be viewed as an extension of RMSProp [166].

Update Rule 3.3.8 [Adam]

$$\boldsymbol{m}^{(t)} = \beta_1 \boldsymbol{m}^{(t-1)} + (1-\beta_1)\boldsymbol{g}_{\mathbb{B}^{(t)}} \tag{3.33}$$
$$\boldsymbol{v}^{(t)} = \beta_2 \boldsymbol{v}^{(t-1)} + (1-\beta_2)\boldsymbol{g}_{\mathbb{B}^{(t)}} \odot \boldsymbol{g}_{\mathbb{B}^{(t)}} \tag{3.34}$$
$$\hat{\boldsymbol{m}}^{(t)} = \frac{\boldsymbol{m}^{(t)}}{1-\beta_1^t} \tag{3.35}$$
$$\hat{\boldsymbol{v}}^{(t)} = \frac{\boldsymbol{v}^{(t)}}{1-\beta_2^t} \tag{3.36}$$
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}^{(t)}}+\varepsilon} \odot \hat{\boldsymbol{m}}^{(t)}\,. \tag{3.37}$$

Compared to the RMSProp optimizer, Adam includes momentum and a *bias correction* for the two exponential moving averages. Both the estimates of the first moment (the *momentum* term, Equation (3.33)) and of the (raw) second moment (see Equation (3.34)) are usually initialized by zero. This could result in a high bias early during training when this initialization still has a considerable effect compared to the observed gradients. Adam addresses this by adding bias corrections for both estimations (see Equations (3.35) and (3.36)).

Adam uses two separate exponential decay rates denoted β_1 and β_2 for both moving averages. Although this requires additional hyperparameters, Adam is often said to require less tuning, partly because there are well-working default values for all hyperparameters [*e.g.*, 101, 271]. Kingma and Ba [166] suggest 10^{-3} for the learning rate η, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 10^{-8}$. Nevertheless, for many practical applications tuning the learning is still helpful and tuning β_1, β_2, or even ε has shown to offer further performance gains [*e.g.*, 60].

The Adam optimizer remains one of the most popular choices for deep learning. It has found strong application for training GANs [*e.g.*, 158, 159, 233, 360] and in natural language modeling [*e.g.*, 42, 72, 307]. Adam is often said to generalize worse than SGD [*e.g.*, 163, 318], but it has not been proven or explained conclusively and the debate on the generalization performance of adaptive optimizers continues [*e.g.*, 5, 354].

24: An analogous view is to interpret these *adaptive methods* as changing the update direction or using a diagonal preconditioner.

[78] Duchi et al. (2011), "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization"

25: Recently, the ε-parameter has been re-interpreted as a tunable hyperparameter. See the discussion in Section 6.2.3. Note, that depending on the implementation, ε is sometimes included in the square root. This also extends to other optimization algorithms with ε-like parameters.

[219] Naumov et al. (2019), "Deep Learning Recommendation Model for Personalization and Recommendation Systems"

[101] Goodfellow et al. (2016), "Deep Learning"

[294] Tieleman et al. (2012), "Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude"

26: The RMSProp method was famously presented as part of a Coursera slide and is unpublished. Nevertheless, it is one of the most popular optimization methods for deep learning and the slides accumulated roughly 5.000 citations up until now.

[108] Graves (2013), "Generating Sequences With Recurrent Neural Networks"

[131] Howard et al. (2019), "Searching for MobileNetV3"

[283] Szegedy et al. (2017), "Inception-v4, Inception-ResNet and the impact of residual connections on learning."

[288] Tan et al. (2019), "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks"

[341] Zeiler (2012), "ADADELTA: An Adaptive Learning Rate Method"

[166] Kingma et al. (2015), "Adam: A Method for Stochastic Optimization"

[101] Goodfellow et al. (2016), "Deep Learning"

[271] Sivaprasad et al. (2020), "Optimizer Benchmarking Needs to Account for Hyperparameter Tuning"

NAdam

Building on Adam, Dozat [76] presented **Nadam** which introduces NAG into the Adam optimizer. We can extend the update rule of Adam (Equation (3.37)) by including Equations (3.33) and (3.35), resulting in

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}^{(t)}} + \varepsilon} \odot \left(\frac{\beta_1 \boldsymbol{m}^{(t-1)}}{1-\beta_1^t} + \frac{(1-\beta_1)\boldsymbol{g}_{\mathbb{B}^{(t)}}}{1-\beta_1^t} \right). \quad (3.38)$$

Intuitively, we can see, that we are effectively still using the "old" momentum term $\boldsymbol{m}^{(t-1)}$. Replacing it with the current momentum vector $\boldsymbol{m}^{(t)}$ lets us introduce the accelerated momentum of NAG into Adam, where the only change to its update rule is replacing Equation (3.37) by

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}^{(t)}} + \varepsilon} \odot \left(\beta_1 \hat{\boldsymbol{m}}^{(t)} + \frac{(1-\beta_1)\boldsymbol{g}_{\mathbb{B}^{(t)}}}{1-\beta_1^t} \right). \quad (3.39)$$

[166] Kingma et al. (2015), "Adam: A Method for Stochastic Optimization"
[60] Choi et al. (2019), "On Empirical Comparisons of Optimizers for Deep Learning"
[158] Karras et al. (2018), "Progressive Growing of GANs for Improved Quality, Stability, and Variation"
[159] Karras et al. (2020), "Analyzing and Improving the Image Quality of StyleGAN"
[233] Radford et al. (2016), "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks"
[360] Zhu et al. (2017), "Unpaired Image-To-Image Translation Using Cycle-Consistent Adversarial Networks"
[42] Brown et al. (2020), "Language Models are Few-Shot Learners"
[72] Devlin et al. (2019), "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"
[307] Vaswani et al. (2017), "Attention Is All You Need"
[163] Keskar et al. (2017), "Improving Generalization Performance by Switching from Adam to SGD"
[318] Wilson et al. (2017), "The Marginal Value of Adaptive Gradient Methods in Machine Learning"
[5] Agarwal et al. (2020), "Revisiting the Generalization of Adaptive Gradient Methods"
[354] Zhou et al. (2020), "Towards Theoretically Understanding Why SGD Generalizes Better Than Adam in Deep Learning"
[76] Dozat (2016), "Incorporating Nesterov Momentum into Adam"
[238] Reddi et al. (2018), "On the Convergence of Adam and Beyond"

AMSGrad

Reddi et al. [238] identified an issue with the convergence proof of Adam and constructed an example where Adam converges to a highly sub-optimal solution. The effective learning rate of Adam is given by $\eta/\sqrt{\hat{\boldsymbol{v}}^{(t)}} + \varepsilon$. As a result, if $\hat{\boldsymbol{v}}^{(t)}$ decreases, the learning rate increases, which can lead to poor convergence in certain settings. **AMSGrad** aims to fix this problem by not allowing $\boldsymbol{v}^{(t)}$ to decrease. Adding this operation, the full update rule for AMSGrad is given by:

Update Rule 3.3.9 [AMSGrad]

$$\boldsymbol{m}^{(t)} = \beta_1 \boldsymbol{m}^{(t-1)} + (1-\beta_1)\boldsymbol{g}_{\mathbb{B}^{(t)}} \quad (3.40)$$

$$\boldsymbol{v}^{(t)} = \beta_2 \boldsymbol{v}^{(t-1)} + (1-\beta_2)\boldsymbol{g}_{\mathbb{B}^{(t)}} \odot \boldsymbol{g}_{\mathbb{B}^{(t)}} \quad (3.41)$$

$$\hat{\boldsymbol{m}}^{(t)} = \frac{\boldsymbol{m}^{(t)}}{1-\beta_1^t} \quad (3.42)$$

$$\boldsymbol{v}^{(t)} = \max(\boldsymbol{v}^{(t-1)}, \boldsymbol{v}^{(t)}) \quad (3.43)$$

$$\hat{\boldsymbol{v}}^{(t)} = \frac{\boldsymbol{v}^{(t)}}{1-\beta_2^t} \quad (3.44)$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}^{(t)}} + \varepsilon} \odot \hat{\boldsymbol{m}}^{(t)}. \quad (3.45)$$

AdaBound and AMSBound

An often stated observation is that adaptive optimization methods, *e.g.* Adam, generally offer fast progress. However, non-adaptive methods such as SGD are said to reach a better final performance, especially when measured on unseen data [*e.g.*, 163, 318]. Luo et al. [196] presented **AdaBound** and **AMSBound** as variants of Adam and AMSGrad respectively. By applying bounds to the learning rates they could effectively smoothly transition from the adaptive methods to SGD. The proposed methods clip the effective learning rate — computed by either the Adam or AMSGrad update rule — by a lower and an upper bound function. These bounding functions are designed such that they start at zero and infinity, respectively, and smoothly converge to a constant final learning rate. For either Adam or AMSGrad we can replace the update rule to get AdaBound or AMSBound:

[163] Keskar et al. (2017), "Improving Generalization Performance by Switching from Adam to SGD"
[318] Wilson et al. (2017), "The Marginal Value of Adaptive Gradient Methods in Machine Learning"

[196] Luo et al. (2019), "Adaptive Gradient Methods with Dynamic Bound of Learning Rate"

Update Rule 3.3.10 [AdaBound and AMSBound]
Given the update rule of Adam (Update Rule 3.3.8) or AMSGrad (Update Rule 3.3.9), modifying the final equation by

$$\hat{\eta}^{(t)} = \operatorname{clip}\left({}^{\eta}/_{\sqrt{\hat{v}^{(t)}}+\varepsilon}, \eta_{\mathrm{l}}(t), \eta_{\mathrm{u}}(t)\right) \tag{3.46}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \hat{\eta}^{(t)} \odot \hat{\boldsymbol{m}}^{(t)}\,, \tag{3.47}$$

results in the update rule of AdaBound or AMSBound, respectively.

The function $\eta_{\mathrm{l}}(t)$ is non-decreasing and starts from zero at $t = 0$ and converges to some final learning rate η^* with convergence speed γ. Analogously, $\eta_{\mathrm{u}}(t)$ is a non-increasing upper bound starting from infinity and converging to the same final learning rate. Luo et al. [196] offer default bounding functions and state that this bounding is designed to make their method more robust to extreme learning rates.

[196] Luo et al. (2019), "Adaptive Gradient Methods with Dynamic Bound of Learning Rate"

Lookahead

The **Lookahead** optimizer [349] keeps two separate sets of parameters, called *fast* and *slow* weights. The update direction of the *slow* weights $\boldsymbol{\theta}^{(t)}$ is chosen by "looking ahead" a few optimization steps with the *fast* weights $\phi^{(t)}$.[27] The update step of the *fast* weights $\Delta\phi^{(t)}$ is determined by an arbitrary *inner* optimization method, *e.g.* by Momentum which we will denote LA(Mom.) or using RAdam to get LA(RAdam) also called Ranger [321]. Effectively Lookahead performs k steps of the inner update

[349] Zhang et al. (2019), "Lookahead Optimizer: k steps forward, 1 step back"

27: The symbols for the fast and slow weights are reversed compared to the original paper to be consistent with the rest of the chapter. Here, $\boldsymbol{\theta}$ denotes the parameters that are used, *e.g.* when performing inference.

[321] Wright (2020), "Ranger"

$$\phi^{(t)} = \phi^{(t)} + \Delta\phi^{(t)}\,, \tag{3.48}$$

with the update step $\Delta\phi^{(t)}$ determined by the inner optimizer.[28] Following k updates of the *fast* weights, Lookahead performs the outer update of the *slow* weights:

28: Naturally, Lookahead also inherits the hyperparameters of the inner optimizer. We will denote the learning rate of the *inner* optimizer with η_f to emphasize that it refers to the update of the *fast* weights.

Update Rule 3.3.11 [Lookahead]

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \eta(\phi^{(t)} - \boldsymbol{\theta}^{(t)}) \,. \tag{3.49}$$

After this update step in the direction where the *fast* weights have moved after k updates, the *fast* weights are reset to the current *slow* weights to be able to "explore" and "lookahead" for another k steps.

RAdam

Liu et al. [189] noticed that during the very early stages of training Adam's adaptive learning rate can exhibit a large variance caused by the small number of training samples processed so far. They show that adding a heuristically inspired learning rate warmup can help not only reduce this large variance but also lead to improved convergence behavior. They propose **RAdam** (derived from *rectified* Adam) a variation of Adam that aims to have a consistent variance.

[189] Liu et al. (2020), "On the Variance of the Adaptive Learning Rate and Beyond"

Update Rule 3.3.12 [RAdam]
RAdam adapts the update rule of Adam (Update Rule 3.3.8) by

$$\rho_\infty = {}^{2}/_{(1-\beta_2)} - 1 \tag{3.50}$$

$$\rho^{(t)} = \rho_\infty - 2t\beta_2^t/_{(1-\beta_2^t)} \tag{3.51}$$

$$w^{(t)} = \sqrt{\frac{(\rho^{(t)}-4)(\rho^{(t)}-2)\rho_\infty}{(\rho_\infty-4)(\rho_\infty-2)\rho^{(t)}}} \tag{3.52}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - w^{(t)}\frac{\eta}{\sqrt{\hat{\boldsymbol{v}}^{(t)}} + \varepsilon} \odot \hat{\boldsymbol{m}}^{(t)} \,, \tag{3.53}$$

where Equation (3.53) is only used if the variance of the adaptive learning rate is tractable. If $\rho^{(t)} \leq 4$, then the variance is intractable and RAdam falls back to regular momentum, replacing Equation (3.53) with

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta\hat{\boldsymbol{m}}^{(t)} \,. \tag{3.54}$$

Ma and Yarats [198] state that RAdam essentially performs four steps of SGD with Momentum, followed by Adam with a fixed warmup schedule. Their analysis suggests that using Adam with a linear learning rate warmup scheduled over $2(1-\beta_2)^{-1}$ iterations

[198] Ma et al. (2021), "On the Adequacy of Untuned Warmup for Adaptive Optimization"

is functionally equivalent to using RAdam for many practically relevant settings.

AdaBelief

Zhuang et al. [361] propose a slight modification to the Adam algorithm, where instead of estimating the *raw* second moment (see Equation (3.34)), they compute the second *central* moment, the variance. This variance describes how much the past gradients in each batch deviate from the current direction, *i.e.* viewing the momentum term $\boldsymbol{m}$ as the current prediction of the true gradient. This describes how much one can "belief" this estimator. The full update rule of the suggested method called **AdaBelief** can be written as

[361] Zhuang et al. (2020), "AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients"

Update Rule 3.3.13 [AdaBelief]

$$\boldsymbol{m}^{(t)} = \beta_1 \boldsymbol{m}^{(t-1)} + (1-\beta_1) \boldsymbol{g}_{\mathbb{B}^{(t)}} \tag{3.55}$$

$$\boldsymbol{s}^{(t)} = \beta_2 \boldsymbol{s}^{(t-1)} + (1-\beta_2) \left(\boldsymbol{g}_{\mathbb{B}^{(t)}} - \boldsymbol{m}^{(t)}\right)^{\odot 2} \tag{3.56}$$

$$\hat{\boldsymbol{m}}^{(t)} = \frac{\boldsymbol{m}^{(t)}}{1-\beta_1^t} \tag{3.57}$$

$$\hat{\boldsymbol{s}}^{(t)} = \frac{\boldsymbol{s}^{(t)}}{1-\beta_2^t} \tag{3.58}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\eta}{\sqrt{\hat{\boldsymbol{s}}^{(t)}} + \varepsilon} \odot \hat{\boldsymbol{m}}^{(t)}, \tag{3.59}$$

Observing large and consistent gradients would then result in a large momentum term and a small variance which would consequently result in a large adapted learning rate. Conversely, an observation of large, but opposing gradients would cause AdaBelief to reduce the step size. Since Adam does not take into account the sign of the gradients when computing the second *raw* momentum, it would treat both cases similarly, selecting a small learning rate in both cases.

Other

Two first-order methods that enjoy popularity in the regime of large batch sizes, are LARS and LAMB. You et al. [334] noticed that when using large batch sizes one also needs to use large learning rates and training can become unstable. In those cases the ratio between the norm of the parameter updates and the norm of the parameters is large. To alleviate this, they proposed **LARS** (short for *Layer-wise Adaptive Rate Scaling*) which uses a layer-wise learning rate $\lambda^{(l)}$ in addition to the regular global learning rate

[334] You et al. (2017), "Large Batch Training of Convolutional Networks"

η so that the parameter update is independent of the gradient magnitude. For a single layer l, the update rule of LARS can be written as[29]

29: To avoid overloading the notation, Equation (3.62) describes the parameter update only for a single layer, *i.e.* here $\boldsymbol{\theta}^{(t)}$ denotes only the parameters belonging to layer l.

Update Rule 3.3.14 [LAMB]

$$\lambda^{(l)} = \frac{\|\boldsymbol{\theta}^{(t)}\|}{\|\boldsymbol{g}_{\mathbb{B}^{(t)}}\|} \tag{3.60}$$

$$\boldsymbol{v}^{(t)} = \rho \boldsymbol{v}^{(t-1)} + \eta \lambda^{(l)} \boldsymbol{g}_{\mathbb{B}^{(t)}} \tag{3.61}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \boldsymbol{v}^{(t)} . \tag{3.62}$$

[335] You et al. (2020), "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes"

[215] Nado et al. (2021), "A Large Batch Optimizer Reality Check: Traditional, Generic Optimizers Suffice Across Batch Sizes"

LAMB (short for *Layer-wise Adaptive Moments*) [335] uses a very similar strategy of normalizing the parameter updates but applies it to Adam instead. Although LARS and LAMB are popular choices when working in the regime of large batch sizes, Nado et al. [215] showed that with careful tuning traditional optimization methods such as NAG or Adam can achieve the competitive results of LARS and LAMB even when operating with large batch sizes.

[194] Loshchilov et al. (2019), "Decoupled weight decay regularization"

Often the terms *weight decay* and L^2 *regularization* are used interchangeably (see Section 2.3.1). Loshchilov and Hutter [194] clarified that while, up to reparameterization of the regularization strength, this is the case for SGD, it is not correct for Adam. Adding weight decay to Adam results in a new optimization method called **AdamW** that explicitly does a weight decay step when updating the parameters

Update Rule 3.3.15 [AdamW]

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \left(\frac{1}{\sqrt{\hat{\boldsymbol{v}}^{(t)}} + \varepsilon} \odot \hat{\boldsymbol{m}}^{(t)} + \lambda \boldsymbol{\theta}^{(t)} \right) . \tag{3.63}$$

Here, compared to Adam, the regularization of strength λ enters via the parameter updates instead of modifying the gradient via $\boldsymbol{g}_{\mathbb{B}^{(t)}} = \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} \ell(f_{\boldsymbol{\theta}}) + \lambda \boldsymbol{\theta}$.

3.3.2 Second-order Methods

The benefits of using second-order derivative information in the form of the Hessian are obvious. Using additional curvature information can more accurately describe the (local) loss landscape, making it easier to navigate it efficiently. Turning back to the example that motivated Momentum (Figure 3.6), we can see that SGD's optimization issues arose from the drastic variation in curvature for different directions. In this case, a method with access to Hessian information could predict that the direction of steepest descent does not provide the most promising update direction. Because of the difference in curvature, which is apparent in the Hessian, the update direction could instead be shifted more towards the direction of low curvature and thus toward the minimum.

An additional advantage of second-order methods, that goes beyond the more effective navigation in ill-conditioned problems, is their possibility of reducing the need for hyperparameter selection. This can be motivated by taking the second-order Taylor series expansion of the loss L near $\bar{\boldsymbol{\theta}}$

$$L(\boldsymbol{\theta}) \approx L(\bar{\boldsymbol{\theta}}) + \left(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}}\right)^\top \boldsymbol{g} + \frac{1}{2}\left(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}}\right)^\top \boldsymbol{H}\left(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}}\right) , \tag{3.64}$$

where $\boldsymbol{g}$ and $\boldsymbol{H}$ are the gradient and Hessian of L at the point $\bar{\boldsymbol{\theta}}$. Selecting $\boldsymbol{\theta}$ to minimize Equation (3.64) results in an update rule known as Newton's method:

Update Rule 3.3.16 [Newton's method]

$$\boldsymbol{\theta} = \bar{\boldsymbol{\theta}} - \boldsymbol{H}^{-1}\boldsymbol{g} . \tag{3.65}$$

If the loss function L is a deterministic positive definite quadratic function, Newton's method will find the exact minimum in a single step.[30] If L is not quadratic, we can still use Newton's method by iteratively applying Equation (3.65) as long as the Hessian remains positive definite. However, additional steps and hyperparameters are needed to account for the non-convexity of the loss landscape.

30: Newton's method is a special case of the *Newton-Raphson method* for finding roots of real-valued functions. Applying the *Newton-Raphson method* to the gradient of a function results in Newton's method described here. Although both the *Newton-Raphson method* and Newton's method are often attributed and indeed named after Sir Isaac Newton, it is generally agreed upon that closely related techniques were invented much earlier and most likely multiple times independently.

Comparing Equations (3.20) and (3.65) we can see that the inverse Hessian $\boldsymbol{H}^{-1}$ replaces the learning rate η. Looking at it geometrically, we can see that at each iteration Newton's method fits a parabola to the loss $L(\boldsymbol{\theta})$ and then steps to this parabola's minimum. Using only the gradient, we are effectively using a first-order approximation of the loss, which does not have a minimum and we instead have to manually select a step size.[31]

31: Alternatively, you can view first-order methods as estimating the Hessian via $\boldsymbol{H} \approx {}^1\!/\!_\eta \boldsymbol{I}$.

Despite these clear advantages, second-order methods face many practical challenges when it comes to deep learning. For a neural network with D parameters, the Hessian matrix of the loss with respect to the network's parameters consists of $D \times D$ elements. For modern models which can easily have millions of parameters, it is infeasible to represent the entire Hessian, much less compute its inverse at the costs of $\mathcal{O}(D^3)$. Although using curvature information can lead to requiring fewer iterations to reach an acceptable solution, *each iteration* is more expensive to compute.

Luckily, we do not need to explicitly form the Hessian, since we only ever use matrix-vector products of its inverse with a vector (see Equation (3.65)). Instead of computing the parameter update $\Delta\boldsymbol{\theta}$ via this matrix-vector product, we can instead solve the linear system $\boldsymbol{H}\Delta\boldsymbol{\theta} = -\boldsymbol{g}$, *e.g.* using the CONJUGATE GRADIENT method (CG). Solving this system using CG only requires matrix-vector products between the Hessian matrix $\boldsymbol{H}$ and an arbitrary vector $\boldsymbol{v}$ and is therefore known as ***Hessian-free*** optimization [*e.g.*, 202, 224].[32] For neural networks, we can leverage the automatic differentiation functionality to efficiently compute $\boldsymbol{H}\boldsymbol{v}$ at the cost of only a single additional forward and backward pass [229, 264].

[202] Martens (2010), "Deep learning via Hessian-free optimization."
[224] Nocedal et al. (2006), "Numerical Optimization"

32: In the worst case, CG will converge after D iterations, thus requiring D matrix-vector products of the Hessian. In practice, it is more efficient to only use a few iterations of CG in each training iteration and use this approximate solution instead of waiting for convergence of the CG method. Since this inner solver is *truncated* this method is known as *truncated-Newton*.

[229] Pearlmutter (1994), "Fast Exact Multiplication by the Hessian"
[264] Schraudolph (2002), "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent"

Instead of using the exact Hessian in Equation (3.65), ***quasi-Newton*** methods use approximations thereof to reduce the computational cost per iteration. These approximations can, for example, be in the form of (block-)diagonal, or low-rank approximations which allow efficient inversion or be based solely on first-order information.[33] One approximation comes in the form of the *generalized Gauss-Newton* matrix.

33: Adaptive gradient methods such as ADAGRAD can be seen as performing a diagonal approximation of the Hessian. For example, $\eta/\sqrt{s}+\varepsilon$ in Equation (3.26) takes the role of the diagonal of the Hessian approximation.

Generalized Gauss-Newton

Remember that the Hessian of the loss is given by[34]

$$\nabla^2_{\boldsymbol{\theta}} L_{\mathbb{D}}(f_{\boldsymbol{\theta}}) = \frac{1}{|\mathbb{D}|} \sum_{i=1}^{|\mathbb{D}|} \nabla^2_{\boldsymbol{\theta}} \ell(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), y^{(i)}) . \tag{3.66}$$

34: Here, we use $\mathbb{D}$ for a generic data set as it is true for both the empirical training data set $\mathbb{D} = \mathbb{D}_{\text{train}}$ as well as a single batch $\mathbb{D} = \mathbb{B}$.

This requires the computation of the Hessian of a composition of two functions, the model function f with $f : \mathbb{R}^D \to \mathbb{R}^M$ and the loss function ℓ with $\ell : \mathbb{R}^M \to \mathbb{R}$. Here, M is given by the dimensionality of the network's output that is subsequently processed by the loss function. In image classification, for example, the model function would output the (log) probability prediction for each class thus M would be the number of classes.

We can re-write the Hessian matrix using the chain rule twice and the product rule once into:[35]

35: To simplify the notation, we ignored the dependency of the loss function on y and of the model on $\boldsymbol{x}$ as they are irrelevant for constructing the Hessian with respect to the parameters $\boldsymbol{\theta}$.

$$\begin{aligned}
&\left[\nabla^2_{\boldsymbol{\theta}} \ell(f(\boldsymbol{\theta}))\right]_{i,j} \\
&= \frac{\partial}{\partial\theta_j}\frac{\partial}{\partial\theta_i}\ell(f(\boldsymbol{\theta})) = \sum_{m=1}^{M} \frac{\partial}{\partial\theta_j}\left(\frac{\partial \ell(f(\boldsymbol{\theta}))}{\partial f_m}\frac{\partial f_m}{\partial\theta_i}\right) \\
&= \sum_{m=1}^{M} \frac{\partial}{\partial\theta_j}\left(\frac{\partial \ell(f(\boldsymbol{\theta}))}{\partial f_m}\right)\frac{\partial f_m}{\partial\theta_i} + \sum_{m=1}^{M}\frac{\partial \ell(f(\boldsymbol{\theta}))}{\partial f_m}\frac{\partial^2 f_m}{\partial\boldsymbol{\theta}_j\,\partial\boldsymbol{\theta}_i} \\
&= \sum_{m=1}^{M}\left(\sum_{n=1}^{M}\frac{\partial^2 \ell(f(\boldsymbol{\theta}))}{\partial f_m\,\partial f_n}\frac{\partial f_n}{\partial\theta_j}\right)\frac{\partial f_m}{\partial\theta_i} + \sum_{m=1}^{M}\frac{\partial \ell(f(\boldsymbol{\theta}))}{\partial f_m}\frac{\partial^2 f_m}{\partial\boldsymbol{\theta}_j\,\partial\boldsymbol{\theta}_i} \\
&= \sum_{n=1}^{M}\frac{\partial f_n}{\partial\theta_j}\sum_{m=1}^{M}\frac{\partial f_m}{\partial\theta_i}\frac{\partial^2 \ell(f(\boldsymbol{\theta}))}{\partial f_m\,\partial f_n} + \sum_{m=1}^{M}\frac{\partial \ell(f(\boldsymbol{\theta}))}{\partial f_m}\frac{\partial^2 f_m}{\partial\boldsymbol{\theta}_j\,\partial\boldsymbol{\theta}_i} .
\end{aligned} \tag{3.67}$$

Written in matrix notation

$$\nabla^2_{\boldsymbol{\theta}} \ell(f(\boldsymbol{\theta})) = \underbrace{\boldsymbol{J}^{f\top}_{\boldsymbol{\theta}} \boldsymbol{H}^{\ell}_{f} \boldsymbol{J}^{f}_{\boldsymbol{\theta}}}_{:=\boldsymbol{G}} + \sum_{m=1}^{M} \left[\nabla_f \ell\right]_m \nabla^2_{\boldsymbol{\theta}} f_m \,, \tag{3.68}$$

where $\boldsymbol{J}^{f\top}_{\boldsymbol{\theta}}$ is the $M \times D$ Jacobian of f and $\boldsymbol{H}^{\ell}_{f}$ the $M \times M$ Hessian of the loss with respect to f. The ***generalized Gauss-Newton*** (GGN) [*e.g.*, 264] matrix $\boldsymbol{G}$ is defined to be the first term of this expression and provides an approximation of the Hessian. If the second term vanishes then the GGN matrix is identical to the Hessian. Crucially, the GGN matrix only models the Hessian of the (outer) loss function ℓ but ignores the curvature introduced by f. Note, the "split" between the inner function f and the outer function ℓ is ambiguous and there exist multiple different approximations that can be called GGN that incorporate different amounts of the overall curvature depending on the chosen split.

[264] Schraudolph (2002), "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent"

Second-order Methods for Deep Learning

Several notable second-order methods have been developed for large-scale machine learning and in particular deep learning, including **L-BFGS** [184], **K-BFGS** [100], **K-FAC** [203], **KFRA** [35], **Shampoo** [10, 110], or **ADAHESSIAN** [333]. Nevertheless, first-order methods still dominate in most deep learning applications. But this may also be due to the fact that the widely-used software libraries, the available hardware, and even the models themselves have been designed with these first-order methods in mind and co-evolved with them. Naturally, this turns the adoption of second-order methods into an uphill battle.

[184] Liu et al. (1989), "On the Limited Memory BFGS Method for Large Scale Optimization"

[100] Goldfarb et al. (2020), "Practical Quasi-Newton Methods for Training Deep Neural Networks"

[203] Martens et al. (2015), "Optimizing Neural Networks with Kronecker-Factored Approximate Curvature"

[35] Botev et al. (2017), "Practical Gauss-Newton Optimisation for Deep Learning"

[10] Anil et al. (2020), "Second Order Optimization Made Practical"

[110] Gupta et al. (2018), "Shampoo: Preconditioned Stochastic Tensor Optimization"

[333] Yao et al. (2020), "ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning"

3.4 Hyperparameter Tuning

In machine learning, we are often confronted with **hyperparameters**. *Parameters* and *hyperparameters* differ in that the former can be derived during training, *e.g.* via gradient-based optimization methods. Hyperparameters, on the other hand, describe the model family or control the training process itself and cannot directly be inferred during training. These hyperparameters are usually classified as *model hyperparameters* or *Training hyperparameters*[36]. Model parameters influence the model architecture, *e.g.* the number of layers in a neural net, the number of neurons in a layer, or the type of activation functions used. Training hyperparameters include the learning rate, batch size, or other hyperparameters exposed by the selected optimization algorithm but even the training algorithm itself can be considered a hyperparameter. In this work, we focus

36: *Training hyperparameters* are sometimes called *algorithm hyperparameters* or *optimization hyperparameters* on the study of optimization methods for training deep neural networks and will, therefore, focus in this section mostly on the training hyperparameters.

Regular model parameters are set directly by gradient-based learning methods during training, which is impossible or impractical to do for hyperparameters. We can, however, compute the loss that was achieved by a specific hyperparameter, such as a particular choice of batch size.[37] Searching for the hyperparameter setting that provides the best performance on a validation set is known as **hyperparameter tuning** and we describe examples of tuning methods in the next section. In practice, hyperparameters are usually set by a combination of tuning and expertise.

37: Of course, we can also use other performance metrics besides the loss to evaluate our hyperparameter choice. The advantage is that the metrics used for model selection or hyperparameter tuning need not be differentiable. Therefore, metrics such as accuracy or FID can be used that may more accurately describe what we consider a "better" model.

The choice of hyperparameter can drastically affect the final model performance. As a result, there exist many heuristics suggesting how to set certain hyperparameters [*e.g.*, 25, 36, 273]. The learning rate, for example, is often suggested to be set as large as possible without resulting in the training diverging. Similarly, the batch size is often set to fully utilize the GPU memory. There exist many more and oftentimes much more intricate heuristics, such as adding a learning rate warm-up to the training process (Section 3.4.3) or different optimization methods that supposedly perform better in certain areas (see Section 6.3). Further complicating the hyperparameter choice is the fact that the optimal value of one hyperparameter often depends on the selected value of another hyperparameter. The optimal learning rate, for example, varies for different choices of batch size. When combining all of these properties, it is no wonder that machine learning is often considered more of an art than a science.

[25] Bengio (2012), "Practical recommendations for gradient-based training of deep architectures"
[36] Bottou (2012), "Stochastic gradient descent tricks"
[273] Smith (2018), "A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay"

3.4.1 Tuning Methods

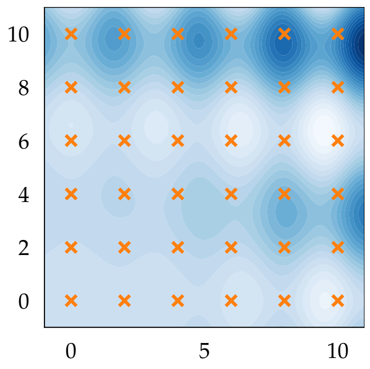


Figure 3.8: Illustration of grid search. For each hyperparameter, 6 different values are tested in the regularly spaced interval $[0, 10]$ for a total of 36 samples (✖). The true objective function is shown by the contour plot in the background.

Hyperparameter tuning aims to find the specific choice of hyperparameters that results in the best performance, as measured by some performance metric. In this section, we will only focus on three common hyperparameter tuning approaches, grid search, random search, and Bayesian optimization.

- **Grid search (Figure 3.8):** A straightforward approach to hyperparameter tuning is to try out a large number of possible values across a grid.
  This method requires a user-defined search space, *e.g.* searching for learning rates in the interval $[10^{-6}, 10^{1}]$ on a logarithmic grid or trying batch sizes on the grid points $\{16, 32, 64, 128, 256, 512\}$. Grid search[38] then performs an exhaustive search, trying out all possible combinations and ranking them via the pre-defined performance measure,

38: Grid search is sometimes also called *parameter sweep*.

e.g. the accuracy on the validation set. The advantages of grid search are that it is easily parallelizable as the runs are independent. If enough computational resources are available so that all runs can be run in parallel, grid search provides an exhaustive search of the search space with a waiting time equal to the training time of a single run. Grid search scales exponentially with the search space's dimensions and can be inefficient since the set of suggested hyperparameter values is independent of the problem and the resulting performance landscape with respect to the hyperparameters.

- **Random search (Figure 3.9):** Random search replaces the grid points from grid search with randomly selected points. Random search shares most of its advantages and disadvantages with grid search. However, it can outperform grid search in cases where only a small number of the tuned hyperparameters significantly affect the final performance. In this case, random search offers better coverage of the relevant dimensions [27].
- **Bayesian optimization (Figure 3.10):** In contrast to the other methods, Bayesian optimization attempts to build a probabilistic model of the performance landscape with respect to the hyperparameters.
  It iteratively evaluates promising hyperparameter settings and updates the current model based on the newly observed results. Contrary to grid or random search, Bayesian optimization adapts to the given problem and is generally considered to require fewer evaluations to find well-performing settings, due to its ability to reason about promising hyperparameter candidates. Bayesian optimization is inherently less parallel as it requires the results of previous experiments to update the current model. It also often introduces new hyperparameters such as the acquisition function[39], the prior[40], the kernel of the modeling function[41], etc. Examples of contemporary hyperparameter tuning approaches using Bayesian optimization are SMAC [138], TPE [26], SPEARMINT [276], and BOHB [83].

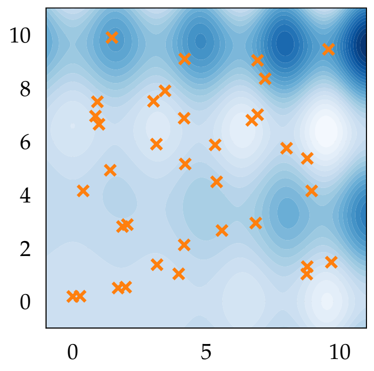


Figure 3.9: Illustration of random search. In total, 36 random samples (✖) are drawn uniformly. Compared to grid search, more different values for each hyperparameter are tested.

[27] Bergstra et al. (2012), "Random Search for Hyper-Parameter Optimization"

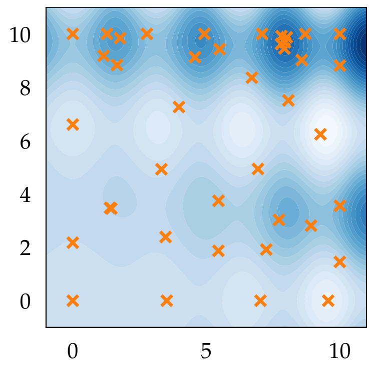


Figure 3.10: Illustration of Bayesian optimization. Based on the previous observations, Bayesian optimization methods adapt to the objective function. It can be observed that this tuning method exploits the local maximum in the top right (darker blue) more than the two previous methods. Shown are 36 samples (✖) that were computed sequentially.

3.4.2 Self-Tuned Methods

An alternative, to setting the hyperparameters via *external* tuning methods, is to use *internal* self-tuning methods. Instead of effectively trying out several hyperparameter settings in a trial-and-error fashion, self-tuned methods aim to find an appropriate hyperparameter setting automatically during training. A prime example of such methods are *line-search* approaches for setting the learning rate of gradient-based training methods. Instead of trying

39: The acquisition function formalizes the trade-off between exploration (trying out hyperparameter settings with large uncertainty) and exploitation (suggesting hyperparameters close to areas that have proven to work well).

40: The prior defines the initial (uninformed) modeling function.

41: Loosely speaking, the kernel describes the family of modeling functions.

out multiple constant learning rates and running them for an entire training run, line-search approaches, test multiple learning rates *in each update iteration* of the optimization algorithm. Line-search approaches usually come with an acceptance condition that needs to be fulfilled for an update step to be accepted, such as a need to reduce the loss compared to the initial point of the iteration.

[138] Hutter et al. (2011), "Sequential Model-Based Optimization for General Algorithm Configuration"

[26] Bergstra et al. (2011), "Algorithms for Hyper-Parameter Optimization"

[276] Snoek et al. (2012), "Practical Bayesian optimization of machine learning algorithms"

[83] Falkner et al. (2018), "BOHB: Robust and Efficient Hyperparameter Optimization at Scale"

While self-tuned methods remove or at least reduce the need for external hyperparameter tuning, they usually come with an increased cost of a single training run. Especially in deep learning, self-tuned methods often result in inferior performances compared to extensively tuned models, albeit at a much smaller total cost. Several self-tuned methods have been suggested for deep learning, to among other things adaptively set the learning rate [*e.g.*, 150, 200, 214, 243, 259, 308], the batch size [*e.g.*, 21, 43, 67], or the momentum parameter [*e.g.*, 172, 346].

[150] Jin et al. (2021), "AutoLRS: Automatic Learning-Rate Schedule by Bayesian Optimization on the Fly"

[200] Mahsereci et al. (2017), "Probabilistic Line Searches for Stochastic Optimization"

[214] Mutschler et al. (2020), "Parabolic Approximation Line Search for DNNs"

[243] Rolínek et al. (2018), "L4: Practical loss-based stepsize adaptation for deep learning"

[259] Schaul et al. (2013), "No more pesky learning rates"

[308] Vaswani et al. (2019), "Painless Stochastic Gradient: Interpolation, Line-Search, and Convergence Rates"

[21] Balles et al. (2017), "Coupling Adaptive Batch Sizes with Learning Rates"

[43] Byrd et al. (2012), "Sample Size Selection in Optimization Methods for Machine Learning"

[67] De et al. (2017), "Automated Inference with Adaptive Batches"

[172] Lancewicki et al. (2019), "Automatic and Simultaneous Adjustment of Learning Rate and Momentum for Stochastic Gradient Descent"

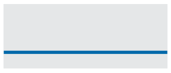
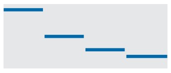
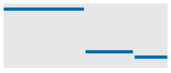
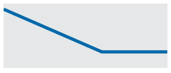
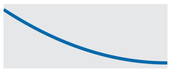
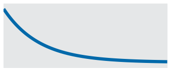
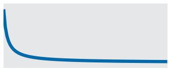
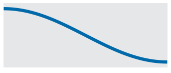
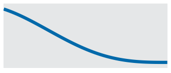
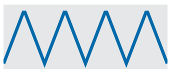
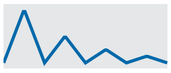
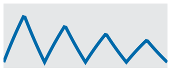
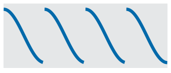
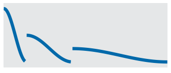
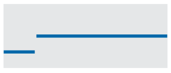
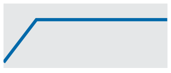
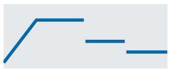
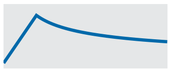
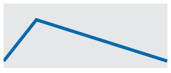
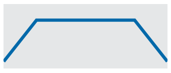
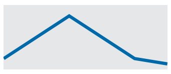
[346] Zhang et al. (2019), "YellowFin and the Art of Momentum Tuning"

3.4.3 Learning Rate Schedules

Besides tuning constant hyperparameters, it can also be beneficial to *schedule* them so that their value changes during the training process. Currently, this is most commonly done for the learning rate, which is often decayed following some **learning rate schedule**. The idea is that if the learning rate remains constant, one might end up bouncing around the minimum at the end of the optimization process in a state of diffusion, instead of reaching the optimum. Decreasing the learning rate in the end, empirically tends to improve the final performance. Table 3.2 provides an overview of commonly used parameter schedules.

Scheduled hyperparameters are one of the more extreme cases of decisions that deep learning practitioners have to deal with when training neural networks. The previous sections presented numerous optimization algorithms, each with their own set of hyperparameters, which can be tuned using different tuning methods. It is not meant as an exhaustive list of all possible options but showcases the agony of choice that one faces in practice in deep learning. Subsequent chapters use thorough benchmarking experiments (Chapters 5 and 6) and specialized debugging tools (Chapter 7) to investigate which of these choices are critical to ensure successful model training, how these choices affect the training process and thereby attempt to bring some order to this ever-growing list of methods.

Table 3.2: Overview of commonly used parameter schedules. Note, while we list the schedules parameters, it isn't clearly defined what aspects of a schedule are (tunable) parameters and what is a-priori fixed. In this column, η_0 denotes the initial learning rate, η_{lo} and η_{up} the lower and upper bound, Δt indicates an epoch count at which to switch decay styles, k denotes a decaying factor.

Name		Ref.	Illustration	Parameters
Constant				η_0
Step Decay	constant factor			$\eta_0, \Delta t_1, \dots, k$
	multi-step			$\eta_0, \Delta t_1, \dots, k_1, \dots$
Smooth Decay	linear decay	[*e.g.*, 101]		$\eta_0, (\Delta t, \eta_{\text{lo}})$
	polynomial decay			$\eta_0, k, (\eta_{\text{lo}})$
	exponential decay			$\eta_0, k, (\eta_{\text{lo}})$
	inverse time decay	[*e.g.*, 36]		$\eta_0, k, (\eta_{\text{lo}})$
	cosine decay	[193]		$\eta_0, (\eta_{\text{lo}})$
	linear cosine decay	[24]		$\eta_0, (\eta_{\text{lo}})$
Cyclical	triangular	[272]		$\eta_{\text{lo}}, \eta_{\text{up}}, \Delta t$
	triangular + decay	[272]		$\eta_{\text{lo}}, \eta_{\text{up}}, \Delta t, k$
	triangular + exponential decay	[272]		$\eta_{\text{lo}}, \eta_{\text{up}}, \Delta t$
	cosine + warm restarts	[193]		$\eta_{\text{up}}, \Delta t, (\eta_{\text{lo}})$
	cosine + warm restarts + decay	[193]		$\eta_{\text{up}}, \Delta t, k, (\eta_{\text{lo}})$
Warmup	constant warmup	[*e.g.*, 117]		$\eta_{\text{lo}}, \eta_0, \Delta t$
	gradual warmup	[105]		$\eta_0, \Delta t, (\eta_{\text{lo}})$
	gradual warmup + multi-step decay	[105]		$\eta_0, \Delta t, \Delta t_{\text{steps}}, k_1, \dots, (\eta_{\text{lo}})$
	gradual warmup + step number decay	[307]		$\eta_0, \Delta t, (\eta_{\text{lo}})$
	slanted triangular	[132]		$\eta_0, \Delta t, (\eta_{\text{lo}})$
	long trapezoid	[330]		$\eta_0, \Delta t_{\text{up}}, \Delta t_{\text{down}}, (\eta_{\text{lo}})$
Super-Convergence	1cycle	[274]		$\eta_{\text{up}}, \Delta t, \Delta t_{\text{cutoff}}, (\eta_{\text{lo}})$

Deep Learning 4

Artificial neural networks (ANNs)[1] are the machine learning model used in deep learning and are inspired by biological neurons, *e.g.* the human brain [118, 204, 245]. In recent years, neural networks have shown to be a particularly successful class of models which allowed unprecedented accomplishments in a number of fields [*e.g.*, 154, 170, 268]. Thanks to the availability of large amounts of data, increasing computing powers especially via GPUs, and algorithmic improvements, neural networks are state of the art for tasks such as speech recognition, image classification, or machine translation.

1: Artificial neural networks are often abbreviated simply as *neural networks* (NNs) or just *neural nets*.

In this chapter, we take a look at what neural networks are (Section 4.1), which typical layer types are used (Section 4.2), and how these layers can be combined to form common neural network architectures for different machine learning tasks (Section 4.3).

[118] Hebb (1949), "Organization of Behavior"
[204] McCulloch et al. (1943), "A logical calculus of the ideas immanent in nervous activity"
[245] Rosenblatt (1958), "The perceptron: a probabilistic model for information storage and organization in the brain."

[154] Jumper et al. (2021), "Highly accurate protein structure prediction with AlphaFold"
[170] Krizhevsky et al. (2012), "ImageNet Classification with Deep Convolutional Neural Networks"
[268] Silver et al. (2016), "Mastering the game of Go with deep neural networks and tree search"

4.1 Artificial Neural Networks

Artificial neural networks are a family of machine learning models mapping from an input space $\mathbb{X} \subseteq \mathbb{R}^I$ non-linearly to an output space $\mathbb{Y} \subseteq \mathbb{R}^C$. A neural network consists of numerous **artificial neurons** that usually take multiple incoming signals and produce a single output signal. The output signal might be broadcast to multiple other neurons. Typically, an artificial neurons computes a weighted sum of its incoming signal which is then passed through a non-linear activation function. Mathematically, a single neuron can be expressed as

$$z^{(\text{out})} = \phi(\underbrace{\boldsymbol{\theta}^\top \boldsymbol{z}^{(\text{in})} + b}_{:=\hat{z}^{(\text{out})}}) \,, \tag{4.1}$$

where $\boldsymbol{z}^{(\text{in})} \in \mathbb{R}^{n_{\text{in}}}$ is the vector of all incoming signals to the neuron, $\boldsymbol{\theta} \in \mathbb{R}^{n_{\text{in}}}$ are the **weights** of each input connection, $b \in \mathbb{R}$ is a **bias**[2], and $\phi : \mathbb{R} \to \mathbb{R}$ is the **activation function**. With $\hat{z}^{(\text{out})}$ we denote the result of the neuron prior to computing the activation, the so-called **pre-activation**.

2: To lighten the notation, we will sometimes drop the bias as it can be handily integrated into the weights, by adding an additional incoming signal with a fixed value of 1.

The modeling power of neural networks arises from combining many of those neurons in an acyclical **computational graph**. It is common to not count the number of neurons, *i.e.* the nodes of the graph, but the number of parameters, *i.e.* the edges of the graph[3]. Modern neural networks frequently have millions of

3: This equality between the number of *parameters* and the number of *edges* in the computation graph is not always true. Convolutional neural networks, for example, have a higher number of *effective* connections due to parameter sharing.

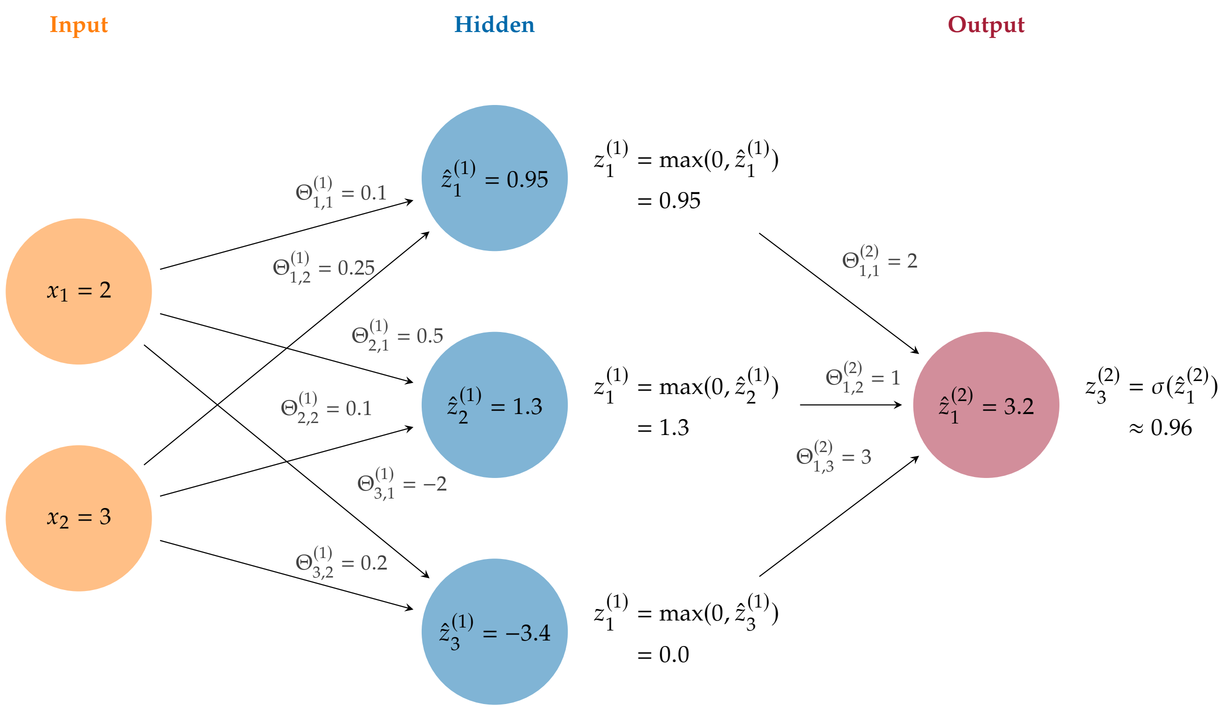

Input
Hidden
Output
$x_1 = 2$
$x_2 = 3$
$\Theta^{(1)}_{1,1} = 0.1$
$\Theta^{(1)}_{1,2} = 0.25$
$\Theta^{(1)}_{2,1} = 0.5$
$\Theta^{(1)}_{2,2} = 0.1$
$\Theta^{(1)}_{3,1} = -2$
$\Theta^{(1)}_{3,2} = 0.2$
$\hat{z}^{(1)}_1 = 0.95$
$\hat{z}^{(1)}_2 = 1.3$
$\hat{z}^{(1)}_3 = -3.4$
$z^{(1)}_1 = \max(0, \hat{z}^{(1)}_1) = 0.95$
$z^{(1)}_1 = \max(0, \hat{z}^{(1)}_2) = 1.3$
$z^{(1)}_1 = \max(0, \hat{z}^{(1)}_3) = 0.0$
$\Theta^{(2)}_{1,1} = 2$
$\Theta^{(2)}_{1,2} = 1$
$\Theta^{(2)}_{1,3} = 3$
$\hat{z}^{(2)}_1 = 3.2$
$z^{(2)}_3 = \sigma(\hat{z}^{(2)}_1) \approx 0.96$

Figure 4.1: Illustration of a very simple neural network with nine parameters. Information "flows" from the input nodes (●) via the hidden nodes (●) to the output nodes (●). The inputs $x = \begin{pmatrix} 2 & 3 \end{pmatrix}^\top$ could, for instance, be the (vectorized) pixel values of an image which should be classified. In this architecture, the input values are broadcast to all neurons in the hidden layer, which compute a weighted sum of all incoming signals. The current weights are denoted along the connection ($\rightarrow$), *e.g.* $\Theta^{(1)}_{3,2} = 0.2$ represents that the third neuron in the hidden layer, weights the second input by 0.2. The result of the weighted sum, the pre-activation, is denoted in the node. Crucially, a non-linear activation function is applied, which for the hidden layer in this example is simply the ReLU function, *i.e.* $\max(0, \hat{z})$, where $\hat{z}$ is the pre-activation. The result after the activation is written to the right of the neuron. A single output neuron weights the outputs of the hidden neurons and applies another activation function, this time a sigmoid. The output of the network for this example is 0.96 which could, for instance, represent the probability that the given image contains a hot dog.

parameters and larger models can have billions or even trillions of parameters [*e.g.*, 84, 182].

[84] Fedus et al. (2021), "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity"
[182] Lin et al. (2021), "M6-10T: A Sharing-Delinking Paradigm for Efficient Multi-Trillion Parameter Pretraining"

Figure 4.1 shows a simplified example of a neural network and how in such a computational graph information "flows" first from the inputs to the neurons, then from neurons to other neurons, and finally from neurons to the output. Such a computation of the output values of the network, given input data, is called a **forward pass**. The name *forward* pass stands in contrast to the *backwards* pass that computes the gradient and will be discussed in the next section.

In a typical neural networks structure, the neurons are not arranged randomly in the computational graph but are organized in **layers**. The term *layer* applies to an entire collection of neurons that perform transformations on the same inputs. The overall network signal then "travels" from layer to layer, before it reaches the output. A central reasons for arranging the neurons in layers is that this allows efficient computation of the forward (and backward) pass.

Taking the example illustrated in Figure 4.1, we can see that all neurons in the hidden layer can be computed in parallel since they are independent of each other. Afterward, the output layer can be computed once the hidden layer is done.

Similarly to stacking the neurons into layers, contemporary neuronal networks stack multiple layers to form a **deep** neural network. Deeper layers[4] can extract progressively higher-level information or features from the input [*e.g.*, 175]. To illustrate this, we can think of logistic regression as a special case of a single-layer neural network. Instead of performing the logistic regression on the original raw input, we could extend this single-layer neural network to multiple layers. Effectively, this means that we perform logistic regression (performed by the last layer of our neural network) not on the original raw input but instead on the *learned* output of the previous neural network layers. This additional computation done by these additional layers can transform the original data and build representations that allow a logistic regressor to easily distinguish the classes. One way to think about this is that the shallower layers might extract lower-level information such as edges. Deeper layers can then use these features to detect shapes or faces.[5] The lower levels perform a form of *feature engineering* which previously was mostly done manually.

4: The common convention is that *deeper* layers are further from the input, *i.e.* closer to the output. Analogously, *shallower* layers are close to the input. This informal description follows the idea that when neural network got progressively deeper, these additional layers got "stacked on top" of the existing lower-level layers.

[175] LeCun et al. (2015), "Deep learning"

5: Geirhos et al. [93] showed that neural networks most likely learn *textures* not *shapes*.

[93] Geirhos et al. (2019), "ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness."

The representational power of neural networks is therefore a result of both its width, *i.e.* combining neurons into distinct layers, and its depth, *i.e.* combining multiple layers into deep networks. Increasing either the width or the depth also increases the capacity of the network and such also the network's capability to overfit. Empirically though, bigger neural networks have shown to be able to achieve superior performance, even if they have the ability to memorize the entire training data set [343]. It remains unclear why these neural networks *learn* meaningful patterns and *generalize* instead of simply *memorizing* the data set. Parts of it might have to do with the used optimization methods which bias the model to solutions that *learn* rather than *memorize* (see Chapter 3), the regularization techniques (see Section 2.3), or the bias introduced by the structure of the model.

[343] Zhang et al. (2017), "Understanding deep learning requires rethinking generalization"

The type of computation described by Equation (4.1) is just one of many possible computations that can be added to the computational graph of a neural network. Section 4.2 provides a list of layer types common in deep learning. Various layer types, such as convolutional layers, fully connected layers, or dropout layers can be combined as long as they are differentiable and allow efficient computation of the gradient (see next section). Section 4.3 will provide examples for how these layer types are combined and stacked to create popular neural network architectures.

4.1.1 Backpropagation: Computing the Gradients in a Neural Network

As described in Section 3.2.1, we want to use efficient numerical optimization algorithms to set the parameters of our model. Ideally, they should describe the relationship in the data as best as possible, *i.e.* they should achieve a low loss. To use these efficient first-order optimization methods, we require access to the gradient of the loss with respect to all the network's parameters. In a computational graph such as a neural network, the gradient can be efficiently computed by an algorithm called **backpropagation** or simply *backprop*.[6]

6: The attribution of the backpropagation algorithm is non-trivial. Effectively, it reduces to the repeated application of the chain rule. Both Kelley [160] and Linnainmaa [183] describe versions of the backpropagation algorithm similar to its modern version. The first specific application of the backprop for neural networks might be described by Werbos [317]. It was popularized for deep learning by Rumelhart et al. [248] and proposed by LeCun [174] in its current form.

[160] Kelley (1960), "Gradient Theory of Optimal Flight Paths"

[174] LeCun (1985), "Une procedure d'apprentissage pour reseau a seuil asymmetrique"

[183] Linnainmaa (1970), "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors"

[248] Rumelhart et al. (1986), "Learning representations by backpropagating errors"

[317] Werbos (1982), "Applications of advances in nonlinear sensitivity analysis"

Similar to the *forward pass*, where we computed a complicated, composite function by iteratively computing the individual parts, we can compute the gradients in a **backward pass** by iteratively considering each component. Figure 4.2[7] shows the computational graph of a simple model of the form

7: Figure 4.2 and much of the explanation in this section is inspired by the examples shown in the lecture CS231n by the Stanford University.

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \phi(\boldsymbol{\theta}^\top \boldsymbol{x}) \,, \tag{4.2}$$

where ϕ is the sigmoid activation function, with $\phi(x) = \frac{1}{1+\exp^{-x}}$. The final loss of this simple model is given by

$$L(f_{\boldsymbol{\theta}}(\boldsymbol{x})) = y - f_{\boldsymbol{\theta}}(\boldsymbol{x}) \,, \tag{4.3}$$

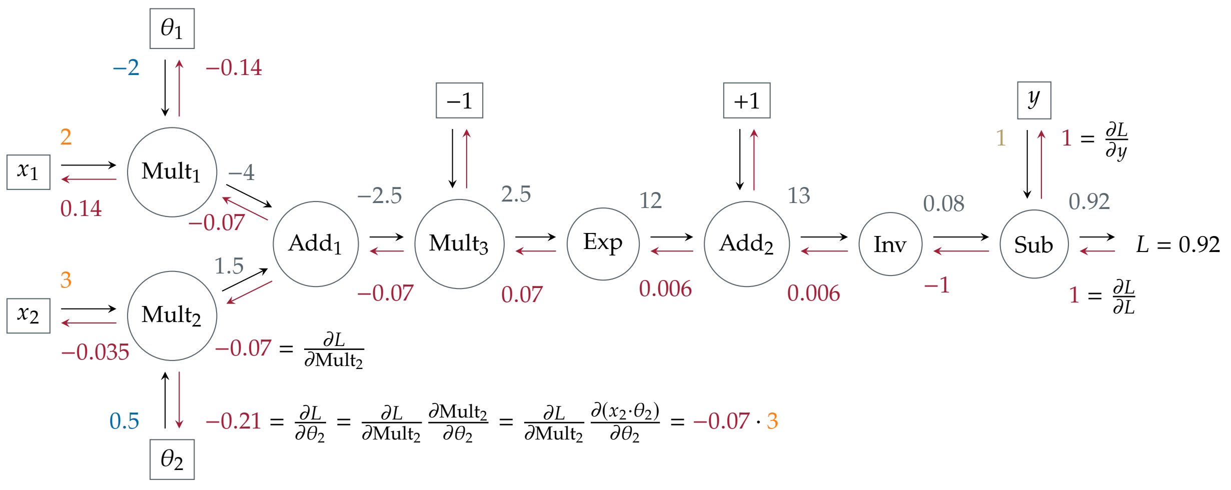
where y is the label corresponding to the inputs $\boldsymbol{x}$.

