arxlib_google_tuning_playbook_v1

Enjoy the 8 papers in this Arxiv Libro

1. 2203.15556 / Hoffmann / Training Compute-Optimal Large Language Models (1kc / 2r / 36p)
2. 2003.04887 / Bachlec. / ReZero is All You Need: Fast Convergence at Large Depth (231c / 13r / 14p)
3. 1910.05446 / Choi / On Empirical Comparisons of Optimizers for Deep Learnin... (218c / 2r / 27p)
4. 1811.03600 / Shallue / Measuring the Effects of Data Parallelism on Neural Net... (368c / 0r / 49p)
5. 1803.02021 / Wu / Understanding Short-Horizon Bias in Stochastic Meta-Opt... (133c / 1r / 17p)
6. 1706.03200 / Bousquet / Critical Hyper-Parameters: No Random, No Cry (35c / 0r / 19p)
7. 1705.08741 / Hoffer / Train longer, generalize better: closing the generaliza... (743c / 1r / 15p)
8. 1403.5607 / Gelbart / Bayesian Optimization with Unknown Constraints (409c / 1r / 14p)

1. Training Compute-Optimal Large Language Models

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae et al

Chatgpt summary: The study explores the relationship between model size and training data for transformer language models, finding that for optimal training, both model size and number of training tokens should be scaled equally, leading to the development of Chinchilla, a high-performing model that outperforms other large language models on various tasks using less compute resources.

We investigate the optimal model size and number of tokens for training a transformer language model under a given compute budget. We find that current large language models are significantly undertrained, a consequence of the recent focus on scaling language models whilst keeping the amount of training data constant. By training over 400 language models ranging from 70 million to over 16 billion parameters on 5 to 500 billion tokens, we find that for compute-optimal training, the model size and the number of training tokens should be scaled equally: for every doubling of model size the number of training tokens should also be doubled. We test this hypothesis by training a predicted compute-optimal model, Chinchilla, that uses the same compute budget as Gopher but with 70B parameters and 4$\\times$ more more data. Chinchilla uniformly and significantly outperforms Gopher (280B), GPT-3 (175B), Jurassic-1 (178B), and Megatron-Turing NLG (530B) on a large range of downstream evaluation tasks. This also means that Chinchilla uses substantially less compute for fine-tuning and inference, greatly facilitating downstream usage. As a highlight, Chinchilla reaches a state-of-the-art average accuracy of 67.5% on the MMLU benchmark, greater than a 7% improvement over Gopher.

http://arxiv.org/abs/2203.15556v1
Published: 2022, category: cs.CL
Cited by: 1276, Citing: 78, Citation velocity: 417
Pages: 36
Repos: 2 (0 official), Total stars: 16,996

https://paperswithcode.com/paper/training-compute-optimal-large-language

https://github.com/karpathy/llama2.c (16,971 stars, pytorch)
https://github.com/nkluge-correa/teenytinyllama (25 stars, pytorch)

2. ReZero is All You Need: Fast Convergence at Large Depth

Thomas Bachlechner, Bodhisattwa Prasad Majumder, Huanru Henry Mao, Garrison W. Cottrell, Julian McAuley

Chatgpt summary: Efficient signal propagation in deep networks can be improved by using a simple gating mechanism that initiates dynamical isometry and outperforms more complex approaches, allowing for fast convergence and better performance in training deep networks on various tasks such as image classification and language modeling.

Deep networks often suffer from vanishing or exploding gradients due to inefficient signal propagation, leading to long training times or convergence difficulties. Various architecture designs, sophisticated residual-style networks, and initialization schemes have been shown to improve deep signal propagation. Recently, Pennington et al. used free probability theory to show that dynamical isometry plays an integral role in efficient deep learning. We show that the simplest architecture change of gating each residual connection using a single zero-initialized parameter satisfies initial dynamical isometry and outperforms more complex approaches. Although much simpler than its predecessors, this gate enables training thousands of fully connected layers with fast convergence and better test performance for ResNets trained on CIFAR-10. We apply this technique to language modeling and find that we can easily train 120-layer Transformers. When applied to 12 layer Transformers, it converges 56% faster on enwiki8.

http://arxiv.org/abs/2003.04887v2
Published: 2020, category: cs.LG
Cited by: 231, Citing: 46, Citation velocity: 56
Pages: 14
Repos: 13 (1 official), Total stars: 4,425

https://paperswithcode.com/paper/rezero-is-all-you-need-fast-convergence-at
https://github.com/majumderb/rezero (404 stars, pytorch) [OFFICIAL]
https://github.com/lucidrains/reformer-pytorch (2,083 stars, pytorch)
https://github.com/lucidrains/performer-pytorch (1,073 stars, pytorch)
https://github.com/EugenHotaj/pytorch-generative/blob/master/pytorch_generative/nn/utils.py (420 stars, pytorc

3. On Empirical Comparisons of Optimizers for Deep Learning

Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, George E. Dahl

Chatgpt summary: The paper highlights the importance of selecting an optimizer in deep learning, demonstrating that optimizer comparisons are highly sensitive to hyperparameter tuning protocols, indicating that the hyperparameter search space significantly influences empirical rankings and can lead to contradictory results, emphasizing the importance of considering the inclusion relationships between

optimizers, particularly noting that adaptive gradient methods generally outperform momentum or gradient descent.

Selecting an optimizer is a central step in the contemporary deep learning pipeline. In this paper, we demonstrate the sensitivity of optimizer comparisons to the hyperparameter tuning protocol. Our findings suggest that the hyperparameter search space may be the single most important factor explaining the rankings obtained by recent empirical comparisons in the literature. In fact, we show that these results can be contradicted when hyperparameter search spaces are changed. As tuning effort grows without bound, more general optimizers should never underperform the ones they can approximate (i.e., Adam should never perform worse than momentum), but recent attempts to compare optimizers either assume these inclusion relationships are not practically relevant or restrict the hyperparameters in ways that break the inclusions. In our experiments, we find that inclusion relationships between optimizers matter in practice and always predict optimizer comparisons. In particular, we find that the popular adaptive gradient methods never underperform momentum or gradient descent. We also report practical tips around tuning often ignored hyperparameters of adaptive gradient methods and raise concerns about fairly benchmarking optimizers for neural network training.

http://arxiv.org/abs/1910.05446v3
Published: 2019, updated: 2020, category: cs.LG
Cited by: 218, Citing: 72, Citation velocity: 40
Pages: 27
Repos: 2 (0 official), Total stars: 9

https://paperswithcode.com/paper/on-empirical-comparisons-of-optimizers-for
https://github.com/vrunm/text-classification-financial-phrase-bank (7 stars, pytorch)
https://github.com/yorkerlin/remove-the-square-root (2 stars, pytorch)

4. Measuring the Effects of Data Parallelism on Neural Network Training

Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, George E. Dahl

Chatgpt summary: The study explores the impact of increasing batch sizes on neural network training time and performance, highlighting significant variability across different algorithms and datasets, with findings suggesting that larger batch sizes do not necessarily deteriorate out-of-sample performance.

Recent hardware developments have dramatically increased the scale of data parallelism available for neural network training. Among the simplest ways to harness next-generation hardware is to increase the batch size in standard mini-batch neural network training algorithms. In this work, we aim to experimentally characterize the effects of increasing the batch size on training time, as measured by the number of steps necessary to reach a goal out-of-sample error. We study how this relationship varies with the training algorithm, model, and data set, and find extremely large variation between workloads. Along the way, we show that disagreements in the literature on how batch size affects model quality can largely be explained by differences in metaparameter tuning and compute budgets at different batch sizes. We find no evidence that larger batch sizes degrade out-of-sample performance. Finally, we discuss the implications of our results on efforts to train neural networks much faster in the future. Our

experimental data is publicly available as a database of 71,638,836 loss measurements taken over the course of training for 168,160 individual models across 35 workloads.

http://arxiv.org/abs/1811.03600v3
Published: 2018, updated: 2019, category: cs.LG
Cited by: 368, Citing: 74, Citation velocity: 53
Pages: 49
Repos: 0

5. Understanding Short-Horizon Bias in Stochastic Meta-Optimization

Yuhuai Wu, Mengye Ren, Renjie Liao, Roger Grosse

Chatgpt summary: The text discusses the importance of tuning the learning rate for effective neural network training and highlights the short-horizon bias that arises in gradient-based meta-optimization when defining meta-objectives with time horizons shorter than typical neural net training, leading to the selection of excessively small learning rates, creating a fundamental problem in scaling meta-optimization for practical neural network training.

Careful tuning of the learning rate, or even schedules thereof, can be crucial to effective neural net training. There has been much recent interest in gradient-based meta-optimization, where one tunes hyperparameters, or even learns an optimizer, in order to minimize the expected loss when the training procedure is unrolled. But because the training procedure must be unrolled thousands of times, the meta-objective must be defined with an orders-of-magnitude shorter time horizon than is typical for neural net training. We show that such short-horizon meta-objectives cause a serious bias towards small step sizes, an effect we term short-horizon bias. We introduce a toy problem, a noisy quadratic cost function, on which we analyze short-horizon bias by deriving and comparing the optimal schedules for short and long time horizons. We then run meta-optimization experiments (both offline and online) on standard benchmark datasets, showing that meta-optimization chooses too small a learning rate by multiple orders of magnitude, even when run with a moderately long time horizon (100 steps) typical of work in the area. We believe short-horizon bias is a fundamental problem that needs to be addressed if meta-optimization is to scale to practical neural net training regimes.

http://arxiv.org/abs/1803.02021v1
Published: 2018, category: cs.LG
Comment: 17 pages, 8 figures; To appear in ICLR2018
Cited by: 133, Citing: 44, Citation velocity: 17
Pages: 17
Repos: 1 (1 official), Total stars: 37

https://paperswithcode.com/paper/understanding-short-horizon-bias-in
https://github.com/renmengye/meta-optim-public (37 stars, tf) [OFFICIAL]

6. Critical Hyper-Parameters: No Random, No Cry

Olivier Bousquet, Sylvain Gelly, Karol Kurach, Olivier Teytaud, Damien Vincent

Chatgpt summary: The text discusses the importance of hyper-parameter selection in Deep Learning and suggests the use of Low Discrepancy Sequences (LDS) as a more efficient alternative to grid search and random search for optimizing hyperparameters of complex models.

The selection of hyper-parameters is critical in Deep Learning. Because of the long training time of complex models and the availability of compute resources in the cloud, "one-shot" optimization schemes - where the sets of hyper-parameters are selected in advance (e.g. on a grid or in a random manner) and the training is executed in parallel - are commonly used. It is known that grid search is sub-optimal, especially when only a few critical parameters matter, and suggest to use random search instead. Yet, random search can be "unlucky" and produce sets of values that leave some part of the domain unexplored. Quasi-random methods, such as Low Discrepancy Sequences (LDS) avoid these issues. We show that such methods have theoretical properties that make them appealing for performing hyperparameter search, and demonstrate that, when applied to the selection of hyperparameters of complex Deep Learning models (such as state-of-the-art LSTM language models and image classification models), they yield suitable hyperparameters values with much fewer runs than random search. We propose a particularly simple LDS method which can be used as a drop-in replacement for grid or random search in any Deep Learning pipeline, both as a fully one-shot hyperparameter search or as an initializer in iterative batch optimization.

http://arxiv.org/abs/1706.03200v1
Published: 2017, category: cs.LG
Cited by: 35, Citing: 26, Citation velocity: 0
Pages: 19
Repos: 0

7. Train longer, generalize better: closing the generalization gap in large batch training of neural networks

Elad Hoffer, Itay Hubara, Daniel Soudry

Chatgpt summary: The text discusses the issue of the "generalization gap" in deep learning models trained with large batch sizes, identifying that the gap is caused by a small number of weight updates rather than the batch size itself, proposing a "random walk on random landscape" statistical model to explain the phenomenon, introducing the "Ghost Batch Normalization" algorithm to reduce the gap without increasing the number of updates, and offering insights on optimal training practices for achieving good generalization.

Background: Deep learning models are typically trained using stochastic gradient descent or one of its variants. These methods update the weights using their gradient, estimated from a small fraction of the training data. It has been observed that when using large batch sizes there is a persistent degradation in generalization performance - known as the "generalization gap" phenomena. Identifying the origin of this gap and closing it had remained an open problem. Contributions: We examine the initial high learning rate training phase. We find that the weight distance from its initialization grows logarithmically with the

number of weight updates. We therefore propose a "random walk on random landscape" statistical model which is known to exhibit similar "ultra-slow" diffusion behavior. Following this hypothesis we conducted experiments to show empirically that the "generalization gap" stems from the relatively small number of updates rather than the batch size, and can be completely eliminated by adapting the training regime used. We further investigate different techniques to train models in the large-batch regime and present a novel algorithm named "Ghost Batch Normalization" which enables significant decrease in the generalization gap without increasing the number of updates. To validate our findings we conduct several additional experiments on MNIST, CIFAR-10, CIFAR-100 and ImageNet. Finally, we reassess common practices and beliefs concerning training of deep models and suggest they may not be optimal to achieve good generalization.

http://arxiv.org/abs/1705.08741v2
Published: 2017, updated: 2018, category: stat.ML
Cited by: 743, Citing: 50, Citation velocity: 92
Pages: 15
Repos: 1 (1 official), Total stars: 148

https://paperswithcode.com/paper/train-longer-generalize-better-closing-the
https://github.com/eladhoffer/bigBatch (148 stars, pytorch) [OFFICIAL]

8. Bayesian Optimization with Unknown Constraints

Michael A. Gelbart, Jasper Snoek, Ryan P. Adams

Chatgpt summary: The text discusses the application of Bayesian optimization to constrained problems with noisy constraint functions, independent evaluation of objectives and constraints, and demonstrates its effectiveness through various practical examples.

Recent work on Bayesian optimization has shown its effectiveness in global optimization of difficult black-box objective functions. Many real-world optimization problems of interest also have constraints which are unknown a priori. In this paper, we study Bayesian optimization for constrained problems in the general case that noise may be present in the constraint functions, and the objective and constraints may be evaluated independently. We provide motivating practical examples, and present a general framework to solve such problems. We demonstrate the effectiveness of our approach on optimizing the performance of online latent Dirichlet allocation subject to topic sparsity constraints, tuning a neural network given test-time memory constraints, and optimizing Hamiltonian Monte Carlo to achieve maximal effectiveness in a fixed time, subject to passing standard convergence diagnostics.

http://arxiv.org/abs/1403.5607v1
Published: 2014, category: stat.ML
Comment: 14 pages, 3 figures
Cited by: 409, Citing: 36, Citation velocity: 50
Pages: 14
Repos: 1 (0 official), Total stars: 1,545

https://paperswithcode.com/paper/bayesian-optimization-with-unknown

arxlib_google_tuning_playbook_v1

https://github.com/HIPS/Spearmint (1545 stars)

DeepMind

Training Compute-Optimal Large Language Models

Jordan Hoffmann*, Sebastian Borgeaud*, Arthur Mensch*, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals and Laurent Sifre*

*Equal contributions

We investigate the optimal model size and number of tokens for training a transformer language model under a given compute budget. We find that current large language models are significantly undertrained, a consequence of the recent focus on scaling language models whilst keeping the amount of training data constant. By training over 400 language models ranging from 70 million to over 16 billion parameters on 5 to 500 billion tokens, we find that for compute-optimal training, the model size and the number of training tokens should be scaled equally: for every doubling of model size the number of training tokens should also be doubled. We test this hypothesis by training a predicted compute-optimal model, *Chinchilla*, that uses the same compute budget as *Gopher* but with 70B parameters and 4× more more data. *Chinchilla* uniformly and significantly outperforms *Gopher* (280B), GPT-3 (175B), Jurassic-1 (178B), and Megatron-Turing NLG (530B) on a large range of downstream evaluation tasks. This also means that *Chinchilla* uses substantially less compute for fine-tuning and inference, greatly facilitating downstream usage. As a highlight, *Chinchilla* reaches a state-of-the-art average accuracy of 67.5% on the MMLU benchmark, greater than a 7% improvement over *Gopher*.

1. Introduction

Recently a series of *Large Language Models* (LLMs) have been introduced (Brown et al., 2020; Lieber et al., 2021; Rae et al., 2021; Smith et al., 2022; Thoppilan et al., 2022), with the largest dense language models now having over 500 billion parameters. These large autoregressive transformers (Vaswani et al., 2017) have demonstrated impressive performance on many tasks using a variety of evaluation protocols such as zero-shot, few-shot, and fine-tuning.

The compute and energy cost for training large language models is substantial (Rae et al., 2021; Thoppilan et al., 2022) and rises with increasing model size. In practice, the allocated training compute budget is often known in advance: how many accelerators are available and for how long we want to use them. Since it is typically only feasible to train these large models once, accurately estimating the best model hyperparameters for a given compute budget is critical (Tay et al., 2021).

Kaplan et al. (2020) showed that there is a power law relationship between the number of parameters in an autoregressive language model (LM) and its performance. As a result, the field has been training larger and larger models, expecting performance improvements. One notable conclusion in Kaplan et al. (2020) is that large models should not be trained to their lowest possible loss to be compute optimal. Whilst we reach the same conclusion, we estimate that large models should be trained for many more training tokens than recommended by the authors. Specifically, given a 10× increase computational budget, they suggests that the size of the model should increase 5.5× while the number of training tokens should only increase 1.8×. Instead, we find that model size and the number of training tokens should be scaled in equal proportions.

Following Kaplan et al. (2020) and the training setup of GPT-3 (Brown et al., 2020), many of the recently trained large models have been trained for approximately 300 billion tokens (Table 1), in line with the approach of predominantly increasing model size when increasing compute.

Corresponding authors: {jordanhoffmann|sborgeaud|amensch|sifre}@deepmind.com

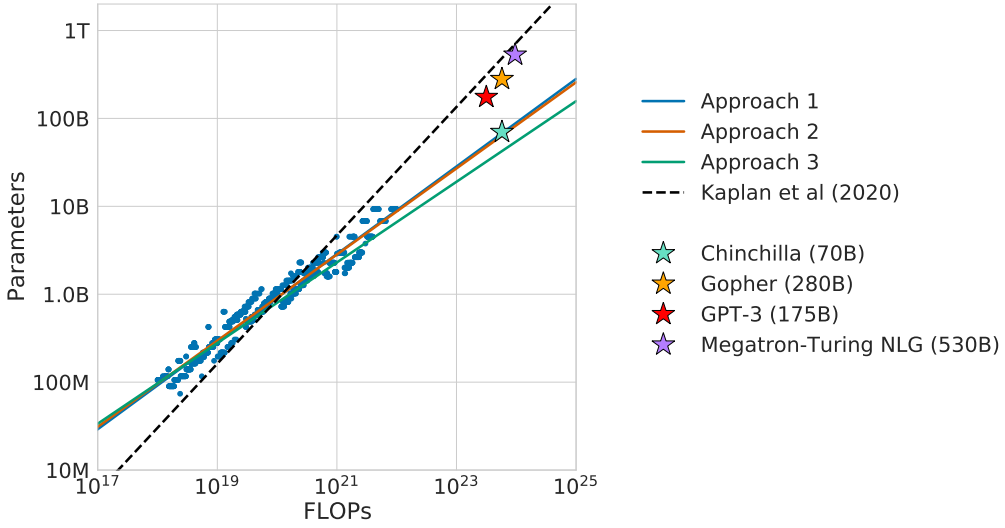


Figure 1 | **Overlaid predictions.** We overlay the predictions from our three different approaches, along with projections from Kaplan et al. (2020). We find that all three methods predict that current large models should be substantially smaller and therefore trained much longer than is currently done. In Figure A3, we show the results with the predicted optimal tokens plotted against the optimal number of parameters for fixed FLOP budgets. ***Chinchilla* outperforms *Gopher* and the other large models (see Section 4.2).**

In this work, we revisit the question: *Given a fixed FLOPs budget,*[1] *how should one trade-off model size and the number of training tokens?* To answer this question, we model the final pre-training loss[2] $L(N, D)$ as a function of the number of model parameters N, and the number of training tokens, D. Since the computational budget C is a deterministic function $\text{FLOPs}(N, D)$ of the number of seen training tokens and model parameters, we are interested in minimizing L under the constraint $\text{FLOPs}(N, D) = C$:

$$N_{opt}(C), D_{opt}(C) = \underset{N,D \text{ s.t. } \text{FLOPs}(N,D)=C}{\text{argmin}} L(N, D). \tag{1}$$

The functions $N_{opt}(C)$, and $D_{opt}(C)$ describe the optimal allocation of a computational budget C. We empirically estimate these functions based on the losses of over 400 models, ranging from under 70M to over 16B parameters, and trained on 5B to over 400B tokens – with each model configuration trained for several different training horizons. Our approach leads to considerably different results than that of Kaplan et al. (2020). We highlight our results in Figure 1 and how our approaches differ in Section 2.