We are now interested in how changing the model's parameters would affect the final loss, or more specifically, how we should change the network's parameters to achieve a lower loss. Mathematically, this corresponds to computing the gradient $\frac{\partial L}{\partial \boldsymbol{\theta}}$. For this, we can apply the chain rule and compute it via

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = \frac{\partial L}{\partial f_{\boldsymbol{\theta}}} \frac{\partial f_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}} \overset{(4.3)}{=} -1 \cdot \frac{\partial f_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}} \,, \tag{4.4}$$

which splits the more complicated computation of $\frac{\partial L}{\partial \boldsymbol{\theta}}$ into two simpler computations, the first of which is so easy, we can directly write it down.

Instead of computing this for every parameter individually, we want to re-use existing computation by leveraging the structure encoded in the computational graph. Let us focus for now only on the node "Mult$_1$", which computes the product of the model parameter θ_1 and the input x_1. For convenience, we will denote the result of this computation with $z = \theta_1 \cdot x_1$. It is easy to compute



θ_1
−2
−0.14
x_1
2
0.14
Mult$_1$
−4
−0.07
x_2
3
−0.035
Mult$_2$
1.5
$-0.07 = \frac{\partial L}{\partial \text{Mult}_2}$
0.5
θ_2
$-0.21 = \frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial \text{Mult}_2}\frac{\partial \text{Mult}_2}{\partial \theta_2} = \frac{\partial L}{\partial \text{Mult}_2}\frac{\partial (x_2 \cdot \theta_2)}{\partial \theta_2} = -0.07 \cdot 3$
Add$_1$
−2.5
−0.07
−1
Mult$_3$
2.5
0.07
Exp
12
0.006
+1
Add$_2$
13
0.006
Inv
0.08
−1
y
1
$1 = \frac{\partial L}{\partial y}$
Sub
0.92
$L = 0.92$
$1 = \frac{\partial L}{\partial L}$

Figure 4.2: Illustration of the forward and backward pass in a computational graph. The circled nodes represent computations, *e.g.* "Add" represents a node that adds its two inputs and returns the sum as its output. For reference, multiple nodes of the same type are numbered, *i.e.* "Mult$_1$". In the **forward pass**, values are computed from the inputs shown in orange, *i.e.* $\boldsymbol{x} = \begin{pmatrix} 2 & 3 \end{pmatrix}^\top$. The result of this computational graph also depends on the parameters of the model $\boldsymbol{\theta} = \begin{pmatrix} -2 & 0.5 \end{pmatrix}^\top$, shown in blue. The computed values of the forward pass are shown in light gray either on top of or to the left of the corresponding arrow ($\rightarrow$) and result in $L = 0.92$ which takes into account the label $y = 1$ (shown in gold). In the **backward pass** (shown in red, below or right of the arrow ($\rightarrow$)), we recursively apply the chain rule to compute the derivative of each intermediate result and parameter with respect to the final output L. Effectively, the gradients "flow" backwards from the output all the way to the parameters or the inputs of the graph.

the derivatives of z with respect to both its inputs:

$$\frac{\partial z}{\partial \theta_1} = x_1 \qquad \frac{\partial z}{\partial x_1} = \theta_1 \,. \tag{4.5}$$

Let us assume, that we are given the gradient $\frac{\partial L}{\partial z}$, then it is also easy to compute the derivative of the *loss* with respect to both inputs of this "Mult$_1$" node:

$$\frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial \theta_1} = \frac{\partial L}{\partial z}x_1 \qquad \frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x_1} = \frac{\partial L}{\partial z}\theta_1 \,. \tag{4.6}$$

We can see that the derivative of the loss with respect to the inputs of the node, *e.g.* $\frac{\partial L}{\partial \theta_1}$, is the product of a "local" gradient $\frac{\partial z}{\partial \theta_1}$ and an "incoming" gradient $\frac{\partial L}{\partial z}$. The "local" gradient $\frac{\partial z}{\partial \theta_1}$ is a result of the computation done by the node itself, while the "incoming" gradient $\frac{\partial L}{\partial z}$ describes the effect the output of said node has on the rest of the computation.

Let us imagine that x_1 is not provided by the inputs to the network, but is instead a node itself. Similar to before, computing the gradients for the inputs of this "x_1" node would require both a "local" gradient, determined by the computation done in this node, and an "incoming" gradient $\frac{\partial L}{\partial x_1}$. Equation (4.6) provides exactly this "incoming" gradient for the node at x_1. In other words, if we start with some "incoming" gradient, we have a recipe for propagating this gradient of the loss through the entire network. All we need for this is a first "incoming" gradient to start with and knowledge about how to compute the local gradient for each individual node.

To get the first "incoming" gradient, we can start with the final node in our graph which computes the loss. Trivially, the output of this node has a gradient of 1, since $\frac{\partial L}{\partial L} = 1$. This provides us with the first gradient that we can subsequently propagate *backwards* through the network until we get the gradients at the parameters. Figure 4.2 provides a specific example of this backpropagation procedure with all its intermediate results. All that is required is for each node to "know" how to compute its "local" gradient (see Algorithm 4.1 for a example implementation). Fortunately, automatic differentiation frameworks such as PyTorch [228] or TensorFlow [1] can provide code for these nodes as basic computational building blocks. Specifically, they provide entire neural network layers, *e.g.* convolutional layers, that "know" how to efficiently compute both their forward and backward passes.

[228] Paszke et al. (2019), "PyTorch: An Imperative Style, High-Performance Deep Learning Library"

[1] Abadi et al. (2015), "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems"

```python
class MultiplyNode:
  def forward(self, x, y):
    """Forward pass."""
    self.x = x
    self.y = y
    z = x * y
    return z

  def backward(self,
    incoming_grad):
    """Backward pass."""
    dx = self.y *
     incoming_grad
    dy = self.x *
     incoming_grad
    return [dx, dy]
```

Algorithm 4.1: Example implementation for a multiplication node with forward and backward pass for automatic differentiation.

4.2 Neural Network Layers

Layers are the building blocks of deep neural networks. In general, layers are defined as receiving some input, transforming it and then passing it on to another layer. By stacking multiple layers, neural networks provide expressive and flexible modeling functions for many machine learning tasks. Neural network layers can contain learnable parameters, *e.g.* such as in a fully connected layer, but this is not necessary, *e.g.* pooling layers. In the next section, we will describe layer types that are commonly used as hidden layers, *i.e.* layers that are between the input and the output layer. In Section 4.2.2, we will take a closer look at possible activation functions and will provide an overview of the many functions that have been suggested. Section 4.2.3 is concerned with the loss functions. Although they are not technically part of the neural network, they are part of the larger computational graph.

4.2.1 Layer Types

The neurons in the hidden layers do most of the "heavy lifting" in neural networks. As a result, numerous layer types have been proposed in recent years. In the following, we will provide a short description of the some of the most common types in contemporary deep learning networks.

Fully Connected layer

An example of a fully connected layer is shown in Figure 4.1. In a fully connected layer[8] every neuron in the previous layer is connected to every neuron in this layer. Effectively, it is a vectorized version of the single neuron from Equation (4.1), and can be described mathematically as

$$z(\boldsymbol{x}) = \phi(\boldsymbol{\Theta}\boldsymbol{x} + \boldsymbol{b})\,. \tag{4.7}$$

It is parameterized by both a **weight matrix** $\boldsymbol{\Theta} \in \mathbb{R}^{n_l \times n_{\text{in}}}$ and a **bias vector** $\boldsymbol{b} \in \mathbb{R}^{n_l}$, where n_l is the number of neurons in this layer, and n_{in} the number of incoming connections, *e.g.* the number of neurons in the previous layer. Typical design choices for a fully connected layer are the number of neurons in this layer and the choice of activation function. Fully connected layers tend to be computationally expensive and in modern networks are mostly used at the end of the network, *i.e.* close to the network's output.

8: A *fully connected layer* is also called a *dense* layer, or sometimes a *linear* layer. The *linear* layer usually does not include the activation function, *i.e.* only the transformation $\boldsymbol{z} = \boldsymbol{\Theta}\boldsymbol{x} + \boldsymbol{b}$.

Convolutional layer

Convolutional layers use a **filters**[9] to compute feature maps from a given input data. Originally, they have been designed for images [90, 176], to leverage their spatial invariance, *i.e.* the fact that a cat remains a cat, even when shifted by a few pixels in an image. Figure 4.3 shows an example of how a convolutional layer computes a feature map by performing a convolution of the input image with its learnable filter. In this example, a 2×2 filter is shifted over a 2×2 sub-array, called the *receptive field*, of the 5×5 input image. Each pixel value in the input image is multiplied by the associated value in the filter and all values are added to produce the corresponding result in the feature map. Afterwards, the filter shifts its position, *e.g.* by one pixel to the right, repeating the process to compute the next value in the feature map. Note, that the filter only has four parameters, but produces an entire output array. If the filter was designed to, for example, detect edges, only four parameters[10] would be sufficient to provide a feature map of detected edges for the entire image. The same would require much more parameters, if performed by a fully connected layer. Commonly multiple filters are used in parallel to compute multiple feature maps independently from the same input image. Typical design choices for a convolutional layer are the size of the kernel, the stride, *i.e.* the number of pixels the kernel moves for each operation, the number of filters, and padding options for how to deal with the borders of the image.

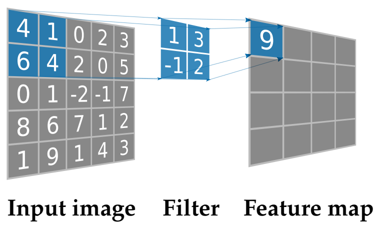


Figure 4.3: Illustration of a convolutional layer. An element of the feature map is computed by multiplying the filter element-wise with a portion of the input image and summing it together.

9: The *filter* is often also called a *kernel*.

[90] Fukushima (1980), "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position"

[176] LeCun et al. (1989), "Backpropagation Applied to Handwritten Zip Code Recognition"

10: Typically, the a convolutional layer would also contain a bias. This bias would add a single parameter per filter.

Pooling

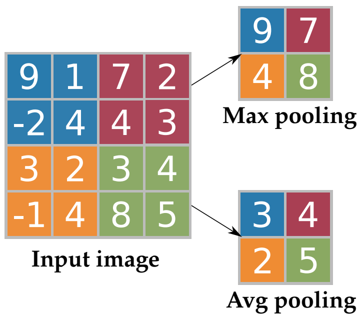


Figure 4.4: Illustration of the different types of pooling layer. The pooling operation illustrated here uses a 2×2 kernel size and a stride of 2, meaning that it will move two pixels before computing the next output.

Pooling layers aim to downsample the feature maps therefor providing a reduction of the spatial dimension. They work similar to convolutional layers, where an operation, this time the pooling operation computing a summary statistic of the receptive field, sweeps across the entire input. **Max pooling** [355] layers, for example, report the maximum output within a certain receptive field. The second common type of pooling is the **average pooling** layer which calculates the average, see Figure 4.4. Pooling layers help to reduce the dimensionality of the propagated features and can tell whether some feature is present, if it is not important where exactly it is. Common hyperparameters of pooling layers are the size of the pooling window, the stride, and the padding options.

[355] Zhou et al. (1988), "Computation of optical flow using a neural network"

[278] Srivastava et al. (2014), "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

Dropout

Dropout [278] is a special layer since its main advantage is providing a cheap way of regularization (see Section 2.3). During training, dropout sets certain inputs to a layer randomly to zero. The probability of setting inputs to zero is a hyperparameter of this layer. The idea behind the regularization effect of dropout is that by randomly dropping connections, no single node in the network is solely responsible for the network's prediction. The network cannot rely on individual nodes, as they might be inactivated due to the dropout layer, and must instead learn more robust and redundant features. It provides an efficient way for model averaging with neural networks, since it effectively trains multiple different sub-networks jointly.

Batch Normalization

[141] Ioffe et al. (2015), "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift"

11: *Batch normalization* is often abbreviated to *batch norm*.

Ioffe and Szegedy [141] introduced **batch normalization**[11] to speed up and stabilize neural network training by re-centering and re-scaling the layer's input. Batch normalization first performs a normalization step of its B inputs $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(B)}\}$ that are part of the mini-batch:

$$\tilde{\boldsymbol{x}}^{(i)} = \frac{\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_{\mathbb{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathbb{B}}^2 + \varepsilon}}, \tag{4.8}$$

where $\boldsymbol{\mu}_{\mathbb{B}} = \frac{1}{B}\sum_{i=1} \boldsymbol{x}^{(i)}$ and $\boldsymbol{\sigma}_{\mathbb{B}}^2 = \frac{1}{B}\sum_{i=1}(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_{\mathbb{B}})^2$ are the empirical mean and variance over the batch and ε is a small constant added for numerical stability. To restore some of the representational power of the network, batch norm includes a *learned* transformation through the learned parameters γ and β:

$$z^{(i)} = \gamma \odot \tilde{x}^{(i)} + \beta \,. \tag{4.9}$$

Crucially, this correlates the individual gradients in a mini-batch, a fact, that we will encounter again in Chapter 7.

Empirically, batch normalization has shown to succesfully help with neural network training, though the reasons for this remains unclear. Originally, it was suggested to mitigate the problem of *internal covariate shift* but this has been called into question by recent investigations [*e.g.*, 255]. More recently, other normalization layers have been suggested such as layer normalization [13], instance normalization [304] or group normalization [325].

[255] Santurkar et al. (2018), "How Does Batch Normalization Help Optimization?"

[13] Ba et al. (2016), "Layer Normalization"

[304] Ulyanov et al. (2017), "Instance Normalization: The Missing Ingredient for Fast Stylization"

[325] Wu et al. (2018), "Group Normalization"

4.2.2 Activation Functions

An integral part of a neural network architecture are its **activation functions**. These activation functions link two layers, such as two convolutional layers, to provide a non-linear transformation. Using these *non-linearities* in the network allows the network to compute non-trivial functions, since combining only layers with linear operations would effectively collapse into a single affine transformation.[12] Non-linear activation function are therefor necessary and crucial elements for the expressive power of modern neural networks. Figure 4.6 provides a visual overview of popular activation functions used in deep learning.

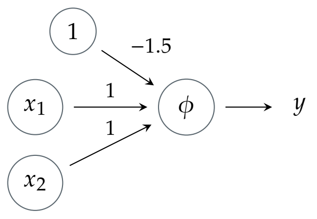


Figure 4.5: Example of an AND gate as a neural network. Using the binary step function as the activation function ϕ, we can build an AND gate for the inputs x_1 and x_2. Setting the bias, as well as the network weights for both inputs to 1 and the weight of the bias to −1.5 the output y of the node is equivalent to an AND gate. If instead the weight of the bias term were set to −0.5, the node would turn into an OR gate.

The **binary step** (or Heaviside) function is a rather simple activation function that switches on a neuron if a certain threshold is passed. Using this activation function, we can already build single layer networks that replicate logical AND or OR gates (see Figure 4.5). The derivative of this step function, however, is zero (and it is non-differentiable at the step) and is thus impractical to use for gradient-based learning (see Section 3.2.1). Instead, the logistic **sigmoid** and the hyperbolic tangent function (**tanh**) were used in early popular networks [*e.g.*, 123, 177] and are still used in many RNN models [*e.g.*, 59, 127].

12: Consider the affine transformations $g(x) = ax+b$ and $h(x) = cx+d$. The concatenation $g(h(x)) = acx + ad + b$ is simply another affine transformation given by $\tilde{g}(x) = \tilde{a}x + \tilde{b}$ with $\tilde{a} = ac$ and $\tilde{b} = ad + b$.

[123] Hinton et al. (2012), "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups"

[177] LeCun et al. (1998), "Gradient-Based Learning Applied to Document Recognition"

[59] Cho et al. (2014), "Learning Phrase Representations using RNN Encoder-Decoderfor Statistical Machine Translation"

[127] Hochreiter et al. (1997), "Long Short-Term Memory"

However, the derivative becomes close to zero for both activation functions when the magnitude of its input becomes large. This leads to a problem known as **vanishing gradients** [124] where this small gradient signal decreases exponentially as we propagate these small derivatives down to all layers. Since the *tanh* function is zero centered and it provides a stronger gradient signal, it is generally preferred over the *sigmoid* function [101].

[124] Hochreiter (1991), "Untersuchungen zu dynamischen neuronalen Netzen"

[101] Goodfellow et al. (2016), "Deep Learning"

An approach to address this problem of vanishing gradients is to use the *Rectified Linear Unit* (**ReLU**) [89, 98, 217], a piece-wise linear

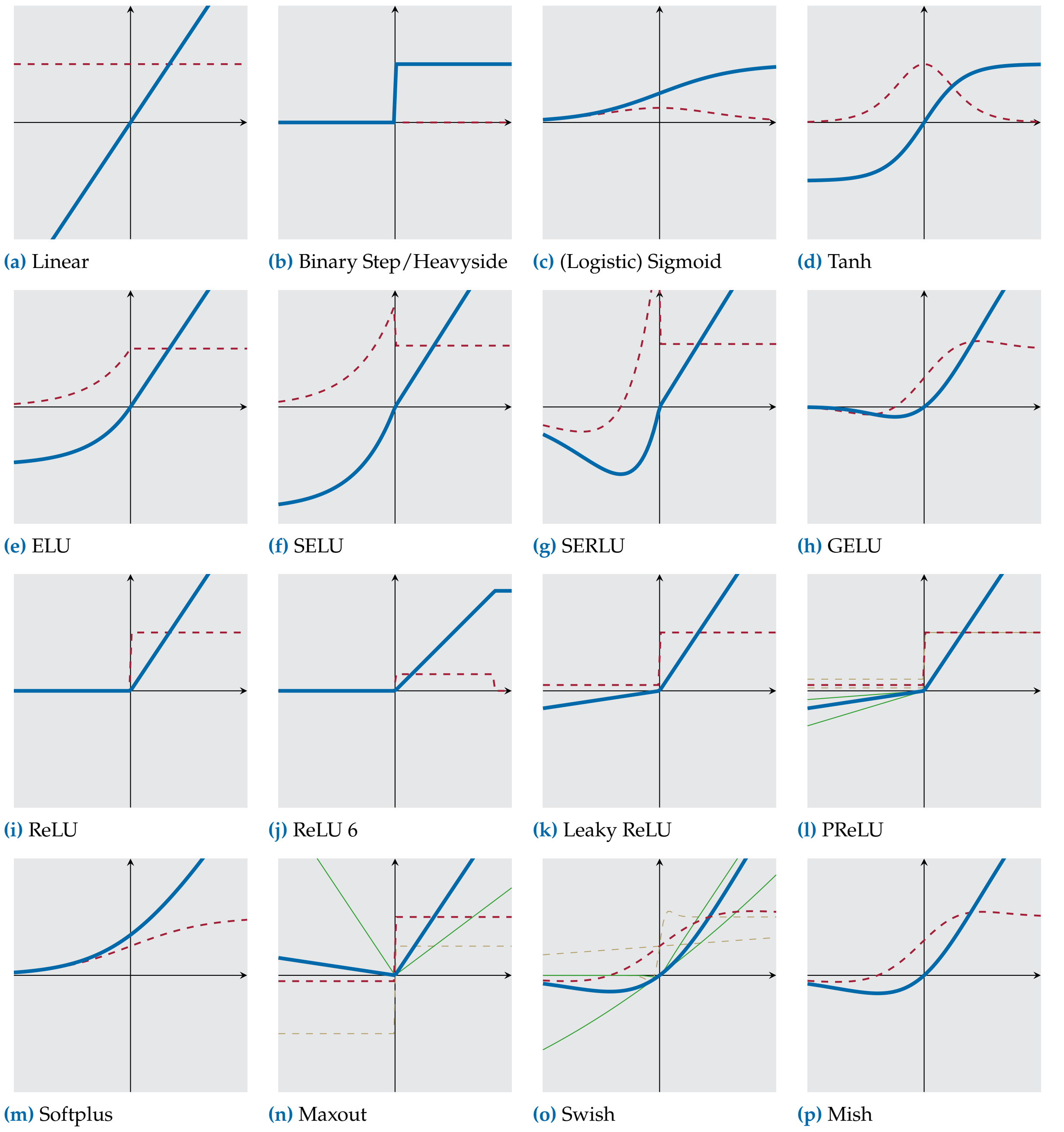


Figure 4.6: **Overview of commonly used activation functions in neural networks.** The activation function $\phi(x)$ is shown with a thick blue line (—), its derivative $\phi'(x)$ is shown with a dashed red line (- -). All plots show the domain -3 to 3, with the exception of ReLU 6 that was scaled to show the kink at $x = 6$. To highlight the difference between ReLU and Leaky ReLU, we increased the leakage factor from its usual default of 0.01 to 0.1. For parametric activation functions, *e.g.* PReLU or Swish, we show activations (—) and derivatives (- -) for several values of their learnable parameters.

function defined by $\phi(x) = \max(0, x)$. It is easy and cheap to compute and gradients can flow easily whenever the unit is activated. The similarity of the ReLU activation function to the linear function means that it retains many of the beneficial properties of the linear function. ReLU remains the most used activation function in deep neural networks, particular in convolutional neural networks for image classification [*e.g.*, 9, 117, 169, 270, 283, 284, 328].

ReLU activations can suffer from the **dying ReLU problem** if neurons are inactive for essentially all relevant inputs. Since the gradient is zero in those cases, no learning can occur and without any update the neuron becomes stuck and effectively "dies". **Leaky ReLU** addresses this by allowing a small, positive gradient for negative inputs. Non-linear variants of ReLU such as the **Softplus** [79], *Exponential Linear Unit* (**ELU**) [62], *Scaled Exponential Linear Unit* (**SELU**) [167], *Scaled Exponentially-Regularized Linear Unit* (**SERLU**) [345], or the *Gaussian Error Linear Unit* (**GELU**) [119] have been suggested with GELUs particularly popular for TRANSFORMER models such as GPT-3 and BERT [42, 72]. Another variant of ReLU is the **ReLU 6** activation [168] which clips the maximum activation at 6, *i.e.* $\phi(x) = \min(\max(0, x), 6)$. Krizhevsky [168] note that in their tests, it encouraged the model to learn spares features and it is used, for example in MobileNet [253].

Activation functions can itself have learnable parameters. *Parametric ReLUs* (**PReLU**) [116], for example, turn the leakage coefficient of *Leaky ReLU*s into a parameter of the model that can be learned jointly with other neural network parameters. Similarly, **Maxout** [103] is a learnable piece-wise linear function that can emulate both ReLU and Leaky ReLU as special cases. **Swish**, described by $\phi(x) = x\sigma(\beta x)$, with σ the logistic sigmoid function and β a learnable parameter, is an activation function found by automatic search techniques [235]. It is often used with β fixed to one, in which case it is identical to the *Sigmoid Linear Unit* **SiLU** [80] and similar to the recently suggested **Mish** activation function [207].

The choice of activation function can have a significant impact on the training speed and the final achievable prediction quality (see, for example, Section 7.3.2). However, it is usually chosen empirically, with ReLU being considered a default choice for many architectures [101].

4.2.3 Loss Functions

We will now take a look at the *cross-entropy* loss and the *mean squared error* loss as examples of common loss functions for classification and regression in machine learning and deep learning.

[89] Fukushima (1969), "Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements"
[98] Glorot et al. (2011), "Deep Sparse Rectifier Neural Networks"
[217] Nair et al. (2010), "Rectified Linear Units Improve Restricted Boltzmann Machines"
[9] Amodei et al. (2016), "Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin"
[117] He et al. (2016), "Deep Residual Learning for Image Recognition"
[169] Krizhevsky et al. (2009), "Learning multiple layers of features from tiny images"
[270] Simonyan et al. (2015), "Very Deep Convolutional Networks for Large-Scale Image Recognition"
[283] Szegedy et al. (2017), "Inception-v4, Inception-ResNet and the impact of residual connections on learning."
[284] Szegedy et al. (2015), "Going Deeper With Convolutions"
[328] Xie et al. (2017), "Aggregated Residual Transformations for Deep Neural Networks"
[79] Dugas et al. (2000), "Incorporating Second-Order Functional Knowledge for Better Option Pricing "
[62] Clevert et al. (2016), "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)"
[167] Klambauer et al. (2017), "Self-Normalizing Neural Networks"
[345] Zhang et al. (2018), "Effectiveness of Scaled Exponentially-Regularized Linear Units (SERLUs)"
[119] Hendrycks et al. (2016), "Gaussian Error Linear Units (GELUs)"
[42] Brown et al. (2020), "Language Models are Few-Shot Learners"
[72] Devlin et al. (2019), "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"
[168] Krizhevsky (2010), "Convolutional Deep Belief Networks on CIFAR-10"
[168] Krizhevsky (2010), "Convolutional Deep Belief Networks on CIFAR-10"
[253] Sandler et al. (2018), "MobileNetV2: Inverted Residuals and Linear Bottlenecks"

[116] He et al. (2015), "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"
[103] Goodfellow et al. (2013), "Maxout Networks"
[235] Ramachandran et al. (2017), "Searching for Activation Functions"
[80] Elfwing et al. (2018), "Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning"
[207] Misra (2019), "Mish: A Self Regularized Non-Monotonic Activation Function"
[101] Goodfellow et al. (2016), "Deep Learning"

Cross-entropy

Cross-entropy is a loss function used for classification tasks. For the sake of simplicity, we start with *binary* classification.

Definition 4.2.1 [Binary Cross-Entropy]
The **binary cross-entropy** loss function, for a set of N examples, is given by

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right] , \qquad (4.10)$$

where $\hat{y}^{(i)}$ is the model prediction and $y^{(i)}$ is the corresponding true target label of example i.

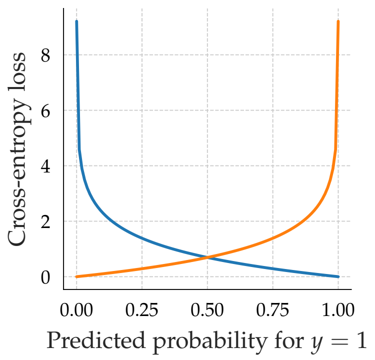


Figure 4.7: Illustration of the binary cross-entropy loss. The figure shows the binary cross-entropy loss for different predicted probabilities $\hat{y}$ of the machine learning model, if the true label $y = 1$ (—) or if the true label $y = 0$ (—). The loss vanishes as the predicted probability of the correct class approaches 1. In contrast, the loss increases rapidly for larger incorrect probabilities, penalizing wrong but confident predictions severely.

Since the true label is either 0 or 1 for a specific datum i, the loss uses either only the first or only the second term within the sum. If $y^{(i)} = 1$, then the second term of Equation (4.10) is zero. The individual loss of this example is then given by the log probability that the model predicted, which is why the cross-entropy loss is also sometimes called the *log loss* (see Figure 4.7). Consequently, predictions of higher probability of the correct class will lead to a lower loss. If $y^{(i)} = 0$, then the first term in the sum is zero. Here, the negative loss is given by the log probability that it is *not* part of the class labeled by 1. A perfect model would achieve a binary cross-entropy loss of 0.

We can generalize the binary cross-entropy to multiclass classification tasks with C different classes:

Definition 4.2.2 [Categorical Cross-Entropy]
Extending Definition 4.2.1 to C different classes, we can define the **categorical cross-entropy** as:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[\sum_{c=1}^{C} \boldsymbol{y}_c^{(i)}\log(\hat{\boldsymbol{y}}_c^{(i)})\right] , \qquad (4.11)$$

where $\boldsymbol{y}_c^{(i)}$ is simply a binary indicator if class label c is the correct class for observation i.

Since the labels are usually one-hot encoded, only one element of $\boldsymbol{y}^{(i)}$ is non-zero and the inner sum reduces to a single term, that of the correct class.

Figure 4.8 illustrates the different distributions arising from the true labels $\boldsymbol{y}$ and the predicted labels $\hat{\boldsymbol{y}}$. To measure the difference between these two distributions, we can use the *Kullback-Leibler (KL) divergence*:[13]

13: Although the KL divergence is often intuited as a *distance* measure, mathematical speaking it is not a distance metric. Crucially, the KL divergence does not fulfill the symmetry prerequisite of a metric, *i.e.* $D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$. Furthermore, it does not satisfy the triangular inequality.

Definition 4.2.3 [Kullback-Leibler Divergence]
We define the **Kullback-Leibler (KL) divergence** as a difference between two (discrete) distributions, $P(\boldsymbol{x})$ and $Q(\boldsymbol{x})$ over the same random variable $\boldsymbol{x}$, as

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{\boldsymbol{x}\sim P}\left[\log \frac{P(\boldsymbol{x})}{Q(\boldsymbol{x})}\right] = \mathbb{E}_{\boldsymbol{x}\sim P}\left[\log P(\boldsymbol{x}) - \log Q(\boldsymbol{x})\right] . \tag{4.12}$$

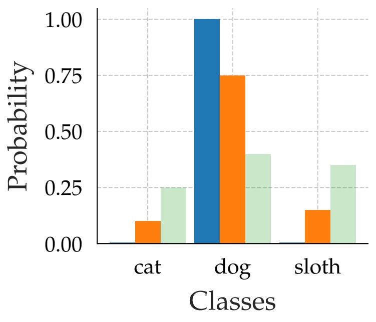


Figure 4.8: Difference between the predicted and the true label distribution. For the true label distribution (🟦) all weight falls onto a single, the correct, label. The predicted distribution (🟧) is spread out over multiple classes, but in this case, still gives the most weight to the correct label. A second predictive distribution (🟩), which gives less weight to the correct label has a higher cross-entropy (H(🟦, 🟩) ≈ 1.63) than the original predictive distribution (H(🟦, 🟧) ≈ 0.55) that is closer to the true label distribution.

If we assume that $P(\boldsymbol{x})$ is a fixed distribution, *i.e.* the true label distribution, and we want to change $Q(\boldsymbol{x})$, *i.e.* the predictive distribution, to minimize the KL divergence, we are left with

$$\min_Q D_{\mathrm{KL}}(P\|Q) = \min_Q \mathbb{E}_{\boldsymbol{x}\sim P}\left[\log P(\boldsymbol{x}) - \log Q(\boldsymbol{x})\right] \tag{4.13}$$
$$= \min_Q \mathbb{E}_{\boldsymbol{x}\sim P}\left[\log Q(\boldsymbol{x})\right] , \tag{4.14}$$

which includes the general definition of the *cross-entropy*:

Definition 4.2.4 [Cross-Entropy]
The **cross-entropy** H between two probability distributions $P(\boldsymbol{x})$ and $Q(\boldsymbol{x})$ is defined as

$$H(P, Q) = -\mathbb{E}_{\boldsymbol{x}\sim P}\left[\log Q(\boldsymbol{x})\right] . \tag{4.15}$$

Changing Q to minimize the *cross-entropy* is thus equivalent to minimizing the KL divergence, assuming that P is fixed and not part of the minimization process.

Mean Squared Error

A common and straightforward loss function for regression problems is the *mean squared error* (MSE) loss:

Definition 4.2.5 [Mean Squared Error]
The **mean squared error (MSE)** between a set of N model predictions $\hat{y}$ and true labels y is computed as

$$L = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 . \tag{4.16}$$

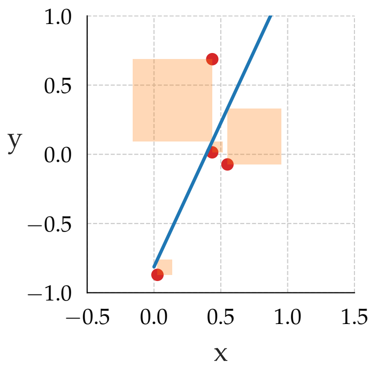


Figure 4.9: Visualization of the squared errors in a regression problem. The MSE considers the squared differences (🟧) between the data points (●) and the regression model (—).

The mean squared error simply computes the squared difference between the model's prediction $\hat{y}$ and the true label y, as illustrated in Figure 4.9. The MSE loss is always positive, due to the squaring, and a perfect fitting would result in an MSE loss of zero. The squaring ensures that large errors are penalized strongly, thus,

[137] Huber (1964), "Robust Estimation of a Location Parameter"

[307] Vaswani et al. (2017), "Attention Is All You Need"

[104] Gori et al. (2005), "A new model for learning in graph domains"

[257] Scarselli et al. (2009), "The Graph Neural Network Model"

[102] Goodfellow et al. (2014), "Generative Adversarial Nets"

putting a strong emphasis on describing all available data points at least somewhat accurately, avoiding large outliers.

An alternative to the mean squared error loss putting less emphasis on outliers would be the *mean absolute error* (MAE). Rather than taking the squared difference, MAE takes the absolute differences, rating the errors on a linear scale instead. As a compromise, the *Huber loss* [137] provides a mixture of both, by effectively using the MSE loss for small loss values and the MAE loss for larger loss values.

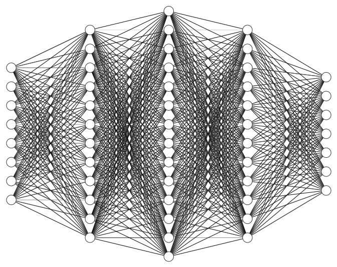


Figure 4.10: Illustration of a fully connected neural network. The width and opacity of the connections are proportional to the edge weights. Created with NN-SVG [178].

4.3 Common Architectures

A neural network *architecture* describes the overall structure of the neurons in the computational graph. It is therefore an umbrella term for the used layers, the number of neurons in that layer, or other hyperparameters. In this section, we want to describe common architectures of deep learning models, some of which we will re-visit in Chapter 5, when we aim to identify meaningful test problems for benchmarking deep learning optimizers. This list is far from exhaustive and is missing important architectures such as TRANSFORMERS [307], GNNs (GRAPH NEURAL NETWORKS) [104, 257], or models used in GENERATIVE ADVERSARIAL NETWORKS (GANs) [102].

[178] LeNail (2019), "NN-SVG: Publication-Ready Neural Network Architecture Schematics"

[142] Ivakhnenko et al. (1965), "Cybernetic Predicting Devices"

[245] Rosenblatt (1958), "The perceptron: a probabilistic model for information storage and organization in the brain."

[129] Hornik (1991), "Approximation capabilities of multilayer feedforward networks"

[195] Lu et al. (2017), "The Expressive Power of Neural Networks: A View from the Width"

4.3.1 Fully Connected Neural Networks

Fully connected neural networks, or multilayer perceptrons (MLPs) [142, 245] are a basic form of the feedforward neural network, whose main computational layers are fully connected layers, see Figure 4.10. By stacking multiple such layers, linking them via non-linear activation functions, they can express increasingly complex functions. MLPs with arbitrary width or depth are universal function approximators [*e.g.*, 129, 195]. Compared to other network architectures, fully connected ones tend to require much more parameters. CONVOLUTIONAL NEURAL NETWORKS, described in the next sections, for example, cleverly uses the inherent structure of images to reduce the required parameters. Recent work by Tolstikhin et al. [297] put MLPs back into focus, showing that they can attain competitive scores on relevant image classification benchmarks.

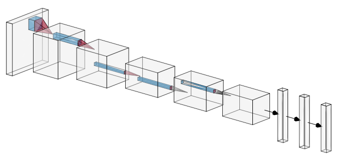


Figure 4.11: Illustration of a CONVOLUTIONAL NEURAL NETWORKS. Starting from an image with three color channels (shown as the depth), multiple convolutional layers transform it into feature maps. At the end, fully connected layers provide the final computation just before the output layer. Created with NN-SVG [178].

4.3.2 Convolutional Neural Networks

Just as their name suggests, CONVOLUTIONAL NEURAL NETWORKS (CNNs) [90, 176] use convolutional layers as their main computational unit. This leverages the spatial information encoded in images to reduce the number of parameters required to extract meaningful information. Traditionally, convolutional layers are combined with activation functions and pooling layers repeatedly, before using fully connected layers to provide the final output, see for example Figure 4.11. Figure 4.12 illustrates the typical architecture of a CNN using the All-CNN-C [277] as an example.

[277] Springenberg et al. (2015), "Striving for simplicity: The all convolutional net"

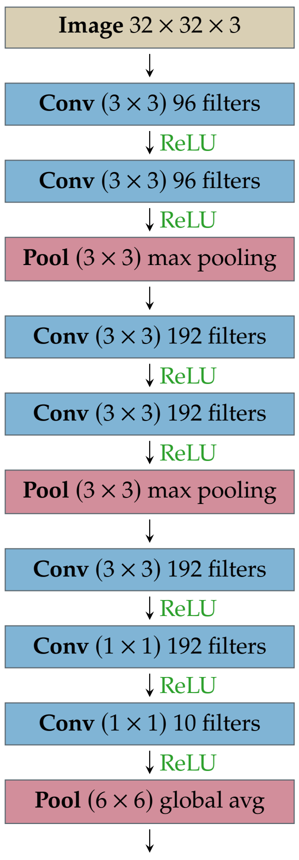


Figure 4.12: Schematic illustration of the layers in the All-CNN-C network. Convolution layers (●) are interspersed by pooling layer (●) and activation functions (●).

4.3.3 Residual Networks

Residual networks [117] introduce so-called *residual* or *skip* connections to neural networks. They slightly modify the usual structure of one layer feeding directly into the next (see for example Figure 4.12). Instead, it introduces an additional connection between two layers that are several steps apart, see Figure 4.13. This connection "skips" multiple layers and has been shown empirically to improve the training process especially of deeper neural networks. Any neural network that includes such a skip connection could be called a *residual* network, most prominently the ResNet-50, ResNet-101, and ResNet-152 series presented in [117].

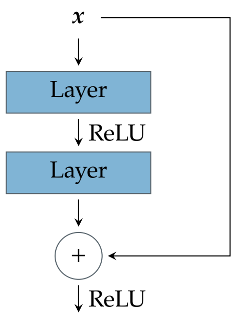


Figure 4.13: Illustration of a residual connection.

4.3.4 Recurrent Neural Networks

Recurrent neural networks (RNNs) are designed to process *sequences* of variable length as inputs, such as a sequence of characters, *e.g.* for machine translation. In contrast to regular *feed-forward networks* they do not assume that the inputs are independent of each other. Instead, their hidden layers not only use the current element of the sequence $x^{(i)}$ as an input, but also uses some extra information from the hidden layers of the previous example to influence the current output. If we consider, for example, the task of translating and English sentence word-by-word into German, it becomes clear while information from prior elements of the sequence should be considered: The correct translation of the word "second" could either be "zweiter" (as in "achieving second place") or "Sekunde" (as in "this thesis is ready in a second") depending on the context. By propagating this information from one word to the next, see Figure 4.14, recurrent neural networks can take this context into account.

[117] He et al. (2016), "Deep Residual Learning for Image Recognition"

There are multiple different recurrent architectures that differ in how the output and the propagated information are computed. Among the most popular ones are LSTM (long short-term memory) networks [127] or GRU (gated recurrent unit) networks [59].

[127] Hochreiter et al. (1997), "Long Short-Term Memory"

[59] Cho et al. (2014), "Learning Phrase Representations using RNN Encoder-Decoderfor Statistical Machine Translation"

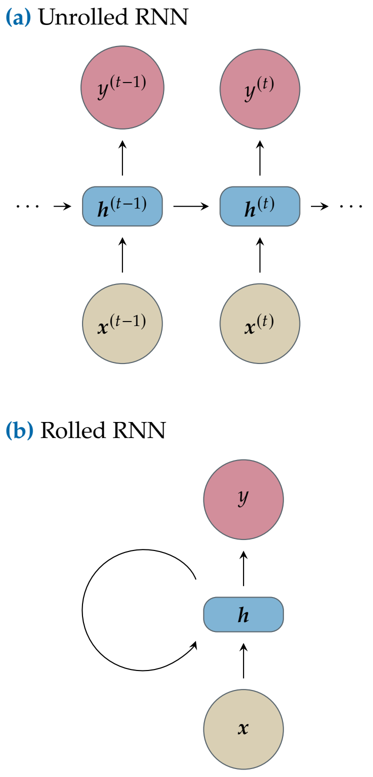


Figure 4.14: Illustration of a RECURRENT NEURAL NETWORK. (a) shows the unfolded version of a basic RNN. The inputs $x^{(t-1)}$, $x^{(t)}$, ... are individual elements of the input sequence. (b) Since the network re-uses the parameters of h for each element in the sequence we can visualize the network as having a feedback mechanism.

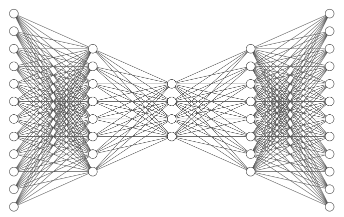


Figure 4.15: Illustration of an autoencoder network. The first part of the network encodes the input into a lower dimensional space. The decoder then aims to reconstruct the original input for this latent space representation. Created with NN-SVG [178].

[178] LeNail (2019), "NN-SVG: Publication-Ready Neural Network Architecture Schematics"

[305] Vahdat et al. (2020), "NVAE: A Deep Hierarchical Variational Autoencoder"

4.3.5 Autoencoder

In an **autoencoder**, the first part of a network, called the **encoder**, maps the input $x \in \mathbb{R}^I$ to a much lower-dimensional space $\mathbb{Z} \subseteq \mathbb{R}^m$, called the *latent space*. This latent space is usually much smaller than the original input space, *i.e.* $I \ll m$. Subsequently, the **decoder** part of the network aims to reconstruct the original image from this latent vector $z \in \mathbb{Z}$. Both encoder and decoder are usually trained jointly to minimize a reconstruction loss, such as the average squared error between the input and output pixels of an image. The encoding and decoding can be performed by any suitable neural network architecture such as fully connected or convolutional networks. The latent space serves as a bottleneck, visualized in Figure 4.15, which can be used, *e.g.* for compression.

VARIATIONAL AUTOENCODER (VAEs) share much of the architecture with traditional autoencoders but the output of the decoder is not interpreted directly as a vector in a latent space but as parameters of a pre-defined distribution. The distribution is typically a Gaussian distribution and the latent vector is subsequently sampled from it and used as the input to the decoder. VARIATIONAL AUTOENCODER can be used as a generative model, for example, to create realistic images [*e.g.*, 305].

Part II

Evaluating & Understanding Deep Learning Optimization

A Benchmark Suite for Deep Learning Optimizers 5

There is significant past and ongoing research on optimization methods for deep learning. Yet, perhaps surprisingly, there is no generally agreed-upon protocol for the quantitative and reproducible evaluation of such optimizers. In this chapter, we suggest routines and benchmarks for stochastic optimization, with special focus on the unique aspects of deep learning, such as stochasticity, tunability and generalization. As the primary contribution, we present DeepOBS, a Python package for benchmarking deep learning optimizers. The package addresses key challenges in the quantitative assessment of stochastic optimizers, and automates most steps of benchmarking. The library includes a wide and extensible set of ready-to-use realistic optimization problems, such as training Residual Networks for image classification on ImageNet or character-level language prediction models, as well as popular classics like MNIST and CIFAR-10. It comes with output back-ends that directly produce LaTeX code for inclusion in academic publications. The standardization of the benchmark process allows re-using existing results as baselines for novel optimization methods, without having to run costly experiments. Using the standardized evaluation protocol of DeepOBS, Chapter 6 will follow up with an elaborate and detailed comparison of current optimization methods. This chapter is largely based on [262].

[262] Schneider et al. (2019), "DeepOBS: A Deep Learning Optimizer Benchmark Suite"

5.1 Introduction

As deep learning has become mainstream, research on aspects like architectures [109, 117, 249, 283, 307] and hardware [52, 153, 227, 237] has exploded, and helped professionalize the field. In comparison, the optimization routines used to train deep nets have arguable changed only little. Comparably simple first-order methods like SGD [242] (see Update Rule 3.3.2), its momentum variants (Momentum) [221, 230] (see Update Rules 3.3.3 and 3.3.4), and Adam [166] (see Update Rule 3.3.8) remain standards [101, 156]. The low practical relevance of more advanced optimization methods is not for lack of research, though. There is a host of papers proposing new ideas for acceleration of first-order methods [*e.g.*, 24, 193], incorporation of second-order information [*e.g.*, 35, 203], and automating optimization [*e.g.*, 200, 258], to name just a few (see also Table 3.1 for a more complete collection of deep learning optimization methods). One problem that some of these methods face is that they are algorithmically involved and difficult to recreate

[109] Graves et al. (2014), "Neural Turing machines"
[117] He et al. (2016), "Deep Residual Learning for Image Recognition"
[249] Sabour et al. (2017), "Dynamic routing between capsules"
[283] Szegedy et al. (2017), "Inception-v4, Inception-ResNet and the impact of residual connections on learning."
[307] Vaswani et al. (2017), "Attention Is All You Need"
[52] Chen et al. (2016), "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks"
[153] Jouppi et al. (2017), "In-Datacenter Performance Analysis of a Tensor Processing Unit"
[227] Ovtcharov et al. (2015), "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware"
[237] Reagen et al. (2016), "Minerva: Enabling low-power, highly-accurate deep neural network accelerators"
[242] Robbins et al. (1951), "A Stochastic Approximation Method"
[221] Nesterov (1983), "A method for solving the convex programming problem with convergence rate $O(1/k^2)$"
[230] Polyak (1964), "Some methods of speeding up the convergence of iteration methods"
[166] Kingma et al. (2015), "Adam: A Method for Stochastic Optimization"

by practitioners. If they are not provided in packages for popular frameworks like TensorFlow [1], PyTorch [228], etc., they get little traction. Another problem, which we hope to address here, is that new optimization routines are often not convincingly compared to simpler alternatives in research papers, so practitioners are left wondering which of the many new choices is the best (and which ones even really work in the first place).

[101] Goodfellow et al. (2016), "Deep Learning"

[156] Karpathy (2017), "A peek at trends in machine learning"

[24] Bello et al. (2017), "Neural Optimizer Search with Reinforcement Learning"

[193] Loshchilov et al. (2017), "SGDR: Stochastic Gradient Descent with Warm Restarts"

[35] Botev et al. (2017), "Practical Gauss-Newton Optimisation for Deep Learning"

[203] Martens et al. (2015), "Optimizing Neural Networks with Kronecker-Factored Approximate Curvature"

[200] Mahsereci et al. (2017), "Probabilistic Line Searches for Stochastic Optimization"

[258] Schaul et al. (2013), "Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients"

[1] Abadi et al. (2015), "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems"

[228] Paszke et al. (2019), "PyTorch: An Imperative Style, High-Performance Deep Learning Library"

Designing an empirical protocol for the assessment of deep learning optimizers is not straightforward, and the corresponding experiments can be time-consuming. This is partly due to the idiosyncrasies of the domain:

- **Generalization:** While the optimization algorithm (should) only ever see the training-set, the practitioner cares about performance of the trained model on the test set, see Section 2.2. Worse, in some important application domains, the optimizer's loss function is not the objective we ultimately care about. For instance in image classification, practitioners often use a cross-entropy loss although the real interest may be the percentage of correctly labeled images, the accuracy. So which score should actually be presented in a comparison of optimizers? Train loss, because that is what the optimizer actually works on; test loss, because an overfitting optimizer is useless, or test accuracy, because that's what the human user cares about?
- **Stochasticity:** Sub-sampling (batching) the data-set to compute estimates of the loss function and its gradient introduces stochasticity, see Section 3.2.1. Thus, when an optimizer is run only once on a given problem, its performance may be misleading due to random fluctuations. The same stochasticity also causes many optimization algorithms to have one or several tuning parameters (learning rates, etc.). How should an optimizer with two free parameter be compared in a fair way with one that has only one, or even no free parameters?
- **Realistic settings, fair competition:** There is a widely-held belief that popular standards like MNIST [177] and CIFAR-10 [169] are too simplistic to serve as a realistic placeholder for a contemporary combination of large-scale data set and architecture. While this worry is not unfounded, researchers, ourselves included, have sometimes found it hard to satisfy the demands of reviewers for ever new data sets and architectures. Finding and preparing such data sets and building a reasonable architecture for them is time-consuming for researchers who want to focus on their novel algorithm. Even when this is done, one then has to not just run one's own algorithm, but also various competing baselines, like SGD, Momentum, Adam, etc. This

[177] LeCun et al. (1998), "Gradient-Based Learning Applied to Document Recognition"

[169] Krizhevsky et al. (2009), "Learning multiple layers of features from tiny images"

step does not just cost time, it also poses a risk of bias, as the competition invariably receives less care than one's own method. Reviewers and readers can never be quite sure that an author has not tried a bit too much to make their own method look good, either by choosing a convenient training problem, or by neglecting to tune the competition.

To address these problems, we propose an extensible, open-source benchmark specifically for optimization methods on deep learning architectures. We make the following three contributions:

- **A protocol for benchmarking stochastic optimizers.** Section 5.2 discusses and recommends best practices for the evaluation of deep learning optimizers. We define three key performance indicators: final performance, speed, and tunability, and suggest means of measuring all three in practice. We provide evidence that it is necessary to show the results of multiple runs in order to get a realistic assessment. Finally, we strongly recommend reporting both loss and accuracy, for both training and test set, when demonstrating a new optimizer as there is no obvious way those four learning curves are connected in general.
- **DeepOBS, a deep learning optimizer benchmark suite.**[1] We have distilled the above ideas into an open-source Python package, written in TensorFlow [1][2], which automates most of the steps presented in Section 5.2. The package provides over twenty off-the-shelf test problems across four application domains, including image classification and natural language processing, and this collection can be extended and adapted as the field makes progress. The test problems range in complexity from stochastic two dimensional functions to contemporary deep neural networks on data sets such as ImageNet. The package is easy to install in Python, using the pip toolchain. It automatically downloads data sets, sets up models, and provides a back-end to automatically produce LaTeX code that can directly be included in academic publications. This automation does not just save time, it also helps researchers to create reproducible, comparable, and interpretable results.
- **Showcase of its benchmarking capabilities.** From the collection of test problems, two sets, of four simple ("small") and four more demanding ("large") problems, respectively, are selected as a core set of benchmarks. Researchers can design their algorithm in rapid iterations on the simpler set, then test on the more demanding set. We argue that this protocol saves time, while also reducing the risk of overfitting in the algorithm design loop. In Section 5.4 we showcase DeepOBS by comparing the performance of SGD, SGD with

1: Code available at `https://github.com/fsschneider/deepobs`.

[1] Abadi et al. (2015), "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems"

2: After publication of the original manuscript [262], the framework was extended to provide additional support for PyTorch, which is described in [16].

[262] Schneider et al. (2019), "DeepOBS: A Deep Learning Optimizer Benchmark Suite"

[16] Bahde (2019), "Towards Meaningful Deep Learning Optimizer Benchmarks"

momentum (Momentum) and Adam on the small and large benchmarks. This aims at demonstrating the output of the benchmark suite. Chapter 6 uses DeepOBS and the presented protocol to provide an exhaustive benchmark of more than a dozen popular optimization methods for deep learning. Thanks to the standardized benchmark process, these results can be used as fair baselines when evaluating novel methods, without the need to re-compute these baselines. We hope that the benchmark suite will offer a common platform, allowing researchers to publicize their algorithms, giving practitioners a clear view of the state of the art, and helping the field to more rapidly make progress.

5.1.1 Related Works

To our knowledge, there is currently no commonly accepted benchmark for optimization algorithms that is well adapted to the deep learning setting.[3] This impression is corroborated by a more or less random sample of recent research papers on deep learning optimization [24, 76, 78, 166, 193, 203, 238, 341], whose empirical sections follow no joint standard (beyond a popularity of the MNIST data set). There *are* a number of existing benchmarks for deep learning as such. However, they do not focus on the optimization method. Instead, they are either framework or hardware-specific, or cover deep learning as a holistic process, wrapping together architecture, hardware and training procedure, The following are among the most popular ones:

3: This section describes the state of deep learning benchmarking at the time of publication of the original paper [262]. Section 6.1.1 provides a more up-to-date overview of related works, including the subsequent works of Choi et al. [60] and Sivaprasad et al. [271]. Here, we would like to put DeepOBS in the context of its original publication.

[262] Schneider et al. (2019), "DeepOBS: A Deep Learning Optimizer Benchmark Suite"

[60] Choi et al. (2019), "On Empirical Comparisons of Optimizers for Deep Learning"

[271] Sivaprasad et al. (2020), "Optimizer Benchmarking Needs to Account for Hyperparameter Tuning"

[24] Bello et al. (2017), "Neural Optimizer Search with Reinforcement Learning"

[76] Dozat (2016), "Incorporating Nesterov Momentum into Adam"

[78] Duchi et al. (2011), "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization"

[166] Kingma et al. (2015), "Adam: A Method for Stochastic Optimization"

[193] Loshchilov et al. (2017), "SGDR: Stochastic Gradient Descent with Warm Restarts"

[203] Martens et al. (2015), "Optimizing Neural Networks with Kronecker-Factored Approximate Curvature"

[238] Reddi et al. (2018), "On the Convergence of Adam and Beyond"

[341] Zeiler (2012), "ADADELTA: An Adaptive Learning Rate Method"

DAWNBench [63] The task in this challenge is to train a model for ImageNet, CIFAR-10 or SQuAD [234] as quickly as possible to a specified validation accuracy, tuning the entire tool-chain from architecture to hardware and optimizer.

MLPerf [4] [208] is another holistic benchmark similar to DAWNBench. It has two different rule sets; only the 'open' set allows a choice of optimization algorithm.

Deep Learning Frameworks (Comparison) [206] compares runtimes of different high-level frameworks.

DLBS [122] is a benchmark focused on the performance of deep learning models on various hardware systems with various software.

DeepBench [19] tests the speed of hardware for the low-level operations of deep learning, like matrix products and convolutions.

Fathom [3] is another hardware-centric benchmark, which among other things assesses how computational resources are spent.

TBD [359] focuses on the performance of three deep learning frameworks.

[63] Coleman et al. (2017), "DAWNBench: An End-to-End Deep Learning Benchmark and Competition"

[234] Rajpurkar et al. (2016), "SQuAD: 100,000+ Questions for Machine Comprehension of Text"

None of these benchmarks are good test beds for optimization research. Schaul et al. [259] defined unit tests for stochastic optimization. In contrast to the present work, they focus on small-scale problems like quadratic bowls and cliffs. In the context of deep learning, these problems provide unit tests, but do not give a realistic impression of an algorithm's performance in practice.

4: MLPerf has since transitioned to be a part of the non-profit consortium MLCommons. MLCommons has an *Algorithmic Efficiency Working Group* that is planning to hold a competition specifically for training algorithms. Section 8.2.1 provides a more detailed description of this competition and the working group, of which the author of this thesis is one of the two elected chairs.

[208] MLPerf (2018), "MLPerf.org"

[206] Microsoft Machine Learning (2018), "Comparing Deep Learning Frameworks: A Rosetta Stone Approach"

[122] Hewlett Packard Enterprise (2017), "Deep Learning Benchmarking Suite (DLBS)"

[19] Baidu Research (2016), "DeepBench"

[3] Adolf et al. (2016), "Fathom: Reference workloads for modern deep learning methods"

[359] Zhu et al. (2018), "TBD: Benchmarking and Analyzing Deep Neural Network Training"

[259] Schaul et al. (2013), "No more pesky learning rates"

5.2 Benchmarking Deep Learning Optimizers

This section expands the discussion from Section 5.1 of design desiderata for a good benchmark protocol, and proposes ways to nevertheless arrive at an informative, fair, and reproducible benchmark.

5.2.1 Stochasticity

The optimizer's performance in a concrete training run is noisy, due to the random sampling of mini-batches and initial parameters, see Section 3.2.1. There is an easy remedy, which nevertheless is not universally adhered to: Optimizers should be run on the same problem repeatedly with different random seeds, and all relevant quantities should be reported as mean and standard deviation of these samples. This allows judging the statistical significance of small performance differences between optimizers, and exposes the "variability" of performance of an optimizer on any given problem. The obvious reason why researchers are reluctant to follow this standard is that it requires substantial computational effort. DeepOBS alleviates this issue in two ways: It provides functionality to conveniently run multiple runs of the same setting with different seeds. More importantly, it provides stored baselines of popular optimizers, freeing computational resources to collect statistics rather than baselines.[5]

5: Both, the results from Section 5.3.4 and the more exhaustive benchmark presented in Chapter 6 are available open source and can be used as baselines.

5.2.2 Choice of Performance Metric

As described in Section 2.2, training a machine learning system is more than a pure optimization problem. The optimizers' immediate objective is the *training loss*, but the users' interest is in generalization performance, as estimated on a held-out test set. It has been observed repeatedly that in deep learning, different optimizers of similar training-set performance can have surprisingly different generalization [*e.g.*, 318]. Moreover, the loss function is regularly just a surrogate for the metric the user is ultimately interested in. In classification problems, for example, we are interested in classification accuracy, but this is infeasible to optimize directly. Thus,

[318] Wilson et al. (2017), "The Marginal Value of Adaptive Gradient Methods in Machine Learning"

there are up to four relevant metrics to consider: training loss, test loss, training accuracy and test accuracy. We strongly recommend reporting all four of these to give a comprehensive assessment of a deep learning optimizer. For hyperparameter tuning, we use test accuracy or, if that is not available, test loss, as the criteria. We also use them as the performance metrics in Table 5.2.

For empirical plots, many authors compute train loss (or accuracy) only on mini-batches of data, since these are computed during training anyway. But these mini-batch quantities are subject to significant noise. To get a decent estimate of the training-set performance, whenever we evaluate on the test set, we also evaluate on a larger chunk of training data, which we call a *train eval* set. In addition to providing a more accurate estimate, this allows us to "switch" the architecture to evaluation mode (*e.g.* dropout is not used during evaluation).

5.2.3 Measuring speed

Relevant in practice is not only the quality of a solution, but also the time required to reach it. A fast optimizer that finds a decent albeit imperfect solution using a fraction of other methods' resources can be very relevant in practice. Unfortunately, since learning curves have no parametric form, there is no uniquely correct way to define "time to convergence". In DeepOBS, we take a pragmatic approach and measure the time it takes to reach an "acceptable" *convergence performance*, which is individually defined for each test problem from the baselines SGD, Momentum and Adam each with their best hyperparameter setting.

Arguably the most relevant measure of speed is the wall-clock time to reach this convergence performance. However, wall-clock runtime has well-known drawbacks, such as dependency on hardware or weak reproducibility. Many authors report performance against gradient evaluations, since these often dominate the total computational costs. However, this can hide large per-iteration overheads. We recommend first measuring wall-clock time of both the new competitor and SGD on one of the small test problems for a few iterations, and computing their ratio. This computation, which can be done automatically using DeepOBS, can be done sequentially on the same hardware. One can then report performance against the products of iterations and per-iteration cost relative to SGD.

For many first-order optimization methods, such as SGD, SGD with Momentum or Adam, the choice of hyperparameters does not affect the runtime of the algorithm. However, for more evolved optimization methods, *e.g.* ones that dynamically estimate the

Hessian, the hyperparameters can influence the runtime significantly. In those cases, we suggest repeating the runtime estimate for different hyperparameters.

5.2.4 Hyperparameter tuning

Almost all deep learning optimizers expose tunable hyperparameters, *e.g.*, learning rates or averaging constants. The ease of tuning these hyperparameters is a relevant characteristic of an optimization method. How does one "fairly" compare optimizers with tunable hyperparameters?

A full analysis of the effects of an optimizer's hyperparameters on its performance and speed is tedious, especially since they often interact. Even a simpler sensitivity analysis requires a large number of optimization runs, which are infeasible for most users. Such analyses also do not take into account if hyperparameters have default values that work for almost all optimization problems and therefore require no tuning in general. Instead we recommend that authors find and report the best-performing hyperparameters for each test problem. Since DeepOBS covers multiple test problems, the spread of these best choices gives a good impression of the required tuning. Additionally, we suggest reporting the relative performance of the hyperparameter settings used during this tuning process (Figure 5.3 shows an example). Doing so yields a characterization of tunability without additional computations.

For the baselines presented in this chapter, we chose a simple log-grid search to tune the learning rate. While this is certainly not an optimal tuning method, and more sophisticated methods exists, see Section 3.4, it is nevertheless used often in practice and reveals interesting properties about the optimizers and their tunability.

DeepOBS supports authors in adhering to good scientific practice by removing various moral hazards. The baseline results for popular optimizers (whose hyperparameters have been tuned by us or, in the future, the very authors of the competing methods) avoid "starving" the competition of attention. When using different hyperparameter tuning methods, it is necessary to allocate the same computational budget for all methods in particular when comparing optimization methods of varying number of hyperparameters.

The fixed set of test problems provided by the benchmark makes it impossible to (knowingly or subconsciously) cherry-pick problems tuned to a new method. And finally, the fact that the benchmark spreads over multiple such problem sets constitutes a mild but natural barrier to "overfit" the optimizer method to established architectures or data sets like MNIST.

5.3 Benchmark Suite Overview

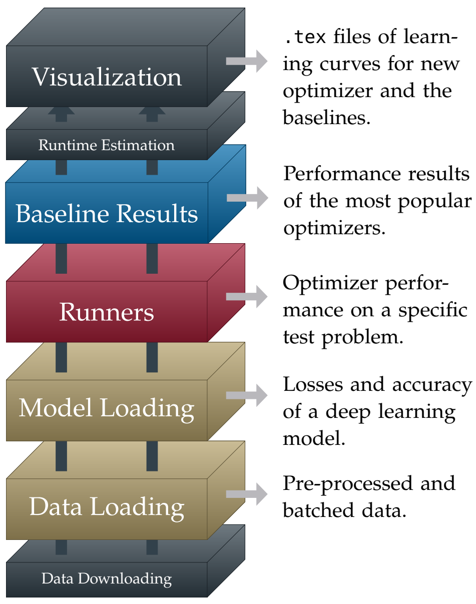

Visualization
Runtime Estimation
Baseline Results
Runners
Model Loading
Data Loading
Data Downloading
.tex files of learning curves for new optimizer and the baselines.
Performance results of the most popular optimizers.
Optimizer performance on a specific test problem.
Losses and accuracy of a deep learning model.
Pre-processed and batched data.

Figure 5.1: Illustration of the different steps implemented in the DeepOBS package and their outputs. The color of each block highlights the way a user mostly interacts with this part. Blocks in ● signify classes, those in ● are used via command line scripts. ● signals data packaged with DeepOBS and ● denotes parts provided through template scripts.

DeepOBS provides the full stack required for rapid, reliable, and reproducible benchmarking of deep learning optimizers. At the lowest level, a **data loading** (Section 5.3.1) module automatically loads and pre-processes data sets downloaded from the net. These are combined with a set of **models** (Section 5.3.2) to define test problems. At the core of the library, **runners** (Section 5.3.3) take care of the actual training, and log a multitude of statistics, *e.g.*, training loss or test accuracy. **Baseline** results from other optimization methods can be used directly, provided they were computed using the same protocol as suggested by DeepOBS.[6] The **visualization** (Section 5.3.6) script maps the results directly to LaTeX output.

Future releases of DeepOBS will include a version number that follows the pattern MAJOR.MINOR.PATCH, where MAJOR versions will differ in the selection of the benchmark sets, MINOR versions signify changes that could affect the results. PATCHES will not affect the benchmark results. All results obtained with the same MAJOR.MINOR version of DeepOBS will be directly comparable, all results with the same MAJOR version will compare results on the same problems.

We now give a brief overview of the functionality of DeepOBS.[7]

6: The results from the extensive benchmark presented in the next chapter, for example, can be used. They provided extensively tuned results from fifteen popular training algorithms.

7: The full documentation can be found online at `https://deepobs.readthedocs.io/`.

5.3.1 Data Loading

DeepOBS can automatically download and pre-process all necessary data sets.[8] Excluding ImageNet, the downloaded data sets require less than one GB of disk space.

8: At the moment, ImageNet is not part of this automatic procedure, since ImageNet requires registration to download the data set, and is comparably large, thus impractical for many users.

The DeepOBS data loading module then performs all necessary processing of the data sets to return inputs and outputs for the deep learning model (*e.g.* images and labels for image classification). This processing includes splitting, shuffling, batching and data augmentation. The data loading module can also be used to build new deep learning models that are not (yet) part of DeepOBS.

5.3.2 Models

Together, data set and model imply a loss function and together, they form an optimization problem. Table 5.2 provides an overview of the data sets and models included in DeepOBS. We selected problems for diversity of task as well as the difficulty of the optimization problem itself. The list includes popular image classification models on data sets like MNIST, CIFAR-10 or ImageNet, but also models for natural language processing and generative

Table 5.1: Overview of the test problems included in DeepOBS with their properties showing if the test problem includes convolutional layers (*Conv*), recurrent neural network cells (*RNN*), dropout layers (*Drop*), batch normalization layers (*BN*), or L^2 regularization (L^2). The first column highlights the machine learning task that the model performs, *i.e.* image classification 🟤, generative model 🔵, natural language processing 🔴 or problems where the loss function is given explicitly ⚫. Test problems marked in 🟩 and 🟧 are part of the small and large benchmark set, respectively.

	Data set	Model	Description	*Conv*	*RNN*	*Drop*	*BN*	L^2
⚫	2D	Noisy Beale	*Noisy version of the Beale function* [22]					
⚫		Noisy Branin	*Noisy version of the Branin function* [40]					
⚫		Noisy Rosenbrock	*Noisy version of the Rosenbrock function* [246]					
⚫	Quadratic	🟩 Deep	*100-dimensional ill-conditioned noisy quadratic* [50]					
🟤	MNIST [177]	Log. Regr.	*Logistic regression*					
🟤		MLP	*Four layer fully connected network*					
🟤		2c2d	*Two conv. and two fully connected layers*	✓				
🔵		🟩 VAE	*Variational Autoencoder*	✓		✓		
🟤	Fashion MNIST [326]	Log. Regr.	*Logistic regression*					
🟤		MLP	*Four layer fully connected network*					
🟤		🟩 2c2d	*Two conv. and two fully connected layers*	✓				
🔵		🟧 VAE	*Variational Autoencoder*	✓		✓		
🟤	CIFAR-10 [169]	🟩 3c3d	*Three conv. and three fully connected layers*	✓				✓
🟤		VGG16	*Adapted version of VGG16* [270]	✓		✓		✓
🟤		VGG19	*Adapted version of VGG19*	✓		✓		✓
🟤	CIFAR-100 [169]	3c3d	*Three conv. and three fully connected layers*	✓				✓
🟤		VGG16	*Adapted version of VGG16*	✓		✓		✓
🟤		VGG19	*Adapted version of VGG19*	✓		✓		✓
🟤		🟧 All-CNN-C	*The all convolutional net from Springenberg et al.* [277]	✓		✓		✓
🟤		Wide ResNet-40-4	*Wide Residual Network* [339]	✓			✓	✓
🟤	SVHN [223]	3c3d	*Three conv. and three fully connected layers*	✓				✓
🟤		🟧 Wide ResNet-16-4	*Wide Residual Network*	✓			✓	✓
🟤	ImageNet [70]	VGG16	*VGG16*	✓		✓		✓
🟤		VGG19	*VGG19*	✓		✓		✓
🟤		Inception-v3	*Inception-v3 network as described by Szegedy et al.* [285]	✓		✓	✓	✓
🔴	Tolstoi	🟧 CharRNN	*Recurrent Neural Network for character-level language modeling*		✓	✓		

models. Additionally, three two-dimensional problems and an ill-conditioned quadratic problem are included. These simple tests can be used as illustrative toy problems to highlight properties of an algorithm and perform sanity-checks. Over time, we plan to expand this list when hardware and research progress renders small problems out of date, and introduces new research directions and more challenging problems.[9] The eight test problems that have been selected for the small and large benchmark set are described in more detail in Appendix A. All data sets, models and test problems are described extensively in DeepOBS's documentation.[10]

5.3.3 Runners

The runners of the DeepOBS package handle training and the logging of statistics measuring the optimizers performance. For optimizers following the standard TensorFlow optimizer API it is enough to provide the runners with a list of the optimizer's hyperparameters. We provide a template for this, as well as an example of including a more sophisticated optimizer that can't be described as a subclass of the TensorFlow optimizer API.

9: Table 5.2 presents the set of test problems at the time of publication. In the meantime, Sivaprasad et al. [271] extended it to include a sentiment classification task for text data. Tsingunidis [302] studied how GANs can be integrated in DeepOBS, adding four new test problems.

[271] Sivaprasad et al. (2020), "Optimizer Benchmarking Needs to Account for Hyperparameter Tuning"

[302] Tsingunidis (2020), "Bring the GANs into Action - Extending DeepOBS with novel test problems"

10: Available at `https://deepobs.readthedocs.io/`.

[22] Beale (1958), "On an iterative method for finding a local minimum of a function of more than one variable"

[40] Branin (1972), "Widely convergent method for finding multiple solutions of simultaneous nonlinear equations"

5.3.4 Baselines

DeepOBS can also use pre-computed baselines results. These allow comparing a newly developed algorithm to the competition without computational overhead, and without risk of conscious or unconscious bias against the competition. The results from the showcase in Section 5.3.4 can be used as well as those from the more exhaustive benchmark described in Chapter 6. The baselines can be updated continuously, with further optimizers, assuming the methods perform competitively and they follow a common evaluation protocol.

[246] Rosenbrock (1960), "An automatic method for finding the greatest or least value of a function"

[50] Chaudhari et al. (2017), "Entropy-SGD: Biasing gradient descent into wide valleys"

[177] LeCun et al. (1998), "Gradient-Based Learning Applied to Document Recognition"

[326] Xiao et al. (2017), "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms"

[169] Krizhevsky et al. (2009), "Learning multiple layers of features from tiny images"

[270] Simonyan et al. (2015), "Very Deep Convolutional Networks for Large-Scale Image Recognition"

[277] Springenberg et al. (2015), "Striving for simplicity: The all convolutional net"

[339] Zagoruyko et al. (2016), "Wide Residual Networks"

[223] Netzer et al. (2011), "Reading digits in natural images with unsupervised feature learning"

[70] Deng et al. (2009), "ImageNet: A Large-Scale Hierarchical Image Database"

[285] Szegedy et al. (2016), "Rethinking the Inception Architecture for Computer Vision"

5.3.5 Estimate Runtime

DeepOBS provides an option to quickly estimate the runtime overhead of a new optimization method compared to SGD. It measures the ratio of wall-clock time between the new optimizer and SGD. By default this ratio is measured on five runs each, for three epochs, on a fully connected network on MNIST. However, this can be adapted to a setting which fairly evaluates the new optimizer, as some optimizers might have a high initial cost that amortizes over many epochs.

5.3.6 Visualizations

The DeepOBS visualization module reduces the overhead for the preparation of results, and simultaneously standardizes the presentation, making it possible to include a comparably large amount of information in limited space. The module produces `.tex` files with pgfplots code for all learning curves for the proposed optimizer as well as the most relevant baselines (Section 5.4 includes an example of this output).

5.4 Demonstrating the DeepOBS Suite

To showcase DeepOBS and provide an example of its output, we evaluate three popular deep learning optimizers (SGD, Momentum and Adam) on the eight test problems that are part of the small (problems P1 to P4) and large (problems P5 to P8) benchmark set.[11] The experiments were conducted with version `1.1.0` of DeepOBS. All experiments used 0.99 for the Momentum parameter and default parameters for Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$). The learning rate η was tuned for each optimizer and test problem individually, by evaluating on a logarithmic grid from $\eta_{\min} = 10^{-5}$ to $\eta_{\max} = 10^{2}$ with 36 samples. Once the best learning rate has

11: This section serves mostly as a demonstration of DeepOBS's capabilities and illustrates the benefits of a standardized comparison. Chapter 6 uses the evaluation protocol of DeepOBS to provide a standardized and extensive benchmark of fifteen optimization methods to draw more evidence-backed conclusions on the current state of deep learning optimizers.

been determined, we run those settings ten times with different random seeds. While we are using a log grid search, researchers are free to use any other hyperparameter tuning method, however this would require re-running the baselines as well.

Figure 5.2 shows the learning curves of the eight problems in the small and large benchmark set using tuned hyperparameters. Table 5.2 summarizes the results from both benchmark sets. We focus on three main observations, which corroborate widely-held beliefs and support the case for an extensive and standardized benchmark.

There is no optimal optimizer for all test problems. While Adam compares favorably on most test problems, in some cases the other optimizers are considerably better. This is most notable on CIFAR-100 (P6), where Momentum is significantly better then the other two. We will investigate this statement in much more detail in Chapter 6.

The connection between the four learning metrics is non-trivial. Looking at P6 and P7 we note that the optimizers rank differently on train vs. test loss. However, there is no optimizer that universally generalizes better than the others; the generalization performance is evidently problem dependent. The same holds for the generalization from loss to accuracy (*e.g.* P3 or P6).

Adam is *somewhat* easier to tune. Between the eight test problems, the optimal learning rate for each optimizer varies significantly. Figure 5.3 shows the final performance against learning rate for each of the eight test problems. There is no significant difference between the three optimizers in terms of their learning rate sensitivity. However, in most cases, the order of magnitude of the optimal learning rate for Adam is in the order of 10^{-4} and 10^{-3} (with the exception of P1), while for SGD and Momentum this spread is slightly larger.

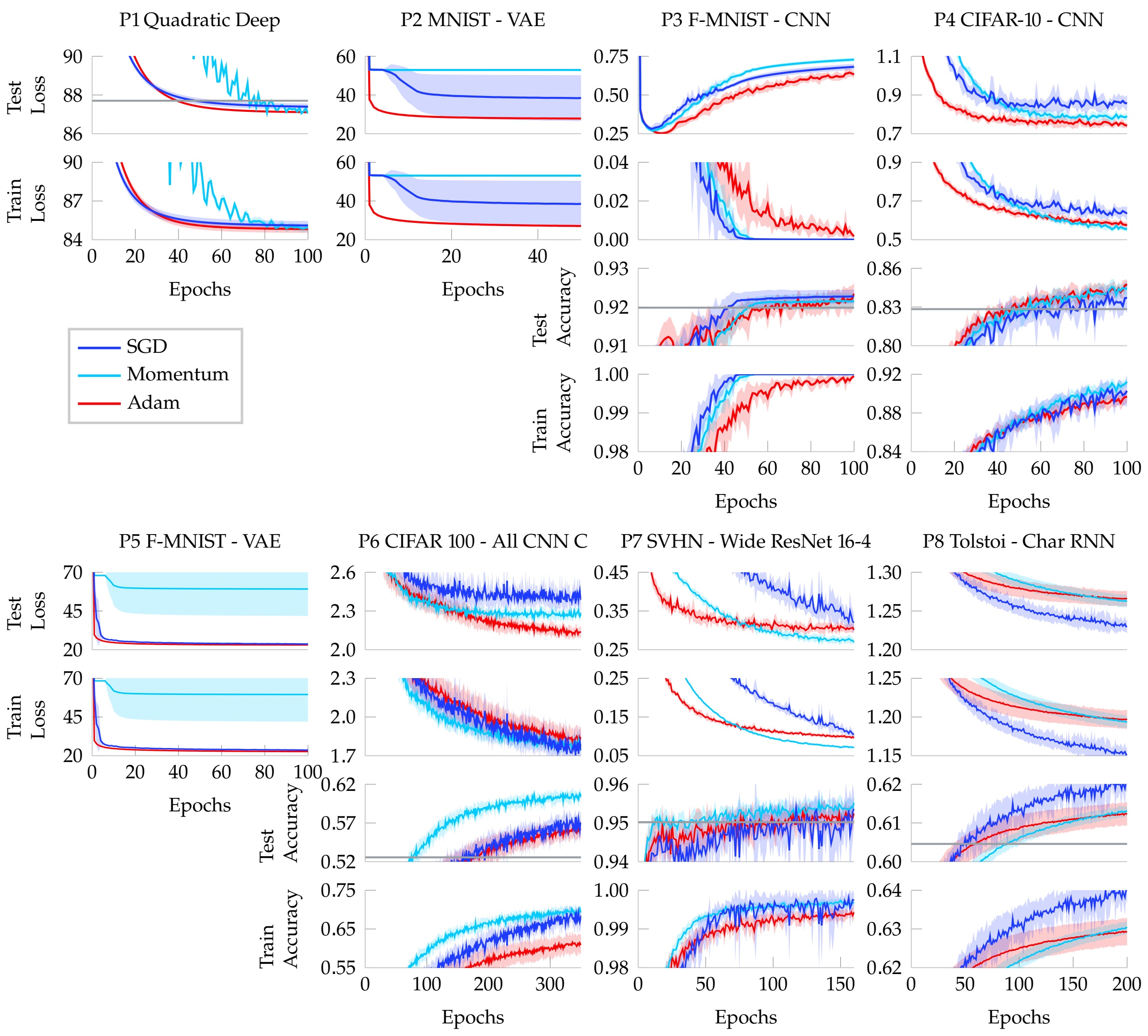


Figure 5.2: Learning curves for all eight test problems showing the performance of SGD, Momentum, and Adam produced with DeepOBS. Each column represents a single test problem, each row one of the four main performance metrics. The thick colored lines show the mean performance over ten runs of each optimizer with tuned hyperparameters, with the standard deviation shown via the shaded area. The convergence performances used to determine the speed of the optimizer (see Table 5.2) are shown with a horizontal gray line (—) in the subplot of the corresponding metric.

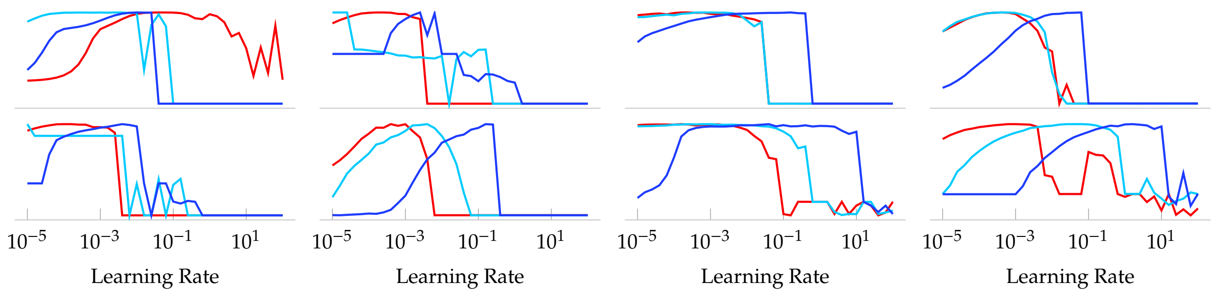


Figure 5.3: Relative performance against learning rate for each test problem and optimizer. Top row shows test problems P1 to P4, bottom row the test problems P5 to P8. The optimizers are represented in the same color as in Figure 5.2 and throughout this thesis, where ● represents SGD, ● represents Momentum, and ● the Adam optimizer.

Table 5.2: **DeepOBS benchmark for three popular optimizers**, showing the performance, speed and tuneability measures of SGD, Momentum and Adam on all eight test problems. The performance is measured using the test accuracy in percent (when available, otherwise the test loss) and the speed using the number of iterations to reach the convergence performance. All numbers are averaged over ten runs with the same hyperparameter settings. The tuneability row indicates the best performing set of hyperparameters per test problem, untuned hyperparameters are shown in gray for completeness.

Test Problem		**SGD**	**Momentum**	**Adam**
P1 **Quadratic Deep**	Performance	87.40	87.05	87.11
	Speed	51.1	70.5	39.9
	Tuneability	η: 1.58e-02	η: 2.51e-03 μ: 0.99	η: 3.98e-02 β_1: 0.9 β_2: 0.999 ε: 1e-08
P2 **MNIST VAE**	Performance	38.46	52.93	27.83
	Speed	1.0	1.0	1.0
	Tuneability	η: 3.98e-03	η: 2.51e-05 μ: 0.99	η: 1.58e-04 β_1: 0.9 β_2: 0.999 ε: 1e-08
P3 **F-MNIST CNN**	Performance	92.27 %	92.14 %	92.34 %
	Speed	40.6	59.1	40.1
	Tuneability	η: 1.58e-01	η: 2.51e-03 μ: 0.99	η: 2.51e-04 β_1: 0.9 β_2: 0.999 ε: 1e-08
P4 **CIFAR-10 CNN**	Performance	83.71 %	84.41 %	84.75 %
	Speed	42.5	40.7	36.0
	Tuneability	η: 6.31e-02	η: 3.98e-04 μ: 0.99	η: 3.98e-04 β_1: 0.9 β_2: 0.999 ε: 1e-08
P5 **F-MNIST VAE**	Performance	23.80	59.23	23.07
	Speed	1.0	1.0	1.0
	Tuneability	η: 3.98e-03	η: 1.00e-05 μ: 0.99	η: 1.58e-04 β_1: 0.9 β_2: 0.999 ε: 1e-08
P6 **CIFAR-100 All CNN C**	Performance	57.06 %	60.33 %	56.15 %
	Speed	128.7	72.8	152.6
	Tuneability	η: 1.58e-01	η: 3.98e-03 μ: 0.99	η: 1.00e-03 β_1: 0.9 β_2: 0.999 ε: 1e-08
P7 **SVHN Wide ResNet**	Performance	95.37 %	95.53 %	95.25 %
	Speed	28.3	10.8	12.1
	Tuneability	η: 2.51e-02	η: 6.31e-04 μ: 0.99	η: 1.58e-04 β_1: 0.9 β_2: 0.999 ε: 1e-08
P8 **Tolstoi Char RNN**	Performance	62.07 %	61.30 %	61.23 %
	Speed	47.7	88.0	62.8
	Tuneability	η: 1.58e+00	η: 3.98e-02 μ: 0.99	η: 2.51e-03 β_1: 0.9 β_2: 0.999 ε: 1e-08

5.5 Conclusion

Deep learning continues to pose a challenging domain for optimization algorithms. Aspects like stochasticity and generalization make it challenging to benchmark optimization algorithms against each other. We have discussed best practices for experimental protocols and presented the DeepOBS package, which provides an open-source implementation of these standards. We hope that DeepOBS can help researchers working on optimization for deep learning to build better algorithms, by simultaneously making the empirical evaluation simpler, yet also more reproducible and fair. By providing a common ground for methods to be compared on, we aim to speed up the development of deep-learning optimizers, and aid practitioners in their decision for an algorithm.

Empirically Comparing Deep Learning Optimizers 6

Choosing the optimizer is considered to be among the most crucial design decisions in deep learning, and it is not an easy one. The growing literature now lists hundreds of optimization methods, see Table 3.1. In the absence of clear theoretical guidance and conclusive empirical evidence, the decision is often made based on anecdotes. In this chapter, we aim to replace these anecdotes, if not with a conclusive ranking, then at least with evidence-backed heuristics. Having developed a standardized evaluation protocol in the previous chapter, we now perform an extensive, standardized benchmark of fifteen particularly popular deep learning optimizers. Analyzing more than 50,000 individual runs, we contribute the following three points: (i) Optimizer performance varies greatly across tasks. (ii) We observe that evaluating multiple optimizers with default parameters works approximately as well as tuning the hyperparameters of a single, fixed optimizer. (iii) While we cannot discern an optimization method clearly dominating across all tested tasks, we identify a significantly reduced subset of specific optimizers and parameter choices that generally lead to competitive results in our experiments: Adam remains a strong contender, with newer methods failing to significantly and consistently outperform it. Our open-sourced results[1] are available as challenging and well-tuned baselines for more meaningful evaluations of novel optimization methods without requiring any further computational efforts. This chapter is largely based on the publication [261].

1: Available at https://github.com/SirRob1997/Crowded-Valley---Results.

[261] Schmidt et al. (2021), "Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers"

6.1 The Crowded Field of Deep Learning Optimizers

Choosing the right optimization method and effectively tuning its hyperparameters heavily influences the training speed and final performance of neural networks, it is an important, every-day challenge to practitioners. It is probably the task that requires the most time and resources in many applications. Hence, optimization for machine learning has been a focal point of research, engendering an ever-growing list of methods (cf. Figure 6.1), many of them targeted at deep learning. The hypothetical machine learning practitioner who is able to keep up with the literature now has the choice among hundreds of methods (see Table 3.1), each with their own set of tunable hyperparameters, when deciding how to train a model.

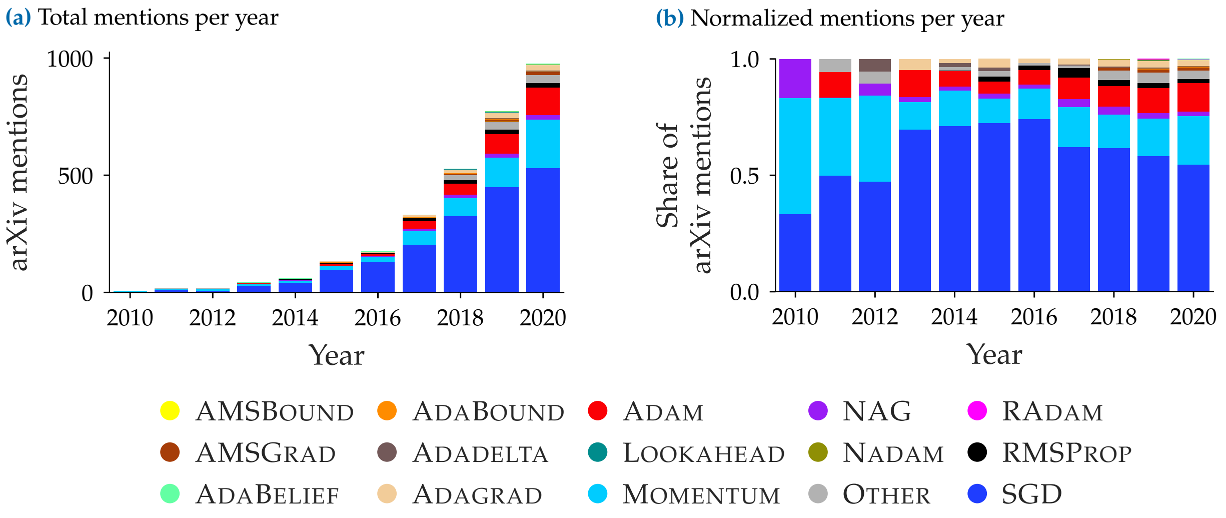


Figure 6.1: Number of times arXiv titles and abstracts mention specific optimizer per year. All non-selected optimizers from Table 3.1 are grouped into *Other*. This figure illustrates not only the expected increase in both methods and mentions (see left subplot (a) showing the total mentions per year), but also that our selection covers the most popular methods. In 2020, the excluded methods accounted for $< 4\,\%$ of the mentions (see right subplot (b)).

There is limited theoretical analysis that clearly favors one of these choices over the others. Some authors have offered empirical comparisons on comparably small sets of popular methods [*e.g.*, 60, 271, 318]; but for most optimizers, the only empirical evaluation is offered by the original work introducing the method. Many practitioners and researchers, meanwhile, rely on personal and anecdotal experience, and informal discussion with colleagues or on social media. The result is an often unclear, ever-changing "state of the art" occasionally driven by hype. The key obstacle for an objective benchmark is the combinatorial cost of such an endeavor posed by comparing a large number of methods on a large number of problems, with the high resource and time cost of tuning each method's parameters and repeating each (stochastic) experiment repeatedly for fidelity.

[60] Choi et al. (2019), "On Empirical Comparisons of Optimizers for Deep Learning"
[271] Sivaprasad et al. (2020), "Optimizer Benchmarking Needs to Account for Hyperparameter Tuning"
[318] Wilson et al. (2017), "The Marginal Value of Adaptive Gradient Methods in Machine Learning"

Leveraging the optimizer benchmark suite DeepOBS, presented in Chapter 5, we conduct a large-scale benchmark of optimizers to ground the ongoing debate about deep learning optimizers on empirical evidence, and to help understand how the choice of optimization methods and hyperparameters influences the training performance. Specifically, we examine whether recently proposed methods show an improved performance compared to more established methods such as SGD or Adam. Additionally, we assess whether there exist optimization methods with well-working default hyperparameters that are able to keep up with tuned optimizers. To this end, we evaluate fifteen optimization methods, selected for their perceived popularity, on a range of representative deep learning problems (see Figure 6.4) drawing conclusions from tens of thousands of individual training runs.

Right up front, we want to state that it is impossible to include all optimizers (see Table 3.1 for just a subset), and to satisfy any and all expectations readers may have on tuning, initialization, or the choice of problems—not least because everyone has different expectations in this regard. In our *personal opinion*, what is needed is an empirical comparison by a third party not involved in the original works. As the target audience of our work, we assume a careful practitioner who does not have access to near-limitless resources, nor to a broad range of personal experiences. As such, the core contributions of this chapter are:

1. **Assessing the progress in deep learning optimization.** In this work, we identify more than a hundred optimization methods (see Table 3.1) and more than 20 families of hyperparameter schedules (see Table 3.2) proposed for deep learning. We conduct a large-scale optimizer benchmark, specifically focusing on problems arising in deep learning. We evaluate fifteen optimizers on eight deep learning problems using four different schedules, tuning over dozens of hyperparameter settings. To our knowledge, this is the most comprehensive empirical evaluation of deep learning optimizers to date (see Section 6.1.1 on related work).
2. **Insights from more than 50,000 optimization runs.** Our empirical experiments indicate that an optimizer's performance highly depends on the problem (see Figure 6.4).[2] But some high-level trends emerge, too: (1) Evaluating multiple optimizers with default hyperparameters works approximately as well as tuning the hyperparameters for a fixed optimizer. (2) Using an additional untuned learning rate schedule helps on average, but its effect varies greatly depending on the optimizer and the problem. (3) While there is no optimizer that clearly dominates across all tested workloads, some of the methods we tested exhibited highly variable performance. Others demonstrated decent performance consistently. We deliberately abstain from recommending a single one among them, because we could not find a clear winner with statistical confidence.
3. **An open-source baseline for future optimizer benchmarks and meta-learning approaches.** Our results are available in an open and easily accessible form.[3] This data set contains 53,760 unique runs, each consisting of thousands of individual data points, such as the mini-batch training losses of every iteration or epoch-wise performance measures, for example, the loss on the full validation set or test set accuracy. These results can be used as competitive and well-tuned baselines for future benchmarks of new optimizers, drastically reducing the amount of computational budget required for a

2: This also echoes the results shown in the small showcase in Section 5.4.

3: Available at `https://github.com/SirRob1997/Crowded-Valley---Results`.

meaningful optimizer comparison. This collection of training curves could also be used for meta-learning novel optimization methods, hyperparameter search strategies, or hyperparameter adaptation strategies. To encourage researches to contribute to this collection, we made our baselines easily expandable.

The high-level result of our benchmark is, perhaps expectedly, *not* a clear winner. Instead, our comparison shows that, while some optimizers are frequently decent, they also generally perform similarly, often switching their positions in the ranking. This result is reminiscent, albeit not formally a rigorous result of the No Free Lunch Theorem [319]. A key insight of our comparison is that a practitioner with a new deep learning task can expect to do about *equally well* by taking almost any method from our benchmark and *tuning* it, as they would by investing the same computational resources into running a set of optimizers with their *default* settings and picking the winner.

[319] Wolpert et al. (1997), "No free lunch theorems for optimization"

Possibly the most important takeaway from our comparison is that "there are now enough optimizers". Methods research in stochastic optimization should focus on *significant* (conceptual, functional, performance) improvements—such as methods specifically suited for certain problem types, inner-loop parameter tuning or structurally novel methods. We make this claim not to discourage research but, quite on the contrary, to offer a motivation for more meaningful, non-incremental research.

6.1.1 Related Works

Following the rapid increase in publications on optimizers, *benchmarking* these methods for the application in deep learning has only recently attracted significant interest. The benchmarking framework DeepOBS, described in Chapter 5 and [262] includes a wide range of realistic deep learning problems together with standardized procedures for evaluating optimizers. Metz et al. [205] presented TaskSet, another collection of optimization problems focusing on smaller but more numerous problems. For the empirical analysis presented here, we use DeepOBS as it provides optimization problems closer to real-world deep learning tasks. In contrast to our evaluation of *existing* methods, TaskSet and its analysis focuses on meta-learning *new* optimizers or hyperparameters.

[262] Schneider et al. (2019), "DeepOBS: A Deep Learning Optimizer Benchmark Suite"

[205] Metz et al. (2020), "Using a thousand optimization tasks to learn hyperparameter search strategies"

Both Choi et al. [60] and Sivaprasad et al. [271] analyzed specific aspects of the benchmarking process. Sivaprasad et al. [271] used DeepOBS to illustrate that the performance of an optimizer depends significantly on the hyperparameter tuning budget. The analysis by Choi et al. [60] supports this point, stating that "the

[60] Choi et al. (2019), "On Empirical Comparisons of Optimizers for Deep Learning"

[271] Sivaprasad et al. (2020), "Optimizer Benchmarking Needs to Account for Hyperparameter Tuning"

hyperparameter search space may be the single most important factor explaining the rankings". They further stress an optimizer hierarchy, demonstrating that, given sufficient hyperparameter tuning, more general optimizers can never be outperformed by special cases. In their study, however, they manually defined a hyperparameter search space *per optimizer and problem* basing it on prior published results, prior experiences, or pre-tuning trials.

Here, we instead aim to identify well-performing general-purpose optimizers for deep learning, especially when there is no prior knowledge about well-working hyperparameter values for each specific problem. We further elaborate on the influence of our chosen hyperparameter search strategy in Section 6.4 discussing the limitations of our empirical study.

This benchmark also relates to empirical generalization studies of adaptive methods, such as that of Wilson et al. [318] which sparked an extensive discussion whether adaptive methods, such as ADAM, tend to generalize worse than standard first-order methods (*i.e.* SGD). By focusing on and reporting the *test set accuracy* we implicitly include the generalization capabilities of different optimizers in our benchmark results, an important characteristic of deep learning optimization.

[318] Wilson et al. (2017), "The Marginal Value of Adaptive Gradient Methods in Machine Learning"

6.2 Benchmarking Process

Any benchmarking effort requires tricky decisions on the experimental setup that influence the results. Evaluating on a specific task or picking a certain tuning budget may favor or disadvantage certain methods [271]. It is impossible to avoid these decisions or to cover all possible choices. Generally, we follow the protocol proposed in Chapter 5 and formalized in the DEEPOBS package. Aiming for generality, we evaluate the performance on the eight problems of DEEPOBS's small and large benchmark set, which consists of diverse real-world deep learning problems from different disciplines (Section 6.2.1). From a collection of more than a hundred deep learning optimizers (see Table 3.1) we select fifteen of the most popular choices (see Figure 6.1) for this benchmark (Section 6.2.2). For each problem and optimizer we evaluate all possible combinations of four different tuning budgets (Section 6.2.3) and four selected learning rate schedules (Section 6.2.4), covering

[271] Sivaprasad et al. (2020), "Optimizer Benchmarking Needs to Account for Hyperparameter Tuning"

Table 6.1: Summary of the DeepOBS problems used in our benchmark. Exact model configurations can be found in Appendix A.1 or [262]. The run time approximations are based on the run time for Adam on a Tesla K80 GPU.

Data set	Model	Task	Metric	Batch size	Budget *in epochs*	Approx. run time
P1 Artificial	Noisy quadratic	Minimization	Loss	128	100	< 1 min
P2 MNIST	VAE	Generative	Loss	64	50	10 min
P3 F-MNIST	Basic CNN: 2c2d	Classification	Accuracy	128	100	20 min
P4 CIFAR-10	Basic CNN: 3c3d	Classification	Accuracy	128	100	35 min
P5 F-MNIST	VAE	Generative	Loss	64	100	20 min
P6 CIFAR-100	All-CNN-C	Classification	Accuracy	256	350	4 h 00 min
P7 SVHN	Wide ResNet 16-4	Classification	Accuracy	128	160	3 h 30 min
P8 Tolstoi	RNN	Char. Prediction	Accuracy	50	200	5 h 30 min

the following combinatorial space:

$$\underset{\textbf{Problem}}{\begin{Bmatrix}\text{P1}\\ \text{P2}\\ \dots\\ \text{P8}\end{Bmatrix}_8} \times \underset{\textbf{Optimizer}}{\begin{Bmatrix}\text{Adam}\\ \text{NAG}\\ \dots\\ \text{SGD}\end{Bmatrix}_{15}} \times \underset{\textbf{Tuning}}{\begin{Bmatrix}\text{one-shot}\\ \text{small}\\ \text{medium}\\ \text{large}\end{Bmatrix}_4} \times \underset{\textbf{Schedule}}{\begin{Bmatrix}\text{constant}\\ \text{cosine}\\ \text{cosine warm restarts}\\ \text{trapez.}\end{Bmatrix}_4} .$$

Combining those options results in 1,920 configurations, where each of the fifteen optimizers is evaluated in 128 settings (*i.e.* on *eight* problems, with *four* budgets and *four* schedules). Including hyperparameter search and estimating the confidence interval, our main benchmark consists of 53,760 unique training curves.

6.2.1 Problems

We consider the eight optimization tasks summarized in Table 6.1, available as the "small" (P1–P4) and "large" (P5–P8) problem sets in DeepOBS. A detailed description of these problems, including architectures, training parameters, etc. can be found in Appendix A.1. All experiments were performed using version `1.2.0-beta` of DeepOBS and version `1.15` of TensorFlow [1].

[1] Abadi et al. (2015), "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems"

DeepOBS provides several performance metrics, including the training and test loss, and the validation accuracy. While these are all relevant, any comparative evaluation of optimizers requires picking only a few, if not just one particular performance metric. Following DeepOBS, our analysis (Section 6.3) focuses on the final test accuracy (or loss, if accuracy is not defined for this problem). This metric captures the optimizer's generalization ability and is thus highly relevant for practical use. Our publicly released results include all metrics for completeness. An example of training loss performance is shown in Figure B.12. Accordingly, the tuning (Section 6.2.3) is done with respect to the validation metric. We discuss possible limitations resulting from these choices in Section 6.4.

6.2.2 Optimizer

In Table 3.1 we collect over a hundred optimization methods introduced for or used in deep learning. This list was collected by multiple researchers trying to keep up with the field over recent years. It is thus necessarily incomplete, although it may well represent one of the most exhaustive of such collections. Even this incomplete list, though, contains too many entries for a benchmark with the degrees of freedom collected above. This is a serious problem for research: Even an author of a new optimizer, let alone a practitioner, cannot be expected to compare their work with every possible previous method.

We thus select a subset of fifteen optimizers, which we consider to be currently the most popular choices in the community, see Table 6.2. These do not necessarily reflect the "best" methods, but are either commonly used by practitioners and researchers, or have recently generated attention. Our selection is focused on first-order optimization methods, both due to their prevalence for non-convex optimization problems in deep learning as well as to simplify the comparison. Whether there is a significant difference between these optimizers or if they are inherently redundant is one of the questions this work investigates.

Our list focuses on optimizers over optimization techniques, although the line between the two is admittedly blurry. Techniques such as averaging weights [*e.g.*, 143] or ensemble methods [*e.g.*, 92] have been shown to be simple but effective at improving the optimization performance. Those methods, however, can be applied to all methods in our lists, similar to regularization techniques, learning rate schedules, or tuning method. We have, therefore, decided to omit them from Table 3.1.

[143] Izmailov et al. (2018), "Averaging weights leads to wider optima and better generalization"

[92] Garipov et al. (2018), "Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs"

6.2.3 Tuning

Budget: Optimization methods for deep learning regularly expose hyperparameters to the user, see Section 3.4. The user either relies on the default suggestion or sets them using experience from previous experiments, or using additional tuning runs to find the best-performing setting. All optimizers in our benchmark have tunable hyperparameters, and we consider four different *tuning budgets*.

Table 6.2: **Optimizers selected for our benchmarking process** with their respective color, hyperparameters, default values, tuning distributions and scheduled hyperparameters. Here, $\mathcal{LU}(\cdot,\cdot)$ denotes the log-uniform distribution while $\mathcal{U}(\cdot,\cdot)$ denotes the discrete uniform distribution.

Optimizer	Ref.	Parameters	Default	Tuning Distribution	Scheduled
● AMSBound	[196]	η	10^{-3}	$\mathcal{LU}(10^{-4}, 1)$	✓
		η^*	0.1	$\mathcal{LU}(10^{-3}, 0.5)$	
		β_1	0.9	$\mathcal{LU}(0.5, 0.999)$	
		β_2	0.999	$\mathcal{LU}(0.8, 0.999)$	
		γ	10^{-3}	$\mathcal{LU}(10^{-4}, 10^{-1})$	
		ε	10^{-8}	✗	
● AMSGrad	[238]	η	10^{-2}	$\mathcal{LU}(10^{-4}, 1)$	✓
		β_1	0.9	$\mathcal{LU}(0.5, 0.999)$	
		β_2	0.999	$\mathcal{LU}(0.8, 0.999)$	
		ε	10^{-8}	✗	
● AdaBelief	[361]	η	10^{-3}	$\mathcal{LU}(10^{-4}, 1)$	✓
		β_1	0.9	$\mathcal{LU}(0.5, 0.999)$	
		β_2	0.999	$\mathcal{LU}(0.8, 0.999)$	
		ε	10^{-14}	✗	
● AdaBound	[196]	η	10^{-3}	$\mathcal{LU}(10^{-4}, 1)$	✓
		η^*	0.1	$\mathcal{LU}(10^{-3}, 0.5)$	
		β_1	0.9	$\mathcal{LU}(0.5, 0.999)$	
		β_2	0.999	$\mathcal{LU}(0.8, 0.999)$	
		γ	10^{-3}	$\mathcal{LU}(10^{-4}, 10^{-1})$	
● AdaDelta	[341]	η	10^{-3}	$\mathcal{LU}(10^{-4}, 1)$	✓
		ε	10^{-8}	✗	
		$1-\rho$	0.05	$\mathcal{LU}(10^{-4}, 1)$	
● AdaGrad	[78]	η	10^{-2}	$\mathcal{LU}(10^{-4}, 1)$	✓
		ε	10^{-7}	✗	
● Adam	[166]	η	10^{-3}	$\mathcal{LU}(10^{-4}, 1)$	✓
		β_1	0.9	$\mathcal{LU}(0.5, 0.999)$	
		β_2	0.999	$\mathcal{LU}(0.8, 0.999)$	
		ε	10^{-8}	✗	
● Lookahead	[349]	η	0.5	$\mathcal{LU}(10^{-4}, 1)$	
Momentum		η_f	10^{-2}	$\mathcal{LU}(10^{-4}, 1)$	✓
abbr. LA(Mom.)		k	5	$\mathcal{U}(1, 20)$	
		$1-\rho$	0.01	$\mathcal{LU}(10^{-4}, 1)$	
● Lookahead	[349]	η	0.5	$\mathcal{LU}(10^{-4}, 1)$	
RAdam		η_f	10^{-3}	$\mathcal{LU}(1e-4, 1)$	✓
abbr. LA(RAdam)		β_1	0.9	$\mathcal{LU}(0.5, 0.999)$	
		β_2	0.999	$\mathcal{LU}(0.8, 0.999)$	
		ε	10^{-7}	✗	
		k	5	$\mathcal{U}(1, 20)$	
● Momentum	[230]	η	10^{-2}	$\mathcal{LU}(10^{-4}, 1)$	✓
		$1-\rho$	0.01	$\mathcal{LU}(10^{-4}, 1)$	
● NAG	[221]	η	10^{-2}	$\mathcal{LU}(10^{-4}, 1)$	✓
		$1-\rho$	0.01	$\mathcal{LU}(10^{-4}, 1)$	
● Nadam	[76]	η	10^{-3}	$\mathcal{LU}(10^{-4}, 1)$	✓
		β_1	0.9	$\mathcal{LU}(0.5, 0.999)$	
		β_2	0.999	$\mathcal{LU}(0.8, 0.999)$	
		ε	10^{-7}	✗	
● RAdam	[189]	η	10^{-3}	$\mathcal{LU}(10^{-4}, 1)$	✓
		β_1	0.9	$\mathcal{LU}(0.5, 0.999)$	
		β_2	0.999	$\mathcal{LU}(0.8, 0.999)$	
		ε	10^{-7}	✗	
● RMSProp	[294]	η	10^{-3}	$\mathcal{LU}(10^{-4}, 1)$	✓
		ε	10^{-10}	✗	
		$1-\rho$	0.1	$\mathcal{LU}(10^{-4}, 1)$	
● SGD	[242]	η	10^{-2}	$\mathcal{LU}(10^{-4}, 1)$	✓

The first budget consists of just a single run. This *one-shot* budget uses the default values proposed by the original authors, where available.[4] If an optimizer performs well in this setting, this has great practical value, as it drastically reduces the computational resources required for successful training.

4: Table 6.2 lists the default parameters used in our benchmark

The *small*, *medium* and *large* budgets consist of 25, 50, and 75 tuning runs, where the parameters for each setting are sampled using random search. Tuning runs for the small and medium budget were sampled using the distributions defined in Table 6.2. The additional 25 tuning runs of the large budget, however, were sampled using refined bounds: For each combination of optimizer, problem, and learning rate schedule we use the same distribution as before, but restrict the search space, to contain all hyperparameter configurations of the top-performing 20 % tuning runs from the medium budget are included.

We use a single seed for tuning, but for all configurations repeat the best setting with ten different seeds. This allows us to report standard deviations in addition to means, assessing stability. Our tuning process can sometimes pick "lucky" seeds, which do not perform well when averaging over multiple runs. This is arguably a feature rather than a bug, since it reflects practical reality. If an optimizer is so unstable that ten random seeds are required for tuning—which would render this benchmark practically infeasible—it would be impractical for the end-user as well. Our scoring naturally prefers stable optimizers. Appendices B.1 and B.2 provide further analysis of these cases and the general stability of our benchmark, showing among other things that failing seeds occur in less than 0.5 % of the tuning runs.

Tuning method: We tune parameters by random search without early-stopping for the small, medium and large budget. Random search is a popular choice due to its efficiency over grid search [27] and its ease of implementation and parallelization compared to Bayesian optimization (further discussed in Section 6.4). A minor complication of random search is that the sampling distribution affects the performance of the optimizer. The sampling distribution acts as a prior over good parameter settings, and bad priors consequently ruin performance. We followed the valid interval and intuition provided by the optimizers' authors for relevant hyperparameters. The resulting sampling distributions can be found in Table 6.2. Even though a hyperparameter might have a similar name in different optimization methods (*e.g.* learning rate α), its appropriate search space can differ. However, without grounded heuristics guiding the practitioner on how the hyperparameters differ between optimizers, the most straightforward approach for any user is to use the same search space. Therefore, in case there was no prior knowledge provided in the cited work we chose

[27] Bergstra et al. (2012), "Random Search for Hyper-Parameter Optimization"

similar distributions for similar hyperparameters across different optimizers.

What should be considered a hyperparameter? There is a fuzzy boundary between (tunable) hyperparameters and (fixed) design parameters. A recently contentious example is the ε in adaptive methods like Adam. It was originally introduced as a safeguard against division by zero, but has recently been re-interpreted as a problem-dependent hyperparameter.[5] Under this view, one can actually consider several optimizers called Adam: From an easy-to-tune but potentially limited Adam_{α}, only tuning the learning rate, to the tricky-to-tune but all-powerful $\text{Adam}_{\alpha,\beta_1,\beta_2,\varepsilon}$, which can approximate SGD in its hyperparameter space. While both share the update rule, we consider them to be different optimizers. For each update rule, we selected one popular choice of tunable parameters, *e.g.* $\text{Adam}_{\alpha,\beta_1,\beta_2}$, see Table 6.2.

5: See Choi et al. [60] for a more detailed discussion

[60] Choi et al. (2019), "On Empirical Comparisons of Optimizers for Deep Learning"

6.2.4 Schedules

The literature on learning rate schedules is now nearly as extensive as that on optimizers, see Table 3.2. *In theory*, schedules can be applied to all hyperparameters of an optimization method but to keep our configuration space feasible, we only apply schedules to the learning rate, by far the most popular practical choice [101, 342]. We choose four different learning rate schedules, trying to cover all major types of schedules:

[101] Goodfellow et al. (2016), "Deep Learning"

[342] Zhang et al. (2020), "Dive into Deep Learning"

- A *constant* learning rate;
- A *cosine decay* [193] as an example of a smooth decay;
- A *cosine with warm restarts* schedule [193] as a cyclical schedule;
- A *trapezoidal* schedule [330] from the warm-up schedules introduced in Goyal et al. [105].

[193] Loshchilov et al. (2017), "SGDR: Stochastic Gradient Descent with Warm Restarts"

[330] Xing et al. (2018), "A Walk with SGD"

[105] Goyal et al. (2017), "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour"

Appendix B.3 in the appendix provides a more detailed description of the used schedules.

6.3 Results of Comparing Popular Optimization Methods

How well do optimizers work out-of-the-box? By comparing each optimizer's one-shot results against the tuned versions of all fifteen optimizers, we can construct a 15×15 matrix of performance gains. Figure 6.2 illustrates this on five problems showing improvements by a positive sign and an orange cell. Detailed plots for all problems are in Figures B.5 and B.6 in the appendix. For example, the bottom

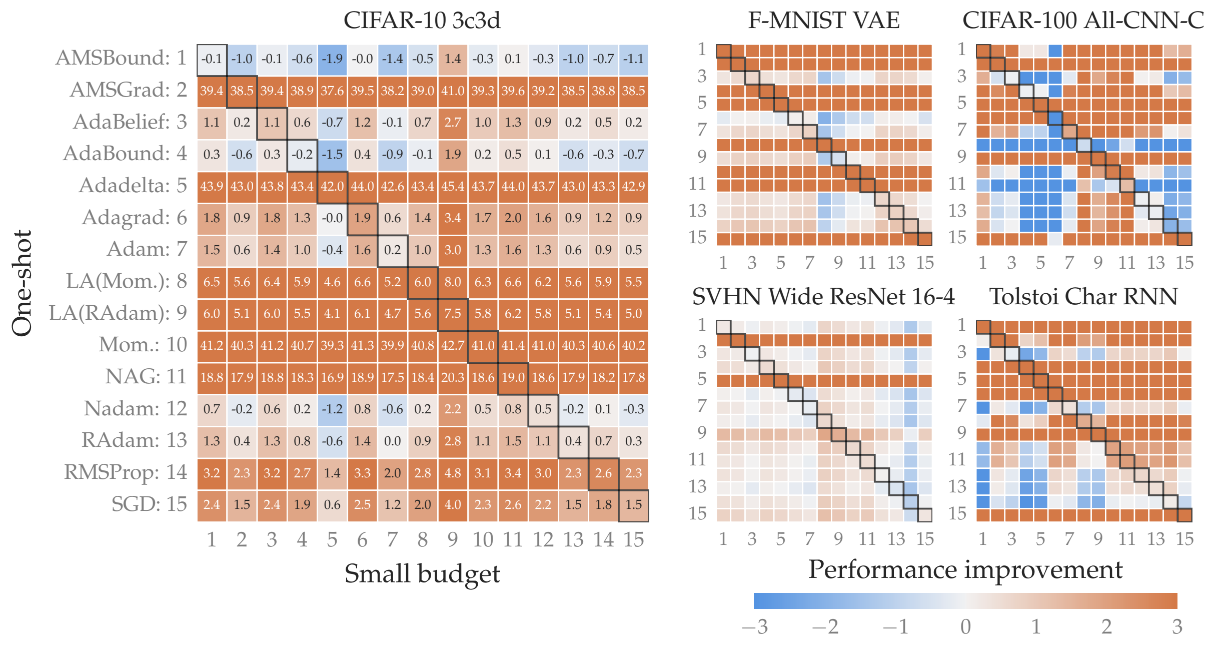


Figure 6.2: The test set performance improvement after switching from any untuned optimizer (y-axis, *one-shot*) to any tuned optimizer (x-axis, *small budget*) as an average over 10 random seeds for the *constant* schedule. For example, the bottom left cell of the largest matrix indicates that the tuned version of AMSBound (1) reaches a 2.4 % higher test accuracy than untuned SGD (15). We discuss the unintuitive occurrence of negative diagonal entries in Appendix B.5. The colormap is capped at ±3 to improve presentation, although larger values occur.

left cell of the largest matrix in Figure 6.2 shows that AMSBound *(1)* tuned using a small budget performs 2.4% better than SGD *(15)* with default parameters on this specific problem.

An **orange row** in Figure 6.2 indicates that an optimizer's default setting is performing badly, since it can be beaten by any well-tuned competitor. We can observe badly-performing default settings for Momentum, NAG and SGD, advocating the intuition that non-adaptive optimization methods require more tuning, but also for AMSGrad and Adadelta. This is just a statement about the default parameters suggested by the authors or the popular frameworks; well-working default parameters might well exist for those methods. Conversely, a **white & blue row** signals a well-performing default setting, since even tuned optimizers do not significantly outperform it. Adam, Nadam and RAdam, as well as AMSBound, AdaBound and AdaBelief all have white or blue rows on several (but not all!) problems, supporting the rule of thumb that adaptive methods have well-working default parameters. Conversely, **orange** (or **blue**) **columns** highlight optimizers that, when tuned, perform better (or worse) than all untuned optimization methods. We do not observe such columns consistently across tasks. This supports the conclusion that an optimizer's performance is heavily problem-dependent and that there is no single *best* optimizer across workloads.

Figures B.5 to B.8 in the appendix suggest an interesting alternative

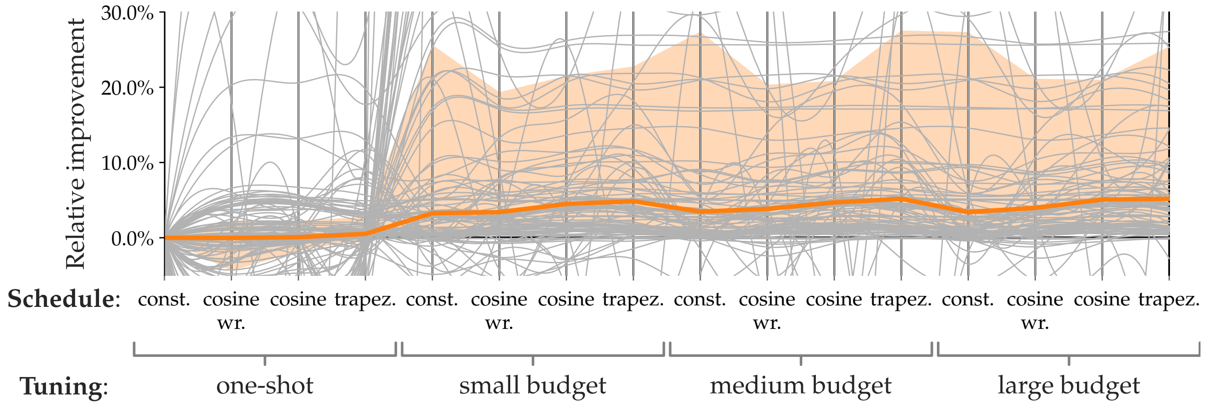


Figure 6.3: Lines in gray (—, smoothed by cubic splines for visual guidance only) show the relative improvement for a certain tuning budget and schedule (compared to the *one-shot* tuning without schedule) for all fifteen optimizers on all eight problems. The median over all lines is plotted in orange (—) with the shaded area (▮) indicating the area between the 25th and 75th percentile. With an increased budget and a schedule, one can expect a performance increase *on average* (orange lines), but not automatically for individual settings (*i.e.* gray lines can be unaffected or even decrease).

approach for machine learning practitioners: Instead of picking a single optimizer and tuning its hyperparameters extensively, trying out a few optimizers with default settings and picking the best one yields competitive results with less computational and tuning choice efforts. However, this might not hold for more complicated, structurally different tasks such as GANs [102] or TRANSFORMER models [307]. The similarity of those two approaches might be due to the fact that optimizers have implicit learning rate schedules [4] and trying out different optimizers is similar to trying out different (well-tested) schedules.

[102] Goodfellow et al. (2014), "Generative Adversarial Nets"
[307] Vaswani et al. (2017), "Attention Is All You Need"
[4] Agarwal et al. (2020), "Disentangling Adaptive Gradient Methods from Learning Rates"

How much do tuning and schedules help? We consider the final performance achieved by varying budgets and schedules to quantify the usefulness of tuning and applying parameter-free schedules (Figure 6.3). While there is no clear trend for any individual setting (gray lines), in the median we observe that increasing the budget improves performance, albeit with diminishing returns. For example, using the medium budget without any schedule leads to a median relative improvement of roughly 3.4 % compared to the default parameters (without schedule).

Applying an untuned schedule improves median performance as well. For example, the large tuning budget coupled with a trapezoidal learning rate schedule leads to a median relative improvement of the performance of roughly 5.2 % compared to the default parameters. However, while these trends hold in the median, their individual effect varies wildly among optimizers and problems, as is apparent from the noisy structure of the individual lines shown in Figure 6.3.

Which optimizers work well after tuning? Figure 6.4 compares the optimizers' performance across all eight problems. There is no single optimizer that dominates its competitors across all tasks.