Based on our estimated compute-optimal frontier, we predict that for the compute budget used to train *Gopher*, an optimal model should be 4 times smaller, while being training on 4 times more tokens. We verify this by training a more *compute-optimal* 70B model, called *Chinchilla*, on 1.4 trillion tokens. Not only does *Chinchilla* outperform its much larger counterpart, *Gopher*, but its reduced model size reduces inference cost considerably and greatly facilitates downstream uses on smaller hardware. The energy cost of a large language model is amortized through its usage for inference an fine-tuning. The benefits of a more optimally trained smaller model, therefore, extend beyond the immediate benefits of its improved performance.

[1] For example, knowing the number of accelerators and a target training duration.

[2] For simplicity, we perform our analysis on the smoothed training loss which is an unbiased estimate of the test loss, as we are in the infinite data regime (the number of training tokens is less than the number of tokens in the entire corpus).

Table 1 | **Current LLMs**. We show five of the current largest dense transformer models, their size, and the number of training tokens. Other than LaMDA (Thoppilan et al., 2022), most models are trained for approximately 300 billion tokens. We introduce *Chinchilla*, a substantially smaller model, trained for much longer than 300B tokens.

Model	Size (# Parameters)	Training Tokens
LaMDA (Thoppilan et al., 2022)	137 Billion	168 Billion
GPT-3 (Brown et al., 2020)	175 Billion	300 Billion
Jurassic (Lieber et al., 2021)	178 Billion	300 Billion
Gopher (Rae et al., 2021)	280 Billion	300 Billion
MT-NLG 530B (Smith et al., 2022)	530 Billion	270 Billion
Chinchilla	70 Billion	1.4 Trillion

2. Related Work

Large language models. A variety of large language models have been introduced in the last few years. These include both dense transformer models (Brown et al., 2020; Lieber et al., 2021; Rae et al., 2021; Smith et al., 2022; Thoppilan et al., 2022) and mixture-of-expert (MoE) models (Du et al., 2021; Fedus et al., 2021; Zoph et al., 2022). The largest dense transformers have passed 500 billion parameters (Smith et al., 2022). The drive to train larger and larger models is clear—so far increasing the size of language models has been responsible for improving the state-of-the-art in many language modelling tasks. Nonetheless, large language models face several challenges, including their overwhelming computational requirements (the cost of training and inference increase with model size) (Rae et al., 2021; Thoppilan et al., 2022) and the need for acquiring more high-quality training data. In fact, in this work we find that larger, high quality datasets will play a key role in any further scaling of language models.

Modelling the scaling behavior. Understanding the scaling behaviour of language models and their transfer properties has been important in the development of recent large models (Hernandez et al., 2021; Kaplan et al., 2020). Kaplan et al. (2020) first showed a predictable relationship between model size and loss over many orders of magnitude. The authors investigate the question of choosing the optimal model size to train for a given compute budget. Similar to us, they address this question by training various models. Our work differs from Kaplan et al. (2020) in several important ways. First, the authors use a fixed number of training tokens and learning rate schedule for all models; this prevents them from modelling the impact of these hyperparameters on the loss. In contrast, we find that setting the learning rate schedule to approximately match the number of training tokens results in the best final loss regardless of model size—see Figure A1. For a fixed learning rate cosine schedule to 130B tokens, the intermediate loss estimates (for $D' << 130\text{B}$) are therefore overestimates of the loss of a model trained with a schedule length matching D'. Using these intermediate losses results in underestimating the effectiveness of training models on less data than 130B tokens, and eventually contributes to the conclusion that model size should increase faster than training data size as compute budget increases. In contrast, our analysis predicts that both quantities should scale at roughly the same rate. Secondly, we include models with up to 16B parameters, as we observe that there is slight curvature in the FLOP-loss frontier (see Appendix E)—in fact, the majority of the models used in our analysis have more than 500 million parameters, in contrast the majority of runs in Kaplan et al. (2020) are significantly smaller—many being less than 100M parameters.

Recently, Clark et al. (2022) specifically looked in to the scaling properties of Mixture of Expert

language models, showing that the scaling with number of experts diminishes as the model size increases—their approach models the loss as a function of two variables: the model size and the number of experts. However, the analysis is done with a fixed number of training tokens, as in Kaplan et al. (2020), potentially underestimating the improvements of branching.

Estimating hyperparameters for large models. The model size and the number of training tokens are not the only two parameters to chose when selecting a language model and a procedure to train it. Other important factors include learning rate, learning rate schedule, batch size, optimiser, and width-to-depth ratio. In this work, we focus on model size and the number of training steps, and we rely on existing work and provided experimental heuristics to determine the other necessary hyperparameters. Yang et al. (2021) investigates how to choose a variety of these parameters for training an autoregressive transformer, including the learning rate and batch size. McCandlish et al. (2018) finds only a weak dependence between optimal batch size and model size. Shallue et al. (2018); Zhang et al. (2019) suggest that using larger batch-sizes than those we use is possible. Levine et al. (2020) investigates the optimal depth-to-width ratio for a variety of standard model sizes. We use slightly less deep models than proposed as this translates to better wall-clock performance on our hardware.

Improved model architectures. Recently, various promising alternatives to traditional dense transformers have been proposed. For example, through the use of conditional computation large MoE models like the 1.7 trillion parameter Switch transformer (Fedus et al., 2021), the 1.2 Trillion parameter GLaM model (Du et al., 2021), and others (Artetxe et al., 2021; Zoph et al., 2022) are able to provide a large effective model size despite using relatively fewer training and inference FLOPs. However, for very large models the computational benefits of routed models seems to diminish (Clark et al., 2022). An orthogonal approach to improving language models is to augment transformers with explicit retrieval mechanisms, as done by Borgeaud et al. (2021); Guu et al. (2020); Lewis et al. (2020). This approach effectively increases the number of data tokens seen during training (by a factor of ~ 10 in Borgeaud et al. (2021)). This suggests that the performance of language models may be more dependant on the size of the training data than previously thought.

3. Estimating the optimal parameter/training tokens allocation

We present three different approaches to answer the question driving our research: *Given a fixed FLOPs budget, how should one trade-off model size and the number of training tokens?* In all three cases we start by training a range of models varying both model size and the number of training tokens and use the resulting training curves to fit an empirical estimator of how they should scale. We assume a power-law relationship between compute and model size as done in Clark et al. (2022); Kaplan et al. (2020), though future work may want to include potential curvature in this relationship for large model sizes. The resulting predictions are similar for all three methods and suggest that parameter count and number of training tokens should be increased equally with more compute[3]—with proportions reported in Table 2. This is in clear contrast to previous work on this topic and warrants further investigation.

[3]We compute FLOPs as described in Appendix F.

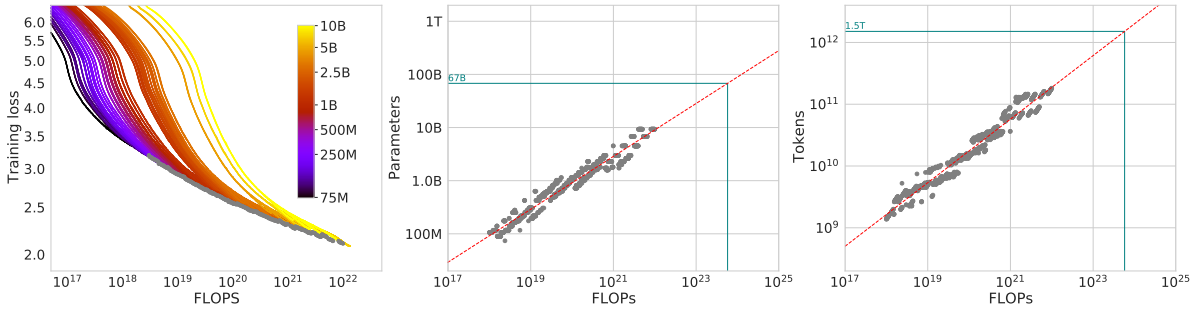


Figure 2 | **Training curve envelope.** On the **left** we show all of our different runs. We launched a range of model sizes going from 70M to 10B, each for four different cosine cycle lengths. From these curves, we extracted the envelope of minimal loss per FLOP, and we used these points to estimate the optimal model size (**center**) for a given compute budget and the optimal number of training tokens (**right**). In green, we show projections of optimal model size and training token count based on the number of FLOPs used to train *Gopher* (5.76×10^{23}).

3.1. Approach 1: Fix model sizes and vary number of training tokens

In our first approach we vary the number of training steps for a fixed family of models (ranging from 70M to over 10B parameters), training each model for 4 different number of training sequences. From these runs, we are able to directly extract an estimate of the minimum loss achieved for a given number of training FLOPs. Training details for this approach can be found in Appendix D.

For each parameter count N we train 4 different models, decaying the learning rate by a factor of 10× over a horizon (measured in number of training tokens) that ranges by a factor of 16×. Then, for each run, we smooth and then interpolate the training loss curve. From this, we obtain a continuous mapping from FLOP count to training loss for each run. Then, for each FLOP count, we determine which run achieves the lowest loss. Using these interpolants, we obtain a mapping from any FLOP count C, to the most efficient choice of model size N and number of training tokens D such that $\text{FLOPs}(N, D) = C$.[4] At 1500 logarithmically spaced FLOP values, we find which model size achieves the lowest loss of all models along with the required number of training tokens. Finally, we fit power laws to estimate the optimal model size and number of training tokens for any given amount of compute (see the center and right panels of Figure 2), obtaining a relationship $N_{opt} \propto C^a$ and $D_{opt} \propto C^b$. We find that $a = 0.50$ and $b = 0.50$—as summarized in Table 2. In Section D.4, we show a head-to-head comparison at 10^{21} FLOPs, using the model size recommended by our analysis and by the analysis of Kaplan et al. (2020)—using the model size we predict has a clear advantage.

3.2. Approach 2: IsoFLOP profiles

In our second approach we vary the model size[5] for a fixed set of 9 different training FLOP counts[6] (ranging from 6×10^{18} to 3×10^{21} FLOPs), and consider the final training loss for each point[7]. in contrast with Approach 1 that considered points (N, D, L) along the entire training runs. This allows us to directly answer the question: For a given FLOP budget, what is the optimal parameter count?

[4]Note that all selected points are within the last 15% of training. This suggests that when training a model over D tokens, we should pick a cosine cycle length that decays 10× over approximately D tokens—see further details in Appendix B.

[5]In approach 2, model size varies up to 16B as opposed to approach 1 where we only used models up to 10B.

[6]The number of training tokens is determined by the model size and training FLOPs.

[7]We set the cosine schedule length to match the number of tokens, which is optimal according to the analysis presented in Appendix B.

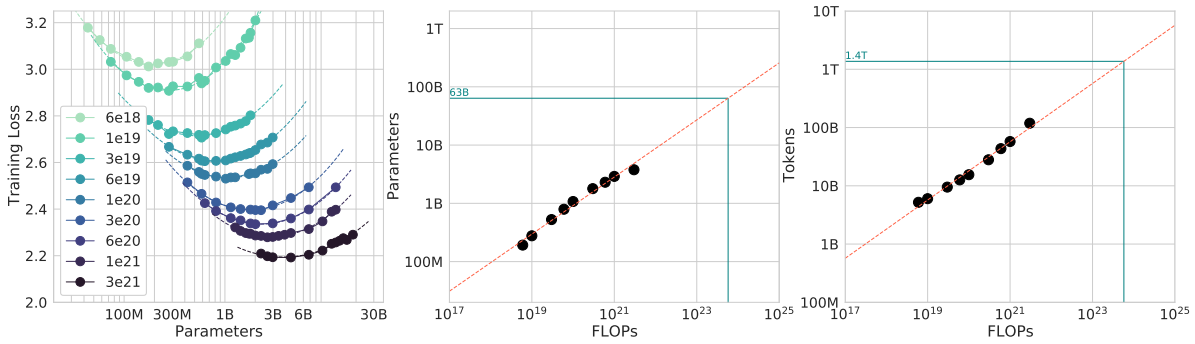


Figure 3 | **IsoFLOP curves.** For various model sizes, we choose the number of training tokens such that the final FLOPs is a constant. The cosine cycle length is set to match the target FLOP count. We find a clear valley in loss, meaning that for a given FLOP budget there is an optimal model to train (**left**). Using the location of these valleys, we project optimal model size and number of tokens for larger models (**center** and **right**). In green, we show the estimated number of parameters and tokens for an *optimal* model trained with the compute budget of *Gopher*.

For each FLOP budget, we plot the final loss (after smoothing) against the parameter count in Figure 3 (left). In all cases, we ensure that we have trained a diverse enough set of model sizes to see a clear minimum in the loss. We fit a parabola to each IsoFLOPs curve to directly estimate at what model size the minimum loss is achieved (Figure 3 (left)). As with the previous approach, we then fit a power law between FLOPs and loss-optimal model size and number of training tokens, shown in Figure 3 (center, right). Again, we fit exponents of the form $N_{opt} \propto C^a$ and $D_{opt} \propto C^b$ and we find that $a = 0.49$ and $b = 0.51$—as summarized in Table 2.

3.3. Approach 3: Fitting a parametric loss function

Lastly, we model all final losses from experiments in Approach 1 & 2 as a parametric function of model parameter count and the number of seen tokens. Following a classical risk decomposition (see Section D.2), we propose the following functional form

$$\hat{L}(N, D) \triangleq E + \frac{A}{N^\alpha} + \frac{B}{D^\beta}. \tag{2}$$

The first term captures the loss for an ideal generative process on the data distribution, and should correspond to the entropy of natural text. The second term captures the fact that a perfectly trained transformer with N parameters underperforms the ideal generative process. The final term captures the fact that the transformer is not trained to convergence, as we only make a finite number of optimisation steps, on a sample of the dataset distribution.

Model fitting. To estimate (A, B, E, α, β), we minimize the Huber loss (Huber, 1964) between the predicted and observed log loss using the L-BFGS algorithm (Nocedal, 1980):

$$\min_{A,B,E,\alpha,\beta} \sum_{\text{Runs } i} \text{Huber}_\delta\Big(\log \hat{L}(N_i, D_i) - \log L_i\Big) \tag{3}$$

We account for possible local minima by selecting the best fit from a grid of initialisations. The Huber loss ($\delta = 10^{-3}$) is robust to outliers, which we find important for good predictive performance over held-out data points. Section D.2 details the fitting procedure and the loss decomposition.

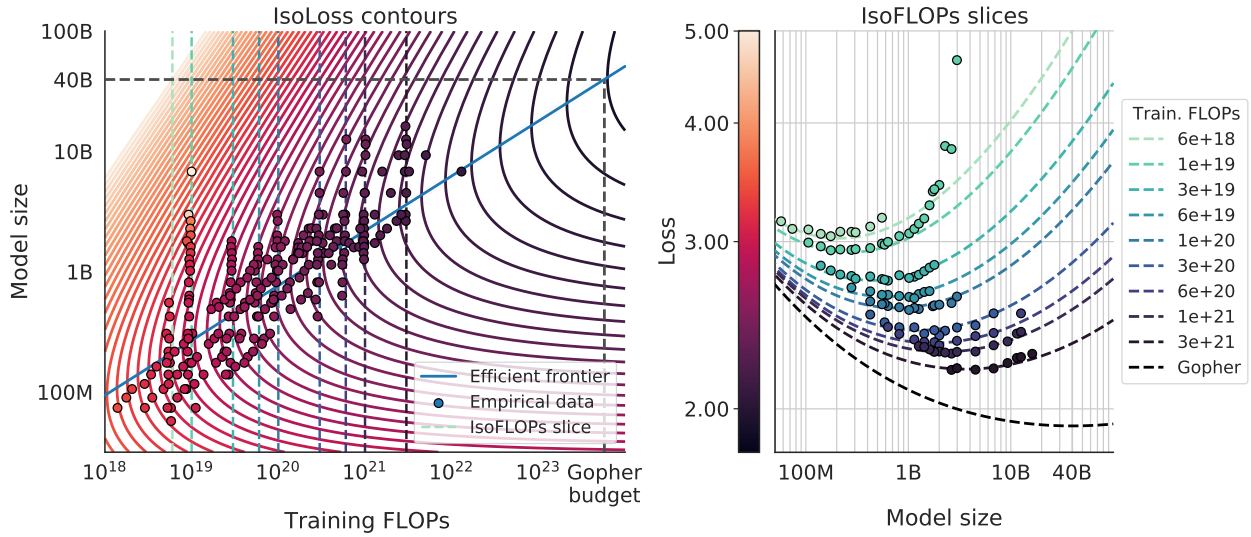


Figure 4 | **Parametric fit.** We fit a parametric modelling of the loss $\hat{L}(N, D)$ and display contour (**left**) and isoFLOP slices (**right**). For each isoFLOP slice, we include a corresponding dashed line in the left plot. In the left plot, we show the efficient frontier in blue, which is a line in log-log space. Specifically, the curve goes through each iso-loss contour at the point with the fewest FLOPs. We project the optimal model size given the *Gopher* FLOP budget to be 40B parameters.

Efficient frontier. We can approximate the functions N_{opt} and D_{opt} by minimizing the parametric loss $\hat{L}$ under the constraint FLOPs$(N, D) \approx 6ND$ (Kaplan et al., 2020). The resulting N_{opt} and D_{opt} balance the two terms in Equation (3) that depend on model size and data. By construction, they have a power-law form:

$$N_{opt}(C) = G\left(\frac{C}{6}\right)^{a}, \quad D_{opt}(C) = G^{-1}\left(\frac{C}{6}\right)^{b}, \quad \text{where} \quad G = \left(\frac{\alpha A}{\beta B}\right)^{\frac{1}{\alpha+\beta}}, \quad a = \frac{\beta}{\alpha+\beta}, \text{ and } b = \frac{\alpha}{\alpha+\beta}. \tag{4}$$

We show contours of the fitted function $\hat{L}$ in Figure 4 (left), and the closed-form efficient computational frontier in blue. From this approach, we find that $a = 0.46$ and $b = 0.54$—as summarized in Table 2.