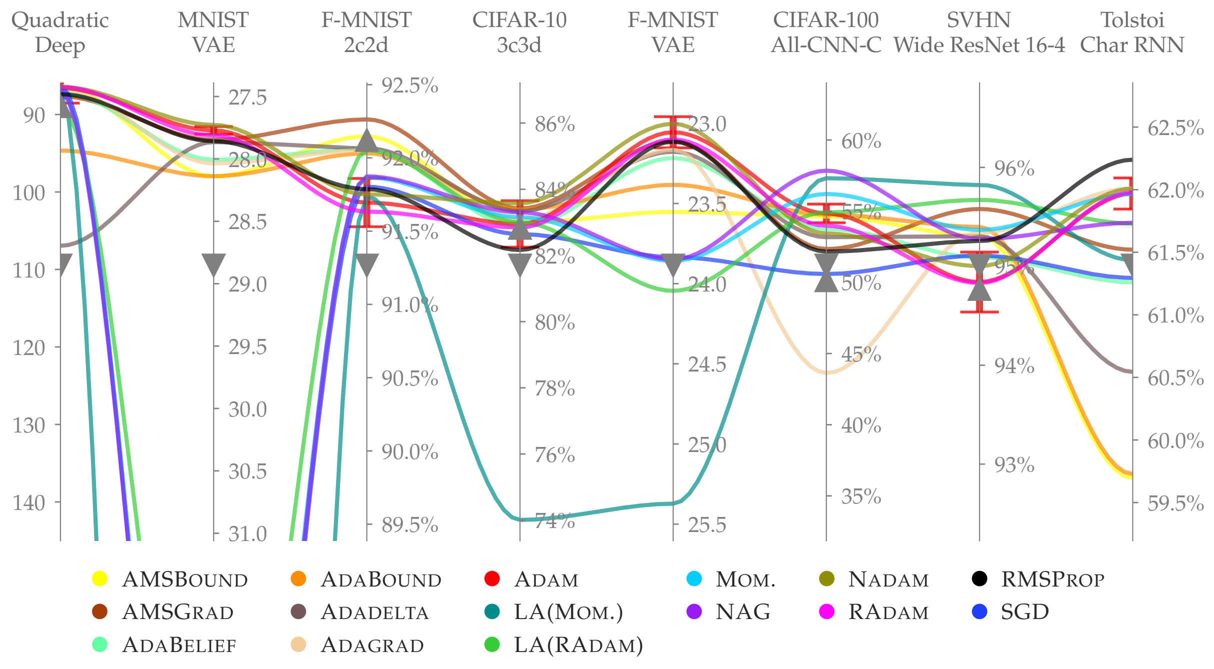


Figure 6.4: Mean test set performance over 10 random seeds of all tested optimizers on all eight optimization problems using the *large budget* for tuning and *no learning rate schedule*. One standard deviation for the *tuned* Adam optimizer is shown with a red error bar (I; error bars for other methods omitted for legibility). The performance of *untuned* Adam (▼) and AdaBound (▲) are marked for reference. The upper bound of each axis represents the best performance achieved in the benchmark, while the lower bound is chosen in relation to the performance of Adam with default parameters. Tabular version available in the Appendix as Table B.2.

Nevertheless, some optimizers generally perform well, while others can vary greatly in their behavior, most notably performing poorly on VAEs.

Further supporting the hypothesis of previous sections, we note that taking the best out of a small set of *untuned* optimizers — for example, Adam and AdaBound — frequently results in competitive performance. Except for the two VAE problems, the best of those two untuned optimizers generally falls within the distribution of the well-tuned methods. Combining these runs with a *tuned* version of Adam (or a variant thereof) provides stable and slightly improved results across many problems in our benchmark. To further increase the performance, our results suggest trying a different optimizer next, such as RMSProp or NAG. Across multiple budgets and schedules, both optimizers show a consistently good performance on the RNN (P8) and All-CNN-C (P6) model, respectively.

Nevertheless, achieving (or getting close to) the absolute best performance still requires testing numerous optimizers. Which optimizer wins in the end is problem-dependent: optimizers that achieve top scores on one problem can perform poorly on other tasks. We note in passing that the individual optimizer rankings changes when considering *e.g.* a smaller budget or an additional learning rate schedule (see Figures B.9 to B.11 in the appendix). However, the overall trends described here are consistent.

The idea that optimizers perform consistently better or worse for specific model architectures or tasks has been regularly theorized and mentioned in the literature. Indeed, our results support this hypothesis, with NAG often beating Adam on image classification tasks, and RMSProp being consistently on top for the natural language modeling task (see Tables B.2 to B.5). Understanding whether and why certain optimizers favor specific problem types presents an interesting research avenue and might lead to more sophisticated optimizers that utilize the problem characteristics.

6.4 Limitations of our Empirical Comparison

Any empirical benchmark has constraints and limitations. Here we highlight some of ours' and characterize the context within which our results should be considered.

Generalization of the results: By using the DeepOBS test problems, which span models and data sets of varying complexity and different domains, we aim for generalization. Our results are, despite our best efforts, reflective of not just these setups, but also of the chosen training parameters, the software framework, and further unavoidable choices. Our comparison's design aims to be close to what an informed practitioner would encounter for a relatively novel problem in practice. It goes without saying that even a carefully curated range of problems cannot cover all challenges of machine learning or even just deep learning. In particular, our conclusions may not generalize to other workloads such as GANs, reinforcement learning, or applications where *e.g.* memory usage is crucial.

[70] Deng et al. (2009), "ImageNet: A Large-Scale Hierarchical Image Database"
[307] Vaswani et al. (2017), "Attention Is All You Need"

Similarly, our benchmark does not cover more large-scale problems such as ImageNet [70] or transformer models [307]. While there is oft-mentioned anecdotal evidence that the characteristics of deep learning problems change for larger models, it would simply be impossible to perform the kind of combinatorial exploration covered in our benchmark, even with significant hardware resources. The inclusion of larger models would require reducing the number of tested optimizers, schedules or tuning methods and would thus shift the focus of the benchmark. Studying whether there are systematic differences between different types of deep learning problems presents an interesting avenue for further research.[6]

6: The MLCommons benchmark for algorithmic efficiency of training methods, described in Section 8.2.1, will consider larger-scale models, such as transformers. This is possible since the benchmark is held in the form of a competition where submitters have to decide, for instance which learning rate schedule they use, which reduces the combinatorial complexity.

We do not consider this study the definitive work on benchmarking deep learning optimizers, but rather an important and significant step in the right direction. While our comparison includes many "dimensions" of deep learning optimization, *e.g.* by considering different problems, tuning budgets, and learning rate schedules, there are certainly many more. To ensure that the benchmark remains feasible, we chose to use the fixed L^2 regularization and batch size that DeepOBS suggests for each problem. We also did not include optimization techniques such as weight averaging or ensemble methods as they can be combined with all evaluated optimizers and hence would increase the computational cost further. Future works could study how these techniques interact with different optimization methods. However, to keep our benchmark feasible, we have selected what we believe to be the most important aspects affecting an optimizer comparison. We hope, that our study lays the groundwork so that other works can build on it and analyze these questions.

Influence of the hyperparameter search strategy: As noted by, *e.g.*, Choi et al. [60] and Sivaprasad et al. [271], the hyperparameter tuning method, its budget, and its search domain, can significantly affect performance. By reporting results from four different hyperparameter optimization budgets (including the tuning-free one-shot setting) we try to quantify the effect of tuning. We argue that our random search process presents a realistic setting for many but certainly not all deep learning practitioners. One may criticize our approach as simplistic, but note that more elaborate schemes, in particular Bayesian optimization, would multiply the number of design decisions (kernels, search utilities, priors, etc.) and thus significantly complicate the analysis.

[60] Choi et al. (2019), "On Empirical Comparisons of Optimizers for Deep Learning"

[271] Sivaprasad et al. (2020), "Optimizer Benchmarking Needs to Account for Hyperparameter Tuning"

The individual hyperparameter sampling distributions significantly affect the relative rankings of the optimizers. A poorly chosen search space can make successful tuning next to impossible. In our benchmark, we use relatively broad initial search spaces, dozens of tuning runs and a refining of those search spaces for the large budget. Note, though, that the problem of finding appropriate search spaces is inherited by practitioners. It is arguably an implicit flaw of an optimization method that expects hyperparameter tuning not to come with well-identified search spaces for those parameters and this should thus be reflected in a benchmark.

6.5 Conclusion

Faced with an avalanche of research developing new stochastic optimizers, practitioners are left with the near-impossible task of not just picking a method from this ever-growing list, but also to guess or tune hyperparameters for them, even to continuously tune them during training. Despite efforts by the community, there is currently no method that clearly dominates the competition.

We have provided an extensive empirical benchmark of optimization methods for deep learning. It reveals structure in the crowded field of training methods for deep learning: First, although many methods perform competitively, a subset of methods tends to come up near the top across the spectrum of problems. Despite years of new research by many committed authors, Adam remains a viable (but also not a clearly superior) choice for many problems, with NAG or RMSProp being interesting alternatives that were able to boost performance on individual problems. Secondly, tuning helps about as much as trying other optimizers. Our open and extendable data set allows many, more technical observations, for example, that the stability to re-runs is an often overlooked challenge.

Perhaps the most important takeaway from our study is hidden in plain sight: the field is in danger of being drowned by noise. Different optimizers exhibit a surprisingly similar performance distribution compared to a single method that is re-tuned or simply re-run with different random seeds. It is thus questionable how much insight the development of new methods yields, at least if they are conceptually and functionally close to the existing population. We hope that benchmarks like ours can help the community to move beyond inventing yet another optimizer and to focus on key challenges, such as automatic, inner-loop tuning for truly robust and efficient optimization. We are releasing our data to allow future authors to ensure that their method contributes to such ends.

A Practical Debugging Tool for the Training of Deep Neural Networks 7

When engineers train deep learning models, they are very much "flying blind". Commonly used methods for real-time training diagnostics, such as monitoring the training or test loss, are limited (see Section 7.1). Assessing a network's training process solely through these performance indicators is akin to debugging software without access to internal states through a debugger. To address this, we present Cockpit, a collection of instruments (see Section 7.2) that enable a closer look into the inner workings of a learning machine, and a more informative and meaningful status report for practitioners. It facilitates the identification of learning phases and failure modes, such as ill-chosen hyperparameters or model inefficiencies (see Sections 7.3 and 7.4). These instruments leverage novel higher-order information about the gradient distribution and curvature, which has only recently become efficiently accessible (see Section 7.5). We believe that such a debugging tool, which we provide open source[1] for PyTorch [228], is a valuable help in troubleshooting the training process. By revealing new insights, it also more generally contributes to explainability and interpretability of deep neural networks. The contents of this chapter are mostly based on publication [263].

1: Cockpit is available at: `https://github.com/f-dangel/cockpit`

[228] Paszke et al. (2019), "PyTorch: An Imperative Style, High-Performance Deep Learning Library"

[263] Schneider et al. (2021), "Cockpit: A Practical Debugging Tool for the Training of Deep Neural Networks"

7.1 Why We Need a New Type of Debugger

Deep learning represents a new programming paradigm: Instead of deterministic programs, users design models and "simply" train them with data. In this metaphor, deep learning is a meta-programming form, where *coding* is replaced by *training*. In this chapter, we ponder the question how we can provide more insight into this training process by building a *debugger* specifically designed for the challenges of deep learning.

Debuggers are crucial tools for traditional software development. When things fail, they provide access to the internal workings of the code, allowing a look "into the box". This is much more efficient than re-running the program with different inputs. And yet, deep learning is arguably closer to the latter. If the attempt to train a deep network fails, a machine learning engineer faces various options: Should they adjust the training hyperparameters (how?); change the optimizer (to which one?); alter the model (how?); or just re-run with a different seed? Machine learning toolboxes provide scant help to guide these decisions.

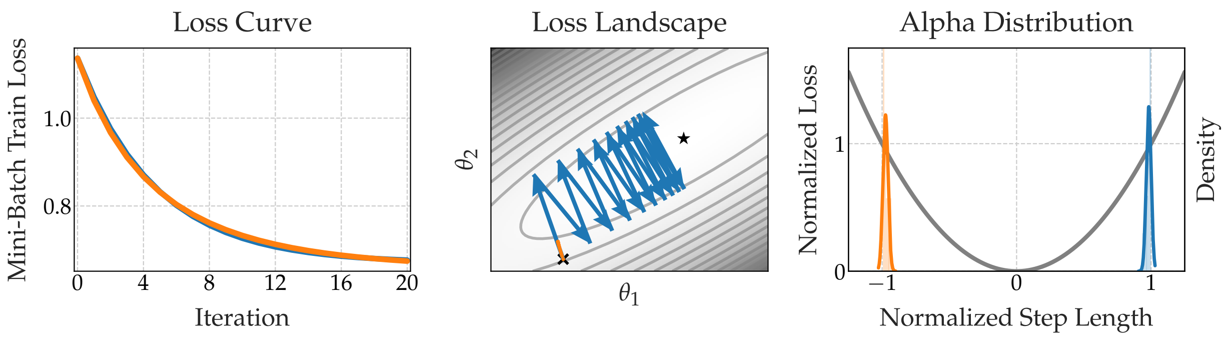


Figure 7.1: Illustrative example: Learning curves do not tell the whole story. Two different optimization runs (▬/▬) can lead to virtually the same loss curve (*left*). However, the actual optimization trajectories (*center*), exhibit vastly different behaviors. In practice, the trajectories are intractably high-dimensional and cannot be visualized directly. Recommendable actions for both scenarios (increase/decrease the learning rate) cannot be inferred from the loss curve alone. The `Alpha` distribution, one of Cockpit's instruments (*right*), not only clearly distinguishes the two scenarios, but also allows for taking decisions regarding how the learning rate should be adapted. See Section 7.3.3 for further details.

Of course, traditional debuggers can be applied to deep learning. They will give access to every single weight of a neural network in each iteration, or the individual pixels of its training data. But this rarely yields insights towards successful training. Extracting meaningful information requires a statistical approach and distillation of the bewildering complexity into a manageable summary. Tools like TensorBoard [1] or Weights & Biases [30] were built in part to streamline this visualization. Yet, the quantities that are widely monitored (mainly loss or accuracy on the training or validation set), provide only scant explanation for relative differences between multiple training runs, because *they do not show the network's internal state*. Figure 7.1 illustrates how such established learning curves can describe the *current* state of the model – whether it is performing well or not – while failing to inform about training state and dynamics. They tell the user *that* things are going well or badly, but not *why*. The situation is similar to flying a plane by sight, without instruments to provide feedback. It is not surprising, then, that achieving state-of-the-art performance in deep learning requires expert intuition, or plain trial & error.

[1] Abadi et al. (2015), "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems"

[30] Biewald (2020), "Experiment Tracking with Weights and Biases"

We aim to enrich the deep learning pipeline with a visual and statistical debugging tool that uses newly proposed observables as well as several established ones (Section 7.2). We leverage and augment recent extensions to automatic differentiation (*i.e.* BackPACK [66] for PyTorch [228]) to efficiently access second-order statistical (*e.g.* gradient variances) and geometric (*e.g.* Hessian) information. We show how these quantities can aid the deep learning engineer in tasks, like learning rate selection, as well as detecting common bugs with data processing or model architectures (Section 7.3).

[66] Dangel et al. (2020), "BackPACK: Packing more into Backprop"

[228] Paszke et al. (2019), "PyTorch: An Imperative Style, High-Performance Deep Learning Library"

Concretely, we introduce Cockpit, a flexible and efficient framework for online-monitoring these observables during training in carefully designed plots we call "instruments" (see Figure 7.2). To

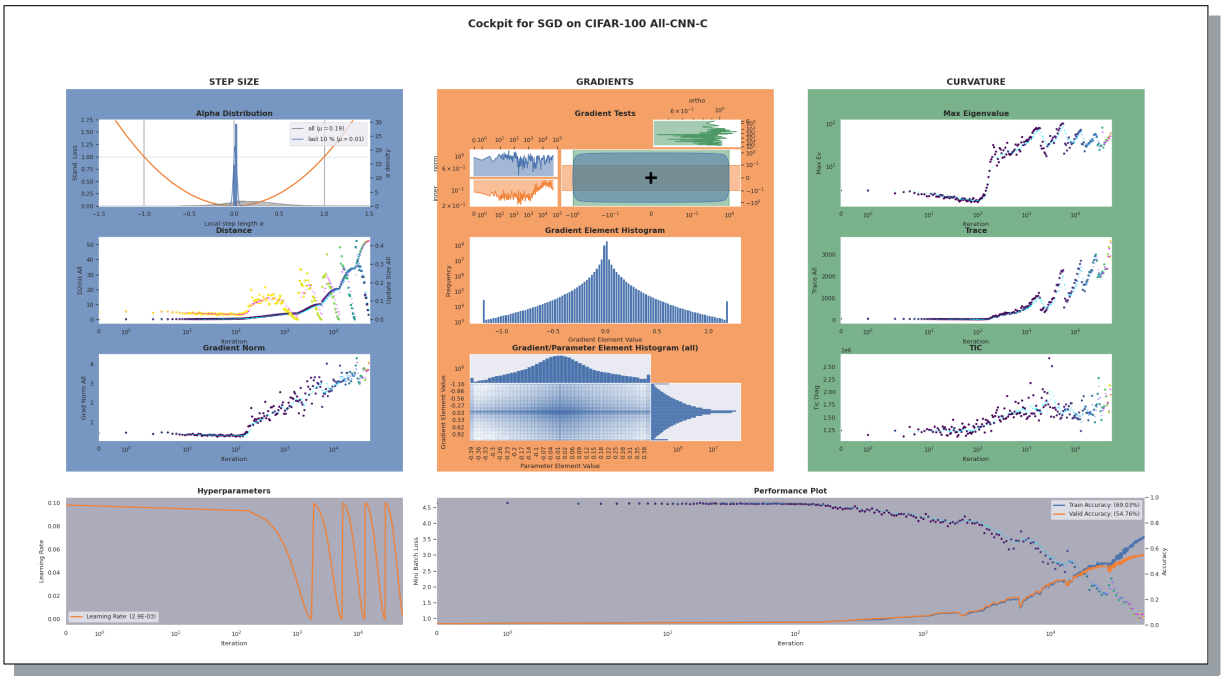


Figure 7.2: **Screenshot of Cockpit's full view** while training the All-CNN-C [277] on CIFAR-100 (P6 in Appendix A.1) with SGD using a cyclical learning rate schedule. This figure and its labels are not meant to be legible, but rather give an impression of Cockpit's user experience. Gray panels (bottom row) show the information currently tracked by most practitioners. The individual instruments are discussed in Section 7.2, and observations are described in Section 7.4. An animated version can be found in the accompanying GitHub repository.

be of practical use, such visualization must have a manageable computational overhead. We show that Cockpit scales well to real-world deep learning problems (see Figure 7.2 and Section 7.4). We also provide three different configurations of varying computational complexity and demonstrate that their instruments keep the computational cost *well below* a factor of 2 in runtime (Section 7.5). Cockpit scales well to real-world deep learning problems (see Figure 7.2 and Section 7.4). It is available as open-source code,[2] extendable, and seamlessly integrates into existing PyTorch training loops (see Algorithm C.1).

2: Package available at: `https://github.com/f-dangel/cockpit`

[277] Springenberg et al. (2015), "Striving for simplicity: The all convolutional net"

7.2 Cockpit's Instruments

Setting: We consider the supervised learning task as described in Section 2.1 and formalized in Section 3.2. In summary, one would like to minimize an inaccessible expected risk $L_{P_{\text{true}}}(\boldsymbol{\theta}) = \int \ell(f(\boldsymbol{x};\boldsymbol{\theta}), y)\, \mathrm{d}P_{\text{true}}(\boldsymbol{x}, y)$ via the empirical approximation $L_{\mathbb{D}_{\text{train}}}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \ell(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), y^{(i)})$.[3] In practice, only stochastically sub-sampled mini-batches $\mathbb{B} = \left\{(\boldsymbol{x}^{(1)}, y^{(1)}), \dots, (\boldsymbol{x}^{(B)}, y^{(B)})\right\} \subseteq \mathbb{D}_{\text{train}}$ are used to compute mini-batch approximations of the loss $L_{\mathbb{B}}(\boldsymbol{\theta}) = \frac{1}{B}\sum_{i=1}^{B} \ell^{(i)}$, the gradient $\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) = \frac{1}{B}\sum_{i=1}^{B} \boldsymbol{g}_{\mathbb{B}}^{(i)}$ and the Hessian $\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}) = \frac{1}{B}\sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}}^{2} \ell^{(i)}$.

3: Here, we define the risk in terms of the *parameters* $L(\boldsymbol{\theta})$ instead of the model function $L(f_{\boldsymbol{\theta}})$. As described in Section 3.2, both descriptions are analogous views.

Design choices: To minimize computational and design overhead, we restrict the metrics to quantities that require no additional model evaluations. This means that, at training step $t \to t+1$ with mini-batches $\mathbb{B}^{(t)}, \mathbb{B}^{(t+1)}$ and parameters $\boldsymbol{\theta}^{(t)}, \boldsymbol{\theta}^{(t+1)}$, we may access information about the mini-batch losses $L_{\mathbb{B}^{(t)}}(\boldsymbol{\theta}^{(t)})$ and $L_{\mathbb{B}^{(t+1)}}(\boldsymbol{\theta}^{(t+1)})$, but no cross-terms that would require additional forward passes.

Key point: $L_{\mathbb{B}}(\boldsymbol{\theta})$, $\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})$, and $\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})$ are just expected values of a *distribution* over the batch, see Section 3.2.1. Only recently, this distribution has begun to attract attention [82] as its computation has become more accessible [39, 66]. Contemporary optimizers leverage only the *mean* gradient and neglect higher moments. One core point of this chapter is making extensive use of these distribution properties, trying to visualize them in various ways. This out-of-the-box support for the carefully selected and efficiently computed quantities distinguishes Cockpit from tools like TensorBoard that offer visualizations as well.[4] Leveraging these distributional quantities, we create instruments and show how they can help adapt hyperparameters (Section 7.2.1), analyze the loss landscape (Section 7.2.2), and track network dynamics (Section 7.2.3). Instruments can sometimes be built from already-computed information or are efficient variants of previously proposed observables. To keep the presentation concise, we highlight the instruments shown in Figure 7.2 and listed in Table 7.1. Appendix C.3 defines them formally and contains more extensions, such as the mean GSNR [187], the early stopping [199] and CABS [21] criterion, which can all be used in Cockpit.

[82] Faghri et al. (2020), "A Study of Gradient Variance in Deep Learning"

[39] Bradbury et al. (2018), "JAX: composable transformations of Python+NumPy programs"

[66] Dangel et al. (2020), "BackPACK: Packing more into Backprop"

4: In TensorBoard, arbitrary quantities can be tracked and visualized. But *what* quantities are tracked and how they are computed is up to the user. With Cockpit, we offer a predefined set of meaningful quantities, provided through efficient computation.

[187] Liu et al. (2020), "Understanding Why Neural Networks Generalize Well Through GSNR of Parameters"

[199] Mahsereci et al. (2017), "Early Stopping without a Validation Set"

[21] Balles et al. (2017), "Coupling Adaptive Batch Sizes with Learning Rates"

Bug types: We distinguish between three types of bugs encountered in deep learning. *Implementation bugs* are low-level software bugs that, for example, trigger syntax errors. *Training bugs* result in unnecessarily inefficient or even unsuccessful training. They can, for example, stem from erroneous data handling (see Section 7.3.1), the chosen model architecture (see Section 7.3.2), or ill-chosen hyperparameters (see Section 7.3.3). *Prediction bugs* describe incorrect predictions of a trained model on specific examples. Traditional debuggers are well-suited to find implementation bugs. Cockpit focuses on efficiently identifying training bugs instead.

[43] Byrd et al. (2012), "Sample Size Selection in Optimization Methods for Machine Learning"

[32] Bollapragada et al. (2017), "Adaptive Sampling Strategies for Stochastic Optimization"

[291] Thomas et al. (2020), "On the interplay between noise and curvature and its effect on optimization and generalization"

[333] Yao et al. (2020), "ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning"

7.2.1 Adapting Hyperparameters

One big challenge in deep learning is setting the hyperparameters correctly, which is currently mostly done by trial and error through parameter searches. We aim to augment this process with instruments that inform the user about the effect that the chosen parameters have on the current training process.

Table 7.1: Overview of Cockpit quantities. They range from cheap byproducts, to non-linear transformations of first-order information and Hessian-based measures. Some quantities have already been proposed, others are first to be considered in this work. They are categorized into configurations *economy* $\subseteq$ *business* $\subseteq$ *full* based on their runtime overhead (see Section 7.5 for a detailed evaluation).

Name	Short Description	Config	Pos. in Figure 7.2
`Alpha`	Normalized step size on a noisy quadratic interpolation between two iterates $\boldsymbol{\theta}^{(t)}, \boldsymbol{\theta}^{(t+1)}$	*economy*	top left
`Distance`	Distance from initialization $\|\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(0)}\|_2$	*economy*	middle left
`UpdateSize`	Update size $\|\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}\|_2$	*economy*	middle left
`GradNorm`	Mini-batch gradient norm $\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\|_2$	*economy*	bottom left
`NormTest`	Normalized fluctuations of the residual norms $\|\boldsymbol{g}_{\mathbb{B}} - \boldsymbol{g}_{\mathbb{B}}^{(i)}\|_2$, proposed in [43]	*economy*	top center
`InnerTest`	Normalized fluctuations of the $\boldsymbol{g}_{\mathbb{B}}^{(i)}$'s parallel components along $\boldsymbol{g}_{\mathbb{B}}$, proposed in [32]	*economy*	top center
`OrthoTest`	Same as `InnerTest` but using the orthogonal components, proposed in [32]	*economy*	top center
`GradHist1d`	Histogram of individual gradient elements, $\{\boldsymbol{g}_{\mathbb{B}}^{(i)}(\theta_j)\}_{i\in\mathbb{B}}^{j=1,\dots,D}$	*economy*	middle center
`TICDiag`	Relation between (diagonal) curvature and gradient noise, inspired by [291]	*business*	bottom right
`HessTrace`	Exact or approximate Hessian trace, $\operatorname{Tr}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$, inspired by [333]	*business*	middle right
`HessMaxEV`	Maximum Hessian eigenvalue, $\lambda_{\max}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$, inspired by [333]	*full*	top right
`GradHist2d`	Histogram of weights and individual gradient elements, $\{(\theta_j, \boldsymbol{g}_{\mathbb{B}}^{(i)}(\theta_j))\}_{i\in\mathbb{B}}^{j=1,\dots,D}$	*full*	bottom center

Alpha: Are we crossing the valley? Using individual loss and gradient observations at the start and end point of each iteration, we build a noise-informed univariate quadratic approximation along the step direction (*i.e.* the loss in the update direction as a one-dimensional function of the step size), and assess to which point on this parabola our optimizer moves. We standardize this value, calling it `Alpha` and denoting it by α, such that stepping to the valley-floor is assigned $\alpha = 0$, the starting point is at $\alpha = -1$ and updates to the point exactly opposite of the starting point have $\alpha = 1$ (see Appendix C.3.1 for a more detailed visual and mathematical description of α). Figure 7.1 illustrates the scenarios $\alpha = \pm 1$ and how monitoring the `Alpha` distribution (right panel) can help distinguish between two training runs with similar performance but distinct failure sources. By default, this Cockpit instrument shows the `Alpha` distribution for the last 10 % of training and the entire training process (*e.g.* top left plot in Figure 7.2). In Section 7.3.3 we demonstrate empirically that, counter-intuitively, it is generally *not* a good idea to choose the step size such that α is close to zero.

Distances: Are we making progress? Another way to discern the trajectories in Figure 7.1 is by measuring the L^2 *distance from initialization* [*e.g.*, 216] (`Distance`) and the *update size* [*e.g.*, 6, 88] (`UpdateSize`) in parameter space. Both are shown together in one Cockpit instrument (see also middle left plot in Figure 7.2) and are far larger for the blue line in Figure 7.1. These distance metrics are also able to disentangle different phases for the blue path: Using the same step size, the blue trajectory will continue to "jump back and forth" between the loss valley's walls but at some point will cease to make progress in the low curvature direction. During this "surfing of the walls", the `Distance` increases, ultimately though, it will stagnate, with the `UpdateSize` remaining non-zero, indicating diffusion. While the initial "surfing the wall"-phase benefits training (see Section 7.3.3), achieving stationarity may require adaptation once the optimizer reaches that diffusion.

[216] Nagarajan et al. (2019), "Generalization in Deep Networks: The Role of Distance from Initialization"
[6] Agrawal et al. (2020), "Investigating Learning in Deep Neural Networks using Layer-Wise Weight Change"
[88] Frankle et al. (2020), "The Early Phase of Neural Network Training"

Gradient norm: How steep is the wall? The `UpdateSize` will show that the orange trajectory is stuck. But why? Such slow-down can result from both a bad learning rate or from plateaus in the loss landscape. The *gradient norm* (`GradNorm`, bottom left panel in Figure 7.2) distinguishes these two causes.

Gradient tests: How noisy is the batch? The batch size B trades off gradient accuracy versus computational cost. Recently, adaptive sampling strategies based on testing geometric constraints between mean and individual gradients have been proposed [32, 43]. The *norm*, *inner product*, and *orthogonality tests* (`NormTest`, `InnerTest`, `OrthoTest`) use a standardized radius and two band widths (parallel and orthogonal to the gradient mean) that indicate how strongly individual gradients scatter around the mean. The original works use these values to adapt batch sizes. Instead, Cockpit combines all three tests into a single gauge (top center plot of Figure 7.2) using the standardized noise radius and band widths for visualization. These noise signals can be used to guide batch size adaptation on- and offline, or to probe the influence of gradient alignment on training speed [254] and generalization [48, 49, 187].

[32] Bollapragada et al. (2017), "Adaptive Sampling Strategies for Stochastic Optimization"
[43] Byrd et al. (2012), "Sample Size Selection in Optimization Methods for Machine Learning"
[254] Sankararaman et al. (2020), "The Impact of Neural Network Overparameterization on Gradient Confusion and Stochastic Gradient Descent"
[48] Chatterjee (2020), "Coherent Gradients: An Approach to Understanding Generalization in Gradient Descent-based Optimization"
[49] Chatterjee et al. (2020), "Making Coherence Out of Nothing At All: Measuring the Evolution of Gradient Alignment"
[187] Liu et al. (2020), "Understanding Why Neural Networks Generalize Well Through GSNR of Parameters"

7.2.2 Hessian Properties for Local Loss Geometry

An intuition for the local loss landscape helps in many ways. It can help diagnose whether training is stuck, to adapt the learning rate, and explain stability or regularization properties [96, 145]. The key challenge lies in the large number of parameters: Low-dimensional projections of surfaces can behave unintuitively [212], but tracking the extreme or average behaviors may help in debugging, especially if first-order metrics fail.

[96] Ginsburg (2020), "On regularization of gradient descent, layer imbalance and flat minima"
[145] Jastrzebski et al. (2020), "The Break-Even Point on the Optimization Trajectories of Deep Neural Networks"
[212] Mulayoff et al. (2020), "Unique Properties of Flat Minima in Deep Networks"

Hessian eigenvalues: A gorge or a lake? In convex optimization, the maximum Hessian eigenvalue crucially determines the appropriate learning rate [260]. Many works have studied the Hessian spectrum in machine learning [*e.g.*, 95, 96, 212, 250, 251, 333]. In short: curvature matters. Established [229] and recent autodiff frameworks [66] can compute Hessian properties without requiring the full matrix. Cockpit leverages this to provide the *Hessian's largest eigenvalue* and *trace* (`HessMaxEV` and `HessTrace`, top right and middle right plots in Figure 7.2). The former resembles the loss surface's sharpest valley and can thus hint at training instabilities [145]. `HessTrace` describes a notion of "average curvature", since the eigenvalues λ_i relate to it by $\sum_i \lambda_i = \operatorname{Tr}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$, which might correlate with generalization [144].

[260] Schmidt (2014), "Convergence rate of stochastic gradient with constant step size"

[250] Sagun et al. (2016), "Eigenvalues of the Hessian in Deep Learning: Singularity and Beyond"

[251] Sagun et al. (2017), "Empirical Analysis of the Hessian of Over-Parametrized Neural Networks"

[95] Ghorbani et al. (2019), "An Investigation into Neural Net Optimization via Hessian Eigenvalue Density"

[96] Ginsburg (2020), "On regularization of gradient descent, layer imbalance and flat minima"

[212] Mulayoff et al. (2020), "Unique Properties of Flat Minima in Deep Networks"

[333] Yao et al. (2020), "ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning"

[229] Pearlmutter (1994), "Fast Exact Multiplication by the Hessian"

[66] Dangel et al. (2020), "BackPACK: Packing more into Backprop"

[145] Jastrzebski et al. (2020), "The Break-Even Point on the Optimization Trajectories of Deep Neural Networks"

[144] Jastrzebski et al. (2020), "Catastrophic Fisher Explosion: Early Phase Fisher Matrix Impacts Generalization"

TIC: How do curvature and gradient noise interact? There is an ongoing debate about curvature's link to generalization [*e.g.*, 75, 127, 163]. The *Takeuchi Information Criterion (TIC)* [286, 291] estimates the generalization gap by a ratio between Hessian and non-central second gradient moment. It also provides intuition for changes in the objective function implied by gradient noise. Inspired by the approximations in [291], Cockpit provides mini-batch TIC estimates (`TICDiag` or `TICTrace`, bottom right plot of Figure 7.2).

[75] Dinh et al. (2017), "Sharp Minima Can Generalize For Deep Nets"

[127] Hochreiter et al. (1997), "Long Short-Term Memory"

[163] Keskar et al. (2017), "Improving Generalization Performance by Switching from Adam to SGD"

[286] Takeuchi (1976), "The distribution of information statistics and the criterion of goodness of fit of models"

[291] Thomas et al. (2020), "On the interplay between noise and curvature and its effect on optimization and generalization"

[291] Thomas et al. (2020), "On the interplay between noise and curvature and its effect on optimization and generalization"

7.2.3 Visualizing Internal Network Dynamics

Histograms provide a natural visual compression of the high-dimensional $B \times D$ individual gradient values. They give insights into the gradient *distribution* and hence offer a more detailed view of the learning signal. Together with the parameter associated to each individual gradient, the entire model status and dynamics can be visualized in a single plot and be monitored during training. This provides a more fine-grained view of training compared to tracking parameters and gradient norms [88].

[88] Frankle et al. (2020), "The Early Phase of Neural Network Training"

Gradient and parameter histograms: What is happening in our network? Cockpit offers `GradHist1d`, a univariate *histogram of the gradient elements* $\{\boldsymbol{g}_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})_j\}_{i\in\mathbb{B}}^{j=1,\dots,D}$. Additionally, `GradHist2d`, a combined *histogram of parameter-gradient pairs* $\{(\theta_j, \boldsymbol{g}_{\mathbb{B}}^{(i)}(\theta_j))\}_{i\in\mathbb{B}}^{j=1,\dots,D}$ provides a two-dimensional look into the network's gradient and parameter values in a mini-batch. Section 7.3.1 shows an example use-case of the gradient histogram; Section 7.3.2 makes the case for the layer-wise variants of the instruments.

7.3 Experiments: Identifying Training Bugs with Cockpit

The diverse information provided by Cockpit can help users and researchers in many ways, some of which, just like for a traditional debugger, only become apparent in practical use. In this section, we present a few motivating examples, selecting specific instruments and scenarios in which they are practically useful. Specifically, we show that Cockpit can help the user discern between, and thus fix, common training bugs (Sections 7.3.1 and 7.3.2) that are otherwise hard to distinguish as they lead to the same failure: bad training. We demonstrate that Cockpit can guide practitioners to choose efficient hyperparameters *within a single training run* (Sections 7.3.2 and 7.3.3). Finally, we highlight that Cockpit's instruments can provide research insights about the optimization process (Section 7.3.3). Our empirical findings are demonstrated on test problems from the DeepOBS benchmark collection, presented in Chapter 5 and [262].

[262] Schneider et al. (2019), "DeepOBS: A Deep Learning Optimizer Benchmark Suite"

7.3.1 Incorrectly Scaled Data

One prominent source of bugs is the data pipeline. To pick a relatively simple example: For standard optimizers to work at their usual learning rates, network inputs must be standardized (*i.e.* between zero and one, or have zero mean and unit variance [*e.g.*, 25]). If the user forgets to do this, optimizer performance is likely to degrade. It can be difficult to identify the source of this problem as it does not cause obvious failures, such as `NaN` or `Inf` gradients. We now construct a semi-realistic example, to show how using Cockpit can help diagnose this problem upon observing slow training performance.

[25] Bengio (2012), "Practical recommendations for gradient-based training of deep architectures"

By default[5], the popular image data sets CIFAR-10/100 [169] are provided as NumPy [113] arrays that consist of integers in the interval $[0, 255]$. This *raw* data, instead of the widely used version with floats in $[0, 1]$, changes the data scale by a factor of 255 and thus the gradients as well. Therefore, the optimizer's optimal learning rate is scaled as well. In other words, the default parameters of popular optimization methods may not work well anymore, or good hyperparameters may take extreme values. Even if the user directly inspects the training images, this may not be apparent (see Figures 7.3 and C.3 in the appendix for the same experiment with the VGG16 network [270] on ImageNet [70]). But the gradient histogram instrument of Cockpit, which has a deliberate default plotting range around $[-1, 1]$ to highlight such problems, immediately and prominently shows that there is an issue.

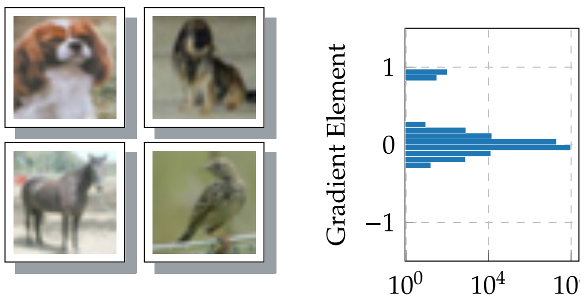
5: As available, for example, at `https://www.cs.toronto.edu/~kriz/cifar.html`.

[169] Krizhevsky et al. (2009), "Learning multiple layers of features from tiny images"

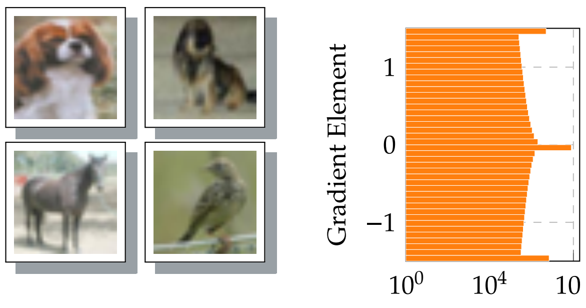
[113] Harris et al. (2020), "Array programming with NumPy"

[270] Simonyan et al. (2015), "Very Deep Convolutional Networks for Large-Scale Image Recognition"

[70] Deng et al. (2009), "ImageNet: A Large-Scale Hierarchical Image Database"



(a) Normalized Data



(b) Raw Data

Figure 7.3: Same inputs, different gradients; Catching data bugs with Cockpit. (a) *normalized* ($[0, 1]$) and (b) *raw* ($[0, 255]$) images look identical in auto-scaled front-ends like matplotlib's `imshow`. The gradient distribution of the 3c3d model from DeepOBS (P4 in Appendix A.1), however, is crucially affected by this scaling.

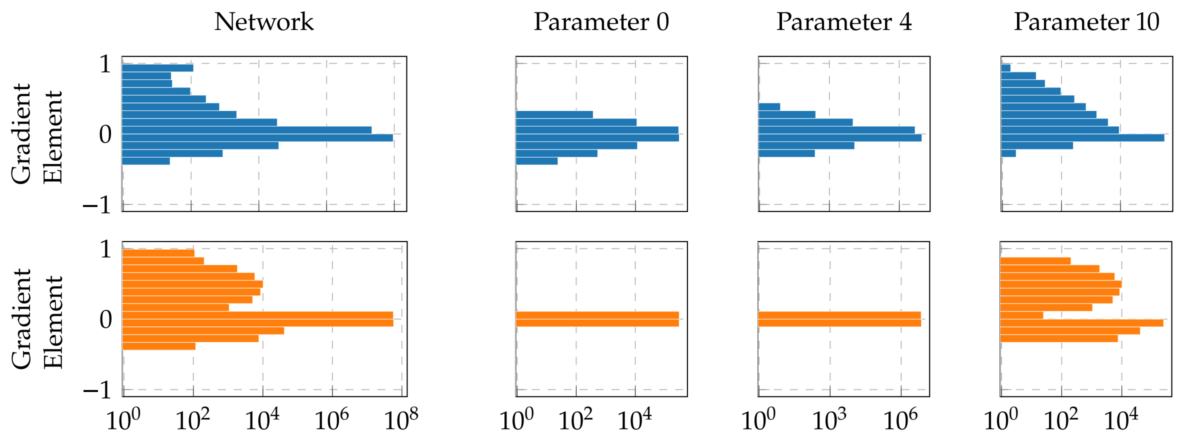
Of course, this particular data is only a placeholder for real practical data sets. While this problem may not frequently arise in the highly pre-processed and readily packaged CIFAR-10, it is not a rare problem for practitioners who work with their personal data sets. This is particularly likely in domains outside standard computer vision, *e.g.* when working with mixed-type data without obvious natural scales.

7.3.2 Vanishing Gradients

The model architecture itself can be a source of training bugs. As before, such problems mostly arise with novel data sets, where well-working architectures are still unknown. The following example shows how even a small (in terms of code) architectural modification can severely harm the training.

Figure 7.4a shows the distribution of gradient values of two different network architectures in blue and orange. Although the blue model trains considerably better than the orange one, their gradient distributions look relatively similar. The difference between the two models becomes evident when inspecting the histogram *layer-wise*. We can see that multiple layers have a degenerated gradient distribution with many elements being practically zero (see Figure 7.4b, bottom row). Since the fully connected layers close to the output have far more parameters (a typical pattern of convolutional networks), they dominate the network-wide histogram. This obscures the fact that a major part of the model is effectively unable to train.

Both the blue and orange networks follow DeepOBS's 3c3d architecture (see P4 in Appendix A.1). The only difference is the non-linearity: The blue network uses the common ReLU activation function, while the orange one has sigmoid activations (see Section 4.2.2 for a description of popular deep learning activation



(a) Network Histogram **(b)** Layer-wise Histograms

Figure 7.4: Gradient distributions of two similar architectures on the same problem. (a) Distribution of individual gradient elements summarized over the entire network. Both networks seem similar. (b) Layer-wise histograms for a subset of layers. Parameter 0 is the layer closest to the network's input, parameter 10 closest to its output. Only the layer-wise view reveals that there are several layers with a degenerated gradient distribution for the orange network making training unnecessary hard.

functions). In this experiment, the layer-wise histogram instrument of Cockpit highlights which part of the architecture makes training unnecessarily hard. Accessing information layer-wise is also essential due to the strong overparameterization in deep models where training can happen in small subspaces [111]. Once again, this is hard to do with common monitoring tools, such as the popular learning curves.

[111] Gur-Ari et al. (2018), "Gradient Descent Happens in a Tiny Subspace"

7.3.3 Tuning Learning Rates

Once the architecture is defined, the optimizer's learning rate is the most important hyperparameter to tune. Getting it right requires extensive hyperparameter searches at high resource costs. Cockpit's instruments can provide intuition and information to streamline this process: In contrast to the raw learning rate, the curvature-standardized step size `Alpha` quantity (see Section 7.2.1) has a natural scale.

Across multiple optimization problems (P4, P10, P11, and P12 in Appendix A.1), we observe, perhaps surprisingly, that the best runs and indeed all good runs have a median $\alpha > 0$ (Figure 7.5). This illustrates a fundamental difference between stochastic optimization, as is typical for machine learning, and classic deterministic optimization. Instead of locally stepping "to the valley floor" (optimal in the deterministic case), stochastic optimizers should *overshoot* the valley somewhat. This need to "surf the walls" has been hypothesized before [*e.g.*, 324, 330] as a property of neural network training. Frequently, learning rates are adapted during training,

[324] Wu et al. (2018), "Understanding short-horizon bias in stochastic meta-optimization"

[330] Xing et al. (2018), "A Walk with SGD"

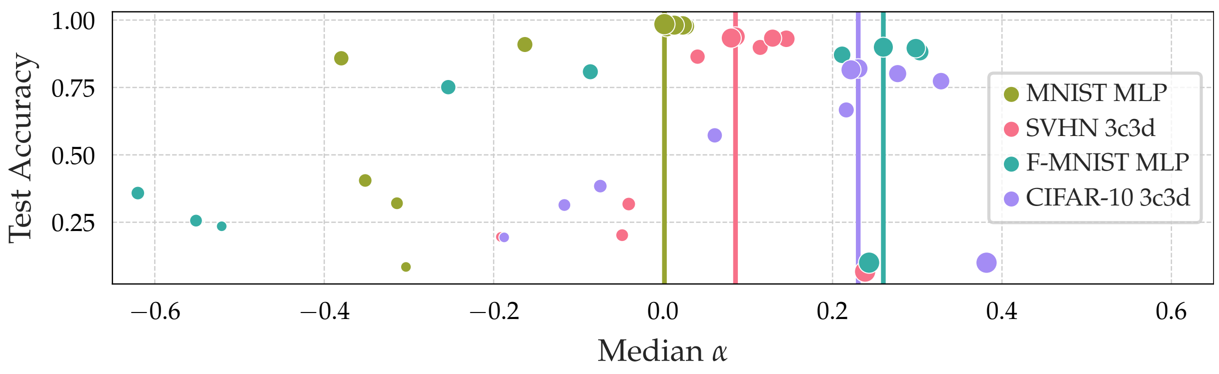


Figure 7.5: Test accuracy as a function of standardized step size `Alpha`. For four DeepOBS problems (see Appendix A.1 for details on the problems), final test accuracy is shown versus the median α-value over the entire training. Marker size indicates the magnitude of the raw learning rate, marker color identifies tasks (see legend). For each problem, the best-performing setting is highlighted by a vertical colored line.

which fits with our observation about positive α-values: "Overshooting" allows fast early progression towards areas of lower loss, but it does not yield convergence in the end. Real-time visualizations of the training state, as offered by Cockpit, can augment these fine-tuning processes.

Figure 7.5 also indicates a major challenge preventing simple automated tuning solutions: The optimal α-value is problem-dependent, and simpler problems, such as the multi-layer perceptron (MLP) on MNIST [177] (P10 in Appendix A.1), behave much more similar to classic optimization problems. Algorithmic research on small problems can thus produce misleading conclusions. The figure also shows that the `Alpha` gauge is not sufficient by itself: extreme overshooting with a too-large learning rate leads to poor performance, which however can be prevented by taking additional instruments into account. This makes the case for the cockpit metaphor of increasing interpretability from several instruments in conjunction. By combining the `Alpha` instrument with other gauges that capture the local geometry or network dynamics, the user can better identify good choices of the learning rate and other hyperparameters.

[177] LeCun et al. (1998), "Gradient-Based Learning Applied to Document Recognition"

7.4 Showcasing Cockpit

Having introduced the tool, we can now return to Figure 7.2 for a closer look. The figure shows a snapshot from training the All-CNN-C [277] on CIFAR-100 (P6 in Appendix A.1) using SGD with a cyclic learning rate schedule (see bottom left panel). Diagonal curvature instruments are configured to use an MC approximation in order to reduce the runtime (here, $C = 100$, compare Section 7.5).

[277] Springenberg et al. (2015), "Striving for simplicity: The all convolutional net"

A glance at all panels shows that the learning rate schedule is reflected in the metrics. However, the instruments also provide insights into the early phase of training (first $\sim$ 100 iterations), where the learning rate is still unaffected by the schedule: There, the loss plateaus and the optimizer takes relatively small steps (compared to later, as can be seen in the small gradient norms, and small distance from initialization). Based on these low-cost instruments, one may thus at first suspect that training was poorly initialized; but training indeed succeeds after iteration 100! Viewing Cockpit entirely though, it becomes clear that optimization in these first steps is not stuck at all: While loss, gradient norms, and distance in parameter space remain almost constant, curvature changes, which expresses itself in a clear downward trend of the maximum Hessian eigenvalue (top right panel).

[88] Frankle et al. (2020), "The Early Phase of Neural Network Training"

The importance of early training phases has recently been hypothesized [88], suggesting a logarithmic timeline. Not only does our showcase support this hypothesis, but it also provides an explanation from the curvature-based metrics, which in this particular case are the only meaningful feedback in the first few training steps. It also suggests monitoring training at log-spaced intervals. Cockpit provides the flexibility to do so, indeed, Figure 7.2 has been created with log-scheduled tracking events.

[292] Thompson et al. (2020), "The Computational Limits of Deep Learning"

As a final note, we recognize that the approach taken here promotes an amount of *manual* work (monitoring metrics, deliberately intervening, *etc.*) that may seem ironic and at odds with the paradigm of automation that is at the heart of machine learning. However, we argue that this might be what is needed at this point in the evolution of the field. Deep learning has been driven notably by scaling compute resources [292], and fully automated, one-shot training may still be some way out. To develop better training methods, researchers, not just users, need *algorithmic* interpretability and explainability: direct insights and intuition about the processes taking place "inside" neural nets. To highlight how Cockpit might provide this, we contrast in Appendix C.6 the Cockpit view of two convex DeepOBS problems: a noisy quadratic (P1 in Appendix A.1) and logistic regression on MNIST (P9 in Appendix A.1). In both cases, the instruments behave differently compared to the deep learning problem in Figure 7.2. In particular, the gradient norm increases (left column, bottom panel) during training, and individual gradients become less scattered (center column, top panel). This is diametrically opposed to the convex problems and shows that deep learning differs even qualitatively from well-understood optimization problems.

7.5 Benchmarking Cockpit's Instruments

Section 7.3 made a case for Cockpit as an effective debugging and tuning tool. To make the library useful in practice, it must also have limited computational cost. We now show that it is possible to compute the proposed quantities at reasonable overhead. The user can control the absolute cost along two dimensions, by reducing the number of instruments, or by reducing their update frequency.

All benchmark results show SGD without momentum. Cockpit's quantities, however, work for generic optimizers and can mostly be used identically without increased costs. One current exception is `Alpha` which can be computed more efficiently if provided with the optimizer's update rule. [6]

6: This is currently implemented for vanilla SGD. For other optimizers, Cockpit falls back to a less efficient computation.

Complexity analysis: Computing more information adds computational overhead, of course. However, recent work [66] has shown that first-order information, like distributional statistics on the batch gradients, can be computed on top of the mean gradient at little extra cost. Similar savings apply for most quantities in Table 7.1, as they are (non-)linear transformations of individual gradients. A subset of Cockpit's quantities also uses second-order information from the Hessian diagonal. For ReLU networks on a classification task with C classes, the additional work is proportional to C gradient backpropagations (*i.e.* $C = 10$ for CIFAR-10, $C = 100$ for CIFAR-100). Parallel processing can, to some extent, process these extra backpropagations in parallel without significant overhead. If this is no longer possible, we can fall back to a Monte Carlo (MC) sampling approximation, which reduces the number of extra backprop passes to the number of samples (1 by default).[7]

[66] Dangel et al. (2020), "BackPACK: Packing more into Backprop"

7: An MC-sampled approximation of the Hessian/generalized Gauss-Newton has been used in Figure 7.2 to reduce the prohibitively large number of extra backprops on CIFAR-100 ($C = 100$).

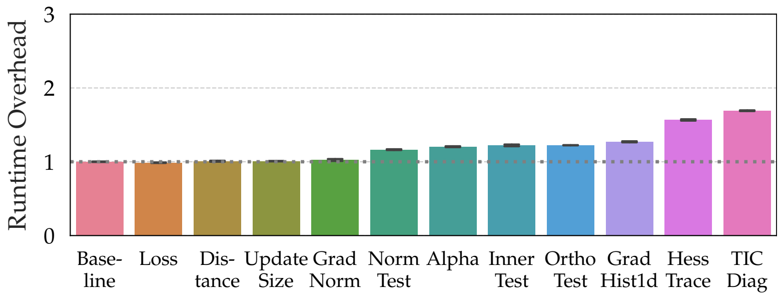
While parallelization is possible for the gradient instruments, computing the maximum Hessian eigenvalue is inherently sequential. Similar to Yao et al. [333], we use matrix-free Hessian-vector products by automatic differentiation [229], where each product's costs are proportional to one gradient computation. Regardless of the underlying iterative eigensolver, multiple such products must be queried to compute the spectral norm (the required number depends on the spectral gap to the second-largest eigenvalue).

[333] Yao et al. (2020), "ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning"

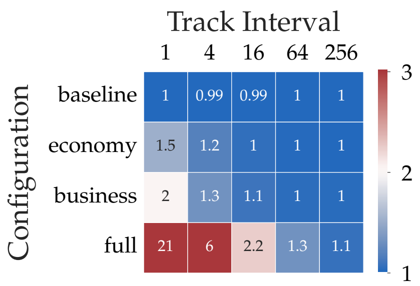
[229] Pearlmutter (1994), "Fast Exact Multiplication by the Hessian"

Runtime benchmark: Figure 7.6a shows the wall-clock computational overhead for individual instruments (details in Appendix C.5).[8] As expected, byproducts are virtually free, and quantities that rely solely on first-order information add little overhead (at most roughly 25 % on this problem). Thanks to parallelization, the ten extra backward passes required for Hessian quantities reduce to less than 100 % overhead. Individual overheads also do

8: To improve readability, we exclude `HessMaxEV` here, because its overhead is large compared to other quantities. Surprisingly, we also observed significant cost for the 2D histogram on GPU. It is caused by an implementation bottleneck for histogram shapes observed in deep models. We thus also omit `GradHist2d` here, as we expect it to be eliminated with future implementations (see Appendix C.5.2 for a detailed analysis and further benchmarks). Both quantities, however, are part of the benchmark shown in Figure 7.6b.



(a) Overhead Cockpit instruments



Configuration	1	4	16	64	256
baseline	1	0.99	0.99	1	1
economy	1.5	1.2	1	1	1
business	2	1.3	1.1	1	1
full	21	6	2.2	1.3	1.1

(b) Overhead Cockpit configurations

Figure 7.6: Runtime overhead for individual Cockpit instruments and configurations as shown on CIFAR-10 using the 3c3d network (P4 in Appendix A.1) on a GPU. (a) The runtime overheads for individual instruments are shown as multiples of the *baseline* (= no tracking). Most instruments add little overhead. This plot shows the overhead in one iteration, determined by averaging over multiple iterations and random seeds. (b) Overhead for different Cockpit configurations. Adjusting the tracking interval and re-using the computation shared by multiple instruments can make the overhead orders of magnitude smaller. Blue fields mark settings that allow tracking without doubling the training time.

not simply add up when multiple quantities are tracked, because quantities relying on the same information share computations.

To allow a rough cost control, Cockpit currently offers three configurations, called *economy*, *business*, and *full*, in increasing order of cost (cf. Table 7.1). As a basic guideline, we consider a factor of two to be an acceptable limit for the increase in training time and benchmark the configurations' runtimes for different tracking intervals. Figure 7.6b shows a runtime matrix for training the 3c3d problem of DeepOBS on CIFAR-10 (P4 in Appendix A.1), where settings that meet this limit are set in blue (more problems including ImageNet are shown in Appendix C.5). Speedups due to shared computations are easy to read off: Summing all the individual overheads shown in Figure 7.6a would result in a total overhead larger than 200 %, while the joint overhead for the *business* configuration reduces to 140 %. The *economy* configuration can easily be tracked at every step of this problem and stay well below our threshold of doubling the execution time. Cockpit's full view, shown in Figure 7.2, can be updated every 64-th iteration without a major increase in training time (this corresponds to about five updates per epoch). Finally, tracking any configuration about once per epoch – which is common in practice – adds overhead close to zero (rightmost column).

[66] Dangel et al. (2020), "BackPACK: Packing more into Backprop"

This good performance is largely due to the efficiency of the BackPACK package [66], which we leverage with custom and optimized modification, that compacts information layer-wise and then discards unneeded buffers. Using layer-wise information (Section 7.3.2) scales better to large networks, where storing the entire model's individual gradients all at once becomes increasingly expensive (see Appendix C.5). To the best of our knowledge, many of the quantities in Table 7.1, especially those relying on individual gradients, have only been explored on rather small problems. With Cockpit they can now be accessed at a reasonable rate for deep learning models outside the toy problem category.

7.6 Conclusion

Contemporary machine learning, in particular deep learning, remains a craft and an art. High dimensionality, stochasticity, and non-convexity require constant tracking and tuning, often resulting in a painful process of trial and error. When things fail, popular performance measures, like the training loss, do not provide enough information by themselves. These metrics only tell *whether* the model is learning, but not *why*. Alternatively, traditional debugging tools can provide access to individual weights and data. However, in models whose power only arises from possessing myriad parameters, this approach is hopeless, like looking for the proverbial needle in a haystack.

To mitigate this, we proposed Cockpit, a practical visual debugging tool for deep learning. It offers instruments to monitor the network's internal dynamics during training, in real-time. In its presentation, we focused on two crucial factors affecting the user experience: Firstly, such a debugger must provide meaningful insights. To demonstrate Cockpit's utility, we showed how it can identify bugs where traditional tools fail. Secondly, it must come at a feasible computational cost. Although Cockpit uses rich second-order information, efficient computation keeps the necessary runtime overhead cheap. The open-source PyTorch package can be added to many existing training loops.

Obviously, such a tool is never complete. Just like there is no perfect universal debugger, the list of current instruments is naturally incomplete. Further practical experience with the tool, for example in the form of a future larger user study, could provide additional evidence for its utility. However, our analysis shows that Cockpit provides useful tools and extracts valuable information presently not easily accessible to the user. We believe that this improves algorithmic interpretability – helping practitioners understand how to make their models work – but may also inspire new research. The code is designed flexibly, deliberately separating the computation and visualization. New instruments can be added easily and also be shown by the user's preferred visualization tool, *e.g.* TensorBoard. Of course, instead of just showing the data, the same information can be used by novel algorithms directly, side-stepping the human in the loop.

Part III

Conclusion & Outlook

Conclusion and Outlook 8

Optimization remains a central and critical component of modern machine learning. The first-order iterative optimization methods that lie at the heart of most neural network training, are often referred to as the "workhorses" of deep learning. Yet, we know surprisingly little about why these "workhorses" work and, more importantly, in which situations they fail. The choice of training method and, perhaps even more so, the choice of their hyperparameters can have such a significant effect on the performance achievable by contemporary machine learning and deep learning models. Unsurprisingly, there exist many tricks and heuristics to augment the unsatisfactory user experience of carefully setting these hyperparameters through extensive and costly trial-and-error searches or human intuition. To improve this unsatisfying status quo, we urgently need more insight into the training process of deep learning models. In this work, we aimed to improve the understanding of deep learning optimization by carefully comparing the existing methods and introducing novel debugging tools to open the black box of neural network training.

8.1 Summary

This work focused on addressing and improving the current state of optimization for deep learning by providing analysis, tools, and insights in three areas:

- In Chapter 5, we identified the need for a standardized and rigorous evaluation tool for optimization methods for deep learning. Although hundreds of novel algorithms have been suggested for deep learning, see Table 3.1, there is no agreed-upon protocol for systematic and reproducible evaluation and comparison of optimizers. The unique challenges of deep learning, such as the high-dimensionality of the parameter space, the stochasticity of the training process, or the need for hyperparameter tuning make it difficult to objectively quantify what constitutes a "better" optimizer. With **DeepOBS**, we presented a protocol and a practical tool that simplifies and automates this benchmarking process. It addresses the challenges of benchmarking deep learning optimizers, offering a more objective and quantitative evaluation of novel methods and thus a way to measure progress in the development of better training methods for deep learning.

- Using the knowledge gained from developing the evaluation tool DeepOBS, we **benchmarked** fifteen popular deep learning optimization methods in Chapter 6. In order to make this comparison as fair as possible, the methods had to be tested on multiple problems, with different hyperparameter settings and using various tuning budgets. A careful experimental design was necessary to keep the computational complexity within reasonable limits so that the benchmark remained feasible. Examining the approximately 50,000 resulting training runs, we found that, contrary to their claims, novel optimizers do not consistently and significantly outperform established methods. Comparatively traditional methods, such as the popular Adam optimizer, remain a viable, but also not superior, choice for many problems. SGD with Nesterov Accelerated Gradient (NAG) or RMSProp are interesting alternatives that were able to boost performance on individual problems.
- The use of machine learning involves a transformation from what has been called *Software 1.0*, *i.e.* the "classic" coding of explicit instructions, toward *Software 2.0*, *i.e.* the training of models that *learn* the strategies on their own to solve a task [157]. With it, the software development tools need to change as well toward more machine learning focused *MLOps* tools. Debuggers are an indispensable tool for traditional software development but have failed to co-evolve to contemporary machine learning. Traditional debuggers can display the exact value of each of the millions of model parameters or the pixel value of each and every training image, but this does not provide insight into why deep learning models fail to train and what changes are needed to fix this. In Chapter 7, we introduced **Cockpit**, a visual and statistical debugger specifically designed for deep learning. It provides a more meaningful status report compared to the learning curves and a much richer view into neural network training. As a showcase of Cockpit's capabilities, we identified that effective training runs consistently *overshoot* the local loss minimum. Cockpit also provides the ability to identify and disentangle of common failure modes in neural network training, such as incorrectly scaled data or inefficient model architectures.

[157] Karpathy (2017), "Software 2.0"

8.2 Future Work

The work presented in this thesis also provides interesting future research directions and natural extensions of the topics presented here. Smaller extensions of the individual works were already discussed in the individual chapters and here we want to focus on longer-term visions. Section 8.2.1 describes a current undertaking to further improve the state of benchmarking algorithms in deep learning. It can be seen as a continuation of the benchmark projects in Chapters 5 and 6. Section 8.2.2 presents a perspective for extending the deep learning debugging tool Cockpit, presented in Chapter 7. Section 8.2.3 outlines how the insights gained from benchmarking and building tools for debugging optimization methods for deep learning can be leveraged to develop new training algorithms. In particular, this section focuses on autonomous training methods that require less human interaction by removing hyperparameters such as the learning rate.

8.2.1 Algorithmic Efficiency Competition of MLCommons

Recently, the non-profit consortium MLCommons formed the *Algorithmic Efficiency Working Group* to develop the first truly community-wide standard for benchmarking training algorithms and models in deep learning.[1] In a collaborative effort, researchers from international artificial intelligence companies and universities, competitions will be held at regular intervals to evaluate submitted *training algorithms* and *models*. The focus will be on measuring speedups in neural network training that can be achieved by modifying the underlying algorithms. Both the hardware and the underlying software framework will be kept fixed to isolate the improvements resulting from changes to the algorithms. The training algorithms and models are compared in separate tracks of the competition to avoid highly specific solutions. Instead, the goal is to identify and encourage the development of widely useful algorithms that work well on general problems.

1: At the time of writing, the author of this thesis is one of the two elected chairs for this working group.

Compared to the work presented in Chapters 5 and 6, MLCommons uses a more encompassing view of what constitutes a training algorithm. This not only includes the optimizer, *i.e.* the update rule, but also the search space for hyperparameter tuning or the data selection process. By conducting a competition, MLCommons avoids some of the complexity that arises when comparing training methods in deep learning. For instance, the burden of selecting search spaces is shifted to the submitters who must decide which specific instance of their algorithm produces the best results. In this way, the competition can make a fairer comparison between algorithms. Due to the collaborative effort of multiple researchers, it is possible to include more diverse and larger-scale test problems from different scientific fields compared to DeepOBS.

The participation of both industry and academic institutions in the working group will allow MLCommons to have wide adoption and visibility. The insights gained from developing DeepOBS and benchmarking deep learning optimizers will help in this endeavor. MLCommons has the opportunity not only to stimulate new research in algorithms for neural network training but also to rigorously measure the progress that the field has made so far. It thus represents a natural continuation of the work presented here.

8.2.2 A Vision for Cockpit 2.0

Cockpit, the debugging tool presented in Chapter 7, provides a set of observables for debugging neural network training. To date, these observables and instruments focus on the training process. Future work on Cockpit could extend it by putting more emphasis on studying the properties and behavior of the *final* parameters found after training. Analyzing the solution provided by the optimization algorithm could help understand whether the parameters are likely to generalize to unseen data points. Additionally, fairness aspects of the current model could be studied, *e.g.* by providing feedback to the practitioners which data points are currently incorrectly classified. This could indicate, for example, that certain classes are not being modeled accurately enough.

An orthogonal direction to extend Cockpit would be to unravel the quantities layer-wise. Figure 7.4 showed an example where significant information could only be extracted by looking at individual layers of a neural network in isolation. One major challenge is visualizing information layer-wise without overloading the instruments. A crucial step in this endeavor could be to find intelligent approaches to summarize the many layers of contemporary neural networks. This might, for example, be done by grouping by layer type, *e.g.* by combining every convolutional layer, or grouping by depth, *e.g.* building clusters of layers based on their positioning in the network. Looking at the training dynamics of a neural network in the form of multiple gradient-parameter histogram clouds, color-coded by layer, could provide a new perspective and reveal interesting new insights.

The future ambition of Cockpit would certainly be to provide a more automated monitoring of neural network training. This requires a deeper understanding of each quantity included in Cockpit to formulate a clear relationship between their behavior and the root of a training bug. Currently, providing such a *handbook* or *manual* for Cockpit is not possible, but future enhancements could bring us closer to this goal of providing more direct feedback Ideally, Cockpit would, for example, simply have a red lamp signaling that the learning rate is too large and that it should be reduced. This would turn Cockpit into a *copilot* or perhaps even an *autopilot*. This will likely require extensive user studies and additional quantities before such an automated experience could be provided.

8.2.3 Putting It All Together: Resource-efficient Autonomous Training Algorithms for Deep Learning

Today's neural network optimizers require vast amounts of human and energy resources that are spent "babysitting" the training process with expensive hyperparameter searches. Replacing this inefficient process with an automated *one-shot* training method has the potential to reduce the overall compute time and thus the energy cost of training a contemporary deep learning model by at least one order of magnitude. This would make neural network training not only more efficient, but also more robust and accessible. The development of such an algorithm could be based on three insights gained from the analyses presented in this work:

Learning is phase-dependent — Dynamic agents, not static update rules

Figure 7.5 showed an example of how, in neural network training, a suboptimal short-term decision can nevertheless yield long-term benefits. In this case, overstepping is detrimental if one myopically considers only the reduction in the loss in a single iteration. However, it appears that this overstepping also provides faster progress towards lower-loss regions and thus a benefit for the entire optimization run. Automatic training algorithms must thus be able to sacrifice short-term benefits for the longer-term performance and thus overcome the "short-horizon bias" [324]. This suggests a larger point, which is that the goal of optimization depends on the current learning phase. More specifically, there might be three distinct learning phases, each with different goals for the training algorithm:

[324] Wu et al. (2018), "Understanding short-horizon bias in stochastic meta-optimization"

- **Initial chaos:** Recent work [*e.g.*, 2, 87, 88, 145] indicates the existence of a critical early phase of training. During this phase, the weights are mostly determined by the random initialization, which results in large gradients and fast movement in weight space. The first few iterations could critically determine which region of the loss landscape the optimizer moves into, which can already irrevocably affect the final performance.
- **Cruise control:** During most of the training process, the optimizer's goal is not to locally minimize the loss, but instead make progress towards areas of low loss. As mentioned earlier, the training algorithm should not be concerned with

[2] Achille et al. (2017), "Critical Learning Periods in Deep Neural Networks"
[87] Fort et al. (2020), "Deep learning versus kernel learning: an empirical study of loss landscape geometry and the time evolution of the Neural Tangent Kernel"
[88] Frankle et al. (2020), "The Early Phase of Neural Network Training"
[145] Jastrzebski et al. (2020), "The Break-Even Point on the Optimization Trajectories of Deep Neural Networks"

short-term improvements during this phase, but should focus primarily on getting to regions where achieving a good final performance is possible. During this "cruise control" phase, intentionally over-stepping local minima and choosing large learning rates are attractive strategies for a training algorithm.

- **Fine-tuning:** It is only during the later stages of training that convergence become important. Once a region of low loss has been reached, highly precise steps towards its minimum are desirable. Currently, tuned methods use a (tuned) learning rate schedule which decays the step size as training progresses to achieve this. Automated methods could replace these schedules by instead using more costly but high-precision gradient or even Hessian estimates to take a few final fine-tuning steps.

Using all the available information — Distributions and confidences

In order to detect these learning phases, training algorithms may need new observables. Most contemporary deep learning optimizers only use the mean batch gradient, which is readily available in the widely used software frameworks such as TensorFlow [1], PyTorch [228], or JAX [39]. But this obfuscates the fact that each batch provides more information. Recent advances in automatic differentiation, *e.g.* BackPACK [66], allow the efficient computation of, among other things, the individual gradients or their variance. In Chapter 7, we have already made extensive use of the individual gradients by providing (non-)linear transformations thereof. In addition to their use for monitoring or debugging the training process, these novel observables could provide essential information to effectively steer and adjust the optimization process.

[1] Abadi et al. (2015), "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems"

[228] Paszke et al. (2019), "PyTorch: An Imperative Style, High-Performance Deep Learning Library"

[39] Bradbury et al. (2018), "JAX: composable transformations of Python+NumPy programs"

[66] Dangel et al. (2020), "BackPACK: Packing more into Backprop"

Extensive evaluation on practical problems — DeepOBS and MLCommons

The current literature encompasses more than one hundred deep learning optimizers, see Table 3.1. Although most claim superior performance to popular methods, they have largely failed to replace more established methods in practice. Extensive testing using either the DeepOBS toolkit or the MLCommons benchmark for training algorithms can help ensure that a newly developed optimizer is widely applicable and able to reach competitive performance consistently. Such an extensive evaluation is necessary because optimizers are applied across a diverse set of tasks as one of the "plug-and-play" building blocks of a deep learning pipeline.

Following a rigorous and standardized benchmark protocol would help demonstrate the capabilities of the method.

Concluding this work, we believe that developing autonomous training algorithms are a promising and relevant research direction that could significantly improve the efficiency and usability of deep learning. It would make the user experience much more satisfying, which is currently marred by the tedious process of setting sensitive hyperparameters through expensive trial and error. At the very least, attempting to develop such an autonomous training algorithm would require much more insight into why current methods work and in what cases they fail. This would tell us much more about why neural network training is as successful as it is, and help open the black box of deep learning.

Appendix

Appendix for Chapter: A Benchmark Suite for Deep Learning Optimizers

A

A.1 Detailed Description of the Benchmarking Test Problems

In the following, we describe the in total eight DeepOBS test problems that are part of its small and large benchmark set in more detail. In Appendix A.1.3 we provide a description of test problems outside of the small and large benchmark set, that were used in Chapter 7. This is a summary of the more detailed description of every test problem of DeepOBS which can be found in the official documentation.[1] See also Table 6.1 in Chapter 6 which provides a more concise overview of the eight test problems included in the small and large benchmark set.

1: Available at `https://deepobs.readthedocs.io/`.

A.1.1 Small Benchmark Set

P1 **Quadratic Deep:** A 100-dimensional stochastic quadratic loss function. 90% of the eigenvalues are drawn from $[0, 1]$, and 10% from $[30, 60]$ creating an ill-conditioned problem with a structured eigenspectrum similar to the one reported for neural networks, *e.g.* by Chaudhari et al. [50]. No regularization is used. By default, we train with a batch size of 128 for 100 epochs.

[50] Chaudhari et al. (2017), "Entropy-SGD: Biasing gradient descent into wide valleys"

P2 **MNIST — VAE:** A variational autoencoder [166] with three convolutional and three deconvolutional layers with dropout layers (dropout rate of 0.2) and a latent space of size 8 on the MNIST data set. Leaky ReLU activations (see Section 4.2.2) are used with a factor of 0.3. By default, trained with a batch size of 64 for 50 epochs.

[166] Kingma et al. (2015), "Adam: A Method for Stochastic Optimization"

P3 **Fashion-MNIST — CNN (2c2d):** A vanilla convolutional network with two convolutional and two fully connected layers and ReLU activations for image classification on the Fashion-MNIST data set. No regularization is used. Default batch size of 128 and training time of 100 epochs.

P4 **CIFAR-10 — CNN (3c3d):** A slightly larger convolutional network with three convolutional and three fully connected layers on CIFAR-10. L^2 regularization of 0.002 is used on the weights but not the biases. By default, trained with a batch size of 128 for 100 epochs.

A.1.2 Large Benchmark Set

P5 **Fashion-MNIST — VAE:** Same variational autoencoder as P2 with three convolutional and three deconvolutional layers with dropout layers and a latent space of size 8 on the Fashion-MNIST data set. Default training time is 100 epochs with a batch size of 64.

P6 **CIFAR-100 — All-CNN-C:** The all convolutional network All-CNN-C from Springenberg et al. [277] for image classification on the CIFAR-100 data set. L^2 regularization of $5 \cdot 10^{-4}$ is used on the weights but not the biases. By default, DeepOBS uses a batch size of 256 for 350 epochs.

[277] Springenberg et al. (2015), "Striving for simplicity: The all convolutional net"

P7 **SVHN — Wide ResNet-16-4:** The wide residual network WRN-16-4 architecture of Zagoruyko and Komodakis [339] on the SVHN data set for image classification. L^2 regularization of $5 \cdot 10^{-4}$ is used on the weights but not the biases. By default, trained with a batch size of 128 for 160 epochs.

[339] Zagoruyko et al. (2016), "Wide Residual Networks"

P8 **Tolstoi — CharRNN:** A two-layer LSTM [127] with 128 units per LSTM cell for character-level language modeling on Tolstoi's War and Peace. It is trained by default with a sequence length of 50 and batch size of 50 for 200 epochs.

[127] Hochreiter et al. (1997), "Long Short-Term Memory"

A.1.3 Additional Benchmarking Problems

P9 **MNIST — Log. Reg.:** Multinomial logistic regression on MNIST. No regularization is used and both the weights and biases are initialized to 0.0. The default batch size is 128 and the default number of epochs 50.

P10 **MNIST — MLP:** Multi-layer perceptron neural network on MNIST. The model uses four fully connected layers with 1000, 500, 100, and 10 units per layer. The first three layers use ReLU activation, the last one a softmax activation. Initialization is done via truncated normal with standard deviation of $3 \cdot 10^{-2}$ for the weights. Biases are initialized to 0.0. No regularization is used. The model is trained by default with a batch size 128 for 100 epochs.

P11 **Fashion-MNIST — MLP:** Multi-layer perceptron neural network on Fashion-MNIST. Uses the same model as P10 with the same default training parameters, *e.g.* batch size 128 and 100 epochs.

P12 **SVHN — CNN (3c3d):** Convolutional neural network for the SVHN data set. It uses the same 3c3d architecture than P4. The default batch size is 128 and the default number of epochs 100.

Appendix for Chapter: Empirically Comparing Deep Learning Optimizers B

B.1 Robustness to Random Seeds

Data subsampling, random weight initialization, dropout and other aspects of deep learning introduce stochasticity to the training process. As such, judging the performance of an optimizer on a single run may be misleading due to random fluctuations. In our benchmark we use 10 different seeds of the final setting for each budget in order to judge the stability of the optimizer and the results. However, to keep the magnitude of this benchmark feasible, we only use a single seed while tuning, analogously to how a single user would progress. This means that our tuning process can sometimes choose hyperparameter settings which might not even converge for seeds other than the one used for tuning.

Figure B.1 illustrates this behavior on an example problem where we used 10 seeds throughout a tuning process using grid search.[1] The figure shows that in the beginning performance increases when increasing the learning rate, followed by an area were it sometimes works but other times diverges. Picking hyperparameters from this "danger zone" can lead to unstable results. In this case, where we only consider the learning rate, it is clear that decreasing the learning rate a bit to get away from this "danger zone" would lead to a more stable, but equally well-performing algorithm. In more complicated cases, however, we are unable to use a simple heuristic such as this. This might be the case, for example, when tuning multiple hyperparameters or when the effect of the hyperparameter on the performance is less straight forward. Thus, this is a problem not created by improperly using the tuning method, but by an unstable optimization method.

1: The data for this figure was taken, with permission, from Bahde [16].

[16] Bahde (2019), "Towards Meaningful Deep Learning Optimizer Benchmarks"

In our benchmark, we observe a total of 18, 24, and 17 divergent seeds for the small, medium, and large budget respectively. This amounts to roughly 0.5% of the runs in each budget. Most of them occur when using SGD (10, 15, and 7 cases for the small, medium and large budget respectively), AdaGrad (5, 3, and 5 cases for the small, medium and large budget respectively) or Adadelta (3, 5, and 3 cases for the small, medium and large budget respectively), which might indicate that modern adaptive methods are less prone to this kind of behavior. None of these cases occur when using a constant schedule, and most of them occur when using the *trapezoidal* schedule (11, 11, and 9 cases for the small, medium and large budget respectively). However, as our data on diverging

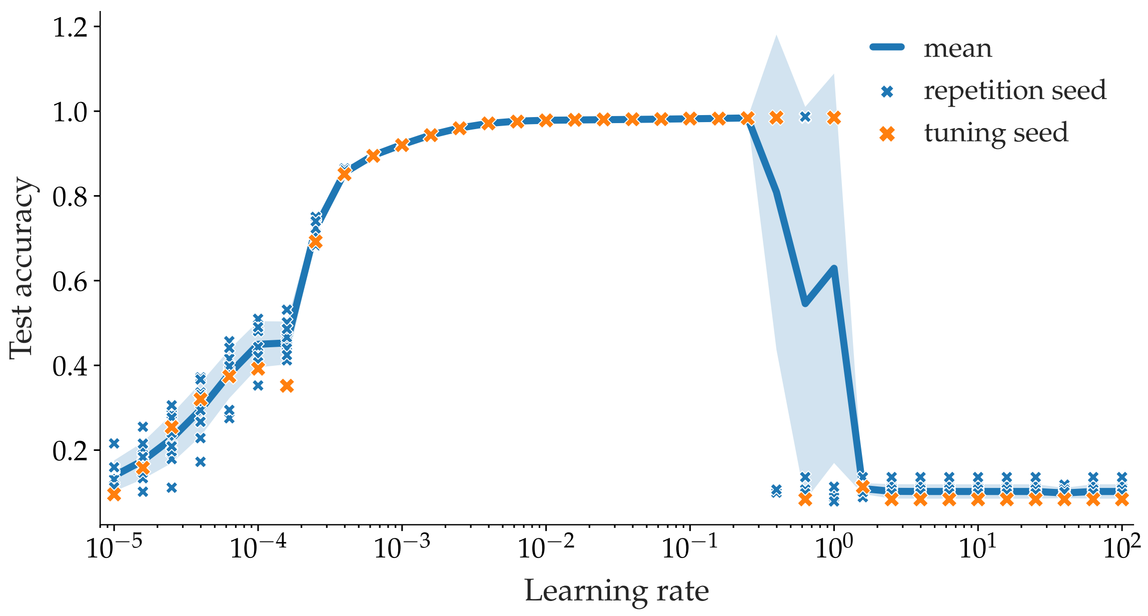


Figure B.1: Performance of SGD on a simple multilayer perceptron (P10 in Appendix A.1.3) For each learning rate, markers in orange (✖) show the initial seed which would be used for tuning, blue markers (✖) illustrate nine additional seeds with otherwise unchanged settings. The mean over all seeds is plotted as a blue line (—), showing one standard deviation as a shaded area (▮).

seeds is very limited, it is not conclusive enough to draw solid conclusions.

B.2 Re-Tuning Experiments

In order to test the stability of our benchmark and especially the tuning method, we selected two optimizers in our benchmark and re-tuned them on all problems a second time. We used completely independent random seeds for both tuning and the 10 repetitions with the final setting. Figure B.2 and Figure B.3 show the distribution of all 10 random seeds for both the original tuning as well as the re-tuning runs for RMSProp and Adadelta. It is evident, that re-tuning results in a shift of this distribution, since small (stochastic) changes during tuning can result in a different chosen hyperparameter setting.

These differences also highlight how crucial it is to look at multiple problems. Individually, small changes, such as re-doing the tuning with different seeds can lead to optimization methods changing rankings. However, they tend to average out when looking at an unbiased list of multiple problems. These results also further supports the statement made in Section 6.3 that there is no optimization method clearly domination the competition, as small performance margins might vanish when re-tuning.

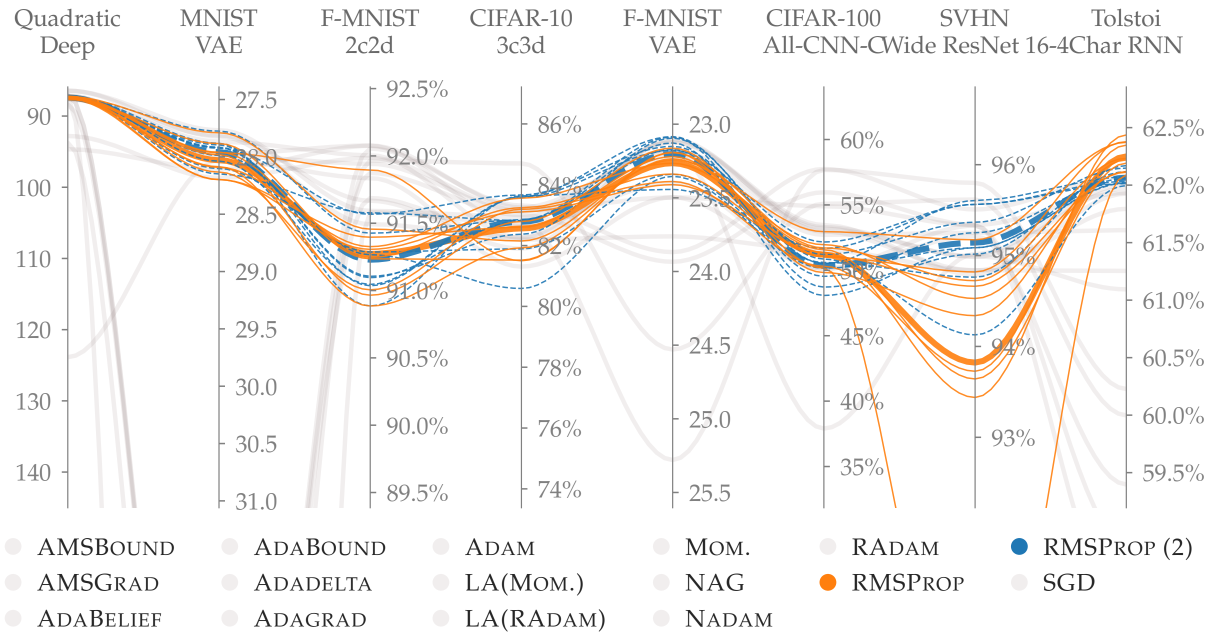


Figure B.2: Mean test set performance of all 10 seeds of RMSProp (—) on all eight optimization problems using the *small budget* for tuning and *no learning rate schedule*. The mean is shown with a thicker line. We repeated the full tuning process on all eight problems using different random seeds, which is shown in dashed lines blue (- -). The mean performance of all other optimizers is shown in transparent gray lines.

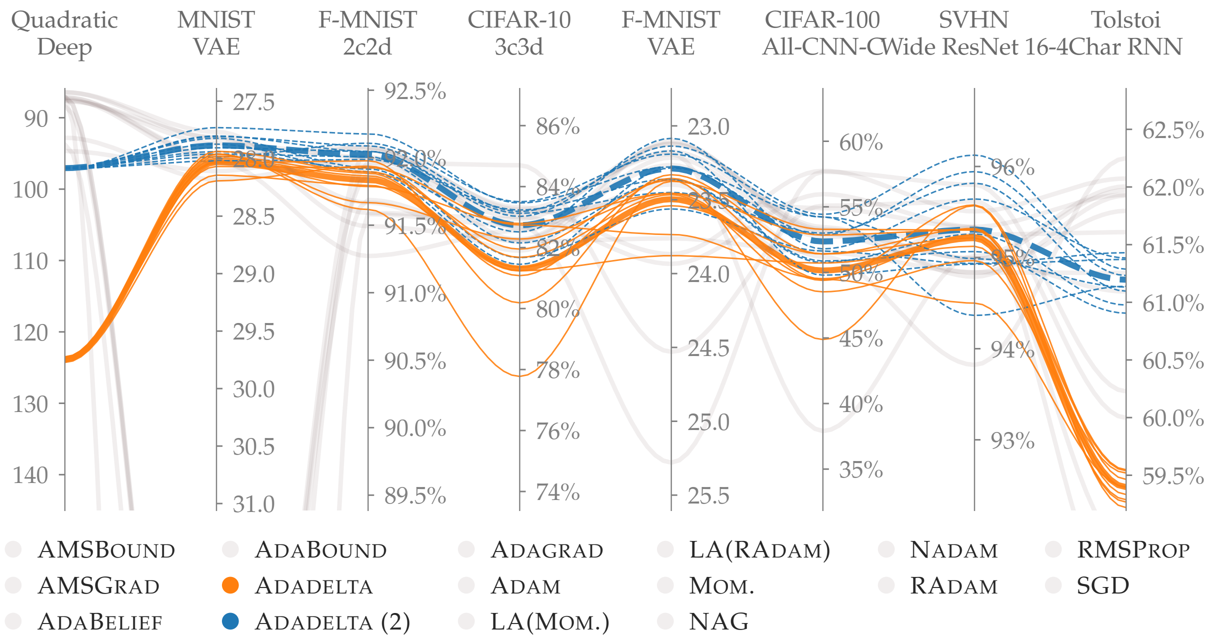


Figure B.3: Mean test set performance of all 10 seeds of AdaDelta (—) on all eight optimization problems using the *small budget* for tuning and *no learning rate schedule*. The mean is shown with a thicker line. We repeated the full tuning process on all eight problems using different random seeds, which is shown in dashed lines blue (- -). The mean performance of all other optimizers is shown in transparent gray lines.

B.3 List of Schedules Selected

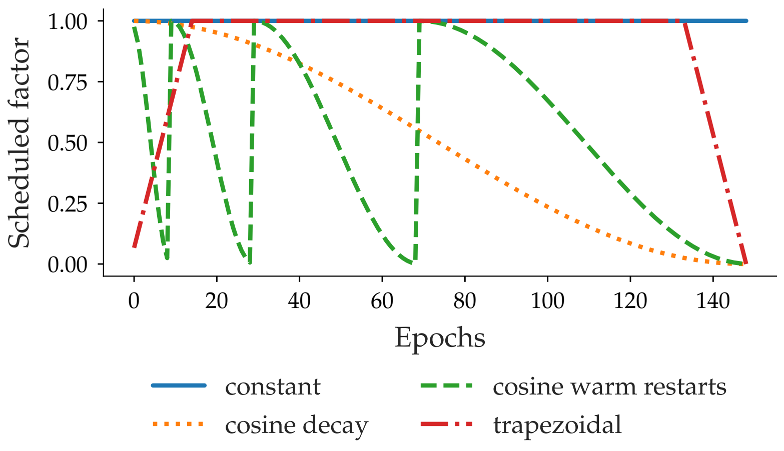


Figure B.4: Illustration of the selected learning rate schedules for a training duration of 150 epochs.

The schedules selected for our benchmark are illustrated in Figure B.4. All learning rate schedules are multiplied by the initial learning rate found via tuning or picked as the default choice.

[193] Loshchilov et al. (2017), "SGDR: Stochastic Gradient Descent with Warm Restarts"

We use a *cosine decay* [193] that starts at 1 and decays in the form of a half period of a cosine to 0. As an example of a cyclical learning rate schedule, we test a *cosine with warm restarts* schedule with a cycle length $\Delta t = 10$ which increases by a factor of 2 after each cycle without any discount factor. Depending on the number of epochs we train our model, it is possible that training stops shortly after one of those warm restarts. Since performance typically declines shortly after increasing the learning rate, we don't report the final performance for this schedule, but instead the performance achieved after the last complete period (just before the next restart). This approach is suggested by the original work of Loshchilov and Hutter [193]. However, we still use the final performance while tuning.

[193] Loshchilov et al. (2017), "SGDR: Stochastic Gradient Descent with Warm Restarts"

[330] Xing et al. (2018), "A Walk with SGD"

A representation of a schedule including warm-up is the *trapezoidal* schedule from Xing et al. [330]. For our benchmark we set a warm-up and cool-down period of $^1/_{10}$ the training time.

B.4 arXiv Mentions

Table B.1: Mentions of each optimizer in titles and abstracts of papers on arXiv per year. All non-selected optimizers from Table 3.1 are grouped into Other.

Optimizer	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
● AMSBound	0	0	0	0	0	0	0	0	0	1	0
● AMSGrad	0	0	0	0	0	0	0	0	7	9	11
● AdaBelief	0	0	0	0	0	0	0	0	0	0	3
● AdaBound	0	0	0	0	0	0	0	0	0	4	4
● AdaDelta	0	0	1	0	1	2	0	1	2	3	3
● AdaGrad	0	0	0	2	1	5	3	8	16	22	24
● Adam	0	2	0	5	4	7	11	31	47	83	119
● Lookahead	0	0	0	0	0	0	0	0	0	2	1
● Momentum	3	6	7	5	9	14	23	57	76	124	205
● NAG	1	0	1	1	1	3	3	11	17	18	19
● Nadam	0	0	0	0	0	0	0	0	1	2	0
● Other	0	1	1	0	1	3	2	4	22	34	36
● RAdam	0	0	0	0	0	0	0	0	0	2	1
● RMSProp	0	0	0	0	0	3	3	13	13	18	18
● SGD	2	9	9	30	42	98	129	205	326	451	532

B.5 Improvement after Tuning

When looking at Figure 6.2, one might realize that few diagonal entries contain negative values. Since diagonal entries reflect the intra-optimizer performance change when tuning on the respective task, this might feel quite counterintuitive at first. *In theory*, this can occur if the respective tuning distributions is chosen poorly, the tuning randomness simply got "unlucky", or we observe significantly worse results for our additional seeds (see Figure B.1).

If we compare Figures B.5 and B.6 to Figures B.7 and B.8 we can see most negative diagonal entries vanish or at least diminish in magnitude. For the latter two figures we allow for more tuning runs and only consider the seed that has been used for this tuning process. The fact that the effect of negative diagonal entries reduces is an indication that they mostly result from the two latter reasons mentioned.

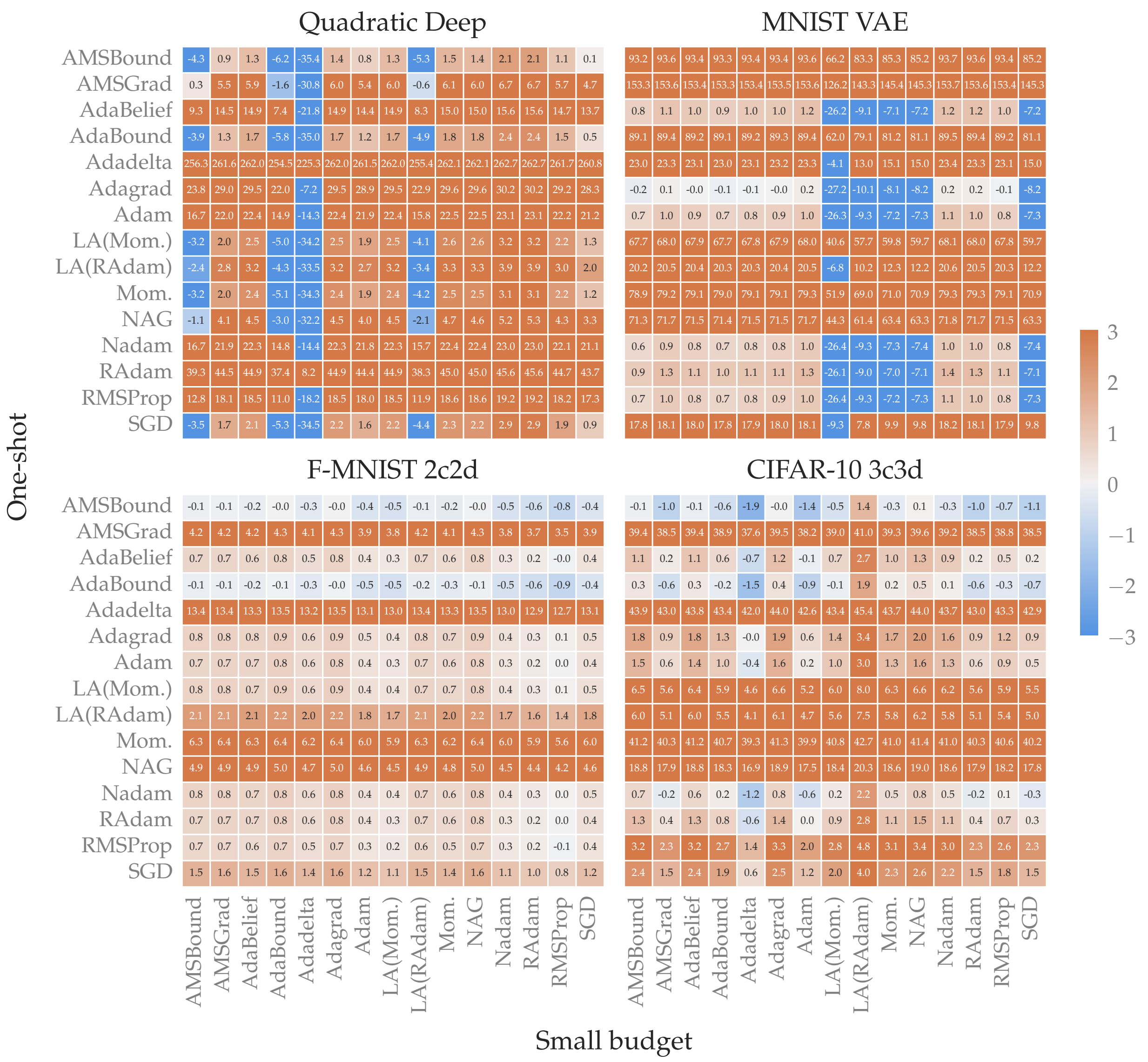


Figure B.5: The absolute test set performance improvement after switching from any untuned optimizer (y-axis, *one-shot*) to any tuned optimizer (x-axis, *small budget*) as an average over 10 random seeds for the *constant* schedule. This is a detailed version of Figure 6.2 in the main text showing the first four problems.

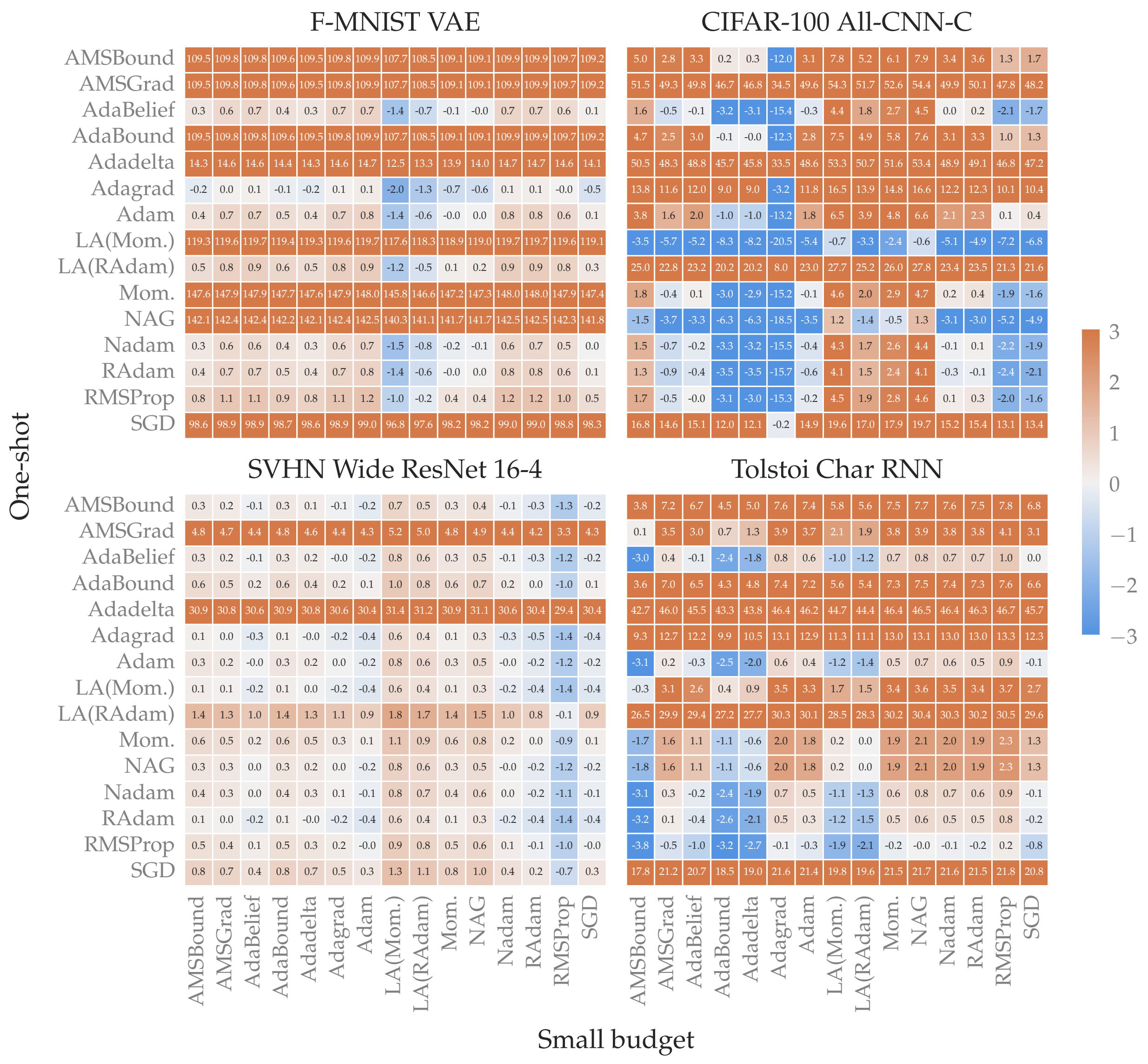


Figure B.6: The absolute test set performance improvement after switching from any untuned optimizer (y-axis, *one-shot*) to any tuned optimizer (x-axis, *small budget*) as an average over 10 random seeds for the *constant* schedule. This is a detailed version of Figure 6.2 in the main text showing the last four problems.

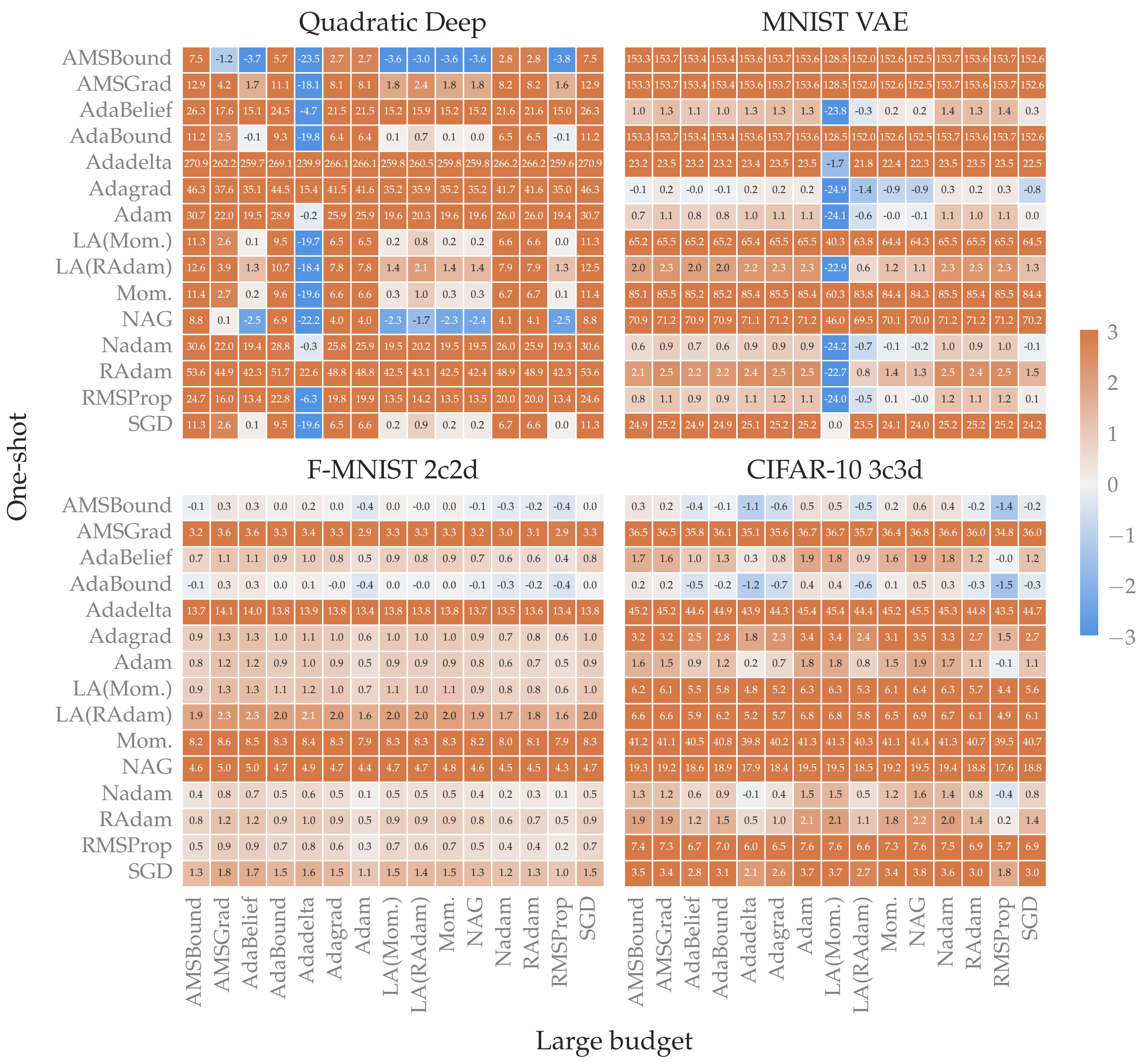


Figure B.7: The absolute test set performance improvement after switching from any untuned optimizer (y-axis, *one-shot*) to any tuned optimizer (x-axis, *large budget*) for the *constant* schedule. This is structurally the same plot as Figure B.5 but comparing to the *large budget* and only considering the seed that has been used for tuning.

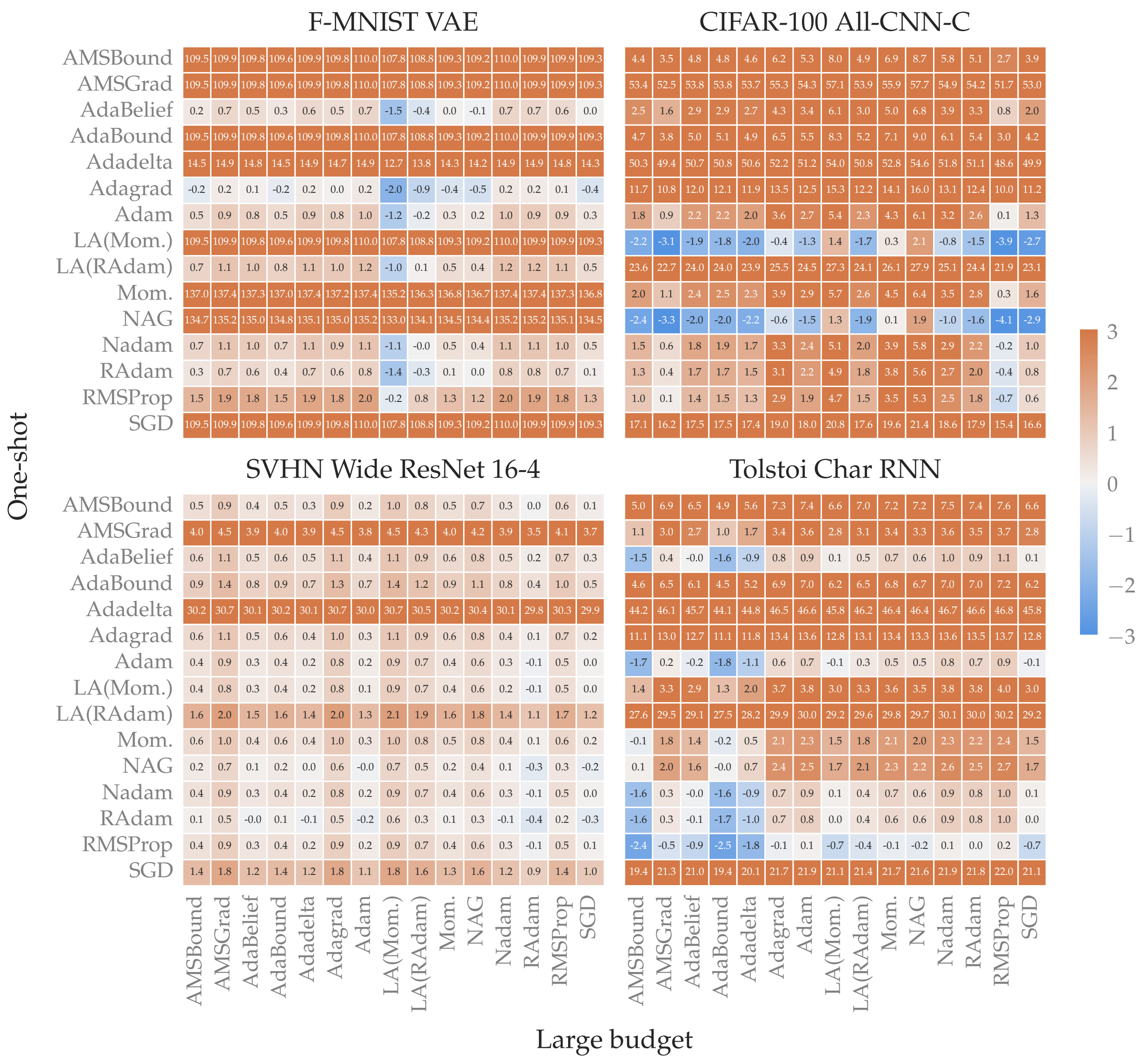


Figure B.8: The absolute test set performance improvement after switching from any untuned optimizer (y-axis, *one-shot*) to any tuned optimizer (x-axis, *large budget*) for the *constant* schedule. This is structurally the same plot as Figure B.6 but comparing to the *large budget* and only considering the seed that has been used for tuning.

B.6 Optimizer Performance Across Test Problems

Similarly to Figure 6.4, we show the corresponding plots for the *small budget* with *no learning rate schedule* in Figure B.9 and the *medium budget* with the *cosine* and *trapezoidal learning rate schedule* in Figures B.10 and B.11. Additionally, in Figure B.12 we show the same setting as Figure 6.4 but showing the training loss instead of the test loss/accuracy.

The high-level trends mentioned in Section 6.3 also hold for the smaller tuning budget in Figure B.9. Namely, taking the winning optimizer for several untuned algorithms (here marked for Adam and AdaBound) will result in a decent performance in most problems with much less effort. Adding a tuned version Adam (or variants thereof) to this selection would result in a very competitive performance. The absolute top-performance however, is achieved by changing optimizers across different problems.

Note, although the *medium budget* is a true superset of the *small budget* it is not given that it will always perform better. Our tuning procedure guarantees that the *validation* performance on the seed that has been used for tuning is as least as good on the medium budget than on the small budget. But due to averaging over multiple seeds and reporting *test* performance instead of *validation* performance, this hierarchy is no longer guaranteed. We discuss the possible effects of averaging over multiple seeds further in Appendix B.1.

The same high-level trends also emerge when considering the *cosine* or *trapezoidal learning rate schedule* in Figures B.10 and B.11. We can also see that the top performance in general increase when adding a schedule (cf. Figure 6.4 and Figure B.11).

Comparing Figure 6.4 and Figure B.12 we can assess the generalization performance of the optimization method not only to an unseen test set, but also to a different performance metric (accuracy instead of loss). Again, the overall picture of varying performance across different problems remains consistent when considering the training loss performance. Similarly to the figures showing test set performance we cannot identify a clear winner, although Adam ands its variants, such as RAdam perform near the top consistently. Note that while Figure B.12 shows the training loss, the optimizers have still be tuned to achieve the best validation performance (*i.e.* accuracy if available, else the loss).

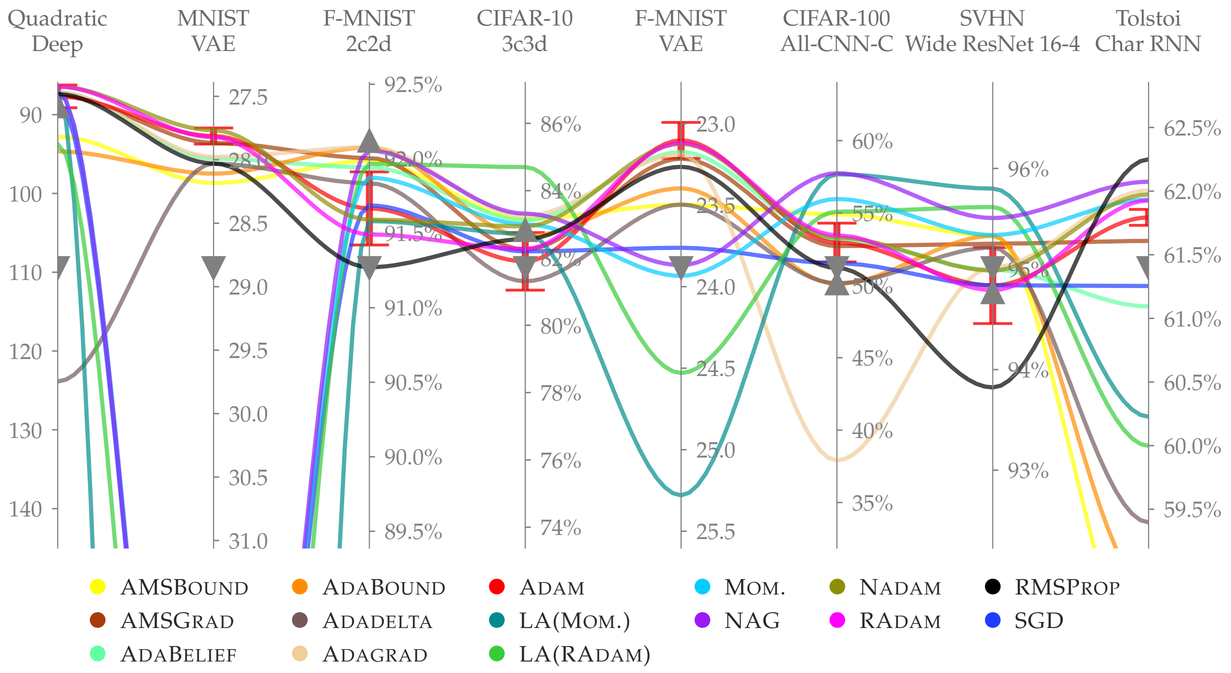


Figure B.9: Mean test set performance over 10 random seeds of all tested optimizers on all eight optimization problems using the *small budget* for tuning and *no learning rate schedule*. One standard deviation for the tuned Adam optimizer is shown with a red error bar (I). The performance of the untuned versions of Adam (▼) and AdaBound (▲) are marked for reference. Note, the upper bound of each axis represents the best performance achieved in the benchmark, while the lower bound is chosen in relation to the performance of Adam with default parameters. Tabular version available in the Appendix as Table B.3.

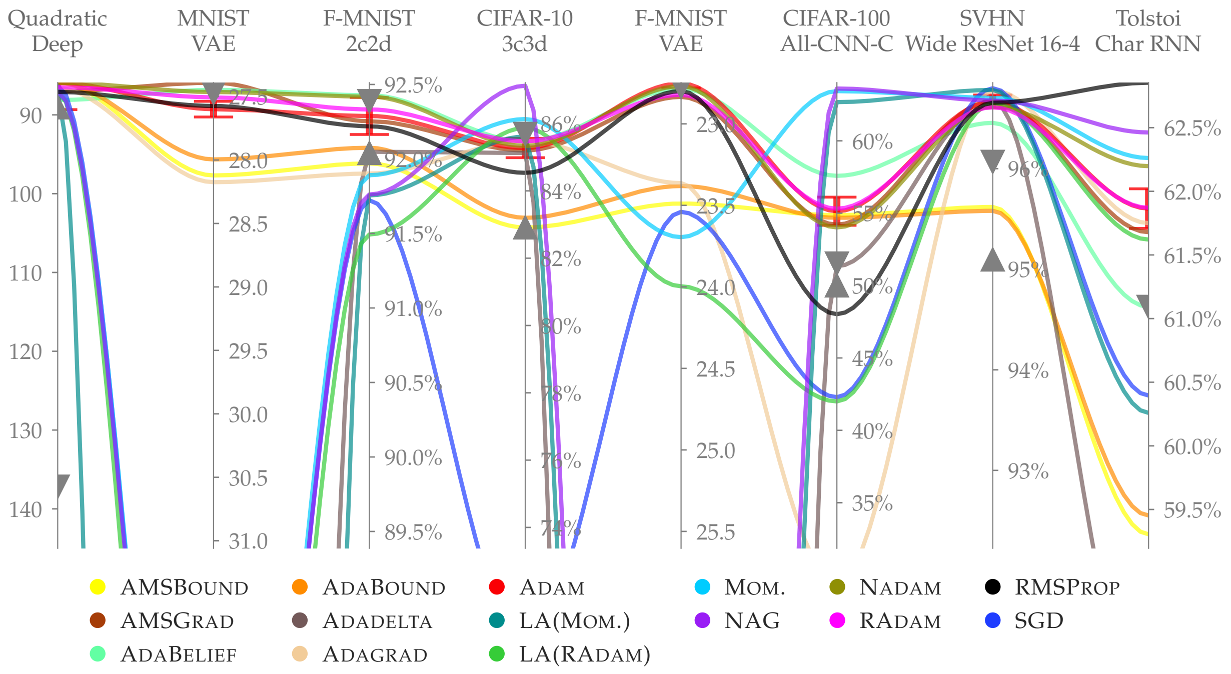


Figure B.10: Mean test set performance over 10 random seeds of all tested optimizers on all eight optimization problems using the *medium budget* for tuning and the *cosine learning rate schedule*. One standard deviation for the tuned Adam optimizer is shown with a red error bar (I). The performance of the untuned versions of Adam (▼) and AdaBound (▲) are marked for reference (this time with the *cosine* learning rate schedule). Note, the upper bound of each axis represents the best performance achieved in the benchmark, while the lower bound is chosen in relation to the performance of Adam with default parameters (and no schedule). Tabular version available in the Appendix as Table B.4.

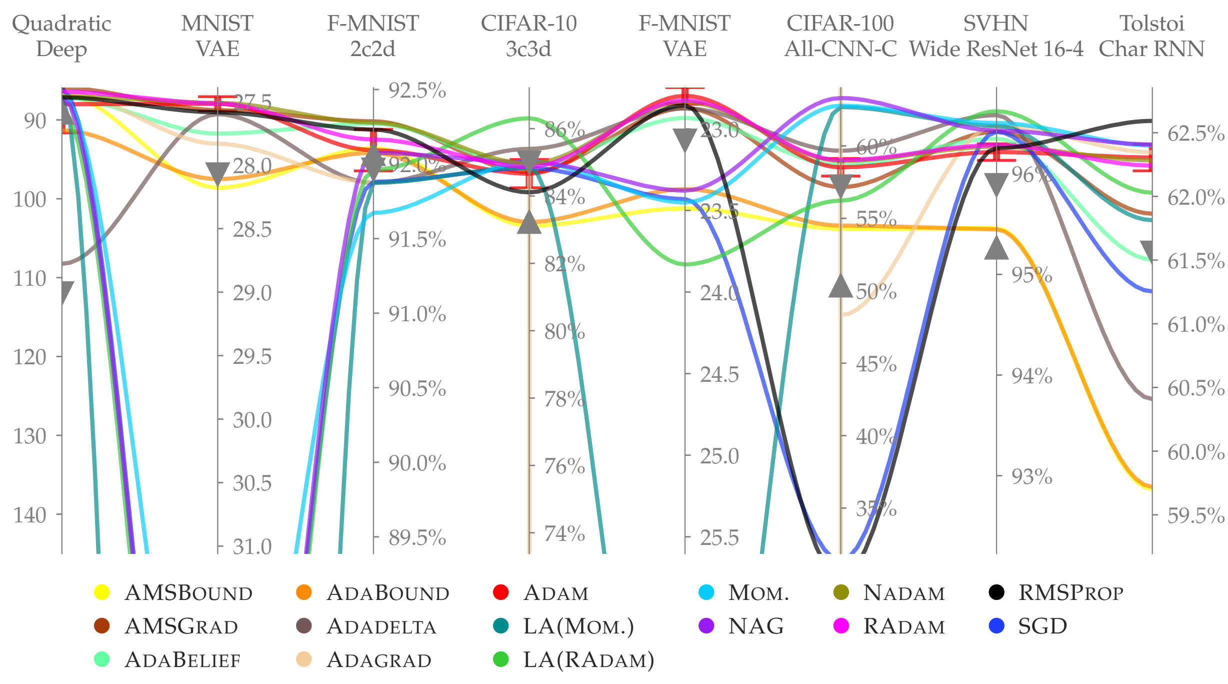


Figure B.11: **Mean test set performance** over 10 random seeds of all tested optimizers on all eight optimization problems using the *large budget* for tuning and the *trapezoidal learning rate schedule*. One standard deviation for the tuned Adam optimizer is shown with a red error bar (I). The performance of the untuned versions of Adam (▼) and AdaBound (▲) are marked for reference (this time with the *trapezoidal* learning rate schedule). Note, the upper bound of each axis represents the best performance achieved in the benchmark, while the lower bound is chosen in relation to the performance of Adam with default parameters (and no schedule). Tabular version available in the Appendix as Table B.5.

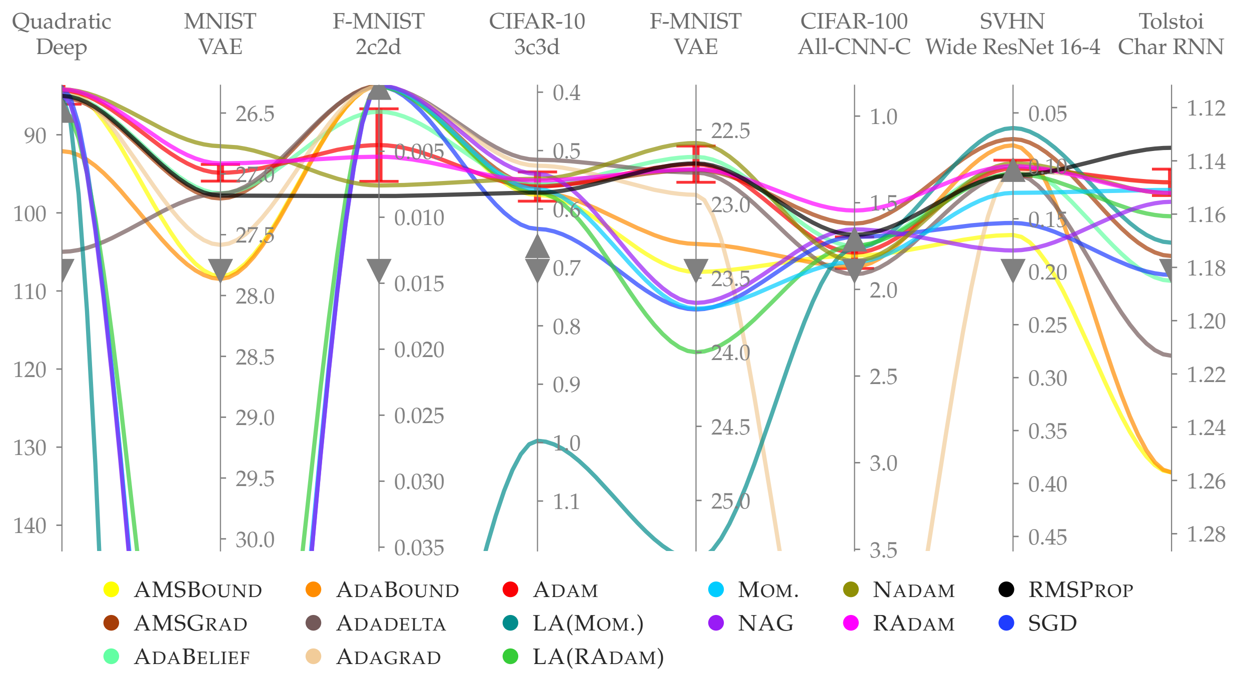


Figure B.12: **Mean *training* loss performance** over 10 random seeds of all tested optimizers on all eight optimization problems using the *large budget* for tuning and *no learning rate schedule*. One standard deviation for the tuned Adam optimizer is shown with a red error bar (I). The performance of the untuned versions of Adam (▼) and AdaBound (▲) are marked for reference. Note, the upper bound of each axis represents the best performance achieved in the benchmark, while the lower bound is chosen in relation to the performance of Adam with default parameters (and no schedule). This figure is very similar to Figure 6.4, but showing the *training loss* performance instead of the *test accuracy* (or *test loss* if no accuracy is available). Tabular version available in the Appendix as Table B.6.