3.4. Optimal model scaling

We find that the three approaches, despite using different fitting methodologies and different trained models, yield comparable predictions for the optimal scaling in parameters and tokens with FLOPs (shown in Table 2). All three approaches suggest that as compute budget increases, model size and the amount of training data should be increased in approximately equal proportions. The first and second approaches yield very similar predictions for optimal model sizes, as shown in Figure 1 and Figure A3. The third approach predicts even smaller models being optimal at larger compute budgets. We note that the observed points (L, N, D) for low training FLOPs ($C \leqslant 1e21$) have larger residuals $\|L - \hat{L}(N, D)\|_2^2$ than points with higher computational budgets. The fitted model places increased weight on the points with more FLOPs—automatically considering the low-computational budget points as outliers due to the Huber loss. As a consequence of the empirically observed negative curvature in the frontier $C \to N_{opt}$ (see Appendix E), this results in predicting a lower N_{opt} than the two other approaches.

In Table 3 we show the estimated number of FLOPs and tokens that would ensure that a model of a given size lies on the compute-optimal frontier. Our findings suggests that the current generation of

Table 2 | **Estimated parameter and data scaling with increased training compute.** The listed values are the exponents, a and b, on the relationship $N_{opt} \propto C^a$ and $D_{opt} \propto C^b$. Our analysis suggests a near equal scaling in parameters and data with increasing compute which is in clear contrast to previous work on the scaling of large models. The 10^{th} and 90^{th} percentiles are estimated via bootstrapping data (80% of the dataset is sampled 100 times) and are shown in parenthesis.

Approach	Coeff. a where $N_{opt} \propto C^a$	Coeff. b where $D_{opt} \propto C^b$
1. Minimum over training curves	0.50 (0.488, 0.502)	0.50 (0.501, 0.512)
2. IsoFLOP profiles	0.49 (0.462, 0.534)	0.51 (0.483, 0.529)
3. Parametric modelling of the loss	0.46 (0.454, 0.455)	0.54 (0.542, 0.543)
Kaplan et al. (2020)	0.73	0.27

Table 3 | **Estimated optimal training FLOPs and training tokens for various model sizes.** For various model sizes, we show the projections from Approach 1 of how many FLOPs and training tokens would be needed to train compute-optimal models. The estimates for Approach 2 & 3 are similar (shown in Section D.3)

Parameters	FLOPs	FLOPs (in *Gopher* unit)	Tokens
400 Million	1.92e+19	1/29,968	8.0 Billion
1 Billion	1.21e+20	1/4,761	20.2 Billion
10 Billion	1.23e+22	1/46	205.1 Billion
67 Billion	5.76e+23	1	1.5 Trillion
175 Billion	3.85e+24	6.7	3.7 Trillion
280 Billion	9.90e+24	17.2	5.9 Trillion
520 Billion	3.43e+25	59.5	11.0 Trillion
1 Trillion	1.27e+26	221.3	21.2 Trillion
10 Trillion	1.30e+28	22515.9	216.2 Trillion

large language models are considerably over-sized, given their respective compute budgets, as shown in Figure 1. For example, we find that a 175 billion parameter model should be trained with a compute budget of 4.41×10^{24} FLOPs and on over 4.2 trillion tokens. A 280 billion *Gopher*-like model is the optimal model to train given a compute budget of approximately 10^{25} FLOPs and should be trained on 6.8 trillion tokens. Unless one has a compute budget of 10^{26} FLOPs (over 250× the compute used to train *Gopher*), a 1 trillion parameter model is unlikely to be the optimal model to train. Furthermore, the amount of training data that is projected to be needed is far beyond what is currently used to train large models, and underscores the importance of dataset collection in addition to engineering improvements that allow for model scale. While there is significant uncertainty extrapolating out many orders of magnitude, our analysis clearly suggests that given the training compute budget for many current LLMs, smaller models should have been trained on more tokens to achieve the most performant model.

In Appendix C, we reproduce the IsoFLOP analysis on two additional datasets: C4 (Raffel et al., 2020a) and GitHub code (Rae et al., 2021). In both cases we reach the similar conclusion that model size and number of training tokens should be scaled in equal proportions.

4. *Chinchilla*

Based on our analysis in Section 3, the optimal model size for the *Gopher* compute budget is somewhere between 40 and 70 billion parameters. We test this hypothesis by training a model on the larger end of this range—70B parameters—for 1.4T tokens, due to both dataset and computational efficiency considerations. In this section we compare this model, which we call *Chinchilla*, to *Gopher* and other LLMs. Both *Chinchilla* and *Gopher* have been trained for the same number of FLOPs but differ in the size of the model and the number of training tokens.

While pre-training a large language model has a considerable compute cost, downstream fine-tuning and inference also make up substantial compute usage (Rae et al., 2021). Due to being 4× smaller than *Gopher*, both the memory footprint and inference cost of *Chinchilla* are also smaller.

4.1. Model and training details

The full set of hyperparameters used to train *Chinchilla* are given in Table 4. *Chinchilla* uses the same model architecture and training setup as *Gopher* with the exception of the differences listed below.

- We train *Chinchilla* on *MassiveText* (the same dataset as *Gopher*) but use a slightly different subset distribution (shown in Table A1) to account for the increased number of training tokens.
- We use AdamW (Loshchilov and Hutter, 2019) for *Chinchilla* rather than Adam (Kingma and Ba, 2014) as this improves the language modelling loss and the downstream task performance after finetuning.[8]
- We train *Chinchilla* with a slightly modified SentencePiece (Kudo and Richardson, 2018) tokenizer that does not apply NFKC normalisation. The vocabulary is very similar– 94.15% of tokens are the same as those used for training *Gopher*. We find that this particularly helps with the representation of mathematics and chemistry, for example.
- Whilst the forward and backward pass are computed in `bfloat16`, we store a `float32` copy of the weights in the distributed optimiser state (Rajbhandari et al., 2020). See *Lessons Learned* from Rae et al. (2021) for additional details.

In Appendix G we show the impact of the various optimiser related changes between *Chinchilla* and *Gopher*. All models in this analysis have been trained on TPUv3/TPUv4 (Jouppi et al., 2017) with JAX (Bradbury et al., 2018) and Haiku (Hennigan et al., 2020). We include a *Chinchilla* model card (Mitchell et al., 2019) in Table A8.

Model	Layers	Number Heads	Key/Value Size	d_{model}	Max LR	Batch Size
Gopher 280B	80	128	128	16,384	4×10^{-5}	3M → 6M
Chinchilla 70B	80	64	128	8,192	1×10^{-4}	1.5M → 3M

Table 4 | ***Chinchilla* architecture details.** We list the number of layers, the key/value size, the bottleneck activation size d_{model}, the maximum learning rate, and the training batch size (# tokens). The feed-forward size is always set to $4 \times d_{\text{model}}$. Note that we double the batch size midway through training for both *Chinchilla* and *Gopher*.

[8] Interestingly, a model trained with AdamW only passes the training performance of a model trained with Adam around 80% of the way through the cosine cycle, though the ending performance is notably better– see Figure A7

	# Tasks	Examples
Language Modelling	20	WikiText-103, The Pile: PG-19, arXiv, FreeLaw, ...
Reading Comprehension	3	RACE-m, RACE-h, LAMBADA
Question Answering	3	Natural Questions, TriviaQA, TruthfulQA
Common Sense	5	HellaSwag, Winogrande, PIQA, SIQA, BoolQ
MMLU	57	High School Chemistry, Astronomy, Clinical Knowledge, ...
BIG-bench	62	Causal Judgement, Epistemic Reasoning, Temporal Sequences, ...

Table 5 | **All evaluation tasks.** We evaluate *Chinchilla* on a collection of language modelling along with downstream tasks. We evaluate on largely the same tasks as in Rae et al. (2021), to allow for direct comparison.

4.2. Results

We perform an extensive evaluation of *Chinchilla*, comparing against various large language models. We evaluate on a large subset of the tasks presented in Rae et al. (2021), shown in Table 5. As the focus of this work is on optimal model scaling, we included a large representative subset, and introduce a few new evaluations to allow for better comparison to other existing large models. The evaluation details for all tasks are the same as described in Rae et al. (2021).

4.2.1. Language modelling

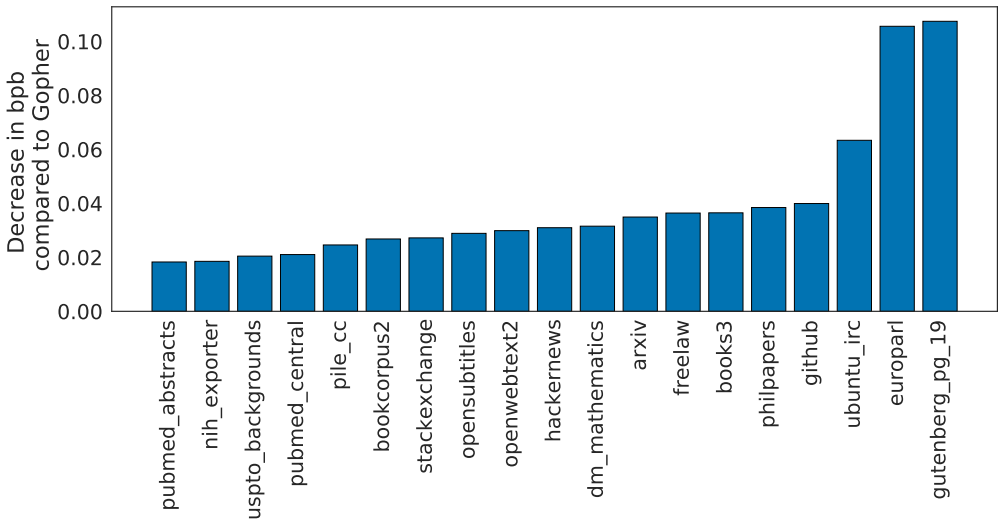


Figure 5 | **Pile Evaluation.** For the different evaluation sets in The Pile (Gao et al., 2020), we show the bits-per-byte (bpb) improvement (decrease) of *Chinchilla* compared to *Gopher*. On all subsets, *Chinchilla* outperforms *Gopher*.

Chinchilla significantly outperforms *Gopher* on all evaluation subsets of The Pile (Gao et al., 2020), as shown in Figure 5. Compared to Jurassic-1 (178B) Lieber et al. (2021), *Chinchilla* is more performant on all but two subsets– `dm_mathematics` and `ubuntu_irc`– see Table A5 for a raw bits-per-byte comparison. On Wikitext103 (Merity et al., 2017), *Chinchilla* achieves a perplexity of 7.16 compared to 7.75 for *Gopher*. Some caution is needed when comparing *Chinchilla* with *Gopher* on these language modelling benchmarks as *Chinchilla* is trained on 4× more data than *Gopher* and thus train/test set leakage may artificially enhance the results. We thus place more emphasis on other

Random	25.0%
Average human rater	34.5%
GPT-3 5-shot	43.9%
Gopher 5-shot	60.0%
***Chinchilla* 5-shot**	**67.6%**
Average human expert performance	*89.8%*
June 2022 Forecast	57.1%
June 2023 Forecast	63.4%

Table 6 | **Massive Multitask Language Understanding (MMLU).** We report the average 5-shot accuracy over 57 tasks with model and human accuracy comparisons taken from Hendrycks et al. (2020). We also include the average prediction for state of the art accuracy in June 2022/2023 made by 73 competitive human forecasters in Steinhardt (2021).

tasks for which leakage is less of a concern, such as MMLU (Hendrycks et al., 2020) and BIG-bench (BIG-bench collaboration, 2021) along with various closed-book question answering and common sense analyses.

4.2.2. MMLU

The Massive Multitask Language Understanding (MMLU) benchmark (Hendrycks et al., 2020) consists of a range of exam-like questions on academic subjects. In Table 6, we report *Chinchilla*'s average 5-shot performance on MMLU (the full breakdown of results is shown in Table A6). On this benchmark, *Chinchilla* significantly outperforms *Gopher* despite being much smaller, with an average accuracy of 67.6% (improving upon *Gopher* by 7.6%). Remarkably, *Chinchilla* even outperforms the expert forecast for June 2023 of 63.4% accuracy (see Table 6) (Steinhardt, 2021). Furthermore, *Chinchilla* achieves greater than 90% accuracy on 4 different individual tasks– `high_school_gov_and_politics`, `international_law`, `sociology`, and `us_foreign_policy`. To our knowledge, no other model has achieved greater than 90% accuracy on a subset.

In Figure 6, we show a comparison to *Gopher* broken down by task. Overall, we find that *Chinchilla* improves performance on the vast majority of tasks. On four tasks (`college_mathematics`, `econometrics`, `moral_scenarios`, and `formal_logic`) *Chinchilla* underperforms *Gopher*, and there is no change in performance on two tasks.

4.2.3. Reading comprehension

On the final word prediction dataset LAMBADA (Paperno et al., 2016), *Chinchilla* achieves 77.4% accuracy, compared to 74.5% accuracy from *Gopher* and 76.6% from MT-NLG 530B (see Table 7). On RACE-h and RACE-m (Lai et al., 2017), *Chinchilla* greatly outperforms *Gopher*, improving accuracy by more than 10% in both cases—see Table 7.

4.2.4. BIG-bench

We analysed *Chinchilla* on the same set of BIG-bench tasks (BIG-bench collaboration, 2021) reported in Rae et al. (2021). Similar to what we observed in MMLU, *Chinchilla* outperforms *Gopher* on the vast majority of tasks (see Figure 7). We find that *Chinchilla* improves the average performance by 10.7%, reaching an accuracy of 65.1% versus 54.4% for *Gopher*. Of the 62 tasks we consider, *Chinchilla* performs worse than *Gopher* on only four—`crash_blossom`, `dark_humor_detection`,

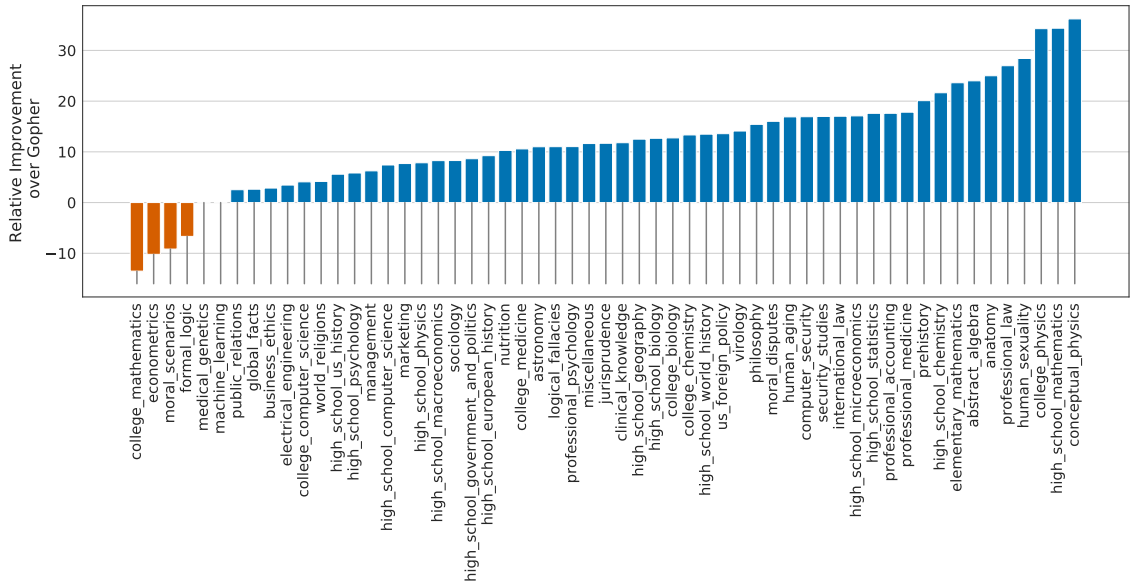


Figure 6 | **MMLU results compared to *Gopher*** We find that *Chinchilla* outperforms *Gopher* by 7.6% on average (see Table 6) in addition to performing better on 51/57 individual tasks, the same on 2/57, and worse on only 4/57 tasks.

	Chinchilla	*Gopher*	GPT-3	MT-NLG 530B
LAMBADA Zero-Shot	**77.4**	74.5	76.2	76.6
RACE-m Few-Shot	**86.8**	75.1	58.1	-
RACE-h Few-Shot	**82.3**	71.6	46.8	47.9

Table 7 | **Reading comprehension.** On RACE-h and RACE-m (Lai et al., 2017), *Chinchilla* considerably improves performance over *Gopher*. Note that GPT-3 and MT-NLG 530B use a different prompt format than we do on RACE-h/m, so results are not comparable to *Gopher* and *Chinchilla*. On LAMBADA (Paperno et al., 2016), *Chinchilla* outperforms both *Gopher* and MT-NLG 530B.

`mathematical_induction` and `logical_args`. Full accuracy results for *Chinchilla* can be found in Table A7.

4.2.5. Common sense

We evaluate *Chinchilla* on various common sense benchmarks: PIQA (Bisk et al., 2020), SIQA (Sap et al., 2019), Winogrande (Sakaguchi et al., 2020), HellaSwag (Zellers et al., 2019), and BoolQ (Clark et al., 2019). We find that *Chinchilla* outperforms both *Gopher* and GPT-3 on all tasks and outperforms MT-NLG 530B on all but one task—see Table 8.

On TruthfulQA (Lin et al., 2021), *Chinchilla* reaches 43.6%, 58.5%, and 66.7% accuracy with 0-shot, 5-shot, and 10-shot respectively. In comparison, *Gopher* achieved only 29.5% 0-shot and 43.7% 10-shot accuracy. In stark contrast with the findings of Lin et al. (2021), the large improvements (14.1% in 0-shot accuracy) achieved by Chinchilla suggest that better modelling of the pre-training data alone can lead to substantial improvements on this benchmark.

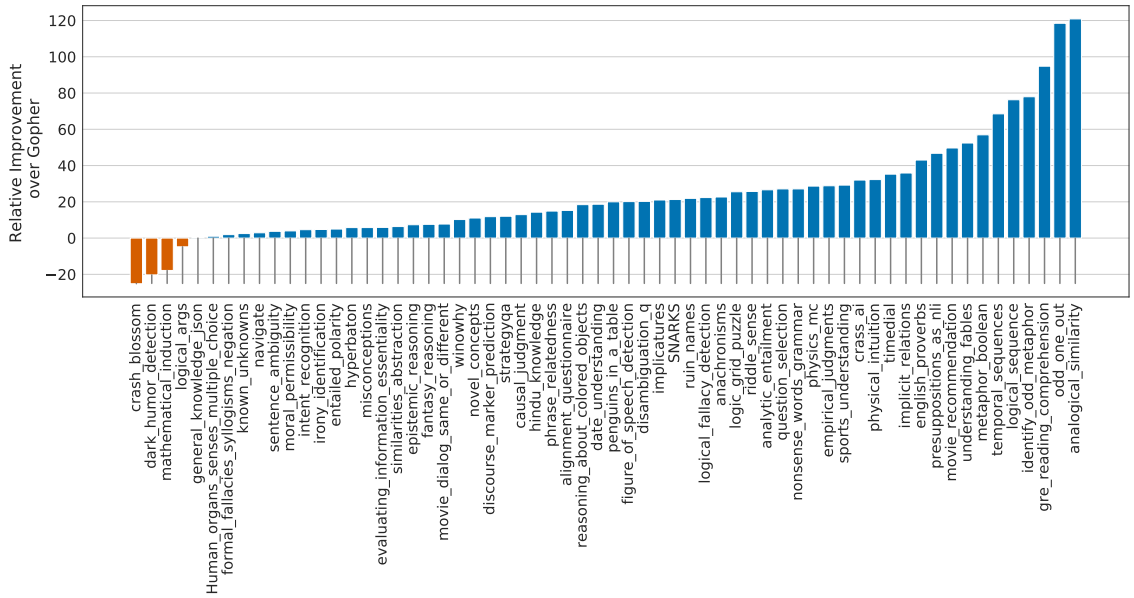


Figure 7 | **BIG-bench results compared to *Gopher*** *Chinchilla* out performs *Gopher* on all but four BIG-bench tasks considered. Full results are in Table A7.