B.7 Tabular Version

Table B.2: Tabular version of Figure 6.4. Mean test set performance and standard deviation over 10 random seeds of all tested optimizers on all eight optimization problems using the *large budget* for tuning and *no learning rate schedule*. For comprehensability, mean and standard deviation are rounded.

Optimizer	Quadratic Deep	MNIST VAE	F-MNIST 2c2d	CIFAR-10 3c3d	F-MNIST VAE	CIFAR-100	SVHN	Tolstoi
● AMSBound	86.35 ± 3.47	28.14 ± 0.15	92.15 ± 0.13	82.99 ± 0.78	23.55 ± 0.18	54.64 ± 1.33	95.31 ± 0.31	59.70 ± 0.16
● AMSGrad	87.64 ± 1.00	27.85 ± 0.07	**92.26** ± 0.16	83.42 ± 0.65	23.11 ± 0.10	52.34 ± 1.03	95.58 ± 0.31	61.52 ± 0.13
● AdaBelief	87.17 ± 0.03	28.01 ± 0.06	92.06 ± 0.24	82.85 ± 0.59	23.22 ± 0.08	53.76 ± 1.35	95.09 ± 0.30	61.26 ± 0.17
● AdaBound	94.66 ± 6.25	28.14 ± 0.13	92.03 ± 0.13	83.39 ± 0.53	23.38 ± 0.09	54.77 ± 0.94	95.40 ± 0.29	59.73 ± 0.20
● AdaDelta	106.95 ± 0.14	27.87 ± 0.10	92.07 ± 0.11	83.34 ± 0.74	23.18 ± 0.13	53.18 ± 2.48	95.30 ± 0.60	60.54 ± 0.15
● AdaGrad	86.70 ± 1.99	28.04 ± 0.29	92.05 ± 0.17	83.08 ± 0.41	23.16 ± 0.04	43.63 ± 21.35	95.34 ± 0.49	62.01 ± 0.10
● Adam	86.58 ± 1.95	27.77 ± 0.03	91.69 ± 0.16	82.95 ± 0.70	23.06 ± 0.10	54.84 ± 0.65	94.84 ± 0.30	61.97 ± 0.12
● LA(Mom.)	87.17 ± 0.07	52.86 ± 0.84	91.74 ± 0.19	74.01 ± 3.70	25.37 ± 0.35	57.32 ± 0.80	**95.82** ± 0.11	61.44 ± 0.17
● LA(RAdam)	89.03 ± 0.87	34.26 ± 9.37	92.05 ± 0.16	83.00 ± 0.64	24.04 ± 0.25	54.92 ± 0.97	95.67 ± 0.11	61.73 ± 0.10
● Momentum	87.04 ± 0.02	36.00 ± 11.09	91.87 ± 0.12	83.16 ± 0.56	23.86 ± 0.15	56.21 ± 0.67	95.37 ± 0.27	61.97 ± 0.12
● NAG	87.08 ± 0.02	36.16 ± 10.99	91.87 ± 0.12	83.30 ± 0.88	23.85 ± 0.22	**57.85** ± 0.77	95.28 ± 0.23	61.74 ± 0.12
● NAdam	86.45 ± 1.94	**27.73** ± 0.09	91.75 ± 0.42	**83.58** ± 0.45	**23.00** ± 0.07	53.44 ± 1.27	95.00 ± 0.25	62.01 ± 0.11
● RAdam	86.43 ± 1.93	27.81 ± 0.06	91.63 ± 0.24	82.85 ± 0.52	23.10 ± 0.11	53.98 ± 1.00	94.83 ± 0.38	61.98 ± 0.13
● RMSProp	87.38 ± 0.12	27.86 ± 0.08	91.79 ± 0.36	82.16 ± 0.65	23.11 ± 0.08	52.16 ± 0.99	95.25 ± 0.34	**62.24** ± 0.07
● SGD	**86.29** ± 3.44	36.17 ± 10.97	91.80 ± 0.23	82.64 ± 0.91	23.83 ± 0.22	50.58 ± 1.49	95.11 ± 0.31	61.29 ± 0.14

Table B.3: Tabular version of Figure B.9. Mean test set performance and standard deviation over 10 random seeds of all tested optimizers on all eight optimization problems using the *small budget* for tuning and *no learning rate schedule*. For comprehensability, mean and standard deviation are rounded.

Optimizer	Quadratic Deep	MNIST VAE	F-MNIST 2c2d	CIFAR-10 3c3d	F-MNIST VAE	CIFAR-100	SVHN	Tolstoi
● AMSBound	92.80 ± 5.99	28.18 ± 0.14	91.99 ± 0.10	83.15 ± 0.65	23.50 ± 0.11	54.91 ± 0.54	95.33 ± 0.17	58.25 ± 0.19
● AMSGrad	87.58 ± 0.71	27.87 ± 0.08	92.01 ± 0.09	82.25 ± 0.54	23.21 ± 0.06	52.71 ± 0.97	95.25 ± 0.21	61.61 ± 0.14
● AdaBelief	87.18 ± 0.03	27.99 ± 0.06	91.94 ± 0.33	83.13 ± 0.60	23.17 ± 0.07	53.17 ± 1.15	94.99 ± 0.31	61.09 ± 0.09
● AdaBound	94.66 ± 6.25	28.11 ± 0.09	**92.08** ± 0.20	82.64 ± 1.03	23.40 ± 0.06	50.10 ± 16.39	95.33 ± 0.16	58.88 ± 0.16
● AdaDelta	123.86 ± 0.24	28.03 ± 0.08	91.84 ± 0.11	81.31 ± 1.40	23.50 ± 0.17	50.14 ± 2.29	95.21 ± 0.29	59.40 ± 0.11
● AdaGrad	87.14 ± 1.02	27.98 ± 0.16	**92.08** ± 0.23	83.25 ± 0.51	23.19 ± 0.08	37.90 ± 24.22	95.02 ± 0.21	62.01 ± 0.11
● Adam	87.68 ± 1.44	27.81 ± 0.06	91.67 ± 0.25	81.90 ± 0.86	**23.10** ± 0.11	52.96 ± 1.34	94.84 ± 0.38	61.79 ± 0.06
● LA(Mom.)	87.16 ± 0.06	55.20 ± 0.86	91.58 ± 0.15	82.72 ± 1.24	25.28 ± 0.23	57.68 ± 0.60	**95.80** ± 0.10	60.23 ± 0.26
● LA(RAdam)	93.75 ± 3.15	38.11 ± 9.73	91.97 ± 0.22	**84.70** ± 0.30	24.53 ± 0.15	55.09 ± 0.98	95.62 ± 0.19	60.00 ± 0.11
● Momentum	87.03 ± 0.02	36.08 ± 11.04	91.87 ± 0.16	83.00 ± 0.71	23.93 ± 0.30	55.96 ± 0.92	95.34 ± 0.23	61.93 ± 0.10
● NAG	87.08 ± 0.02	36.18 ± 10.97	92.05 ± 0.13	83.32 ± 0.57	23.87 ± 0.33	**57.75** ± 0.71	95.51 ± 0.21	62.07 ± 0.10
● NAdam	86.45 ± 1.94	**27.77** ± 0.06	91.59 ± 0.25	82.94 ± 0.61	23.12 ± 0.06	53.30 ± 0.90	94.99 ± 0.18	61.97 ± 0.08
● RAdam	**86.43** ± 1.93	27.82 ± 0.06	91.49 ± 0.40	82.27 ± 0.53	23.12 ± 0.07	53.47 ± 0.86	94.79 ± 0.38	61.93 ± 0.14
● RMSProp	87.40 ± 0.14	28.03 ± 0.13	91.27 ± 0.28	82.56 ± 0.71	23.26 ± 0.08	51.20 ± 0.89	93.82 ± 1.64	**62.25** ± 0.12
● SGD	88.37 ± 3.55	36.18 ± 10.96	91.69 ± 0.15	82.20 ± 1.32	23.76 ± 0.25	51.53 ± 1.37	94.84 ± 0.56	61.25 ± 0.12

Table B.4: Tabular version of Figure B.10. Mean test set performance and standard deviation over 10 random seeds of all tested optimizers on all eight optimization problems using the *medium budget* for tuning and the *cosine learning rate schedule*. For comprehensability, mean and standard deviation are rounded.

Optimizer	Quadratic Deep	MNIST VAE	F-MNIST 2c2d	CIFAR-10 3c3d	F-MNIST VAE	CIFAR-100	SVHN	Tolstoi
● AMSBound	85.94 ± 3.41	28.12 ± 0.19	91.97 ± 0.15	82.91 ± 0.83	23.49 ± 0.07	54.87 ± 0.70	95.62 ± 0.15	59.31 ± 0.36
● AMSGrad	87.00 ± 0.55	**27.39** ± 0.04	92.25 ± 0.22	85.20 ± 0.34	22.83 ± 0.06	54.21 ± 1.99	96.68 ± 0.07	61.68 ± 0.17
● AdaBelief	88.12 ± 0.04	27.45 ± 0.05	**92.43** ± 0.14	85.47 ± 0.26	22.78 ± 0.04	57.58 ± 0.57	96.46 ± 0.08	61.09 ± 0.17
● AdaBound	**85.92** ± 3.41	28.00 ± 0.09	92.08 ± 0.17	83.20 ± 0.62	23.38 ± 0.08	54.68 ± 0.81	95.58 ± 0.10	59.45 ± 0.36
● Adadelta	164.58 ± 0.35	58.46 ± 61.52	92.05 ± 0.08	85.12 ± 0.28	60.55 ± 49.27	51.34 ± 0.64	96.68 ± 0.05	57.77 ± 0.19
● AdaGrad	86.61 ± 1.94	28.17 ± 0.27	91.90 ± 0.23	85.48 ± 0.35	23.36 ± 0.05	29.40 ± 28.41	96.78 ± 0.07	61.75 ± 0.07
● Adam	**85.92** ± 3.41	27.60 ± 0.06	92.29 ± 0.12	85.27 ± 0.29	**22.75** ± 0.03	55.14 ± 0.97	96.67 ± 0.06	61.86 ± 0.16
● LA(Mom.)	87.06 ± 0.02	76.78 ± 24.04	91.76 ± 0.20	85.61 ± 0.24	46.09 ± 21.85	62.67 ± 0.81	96.78 ± 0.08	60.26 ± 0.23
● LA(RAdam)	87.08 ± 0.42	37.41 ± 10.15	91.49 ± 0.24	85.87 ± 0.18	24.00 ± 0.12	42.00 ± 27.55	96.65 ± 0.09	61.62 ± 0.16
● Momentum	87.06 ± 0.02	36.33 ± 10.85	91.89 ± 0.12	86.13 ± 0.19	23.70 ± 0.18	63.43 ± 0.56	96.71 ± 0.05	62.26 ± 0.13
● NAG	87.06 ± 0.02	36.53 ± 10.71	91.76 ± 0.13	**87.12** ± 0.19	41.41 ± 21.65	**63.61** ± 0.46	96.68 ± 0.08	62.46 ± 0.10
● Nadam	85.93 ± 3.41	27.46 ± 0.10	92.42 ± 0.12	85.34 ± 0.34	22.77 ± 0.07	54.02 ± 0.71	96.62 ± 0.07	62.20 ± 0.12
● RAdam	86.49 ± 1.94	27.51 ± 0.05	92.33 ± 0.10	85.47 ± 0.36	22.82 ± 0.08	55.31 ± 0.86	96.61 ± 0.07	61.87 ± 0.19
● RMSProp	87.09 ± 0.01	27.57 ± 0.05	92.22 ± 0.18	84.54 ± 0.25	22.80 ± 0.04	48.02 ± 15.69	96.65 ± 0.06	**62.85** ± 0.06
● SGD	86.30 ± 3.41	36.47 ± 10.76	91.72 ± 0.21	70.50 ± 30.76	23.54 ± 0.13	42.29 ± 27.05	**96.80** ± 0.08	60.40 ± 0.11

Table B.5: Tabular version of Figure B.11. Mean test set performance and standard deviation over 10 random seeds of all tested optimizers on all eight optimization problems using the *large budget* for tuning and *trapezoidal learning rate schedule*. For comprehensability, mean and standard deviation are rounded.

Optimizer	Quadratic Deep	MNIST VAE	F-MNIST 2c2d	CIFAR-10 3c3d	F-MNIST VAE	CIFAR-100	SVHN	Tolstoi
● AMSBound	86.78 ± 2.04	28.18 ± 0.19	92.11 ± 0.16	83.11 ± 0.84	23.49 ± 0.11	54.28 ± 1.23	95.46 ± 0.21	59.70 ± 0.14
● AMSGrad	**85.94** ± 3.42	27.57 ± 0.06	**92.29** ± 0.12	84.71 ± 0.31	22.87 ± 0.06	57.15 ± 0.89	96.42 ± 0.06	61.86 ± 0.14
● AdaBelief	87.19 ± 0.02	27.75 ± 0.05	92.27 ± 0.10	84.90 ± 0.32	22.93 ± 0.07	58.66 ± 0.50	96.35 ± 0.07	61.50 ± 0.15
● AdaBound	91.34 ± 5.60	28.11 ± 0.09	92.08 ± 0.14	83.23 ± 0.58	23.37 ± 0.05	54.50 ± 1.23	95.45 ± 0.18	59.72 ± 0.17
● Adadelta	108.26 ± 0.14	27.60 ± 0.08	91.87 ± 0.20	85.40 ± 0.17	22.87 ± 0.08	59.67 ± 0.38	96.58 ± 0.07	60.41 ± 0.11
● AdaGrad	86.51 ± 1.95	27.83 ± 0.15	91.88 ± 0.12	84.84 ± 0.23	7e23 ± 2e24	48.31 ± 23.66	96.48 ± 0.10	62.35 ± 0.16
● Adam	88.01 ± 3.63	27.52 ± 0.06	92.09 ± 0.14	84.66 ± 0.42	**22.80** ± 0.05	58.52 ± 0.61	96.22 ± 0.08	62.31 ± 0.10
● LA(Mom.)	87.12 ± 0.02	52.89 ± 0.00	91.87 ± 0.17	84.85 ± 0.60	28.24 ± 13.23	62.69 ± 0.42	96.48 ± 0.10	61.81 ± 0.17
● LA(RAdam)	88.67 ± 1.24	36.14 ± 10.99	91.96 ± 0.14	**86.31** ± 0.25	23.83 ± 0.14	56.22 ± 18.42	**96.62** ± 0.08	62.03 ± 0.14
● Momentum	87.06 ± 0.02	33.77 ± 9.62	91.67 ± 0.22	85.02 ± 0.30	23.45 ± 0.22	62.78 ± 0.34	96.50 ± 0.08	62.40 ± 0.08
● NAG	87.06 ± 0.02	35.80 ± 11.20	92.08 ± 0.16	85.00 ± 0.44	23.38 ± 0.16	**63.30** ± 0.31	96.43 ± 0.11	62.41 ± 0.09
● Nadam	87.03 ± 3.66	**27.51** ± 0.08	92.28 ± 0.11	84.96 ± 0.37	22.83 ± 0.08	58.96 ± 0.77	96.27 ± 0.10	62.28 ± 0.11
● RAdam	86.43 ± 1.93	**27.51** ± 0.05	92.17 ± 0.17	84.86 ± 0.32	22.83 ± 0.07	59.01 ± 0.73	96.29 ± 0.09	62.24 ± 0.13
● RMSProp	87.14 ± 0.03	27.58 ± 0.07	92.23 ± 0.13	84.11 ± 0.16	22.85 ± 0.05	30.15 ± 29.15	96.25 ± 0.09	**62.59** ± 0.11
● SGD	86.05 ± 3.40	35.71 ± 11.26	91.88 ± 0.17	84.83 ± 0.27	23.43 ± 0.19	31.36 ± 30.38	96.42 ± 0.07	61.25 ± 0.11

Table B.6: Tabular version of Figure B.12. Mean *training* set performance and standard deviation over 10 random seeds of all tested optimizers on all eight optimization problems using the *large budget* for tuning and *no learning rate schedule*. For comprehensability, mean and standard deviation are rounded.

Optimizer	Quadratic Deep	MNIST VAE	F-MNIST 2c2d	CIFAR-10 3c3d	F-MNIST VAE	CIFAR-100	SVHN	Tolstoi
● AMSBound	84.10 ± 3.34	27.84 ± 0.15	**0.00** ± 0.00	0.58 ± 0.02	23.46 ± 0.22	1.80 ± 0.09	0.17 ± 0.00	1.26 ± 0.01
● AMSGrad	85.24 ± 1.21	27.20 ± 0.09	**0.00** ± 0.00	0.56 ± 0.01	22.77 ± 0.10	1.62 ± 0.11	0.07 ± 0.00	1.18 ± 0.01
● AdaBelief	84.87 ± 0.30	27.16 ± 0.07	**0.00** ± 0.00	0.56 ± 0.02	22.68 ± 0.07	1.75 ± 0.11	0.10 ± 0.00	1.19 ± 0.01
● AdaBound	92.10 ± 5.64	27.86 ± 0.17	**0.00** ± 0.00	0.57 ± 0.02	23.27 ± 0.14	1.87 ± 0.09	0.08 ± 0.01	1.26 ± 0.01
● Adadelta	104.99 ± 0.30	27.16 ± 0.12	**0.00** ± 0.00	**0.52** ± 0.01	22.79 ± 0.15	1.91 ± 0.17	0.11 ± 0.01	1.21 ± 0.01
● AdaGrad	84.40 ± 1.73	27.58 ± 0.34	**0.00** ± 0.00	0.53 ± 0.02	22.94 ± 0.06	5.25 ± 6.85	0.10 ± 0.01	1.15 ± 0.01
● Adam	84.33 ± 1.76	26.99 ± 0.07	**0.00** ± 0.00	0.56 ± 0.03	22.73 ± 0.12	1.79 ± 0.09	0.10 ± 0.01	1.15 ± 0.00
● LA(Mom.)	84.85 ± 0.30	52.85 ± 0.74	0.06 ± 0.02	1.00 ± 0.16	25.40 ± 0.39	1.76 ± 0.06	**0.06** ± 0.00	1.17 ± 0.01
● LA(RAdam)	86.68 ± 1.10	34.33 ± 9.29	**0.00** ± 0.00	0.57 ± 0.02	24.00 ± 0.26	1.75 ± 0.08	0.11 ± 0.00	1.16 ± 0.00
● Momentum	84.77 ± 0.30	35.98 ± 11.06	**0.00** ± 0.00	0.57 ± 0.02	23.71 ± 0.13	1.84 ± 0.07	0.13 ± 0.00	1.15 ± 0.01
● NAG	84.77 ± 0.30	36.15 ± 10.96	**0.00** ± 0.00	0.54 ± 0.02	23.67 ± 0.22	1.65 ± 0.03	0.18 ± 0.00	1.16 ± 0.01
● Nadam	84.19 ± 1.74	**26.77** ± 0.12	0.01 ± 0.01	0.55 ± 0.02	**22.59** ± 0.08	1.84 ± 0.08	0.10 ± 0.00	1.15 ± 0.01
● RAdam	84.18 ± 1.74	26.91 ± 0.10	0.01 ± 0.00	0.55 ± 0.02	22.77 ± 0.08	**1.54** ± 0.10	0.10 ± 0.00	1.15 ± 0.01
● RMSProp	85.02 ± 0.32	27.18 ± 0.13	0.01 ± 0.01	0.57 ± 0.02	22.73 ± 0.08	1.69 ± 0.13	0.11 ± 0.01	**1.14** ± 0.00
● SGD	**83.95** ± 3.25	36.15 ± 10.95	**0.00** ± 0.00	0.63 ± 0.02	23.71 ± 0.23	1.70 ± 0.10	0.15 ± 0.01	1.18 ± 0.00

Appendix for Chapter: A Practical Debugging Tool for the Training of Deep Neural Networks

C

C.1 Code Example

One design principle of Cockpit is its easy integration with conventional PyTorch [228] training loops. Algorithm C.1 shows a working example of a standard training loop with Cockpit integration. Lines that show Cockpit-specific code or required changes to traditional training codes are highlighted. More examples and tutorials are described in Cockpit's documentation.[1] Cockpit's syntax is inspired by BackPACK: It can be used interchangeably with the library responsible for most of its back-end computations. Changes to the code are straightforward and include:

- **Importing** (*Lines 5, 7* and *8*): Besides importing Cockpit we also need to import BackPACK which is required for extending (parts of) the model (see next step).
- **Extending** (*Lines 11* and *12*): When defining the model and the loss function, we need to *extend* both of them using BackPACK. This is as trivial as wrapping them in the `extend()` function provided by BackPACK and lets BackPACK know that additional quantities (such as the individual gradients) should be computed for them. Note, that while applying BackPACK is easy, it currently does not support all possible model architectures and layer types. Specifically, *batch norm* layers are not supported since using them results in ill-defined individual gradients.
- **Individual losses** (*Line 13*): For the `Alpha` quantity, Cockpit also requires the individual loss values. This is necessary in order to estimate the variance of the loss estimate. These individual loss values, and consequently the variance, can be computed cheaply but its computation is not usually part of a conventional training loop. Creating this loss is done analogously to creating any other loss, with the only exception of setting `reduction="none"`. Since we don't differentiate this loss, we don't need to extend it.
- **Cockpit configuration** (*Line 16* and *17*): Initializing Cockpit requires passing the (extended) model parameters as well as a list of quantities that should be tracked. Table 7.1 provides an overview of all possible quantities. In this example, we use one of the pre-defined configurations offered by Cockpit. Separately, we initialize the plotting part of Cockpit. We deliberately detached the visualization from the tracking to allow greater flexibility.

[228] Paszke et al. (2019), "PyTorch: An Imperative Style, High-Performance Deep Learning Library"

1: Available at `https://cockpit.readthedocs.io/en/latest/`.

- **Quantity computation** (*Line 27* and *38*): Performing the training is very similar to a regular training loop, with the only difference being that the backward pass should be surrounded by the Cockpit context (`with cockpit():`). Additionally to the `global_step` we also pass a few additional information to Cockpit that are computed anyway and can be re-used by Cockpit, such as the batch size, the individual losses, or the optimizer itself. After the backward pass (when the context is left) all Cockpit quantities are automatically computed.
- **Logging and visualizing** (*Line 46* and *47*): At any point during the training, we can write all quantities to a log file. We can use this log file, or alternatively Cockpit directly, to visualize all quantities which would result in a status screen similar to Figure 7.2.

```
 1 """Example: Training Loop using Cockpit."""
 2 
 3 import torch
 4 from _utils_examples import cnn, fmnist_data, get_logpath
 5 from backpack import extend
 6 
 7 from cockpit import Cockpit, CockpitPlotter
 8 from cockpit.utils.configuration import configuration as config
 9 
10 fmnist_data = fmnist_data()
11 model = extend(cnn())
12 loss_fn = extend(torch.nn.CrossEntropyLoss(reduction="mean"))
13 losses_fn = torch.nn.CrossEntropyLoss(reduction="none")
14 opt = torch.optim.SGD(model.parameters(), lr=1e-2)
15 
16 cockpit = Cockpit(model.parameters(), quantities=config("full"))
17 plotter = CockpitPlotter()
18 
19 max_steps, global_step = 50, 0
20 for inputs, labels in iter(fmnist_data):
21     opt.zero_grad()
22 
23     outputs = model(inputs)
24     loss = loss_fn(outputs, labels)
25     losses = losses_fn(outputs, labels)
26 
27     with cockpit(
28         global_step,
29         info={
30             "batch_size": inputs.shape[0],
31             "individual_losses": losses,
32             "loss": loss,
33             "optimizer": opt,
34         },
35     ):
36         loss.backward(
37             create_graph=cockpit.create_graph(global_step),
38         )
39 
```

```
    opt.step()
    global_step += 1

    if global_step >= max_steps:
        break

cockpit.write(get_logpath())
plotter.plot(get_logpath())
```

Algorithm C.1: Complete training loop with Cockpit in PyTorch. Line changes compared to a more traditional training loop without Cockpit are highlighted in light orange (🟧).

C.2 Cockpit Instruments Overview

Table C.1 lists all quantities available in the first public release of Cockpit. If necessary, we provide references to their mathematical definition. This table contains additional quantities, compared to Table 7.1 in the main text. To improve the presentation, we decided to not describe every quantity available in Cockpit in the main part and instead focus on the investigated metrics. Custom quantities can be added easily without having to understand the inner-workings.

[21] Balles et al. (2017), "Coupling Adaptive Batch Sizes with Learning Rates"

[199] Mahsereci et al. (2017), "Early Stopping without a Validation Set"

[43] Byrd et al. (2012), "Sample Size Selection in Optimization Methods for Machine Learning"

[32] Bollapragada et al. (2017), "Adaptive Sampling Strategies for Stochastic Optimization"

[333] Yao et al. (2020), "ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning"

[291] Thomas et al. (2020), "On the interplay between noise and curvature and its effect on optimization and generalization"

[187] Liu et al. (2020), "Understanding Why Neural Networks Generalize Well Through GSNR of Parameters"

Table C.1: Overview of all Cockpit quantities with a short description and, if necessary, a reference to mathematical definition.

Name	Description	Math
`Loss`	Mini-batch training loss at current iteration, $L_{\mathbb{B}}$	-
`Parameters`	Parameter values $\boldsymbol{\theta}^{(t)}$ at the current iteration	-
`Distance`	L^2 distance from initialization $\|\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(0)}\|_2$	-
`UpdateSize`	Update size of the current iteration $\|\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}\|_2$	-
`GradNorm`	Mini-batch gradient norm $\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\|_2$	-
`Time`	Time of the current iteration (*e.g.* used in benchmark of Appendix C.5)	-
`Alpha`	Normalized step on a noisy quadratic interpolation between two iterates $\boldsymbol{\theta}^{(t)}, \boldsymbol{\theta}^{(t+1)}$	(C.4)
`CABS`	Adaptive batch size for SGD, optimizes expected objective gain per cost, adapted from [21]	(C.6)
`EarlyStopping`	Evidence-based early stopping criterion for SGD, proposed in [199]	(C.11)
`GradHist1d`	Histogram of individual gradient elements, $\{g_{\mathbb{B}}^{(i)}(\theta_j)\}_{i\in\mathbb{B}}^{j=1,\dots,D}$	(C.12)
`GradHist2d`	Histogram of weights and individual gradient elements, $\{(\theta_j, g_{\mathbb{B}}^{(i)}(\theta_j))\}_{i\in\mathbb{B}}^{j=1,\dots,D}$	(C.13)
`NormTest`	Normalized fluctuations of the residual norms $\|\boldsymbol{g}_{\mathbb{B}} - \boldsymbol{g}_{\mathbb{B}}^{(i)}\|$, proposed in [43]	(C.20)
`InnerTest`	Normalized fluctuations of $\boldsymbol{g}_{\mathbb{B}}^{(i)}$'s parallel components along $\boldsymbol{g}_{\mathbb{B}}$, proposed in [32]	(C.26)
`OrthoTest`	Normalized fluctuations of $\boldsymbol{g}_{\mathbb{B}}^{(i)}$'s orthogonal components along $\boldsymbol{g}_{\mathbb{B}}$, proposed in [32]	(C.33)
`HessMaxEV`	Maximum Hessian eigenvalue, $\lambda_{\max}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$, inspired by [333]	(C.34)
`HessTrace`	Exact or approximate Hessian trace, $\mathrm{Tr}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$, inspired by [333]	-
`TICDiag`	Relation between (diagonal) curvature and gradient noise, inspired by [291]	(C.38)
`TICTrace`	Relation between curvature and gradient noise trace, inspired by [291]	(C.37)
`MeanGSNR`	Average gradient signal-to-noise-ratio (GSNR), inspired by [187]	(C.41)

C.3 Mathematical Details

In this section, we want to provide the mathematical background for the instruments described in Table C.1. This complements the more informal description presented in Section 7.2 in the main text, which focused more on the expressiveness of the individual quantities.

C.3.1 Normalized Step Length (`Alpha`)

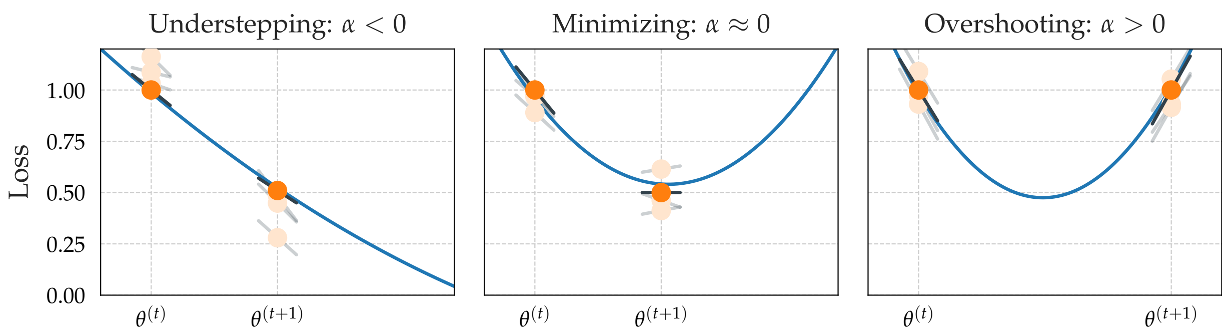


Figure C.1: Motivational sketch for `Alpha`. In each iteration of the optimizer we observe the loss function at two positions $\boldsymbol{\theta}^{(t)}$ and $\boldsymbol{\theta}^{(t+1)}$ (shown in ●). The black lines (—) show the observed slope at this position, which we can get from projecting the gradients onto the current step direction $\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}$. Note, that all four observations (two loss and two slope values) are noisy, due to being computed on a mini-batch. With access to the individual losses and gradients (some samples shown in ●/—), we can estimate their noise level and build a noise-informed quadratic fit (—). Using this fit, we determine whether the optimizer minimizes the local univariate loss (*middle plot*), or whether we understep (*left plot*) or overshoot (*right plot*) the minimum.

Motivation: The goal of `Alpha` is to estimate and quantify the effect that a selected learning rate has on the optimizer's steps. Let's consider the step that the optimizer takes at training iteration t. This parameter update from $\boldsymbol{\theta}^{(t)}$ to $\boldsymbol{\theta}^{(t+1)}$ happens in a one-dimensional space, defined by the update direction $\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)} = \boldsymbol{s}^{(t)}$. The update direction depends on the update rule of the optimizer, *e.g.* for SGD with learning rate η it is simply $\boldsymbol{s}^{(t)} = -\eta \boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}^{(t)})$, see Update Rule 3.3.2.

We build a noise-informed univariate quadratic approximation along this update step ($\boldsymbol{\theta}^{(t)} \longrightarrow \boldsymbol{\theta}^{(t+1)}$) based on the two noisy loss function observations at $\boldsymbol{\theta}^{(t)}$ and $\boldsymbol{\theta}^{(t+1)}$ and the two noisy slope observation at these two points. Examining this quadratic fit, we are able to determine where on this parabola our optimizer steps. Standardizing this, we express a step to the minimum of the loss in the update direction as $\alpha = 0$. Analogously, steps that end short of this minimum result in $\alpha < 0$, and a step over the minimum in $\alpha > 0$. These three different scenarios for `Alpha` are illustrated in Figure C.1 also showing the underlying observations that would lead to them. Figure 7.1 shows the distribution of `Alpha` values for two very different optimization trajectories.

Noisy observations: In order to build an approximation of the loss function in the update direction, we leverage the four observations of the function (and its derivative) that are available in each iteration. Due to the stochasticity of deep learning optimization, we also take into account the noise level of all observations by estimating them (see Section 3.2.1). The first two observations are the mini-batch training losses $L_{\mathbb{B}^{(t)}}(\boldsymbol{\theta}^{(t)})$ and $L_{\mathbb{B}^{(t+1)}}(\boldsymbol{\theta}^{(t+1)})$, which are computed in every standard training loop. Additionally, we consider the slope in the update direction. To compute the slope of the loss function in the direction of the optimizer's update $\boldsymbol{s}^{(t)}$, we project the current gradient along this update direction

$$\mathbb{E}_{\mathbb{B}^{(t)}}\left[\frac{\boldsymbol{s}^{(t)^\top} \boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}^{(t)})}{\|\boldsymbol{s}^{(t)}\|^2}\right] = \frac{1}{B^{(t)}} \sum_{i \in \mathbb{B}^{(t)}} \frac{\boldsymbol{s}^{(t)^\top} \boldsymbol{g}_{\mathbb{B}}^{(i)}(\boldsymbol{\theta}^{(t)})}{\|\boldsymbol{s}^{(t)}\|^2}, \tag{C.1}$$

$$\mathbb{E}_{\mathbb{B}^{(t+1)}}\left[\frac{\boldsymbol{s}^{(t)^\top} \boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}^{(t+1)})}{\|\boldsymbol{s}^{(t)}\|^2}\right] = \frac{1}{B^{(t+1)}} \sum_{i \in \mathbb{B}^{(t+1)}} \frac{\boldsymbol{s}^{(t)^\top} \boldsymbol{g}_{\mathbb{B}}^{(i)}(\boldsymbol{\theta}^{(t+1)})}{\|\boldsymbol{s}^{(t)}\|^2}. \tag{C.2}$$

For all four observations (two loss observations and two observations of the slope), we can also compute their variances by leveraging individual gradients, to estimate the noise levels σ of the observations.

Quadratic fit & normalization: Using our (noisy) observations, we are now ready to build an approximation for the loss as a function of the step size, which we will denote as $f(\tau)$. We assume a quadratic function for f, which follows recent reports for the loss landscape of neural networks [330], *i.e.* a function $f(\tau) = w_1 + w_2\tau + w_3\tau^2$ parameterized by $\boldsymbol{w} \in \mathbb{R}^3$. We further assume a Gaussian likelihood of the form

[330] Xing et al. (2018), "A Walk with SGD"

$$P\left(\tilde{f}|\boldsymbol{w},\boldsymbol{\Phi}\right) = \mathcal{N}\left(\tilde{f};\boldsymbol{\Phi}^\top \boldsymbol{w}, \boldsymbol{Q}\right), \tag{C.3}$$

for observations $\tilde{f}$ of the loss and its slope. The observation matrix $\boldsymbol{\Phi}$ and the noise matrix of the observations $\boldsymbol{Q}$ are

$$\boldsymbol{\Phi} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ \tau_1 & \tau_2 & 1 & 1 \\ \tau_1^2 & \tau_2^2 & 2\tau_1 & 2\tau_2 \end{pmatrix}, \qquad \boldsymbol{Q} = \begin{pmatrix} \sigma_{\tilde{f}_1} & 0 & 0 & 0 \\ 0 & \sigma_{\tilde{f}_2} & 0 & 0 \\ 0 & 0 & \sigma_{\tilde{f}'_1} & 0 \\ 0 & 0 & 0 & \sigma_{\tilde{f}'_2} \end{pmatrix},$$

where τ denotes the position and σ denotes the noise level estimate of the observation. The maximum likelihood solution of Equation (C.3) for the parameters of our quadratic fit is given by

$$\boldsymbol{w} = \left(\boldsymbol{\Phi}\boldsymbol{Q}^{-1}\boldsymbol{\Phi}^\top\right)^{-1} \boldsymbol{\Phi}\boldsymbol{Q}^{-1}\tilde{f}\,. \tag{C.4}$$

Once we have the quadratic fit of the univariate loss function in the update direction, we normalize the scales such that the resulting `Alpha` value expresses the effective step taken by the optimizer sketched in Figure C.1.

Usage: The `Alpha` quantity is related to recent line search approaches [200, 308]. However, instead of searching for an acceptable step by repeated attempts, we instead report the effect of the current step size selection. This could, for example, be used to disentangle the two optimization runs in Figure 7.1. Additionally, this information could also be used to automatically adapt the learning rate during the training process. But, as discussed in Section 7.3.3, it isn't trivial what the "correct" decision is, as it might depend on the optimization problem, the training phase, and other factors. The `Alpha` quantity can, however, provide more insight into what kind of steps are used in well-tuned runs with traditional optimizers such as SGD.

[200] Mahsereci et al. (2017), "Probabilistic Line Searches for Stochastic Optimization"
[308] Vaswani et al. (2019), "Painless Stochastic Gradient: Interpolation, Line-Search, and Convergence Rates"

C.3.2 CABS Criterion: Coupling Adaptive Batch Sizes with Learning Rates (`CABS`)

The CABS criterion, proposed by Balles et al. [21], can be used to adapt the mini-batch size during training with SGD. It relies on the gradient noise and approximately optimizes the objective's expected gain per cost. The adaptation rule is

[21] Balles et al. (2017), "Coupling Adaptive Batch Sizes with Learning Rates"

$$B \leftarrow \eta \frac{\operatorname{Tr}(\boldsymbol{\Sigma}_{P_{\text{true}}}(\boldsymbol{\theta}))}{L_{P_{\text{true}}}(\boldsymbol{\theta})}\,, \tag{C.5}$$

where η is the learning rate and the practical implementation approximates $L_{P_{\text{true}}}(\boldsymbol{\theta}) \approx L_{\mathbb{B}}(\boldsymbol{\theta})$, $\operatorname{Tr}(\boldsymbol{\Sigma}_{P_{\text{true}}}(\boldsymbol{\theta})) \approx \frac{B-1}{B}\operatorname{Tr}(\hat{\boldsymbol{\Sigma}}_{\mathbb{B}}(\boldsymbol{\theta}))$ (compare equations (10, 22) and first paragraph of Section 4 in [21]). This yields the quantity computed in Cockpit's `CABS` instrument,

$$B \leftarrow \eta \frac{\frac{1}{B}\sum_{j=1}^{D}\sum_{i\in\mathbb{B}}\left[g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta}) - g_{\mathbb{B}}(\boldsymbol{\theta})\right]_j^2}{L_{\mathbb{B}}(\boldsymbol{\theta})}\,. \tag{C.6}$$

Usage: The CABS criterion described above suggests a batch size which is optimal under certain assumptions. This suggestion can support practitioners in the batch size selection for their deep learning task.

C.3.3 Early-stopping Criterion for SGD (`EarlyStopping`)

The empirical risk $L_{\mathbb{D}_{\text{train}}}(\boldsymbol{\theta})$, and the mini-batch loss $L_{\mathbb{B}}(\boldsymbol{\theta})$ are only estimators of the target objective $L_{P_{\text{true}}}(\boldsymbol{\theta})$ (see Section 3.2). Mahsereci et al. [199] motivate $P(g_{\mathbb{B},\mathbb{D}}(\boldsymbol{\theta}) \mid g_{P_{\text{true}}}(\boldsymbol{\theta}) = \mathbf{0})$ as a

[199] Mahsereci et al. (2017), "Early Stopping without a Validation Set"

measure for detecting noise in the finite data sets $\mathbb{B}, \mathbb{D}$ due to sampling from P_{true}. They propose an evidence-based criterion for early stopping the training procedure based on mini-batch statistics, and model $P(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}))$ with a sampled diagonal variance approximation,

$$P(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})) \approx \prod_{j=1}^{D} \mathcal{N}\left([\boldsymbol{g}_{P_{\text{true}}}(\boldsymbol{\theta})]_j \, ; \frac{\left[\hat{\boldsymbol{\Sigma}}_{\mathbb{B}}(\boldsymbol{\theta})\right]_{j,j}}{B} \right). \tag{C.7}$$

Their SGD stopping criterion is

$$0 < \frac{2}{D}\left[\log P(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})) - \mathbb{E}_{\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) \sim P(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}))}\left[\log P(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}))\right]\right] \tag{C.8}$$

and translate into

$$0 < 1 - \frac{B}{D}\sum_{j=1}^{D} \frac{[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})]_j^2}{\left[\hat{\boldsymbol{\Sigma}}_{\mathbb{B}}(\boldsymbol{\theta})\right]_{j,j}}\,, \tag{C.9}$$

$$0 < 1 - \frac{B}{D}\sum_{d=1}^{D} \frac{[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})]_d^2}{\frac{1}{B-1}\sum_{i\in\mathbb{B}}\left[\boldsymbol{g}_{\mathbb{B}}^{(i)}(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right]_d^2}\,, \tag{C.10}$$

$$0 < 1 - \frac{B(B-1)}{D}\sum_{d=1}^{D} \frac{[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})]_d^2}{\left(\sum_{i\in\mathbb{B}}\left[\boldsymbol{g}_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})\right]_d^2\right) - B\,[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})]_d^2}\,. \tag{C.11}$$

Cockpit's `EarlyStopping` quantity computes the right-hand side of Equation (C.11).

Usage: The `EarlyStopping` quantity of Cockpit can inform the practitioner that training is about to be completed and the model might be at risk of overfitting.

C.3.4 Individual Gradient Element Histograms (`GradHist1d`, `GradHist2d`)

For the $B \times D$ individual gradient elements, Cockpit's `GradHist1d` instrument displays a histogram of

$$\left\{ \boldsymbol{g}_{\mathbb{B}}^{(i)}(\theta_j) \right\}_{i\in\mathbb{B}, j=1,\dots,D}. \tag{C.12}$$

Cockpit's `GradHist2d` instrument displays a two-dimensional histogram of the $B \times D$ tuples

$$\left\{ \left(\theta_j, \boldsymbol{g}_{\mathbb{B}}^{(i)}(\theta_j) \right) \right\}_{i\in\mathbb{B}, j=1,\dots,D} \tag{C.13}$$

and the marginalized one-dimensional histograms over the parameter and gradient axes.

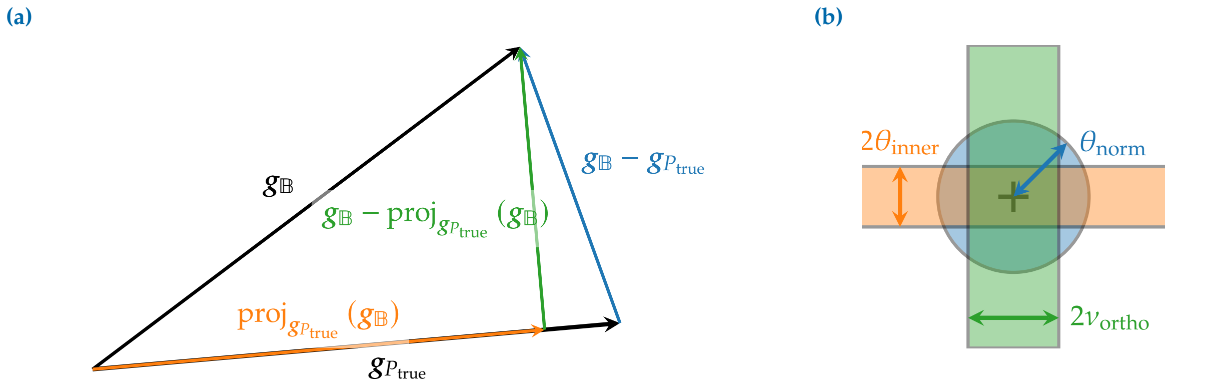

(a)
$g_{\mathbb{B}}$
$g_{\mathbb{B}} - \text{proj}_{g_{P_{\text{true}}}}(g_{\mathbb{B}})$
$g_{\mathbb{B}} - g_{P_{\text{true}}}$
$\text{proj}_{g_{P_{\text{true}}}}(g_{\mathbb{B}})$
$g_{P_{\text{true}}}$
(b)
$2\theta_{\text{inner}}$
θ_{norm}
$2\nu_{\text{ortho}}$

Figure C.2: **Conceptual sketch for gradient tests.** (a) Relevant vectors to formulate the geometric constraints between population and mini-batch gradient probed by the gradient tests. (b) Gradient test visualization in Cockpit.

Usage: Sections 7.3.1 and 7.3.2 provide use cases (identifying data pre-processing issues and vanishing gradients) for both the gradient histogram as well as its layer-wise extension.

C.3.5 Gradient Tests (`NormTest`, `InnerTest`, `OrthoTest`)

Bollapragada et al. [32] and Byrd et al. [43] proposed batch size adaptation schemes based on the gradient noise. They formulate geometric constraints between population and mini-batch gradient and accessible approximations that can be probed to decide whether the mini-batch size should be increased. Mini-batches are assumed to be sampled i.i.d. from P_{true}, it holds that

[32] Bollapragada et al. (2017), "Adaptive Sampling Strategies for Stochastic Optimization"

[43] Byrd et al. (2012), "Sample Size Selection in Optimization Methods for Machine Learning"

$$\mathbb{E}\left[g_{\mathbb{B}}(\boldsymbol{\theta})\right] = g_{P_{\text{true}}}(\boldsymbol{\theta})\,, \tag{C.14}$$

$$\mathbb{E}\left[g_{\mathbb{B}}(\boldsymbol{\theta})^{\top} g_{P_{\text{true}}}(\boldsymbol{\theta})\right] = \|g_{P_{\text{true}}}(\boldsymbol{\theta})\|^2. \tag{C.15}$$

The above works propose enforcing other weaker similarity in expectation during optimization. These geometric constraints reduce to basic vector geometry (see Figure C.2a for an overview of the relevant vectors). We recall their formulation here for consistency and derive the practical versions, which can be computed from training observables and are used in Cockpit (consult Figure C.2b for the visualization).

Usage: All three gradient tests describe the noise level of the gradients. Bollapragada et al. [32] and Byrd et al. [43] adapt the batch size so that the proposed geometric constraints are fulfilled. Practitioners can use the combined gradient test plot, *i.e.* top center plot in Figure 7.2, to monitor gradient noise during training and adjust hyperparameters such as the batch size.

Norm Test (`NormTest`)

[43] Byrd et al. (2012), "Sample Size Selection in Optimization Methods for Machine Learning"

The norm test [43] constrains the residual norm $\|g_{\mathbb{B}}(\boldsymbol{\theta}) - g_{P_{\text{true}}}(\boldsymbol{\theta})\|$, rescaled by $\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|$. This gives rise to a standardized ball of radius $\theta_{\text{norm}} \in (0, \infty)$ around the population gradient, where the mini-batch gradient should reside. Byrd et al. [43] set $\theta_{\text{norm}} = 0.9$ in their experiments and increase the batch size if (in the practical version, see below) the following constraint is not fulfilled

$$\mathbb{E}\left[\frac{\|g_{\mathbb{B}}(\boldsymbol{\theta}) - g_{P_{\text{true}}}(\boldsymbol{\theta})\|^2}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|^2}\right] \leq \theta_{\text{norm}}^2 \,. \tag{C.16}$$

Instead of taking the expectation over mini-batches, Byrd et al. [43] note that the above will be satisfied if

$$\frac{1}{B}\mathbb{E}\left[\frac{\left\|g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta}) - g_{P_{\text{true}}}(\boldsymbol{\theta})\right\|^2}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|^2}\right] \leq \theta_{\text{norm}}^2 \,. \tag{C.17}$$

They propose a practical form of this test,

$$\frac{1}{B(B-1)} \frac{\sum_{i\in\mathbb{B}} \left\|g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta}) - g_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2}{\|g_{\mathbb{B}}(\boldsymbol{\theta})\|^2} \leq \theta_{\text{norm}}^2 \,, \tag{C.18}$$

which can be computed from mini-batch statistics. Rearranging

$$\sum_{i\in\mathbb{B}} \left\|g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta}) - g_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2 = \left(\sum_{i\in\mathbb{B}} \left\|g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})\right\|^2\right) - B\,\|g_{\mathbb{B}}(\boldsymbol{\theta})\|^2 \,, \tag{C.19}$$

we arrive at

$$\frac{1}{B(B-1)}\left[\frac{\sum_{i\in\mathbb{B}} \left\|g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})\right\|^2}{\|g_{\mathbb{B}}(\boldsymbol{\theta})\|^2} - B\right] \leq \theta_{\text{norm}}^2 \tag{C.20}$$

that leverages the norm of both the mini-batch and the individual gradients, which can be aggregated over parameters during a backward pass. Cockpit's `NormTest` corresponds to the maximum radius θ_{norm} for which the above inequality holds.

Inner Product Test (`InnerTest`)

[32] Bollapragada et al. (2017), "Adaptive Sampling Strategies for Stochastic Optimization"

The inner product test [32] constrains the projection of $g_{\mathbb{B}}(\boldsymbol{\theta})$ onto $g_{P_{\text{true}}}(\boldsymbol{\theta})$ (compare Figure C.2a),

$$\operatorname{proj}_{g_{P_{\text{true}}}(\boldsymbol{\theta})}(g_{\mathbb{B}}(\boldsymbol{\theta})) = \frac{g_{\mathbb{B}}(\boldsymbol{\theta})^\top g_{P_{\text{true}}}(\boldsymbol{\theta})}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|^2} g_{P_{\text{true}}}(\boldsymbol{\theta}) \,, \tag{C.21}$$

rescaled by $\|g_{P_\text{true}}(\boldsymbol{\theta})\|$. This restricts the mini-batch gradient to reside in a standardized band of relative width $\theta_\text{inner} \in (0, \infty)$ around the population risk gradient. Bollapragada et al. [32] use $\theta_\text{inner} = 0.9$ (in the practical version, see below) to adapt the batch size if the parallel component's variance does not satisfy the condition

$$\operatorname{Var}\left[\frac{g_\mathbb{B}(\boldsymbol{\theta})^\top g_{P_\text{true}}(\boldsymbol{\theta})}{\|g_{P_\text{true}}(\boldsymbol{\theta})\|^2}\right] = \mathbb{E}\left[\left(\frac{g_\mathbb{B}(\boldsymbol{\theta})^\top g_{P_\text{true}}(\boldsymbol{\theta})}{\|g_{P_\text{true}}(\boldsymbol{\theta})\|^2} - 1\right)^2\right] \leq \theta_\text{inner}^2 \tag{C.22}$$

(note that by Equation (C.15) we have $\mathbb{E}\left[\frac{g_\mathbb{B}(\boldsymbol{\theta})^\top g_{P_\text{true}}(\boldsymbol{\theta})}{\|g_{P_\text{true}}(\boldsymbol{\theta})\|^2}\right] = 1$). Bollapragada et al. [32] bound Equation (C.22) by the individual gradient variance,

$$\frac{1}{B}\operatorname{Var}\left[\frac{g_\mathbb{B}^{(i)}(\boldsymbol{\theta})^\top g_{P_\text{true}}(\boldsymbol{\theta})}{\|g_{P_\text{true}}(\boldsymbol{\theta})\|^2}\right] = \frac{1}{B}\mathbb{E}\left[\left(\frac{g_\mathbb{B}^{(i)}(\boldsymbol{\theta})^\top g_{P_\text{true}}(\boldsymbol{\theta})}{\|g_{P_\text{true}}(\boldsymbol{\theta})\|^2} - 1\right)^2\right] \leq \theta_\text{inner}^2 \,. \tag{C.23}$$

They then propose a practical form of Equation (C.23), which uses the mini-batch sample variance,

$$\begin{aligned}\frac{1}{B}\operatorname{Var}\left[\frac{g_\mathbb{B}^{(i)}(\boldsymbol{\theta})^\top g_\mathbb{B}(\boldsymbol{\theta})}{\|g_\mathbb{B}(\boldsymbol{\theta})\|^2}\right] &= \frac{1}{B(B-1)}\left[\sum_{i\in\mathbb{B}}\left(\frac{g_\mathbb{B}^{(i)}(\boldsymbol{\theta})^\top g_\mathbb{B}(\boldsymbol{\theta})}{\|g_\mathbb{B}(\boldsymbol{\theta})\|^2} - 1\right)^2\right]\\ &\leq \theta_\text{inner}^2 \,.\end{aligned} \tag{C.24}$$

Expanding

$$\sum_{i\in\mathbb{B}}\left(\frac{g_\mathbb{B}^{(i)}(\boldsymbol{\theta})^\top g_\mathbb{B}(\boldsymbol{\theta})}{\|g_\mathbb{B}(\boldsymbol{\theta})\|^2} - 1\right)^2 = \frac{\sum_{i\in\mathbb{B}}\left(g_\mathbb{B}^{(i)}(\boldsymbol{\theta})^\top g_\mathbb{B}(\boldsymbol{\theta})\right)^2}{\|g_\mathbb{B}(\boldsymbol{\theta})\|^4} - B \tag{C.25}$$

and inserting Equation (C.25) into Equation (C.24) yields

$$\frac{1}{B(B-1)}\left[\frac{\sum_{i\in\mathbb{B}}\left(g_\mathbb{B}^{(i)}(\boldsymbol{\theta})^\top g_\mathbb{B}(\boldsymbol{\theta})\right)^2}{\|g_\mathbb{B}(\boldsymbol{\theta})\|^4} - B\right] \leq \theta_\text{inner}^2 \,. \tag{C.26}$$

It relies on pairwise scalar products between individual gradients, which can be aggregated over layers during backpropagation. Cockpit's `InnerTest` computes the maximum band width θ_inner that satisfies Equation (C.26).

Orthogonality Test (`OrthoTest`)

[32] Bollapragada et al. (2017), "Adaptive Sampling Strategies for Stochastic Optimization"

In contrast to the inner product test (Appendix C.3.5) which constrains the projection (Equation (C.21)), the orthogonality test [32] constrains the orthogonal part (see Figure C.2a)

$$g_{\mathbb{B}}(\boldsymbol{\theta}) - \operatorname{proj}_{g_{P_{\text{true}}}(\boldsymbol{\theta})}\left(g_{\mathbb{B}}(\boldsymbol{\theta})\right) , \tag{C.27}$$

rescaled by $\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|$. This restricts the mini-batch gradient to a standardized band of relative width $\nu_{\text{ortho}} \in (0, \infty)$ parallel to the population gradient. Bollapragada et al. [32] use $\nu = \tan(80°) \approx 5.84$ (in the practical version, see below) to adapt the batch size if the following condition is violated,

$$\mathbb{E}\left[\left\|\frac{g_{\mathbb{B}}(\boldsymbol{\theta}) - \operatorname{proj}_{g_{P_{\text{true}}}(\boldsymbol{\theta})}\left(g_{\mathbb{B}}(\boldsymbol{\theta})\right)}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|}\right\|^2\right] \leq \nu_{\text{ortho}}^2 . \tag{C.28}$$

Expanding the norm, and inserting Equation (C.21), this simplifies to

$$\mathbb{E}\left[\left\|\frac{g_{\mathbb{B}}(\boldsymbol{\theta})}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|} - \frac{g_{\mathbb{B}}(\boldsymbol{\theta})^\top g_{P_{\text{true}}}(\boldsymbol{\theta})}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|^2} \frac{g_{P_{\text{true}}}(\boldsymbol{\theta})}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|}\right\|^2\right] \leq \nu_{\text{ortho}}^2 , \tag{C.29}$$

$$\mathbb{E}\left[\frac{\|g_{\mathbb{B}}(\boldsymbol{\theta})\|^2}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|^2} - \frac{\left(g_{\mathbb{B}}(\boldsymbol{\theta})^\top g_{P_{\text{true}}}(\boldsymbol{\theta})\right)^2}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|^4}\right] \leq \nu_{\text{ortho}}^2 . \tag{C.30}$$

Bollapragada et al. [32] bound this inequality using individual gradients instead,

$$\frac{1}{B}\mathbb{E}\left[\left\|\frac{g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|^2} - \frac{g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})^\top g_{P_{\text{true}}}(\boldsymbol{\theta})}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|^2} \frac{g_{P_{\text{true}}}(\boldsymbol{\theta})}{\|g_{P_{\text{true}}}(\boldsymbol{\theta})\|}\right\|^2\right] \leq \nu_{\text{ortho}}^2 . \tag{C.31}$$

They propose the practical form

$$\frac{1}{B(B-1)}\mathbb{E}\left[\left\|\frac{g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})}{\|g_{\mathbb{B}}(\boldsymbol{\theta})\|} - \frac{g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})^\top g_{\mathbb{B}}(\boldsymbol{\theta})}{\|g_{\mathbb{B}}(\boldsymbol{\theta})\|^2} \frac{g_{\mathbb{B}}(\boldsymbol{\theta})}{\|g_{\mathbb{B}}(\boldsymbol{\theta})\|}\right\|^2\right] \leq \nu_{\text{ortho}}^2 , \tag{C.32}$$

which simplifies to

$$\frac{1}{B(B-1)}\sum_{i\in\mathbb{B}}\left(\frac{\|g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})\|^2}{\|g_{\mathbb{B}}(\boldsymbol{\theta})\|^2} - \frac{\left(g_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})^\top g_{\mathbb{B}}(\boldsymbol{\theta})\right)^2}{\|g_{\mathbb{B}}(\boldsymbol{\theta})\|^4}\right) \leq \nu_{\text{ortho}}^2 . \tag{C.33}$$

It relies on pairwise scalar products between individual gradients which can be aggregated over layers during a backward pass. Cockpit's `OrthoTest` computes the maximum band width ν_{ortho} which satisfies Equation (C.33).

Relation to acute angle test: Recently, a novel "acute angle test" was proposed by Bahamou and Goldfarb [15]. While the theoretical constraint between $g_{\mathbb{B}}(\boldsymbol{\theta})$ and $g_{P_{\text{true}}}(\boldsymbol{\theta})$ differs from the orthogonality test, the practical versions coincide. Hence, we do not incorporate the acute angle here.

[15] Bahamou et al. (2019), "A Dynamic Sampling Adaptive-SGD Method for Machine Learning"

C.3.6 Hessian Maximum Eigenvalue (`HessMaxEV`)

The Hessian's maximum eigenvalue $\lambda_{\max}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$ is computed with an iterative eigensolver from Hessian-vector products through PyTorch's automatic differentiation [229]. Like Yao et al. [333], we employ power iterations with similar default stopping parameters (stop after at most 100 iterations, or if the iterate does converged with a relative and absolute tolerance of $10^{-3}, 10^{-6}$, respectively) to compute $\lambda_{\max}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$ through the `HessMaxEV` quantity in Cockpit.