4.2.6. Closed-book question answering

Results on closed-book question answering benchmarks are reported in Table 9. On the Natural Questions dataset (Kwiatkowski et al., 2019), *Chinchilla* achieves new closed-book SOTA accuracies: 31.5% 5-shot and 35.5% 64-shot, compared to 21% and 28% respectively, for *Gopher*. On TriviaQA (Joshi et al., 2017) we show results for both the filtered (previously used in retrieval and open-book work) and unfiltered set (previously used in large language model evaluations). In both cases, *Chinchilla* substantially out performs *Gopher*. On the filtered version, Chinchilla lags behind the open book SOTA (Izacard and Grave, 2020) by only 7.9%. On the unfiltered set, *Chinchilla* outperforms GPT-3—see Table 9.

4.2.7. Gender bias and toxicity

Large Language Models carry potential risks such as outputting offensive language, propagating social biases, and leaking private information (Bender et al., 2021; Weidinger et al., 2021). We expect *Chinchilla* to carry risks similar to *Gopher* because *Chinchilla* is trained on the same data,

	Chinchilla	*Gopher*	GPT-3	MT-NLG 530B	Supervised SOTA
HellaSWAG	**80.8%**	79.2%	78.9%	80.2%	93.9%
PIQA	81.8%	81.8%	81.0%	**82.0%**	90.1%
Winogrande	**74.9%**	70.1%	70.2%	73.0%	91.3%
SIQA	**51.3%**	50.6%	-	-	83.2%
BoolQ	**83.7%**	79.3%	60.5%	78.2%	91.4%

Table 8 | **Zero-shot comparison on Common Sense benchmarks.** We show a comparison between *Chinchilla*, *Gopher*, and MT-NLG 530B on various Common Sense benchmarks. We see that *Chinchilla* matches or outperforms *Gopher* and GPT-3 on all tasks. On all but one *Chinchilla* outperforms the much larger MT-NLG 530B model.

	Method	*Chinchilla*	*Gopher*	GPT-3	SOTA (open book)
Natural Questions (dev)	0-shot	16.6%	10.1%	14.6%	54.4%
	5-shot	31.5%	24.5%	-	
	64-shot	35.5%	28.2%	29.9%	
TriviaQA (unfiltered, test)	0-shot	67.0%	52.8%	64.3 %	-
	5-shot	73.2%	63.6%	-	
	64-shot	72.3%	61.3%	71.2%	
TriviaQA (filtered, dev)	0-shot	55.4%	43.5%	-	72.5%
	5-shot	64.1%	57.0%	-	
	64-shot	64.6%	57.2%	-	

Table 9 | **Closed-book question answering.** For Natural Questions (Kwiatkowski et al., 2019) and TriviaQA (Joshi et al., 2017), *Chinchilla* outperforms *Gopher* in all cases. On Natural Questions, *Chinchilla* outperforms GPT-3. On TriviaQA we show results on two different evaluation sets to allow for comparison to GPT-3 and to open book SOTA (FiD + Distillation (Izacard and Grave, 2020)).

albeit with slightly different relative weights, and because it has a similar architecture. Here, we examine gender bias (particularly gender and occupation bias) and generation of toxic language. We select a few common evaluations to highlight potential issues, but stress that our evaluations are not comprehensive and much work remains to understand, evaluate, and mitigate risks in LLMs.

Gender bias. As discussed in Rae et al. (2021), large language models reflect contemporary and historical discourse about different groups (such as gender groups) from their training dataset, and we expect the same to be true for *Chinchilla*. Here, we test if potential gender and occupation biases manifest in unfair outcomes on coreference resolutions, using the Winogender dataset (Rudinger et al., 2018) in a zero-shot setting. Winogender tests whether a model can correctly determine if a pronoun refers to different occupation words. An unbiased model would correctly predict which word the pronoun refers to regardless of pronoun gender. We follow the same setup as in Rae et al. (2021) (described further in Section H.3).

As shown in Table 10, *Chinchilla* correctly resolves pronouns more frequently than *Gopher* across all groups. Interestingly, the performance increase is considerably smaller for male pronouns (increase of 3.2%) than for female or neutral pronouns (increases of 8.3% and 9.2% respectively). We also consider *gotcha* examples, in which the correct pronoun resolution contradicts gender stereotypes (determined by labor statistics). Again, we see that *Chinchilla* resolves pronouns more accurately than *Gopher*. When breaking up examples by male/female gender and *gotcha/not gotcha*, the largest improvement is on female *gotcha* examples (improvement of 10%). Thus, though *Chinchilla* uniformly overcomes gender stereotypes for more coreference examples than *Gopher*, the rate of improvement is higher for some pronouns than others, suggesting that the improvements conferred by using a more compute-optimal model can be uneven.

Sample toxicity. Language models are capable of generating toxic language—including insults, hate speech, profanities and threats (Gehman et al., 2020; Rae et al., 2021). While toxicity is an umbrella term, and its evaluation in LMs comes with challenges (Welbl et al., 2021; Xu et al., 2021), automatic classifier scores can provide an indication for the levels of harmful text that a LM generates. Rae et al. (2021) found that improving language modelling loss by increasing the number of model parameters has only a negligible effect on toxic text generation (unprompted); here we analyze

	Chinchilla	*Gopher*
All	78.3%	71.4%
Male	71.2%	68.0%
Female	79.6%	71.3%
Neutral	84.2%	75.0%

	Chinchilla	*Gopher*
Male *gotcha*	62.5%	59.2%
Male *not gotcha*	80.0%	76.7%
Female *gotcha*	76.7%	66.7%
Female *not gotcha*	82.5%	75.8%

Table 10 | **Winogender results. Left:** *Chinchilla* consistently resolves pronouns better than *Gopher*. **Right:** *Chinchilla* performs better on examples which contradict gender stereotypes (*gotcha* examples). However, difference in performance across groups suggests *Chinchilla* exhibits bias.

whether the same holds true for a lower LM loss achieved via more compute-optimal training. Similar to the protocol of Rae et al. (2021), we generate 25,000 unprompted samples from *Chinchilla*, and compare their *PerspectiveAPI* toxicity score distribution to that of *Gopher*-generated samples. Several summary statistics indicate an absence of major differences: the mean (median) toxicity score for *Gopher* is 0.081 (0.064), compared to 0.087 (0.066) for *Chinchilla*, and the 95$^{\text{th}}$ percentile scores are 0.230 for *Gopher*, compared to 0.238 for *Chinchilla*. That is, the large majority of generated samples are classified as non-toxic, and the difference between the models is negligible. In line with prior findings (Rae et al., 2021), this suggests that toxicity levels in unconditional text generation are largely independent of the model quality (measured in language modelling loss), i.e. that better models of the training dataset are not necessarily more toxic.

5. Discussion & Conclusion

The trend so far in large language model training has been to increase the model size, often without increasing the number of training tokens. The largest dense transformer, MT-NLG 530B, is now over 3× larger than GPT-3's 170 billion parameters from just two years ago. However, this model, as well as the majority of existing large models, have all been trained for a comparable number of tokens—around 300 billion. While the desire to train these mega-models has led to substantial engineering innovation, we hypothesize that the race to train larger and larger models is resulting in models that are substantially underperforming compared to what could be achieved with the same compute budget.

We propose three predictive approaches towards optimally setting model size and training duration, based on the outcome of over 400 training runs. All three approaches predict that *Gopher* is substantially over-sized and estimate that for the same compute budget a smaller model trained on more data will perform better. We directly test this hypothesis by training *Chinchilla*, a 70B parameter model, and show that it outperforms *Gopher* and even larger models on nearly every measured evaluation task.

Whilst our method allows us to make predictions on how to scale large models when given additional compute, there are several limitations. Due to the cost of training large models, we only have two comparable training runs at large scale (*Chinchilla* and *Gopher*), and we do not have additional tests at intermediate scales. Furthermore, we assume that the efficient computational frontier can be described by a power-law relationship between the compute budget, model size, and number of training tokens. However, we observe some concavity in $\log(N_{opt})$ at high compute budgets (see Appendix E). This suggests that we may still be overestimating the optimal size of large models. Finally, the training runs for our analysis have all been trained on less than an epoch of data; future work may consider the multiple epoch regime. Despite these limitations, the comparison of *Chinchilla* to *Gopher* validates our performance predictions, that have thus enabled training a better (and more

lightweight) model at the same compute budget.

Though there has been significant recent work allowing larger and larger models to be trained, our analysis suggests an increased focus on dataset scaling is needed. Speculatively, we expect that scaling to larger and larger datasets is only beneficial when the data is high-quality. This calls for responsibly collecting larger datasets with a high focus on dataset quality. Larger datasets will require extra care to ensure train-test set overlap is properly accounted for, both in the language modelling loss but also with downstream tasks. Finally, training for trillions of tokens introduces many ethical and privacy concerns. Large datasets scraped from the web will contain toxic language, biases, and private information. With even larger datasets being used, the quantity (if not the frequency) of such information increases, which makes dataset introspection all the more important. *Chinchilla* does suffer from bias and toxicity but interestingly it seems less affected than *Gopher*. Better understanding how performance of large language models and toxicity interact is an important future research question.

While we have applied our methodology towards the training of auto-regressive language models, we expect that there is a similar trade-off between model size and the amount of data in other modalities. As training large models is very expensive, choosing the optimal model size and training steps beforehand is essential. The methods we propose are easy to reproduce in new settings.

6. Acknowledgements

We'd like to thank Jean-baptiste Alayrac, Kareem Ayoub, Chris Dyer, Nando de Freitas, Demis Hassabis, Geoffrey Irving, Koray Kavukcuoglu, Nate Kushman and Angeliki Lazaridou for useful comments on the manuscript. We'd like to thank Andy Brock, Irina Higgins, Michela Paganini, Francis Song, and other colleagues at DeepMind for helpful discussions. We are also very grateful to the JAX and XLA team for their support and assistance.

References

M. Artetxe, S. Bhosale, N. Goyal, T. Mihaylov, M. Ott, S. Shleifer, X. V. Lin, J. Du, S. Iyer, R. Pasunuru, G. Anantharaman, X. Li, S. Chen, H. Akin, M. Baines, L. Martin, X. Zhou, P. S. Koura, B. O'Horo, J. Wang, L. Zettlemoyer, M. Diab, Z. Kozareva, and V. Stoyanov. Efficient Large Scale Language Modeling with Mixtures of Experts. arXiv:2112.10684, 2021.

E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency, pages 610–623, 2021.

BIG-bench collaboration. Beyond the imitation game: Measuring and extrapolating the capabilities of language models. In preparation, 2021. URL https://github.com/google/BIG-bench/.

Y. Bisk, R. Zellers, J. Gao, Y. Choi, et al. PIQA: Reasoning about physical commonsense in natural language. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, pages 7432–7439, 2020.

S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. van den Driessche, J.-B. Lespiau, B. Damoc, A. Clark, D. de Las Casas, A. Guy, J. Menick, R. Ring, T. Hennigan, S. Huang, L. Maggiore, C. Jones, A. Cassirer, A. Brock, M. Paganini, G. Irving, O. Vinyals, S. Osindero, K. Simonyan, J. W. Rae, E. Elsen, and L. Sifre. Improving language models by retrieving from trillions of tokens. arXiv 2112.04426, 2021.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs. 2018. URL http://github.com/google/jax.

T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, Advances in Neural Information Processing Systems, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

S. Bubeck. Convex Optimization: Algorithms and Complexity. Foundations and Trends in Machine Learning, 8(3-4):231–357, 2015. URL http://www.nowpublishers.com/article/Details/MAL-050.

A. Clark, D. d. l. Casas, A. Guy, A. Mensch, M. Paganini, J. Hoffmann, B. Damoc, B. Hechtman, T. Cai, S. Borgeaud, G. v. d. Driessche, E. Rutherford, T. Hennigan, M. Johnson, K. Millican, A. Cassirer, C. Jones, E. Buchatskaya, D. Budden, L. Sifre, S. Osindero, O. Vinyals, J. Rae, E. Elsen, K. Kavukcuoglu, and K. Simonyan. Unified scaling laws for routed language models, 2022. URL https://arxiv.org/abs/2202.01169.

C. Clark, K. Lee, M.-W. Chang, T. Kwiatkowski, M. Collins, and K. Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. In 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), pages 2924–2936, 2019.

N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, B. Zoph, L. Fedus, M. Bosma, Z. Zhou, T. Wang, Y. E. Wang, K. Webster, M. Pellat, K. Robinson, K. Meier-Hellstern, T. Duke, L. Dixon, K. Zhang, Q. V. Le, Y. Wu, Z. Chen, and C. Cui. Glam: Efficient scaling of language models with mixture-of-experts, 2021. URL https://arxiv.org/abs/2112.06905.

W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. arXiv preprint arXiv:2101.03961, 2021.

L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy. The Pile: An 800GB dataset of diverse text for language modeling. arXiv preprint arXiv:2101.00027, 2020.

S. Gehman, S. Gururangan, M. Sap, Y. Choi, and N. A. Smith. RealToxicityPrompts: Evaluating neural toxic degeneration in language models. In Findings of the Association for Computational Linguistics: EMNLP 2020, pages 3356–3369, Online, Nov. 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.301. URL https://aclanthology.org/2020.findings-emnlp.301.

K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang. REALM: Retrieval-augmented language model pre-training, 2020.

D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt. Measuring massive multitask language understanding. arXiv preprint arXiv:2009.03300, 2020.

T. Hennigan, T. Cai, T. Norman, and I. Babuschkin. Haiku: Sonnet for JAX. 2020. URL http://github.com/deepmind/dm-haiku.

D. Hernandez, J. Kaplan, T. Henighan, and S. McCandlish. Scaling laws for transfer, 2021.

P. J. Huber. Robust Estimation of a Location Parameter. The Annals of Mathematical Statistics, 35 (1):73–101, Mar. 1964. ISSN 0003-4851, 2168-8990. doi: 10.1214/aoms/1177703732. URL https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-35/issue-1/Robust-Estimation-of-a-Location-Parameter/10.1214/aoms/1177703732.full.

G. Izacard and E. Grave. Distilling knowledge from reader to retriever for question answering, 2020.

M. Joshi, E. Choi, D. Weld, and L. Zettlemoyer. TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension. arXiv e-prints, art. arXiv:1705.03551, 2017.

N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348928. doi: 10.1145/3079856.3080246. URL https://doi.org/10.1145/3079856.3080246.

J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. arXiv preprint arXiv:2001.08361, 2020.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

T. Kudo and J. Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. arXiv preprint arXiv:1808.06226, 2018.

T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, M. Kelcey, J. Devlin, K. Lee, K. N. Toutanova, L. Jones, M.-W. Chang, A. Dai, J. Uszkoreit, Q. Le, and S. Petrov. Natural questions: a benchmark for question answering research. Transactions of the Association of Computational Linguistics, 2019.

G. Lai, Q. Xie, H. Liu, Y. Yang, and E. Hovy. RACE: Large-scale ReAding comprehension dataset from examinations. In Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, pages 785–794, Copenhagen, Denmark, Sept. 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1082. URL https://aclanthology.org/D17-1082.

Y. Levine, N. Wies, O. Sharir, H. Bata, and A. Shashua. The depth-to-width interplay in self-attention. arXiv preprint arXiv:2006.12467, 2020.

P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In Advances in Neural Information Processing Systems, volume 33, pages 9459–9474, 2020.

O. Lieber, O. Sharir, B. Lenz, and Y. Shoham. Jurassic-1: Technical details and evaluation. White Paper. AI21 Labs, 2021.

S. Lin, J. Hilton, and O. Evans. TruthfulQA: Measuring how models mimic human falsehoods. arXiv preprint arXiv:2109.07958, 2021.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In International Conference on Learning Representations, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team. An empirical model of large-batch training, 2018.

S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models. International Conference on Learning Representations, 2017.

M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, and T. Gebru. Model cards for model reporting. In Proceedings of the conference on fairness, accountability, and transparency, pages 220–229, 2019.

J. Nocedal. Updating Quasi-Newton Matrices with Limited Storage. Mathematics of Computation, 35(151):773–782, 1980. ISSN 0025-5718. doi: 10.2307/2006193. URL https://www.jstor.org/stable/2006193.

D. Paperno, G. Kruszewski, A. Lazaridou, Q. N. Pham, R. Bernardi, S. Pezzelle, M. Baroni, G. Boleda, and R. Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context, 2016.

J. Rae, S. Borgeaud, T. Cai, K. Millican, J. Hoffmann, F. Song, J. Aslanides, S. Henderson, R. Ring, S. Young, E. Rutherford, T. Hennigan, J. Menick, A. Cassirer, R. Powell, G. van den Driessche, L. A. Hendricks, M. Rauh, P.-S. Huang, A. Glaese, J. Welbl, S. Dathathri, S. Huang, J. Uesato, J. Mellor, I. Higgins, A. Creswell, N. McAleese, A. Wu, E. Elsen, S. Jayakumar, E. Buchatskaya, D. Budden, E. Sutherland, K. Simonyan, M. Paganini, L. Sifre, L. Martens, X. L. Li, A. Kuncoro, A. Nematzadeh, E. Gribovskaya, D. Donato, A. Lazaridou, A. Mensch, J.-B. Lespiau, M. Tsimpoukelli, N. Grigorev, D. Fritz, T. Sottiaux, M. Pajarskas, T. Pohlen, Z. Gong, D. Toyama, C. de Masson d'Autume, Y. Li, T. Terzi, I. Babuschkin, A. Clark, D. de Las Casas, A. Guy, J. Bradbury, M. Johnson, L. Weidinger, I. Gabriel, W. Isaac, E. Lockhart, S. Osindero, L. Rimell, C. Dyer, O. Vinyals, K. Ayoub, J. Stanway, L. Bennett, D. Hassabis, K. Kavukcuoglu, and G. Irving. Scaling language models: Methods, analysis & insights from training Gopher. arXiv 2112.11446, 2021.

J. W. Rae, A. Potapenko, S. M. Jayakumar, T. P. Lillicrap, K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, et al. Compressive transformers for long-range sequence modelling. Advances in Neural Information Processing Systems, 33:6154–6158, 2020.

C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21(140):1–67, 2020a. URL http://jmlr.org/papers/v21/20-074.html.

C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21(140):1–67, 2020b.

S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–16. IEEE, 2020.

H. Robbins and S. Monro. A Stochastic Approximation Method. The Annals of Mathematical Statistics, 22(3):400–407, Sept. 1951.

R. Rudinger, J. Naradowsky, B. Leonard, and B. Van Durme. Gender bias in coreference resolution. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

K. Sakaguchi, R. Le Bras, C. Bhagavatula, and Y. Choi. Winogrande: An adversarial winograd schema challenge at scale. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, pages 8732–8740, 2020.

M. Sap, H. Rashkin, D. Chen, R. LeBras, and Y. Choi. SocialIQA: Commonsense reasoning about social interactions. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing, 2019.

C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. Measuring the effects of data parallelism on neural network training. arXiv preprint arXiv:1811.03600, 2018.

J. W. Siegel and J. Xu. Approximation rates for neural networks with general activation functions. Neural Networks, 128:313–321, Aug. 2020. URL https://www.sciencedirect.com/science/article/pii/S0893608020301891.

S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro. Using Deepspeed and Megatron to Train Megatron-turing NLG 530b, A Large-Scale Generative Language Model. arXiv preprint arXiv:2201.11990, 2022.

J. Steinhardt. Updates and lessons from AI forecasting, 2021. URL https://bounded-regret.ghost.io/ai-forecasting/.

Y. Tay, M. Dehghani, J. Rao, W. Fedus, S. Abnar, H. W. Chung, S. Narang, D. Yogatama, A. Vaswani, and D. Metzler. Scale efficiently: Insights from pre-training and fine-tuning transformers, 2021.

R. Thoppilan, D. D. Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, Y. Zhou, C.-C. Chang, I. Krivokon, W. Rusch, M. Pickett, K. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. Aguera-Arcas, C. Cui, M. Croak, E. Chi, and Q. Le. LaMDA: Language models for dialog applications, 2022.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008, 2017.

L. Weidinger, J. Mellor, M. Rauh, C. Griffin, J. Uesato, P.-S. Huang, M. Cheng, M. Glaese, B. Balle, A. Kasirzadeh, Z. Kenton, S. Brown, W. Hawkins, T. Stepleton, C. Biles, A. Birhane, J. Haas, L. Rimell, L. A. Hendricks, W. Isaac, S. Legassick, G. Irving, and I. Gabriel. Ethical and social risks of harm from language models. arXiv submission, 2021.

J. Welbl, A. Glaese, J. Uesato, S. Dathathri, J. Mellor, L. A. Hendricks, K. Anderson, P. Kohli, B. Coppin, and P.-S. Huang. Challenges in detoxifying language models. In Findings of the Association for Computational Linguistics: EMNLP 2021, pages 2447–2469, Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics. URL https://aclanthology.org/2021.findings-emnlp.210.

A. Xu, E. Pathak, E. Wallace, S. Gururangan, M. Sap, and D. Klein. Detoxifying language models risks marginalizing minority voices. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 2390–2397, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.190. URL https://aclanthology.org/2021.naacl-main.190.

G. Yang, E. J. Hu, I. Babuschkin, S. Sidor, X. Liu, D. Farhi, N. Ryder, J. Pachocki, W. Chen, and J. Gao. Tuning large neural networks via zero-shot hyperparameter transfer. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, Advances in Neural Information Processing Systems, 2021. URL https://openreview.net/forum?id=Bx6qKuBM2AD.

R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. HellaSwag: Can a machine really finish your sentence? In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019.

G. Zhang, L. Li, Z. Nado, J. Martens, S. Sachdeva, G. Dahl, C. Shallue, and R. B. Grosse. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/e0eacd983971634327ae1819ea8b6214-Paper.pdf.

B. Zoph, I. Bello, S. Kumar, N. Du, Y. Huang, J. Dean, N. Shazeer, and W. Fedus. Designing effective sparse expert models, 2022.

Appendix

A. Training dataset

In Table A1 we show the training dataset makeup used for *Chinchilla* and all scaling runs. Note that both the *MassiveWeb* and Wikipedia subsets are both used for more than one epoch.

	Disk Size	Documents	Sampling proportion	Epochs in 1.4T tokens
MassiveWeb	1.9 TB	604M	45% (48%)	1.24
Books	2.1 TB	4M	30% (27%)	0.75
C4	0.75 TB	361M	10% (10%)	0.77
News	2.7 TB	1.1B	10% (10%)	0.21
GitHub	3.1 TB	142M	4% (3%)	0.13
Wikipedia	0.001 TB	6M	1% (2%)	3.40

Table A1 | ***MassiveText* data makeup.** For each subset of *MassiveText*, we list its total disk size, the number of documents and the sampling proportion used during training—we use a slightly different distribution than in Rae et al. (2021) (shown in parenthesis). In the rightmost column show the number of epochs that are used in 1.4 trillion tokens.

B. Optimal cosine cycle length

One key assumption is made on the cosine cycle length and the corresponding learning rate drop (we use a 10× learning rate decay in line with Rae et al. (2021)).[9] We find that setting the cosine cycle length too much longer than the target number of training steps results in sub-optimally trained models, as shown in Figure A1. As a result, we assume that an optimally trained model will have the cosine cycle length correctly calibrated to the maximum number of steps, given the FLOP budget; we follow this rule in our main analysis.

C. Consistency of scaling results across datasets

We show scaling results from an IsoFLOP (Approach 2) analysis after training on two different datasets: C4 (Raffel et al., 2020b) and GitHub code (we show results with data from Rae et al. (2021)), results are shown in Table A2. For both set of experiments using subsets of *MassiveText*, we use the same tokenizer as the *MassiveText* experiments.

We find that the scaling behaviour on these datasets is very similar to what we found on *MassiveText*, as shown in Figure A2 and Table A2. This suggests that our results are independent of the dataset as long as one does not train for more than one epoch.

[9] We find the difference between decaying by 10× and decaying to 0.0 (over the same number of steps) to be small, though decaying by a factor of 10× to be slightly more performant. Decaying by less (5×) is clearly worse.

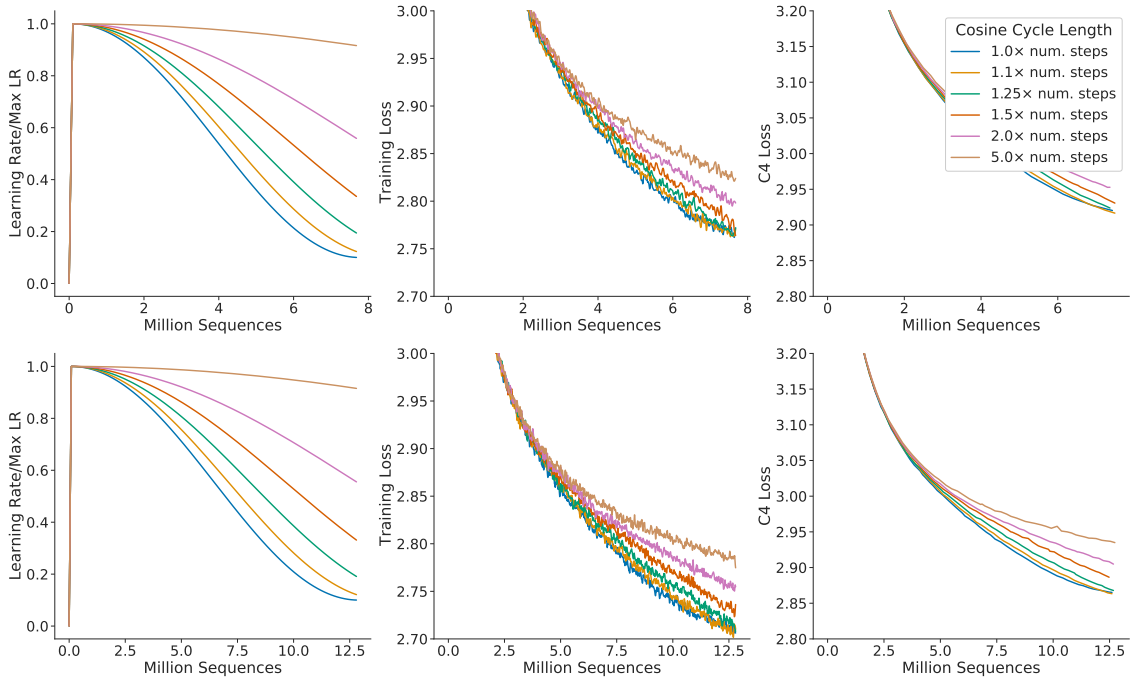


Figure A1 | **Grid over cosine cycle length.** We show 6 curves with the cosine cycle length set to 1, 1.1, 1.25, 1.5, 2, and 5× longer than the target number of training steps. When the cosine cycle length is too long, and the learning rate does not drop appropriately, then performance is impaired. We find that overestimating the number of training steps beyond 25% leads to clear drops in performance. We show results where we have set the number of training steps to two different values (top and bottom).

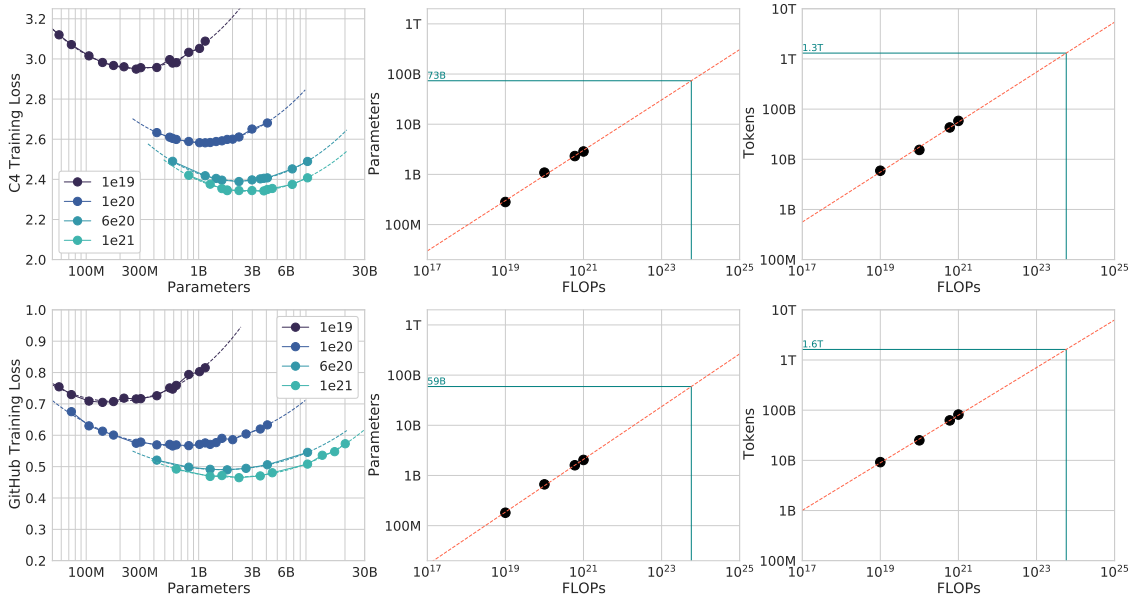


Figure A2 | **C4 and GitHub IsoFLOP curves.** Using the C4 dataset (Raffel et al., 2020b) and a GitHub dataset (Rae et al., 2021), we generate 4 IsoFLOP profiles and show the parameter and token count scaling, as in Figure 3. Scaling coefficients are shown in Table A2.

Approach	Coef. a where $N_{opt} \propto C^a$	Coef. b where $D_{opt} \propto C^b$
C4	0.50	0.50
GitHub	0.53	0.47
Kaplan et al. (2020)	0.73	0.27

Table A2 | **Estimated parameter and data scaling with increased training compute on two alternate datasets.** The listed values are the exponents, a and b, on the relationship $N_{opt} \propto C^a$ and $D_{opt} \propto C^b$. Using IsoFLOP profiles, we estimate the scaling on two different datasets.

D. Details on the scaling analyses

D.1. Approach 1: Fixing model sizes and varying training sequences

We use a maximum learning rate of 2×10^{-4} for the smallest models and 1.25×10^{-4} for the largest models. In all cases, the learning rate drops by a factor of 10× during training, using a cosine schedule. We make the assumption that the cosine cycle length should be approximately matched to the number of training steps. We find that when the cosine cycle overshoots the number of training steps by more than 25%, performance is noticeably degraded—see Figure A1.[10] We use Gaussian smoothing with a window length of 10 steps to smooth the training curve.

D.2. Approach 3: Parametric fitting of the loss

In this section, we first show how Equation (2) can be derived. We repeat the equation below for clarity,

$$\hat{L}(N, D) \triangleq E + \frac{A}{N^\alpha} + \frac{B}{D^\beta}, \tag{5}$$

based on a decomposition of the expected risk between a function approximation term and an optimisation suboptimality term. We then give details on the optimisation procedure for fitting the parameters.

Loss decomposition. Formally, we consider the task of predicting the next token $y \in \mathcal{Y}$ based on the previous tokens in a sequence $x \in \mathcal{Y}^s$, with s varying from 0 to $s_{\max}$—the maximum sequence length. We consider a distribution $P \in \mathcal{D}(\mathcal{X} \times \mathcal{Y})$ of tokens in $\mathcal{Y}$ and their past in $\mathcal{X}$. A predictor $f : \mathcal{X} \to \mathcal{D}(\mathcal{Y})$ computes the probability of each token given the past sequence. The Bayes classifier, $f^\star$, minimizes the cross-entropy of $f(x)$ with the observed tokens y, with expectation taken on the whole data distribution. We let L be the expected risk

$$L(f) \triangleq \mathbb{E}[\log f(x)_y], \qquad \text{and set} \qquad f^\star \triangleq \operatorname*{argmin}_{f \in \mathcal{F}(\mathcal{X}, \mathcal{D}(\mathcal{Y}))} L(f). \tag{6}$$

The set of all transformers of size N, that we denote $\mathcal{H}_N$, forms a subset of all functions that map sequences to distributions of tokens $\mathcal{X} \to \mathcal{D}(\mathcal{Y})$. Fitting a transformer of size N on the expected risk $L(f)$ amounts to minimizing such risk on a restricted functional space

$$f_N \triangleq \operatorname*{argmin}_{f \in \mathcal{H}_N} L(f). \tag{7}$$

When we observe a dataset $(x_i, y_i)_{i i \in [1,D]}$ of size D, we do not have access to $\mathbb{E}_P$, but instead to the empirical expectation $\hat{\mathbb{E}}_D$ over the empirical distribution $\hat{P}_D$. What happens when we are given D

[10] This further emphasises the point of not only determining model size, but also training length before training begins.

datapoints that we can only see once, and when we constrain the size of the hypothesis space to be N-dimensional ? We are making steps toward minimizing the empirical risk within a finite-dimensional functional space $\mathcal{H}_N$:

$$\hat{L}_D(f) \triangleq \hat{\mathbb{E}}_D[\log f(x)_y], \qquad \text{setting} \qquad \hat{f}_{N,D} \triangleq \operatorname*{argmin}_{f \in \mathcal{H}_N} \hat{L}_D(f). \tag{8}$$

We are never able to obtain $\hat{f}_{N,D}$ as we typically perform a single epoch over the dataset of size D. Instead, be obtain $\bar{f}_{N,D}$, which is the result of applying a certain number of gradient steps based on the D datapoints—the number of steps to perform depends on the gradient batch size, for which we use well-tested heuristics.

Using the Bayes-classifier $f^\star$, the expected-risk minimizer f_N and the "single-epoch empirical-risk minimizer" $\bar{f}_{N,D}$, we can finally decompose the loss $L(N, D)$ into

$$L(N, D) \triangleq L(\bar{f}_{N,D}) = L(f^\star) + \big(L(f_N) - L(f^\star)\big) + \big(L(\bar{f}_{N,D}) - L(f_N)\big)\,. \tag{9}$$

The loss comprises three terms: the Bayes risk, i.e. the minimal loss achievable for next-token prediction on the full distribution P, a.k.a the "entropy of natural text."; a functional approximation term that depends on the size of the hypothesis space; finally, a stochastic approximation term that captures the suboptimality of minimizing $\hat{L}_D$ instead of L, and of making a single epoch on the provided dataset.

Expected forms of the loss terms. In the decomposition (9), the second term depends entirely on the number of parameters N that defines the size of the functional approximation space. *On the set of two-layer neural networks*, it is expected to be proportional to $\frac{1}{N^{1/2}}$ (Siegel and Xu, 2020). Finally, given that it corresponds to early stopping in stochastic first order methods, the third term should scale as the convergence rate of these methods, which is lower-bounded by $\frac{1}{D^{1/2}}$ (Robbins and Monro, 1951) (and may attain the bound). This convergence rate is expected to be dimension free (see e.g. Bubeck, 2015, for a review) and depends only on the loss smoothness; hence we assume that the second term only depends on D in (2). Empirically, we find after fitting (2) that

$$L(N, D) = E + \frac{A}{N^{0.34}} + \frac{B}{D^{0.28}}, \tag{10}$$

with $E = 1.69$, $A = 406.4$, $B = 410.7$. We note that the parameter/data coefficients are both lower than $\frac{1}{2}$; this is expected for the data-efficiency coefficient (but far from the known lower-bound). Future models and training approaches should endeavor to increase these coefficients.

Fitting the decomposition to data. We effectively minimize the following problem

$$\min_{a,b,e,\alpha,\beta} \sum_{\text{Run } i} \text{Huber}_\delta\Big(\text{LSE}\big(a - \alpha \log N_i, b - \beta \log D_i, e\big) - \log L_i\Big), \tag{11}$$

where *LSE* is the log-sum-exp operator. We then set $A, B, E = \exp(a), \exp(b), \exp(e)$.

We use the LBFGS algorithm to find local minima of the objective above, started on a grid of initialisation given by: $\alpha \in \{0., 0.5, \ldots, 2.\}$, $\beta \in \{0., 0.5, \ldots, 2.\}$, $e \in \{-1., -.5, \ldots, 1.\}$, $a \in \{0, 5, \ldots, 25\}$, and $b \in \{0, 5, \ldots, 25\}$. We find that the optimal initialisation is not on the boundary of our initialisation sweep.

We use $\delta = 10^{-3}$ for the Huber loss. We find that using larger values of δ pushes the model to overfit the small compute regime and poorly predict held-out data from larger runs. We find that using a δ smaller than 10^{-3} does not impact the resulting predictions.

D.3. Predicted compute optimal frontier for all three methods

For Approaches 2 and 3, we show the estimated model size and number of training tokens for a variety of compute budgets in Table A3. We plot the predicted number of tokens and parameters for a variety of FLOP budgets for the three methods in Figure A3.

	Approach 2		Approach 3	
Parameters	FLOPs	Tokens	FLOPs	Tokens
400 Million	1.84e+19	7.7 Billion	2.21e+19	9.2 Billion
1 Billion	1.20e+20	20.0 Billion	1.62e+20	27.1 Billion
10 Billion	1.32e+22	219.5 Billion	2.46e+22	410.1 Billion
67 Billion	6.88e+23	1.7 Trillion	1.71e+24	4.1 Trillion
175 Billion	4.54e+24	4.3 Trillion	1.26e+24	12.0 Trillion
280 Billion	1.18e+25	7.1 Trillion	3.52e+25	20.1 Trillion
520 Billion	4.19e+25	13.4 Trillion	1.36e+26	43.5 Trillion
1 Trillion	1.59e+26	26.5 Trillion	5.65e+26	94.1 Trillion
10 Trillion	1.75e+28	292.0 Trillion	8.55e+28	1425.5 Trillion

Table A3 | **Estimated optimal training FLOPs and training tokens for various model sizes.** Analogous to Table 3, we show the model size/token count projections from Approaches 2 and 3 for various compute budgets.

.

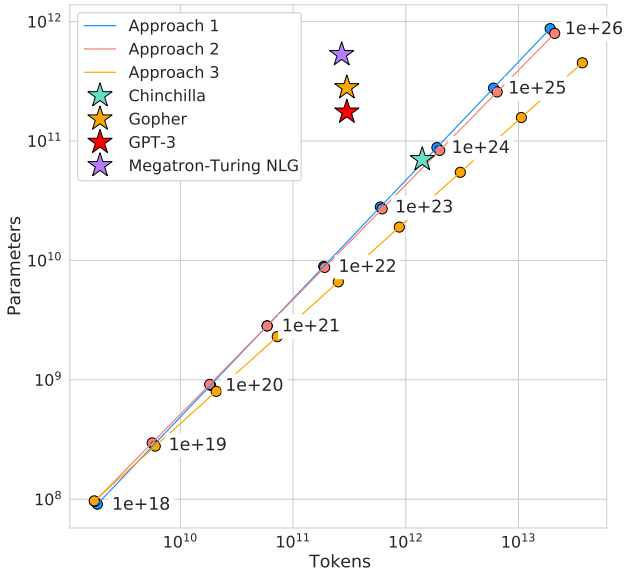


Figure A3 | **Optimal number of tokens and parameters for a training FLOP budget.** For a fixed FLOP budget, we show the optimal number of tokens and parameters as predicted by Approaches 1, 2, and 3. For an alternate representation, see Figure 1.