[229] Pearlmutter (1994), "Fast Exact Multiplication by the Hessian"

[333] Yao et al. (2020), "ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning"

In principle, more sophisticated eigensolvers (for example Arnoldi's method) could be applied to converge in fewer iterations or compute eigenvalues other than the leading ones. Warsa et al. [316] empirically demonstrate that the FLOP ratio between power iteration and implicitly restarted Arnoldi method can reach values larger than 100. While we can use such a beneficial method on a CPU through `scipy.sparse.linalg.eigsh` we are restricted to the GPU-compatible power iteration for GPU training. We expect that extending the support of popular machine learning libraries like PyTorch for such iterative eigensolvers on GPUs can help to save computation time.

[316] Warsa et al. (2004), "Krylov Subspace Iterations for Deterministic k-Eigenvalue Calculations"

$$\lambda_{\max}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})) = \max_{\|\boldsymbol{v}\|=1} \|\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\boldsymbol{v}\| = \max_{\boldsymbol{v}\in\mathbb{R}^D} \frac{\boldsymbol{v}^\top \boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\boldsymbol{v}}{\boldsymbol{v}^\top\boldsymbol{v}}. \tag{C.34}$$

Usage: The Hessian's maximum eigenvalue describes the loss surface's sharpest direction and thus provides an understanding of the current loss landscape. Additionally, in convex optimization, the largest Hessian eigenvalue crucially determines the appropriate step size [260]. In Section 7.4, we can observe that although training seems stuck in the very first few iterations progress is visible when looking at the maximum Hessian eigenvalue `HessMaxEV`.

[260] Schmidt (2014), "Convergence rate of stochastic gradient with constant step size"

C.3.7 Hessian Trace (`HessTrace`)

[333] Yao et al. (2020), "ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning"

[229] Pearlmutter (1994), "Fast Exact Multiplication by the Hessian"

In comparison to Yao et al. [333], who leverage Hessian-vector products [229] to estimate the Hessian trace, we compute the exact value $\operatorname{Tr}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$ with the `HessTrace` quantity in Cockpit by aggregating the output of BackPACK's `DiagHessian` extension, which computes the diagonal entries of $\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})$. Alternatively, the trace can also be estimated from the generalized Gauss-Newton matrix, or an MC-sampled approximation thereof.

Usage: The Hessian trace equals the sum of the eigenvalues and thus provides a notion of "average curvature" of the current loss landscape. It has long been theorized and discussed that curvature and generalization performance may be linked [*e.g.*, 126] (see also Section 2.3.3).

[126] Hochreiter et al. (1997), "Flat Minima"

C.3.8 Takeuchi Information Criterion (TIC) (`TICDiag`, `TICTrace`)

[291] Thomas et al. (2020), "On the interplay between noise and curvature and its effect on optimization and generalization"

[286] Takeuchi (1976), "The distribution of information statistics and the criterion of goodness of fit of models"

Recent work by Thomas et al. [291] suggests that optimizer convergence speed and generalization is mainly influenced by curvature and gradient noise; and hence their interaction is crucial to understand the generalization and optimization behavior of deep neural networks. They reinvestigate the Takeuchi Information Criterion (TIC) [286], an estimator for the generalization gap in overparameterized maximum likelihood estimation. At a local minimum $\boldsymbol{\theta}^\star$, the generalization gap is estimated by the TIC

$$\frac{1}{N}\operatorname{Tr}\left(\boldsymbol{H}_{P_\text{true}}(\boldsymbol{\theta}^\star)^{-1}\boldsymbol{C}_{P_\text{true}}(\boldsymbol{\theta}^\star)\right)\,, \tag{C.35}$$

where $\boldsymbol{H}_{P_\text{true}}(\boldsymbol{\theta}^\star)$ is the population Hessian and $\boldsymbol{C}_{P_\text{true}}(\boldsymbol{\theta}^\star)$ is the gradient's uncentered second moment,

$$\boldsymbol{C}_{P_\text{true}}(\boldsymbol{\theta}^\star) = \int \nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{x};\boldsymbol{\theta}^\star),y)\left(\nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{x};\boldsymbol{\theta}^\star),y)\right)^\top \mathrm{d}P_\text{true}(\boldsymbol{x},y)\,. \tag{C.36}$$

Both matrices are inaccessible in practice. In their experiments, Thomas et al. [291] propose the approximation $\operatorname{Tr}(\boldsymbol{C})/\operatorname{Tr}(\boldsymbol{H})$ for $\operatorname{Tr}(\boldsymbol{H}^{-1}\boldsymbol{C})$. They also replace the Hessian by the Fisher as it is easier to compute. With these practical simplifications, they investigate the TIC of trained neural networks where the curvature and noise matrix are evaluated on a large data set.

The TIC provided in Cockpit differs from this setting, since by design we want to observe quantities during training, while avoiding additional model predictions. Also, BackPACK provides access to the Hessian; hence we don't need to use the Fisher. We propose the following two approximations of the TIC from a mini-batch:

- `TICTrace`: Uses the approximation of Thomas et al. [291] which replaces the matrix-product trace by the product of traces,

$$\frac{\operatorname{Tr}\left(\boldsymbol{C}_{\mathbb{B}}(\boldsymbol{\theta})\right)}{\operatorname{Tr}\left(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\right)} = \frac{\frac{1}{B}\sum_{i\in\mathbb{B}}\|\boldsymbol{g}_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})\|^2}{\operatorname{Tr}\left(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\right)}\,. \tag{C.37}$$

- `TICDiag`: Uses a diagonal approximation of the Hessian, which is cheap to invert,

$$\operatorname{Tr}\left(\operatorname{diag}\left(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\right)^{-1}\boldsymbol{C}_{\mathbb{B}}(\boldsymbol{\theta})\right) = \frac{1}{B}\sum_{j=1}^{D}\left[\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\right]_{j,j}^{-1}\left[\sum_{i\in\mathbb{B}}\boldsymbol{g}_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})^{\odot 2}\right]_j\,. \tag{C.38}$$

Usage: The TIC is a proxy for the model's generalization gap, see Thomas et al. [291].

[291] Thomas et al. (2020), "On the interplay between noise and curvature and its effect on optimization and generalization"

C.3.9 Gradient Signal-to-Noise-Ratio (`MeanGSNR`)

The gradient signal-to-noise-ratio $\operatorname{GSNR}(\theta_j) \in \mathbb{R}$ for a single parameter θ_j is defined as

$$\operatorname{GSNR}(\theta_j) = \frac{\mathbb{E}_{(\boldsymbol{x},y)\sim P_{\text{true}}(\boldsymbol{x},y)}\left[\left[\nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{x};\boldsymbol{\theta}),y)\right]_j\right]^2}{\operatorname{Var}_{(\boldsymbol{x},y)\sim P_{\text{true}}(\boldsymbol{x},y)}\left[\left[\nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{x};\boldsymbol{\theta}),y)\right]_j\right]} = \frac{\left[\boldsymbol{g}_{P_{\text{true}}}(\boldsymbol{\theta})\right]_j^2}{\left[\boldsymbol{\Sigma}_{P_{\text{true}}}(\boldsymbol{\theta})\right]_{j,j}}\,. \tag{C.39}$$

Liu et al. [187] use it to explain generalization properties of models in the early training phase. We apply their estimation to mini-batches,

[187] Liu et al. (2020), "Understanding Why Neural Networks Generalize Well Through GSNR of Parameters"

$$\operatorname{GSNR}(\theta_j) \approx \frac{\left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right]_j^2}{\frac{B-1}{B}\left[\hat{\boldsymbol{\Sigma}}_{\mathbb{B}(\boldsymbol{\theta})}\right]_{j,j}} = \frac{\left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right]_j^2}{\frac{1}{B}\left(\sum_{i\in\mathbb{B}}\left[\boldsymbol{g}_{\mathbb{B}}^{(i)}(\boldsymbol{\theta})\right]_j^2\right) - \left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right]_j^2}\,. \tag{C.40}$$

Inspired by Liu et al. [187], Cockpit's `MeanGSNR` computes the average GSNR over all parameters,

$$\frac{1}{D}\sum_{j=1}^{D}\operatorname{GSNR}(\theta_j)\,. \tag{C.41}$$

Usage: The GSNR describes the gradient noise level which is influenced, among other things, by the batch size. Using the

`MeanGSNR`, perhaps in combination with the gradient tests or the CABS criterion could provide practitioners a clearer picture of suitable batch sizes for their particular problem. As shown by Liu et al. [187], the GSNR is also linked to generalization of neural networks.

C.4 Additional Experiments

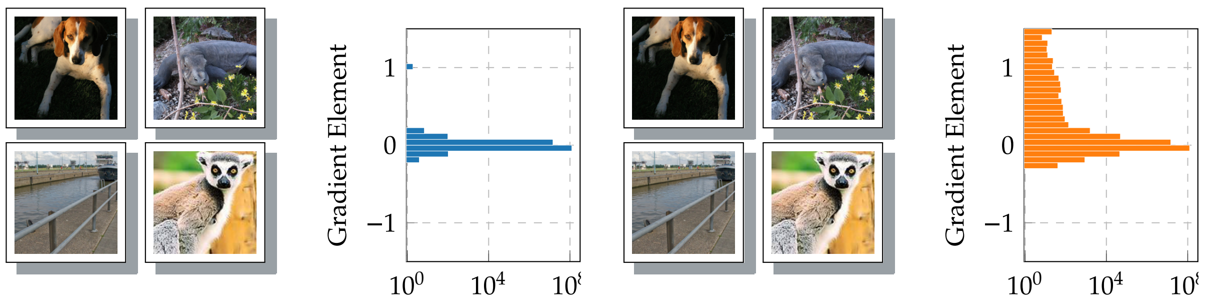
In this section, we present additional experiments and use cases that showcase COCKPIT's utility. Appendix C.4.1 shows that COCKPIT is able to scale to larger data sets by running the experiment with incorrectly scaled data (see Section 7.3.1) on IMAGENET instead of CIFAR-10. Appendix C.4.2 provides another concrete use case similar to Figure 7.1: detecting regularization during training.

C.4.1 Incorrectly Scaled Data for ImageNet

[70] Deng et al. (2009), "ImageNet: A Large-Scale Hierarchical Image Database"
[270] Simonyan et al. (2015), "Very Deep Convolutional Networks for Large-Scale Image Recognition"

We repeat the experiment of Section 7.3.1 on the IMAGENET [70] data set instead of CIFAR-10. We also use a larger neural network model, switching from DEEPOBS' 3C3D to VGG16 [270]. This demonstrates that COCKPIT is able to scale to both larger models and data sets. The input size of the images is almost fifty times larger (224×224 instead of 32×32). The model size increased by roughly a factor of 150 (VGG16 for IMAGENET has roughly 138 million parameters, 3C3D has less than a million).

Similar to the example shown in the main text, the gradients are affected by the scaling introduced via the input images, albeit less drastically (see Figure 7.3). Due to the gradient scaling, default optimization hyperparameters might not work well anymore for the model using the raw input data.



(a) Normalized Data **(b)** Raw Data

Figure C.3: Same inputs, different gradients on IMAGENET. This is structurally the same plot as Figure 7.3, but using IMAGENET and VGG16. (a) *normalized* ($[0, 1]$) and (b) *raw* ($[0, 255]$) images look identical in auto-scaled front-ends like MATPLOTLIB's `imshow`. The gradient distribution on the VGG16 model, however, is affected by this scaling.

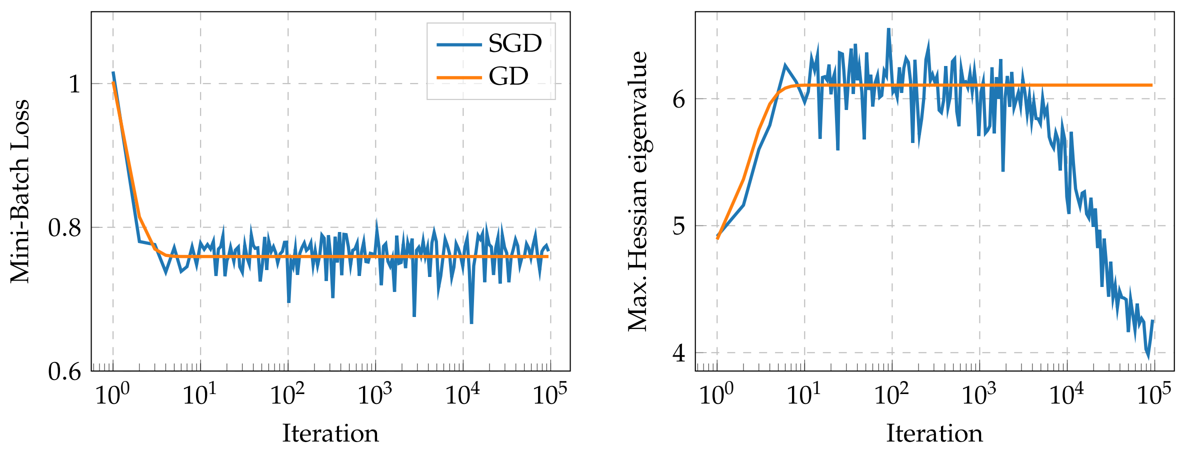


Figure C.4: Observing implicit regularization of the optimizer with Cockpit through a comparison of SGD and GD on a synthetic problem inspired by [96, 212] (details in the text). *Left:* The mini-batch loss of both optimizers looks similar. *Right:* Noise due to mini-batching regularizes the Hessian's maximum eigenvalue in stages where the loss suggests that training has converged.

C.4.2 Detecting Implicit Regularization of the Optimizer

In non-convex optimization, optimizers can converge to local minima which might have different properties. Here, we illustrate this by investigating the effect of sub-sampling noise on a simple task from [96, 212].

[96] Ginsburg (2020), "On regularization of gradient descent, layer imbalance and flat minima"
[212] Mulayoff et al. (2020), "Unique Properties of Flat Minima in Deep Networks"

We generate synthetic data $\mathbb{D}_{\text{train}} = \{(x^{(i)}, y^{(i)}) \in \mathbb{R} \times \mathbb{R}\}_{i=1}^{N=100}$ for a regression task with $x \sim \mathcal{N}(0;1)$ with noisy observations $y = 1.4x + \epsilon$ where $\epsilon \sim \mathcal{N}(0;1)$. The model is a scalar network with parameters $\boldsymbol{\theta} = \begin{pmatrix} w_1 & w_2 \end{pmatrix}^\top \in \mathbb{R}^2$, initialized at $\boldsymbol{\theta}^{(0)} = \begin{pmatrix} 0.1 & 1.7 \end{pmatrix}^\top$, that produces predictions via $f(x;\boldsymbol{\theta}) = w_2 w_1 x$. We seek to minimize the mean squared error

$$L_{\mathbb{D}_{\text{train}}}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\left(f(x^{(i)};\boldsymbol{\theta}) - y^{(i)}\right)^2 \tag{C.42}$$

and compare SGD ($B = 95$) with GD ($B = N = 100$) at a learning rate of 0.1 (see Figure C.4).

We observe that the loss of both SGD and GD is almost identical. Using a noisy gradient regularizes the Hessian's maximum eigenvalue though. It decreases in later stages where the loss curve suggests that training has converged. This regularization effect constitutes an important phenomenon that cannot be observed by monitoring only the loss.

C.5 Implementation Details and Additional Benchmarks

In this section, we provide more details about our implementation (Appendix C.5.1) to access the desired quantities with as little overhead as possible. Additionally, we present more benchmarks for individual instruments (Appendix C.5.2) and Cockpit configurations (Appendix C.5.2). These are similar but extended versions of the ones presented in Figures 7.6a and 7.6b in the main text. Lastly, we benchmark different implementations of computing the two-dimensional gradient histogram (Appendix C.5.3), identifying a computational bottleneck for its current GPU implementation.

Hardware details: Throughout this paper, we conducted benchmarks on the following setup

- **CPU:** Intel Core i7-8700K CPU @ 3.70 GHz × 12 (32 GB)
- **GPU:** NVIDIA GeForce RTX 2080 Ti (11 GB)

Test problem details: The experiments in this paper rely mostly on optimization problems provided by the DeepOBS benchmark suite, see Appendix A.1 or [262]. If not stated otherwise, we use the default training details suggested by DeepOBS.

[262] Schneider et al. (2019), "DeepOBS: A Deep Learning Optimizer Benchmark Suite"

C.5.1 Hooks & Memory Benchmarks

To improve memory consumption, we compact information during the backward pass by adding hooks to the neural network's layers. These are executed after BackPACK extensions and have access to the quantities computed therein. They compress information to what is requested by a quantity and free the memory occupied by BackPACK buffers. Such savings primarily depend on the parameter distribution over layers, and are bigger for more balanced architectures (compare Figure C.5).

Example: Say, we want to compute a histogram over the $B \times D$ individual gradient elements of a network. Suppose that $B = 128$ and the model is DeepOBS's CIFAR-10 3c3d test problem (P4 in Appendix A.1) with $895,210$ parameters. Given that every parameter is stored in single precision, the model requires $895,210 \times 4$ Bytes ≈ 3.41 MB. Storing the individual gradients will require $128 \times 895,210 \times 4$ Bytes ≈ 437 MB (for larger networks this quickly exceeds the available memory as the individual gradients occupy B times the model size). If instead, the layer-wise individual gradients are condensed into histograms of negligible size and immediately freed afterwards during backpropagation, the maximum memory overhead reduces to storing the individual gradients of the largest layer. For our example, the largest layer has $589,824$ parameters, and the associated individual gradients will require $128 \times 589,824 \times 4\,\text{Bytes} \approx 288\,\text{MB}$, saving roughly 149 MB of RAM. In practice, we observe these expected savings, see Figure C.5c.

C.5.2 Additional Run Time Benchmarks

Individual Instrument Overhead

To estimate the computational overhead for individual instruments, we run Cockpit with that instrument for 32 iterations, tracking at every step. Training proceeds with the default batch size specified by the DeepOBS problem and uses SGD with learning rate 10^{-3}. We measure the time between iterations 1 and 32, and average for the overhead per step. Every such estimate is repeated over 10 random seeds to obtain mean and error bars as reported in Figure 7.6a.

Note that this protocol does *not* include initial overhead for setting up data loading and also does *not* include the time for evaluating train/test loss on a larger data set, which is usually done by practitioners. Hence, we even expect the shown overheads to be smaller in a conventional training loop which includes the above steps.

Individual overhead on GPU versus CPU: Figure C.6 and Figure C.7 show the individual overhead for four different DeepOBS problems (P3, P4, P9, and P10 in Appendix A.1) on GPU and CPU, respectively. The left part of Figure C.6c corresponds to Figure 7.6a. Right panels show the expensive quantities, which we omitted in the main text as they were expected to be expensive due to their computational work (`HessMaxEV`) or bottlenecks in the implementation (`GradHist2d`, see Appendix C.5.3 for details). We see that they are in many cases equally or more expensive than computing all other instruments. Another expected feature of the GPU-to-CPU comparison is that parallelism on the CPU is significantly less pronounced. Hence, we observe an increased overhead for all quantities that contain non-linear transformations and contractions of the high-dimensional individual gradients, or require additional backpropagations (curvature).

Configuration Overhead

For the estimation of different Cockpit configuration overheads, we use almost the same setting as described above, training for 512 iterations and tracking only every specified interval.

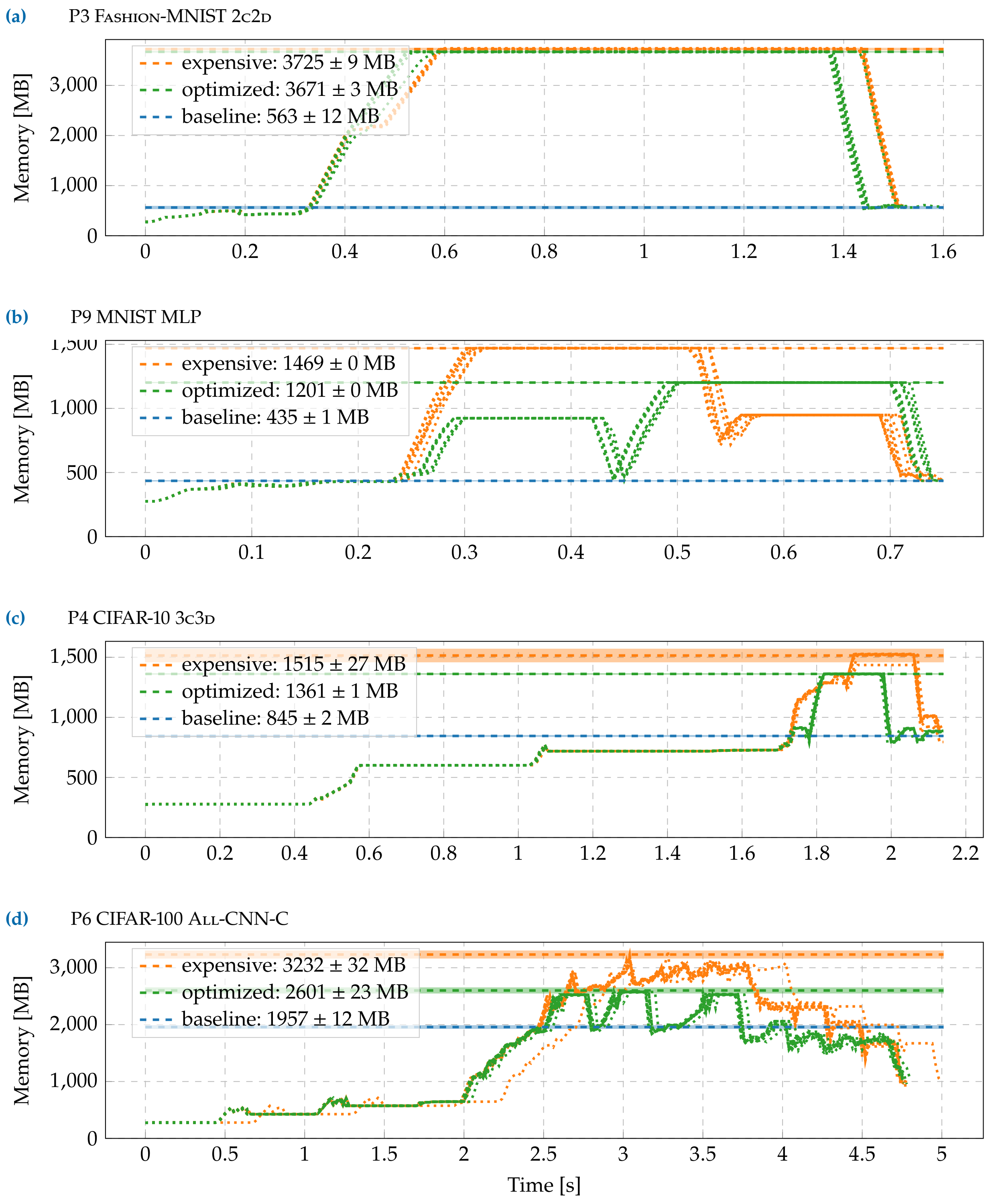
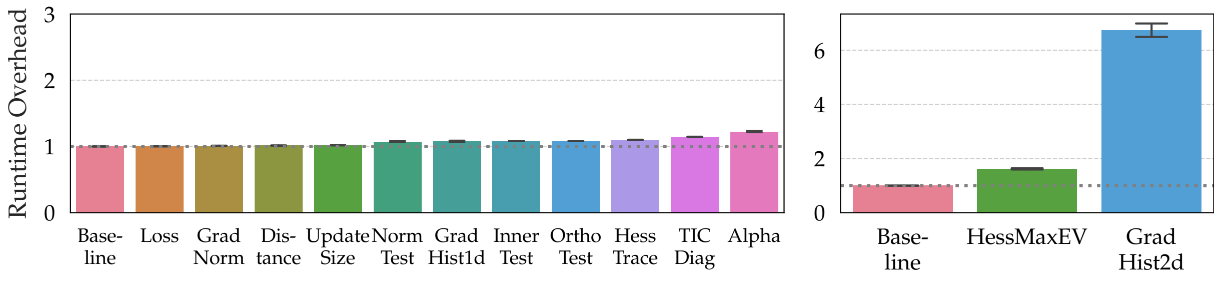
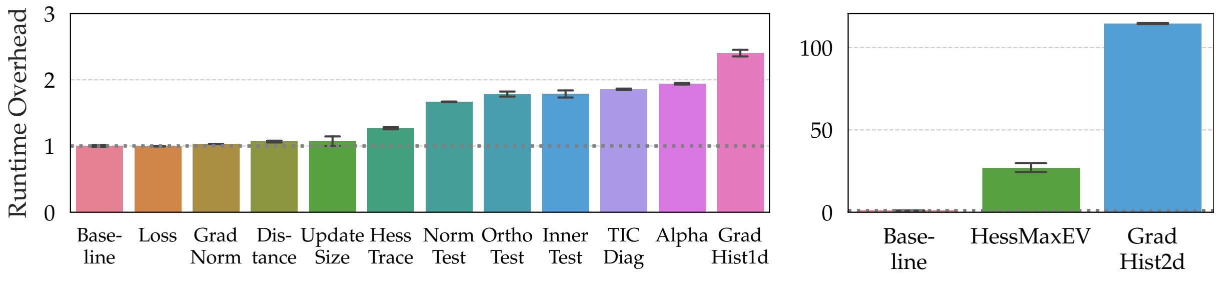


Figure C.5: Memory consumption and savings with hooks during one forward-backward step on a CPU for different DeepOBS problems. We compare three settings; i) without Cockpit (*baseline*); ii) Cockpit with `GradHist1d` with BackPACK (*expensive*); iii) Cockpit with `GradHist1d` with BackPACK and additional hooks (*optimized*). Peak memory consumptions are highlighted by horizontal dashed bars and shown in the legend. Shaded areas, if visible, fill two standard deviations above and below the mean value, all of them result from ten independent runs. Dotted lines indicate individual runs. Our optimized approach allows to free obsolete tensors during backpropagation and thereby reduces memory consumption. From top to bottom: the effect is less pronounced for architectures that concentrate the majority of parameters in a single layer ((a) 3, 274, 634 total, 3, 211, 264 largest layer) and increases for more balanced networks ((b) 1, 336, 610 total, 784, 000 largest layer, (c): 895, 210 total, 589, 824 largest layer).

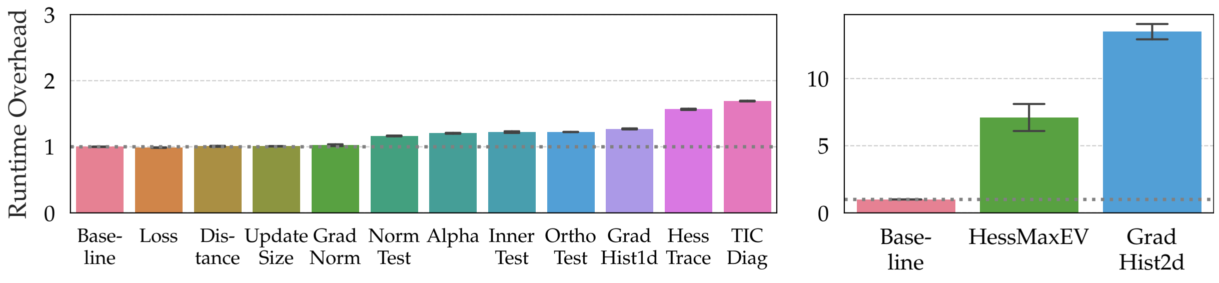
(a) Computational overhead for MNIST Log. Reg. **(GPU)**



(b) Computational overhead for MNIST MLP **(GPU)**



(c) Computational overhead for CIFAR-10 3c3d **(GPU)**



(d) Computational overhead for Fashion-MNIST 2c2d **(GPU)**

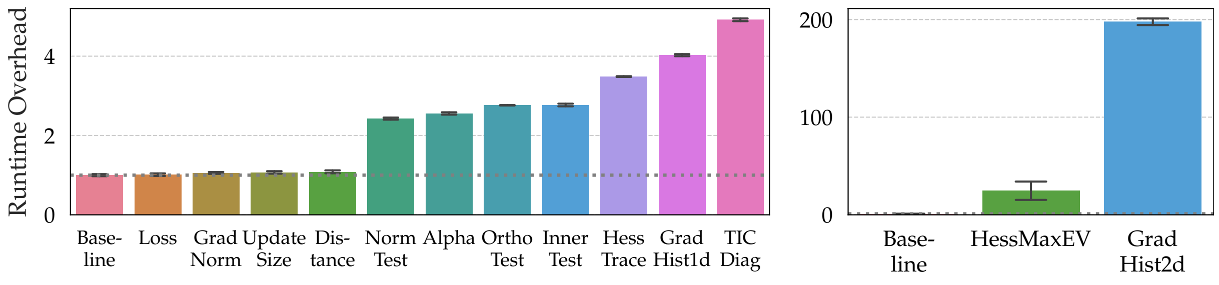


Figure C.6: Individual overhead of Cockpit's instruments on GPU for four different problems. All run times are shown as multiples of the *baseline* without tracking. Expensive quantities are displayed in separate panels on the right. Experimental details in the text.

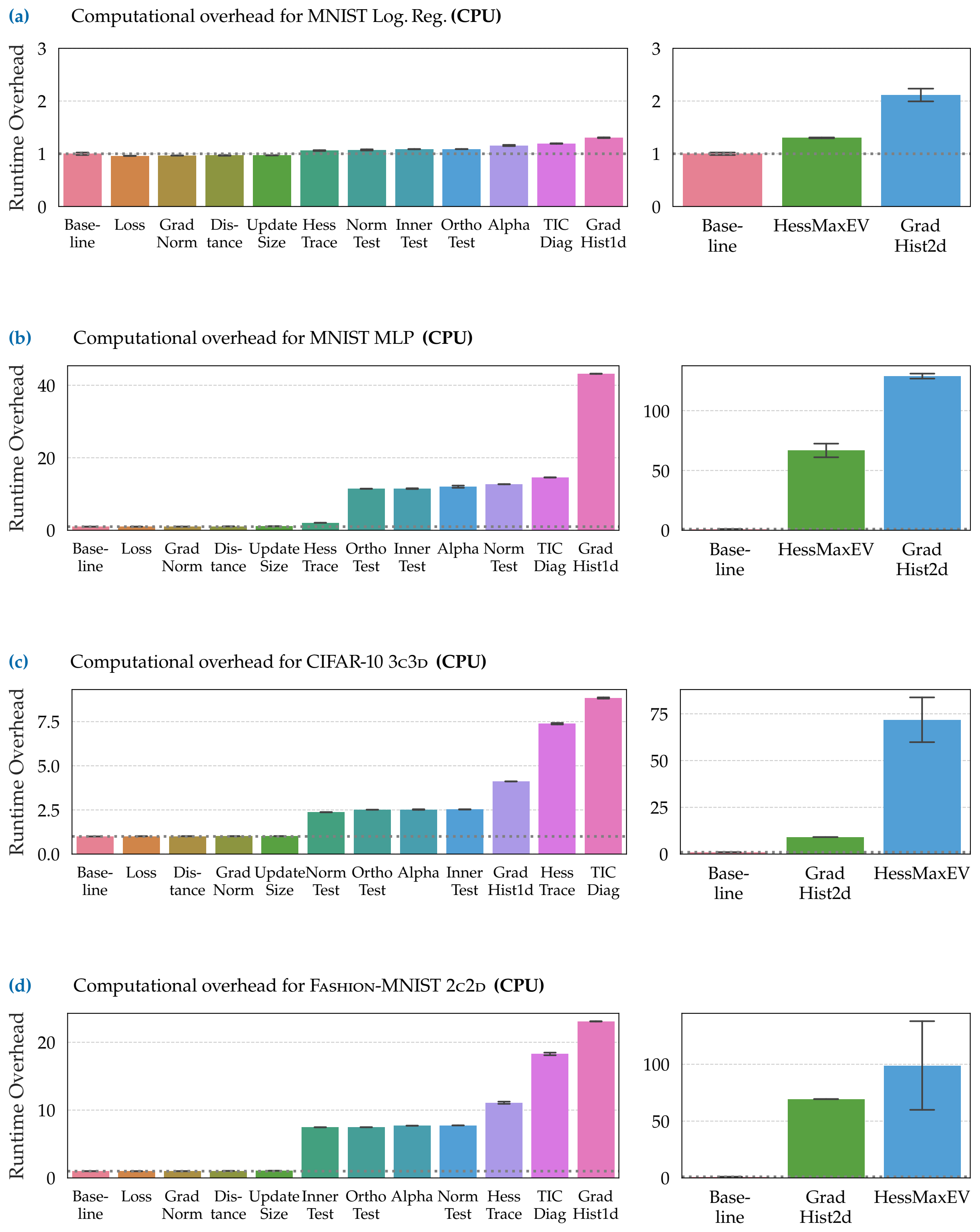


Figure C.7: Individual overhead of Cockpit's instruments on CPU for four different problems. All run times are shown as multiples of the *baseline* without tracking. Expensive quantities are displayed in separate panels on the right. Experimental details in the text.

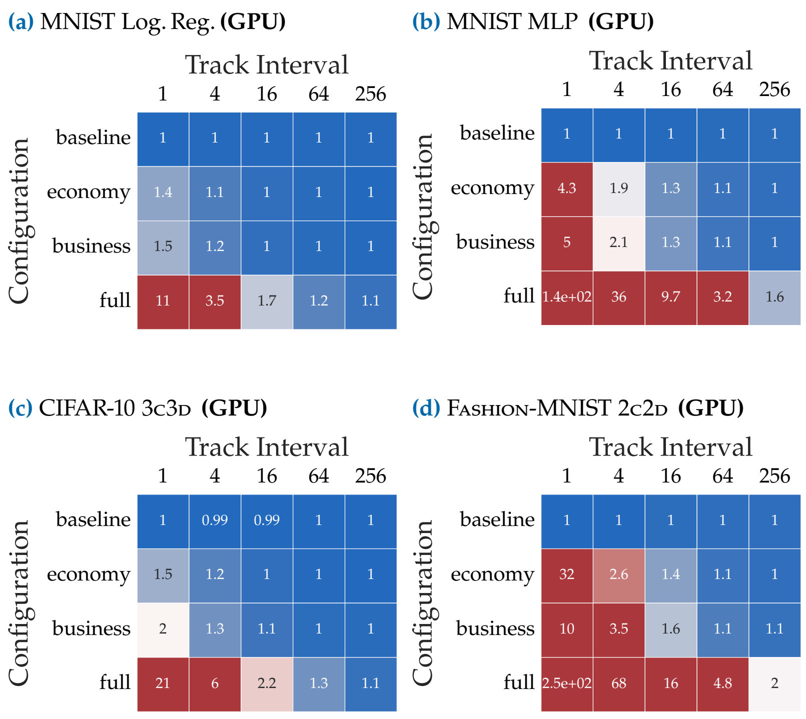


Figure C.8: Overhead of Cockpit configurations on GPU for four different problems with varying tracking interval. Color bar is the same as in Figure 7.6.

Configuration overhead on GPU vs. CPU: Figures C.8 and C.9 show the overhead for four different DeepOBS problems. The bottom left part of Figure C.8 corresponds to Figure 7.6b. In general, we observe that increased parallelism can be exploited on a GPU, leading to smaller overheads compared to a CPU.

Cockpit can even scale to significantly larger problems, such as a ResNet-50 [117] on ImageNet-like data. Figure C.10 shows the computational overhead for different tracking intervals on such a large-scale problem. Using the *economy* configuration, we can achieve our self-imposed goal of at most doubling the run time even when tracking every fourth step. More extensive configurations (such as the *full* set) would indeed have almost prohibitively large costs associated. However, these costs could be dramatically reduced when one decides to only inspect a part of the network using Cockpit. Note, individual gradients are not properly defined when using batch norm, therefore, we replaced these batch norm layers with identity layers when using the ResNet-50.

[117] He et al. (2016), "Deep Residual Learning for Image Recognition"

C.5.3 Performance of Two-dimensional Histograms:

Both one- and two-dimensional histograms require $B \times D$ elements be accessed, and hence perform similarly. However, we observed different behavior on GPU and decided to omit the two-dimensional histogram's run time in the main text. As explained here, this performance lack is not fundamental, but a shortcoming

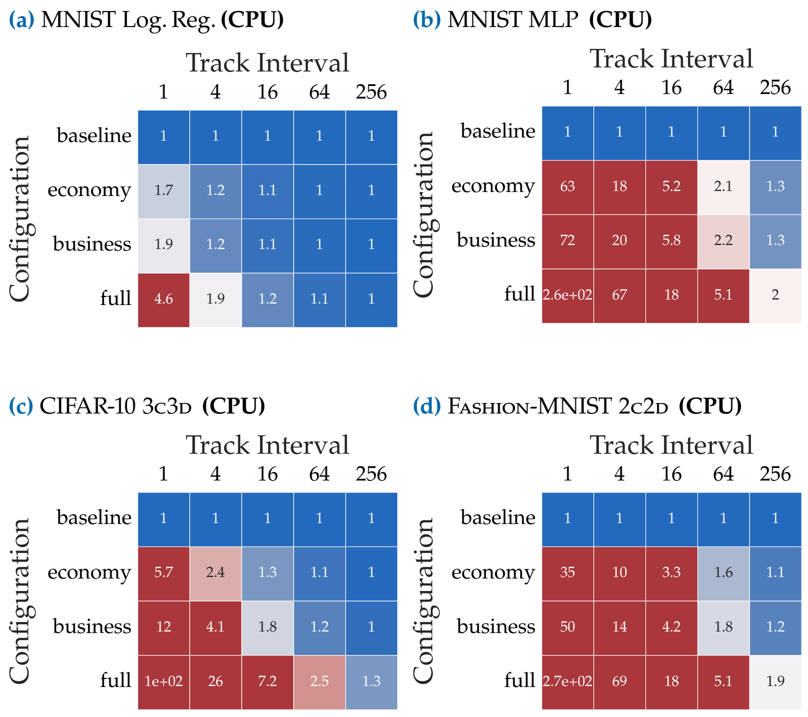


Figure C.9: Overhead of Cockpit configurations on CPU for four different problems with varying tracking interval. Color bar is the same as in Figure 7.6.

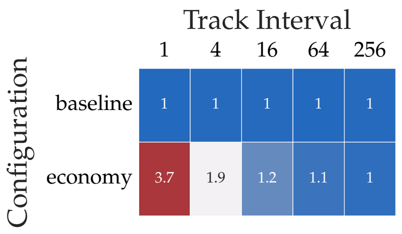


Figure C.10: Overhead of Cockpit configurations on GPU for ResNet-50 on ImageNet. Cockpit's instruments scale efficiently even to very large problems (here: 1000 classes, $(3, 224, 224)$-sized inputs, and a batch size of 64. For individual gradients to be defined, we replaced the batch norm layers of the ResNet-50 model with identities.) Color bar is the same as in Figure 7.6.

of the GPU implementation. PyTorch provides built-in functionality for computing one-dimensional histograms at the time of writing, but is not yet featuring multi-dimensional histograms. We experimented with three implementations:

- **PyTorch (third party):** A third party implementation[2] under review for being integrated into PyTorch [3]. It relies on `torch.bincount`, which uses `atomicAdds` that represent a bottleneck for histograms where most counts are contained in one bin.[4] This occurs often for over-parameterized deep models, as most of the gradient elements are zero.
- **PyTorch (Cockpit):** Our implementation uses a suggested workaround, computes bin indices and scatters the counts into their associated bins with `torch.Tensor.put_`. This circumvents `atomicAdds`, but has poor memory locality.
- **NumPy:** The single-threaded `numpy.histogram2d` serves as baseline, but does not run on GPUs.

2: Permission granted by the authors of https://github.com/miranov25/.../histogramdd_pytorch.py.

3: See the PyTorch pull request at https://github.com/pytorch/pytorch/pull/44485.

4: See also the discussion at https://discuss.pytorch.org/t/torch-bincount-1000x-slower-on-cuda/42654.

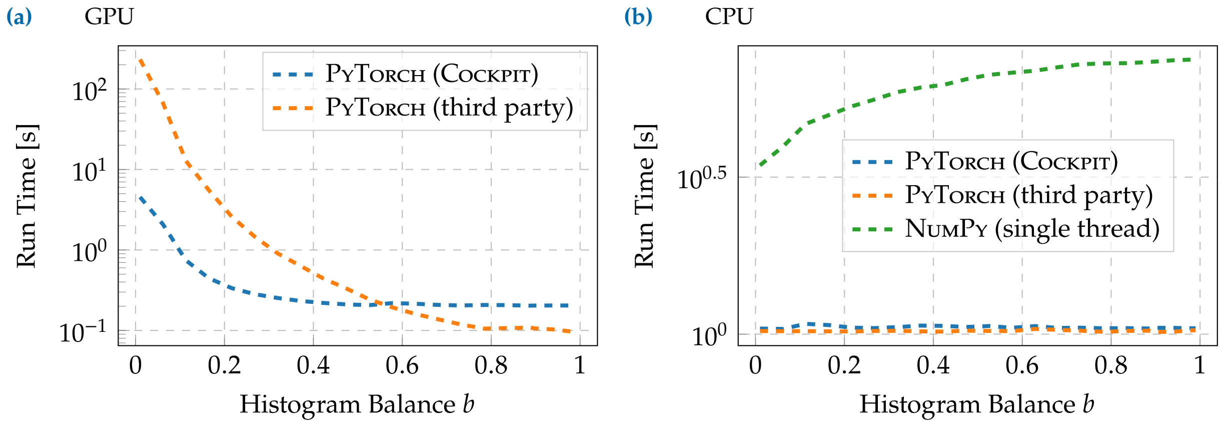


Figure C.11: Performance of two-dimensional histogram GPU implementations depends on the data. (a) Run time for two different GPU implementations with histograms of different imbalance. Cockpit's implementation outperforms the third party solution by more than one order of magnitude in the deep learning regime ($b \ll 1$). (b) On CPU, performance is robust to histogram balance. The run time difference between NumPy and PyTorch is due to multi-threading. Data has the same size as DeepOBS's CIFAR-10 3c3d problem ($D = 895,210, B = 128$). Curves represent averages over 10 independent runs. Error bars are omitted to improve legibility.

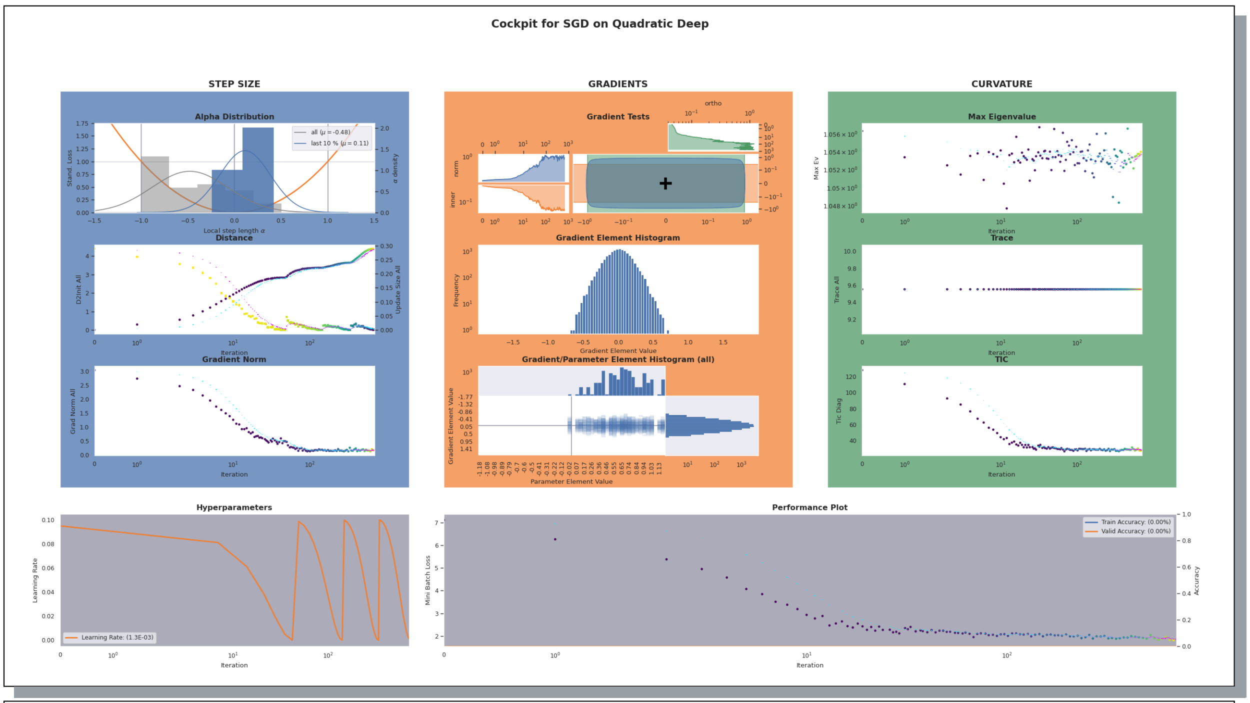
To demonstrate that the performance strongly dependence on the data, we generate data from a uniform distribution over $[0, b] \times [0, b]$, where $b \in (0, 1)$ parametrizes the histogram's balance. We then compute a two-dimensional histograms on $[0, 1] \times [0, 1]$. Figure C.11a clearly shows an increase in run time for both GPU implementations if the histogram is more imbalanced.

Note that even though our custom implementation outperforms the third party implementation by more than one order of magnitude in the deep neural network regime ($b \ll 1$), it is still considerably slower than computing the one-dimensional histogram (see Figure C.6 (c)), and even slower on GPU than on CPU (Figure C.11b). As expected, the CPU implementations do not significantly depend on the histogram's balance (Figure C.11b). The performance difference between PyTorch and NumPy is likely due to multi-threading versus single-threading.

A carefully engineered GPU implementation for efficient histogram computation in the deep learning setting is currently not available. However, we think such an implementation is possible and it would reduce the computational overhead for the two-dimensional histogram roughly to that of the one-dimensional histogram and could be added in future releases of Cockpit.

C.6 Cockpit's View of Convex Stochastic Problems

To contrast the showcase of Cockpit on a deep learning problem in Figure 7.2, we show the Cockpit view of two convex problems. Figure C.12 shows Cockpit's view of a noisy quadratic (*top plot*, P1 in Appendix A.1) and logistic regression on MNIST (*bottom plot*, P9 in Appendix A.1). Comparing Figures 7.2 and C.12 reveals a significant difference between the behavior of the instruments when comparing convex problems with deep learning tasks, highlighting the unique properties of deep learning optimization.



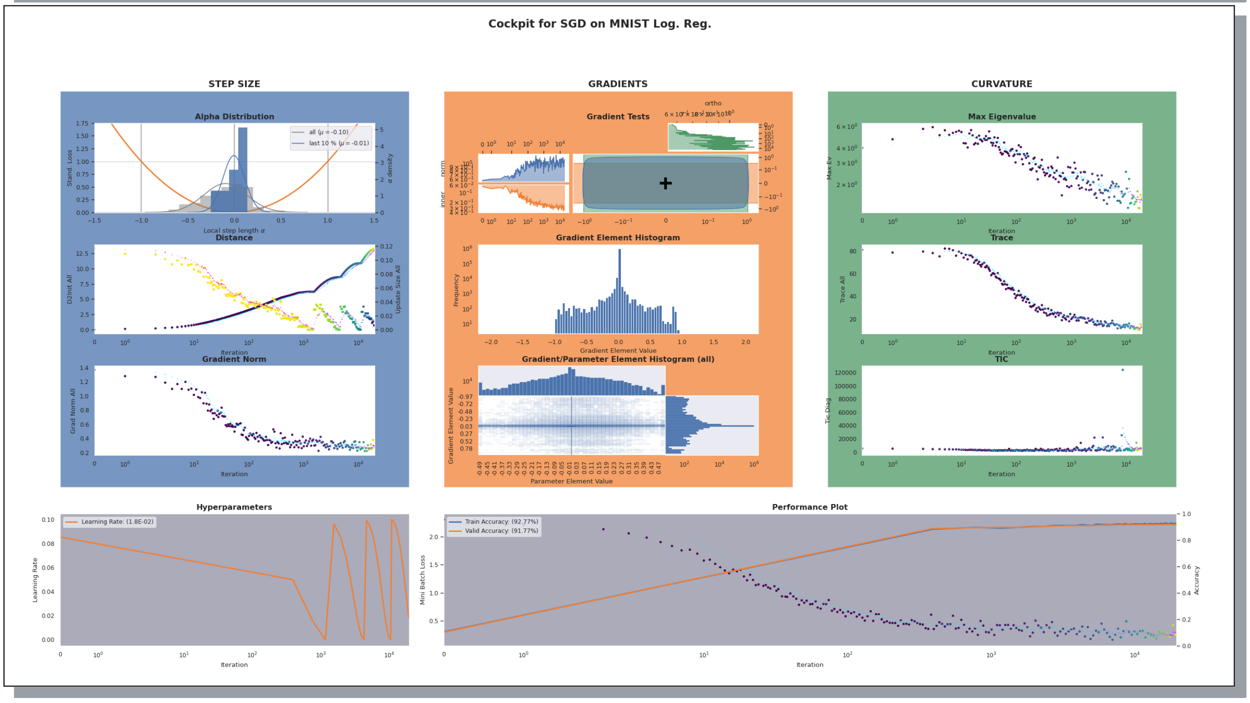


Figure C.12: Screenshot of Cockpit's full view for convex DeepOBS problems. Top Cockpit shows training on a noisy quadratic loss function. Bottom shows training on logistic regression on MNIST. It is evident, that there is a fundamental difference in the optimization process, compared to training deep networks, *i.e.* Figure 7.2. This is, for example, visible when comparing the gradient norms, which converge to zero for convex problems but not for deep learning.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". 2015. URL: http://tensorflow.org/ (cited on pages 2, 64, 78, 79, 96, 108, 130).

[2] Alessandro Achille, Matteo Rovere, and Stefano Soatto. "Critical Learning Periods in Deep Neural Networks". 2017. arXiv: 1711.08856 (cited on page 129).

[3] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. "Fathom: Reference workloads for modern deep learning methods". *IEEE International Symposium on Workload Characterization, IISWC*. 2016 (cited on pages 80, 81).

[4] Naman Agarwal, Rohan Anil, Elad Hazan, Tomer Koren, and Cyril Zhang. "Disentangling Adaptive Gradient Methods from Learning Rates". 2020. arXiv: 2002.11803 (cited on page 102).

[5] Naman Agarwal, Rohan Anil, Elad Hazan, Tomer Koren, and Cyril Zhang. "Revisiting the Generalization of Adaptive Gradient Methods". https://openreview.net/forum?id=BJl6t64tvr. 2020 (cited on pages 45, 46).

[6] Ayush M. Agrawal, Atharva Tendle, Harshvardhan Sikka, Sahib Singh, and Amr Kayid. "Investigating Learning in Deep Neural Networks using Layer-Wise Weight Change". 2020. arXiv: 2011.06735 (cited on page 112).

[7] Laurence Aitchison. "Bayesian filtering unifies adaptive and non-adaptive neural network optimization methods". *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on page 40).

[8] Diogo Almeida, Clemens Winter, Jie Tang, and Wojciech Zaremba. "A Generalizable Approach to Learning Optimizers". 2021. arXiv: 2106.00958 (cited on page 40).

[9] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. "Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin". *33rd International Conference on Machine Learning, ICML*. 2016 (cited on pages 1, 43, 44, 69).

[10] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. “Second Order Optimization Made Practical”. 2020. arXiv: 2002.09018 (cited on pages 40, 53).

[11] Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer. “Memory Efficient Adaptive Optimization”. *Advances in Neural Information Processing Systems 32, NeurIPS*. 2019 (cited on page 40).

[12] Imen Ayadi and Gabriel Turinici. “Stochastic Runge-Kutta methods and adaptive SGD-G2 stochastic gradient descent”. 2020. arXiv: 2002.09304 (cited on page 40).

[13] Jimmy L. Ba, Jamie R. Kiros, and Geoffrey E. Hinton. “Layer Normalization”. 2016. arXiv: 1607.06450 (cited on page 67).

[14] Kiwook Bae, Heechang Ryu, and Hayong Shin. “Does Adam optimizer keep close to the optimal point?” 2019. arXiv: 1911.00289 (cited on page 40).

[15] Achraf Bahamou and Donald Goldfarb. “A Dynamic Sampling Adaptive-SGD Method for Machine Learning”. 2019. arXiv: 1912.13357 (cited on page 163).

[16] Aaron Bahde. “Towards Meaningful Deep Learning Optimizer Benchmarks”. Master’s thesis. University of Tübingen, 2019 (cited on pages 79, 137).

[17] Dariush Bahrami and Sadegh Pouriyan Zadeh. “Gravity Optimizer: a Kinematic Approach on Optimization in Deep Learning”. 2021. arXiv: 2101.09192 (cited on page 40).

[18] Jiyang Bai and Jiawei Zhang. “BGADAM: Boosting based Genetic-Evolutionary ADAM for Convolutional Neural Network Optimization”. 2019. arXiv: 1908.08015 (cited on page 40).

[19] Baidu Research. “DeepBench”. https://github.com/baidu-research/DeepBench. GitHub repository. 2016 (cited on pages 80, 81).

[20] Lukas Balles and Philipp Hennig. “Dissecting Adam: The Sign, Magnitude and Variance of Stochastic Gradients”. *35th International Conference on Machine Learning, ICML*. 2018 (cited on page 40).

[21] Lukas Balles, Javier Romero, and Philipp Hennig. “Coupling Adaptive Batch Sizes with Learning Rates”. *Uncertainty in Artificial Intelligence - Proceedings of the 33rd Conference, UAI*. 2017 (cited on pages 56, 110, 153, 154, 157).

[22] Evelyn M. L. Beale. “On an iterative method for finding a local minimum of a function of more than one variable”. *Statistical Techniques Research Group, Section of Mathematical Statistics, Department of Mathematics, Princeton University*, 1958 (cited on page 85).

[23] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. “Reconciling modern machine-learning practice and the classical bias–variance trade-off”. *Proceedings of the National Academy of Sciences* (2019) (cited on page 20).

[24] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. “Neural Optimizer Search with Reinforcement Learning”. *34th International Conference on Machine Learning, ICML*. 2017 (cited on pages 57, 77, 78, 80).

[25] Yoshua Bengio. “Practical recommendations for gradient-based training of deep architectures”. *Neural Networks: Tricks of the Trade (2nd ed.)* (2012) (cited on pages 54, 114).

[26] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. “Algorithms for Hyper-Parameter Optimization”. *Advances in Neural Information Processing Systems 24, NIPS*. 2011 (cited on pages 55, 56).

[27] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization”. *Journal of Machine Learning Research, JMLR* (2012) (cited on pages 55, 99).

[28] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. "SIGNSGD: Compressed Optimisation for Non-Convex Problems". *35th International Conference on Machine Learning, ICML*. 2018 (cited on page 40).

[29] Leonard Berrada, Andrew Zisserman, and M. Pawan Kumar. "Training Neural Networks for and by Interpolation". *37th International Conference on Machine Learning, ICML*. 2020 (cited on page 40).

[30] Lukas Biewald. "Experiment Tracking with Weights and Biases". Software available from wandb.com. 2020. URL: `https://www.wandb.com/` (cited on page 108).

[31] Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurélie Névéol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. "Findings of the 2016 Conference on Machine Translation". *Proceedings of the First Conference on Machine Translation*. 2016 (cited on page 9).

[32] Raghu Bollapragada, Richard Byrd, and Jorge Nocedal. "Adaptive Sampling Strategies for Stochastic Optimization". *SIAM Journal on Optimization* (2017) (cited on pages 110–112, 153, 154, 159–162).

[33] Oleksandr Borysenko and Maksym Byshkin. "CoolMomentum: A Method for Stochastic Optimization by Langevin Dynamics with Simulated Annealing". 2020. arXiv: `2005.14605` (cited on page 40).

[34] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. "A training algorithm for optimal margin classifiers". *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*. 1992 (cited on page 14).

[35] Aleksandar Botev, Hippolyt Ritter, and David Barber. "Practical Gauss-Newton Optimisation for Deep Learning". *34th International Conference on Machine Learning, ICML*. 2017 (cited on pages 40, 53, 77, 78).

[36] Léon Bottou. "Stochastic gradient descent tricks". *Neural networks: Tricks of the trade*. *Springer*, 2012 (cited on pages 54, 57).

[37] Léon Bottou and Olivier Bousquet. "The Tradeoffs of Large Scale Learning". *Advances in Neural Information Processing Systems 20, NIPS*. 2007 (cited on page 37).

[38] Stephen Boyd and Lieven Vandenberghe. "Convex Optimization". *Cambridge University Press*, 2004 (cited on page 28).

[39] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. "JAX: composable transformations of Python+-NumPy programs". 2018. URL: `http://github.com/google/jax` (cited on pages 2, 110, 130).

[40] Franklin H. Branin. "Widely convergent method for finding multiple solutions of simultaneous nonlinear equations". *IBM Journal of Research and Development* (1972) (cited on page 85).

[41] Leo Breiman, Jerome H. Friedman, Richard. A. Olshen, and Charles. J. Stone. "Classification and Regression Trees". *Wadsworth International Group, Belmont, CA*, 1984 (cited on page 14).

[42] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. "Language Models are Few-Shot Learners". *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on pages 45, 46, 69).

[43] Richard H. Byrd, Gillian M. Chin, Jorge Nocedal, and Yuchen Wu. "Sample Size Selection in Optimization Methods for Machine Learning". *Math. Program.* (2012) (cited on pages 56, 110–112, 153, 154, 159, 160).

[44] Camille Castera, Jérôme Bolte, Cédric Févotte, and Edouard Pauwels. "Second-order step-size tuning of SGD for non-convex optimization". 2021. arXiv: `2103.03570` (cited on page 40).

[45] Augustin-Louis Cauchy. "Méthode générale pour la résolution des systemes d'équations simultanées". *Comp. Rend. Sci. Paris* (1847) (cited on page 41).

[46] Younghwan Chae, Daniel N. Wilke, and Dominic Kafka. "GOALS: Gradient-Only Approximations for Line Searches Towards Robust and Consistent Training of Deep Neural Networks". 2021. arXiv: `2105.10915` (cited on page 40).

[47] Kushal Chakrabarti and Nikhil Chopra. "Generalized AdaGrad (G-AdaGrad) and Adam: A State-Space Perspective". 2021. arXiv: `2106.00092` (cited on page 40).

[48] Satrajit Chatterjee. "Coherent Gradients: An Approach to Understanding Generalization in Gradient Descent-based Optimization". *8th International Conference on Learning Representations, ICLR*. 2020 (cited on page 112).

[49] Satrajit Chatterjee and Piotr Zielinski. "Making Coherence Out of Nothing At All: Measuring the Evolution of Gradient Alignment". 2020. arXiv: `2008.01217` (cited on page 112).

[50] Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. "Entropy-SGD: Biasing gradient descent into wide valleys". *5th International Conference on Learning Representations, ICLR*. 2017 (cited on pages 25, 85, 86, 135).

[51] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. "AdaComp: Adaptive Residual Gradient Compression for Data-Parallel Distributed Training". *32nd AAAI Conference on Artificial Intelligence, AAAI*. 2018 (cited on page 40).

[52] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks". *International Solid-Sate Circuits Conference, ISSCC* (2016) (cited on page 77).

[53] Jinghui Chen, Dongruo Zhou, Yiqi Tang, Ziyan Yang, Yuan Cao, and Quanquan Gu. "Closing the Generalization Gap of Adaptive Gradient Methods in Training Deep Neural Networks". *29th International Joint Conference on Artificial Intelligence, IJCAI*. 2020 (cited on page 40).