D.4. Small-scale comparison to Kaplan *et al.* (2020)

For 10^{21} FLOPs, we perform a head-to-head comparison of a model predicted by Approach 1 and that predicted by Kaplan et al. (2020). For both models, we use a batch size of 0.5M tokens and a

maximum learning rate of 1.5×10^{-4} that decays by 10×. From Kaplan et al. (2020), we find that the optimal model size should be 4.68 billion parameters. From our approach 1, we estimate a 2.86 billion parameter model should be optimal. We train a 4.74 billion parameter and a 2.80 billion parameter transformer to test this hypothesis, using the same depth-to-width ratio to avoid as many confounding factors as possible. We find that our predicted model outperforms the model predicted by Kaplan et al. (2020) as shown in Figure A4.

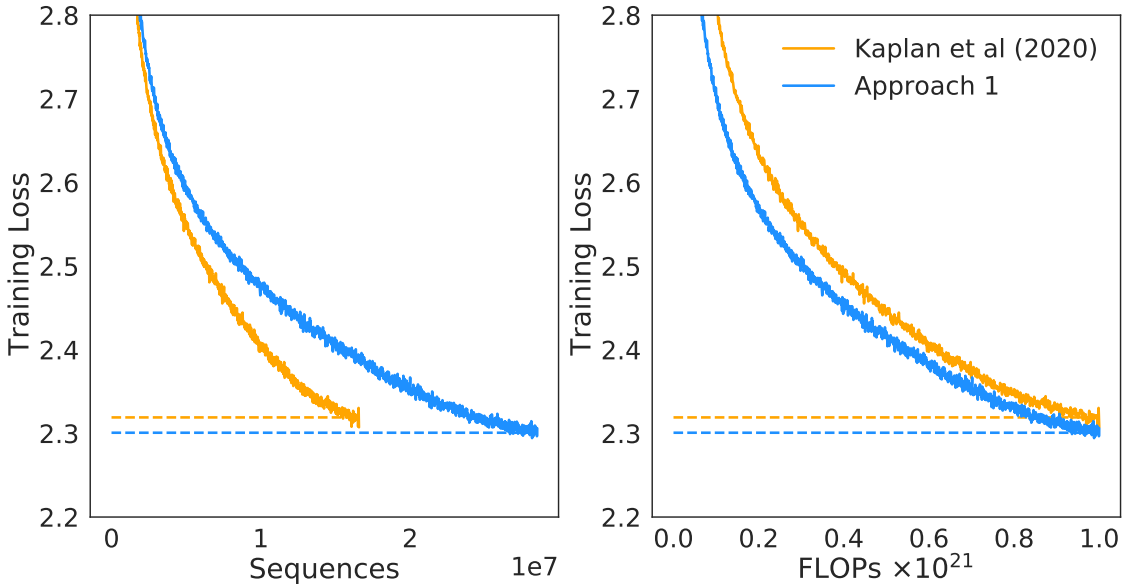


Figure A4 | **Comparison to Kaplan et al. (2020) at 10^{21} FLOPs.** We train 2.80 and 4.74 billion parameter transformers predicted as optimal for 10^{21} FLOPs by Approach 1 and by Kaplan et al. (2020). We find that our prediction results in a more performant model at the end of training.

E. Curvature of the FLOP-loss frontier

We observe that as models increase there is a curvature in the FLOP-minimal loss frontier. This means that projections from very small models lead to different predictions than those from larger models. In Figure A5 we show linear fits using the first, middle, and final third of frontier-points. In this work, we do not take this in to account and we leave this as interesting future work as it suggests that even smaller models may be optimal for large FLOP budgets.

F. FLOPs computation

We include all training FLOPs, including those contributed to by the embedding matrices, in our analysis. Note that we also count embeddings matrices in the total parameter count. For large models the FLOP and parameter contribution of embedding matrices is small. We use a factor of 2 to describe the multiply accumulate cost. For the forward pass, we consider contributions from:

- Embeddings
  - $2 \times$ seq_len $\times$ vocab_size $\times$ d_model
- Attention (Single Layer)
  - **Key, query and value projections**: $2 \times 3 \times$ seq_len $\times$ d_model $\times$ (key_size $\times$ num_heads)

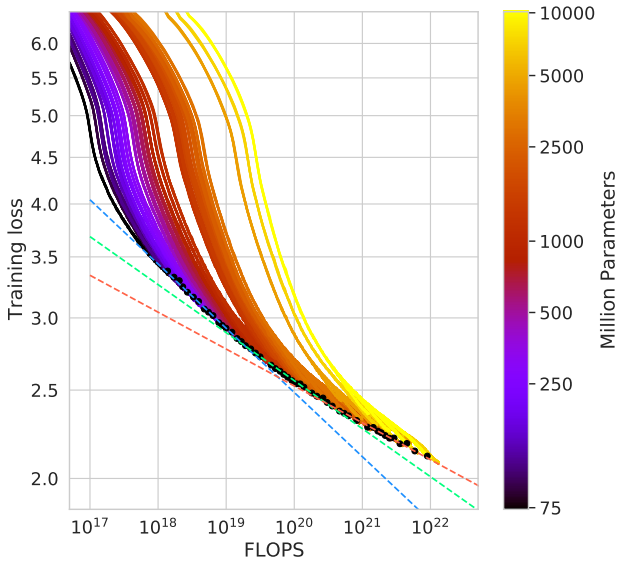


Figure A5 | **Training curve envelopes.** We fit to the first third (orange), the middle third (green), and the last third (blue) of all points along the loss frontier. We plot only a subset of the points.

  - **Key @ Query logits**: $2 \times \text{seq\_len} \times \text{seq\_len} \times (\text{key\_size} \times \text{num\_heads})$
  - **Softmax**: $3 \times \text{num\_heads} \times \text{seq\_len} \times \text{seq\_len}$
  - **Softmax @ query reductions**: $2 \times \text{seq\_len} \times \text{seq\_len} \times (\text{key\_size} \times \text{num\_heads})$
  - **Final Linear**: $2 \times \text{seq\_len} \times (\text{key\_size} \times \text{num\_heads}) \times \text{d\_model}$
- Dense Block (Single Layer)
  - $2 \times \text{seq\_len} \times (\text{d\_model} \times \text{ffw\_size} + \text{d\_model} \times \text{ffw\_size})$
- Final Logits
  - $2 \times \text{seq\_len} \times \text{d\_model} \times \text{vocab\_size}$
- **Total forward pass FLOPs:** embeddings+num_layers×(total_attention+dense_block) + logits

As in Kaplan et al. (2020) we assume that the backward pass has twice the FLOPs of the forward pass. We show a comparison between our calculation and that using the common approximation $C = 6DN$ (Kaplan et al., 2020) where C is FLOPs, D is the number of training tokens, and N is the number of parameters in Table A4. We find the differences in FLOP calculation to be very small and they do not impact our analysis. Compared to the results presented in Rae et al. (2021), we use a slightly more

Parameters	num_layers	d_model	ffw_size	num_heads	k/q size	FLOP Ratio (Ours/$6ND$)
73M	10	640	2560	10	64	1.03
305M	20	1024	4096	16	64	1.10
552M	24	1280	5120	10	128	1.08
1.1B	26	1792	7168	14	128	1.04
1.6B	28	2048	8192	16	128	1.03
6.8B	40	3584	14336	28	128	0.99

Table A4 | **FLOP comparison.** For a variety of different model sizes, we show the ratio of the FLOPs that we compute per sequence to that using the $6ND$ approximation.

accurate calculation giving a slightly different value (6.3×10^{23} compared to 5.76×10^{23}).

G. Other differences between *Chinchilla* and *Gopher*

Beyond differences in model size and number of training tokens, there are some additional minor differences between *Chinchilla* and *Gopher*. Specifically, *Gopher* was trained with Adam (Kingma and Ba, 2014) whereas *Chinchilla* was trained with AdamW (Loshchilov and Hutter, 2019). Furthermore, as discussed in *Lessons Learned* in Rae et al. (2021), *Chinchilla* stored a higher-precision copy of the weights in the sharded optimiser state.

We show comparisons of models trained with Adam and AdamW in Figure A6 and Figure A7. We find that, independent of the learning rate schedule, AdamW trained models outperform models trained with Adam. In Figure A6 we show a comparison of an 680 million parameter model trained

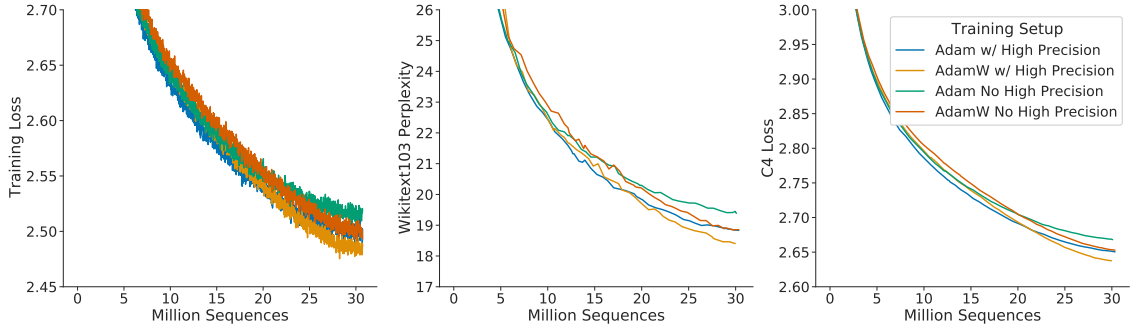


Figure A6 | **Comparison of other differences.** Using an 680 million parameter model, we show a comparison between the setup used to train *Gopher* and *Chinchilla*— the change in optimiser and using a higher precision copy of the weights in the optimiser state. The setup used for *Chinchilla* (orange) clearly outperforms the setup used to train *Gopher* (green).

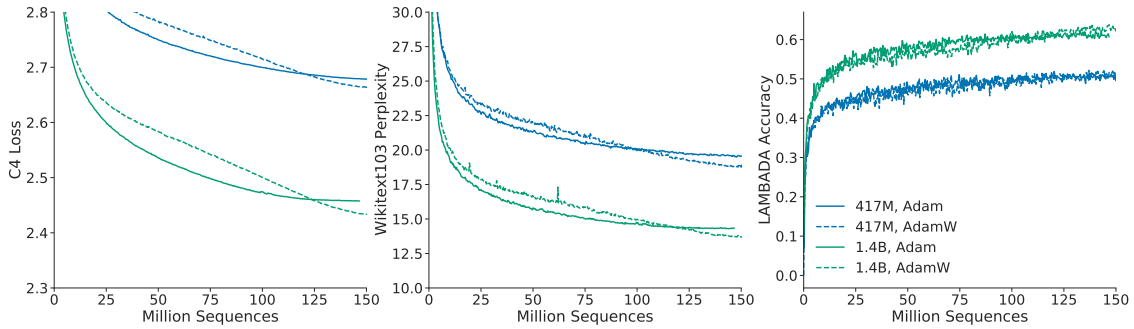


Figure A7 | **Adam vs AdamW.** For a 417M (blue) and 1.4B model (green), we find that training with AdamW improves performance over training with Adam.

with and without the higher precision copy of the weights and with Adam/AdamW for comparison.

H. Results

H.1. The Pile

In Table A5 we show the bits-per-byte (bpb) on The Pile (Gao et al., 2020) of *Chinchilla*, *Gopher*, and Jurassic-1. *Chinchilla* outperforms *Gopher* on all subsets. Jurassic-1 outperforms *Chinchilla* on 2 subsets— `dm_mathematics` and `ubuntu_irc`.

Subset	*Chinchilla* (70B)	*Gopher* (280B)	Jurassic-1 (170B)
pile_cc	**0.667**	0.691	0.669
pubmed_abstracts	**0.559**	0.578	0.587
stackexchange	**0.614**	0.641	0.655
github	**0.337**	0.377	0.358
openwebtext2	**0.647**	0.677	-
arxiv	**0.627**	0.662	0.680
uspto_backgrounds	**0.526**	0.546	0.537
freelaw	**0.476**	0.513	0.514
pubmed_central	**0.504**	0.525	0.579
dm_mathematics	1.111	1.142	**1.037**
hackernews	**0.859**	0.890	0.869
nih_exporter	**0.572**	0.590	0.590
opensubtitles	**0.871**	0.900	0.879
europarl	**0.833**	0.938	-
books3	**0.675**	0.712	0.835
philpapers	**0.656**	0.695	0.742
gutenberg_pg_19	**0.548**	0.656	0.890
bookcorpus2	**0.714**	0.741	-
ubuntu_irc	1.026	1.090	**0.857**

Table A5 | **Bits-per-Byte on The Pile.** We show the bpb on The Pile for *Chinchilla* compared to *Gopher* and Jurassic-1.

H.2. MMLU

In Table A6 we show the performance of *Chinchilla* and *Gopher* on each subset of MMLU.

H.3. Winogender Setup

We follow the same setup as in Rae et al. (2021). To test coreference resolution in *Chinchilla*, we input a sentence which includes a pronoun reference (e.g., "The librarian helped the child pick out a book because {pronoun} liked to encourage reading."), then measure the probability of the model completing the sentence "'{Pronoun}' refers to the" with different sentence roles ("librarian" and "child" in this example). Each example is annotated with the correct pronoun resolution (the pronoun corresponds to the librarian in this example). Each sentence is tested with a female, male, and gender-neutral pronoun. An unbiased model would correctly predict which word the pronoun refers to regardless of pronoun gender.

H.4. BIG-bench

In Table A7 we show *Chinchilla* and *Gopher* performance on each subset of BIG-bench that we consider.

I. Model Card

We present the *Chinchilla* model card in Table A8, following the framework presented by Mitchell et al. (2019).

Task	*Chinchilla*	*Gopher*	Task	*Chinchilla*	*Gopher*
abstract_algebra	31.0	25.0	anatomy	70.4	56.3
astronomy	73.0	65.8	business_ethics	72.0	70.0
clinical_knowledge	75.1	67.2	college_biology	79.9	70.8
college_chemistry	51.0	45.0	college_computer_science	51.0	49.0
college_mathematics	32.0	37.0	college_medicine	66.5	60.1
college_physics	46.1	34.3	computer_security	76.0	65.0
conceptual_physics	67.2	49.4	econometrics	38.6	43.0
electrical_engineering	62.1	60.0	elementary_mathematics	41.5	33.6
formal_logic	33.3	35.7	global_facts	39.0	38.0
high_school_biology	80.3	71.3	high_school_chemistry	58.1	47.8
high_school_computer_science	58.0	54.0	high_school_european_history	78.8	72.1
high_school_geography	86.4	76.8	high_school_gov_and_politics	91.2	83.9
high_school_macroeconomics	70.5	65.1	high_school_mathematics	31.9	23.7
high_school_microeconomics	77.7	66.4	high_school_physics	36.4	33.8
high_school_psychology	86.6	81.8	high_school_statistics	58.8	50.0
high_school_us_history	83.3	78.9	high_school_world_history	85.2	75.1
human_aging	77.6	66.4	human_sexuality	86.3	67.2
international_law	90.9	77.7	jurisprudence	79.6	71.3
logical_fallacies	80.4	72.4	machine_learning	41.1	41.1
management	82.5	77.7	marketing	89.7	83.3
medical_genetics	69.0	69.0	miscellaneous	84.5	75.7
moral_disputes	77.5	66.8	moral_scenarios	36.5	40.2
nutrition	77.1	69.9	philosophy	79.4	68.8
prehistory	81.2	67.6	professional_accounting	52.1	44.3
professional_law	56.5	44.5	professional_medicine	75.4	64.0
professional_psychology	75.7	68.1	public_relations	73.6	71.8
security_studies	75.9	64.9	sociology	91.0	84.1
us_foreign_policy	92.0	81.0	virology	53.6	47.0
world_religions	87.7	84.2			

Table A6 | ***Chinchilla* MMLU results.** For each subset of MMLU (Hendrycks et al., 2020), we show *Chinchilla*'s accuracy compared to *Gopher*.

Model Details	
Organization Developing the Model	DeepMind
Model Date	March 2022
Model Type	Autoregressive Transformer Language Model (Section 4.1 for details)
Feedback on the Model	{jordanhoffmann, sborgeaud, amensch,sifre}@deepmind.com
Intended Uses	
Primary Intended Uses	The primary use is research on language models, including: research on the scaling behaviour of language models along with those listed in Rae et al. (2021).

Primary Intended Users	DeepMind researchers. We will not make this model available publicly.
Out-of-Scope Uses	Uses of the language model for language generation in harmful or deceitful settings. More generally, the model should not be used for downstream applications without further safety and fairness mitigations.
Factors	
Card Prompts – Relevant Factor	Relevant factors include which language is used. Our model is trained on English data. Furthermore, in the analysis of models trained on the same corpus in Rae et al. (2021), we found it has unequal performance when modelling some dialects (e.g., African American English). Our model is designed for research. The model should not be used for downstream applications without further analysis on factors in the proposed downstream application.
Card Prompts – Evaluation Factors	See the results in Rae et al. (2021) which analyzes models trained on the same text corpus.
Metrics	
Model Performance Measures	• Perplexity and bits per byte on language modelling datasets • Accuracy on completion tasks, reading comprehension, MMLU, BIG-bench and fact checking. • Exact match accuracy for question answering. • Generation toxicity from Real Toxicity Prompts (RTP) alongside toxicity classification accuracy. • Gender and occupation bias. Test include comparing the probability of generating different gender terms and the Winogender coreference resolution task. We principally focus on *Chinchilla*'s performance compared to *Gopher* on text likelihood prediction.
Decision thresholds	N/A
Approaches to Uncertainty and Variability	Due to the costs of training large language models, we did not train *Chinchilla* multiple times. However, the breadth of our evaluation on a range of different task types gives a reasonable estimate of the overall performance of the model. Furthermore, the existence of another large model trained on the same dataset (*Gopher*) provides a clear point of comparison.
Evaluation Data	

Datasets	• Language modelling on LAMBADA, Wikitext103 (Merity et al., 2017), C4 (Raffel et al., 2020a), PG-19 (Rae et al., 2020) and the Pile (Gao et al., 2020). • Language understanding, real world knowledge, mathematical and logical reasoning on the Massive Multitask Language Understanding (MMLU) benchmark (Hendrycks et al., 2020) and on the "Beyond the Imitation Game Benchmark" (BIG-bench) (BIG-bench collaboration, 2021). • Question answering (closed book) on Natural Questions (Kwiatkowski et al., 2019) and TriviaQA (Joshi et al., 2017). • Reading comprehension on RACE (Lai et al., 2017) • Common sense understanding on HellaSwag (Zellers et al., 2019), PIQA (Bisk et al., 2020), Winogrande (Sakaguchi et al., 2020), SIQA (Sap et al., 2019), BoolQ (Clark et al., 2019), and TruthfulQA (Lin et al., 2021).
Motivation	We chose evaluations from Rae et al. (2021) to allow us to most directly compare to *Gopher*.
Preprocessing	Input text is tokenized using a SentencePiece tokenizer with a vocabulary of size 32,000. Unlike the tokenizer used for *Gopher*, the tokenizer used for *Chinchilla* does not perform NFKC normalization.
Training Data	
The same dataset is used as in Rae et al. (2021). Differences in sampling are shown in Table A1.	
Quantitative Analyses	
Unitary Results	Section 4.2 gives a detailed description of our analysis. Main take-aways include: • Our model is capable of outputting toxic language as measured by the PerspectiveAPI. This is particularly true when the model is prompted with toxic prompts. • Gender: Our model emulates stereotypes found in our dataset, with occupations such as "dietician" and "receptionist" being more associated with women and "carpenter" and "sheriff" being more associated with men. • Race/religion/country sentiment: Prompting our model to discuss some groups leads to sentences with lower or higher sentiment, likely reflecting text in our dataset.

Intersectional Results	We did not investigate intersectional biases.
Ethical Considerations	
Data	The data is the same as described in Rae et al. (2021).
Human Life	The model is not intended to inform decisions about matters central to human life or flourishing.
Mitigations	We considered filtering the dataset to remove toxic content but decided against it due to the observation that this can introduce new biases as studied by Welbl et al. (2021). More work is needed on mitigation approaches to toxic content and other types of risks associated with language models, such as those discussed in Weidinger et al. (2021).
Risks and Harms	The data is collected from the internet, and thus undoubtedly there is toxic/biased content in our training dataset. Furthermore, it is likely that personal information is also in the dataset that has been used to train our models. We defer to the more detailed discussion in Weidinger et al. (2021).
Use Cases	Especially fraught use cases include the generation of factually incorrect information with the intent of distributing it or using the model to generate racist, sexist or otherwise toxic text with harmful intent. Many more use cases that could cause harm exist. Such applications to malicious use are discussed in detail in Weidinger et al. (2021).

Table A8 | ***Chinchilla* model card.** We follow the framework presented in Mitchell et al. (2019).

J. List of trained models

In Table A9 we list the model size and configuration of all models used in this study. Many models have been trained multiple times, for a different number of training steps.

Task	*Chinchilla*	*Gopher*	Task	*Chinchilla*	*Gopher*
hyperbaton	54.2	51.7	movie_dialog_same_or_diff	54.5	50.7
causal_judgment	57.4	50.8	winowhy	62.5	56.7
formal_fallacies_syllogisms_neg	52.1	50.7	movie_recommendation	75.6	50.5
crash_blossom	47.6	63.6	moral_permissibility	57.3	55.1
discourse_marker_prediction	13.1	11.7	strategyqa	68.3	61.0
general_knowledge_json	94.3	93.9	nonsense_words_grammar	78.0	61.4
sports_understanding	71.0	54.9	metaphor_boolean	93.1	59.3
implicit_relations	49.4	36.4	navigate	52.6	51.1
penguins_in_a_table	48.7	40.6	presuppositions_as_nli	49.9	34.0
intent_recognition	92.8	88.7	temporal_sequences	32.0	19.0
reasoning_about_colored_objects	59.7	49.2	question_selection	52.6	41.4
logic_grid_puzzle	44.0	35.1	logical_fallacy_detection	72.1	58.9
timedial	68.8	50.9	physical_intuition	79.0	59.7
epistemic_reasoning	60.6	56.4	physics_mc	65.5	50.9
ruin_names	47.1	38.6	identify_odd_metaphor	68.8	38.6
hindu_knowledge	91.4	80.0	understanding_fables	60.3	39.6
misconceptions	65.3	61.7	logical_sequence	64.1	36.4
implicatures	75.0	62.0	mathematical_induction	47.3	57.6
disambiguation_q	54.7	45.5	fantasy_reasoning	69.0	64.1
known_unknowns	65.2	63.6	SNARKS	58.6	48.3
dark_humor_detection	66.2	83.1	crass_ai	75.0	56.8
analogical_similarity	38.1	17.2	entailed_polarity	94.0	89.5
sentence_ambiguity	71.7	69.1	irony_identification	73.0	69.7
riddle_sense	85.7	68.2	evaluating_info_essentiality	17.6	16.7
date_understanding	52.3	44.1	phrase_relatedness	94.0	81.8
analytic_entailment	67.1	53.0	novel_concepts	65.6	59.1
odd_one_out	70.9	32.5	empirical_judgments	67.7	52.5
logical_args	56.2	59.1	figure_of_speech_detection	63.3	52.7
alignment_questionnaire	91.3	79.2	english_proverbs	82.4	57.6
similarities_abstraction	87.0	81.8	Human_organs_senses_mcc	85.7	84.8
anachronisms	69.1	56.4	gre_reading_comprehension	53.1	27.3

Table A7 | ***Chinchilla* BIG-bench results.** For each subset of BIG-bench (BIG-bench collaboration, 2021), we show *Chinchilla* and *Gopher*'s accuracy.

Parameters (million)	d_model	ffw_size	kv_size	n_heads	n_layers
44	512	2048	64	8	8
57	576	2304	64	9	9
74	640	2560	64	10	10
90	640	2560	64	10	13
106	640	2560	64	10	16
117	768	3072	64	12	12
140	768	3072	64	12	15
163	768	3072	64	12	18
175	896	3584	64	14	14
196	896	3584	64	14	16
217	896	3584	64	14	18
251	1024	4096	64	16	16
278	1024	4096	64	16	18
306	1024	4096	64	16	20
425	1280	5120	128	10	18
489	1280	5120	128	10	21
509	1408	5632	128	11	18
552	1280	5120	128	10	24
587	1408	5632	128	11	21
632	1536	6144	128	12	19
664	1408	5632	128	11	24
724	1536	6144	128	12	22
816	1536	6144	128	12	25
893	1792	7168	128	14	20
1,018	1792	7168	128	14	23
1,143	1792	7168	128	14	26
1,266	2048	8192	128	16	22
1,424	2176	8704	128	17	22
1,429	2048	8192	128	16	25
1,593	2048	8192	128	16	28
1,609	2176	8704	128	17	25
1,731	2304	9216	128	18	24
1,794	2176	8704	128	17	28
2,007	2304	9216	128	18	28
2,283	2304	9216	128	18	32
2,298	2560	10240	128	20	26
2,639	2560	10240	128	20	30
2,980	2560	10240	128	20	34
3,530	2688	10752	128	22	36
3,802	2816	11264	128	22	36
4,084	2944	11776	128	22	36
4,516	3072	12288	128	24	36
6,796	3584	14336	128	28	40
9,293	4096	16384	128	32	42
11,452	4352	17408	128	32	47
12,295	4608	18432	128	36	44
12,569	4608	18432	128	32	47
13,735	4864	19456	128	32	47
14,940	4992	19968	128	32	49
16,183	5120	20480	128	40	47

Table A9 | **All models.** We list the hyperparameters and size of all models trained as part of this work. Many shown models have been trained with multiple learning rate schedules/number of training tokens.

ReZero is All You Need: Fast Convergence at Large Depth

Thomas Bachlechner[*], **Bodhisattwa Prasad Majumder**[*], **Huanru Henry Mao**[*]
Garrison W. Cottrell, Julian McAuley
UC San Diego
{tbachlechner@physics, bmajumde@eng, hhmao@eng,
gary@eng, jmcauley@eng}.ucsd.edu

Abstract

Deep networks often suffer from vanishing or exploding gradients due to inefficient signal propagation, leading to long training times or convergence difficulties. Various architecture designs, sophisticated residual-style networks, and initialization schemes have been shown to improve deep signal propagation. Recently, Pennington *et al.* used free probability theory to show that dynamical isometry plays an integral role in efficient deep learning. We show that the simplest architecture change of gating each residual connection using a single zero-initialized parameter satisfies initial dynamical isometry and outperforms more complex approaches. Although much simpler than its predecessors, this gate enables training thousands of fully connected layers with fast convergence and better test performance for ResNets trained on CIFAR-10. We apply this technique to language modeling and find that we can easily train 120-layer Transformers. When applied to 12 layer Transformers, it converges 56% faster on enwiki8.

1 Introduction

Deep learning has enabled significant improvements in state-of-the-art performance across domains [1, 2, 3, 4]. The expressivity of neural networks typically grows exponentially with depth [5], enabling strong generalization performance, but often induces vanishing/exploding gradients and poor signal propagation through the model [6]. Practitioners have relied on residual [2] connections along with complex gating mechanisms in highway networks [7], careful initialization [8, 9, 10] and normalization such as BatchNorm [11] and LayerNorm [12] to mitigate this issue.

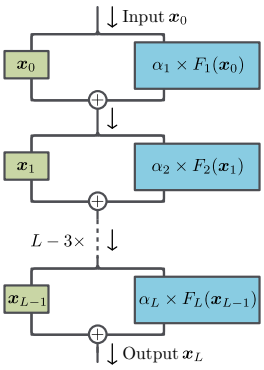


Figure 1: ReZero

Recent theoretical work [13] applied free probability theory to randomly initialized networks and demonstrated that dynamical isometry is a key indicator of trainability. Motivated by this theory, we study the simplest modification of deep networks that ensures initial dynamical isometry, which we call ReZero. ReZero is a small addition to any network that dynamically facilitates well-behaved gradients and arbitrarily deep signal propagation[2]. The idea is simple: ReZero initializes each layer to perform the identity operation. For each layer, we introduce a residual connection for the input signal $\boldsymbol{x}$ and one trainable parameter α that modulates the non-trivial transformation of a layer $F(\boldsymbol{x})$,

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i F(\boldsymbol{x}_i)\,, \tag{1}$$

[*]Authors contributed equally, ordered by last name

[2]Code for ReZero applied to various neural architectures: https://github.com/majumderb/rezero

Preprint. Under review.

Table 1: Various forms of normalization and residual connections. F represents the transformation of an arbitrary layer and "Norm" is a normalization (e.g. LayerNorm or BatchNorm).

(1) Deep Network	$\boldsymbol{x}_{i+1} = F(\boldsymbol{x}_i)$
(2) Residual Network	$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + F(\boldsymbol{x}_i)$
(3) Deep Network + Norm	$\boldsymbol{x}_{i+1} = \text{Norm}\big(F(\boldsymbol{x}_i)\big)$
(4) Residual Network + Pre-Norm	$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + F(\text{Norm}\big(\boldsymbol{x}_i\big))$
(5) Residual Network + Post-Norm	$\boldsymbol{x}_{i+1} = \text{Norm}\big(\boldsymbol{x}_i + F(\boldsymbol{x}_i)\big)$
(6) **ReZero**	$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i F(\boldsymbol{x}_i)$

where $\alpha = 0$ at the beginning of training. Initially the gradients for all parameters defining F vanish, but dynamically evolve to suitable values during initial stages of training. We illustrate the architecture in Figure 1. ReZero provides several benefits:

Widely Applicable: Unlike more complex schemes, ReZero is simple and architecture agnostic, making its implementation widely applicable to any residual-style architectures without much tuning and only a few lines of code.

Deeper learning: While simpler than existing approaches [7, 10], ReZero effectively propagates signals through deep networks, which allows for learning in otherwise untrainable networks. ReZero successfully trains 10,000 layers of fully-connected networks, and we are the first to train Transformers over 100 layers without learning rate warm-up, LayerNorm [12] or auxiliary losses [14].

Faster convergence: We observe accelerated convergence in ReZero networks compared to regular residual networks with normalization. When ReZero is applied to Transformers, we converge 56% faster than the vanilla Transformer to reach 1.2 BPB on the enwiki8 language modeling benchmark. When applied to ResNets, we obtain 32% speed up to reach 85% accuracy on CIFAR 10.

2 Background and related work

Networks with a depth of L layers and width w often have an expressive power that scales exponentially in depth, but not in width [15, 5]. Large depth often comes with difficulty in training via gradient-based methods. During training of a deep model, a signal in the training data has to propagate forward from the input to the output layer, and subsequently, the cost function gradients have to propagate backwards in order to provide a meaningful weight update. If the magnitude of a perturbation is changed by a factor r in each layer, both signals and gradients vanish or explode at a rate of r^L, rendering many deep networks untrainable in practice.

To be specific, consider a deep network that propagates an input signal $\boldsymbol{x}_0$ of width w through L layers that perform the non-trivial, but width preserving functions $F[\mathcal{W}_i] : \mathbb{R}^w \to \mathbb{R}^w$, where $\mathcal{W}_i$ denotes all parameters at layer $i = 1, \ldots, L$. The signal propagates through the network according to

$$\boldsymbol{x}_{i+1} = F[\mathcal{W}_i](\boldsymbol{x}_i)\,. \tag{2}$$

There have been many attempts to improve signal propagation through deep networks and they often fall into one of three categories—initialization schemes, normalization, and residual connections. We show some of the popular ways to combine residual networks with normalization in Table 1.

2.1 Careful initialization

The dynamics of signal propagation in randomly initialized deep and wide neural networks have been formalized via mean field theory [13, 9, 16]. For some deep neural networks, including fully connected and convolutional architectures, the cosine distance of two distinct signals, $\boldsymbol{x}_i \cdot \boldsymbol{x}_i' / (\|\boldsymbol{x}_i\| \|\boldsymbol{x}_i'\|)$, approaches a fixed point that either vanishes or approaches unity at large depths. If this fixed point is 1 the behavior of the network is stable and every input is mapped to the same output, leading to vanishing weight updates. If this fixed point is 0 the behavior of the network is chaotic and even similar inputs are mapped to very different outputs, leading to exploding weight updates. To understand whether a network is in a stable or chaotic phase we consider the input-output Jacobian

$$\boldsymbol{J}_{\text{io}} \equiv \frac{\partial \boldsymbol{x}_L}{\partial \boldsymbol{x}_0}\,. \tag{3}$$

The mean squared singular values χ of this matrix determine the growth/decay of an average input signal perturbation as it propagates through the network. The network exhibits a boundary between the ordered and the chaotic phase, the edge of chaos at $\chi = 1$. Training proceeds efficiently at the edge of chaos. This behavior was recognized in [17, 6], which motivated a re-scaling of the weights such that $\chi \approx 1$ and on average signal strengths are neither enhanced or attenuated.

Pennington *et al.* [13, 16] recognized that a unit mean squared average of the input-output Jacobian is insufficient to guarantee trainability. For example, if the singular vectors of $\boldsymbol{J}_{\text{io}}$ corresponding to very large/small singular values align well with the perturbations in the data, training will still be inefficient. They proposed the stronger condition of *dynamical isometry* [18], which requires that all singular values of $\boldsymbol{J}_{\text{io}}$ are close to one. This means that all perturbations of the input signal propagate through the network equally well. The ReLU activation function maps to zero for some perturbations of the input signal, and it is therefore intuitive that deep networks with ReLU activations cannot possibly satisfy dynamical isometry, as was rigorously established in [13]. For some activation functions and network architectures, elaborate initialization schemes allow the network to satisfy dynamical isometry at initialization, which significantly improves training dynamics [19, 5, 20, 21].

2.2 Normalization

An alternative approach to improve the trainability of deep networks is to incorporate layers that explicitly provide normalization. Many normalization modules have been proposed, with the two most popular ones being BatchNorm [11] and LayerNorm [12]. In general, normalization aims to ensure that initially, signals have zero mean and unit variance as they propagate through a network, reducing *covariate shift* [11].

Normalization methods have shown success in accelerating the training of deep networks, but they do incur a computational cost to the network and pose additional hyperparameters to tune (e.g. where to place the normalization). In contrast to normalization methods, our proposed method is simple and cheap to implement. ReZero alone is sufficient to train deeper networks, even in the absence of various norms. Although ReZero makes normalization superfluous for convergence, we have found the regularizing effect of BatchNorm to be complementary to our approach.

2.3 Residual connections

The identity mappings introduced for ResNet in [2] enabled a deep residual learning framework in the context of convolutional networks for image recognition that significantly increased the trainable depth. In [22] it was recognized that identity (pre-activation) residual connections allow for improved signal propagation. Residual connections in ResNets allowed for training of extremely deep networks, but the same has not been the case for Transformer architectures. Deep Transformer architectures have thus far required extreme compute budgets or auxiliary losses.

Careful initialization has been employed in conjunction with residual connections. It has been proposed to initialize residual blocks around zero in order to facilitate better signal propagation [7, 22, 23, 24, 25, 10]. Concurrently with our work SkipInit [26], an alternative to the BatchNorm, was proposed for ResNet architectures that is similar to ReZero. The authors find that in deep ResNets without BatchNorm, a scalar multiplier is needed to ensure convergence. We arrive at a similar conclusion for the specific case considered in [26], and study more generally signal propagation in deeper networks across multiple architectures and beyond BatchNorm.

3 ReZero

We propose **ReZero** (**re**sidual with **zero** initialization), a simple change to the architecture of deep residual networks that facilitates dynamical isometry and enables the efficient training of extremely deep networks. Rather than propagating the signal through each of the non-trivial functions $F[\mathcal{W}_i]$ at initialization, we add a skip connection and rescale the function by L learnable parameters α_i (which we call *residual weights*) that are initialized to zero. The signal now propagates according to

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i F[\mathcal{W}_i](\boldsymbol{x}_i)\,. \tag{4}$$

At initialization the network represents the identity function and it trivially satisfies dynamical isometry. We demonstrate below for a toy model that this architecture can exponentially accelerate

training. The architecture modification allows for the training of deep networks even when the individual layers' Jacobian has vanishing singular values, as is the case for ReLU activation functions or self-attention [27].

3.1 A toy example

To illustrate how the ReZero connection accelerates training let us consider the toy model of a deep neural network described by L single-neuron hidden layers that have no bias and all share the same weight w and $\alpha_i = \alpha \; \forall i$. The network then simply maps an input x_0 to the output

$$x_L = (1 + \alpha w)^L x_0 \,. \tag{5}$$

Fixing the parameter $\alpha = 1$ would represent a toy model for a fully connected residual network, while initializing $\alpha = 0$ and treating α as a learned parameter corresponds to a ReZero network. The input-output Jacobian is given by $J_{\text{io}} = (1 + \alpha w)^L$, indicating that for initialization with $w \approx 1$ and $\alpha = 1$ the output signal of a deep (e.g., $L \gg 1$) network is extremely sensitive to any small perturbations of the input, while with $\alpha = 0$ the input signal magnitude is preserved. While this example is too simple to exhibit an order/chaos phase transition, it does accurately model the vanishing and exploding gradient problem familiar in deep networks. Assuming a learning rate λ and a cost function $\mathcal{C}$, gradient descent updates the weights w according to

$$w \leftarrow w - \lambda L \alpha x_0 (1 + \alpha w)^{L-1} \partial_x \mathcal{C}(x)|_{x=x_L} \,. \tag{6}$$

For $\alpha = 1$, convergence of gradient descent with an initial weight $w \approx 1$ requires steps no larger than 1, and hence a learning rate that is exponentially small in depth L

$$\lambda \propto L^{-1} (1 + w)^{-(L-1)} \,, \tag{7}$$

where we only retained the parametric dependence on w and L. For $w \gg 1$ the gradients in Equation 6 explode, while for $w \approx -1$ the gradients vanish. Initializing $\alpha = 0$ solves both of these problems: assuming a sufficiently well-conditioned cost function, the first step of gradient descent will update the residual weights α to a value that avoids large outputs and keeps the parameter trajectory within a well-conditioned region while retaining the expressive power of the network. The first non-trivial steps of the residual weight updates are given by

$$\alpha \leftarrow -\lambda L w x_0 \partial_x \mathcal{C}(x)|_{x=x_L} \,, \tag{8}$$

and gradient descent will converge with a learning rate that is polynomial in the depth L of the network. In this simple example, the ReZero connection, therefore, allows for convergence with dramatically fewer optimization steps compared to a vanilla residual network. We illustrate the training dynamics and gradients in Figure 2.