[54] Ricky T. Q. Chen, Dami Choi, Lukas Balles, David Duvenaud, and Philipp Hennig. "Self-Tuning Stochastic Optimization with Curvature-Aware Gradient Filtering". 2020. arXiv: `2011.04803` (cited on page 40).

[55] Tianyi Chen, Ziye Guo, Yuejiao Sun, and Wotao Yin. "CADA: Communication-Adaptive Distributed Adam". *24th International Conference on Artificial Intelligence and Statistics, AISTATS*. 2021 (cited on page 40).

[56] Xiangyi Chen, Sijia Liu, Ruoyu Sun, and Mingyi Hong. “On the Convergence of A Class of Adam-Type Algorithms for Non-Convex Optimization”. *7th International Conference on Learning Representations, ICLR*. 2019 (cited on page 40).

[57] Yushu Chen, Hao Jing, Wenlai Zhao, Zhiqiang Liu, Ouyi Li, Liang Qiao, Wei Xue, Haohuan Fu, and Guangwen Yang. “An Adaptive Remote Stochastic Gradient Method for Training Neural Networks”. 2019. arXiv: 1905.01422 (cited on page 40).

[58] Ziyi Chen and Yi Zhou. “Momentum with Variance Reduction for Nonconvex Composition Optimization”. 2020. arXiv: 2005.07755 (cited on page 40).

[59] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. “Learning Phrase Representations using RNN Encoder-Decoderfor Statistical Machine Translation”. *Conference on Empirical Methods in Natural Language Processing, EMNLP*. 2014 (cited on pages 67, 73).

[60] Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl. “On Empirical Comparisons of Optimizers for Deep Learning”. 2019. arXiv: 1910.05446 (cited on pages 40, 45, 46, 80, 92, 94, 100, 105).

[61] Yunjey Choi, Youngjung Uh, Jaejun Yoo, and Jung-Woo Ha. “StarGAN v2: Diverse Image Synthesis for Multiple Domains”. *The IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2020 (cited on pages 1, 16, 24).

[62] Djork-Arne Clevert, Thomas Unterthiner, and Sepp Hochreiter. “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)”. *4th International Conference on Learning Representations, ICLR*. 2016 (cited on page 69).

[63] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. “DAWNBench: An End-to-End Deep Learning Benchmark and Competition”. *NIPS ML Systems Workshop* (2017) (cited on page 80).

[64] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. *Machine learning* (1995) (cited on page 14).

[65] Brett Daley and Christopher Amato. “Expectigrad: Fast Stochastic Optimization with Robust Convergence Properties”. 2020. arXiv: 2010.01356 (cited on page 40).

[66] Felix Dangel, Frederik Kunstner, and Philipp Hennig. “BackPACK: Packing more into Backprop”. *8th International Conference on Learning Representations, ICLR*. 2020 (cited on pages 108, 110, 113, 119, 120, 130).

[67] Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. “Automated Inference with Adaptive Batches”. *20th International Conference on Artificial Intelligence and Statistics, AISTATS*. 2017 (cited on page 56).

[68] Aaron Defazio and Samy Jelassi. “Adaptivity without Compromise: A Momentumized, Adaptive, Dual Averaged Gradient Method for Stochastic Optimization”. 2021. arXiv: 2101.11075 (cited on page 40).

[69] Giorgia Dellaferrera, Stanislaw Wozniak, Giacomo Indiveri, Angeliki Pantazi, and Evangelos Eleftheriou. “Learning in Deep Neural Networks Using a Biologically Inspired Optimizer”. 2021. arXiv: 2104.11604 (cited on page 40).

[70] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. “ImageNet: A Large-Scale Hierarchical Image Database”. *The IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2009 (cited on pages 1, 85, 86, 104, 114, 166).

[71] Aditya Devarakonda, Maxim Naumov, and Michael Garland. "AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks". 2017. arXiv: 1712.02029 (cited on page 40).

[72] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. *Association for Computational Linguistics*, 2019 (cited on pages 45, 46, 69).

[73] Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, Diogo Moitinho de Almeida, Brian McFee, Hendrik Weideman, Gábor Takács, Peter de Rivaz, Jon Crall, Gregory Sanders, Kashif Rasul, Cong Liu, Geoffrey French, and Jonas Degrave. "Lasagne: First release." 2015. URL: http://dx.doi.org/10.5281/zenodo.27878 (cited on page 2).

[74] Jianbang Ding, Xuancheng Ren, Ruixuan Luo, and Xu Sun. "An Adaptive and Momental Bound Method for Stochastic Learning". 2019. arXiv: 1910.12249 (cited on page 40).

[75] Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. "Sharp Minima Can Generalize For Deep Nets". *34th International Conference on Machine Learning, ICML*. 2017 (cited on pages 24, 25, 113).

[76] Timothy Dozat. "Incorporating Nesterov Momentum into Adam". *4th International Conference on Learning Representations, ICLR*. 2016 (cited on pages 40, 46, 80, 98).

[77] Shiv Ram Dubey, Soumendu Chakraborty, Swalpa Kumar Roy, Snehasis Mukherjee, Satish Kumar Singh, and Bidyut Baran Chaudhuri. "diffGrad: An Optimization Method for Convolutional Neural Networks". *IEEE Transactions on Neural Networks and Learning Systems* (2020) (cited on page 40).

[78] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". *Journal of Machine Learning Research, JMLR* (2011) (cited on pages 40, 43, 45, 80, 98).

[79] Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. "Incorporating Second-Order Functional Knowledge for Better Option Pricing ". *Advances in Neural Information Processing Systems 13, NIPS*. 2000 (cited on page 69).

[80] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. "Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning". *Neural Networks* (2018) (cited on pages 69, 70).

[81] Yanai Eliyahu. "ADAS Optimzier". https://github.com/YanaiEliyahu/AdasOptimizer. GitHub repository. 2020 (cited on page 40).

[82] Fartash Faghri, David Duvenaud, David J. Fleet, and Jimmy Ba. "A Study of Gradient Variance in Deep Learning". 2020. arXiv: 2007.04532 (cited on page 110).

[83] Stefan Falkner, Aaron Klein, and Frank Hutter. "BOHB: Robust and Efficient Hyperparameter Optimization at Scale". *35th International Conference on Machine Learning, ICML*. 2018 (cited on pages 55, 56).

[84] William Fedus, Barret Zoph, and Noam Shazeer. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". 2021. arXiv: 2101.03961 (cited on page 60).

[85] Abraham J. Fetterman, Christina H. Kim, and Joshua Albrecht. “SoftAdam: Unifying SGD and Adam for better stochastic gradient descent”. https://openreview.net/forum?id=Skgfr1rYDH. 2019 (cited on page 40).

[86] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. “Sharpness-aware Minimization for Efficiently Improving Generalization”. *9th International Conference on Learning Representations, ICLR*. 2021 (cited on pages 25, 40).

[87] Stanislav Fort, Gintare Karolina Dziugaite, Mansheej Paul, Sepideh Kharaghani, Daniel M. Roy, and Surya Ganguli. “Deep learning versus kernel learning: an empirical study of loss landscape geometry and the time evolution of the Neural Tangent Kernel”. *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on page 129).

[88] Jonathan Frankle, David J. Schwab, and Ari S. Morcos. “The Early Phase of Neural Network Training”. *8th International Conference on Learning Representations, ICLR*. 2020 (cited on pages 112, 113, 118, 129).

[89] Kunihiko Fukushima. “Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements”. *IEEE Transactions on Systems Science and Cybernetics* (1969) (cited on pages 2, 3, 67, 69).

[90] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. *Biological Cybernetics* (1980) (cited on pages 2, 65, 72).

[91] Kai-Xin Gao, Xiao-Lei Liu, Zheng-Hai Huang, Min Wang, Shuangling Wang, Zidong Wang, Dachuan Xu, and Fan Yu. “Eigenvalue-corrected Natural Gradient Based on a New Approximation”. 2020. arXiv: 2011.13609 (cited on page 40).

[92] Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P. Vetrov, and Andrew G. Wilson. “Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs”. *Advances in Neural Information Processing Systems 31, NeurIPS*. 2018 (cited on page 97).

[93] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. “ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness.” *7th International Conference on Learning Representations, ICLR*. 2019 (cited on page 61).

[94] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. “Fast Approximate Natural Gradient Descent in a Kronecker Factored Eigenbasis”. *Advances in Neural Information Processing Systems 31, NeurIPS*. 2018 (cited on page 40).

[95] Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. “An Investigation into Neural Net Optimization via Hessian Eigenvalue Density”. *36th International Conference on Machine Learning, ICML*. 2019 (cited on page 113).

[96] Boris Ginsburg. “On regularization of gradient descent, layer imbalance and flat minima”. 2020. arXiv: 2007.09286 (cited on pages 112, 113, 167).

[97] Boris Ginsburg, Patrice Castonguay, Oleksii Hrinchuk, Oleksii Kuchaiev, Vitaly Lavrukhin, Ryan Leary, Jason Li, Huyen Nguyen, and Jonathan M. Cohen. “Stochastic Gradient Methods with Layer-wise Adaptive Moments for Training of Deep Networks”. 2019. arXiv: 1905.11286 (cited on page 40).

[98] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. *14th International Conference on Artificial Intelligence and Statistics, AISTATS*. 2011 (cited on pages 2, 3, 67, 69).

[99] Gabriel Goh. “Why Momentum Really Works”. *Distill* (2017) (cited on pages 42, 43).

[100] Donald Goldfarb, Yi Ren, and Achraf Bahamou. “Practical Quasi-Newton Methods for Training Deep Neural Networks”. *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on pages 40, 53).

[101] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. “Deep Learning”. *MIT Press*, 2016 (cited on pages 14, 21, 24, 25, 42, 43, 45, 57, 67, 69, 70, 77, 78, 100).

[102] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. “Generative Adversarial Nets”. *Advances in Neural Information Processing Systems 27, NIPS*. 2014 (cited on pages 1, 16, 72, 102).

[103] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. “Maxout Networks”. *30th International Conference on Machine Learning, ICML*. 2013 (cited on pages 69, 70).

[104] Marco Gori, Gabriele Monfardini, and Franco Scarselli. “A new model for learning in graph domains”. *IEEE International Joint Conference on Neural Networks*. 2005 (cited on page 72).

[105] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. 2017. arXiv: `1706.02677` (cited on pages 57, 100).

[106] Mikhail Grankin. “RangerLars”. `https://github.com/mgrankin/over9000`. GitHub repository. 2020 (cited on page 40).

[107] Diego Granziol, Xingchen Wan, and Stephen Roberts. “Gadam: Combining Adaptivity with Iterate Averaging Gives Greater Generalisation”. 2020. arXiv: `2003.01247` (cited on page 40).

[108] Alex Graves. “Generating Sequences With Recurrent Neural Networks”. 2013. arXiv: `1308.0850` (cited on pages 44, 45).

[109] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural Turing machines”. 2014. arXiv: `1410.5401` (cited on page 77).

[110] Vineet Gupta, Tomer Koren, and Yoram Singer. “Shampoo: Preconditioned Stochastic Tensor Optimization”. *35th International Conference on Machine Learning, ICML*. 2018 (cited on pages 40, 53).

[111] Guy Gur-Ari, Daniel A. Roberts, and Ethan Dyer. “Gradient Descent Happens in a Tiny Subspace”. 2018. arXiv: `1812.04754` (cited on page 116).

[112] Stephen Hanson and Lorien Pratt. “Comparing Biases for Minimal Network Construction with Back-Propagation”. *Advances in Neural Information Processing Systems 1, NIPS*. 1988 (cited on pages 23, 24).

[113] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. “Array programming with NumPy”. *Nature* (2020) (cited on page 114).

[114] Hiroaki Hayashi, Jayanth Koushik, and Graham Neubig. “Eve: A Gradient Based Optimization Method with Locally and Globally Adaptive Learning Rates”. 2018. arXiv: `1611.01505` (cited on page 40).

[115] Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. “Mask R-CNN”. *IEEE/CVF International Conference on Computer Vision, ICCV*. 2017 (cited on pages 43, 44).

[116] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. *IEEE/CVF International Conference on Computer Vision, ICCV*. 2015 (cited on pages 69, 70).

[117] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition”. *The IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2016 (cited on pages 2, 3, 43, 44, 57, 69, 73, 77, 173).

[118] Donald O. Hebb. “Organization of Behavior”. *Wiley, New York*, 1949 (cited on pages 14, 59).

[119] Dan Hendrycks and Kevin Gimpel. “Gaussian Error Linear Units (GELUs)”. 2016. arXiv: `1606.08415` (cited on page 69).

[120] João F. Henriques, Sébastien Ehrhardt, Samuel Albanie, and Andrea Vedaldi. “Small Steps and Giant Leaps: Minimal Newton Solvers for Deep Learning”. *IEEE/CVF International Conference on Computer Vision, ICCV*. 2019 (cited on page 40).

[121] Byeongho Heo, Sanghyuk Chun, Seong Joon Oh, Dongyoon Han, Sangdoo Yun, Youngjung Uh, and Jung-Woo Ha. “AdamP: Slowing Down the Weight Norm Increase in Momentum-based Optimizers”. *7th International Conference on Learning Representations, ICLR*. 2021 (cited on page 40).

[122] Hewlett Packard Enterprise. “Deep Learning Benchmarking Suite (DLBS)”. Online. GitHub repository. 2017. URL: `https://hewlettpackard.github.io/dlcookbook-dlbs/` (cited on pages 80, 81).

[123] Geoffrey E. Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew W. Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. “Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups”. *IEEE Signal Processing Magazine* (2012) (cited on pages 1, 67).

[124] Sepp Hochreiter. “Untersuchungen zu dynamischen neuronalen Netzen”. *Diploma, Technische Universität München* (1991). In German (cited on page 67).

[125] Sepp Hochreiter and Jürgen Schmidhuber. “Simplifying Neural Nets by Discovering Flat Minima”. *Advances in Neural Information Processing Systems 7, NIPS*. 1994 (cited on pages 24, 25).

[126] Sepp Hochreiter and Jürgen Schmidhuber. “Flat Minima”. *Neural Comput.* (1997) (cited on pages 24, 25, 164).

[127] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. *Neural Comput.* (1997) (cited on pages 67, 73, 113, 136).

[128] Arthur E. Hoerl and Robert W. Kennard. “Ridge Regression: Biased Estimation for Non-orthogonal Problems”. *Technometrics* (1970) (cited on pages 22, 23).

[129] Kurt Hornik. “Approximation capabilities of multilayer feedforward networks”. *Neural Networks* (1991) (cited on page 72).

[130] Mahdi S. Hosseini and Konstantinos N. Plataniotis. “AdaS: Adaptive Scheduling of Stochastic Gradients”. 2020. arXiv: `2006.06587` (cited on page 40).

[131] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. "Searching for MobileNetV3". *IEEE/CVF International Conference on Computer Vision, ICCV*. 2019 (cited on pages 44, 45).

[132] Jeremy Howard and Sebastian Ruder. "Universal Language Model Fine-tuning for Text Classification". *56th Annual Meeting of the Association for Computational Linguistics*. 2018 (cited on page 57).

[133] Yifan Hu, Siqi Zhang, Xin Chen, and Niao He. "Biased Stochastic First-Order Methods for Conditional Stochastic Optimization and Applications in Meta Learning". *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on page 40).

[134] Yuzheng Hu, Licong Lin, and Shange Tang. "Second-order Information in First-order Optimization Methods". 2019. arXiv: `1912.09926` (cited on page 40).

[135] Haiwen Huang, Chang Wang, and Bin Dong. "Nostalgic Adam: Weighting More of the Past Gradients When Designing the Adaptive Learning Rate". *28th International Joint Conference on Artificial Intelligence, IJCAI*. 2019 (cited on page 40).

[136] Xunpeng Huang, Hao Zhou, Runxin Xu, Zhe Wang, and Lei Li. "Adaptive Gradient Methods Can Be Provably Faster than SGD after Finite Epochs". 2020. arXiv: `2006.07037` (cited on page 40).

[137] Peter J. Huber. "Robust Estimation of a Location Parameter". *The Annals of Mathematical Statistics* (1964) (cited on page 72).

[138] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential Model-Based Optimization for General Algorithm Configuration". *Learning and Intelligent Optimization*. 2011 (cited on pages 55, 56).

[139] Yasutoshi Ida, Yasuhiro Fujiwara, and Sotetsu Iwamura. "Adaptive Learning Rate via Covariance Matrix Based Preconditioning for Deep Neural Networks". *26th International Joint Conference on Artificial Intelligence, IJCAI*. 2017 (cited on page 40).

[140] Wendyam Eric Lionel Ilboudo, Taisuke Kobayashi, and Kenji Sugimoto. "TAdam: A Robust Stochastic Gradient Optimizer". 2020. arXiv: `2003.00179` (cited on page 40).

[141] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". *32nd International Conference on Machine Learning, ICML*. 2015 (cited on page 66).

[142] Alexey G. Ivakhnenko and Valentin G. Lapa. "Cybernetic Predicting Devices". *CCM Information Corporation*, 1965 (cited on page 72).

[143] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and G. Wilson Andrew. "Averaging weights leads to wider optima and better generalization". *Uncertainty in Artificial Intelligence - Proceedings of the 34th Conference, UAI*. 2018 (cited on page 97).

[144] Stanislaw Jastrzebski, Devansh Arpit, Oliver Astrand, Giancarlo Kerg, Huan Wang, Caiming Xiong, Richard Socher, Kyunghyun Cho, and Krzysztof Geras. "Catastrophic Fisher Explosion: Early Phase Fisher Matrix Impacts Generalization". 2020. arXiv: `2012.14193` (cited on page 113).

[145] Stanislaw Jastrzebski, Maciej Szymczak, Stansilav Fort, Devansh Arpit, Jacek Tabor, Kyunghyun Cho, and Krzystztof Geras. "The Break-Even Point on the Optimization Trajectories of Deep Neural Networks". *8th International Conference on Learning Representations, ICLR*. 2020 (cited on pages 112, 113, 129).

[146] Stanisław Jastrzębski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. “Three Factors Influencing Minima in SGD”. *Artificial Neural Networks and Machine Learning, ICANN*. 2018 (cited on pages 24, 25).

[147] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. “Caffe: Convolutional Architecture for Fast Feature Embedding”. 2014. arXiv: `1408.5093` (cited on page 2).

[148] Yiding Jiang, Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio. “Fantastic Generalization Measures and Where to Find Them”. 2019. arXiv: `1912.02178` (cited on pages 24, 25).

[149] Zhanhong Jiang, Aditya Balu, Sin Yong Tan, Young M. Lee, Chinmay Hegde, and Soumik Sarkar. “On Higher-order Moments in Adam”. 2019. arXiv: `1910.06878` (cited on page 40).

[150] Yuchen Jin, Tianyi Zhou, Liangyu Zhao, Yibo Zhu, Chuanxiong Guo, Marco Canini, and Arvind Krishnamurthy. “AutoLRS: Automatic Learning-Rate Schedule by Bayesian Optimization on the Fly”. *9th International Conference on Learning Representations, ICLR*. 2021 (cited on pages 40, 56).

[151] Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. “Google’s Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation”. 2017. arXiv: `1611.04558` (cited on page 1).

[152] Tyler B. Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. “AdaScale SGD: A User-Friendly Algorithm for Distributed Training”. *37th International Conference on Machine Learning, ICML*. 2020 (cited on page 40).

[153] N. Jouppi, C. Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, R. Bajwa, Sarah Bates, S. Bhatia, N. Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, J. Dean, Ben Gelb, T. Ghaemmaghami, R. Gottipati, William Gulland, R. Hagmann, C. R. Ho, Doug Hogberg, John Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, J. Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle A. Lucke, Alan Lundin, G. MacKean, A. Maggiore, Maire Mahony, K. Miller, R. Nagarajan, Ravi Narayanaswami, Ray Ni, K. Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, A. Phelps, Jonathan Ross, Matt Ross, Amir Salek, E. Samadiani, C. Severn, G. Sizikov, Matthew Snelham, J. Souter, D. Steinberg, Andy Swing, Mercedes Tan, G. Thorson, Bo Tian, H. Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and D. Yoon. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture, ISCA* (2017) (cited on page 77).

[154] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. “Highly accurate protein structure prediction with AlphaFold”. *Nature* (2021) (cited on pages 1, 9, 59).

[155] Dominic Kafka and Daniel Wilke. “Gradient-only line searches: An Alternative to Probabilistic Line Searches”. 2019. arXiv: 1903.09383 (cited on page 40).

[156] Andrej Karpathy. “A peek at trends in machine learning”. https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106. Accessed 24. Sep. 2018. 2017 (cited on pages 77, 78).

[157] Andrej Karpathy. “Software 2.0”. https://karpathy.medium.com/software-2-0-a64152b37c35. Accessed January 11 2022. 2017 (cited on page 126).

[158] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. “Progressive Growing of GANs for Improved Quality, Stability, and Variation”. *6th International Conference on Learning Representations, ICLR*. 2018 (cited on pages 1, 16, 45, 46).

[159] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. “Analyzing and Improving the Image Quality of StyleGAN”. *The IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2020 (cited on pages 16, 45, 46).

[160] Henry J. Kelley. “Gradient Theory of Optimal Flight Paths”. *ARS Journal* (1960) (cited on pages 2, 62).

[161] Chad Kelterborn, Marcin Mazur, and Bogdan V. Petrenko. “Gravilon: Applications of a New Gradient Descent Method to Machine Learning”. 2020. arXiv: 2008.11370 (cited on page 40).

[162] Nitish S. Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping T. P. Tang. “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”. *5th International Conference on Learning Representations, ICLR*. 2017 (cited on pages 24, 25).

[163] Nitish Shirish Keskar and Richard Socher. “Improving Generalization Performance by Switching from Adam to SGD”. 2017. arXiv: 1712.07628 (cited on pages 40, 45–47, 113).

[164] Mohammad E. Khan, Didrik Nielsen, Voot Tangkaratt, Wu Lin, Yarin Gal, and Akash Srivastava. “Fast and Scalable Bayesian Deep Learning by Weight-Perturbation in Adam”. *35th International Conference on Machine Learning, ICML*. 2018 (cited on page 40).

[165] Jack Kiefer and Jacob Wolfowitz. “Stochastic estimation of the maximum of a regression function”. *The Annals of Mathematical Statistics* (1952) (cited on page 41).

[166] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. *3rd International Conference on Learning Representations, ICLR*. 2015 (cited on pages 2, 40, 45, 46, 77, 80, 98, 135).

[167] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. “Self-Normalizing Neural Networks”. *Advances in Neural Information Processing Systems 30, NIPS*. 2017 (cited on page 69).

[168] Alex Krizhevsky. “Convolutional Deep Belief Networks on CIFAR-10”. Technical Report. 2010 (cited on page 69).

[169] Alex Krizhevsky and Geoffrey Hinton. “Learning multiple layers of features from tiny images”. Technical Report. 2009 (cited on pages 69, 78, 85, 86, 114).

[170] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. *Advances in Neural Information Processing Systems 25, NIPS*. 2012 (cited on page 59).

[171] Jungmin Kwon, Jeongseop Kim, Hyunseo Park, and In K. Choi. “ASAM: Adaptive Sharpness-Aware Minimization for Scale-Invariant Learning of Deep Neural Networks”. *38th International Conference on Machine Learning, ICML*. 2021 (cited on page 40).

[172] Tomer Lancewicki and Selcuk Kopru. “Automatic and Simultaneous Adjustment of Learning Rate and Momentum for Stochastic Gradient Descent”. 2019. arXiv: 1908.07607 (cited on page 56).

[173] Nicola Landro, Ignazio Gallo, and Riccardo La Grassa. “Mixing ADAM and SGD: a Combined Optimization Method”. 2020. arXiv: 2011.08042 (cited on page 40).

[174] Yann LeCun. “Une procedure d’apprentissage pour reseau a seuil asymmetrique”. *Proceedings of Cognitiva 85, Paris, France*. 1985 (cited on page 62).

[175] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. *Nature* (2015) (cited on page 61).

[176] Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence. D. Jackel. “Backpropagation Applied to Handwritten Zip Code Recognition”. *Neural Computation* (1989) (cited on pages 2, 65, 72).

[177] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-Based Learning Applied to Document Recognition”. *Proceedings of the IEEE* (1998) (cited on pages 9, 10, 67, 78, 85, 86, 117).

[178] Alexander LeNail. “NN-SVG: Publication-Ready Neural Network Architecture Schematics”. *Journal of Open Source Software* (2019) (cited on pages 72, 74).

[179] Kfir Yehuda Levy, Alp Yurtsever, and Volkan Cevher. “Online Adaptive Methods, Universality and Acceleration”. *Advances in Neural Information Processing Systems 31, NeurIPS*. 2018 (cited on page 40).

[180] Wenjie Li, Zhaoyang Zhang, Xinjiang Wang, and Ping Luo. “AdaX: Adaptive Gradient Descent with Exponential Long Term Memory”. 2020. arXiv: 2004.09740 (cited on page 40).

[181] Zhize Li, Hongyan Bao, Xiangliang Zhang, and Peter Richtárik. “PAGE: A Simple and Optimal Probabilistic Gradient Estimator for Nonconvex Optimization”. 2020. arXiv: 2008.10898 (cited on page 40).

[182] Junyang Lin, An Yang, Jinze Bai, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Yong Li, Wei Lin, Jingren Zhou, and Hongxia Yang. “M6-10T: A Sharing-Delinking Paradigm for Efficient Multi-Trillion Parameter Pretraining”. 2021. arXiv: 2110.03888 (cited on page 60).

[183] Seppo Linnainmaa. “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. In Finnish. Master’s thesis. University of Helsinki, 1970 (cited on pages 2, 62).

[184] Dong C. Liu and Jorge Nocedal. “On the Limited Memory BFGS Method for Large Scale Optimization”. *Mathematical Programming* (1989) (cited on page 53).

[185] Guan-Horng Liu, Tianrong Chen, and Evangelos A. Theodorou. “Dynamic Game Theoretic Neural Optimizer”. 2021. arXiv: 2105.03788 (cited on page 40).

[186] Hailiang Liu and Xuping Tian. “AEGD: Adaptive Gradient Decent with Energy”. 2020. arXiv: 2010.05109 (cited on page 40).

[187] Jinlong Liu, Guo-Qing Jiang, Yunzhi Bai, Huayan Wang, and Ting Chen. “Understanding Why Neural Networks Generalize Well Through GSNR of Parameters”. *8th International Conference on Learning Representations, ICLR*. 2020 (cited on pages 110, 112, 153, 154, 165, 166).

[188] Liang Liu and Xiaopeng Luo. “A New Accelerated Stochastic Gradient Method with Momentum”. 2020. arXiv: 2006.00423 (cited on page 40).

[189] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. “On the Variance of the Adaptive Learning Rate and Beyond”. *8th International Conference on Learning Representations, ICLR*. 2020 (cited on pages 40, 48, 98).

[190] Mingrui Liu, Wei Zhang, Francesco Orabona, and Tianbao Yang. “Adam$^+$: A Stochastic Method with Adaptive Variance Reduction”. 2020. arXiv: 2011.11985 (cited on page 40).

[191] Rui Liu, Tianyi Wu, and Barzan Mozafari. “Adam with Bandit Sampling for Deep Learning”. *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on page 40).

[192] Yang Liu, Jeremy Bernstein, Markus Meister, and Yisong Yue. “Learning by Turning: Neural Architecture Aware Optimisation”. 2021. eprint: arXiv:2102.07227 (cited on page 40).

[193] Ilya Loshchilov and Frank Hutter. “SGDR: Stochastic Gradient Descent with Warm Restarts”. *5th International Conference on Learning Representations, ICLR*. 2017 (cited on pages 40, 57, 77, 78, 80, 100, 140).

[194] Ilya Loshchilov and Frank Hutter. “Decoupled weight decay regularization”. *7th International Conference on Learning Representations, ICLR*. 2019 (cited on pages 23, 24, 40, 50).

[195] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. “The Expressive Power of Neural Networks: A View from the Width”. *Advances in Neural Information Processing Systems 30, NIPS*. 2017 (cited on page 72).

[196] Liangchen Luo, Yuanhao Xiong, Yan Liu, and Xu Sun. “Adaptive Gradient Methods with Dynamic Bound of Learning Rate”. *7th International Conference on Learning Representations, ICLR*. 2019 (cited on pages 40, 47, 98).

[197] Jerry Ma and Denis Yarats. “Quasi-hyperbolic momentum and Adam for deep learning”. *7th International Conference on Learning Representations, ICLR*. 2019 (cited on page 40).

[198] Jerry Ma and Denis Yarats. “On the Adequacy of Untuned Warmup for Adaptive Optimization”. *35th AAAI Conference on Artificial Intelligence, AAAI*. 2021 (cited on page 48).

[199] Maren Mahsereci, Lukas Balles, Christoph Lassner, and Philipp Hennig. “Early Stopping without a Validation Set”. 2017. arXiv: 1703.09580 (cited on pages 25, 110, 153, 154, 157).

[200] Maren Mahsereci and Philipp Hennig. “Probabilistic Line Searches for Stochastic Optimization”. *Journal of Machine Learning Research, JMLR*. 2017 (cited on pages 40, 56, 77, 78, 157).

[201] Itzik Malkiel and Lior Wolf. “MTAdam: Automatic Balancing of Multiple Training Loss Terms”. 2020. arXiv: 2006.14683 (cited on page 40).

[202] James Martens. “Deep learning via Hessian-free optimization.” *27th International Conference on Machine Learning, ICML*. 2010 (cited on page 52).

[203] James Martens and Roger Grosse. “Optimizing Neural Networks with Kronecker-Factored Approximate Curvature”. *32nd International Conference on Machine Learning, ICML*. 2015 (cited on pages 40, 53, 77, 78, 80).

[204] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. *The bulletin of mathematical biophysics* (1943) (cited on pages 14, 59).

[205] Luke Metz, Niru Maheswaranathan, Ruoxi Sun, C. Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. “Using a thousand optimization tasks to learn hyperparameter search strategies”. 2020. arXiv: 2002.11887 (cited on page 94).

[206] Microsoft Machine Learning. “Comparing Deep Learning Frameworks: A Rosetta Stone Approach”. Accessed 24. Sep. 2018. 2018. URL: https://blogs.technet.microsoft.com/machinelearning/2018/03/14/comparing-deep-learning-frameworks-a-rosetta-stone-approach/ (cited on pages 80, 81).

[207] Diganta Misra. “Mish: A Self Regularized Non-Monotonic Activation Function”. 2019. arXiv: 1908.08681 (cited on pages 69, 70).

[208] MLPerf. “MLPerf.org”. Accessed 24. Sep. 2018. 2018. URL: https://mlperf.org/ (cited on pages 80, 81).

[209] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. “Human-level control through deep reinforcement learning”. *Nature* (2015) (cited on pages 1, 11).

[210] Gordon E. Moore. “Cramming more components onto integrated circuits”. *Electronics* (1965) (cited on pages 1, 2).

[211] Mahesh Chandra Mukkamala and Matthias Hein. “Variants of RMSProp and Adagrad with Logarithmic Regret Bounds”. *34th International Conference on Machine Learning, ICML*. 2017 (cited on page 40).

[212] Rotem Mulayoff and Tomer Michaeli. “Unique Properties of Flat Minima in Deep Networks”. *37th International Conference on Machine Learning, ICML*. 2020 (cited on pages 24, 25, 112, 113, 167).

[213] Kevin P. Murphy. “Machine Learning A Probabilistic Perspective”. *MIT press*, 2012 (cited on page 14).

[214] Maximus Mutschler and Andreas Zell. “Parabolic Approximation Line Search for DNNs”. *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on pages 40, 56).

[215] Zachary Nado, Justin M. Gilmer, Christopher J. Shallue, Rohan Anil, and George E. Dahl. “A Large Batch Optimizer Reality Check: Traditional, Generic Optimizers Suffice Across Batch Sizes”. 2021. arXiv: 2102.06356 (cited on page 50).

[216] Vaishnavh Nagarajan and J. Zico Kolter. “Generalization in Deep Networks: The Role of Distance from Initialization”. 2019. arXiv: 1901.01672 (cited on page 112).

[217] Vinod Nair and Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. *27th International Conference on Machine Learning, ICML*. 2010 (cited on pages 2, 3, 67, 69).

[218] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. “Deep Double Descent: Where Bigger Models and More Data Hurt”. *8th International Conference on Learning Representations, ICLR*. 2020 (cited on page 20).

[219] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. “Deep Learning Recommendation Model for Personalization and Recommendation Systems”. 2019. arXiv: 1906.00091 (cited on pages 43, 45).

[220] Parvin Nazari, Davoud Ataee Tarzanagh, and George Michailidis. "DADAM: A Consensus-based Distributed Adaptive Gradient Method for Online Optimization". 2019. arXiv: 1901.09109 (cited on page 40).

[221] Yurii Nesterov. "A method for solving the convex programming problem with convergence rate $O(1/k^2)$". *Soviet Mathematics Doklady* (1983) (cited on pages 40, 42, 44, 77, 98).

[222] Yurii Nesterov. "Lectures on Convex Optimization". *Springer Publishing Company, Incorporated*, 2018 (cited on page 28).

[223] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. "Reading digits in natural images with unsupervised feature learning". *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*. 2011 (cited on pages 85, 86).

[224] Jorge Nocedal and Stephen J. Wright. "Numerical Optimization". Springer S. *Springer-Verlag New York*, 2006 (cited on pages 28, 52).

[225] Francesco Orabona and Dávid Pál. "Scale-Free Algorithms for Online Linear Optimization". *Algorithmic Learning Theory - 26th International Conference, ALT*. 2015 (cited on page 40).

[226] Antonio Orvieto, Jonas Köhler, and Aurélien Lucchi. "The Role of Memory in Stochastic Optimization". *Uncertainty in Artificial Intelligence - Proceedings of the 35th Conference, UAI*. 2019 (cited on page 40).

[227] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware". 2015 (cited on page 77).

[228] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". *Advances in Neural Information Processing Systems 32, NeurIPS*. 2019 (cited on pages 2, 64, 78, 107, 108, 130, 151).

[229] Barak A. Pearlmutter. "Fast Exact Multiplication by the Hessian". *Neural Computation* (1994) (cited on pages 52, 113, 119, 163, 164).

[230] Boris T. Polyak. "Some methods of speeding up the convergence of iteration methods". *USSR Computational Mathematics and Mathematical Physics* (1964) (cited on pages 40, 42, 43, 77, 98).

[231] Konpat Preechakul and Boonserm Kijsirikul. "CProp: Adaptive Learning Rate Scaling from Past Gradient Conformity". 2019. arXiv: 1912.11493 (cited on page 40).

[232] Saugata Purkayastha and Sukannya Purkayastha. "A Variant of Gradient Descent Algorithm Based on Gradient Averaging". 2020. arXiv: 2012.02387 (cited on page 40).

[233] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". *4th International Conference on Learning Representations, ICLR*. 2016 (cited on pages 45, 46).

[234] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". *Conference on Empirical Methods in Natural Language Processing, EMNLP*. 2016 (cited on page 80).

[235] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". 2017. arXiv: 1710.05941 (cited on pages 69, 70).

[236] Ali Ramezani-Kebrya, Ashish Khisti, and Ben Liang. “On the Generalization of Stochastic Gradient Descent with Momentum”. 2021. arXiv: 2102.13653 (cited on page 40).

[237] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. “Minerva: Enabling low-power, highly-accurate deep neural network accelerators”. *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA*. 2016 (cited on page 77).

[238] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. “On the Convergence of Adam and Beyond”. *6th International Conference on Learning Representations, ICLR*. 2018 (cited on pages 40, 46, 80, 98).

[239] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. “You Only Look Once: Unified, Real-Time Object Detection”. *The IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2016 (cited on pages 43, 44).

[240] Yi Ren and Donald Goldfarb. “Kronecker-factored Quasi-Newton Methods for Convolutional Neural Networks”. 2021. arXiv: 2102.06737 (cited on page 40).

[241] Rami Al-Rfou et al. “Theano: A Python framework for fast computation of mathematical expressions”. 2016. arXiv: 1605.02688 (cited on page 2).

[242] Herbert Robbins and Sutton Monro. “A Stochastic Approximation Method”. *The Annals of Mathematical Statistics* (1951) (cited on pages 2, 40, 41, 77, 98).

[243] Michal Rolínek and Georg Martius. “L4: Practical loss-based stepsize adaptation for deep learning”. *Advances in Neural Information Processing Systems 31, NeurIPS*. 2018 (cited on pages 40, 56).

[244] Filip de Roos, Carl Jidling, Adrian Wills, Thomas Schön, and Philipp Hennig. “A Probabilistically Motivated Learning Rate Adaptation for Stochastic Optimization”. 2021. arXiv: 2102.10880 (cited on page 40).

[245] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review* (1958) (cited on pages 59, 72).

[246] Howard H. Rosenbrock. “An automatic method for finding the greatest or least value of a function”. *The Computer Journal* (1960) (cited on pages 85, 86).

[247] Swalpa K. Roy, Mercedes E. Paoletti, Juan M. Haut, Shiv R. Dubey, Purbayan Kar, Antonio Plaza, and Bidyut B. Chaudhuri. “AngularGrad: A New Optimization Technique for Angular Convergence of Convolutional Neural Networks”. 2021. arXiv: 2105.10190 (cited on page 40).

[248] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. *Nature* (1986) (cited on pages 2, 62).

[249] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. “Dynamic routing between capsules”. *Advances in Neural Information Processing Systems 30, NIPS*. 2017 (cited on page 77).

[250] Levent Sagun, Leon Bottou, and Yann LeCun. “Eigenvalues of the Hessian in Deep Learning: Singularity and Beyond”. 2016. arXiv: 1611.07476 (cited on page 113).

[251] Levent Sagun, Utku Evci, V. Ugur Guney, Yann Dauphin, and Leon Bottou. “Empirical Analysis of the Hessian of Over-Parametrized Neural Networks”. 2017. arXiv: 1706.04454 (cited on page 113).

[252] Arnold Salas, Samuel Kessler, Stefan Zohren, and Stephen Roberts. “Practical Bayesian Learning of Neural Networks via Adaptive Subgradient Methods”. 2018. arXiv: 1811.03679 (cited on page 40).

[253] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". *The IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2018 (cited on page 69).

[254] Karthik Abinav Sankararaman, Soham De, Zheng Xu, W. Ronny Huang, and Tom Goldstein. "The Impact of Neural Network Overparameterization on Gradient Confusion and Stochastic Gradient Descent". *37th International Conference on Machine Learning, ICML*. 2020 (cited on page 112).

[255] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. "How Does Batch Normalization Help Optimization?" *Advances in Neural Information Processing Systems 31, NeurIPS*. 2018 (cited on page 67).

[256] Pedro Savarese, David McAllester, Sudarshan Babu, and Michael Maire. "Domain-independent Dominance of Adaptive Methods". 2019. arXiv: 1912.01823 (cited on page 40).

[257] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. "The Graph Neural Network Model". *IEEE Transactions on Neural Networks* (2009) (cited on page 72).

[258] Tom Schaul and Yann LeCun. "Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients". *1st International Conference on Learning Representations, ICLR*. 2013 (cited on pages 40, 77, 78).

[259] Tom Schaul, Sixin Zhang, and Yann LeCun. "No more pesky learning rates". *30th International Conference on Machine Learning, ICML*. 2013 (cited on pages 40, 56, 81).

[260] Mark Schmidt. "Convergence rate of stochastic gradient with constant step size". https://www.cs.ubc.ca/~schmidtm/Documents/2014_Notes_ConstantStepSG.pdf. 2014 (cited on pages 113, 163).

[261] Robin M. Schmidt, Frank Schneider, and Philipp Hennig. "Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers". *38th International Conference on Machine Learning, ICML*. 2021 (cited on pages 6, 91).

[262] Frank Schneider, Lukas Balles, and Philipp Hennig. "DeepOBS: A Deep Learning Optimizer Benchmark Suite". *7th International Conference on Learning Representations, ICLR*. 2019 (cited on pages 5, 77, 79, 80, 94, 96, 114, 168).

[263] Frank Schneider, Felix Dangel, and Philipp Hennig. "Cockpit: A Practical Debugging Tool for the Training of Deep Neural Networks". *Advances in Neural Information Processing Systems 34, NeurIPS*. 2021 (cited on pages 6, 107).

[264] Nicol N. Schraudolph. "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent". *Neural Computation* (2002) (cited on pages 52, 53).

[265] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. "Mastering Atari, Go, chess and shogi by planning with a learned model". *Nature* (2020) (cited on page 1).

[266] Fanhua Shang, Kaiwen Zhou, Hongying Liu, James Cheng, Ivor W. Tsang, Lijun Zhang, Dacheng Tao, and Licheng Jiao. "VR-SGD: A Simple Stochastic Variance Reduction Method for Machine Learning". *IEEE Trans. Knowl. Data Eng.* (2020) (cited on page 40).

[267] Noam Shazeer and Mitchell Stern. "Adafactor: Adaptive Learning Rates with Sublinear Memory Cost". *35th International Conference on Machine Learning, ICML*. 2018 (cited on page 40).

[268] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the game of Go with deep neural networks and tree search". *Nature* (2016) (cited on pages 1, 59).

[269] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". *Science* (2018) (cited on page 1).

[270] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". *3rd International Conference on Learning Representations, ICLR*. 2015 (cited on pages 69, 85, 86, 114, 166).

[271] Prabhu T. Sivaprasad, Florian Mai, Thijs Vogels, Martin Jaggi, and Francois Fleuret. "Optimizer Benchmarking Needs to Account for Hyperparameter Tuning". *37th International Conference on Machine Learning, ICML*. 2020 (cited on pages 45, 80, 85, 92, 94, 95, 105).

[272] Leslie N. Smith. "Cyclical Learning Rates for Training Neural Networks". *IEEE Winter Conference on Applications of Computer Vision, WACV*. 2017 (cited on page 57).

[273] Leslie N. Smith. "A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay". 2018. arXiv: `1803.09820` (cited on pages 42, 43, 54).

[274] Leslie N. Smith and Nicholay Topin. "Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates". 2017. arXiv: `1708.07120` (cited on page 57).

[275] Samuel L. Smith, Benoit Dherin, David G. T. Barrett, and Soham De. "On the Origin of Implicit Regularization in Stochastic Gradient Descent". *9th International Conference on Learning Representations, ICLR*. 2021 (cited on page 25).

[276] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. "Practical Bayesian optimization of machine learning algorithms". *Advances in Neural Information Processing Systems 25, NIPS*. 2012 (cited on pages 55, 56).

[277] Jost T. Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. "Striving for simplicity: The all convolutional net". *3rd International Conference on Learning Representations, ICLR (workshop track)*. 2015 (cited on pages 73, 85, 86, 109, 117, 136).

[278] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". *Journal of Machine Learning Research, JMLR* (2014) (cited on page 66).

[279] Huikang Sun, Lize Gu, and Bin Sun. "Adathm: Adaptive Gradient Method Based on Estimates of Third-Order Moments". *4th IEEE International Conference on Data Science in Cyberspace, DSC*. 2019 (cited on page 40).

[280] Wonyong Sung, Iksoo Choi, Jinhwan Park, Seokhyun Choi, and Sungho Shin. "S-SGD: Symmetrical Stochastic Gradient Descent with Weight Noise Injection for Reaching Flat Minima". 2020. arXiv: `2009.02479` (cited on page 40).

[281] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. "On the importance of initialization and momentum in deep learning". *30th International Conference on Machine Learning, ICML*. 2013 (cited on pages 42–44).

[282] Richard Sutton. “Two problems with back propagation and other steepest descent learning procedures for networks”. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. 1986 (cited on pages 41, 42).

[283] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. “Inception-v4, Inception-ResNet and the impact of residual connections on learning.” *31st AAAI Conference on Artificial Intelligence, AAAI*. 2017 (cited on pages 44, 45, 69, 77).

[284] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. “Going Deeper With Convolutions”. *The IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2015 (cited on pages 43, 44, 69).

[285] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. “Rethinking the Inception Architecture for Computer Vision”. *The IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2016 (cited on pages 85, 86).

[286] Kei Takeuchi. “The distribution of information statistics and the criterion of goodness of fit of models”. *Mathematical Science* (1976) (cited on pages 113, 164).

[287] Conghui Tan, Shiqian Ma, Yu-Hong Dai, and Yuqiu Qian. “Barzilai-Borwein Step Size for Stochastic Gradient Descent”. *Advances in Neural Information Processing Systems 29, NIPS*. 2016 (cited on page 40).

[288] Mingxing Tan and Quoc Le. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”. *36th International Conference on Machine Learning, ICML*. 2019 (cited on pages 44, 45).

[289] Zeyi Tao, Qi Xia, and Qun Li. “A new perspective in understanding of Adam-Type algorithms and beyond”. `https://openreview.net/forum?id=SyxM51BYPB`. 2019 (cited on page 40).

[290] Brian Teixeira, Birgi Tamersoy, Vivek Singh, and Ankur Kapoor. “Adaloss: Adaptive Loss Function for Landmark Localization”. 2019. arXiv: `1908.01070` (cited on page 40).

[291] Valentin Thomas, Fabian Pedregosa, Bart van Merriënboer, Pierre-Antoine Manzagol, Yoshua Bengio, and Nicolas Le Roux. “On the interplay between noise and curvature and its effect on optimization and generalization”. *23rd International Conference on Artificial Intelligence and Statistics, AISTATS*. 2020 (cited on pages 110, 111, 113, 153, 154, 164, 165).

[292] Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F. Manso. “The Computational Limits of Deep Learning”. 2020. arXiv: `2007.05558` (cited on page 118).

[293] Robert Tibshirani. “Regression Shrinkage and Selection via the Lasso”. *Journal of the Royal Statistical Society. Series B (Methodological)* (1996) (cited on pages 22, 23).

[294] Tijmen Tieleman and Geoffrey Hinton. “Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude”. 2012 (cited on pages 40, 44, 45, 98).

[295] Andrey Nikolaevich Tikhonov. “On the stability of inverse problems”. *Dokl. Akad. Nauk SSSR* (1943). In Russian (cited on pages 22, 24).

[296] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. “Chainer: a Next-Generation Open Source Framework for Deep Learning”. *NIPS Workshop on Machine Learning Systems (LearningSys)*. 2015 (cited on page 2).

[297] Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. “MLP-Mixer: An all-MLP Architecture for Vision”. 2021. arXiv: 2105.01601 (cited on page 72).

[298] Qianqian Tong, Guannan Liang, and Jinbo Bi. “Calibrating the Adaptive Learning Rate to Improve Convergence of ADAM”. 2019. arXiv: 1908.00700 (cited on page 40).

[299] Hoang Tran and Ashok Cutkosky. “Correcting Momentum with Second-order Information”. 2021. arXiv: 2103.03265 (cited on page 40).

[300] Phuong Thi Tran and Le Trieu Phong. “On the Convergence Proof of AMSGrad and a New Version”. *IEEE Access* (2019) (cited on page 40).

[301] Trang H. Tran, Lam M. Nguyen, and Quoc Tran-Dinh. “Shuffling Gradient-Based Methods with Momentum”. 2020. arXiv: 2011.11884 (cited on page 40).

[302] Vanessa Tsingunidis. “Bring the GANs into Action - Extending DeepOBS with novel test problems”. Bachelor Thesis. University of Tübingen. 2020 (cited on page 85).

[303] Rasul Tutunov, Minne Li, Alexander I. Cowen-Rivers, Jun Wang, and Haitham Bou-Ammar. “Compositional ADAM: An Adaptive Compositional Solver”. 2020. arXiv: 2002.03755 (cited on page 40).

[304] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. “Instance Normalization: The Missing Ingredient for Fast Stylization”. 2017. arXiv: 1607.08022 (cited on page 67).

[305] Arash Vahdat and Jan Kautz. “NVAE: A Deep Hierarchical Variational Autoencoder”. *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on page 74).

[306] Vladimir Vapnik and Alexey Chervonenkis. “On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities”. *Theory of Probability & Its Applications* (1971) (cited on page 17).

[307] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. “Attention Is All You Need”. *Advances in Neural Information Processing Systems 30, NIPS*. 2017 (cited on pages 2, 3, 45, 46, 57, 72, 77, 102, 104).

[308] Sharan Vaswani, Aaron Mishkin, Issam H. Laradji, Mark Schmidt, Gauthier Gidel, and Simon Lacoste-Julien. “Painless Stochastic Gradient: Interpolation, Line-Search, and Convergence Rates”. *Advances in Neural Information Processing Systems 32, NeurIPS*. 2019 (cited on pages 40, 56, 157).

[309] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. “PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization”. *Advances in Neural Information Processing Systems 32, NeurIPS*. 2019 (cited on page 40).

[310] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J. Lang. “Phoneme recognition using time-delay neural networks”. *IEEE Transactions on Acoustics, Speech, and Signal Processing* (1989) (cited on page 2).

[311] Bao Wang, Tan M. Nguyen, Andrea L. Bertozzi, Richard G. Baraniuk, and Stanley J. Osher. “Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent”. 2020. arXiv: 2002.10583 (cited on pages 40, 42, 43).

[312] Bao Wang and Qiang Ye. “Stochastic Gradient Descent with Nonlinear Conjugate Gradient-Style Adaptive Momentum”. 2020. arXiv: 2012.02188 (cited on page 40).

[313] Dong Wang, Yicheng Liu, Wenwo Tang, Fanhua Shang, Hongying Liu, Qigong Sun, and Licheng Jiao. "signADAM++: Learning Confidences for Deep Neural Networks". *International Conference on Data Mining Workshops, ICDM*. 2019 (cited on page 40).

[314] Jiaxuan Wang and Jenna Wiens. "AdaSGD: Bridging the gap between SGD and Adam". 2020. arXiv: 2006.16541 (cited on page 40).

[315] Shipeng Wang, Jian Sun, and Zongben Xu. "HyperAdam: A Learnable Task-Adaptive Adam for Network Training". *33rd AAAI Conference on Artificial Intelligence, AAAI*. 2019 (cited on page 40).

[316] James Warsa, Todd Wareing, Jim Morel, John Mcghee, and Richard Lehoucq. "Krylov Subspace Iterations for Deterministic k-Eigenvalue Calculations". *Nuclear Science and Engineering* (2004) (cited on page 163).

[317] Paul J. Werbos. "Applications of advances in nonlinear sensitivity analysis". *System Modeling and Optimization*. 1982 (cited on pages 2, 62).

[318] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. "The Marginal Value of Adaptive Gradient Methods in Machine Learning". *Advances in Neural Information Processing Systems 30, NIPS*. 2017 (cited on pages 45–47, 81, 92, 95).

[319] David H. Wolpert and William G. Macready. "No free lunch theorems for optimization". *IEEE Trans. Evol. Comput.* (1997) (cited on pages 14, 94).

[320] Less Wright. "Deep Memory". https://github.com/lessw2020/Best-Deep-Learning-Optimizers/tree/master/DeepMemory. GitHub repository. 2020 (cited on page 40).

[321] Less Wright. "Ranger". https://github.com/lessw2020/Ranger-Deep-Learning-Optimizer. GitHub repository. 2020 (cited on pages 40, 47).

[322] Xiaoxia Wu, Rachel Ward, and Léon Bottou. "WNGrad: Learn the Learning Rate in Gradient Descent". 2018. arXiv: 1803.02865 (cited on page 40).

[323] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". 2016. arXiv: 1609.08144 (cited on page 1).

[324] Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger B. Grosse. "Understanding short-horizon bias in stochastic meta-optimization". *6th International Conference on Learning Representations, ICLR - Conference Track Proceedings* (2018) (cited on pages 116, 129).

[325] Yuxin Wu and Kaiming He. "Group Normalization". *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018 (cited on page 67).

[326] Han Xiao, Kashif Rasul, and Roland Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms". 2017. arXiv: 1708.07747 (cited on pages 85, 86).

[327] Cong Xie, Oluwasanmi Koyejo, Indranil Gupta, and Haibin Lin. "Local AdaAlter: Communication-Efficient Stochastic Gradient Descent with Adaptive Learning Rates". 2019. arXiv: 1911.09030 (cited on page 40).

[328] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. “Aggregated Residual Transformations for Deep Neural Networks”. *The IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2017 (cited on page 69).

[329] Zeke Xie, Xinrui Wang, Huishuai Zhang, Issei Sato, and Masashi Sugiyama. “Adai: Separating the Effects of Adaptive Learning Rate and Momentum Inertia”. 2020. arXiv: 2006.15815 (cited on page 40).

[330] Chen Xing, Devansh Arpit, Christos Tsirigotis, and Yoshua Bengio. “A Walk with SGD”. 2018. arXiv: 1802.08770 (cited on pages 57, 100, 116, 140, 156).

[331] Yangyang Xu. “Momentum-based variance-reduced proximal stochastic gradient method for composite nonconvex stochastic optimization”. 2020. arXiv: 2006.00425 (cited on page 40).

[332] Minghan Yang, Dong Xu, Yongfeng Li, Zaiwen Wen, and Mengyun Chen. “Structured Stochastic Quasi-Newton Methods for Large-Scale Optimization Problems”. 2020. arXiv: 2006.09606 (cited on page 40).

[333] Zhewei Yao, Amir Gholami, Sheng Shen, Kurt Keutzer, and Michael W. Mahoney. “ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning”. 2020. arXiv: 2006.00719 (cited on pages 40, 53, 110, 111, 113, 119, 153, 154, 163, 164).

[334] Yang You, Igor Gitman, and Boris Ginsburg. “Large Batch Training of Convolutional Networks”. 2017. arXiv: 1708.03888 (cited on pages 40, 49).

[335] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. “Large Batch Optimization for Deep Learning: Training BERT in 76 minutes”. *8th International Conference on Learning Representations, ICLR*. 2020 (cited on pages 40, 50).

[336] Wei Yuan and Kai-Xin Gao. “EAdam Optimizer: How ϵ Impact Adam”. 2020. arXiv: 2011.02150 (cited on page 40).

[337] Xubo Yue, Maher Nouiehed, and Raed Al Kontar. “SALR: Sharpness-aware Learning Rates for Improved Generalization”. 2020. arXiv: 2011.05348 (cited on page 40).

[338] Jihun Yun, Aurelie C. Lozano, and Eunho Yang. “Stochastic Gradient Methods with Block Diagonal Matrix Adaptation”. 2019. arXiv: 1905.10757 (cited on page 40).

[339] Sergey Zagoruyko and Nikos Komodakis. “Wide Residual Networks”. *Procedings of the British Machine Vision Conference*. 2016 (cited on pages 85, 86, 136).

[340] Manzil Zaheer, Sashank J. Reddi, Devendra Singh Sachan, Satyen Kale, and Sanjiv Kumar. “Adaptive Methods for Nonconvex Optimization”. *Advances in Neural Information Processing Systems 31, NeurIPS*. 2018 (cited on page 40).

[341] Matthew D. Zeiler. “ADADELTA: An Adaptive Learning Rate Method”. 2012. arXiv: 1212.5701 (cited on pages 40, 44, 45, 80, 98).

[342] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. “Dive into Deep Learning”. 2020 (cited on page 100).

[343] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. “Understanding deep learning requires rethinking generalization”. *5th International Conference on Learning Representations, ICLR*. 2017 (cited on pages 17, 18, 61).

[344] Guodong Zhang, Shengyang Sun, David Duvenaud, and Roger Grosse. “Noisy Natural Gradient as Variational Inference”. *35th International Conference on Machine Learning, ICML*. 2018 (cited on page 40).

[345] Guoqiang Zhang and Haopeng Li. “Effectiveness of Scaled Exponentially-Regularized Linear Units (SERLUs)”. 2018. arXiv: 1807.10117 (cited on page 69).

[346] Jian Zhang and Ioannis Mitliagkas. “YellowFin and the Art of Momentum Tuning”. *Machine Learning and Systems, MLSys*. 2019 (cited on pages 40, 56).

[347] Jiawei Zhang and Fisher B. Gouza. “GADAM: Genetic-Evolutionary ADAM for Deep Neural Network Optimization”. 2018. arXiv: 1805.07500 (cited on page 40).

[348] Jingzhao Zhang, Sai Praneeth Karimireddy, Andreas Veit, Seungyeon Kim, Sashank J. Reddi, Sanjiv Kumar, and Suvrit Sra. “Why are Adaptive Methods Good for Attention Models?” *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on page 40).

[349] Michael R. Zhang, James Lucas, Geoffrey Hinton, and Jimmy Ba. “Lookahead Optimizer: k steps forward, 1 step back”. *Advances in Neural Information Processing Systems 32, NeurIPS*. 2019 (cited on pages 40, 47, 98).

[350] Zijun Zhang, Lin Ma, Zongpeng Li, and Chuan Wu. “Normalized Direction-preserving Adam”. 2017. arXiv: 1709.04546 (cited on page 40).

[351] Ziming Zhang, Yuanwei Wu, and Guanghui Wang. “BPGrad: Towards Global Optimality in Deep Learning via Branch and Pruning”. 2017. arXiv: 1711.06959 (cited on page 40).

[352] Shen-Yi Zhao, Yin-Peng Xie, and Wu-Jun Li. “Stochastic Normalized Gradient Descent with Momentum for Large Batch Training”. 2020. arXiv: 2007.13985 (cited on page 40).

[353] Bingxin Zhou, Xuebin Zheng, and Junbin Gao. “On the Trend-corrected Variant of Adaptive Stochastic Optimization Methods”. *International Joint Conference on Neural Networks, IJCNN*. 2020 (cited on page 40).

[354] Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven C. H. Hoi, and E. Weinan. “Towards Theoretically Understanding Why SGD Generalizes Better Than Adam in Deep Learning”. *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on pages 45, 46).

[355] Yi-Tong Zhou and Rama Chellappa. “Computation of optical flow using a neural network”. *IEEE 1988 International Conference on Neural Networks* (1988) (cited on page 66).

[356] Yangfan Zhou, Kaizhu Huang, Cheng Cheng, Xuguang Wang, and Xin Liu. “FastAdaBelief: Improving Convergence Rate for Belief-based Adaptive Optimizer by Strong Convexity”. 2021. arXiv: 2104.13790 (cited on page 40).

[357] Yingxue Zhou, Xinyan Li, and Arindam Banerjee. “Noisy Truncated SGD: Optimization and Generalization”. 2021. arXiv: 2103.00075 (cited on page 40).

[358] Zhiming Zhou, Qingru Zhang, Guansong Lu, Hongwei Wang, Weinan Zhang, and Yong Yu. “AdaShift: Decorrelation and Convergence of Adaptive Learning Rate Methods”. *7th International Conference on Learning Representations, ICLR*. 2019 (cited on page 40).

[359] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. “TBD: Benchmarking and Analyzing Deep Neural Network Training”. 2018. arXiv: 1803.06905 (cited on pages 80, 81).

[360] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. “Unpaired Image-To-Image Translation Using Cycle-Consistent Adversarial Networks”. *IEEE/CVF International Conference on Computer Vision, ICCV*. 2017 (cited on pages 45, 46).

[361] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James S. Duncan. “AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients”. *Advances in Neural Information Processing Systems 33, NeurIPS*. 2020 (cited on pages 40, 49, 98).

[362] Liu Ziyin, Zhikang T. Wang, and Masahito Ueda. “LaProp: a Better Way to Combine Momentum with Adaptive Gradient”. 2020. arXiv: 2002.04839 (cited on page 40).