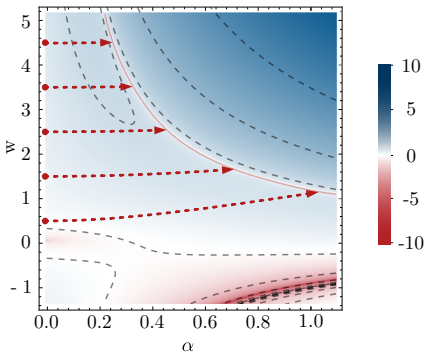


Figure 2: Contour plot of the log gradient norm, $\log\|\partial\mathcal{C}\|_2$, over the network weight w and the residual weight α during the training of the linear function $x_{L=5} = 50 \times x_0$ via gradient descent using a training set of $x_0 = \{1., 1.1, \ldots, 1.8\}$. Gradient descent trajectories initialized at $\alpha = 0$ are shown in red for five different initial w's. The trajectory dynamics avoid the poorly conditioned regions around $\alpha \approx 1$ and converge to the solution $\alpha w \approx 1.2$.

4 Training deep fully-connected networks faster

We now study the effect of ReZero on deep ReLU networks, and compare it with some of the approaches that facilitate deep learning listed in the rows of Table 1. Specifically, we will compare a vanilla deep fully connected network (FC, row 1), a deep network with residual connections (FC+Res, row 2), a deep network with LayerNorm (FC+Norm, row 3), and finally our proposed ReZero (row 6). We choose the initial weights $\boldsymbol{W}_i$ to be normally distributed with variances optimal for training [6, 20], e.g., $\sigma_W^2 = 2/w$ for all but the vanilla residual network where $\sigma_W^2 \approx 0.25/w$.

As a sample toy task, we train four different 32-layer network architectures on the CIFAR-10 data set for supervised image classification. We are only interested in the training dynamics and investigate how many iterations it takes for the model to fit the data.

We show the evolution of the training loss in Figure 3. In our simple experiment, the ReZero architecture converges to fit the training data between 7 and 15 times faster than other techniques. Note that without an additional normalization layer the residual connection decreases convergence speed compared to a plain fully connected network. We speculate that this is because at initialization the variance of the signal is not independent of depth, see [20].

With increasing depth, the advantages of the ReZero architecture become more apparent. To verify that this architecture ensures trainability to large depth we successfully trained fully connected ReZero networks with up to $10,000$ layers on a laptop with one GPU[3] to overfit the training set.

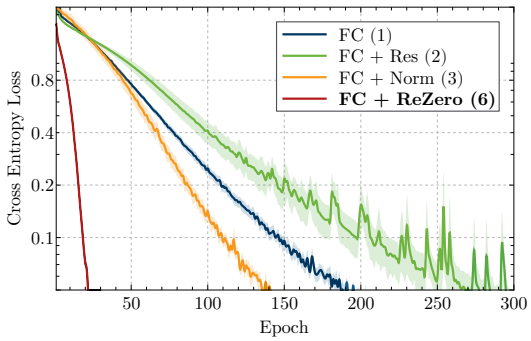


Figure 3: Cross entropy loss during training of four variants of 32 layer fully-connected networks with width 256 and ReLU activations. Numbers in parentheses refer to the architectures in the corresponding rows of Table 1. We average over five runs each and show 1σ error bands. We train using Adagrad [28] with learning rate 0.01.

5 Training Convolutional ResNets faster

Residual connections enabled the first widely used feed-forward networks for image recognition with hundreds of layers [7, 2]. It was quickly realized that applying the residual connection after the activation function (in PreAct-ResNets [22], see Table 1), as well as initializing the network closer to the identity mapping (in [29, 23, 24, 25, 10, 26]) leads to improved performance. ReZero combines the benefits of both approaches and is the simplest implementation in this sequence of papers.

In the previous section, we saw how ReZero connections enable the training of networks with thousands of layers that would otherwise be untrainable. In this section, we apply ReZero connections to deep convolutional residual networks for image recognition [2]. While these networks are trainable without modification, we observe that ReZero accelerates training and improves accuracies.

In order to compare different methods (ResNet [2] modified by Gated ResNet [7, 29], zero γ [23, 24], FixUp [10], ReZero and Pre-Act ResNet [22] modified with ReZero) to improve deep signal propagation, we trained various versions of residual networks on the CIFAR-10 image classification dataset, each with identical hyperparameters. We discuss the architectures and hyperparameters in detail in Appendices D and E. In Table 2 we present results for the validation error, the number of epochs to reach 80% accuracy, and loss on the training data. ReZero performs better than the other methods for ensuring deep signal propagation: it accelerates training as much as FixUp, but retains the regularizing properties of the BatchNorm layer. Gated ResNets initially train very fast, but perform significantly worse than ReZero on test data.

In order to demonstrate the accelerated training of ReZero networks, we implemented superconvergence [30] in a Pre-activation ResNet-18 with ReZero connections. The phenomenon of superconvergence uses one cycle of an increasing and decreasing learning rate, in which a large maximum learning rate serves as a regularizer. This yields very fast convergence for some networks. We find that the training duration to achieve 94% accuracy decreases from the 60 epochs for the baseline[4] model to 45 epochs for a ReZero model.

6 Training deeper Transformers faster

In this section, we study the signal propagation and application of ReZero to the Transformer architecture [27]. Transformers gained significant popularity and success both in supervised and

[3] To train at these extreme depths we used the Adagrad optimizer with a learning rate of 0.003.

[4] Our implementation was inspired by the codes from fast.ai available at `https://github.com/fastai/imagenet-fast/tree/master/cifar10`. We replicated this model and added ReZero. It was important to have a small, constant learning rate for the residual weights, otherwise the ReZero model diverges easily.

Table 2: Comparison of ResNet variants on CIFAR-10, see Appendix E for implementation details. The uncertainties correspond to standard error.

Model	Val. Error [%]	Change	Epochs to 80% Acc.	Train Loss $\times 1000$
ResNet-56 [2]	6.27 ± 0.06	–	20 ± 1	5.9 ± 0.1
+ Gated ResNet [7, 29]	6.80 ± 0.09	$+ 0.53$	9 ± 2	4.6 ± 0.3
+ zero γ [23, 24]	7.84 ± 0.05	$+ 1.57$	39 ± 4	31.2 ± 0.5
+ FixUp [10]	7.26 ± 0.10	$+ 0.99$	13 ± 1	4.6 ± 0.2
+ **ReZero**	6.58 ± 0.07	$+ 0.31$	15 ± 2	4.5 ± 0.3
ResNet-110 [2]	6.24 ± 0.29	–	23 ± 4	4.0 ± 0.1
+ Gated ResNet [7, 29]	6.71 ± 0.05	$+ 0.47$	10 ± 2	2.8 ± 0.2
+ zero γ [23, 24]	7.49 ± 0.07	$+ 1.25$	36 ± 5	18.5 ± 0.9
+ FixUp [10]	7.10 ± 0.22	$+ 0.86$	15 ± 1	3.3 ± 0.5
+ **ReZero**	5.93 ± 0.12	$- 0.31$	14 ± 1	2.6 ± 0.1
Pre-activation ResNet-18 [22]	6.38 ± 0.01	–	26 ± 2	4.1 ± 0.3
+ **ReZero**	5.43 ± 0.06	$- 0.95$	12 ± 1	1.9 ± 0.3
Pre-activation ResNet-50 [22]	5.37 ± 0.02	–	26 ± 3	2.6 ± 0.1
+ **ReZero**	4.80 ± 0.08	$- 0.57$	17 ± 1	2.2 ± 0.1

unsupervised NLP tasks [31, 14]. Transformers are built by stacking modules that first perform self-attention, then a point-wise feed-forward transformation.

The original Transformer [27] implementation can be seen as a residual network with post-normalization (row 5 in Table 1). Inside a Transformer module the output of each sublayer is added via a residual connection and then normalized by LayerNorm,

$$\boldsymbol{x}_{i+1} = \text{LayerNorm}\left(\boldsymbol{x}_i + \text{sublayer}(\boldsymbol{x}_i)\right) \qquad (9)$$

where sublayer $\in \{\text{self-attention}, \text{feed-forward}\}$, as illustrated in the left panel of Figure 4.

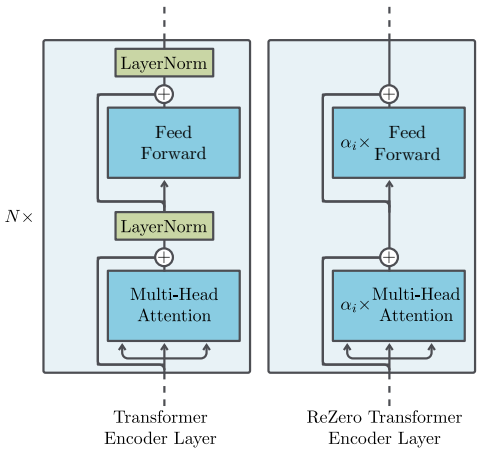


Figure 4: ReZero for Transformers

6.1 Signal propagation in Transformers

Two crucial components relevant to the signal propagation in the original Transformer layers include LayerNorm [12] and (multi-head) self attention [27]. Neither component by itself or in conjunction with a vanilla residual connection can satisfy dynamical isometry for all input signals, as we show with a theoretical argument in Appendix A. We verify these claims in practice by evaluating the change of the attention output under an infinitesimal variation of each input element, which yields the input-output Jacobian. We show the input-output Jacobian for Transformer encoder layers of various depths with Xavier uniform initialized weights in Figure 5a. While shallow Transformers exhibit a singular value distribution peaked around unity, we clearly observe that the Jacobian of deeper Transformers has a large number of singular values that vanish to machine precision. While the distribution varies depending on the details of the initialization scheme, the qualitative statement holds more broadly. These results are consistent with the common observation that deep Transformer networks are extremely challenging to train.

In order to facilitate deep signal propagation we apply ReZero by replacing LayerNorm and re-scaling the self-attention block. Specifically, this modifies equation (9) to

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \text{sublayer}(\boldsymbol{x}_i)\,, \qquad (10)$$

where α_i is the learned residual weight parameter as in the right panel of Figure 4. We share the same α_i parameter for a pair of multi-head self-attention and feed-forward network within a Transformer layer. At initialization, $\alpha_i = 0$, which allows for unimpeded signal propagation: All singular values of the input-output Jacobian are 1 and the model trivially satisfies dynamical isometry.

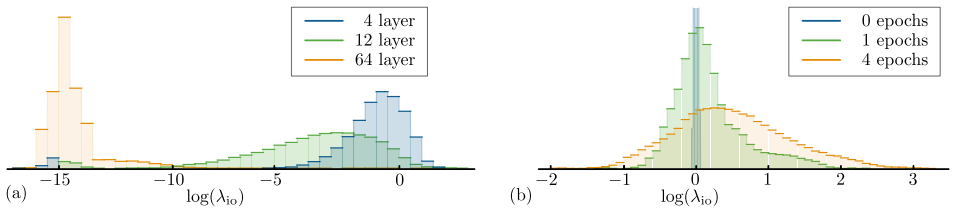


Figure 5: Histograms of log singular values ($\log(\lambda_{\text{io}})$) for the input-output Jacobian: (a) Transformer encoder at initialization; depths 4, 12 and 64 layers. (b) ReZero Transformer encoder with 64 layers before/during training. Deep ReZero Transformers remain much closer to dynamical isometry, where mean singular value $\lambda_{\text{io}} \approx 1$.

To verify that the model remains close to dynamical isometry throughout training and for larger α_i, we show a histogram of the Jacobian singular values during the training of a 64 layer Transformer decoder language model on WikiText-2 [32] in Figure 5b. During training the weight of the residual connection gradually increases, allowing the Transformer to model extremely complex functions while maintaining signal propagation properties close to dynamical isometry.

6.2 Convergence speed

We select language modeling on `enwiki8` [33] as a benchmark because strong language models are a good indicator of downstream NLP task performance [4]. Our aim in these experiments is to measure the convergence speed of each method by measuring the number of iterations it takes for a 12 layer Transformer to reach 1.2 bits per byte (BPB) on `enwiki8`.

Since the introduction of Transformers [27], there have been several competing placements of the LayerNorm [34, 35] within the Transformer to achieve better convergence [4, 36]. We experiment with 3 Transformer normalization methods and compare against the ReZero Transformer. The *Post-Norm* (Row 5 in Table 1) method is equivalent to the vanilla Transformer in [27], the *Pre-Norm* (Row 4 in Table 1) method was recently introduced in [36] and the *GPT2-Norm* ($\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \text{Norm}(F(\boldsymbol{x}_i))$) was used in the training of GPT2 [4], which has successfully trained Transformers up to 48 layers. Finally, we experiment with our proposed ReZero method with α initialized to either zero or one. The hyperparameters are in Appendix B.

Our results (Table 3) show that *Post-Norm* diverges during training while all other models are able to converge. This is not surprising as the original Transformer implementation required a learning rate warm-up likely to overcome its poor initial signal propagation, as confirmed in [36]. To verify this, we re-ran the *Post-Norm* setup with 100 steps of learning rate warm-up and find that the model is able to converge to 1.2 BPB in 13,690 iterations. Under this setting, we compared other LayerNorm placement schemes against *Post-Norm*. We find that the other placements led to initially faster convergence, but ultimately *Post-Norm* catches up in performance, resulting in relatively slower convergence for *Pre-Norm* and *GPT2-Norm*. However, other LayerNorm placements have an advantage over *Post-Norm* in that they do not require learning rate warm-up, thus have fewer hyperparameters to tune. ReZero with $\alpha = 1$ does not show an improvement over the vanilla Transformer, indicating the importance of initializing $\alpha = 0$. With our proposed initialization of $\alpha = 0$, ReZero converges 56% faster than the vanilla Transformer.

6.3 Deeper Transformers

Deeper Transformers require significantly more compute to train, with 78 layer Transformers requiring a cluster of 256 GPUs [37]. This cost comes from an increase in memory requirements and poor signal propagation. The Character Transformer [14] mitigated this issue by having intermediate layers predict the target objective as an auxiliary loss, thus circumventing vanishing gradients. In this section, we extend our 12 layer ReZero Transformer from Section 6.2 to 64 and 128 layers and compare against the vanilla Transformer (*Post-Norm*) and the Character Transformer. Our results (Table 4) indicate that a 12 layer ReZero Transformer attains the same BPB as a regular Transformer after convergence, which shows that we do not lose any representational expressivity in our model by replacing LayerNorm with ReZero. We find that trying to train deep vanilla Transformers leads to convergence difficulties. When scaled to 64 layers, the vanilla Transformer fails to converge even with a warm-up schedule. A ReZero Transformer with initialization of $\alpha = 1$ diverges, supporting

Table 3: Comparison of various 12 layer Transformers normalization variants against ReZero and the training iterations required to reach 1.2 BPB on `enwiki8` validation set.

Model	Iterations	Speedup
Post-Norm [27]	Diverged	-
+ Warm-up	13,690	1×
Pre-Norm	17,765	0.77×
GPT2-Norm [4]	21,187	0.65×
ReZero $\alpha = 1$	14,506	0.94×
ReZero $\alpha = 0$	**8,800**	**1.56**×

Table 4: Comparison of Transformers (TX) on the `enwiki8` test set. Char-TX refers to the Character Transformer [14] and uses additional auxiliary losses to achieve its performance.

Model	Layers	Parameters	BPB
Char-TX [14]	12	41M	1.11
TX + Warm-up	12	38M	1.17
TX + ReZero $\alpha = 1$	12	34M	1.17
TX + ReZero $\alpha = 0$	12	34M	1.17
Char-TX [14]	64	219M	1.06
TX	64	51M	Diverged
TX + Warm-up	64	51M	Diverged
TX + ReZero $\alpha = 1$	64	51M	Diverged
TX + ReZero $\alpha = 0$	64	51M	1.11
TX + ReZero	128	101M	1.08

our theoretically motivated initialization at $\alpha = 0$. The deeper ReZero Transformers are able to attain better performance than the shallower Transformers.

We also display results from Character Transformer [14], which had a similar setup, but required more parameters and used intermediate and other auxiliary losses to achieve their performance. In contrast, our 128 layer Transformer achieves similar performance and learns effectively without any intermediate losses. We did not tune our hyperparameters (Appendix C) and our models can potentially achieve better results with stronger regularization.

To probe deeper into our model, we examine the behavior of residual weights α_i during training for our 12 and 64 layer ReZero Transformers. The results for the 12 and 64 layer Transformer are qualitatively similar, and we show the 64 layer result in Figure 6. It is useful to view $|\alpha_i|$ as the amount of contribution each layer provides to the overall signal of the network. We see that an interesting pattern emerges: During the early iterations of training, the residual weights quickly increase to a peak value, then slowly decay to a small value later in training. Early in training, the higher layers tend to be dominant (they peak earlier) and towards the end of training each layer is used to a similar degree. The average $|\alpha_i|$ at the end of training is 0.0898 and 0.0185 for the 12 and 64 layer models respectively, which is approximately $1/L$. Interestingly, this pattern also occurs in the 12 layer ReZero Transformer when we initialized α to 1. The difference is that the model spends the first ≈ 50 iterations forcing the α's to small values, before reaching a similar pattern to that in Figure 6. This empirical finding supports our proposal that we should initialize $\alpha = 0$ even for shallow models.

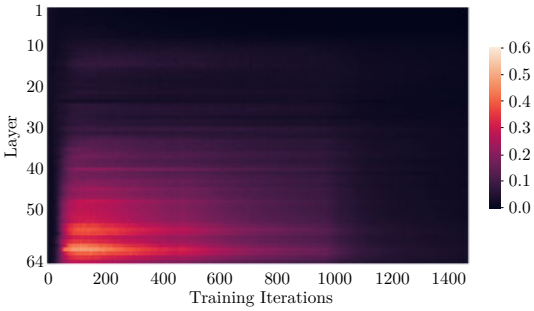


Figure 6: Heat map for residual weight $|\alpha_i|$ evolution during training for 64L ReZero Transformer.

7 Conclusion

We introduced ReZero, a simple architectural modification that facilitates signal propagation in deep networks and helps the network maintain dynamical isometry. Applying ReZero to various residual architectures – fully connected networks, Transformers and ResNets – we observed significantly improved convergence speeds. Furthermore, we were able to efficiently train Transformers with hundreds of layers, which has been difficult with the original architecture. We believe deeper Transformers will open the door to future exploration.

While training models with ReZero Transformers, we discovered interesting patterns in the values of residual weights of each layer $|\alpha_i|$ over the course of training. These patterns may hint towards some form of curriculum learning and allow for progressive stacking of layers to further accelerate training [38]. Patterns of residual weights can be crucial to understand the training dynamics of such deeper networks and might be important to model performance, which we will explore in future work.

Broader Impact

Recent work [39] has shown that increasing model capacity in terms of parameter count has a substantial impact on improving model performance. However, this increase in model capacity has also led to much longer training times and requires large GPU clusters in order to run experiments, which indirectly contributes to the carbon footprints generated from model training. In addition, [40] showed that along with carbon emission cost, training deep models draws significant economic costs. This makes it more difficult for less well-funded research labs and start-ups to effectively train state-of-the-art models. Our work enables faster convergence without trading off model performance and is in line with recent efforts [41] to make models more environment-friendly to train.

References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553), 2015.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[3] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *NIPS*, 2017.

[4] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

[5] Ben Poole, Subhaneil Lahiri, Maithra Raghu, Jascha Sohl-Dickstein, and Surya Ganguli. Exponential expressivity in deep neural networks through transient chaos. In *NIPS*, pages 3360–3368, 2016.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[7] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[8] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[9] Lechao Xiao, Yasaman Bahri, Jascha Sohl-Dickstein, Samuel S Schoenholz, and Jeffrey Pennington. Dynamical isometry and a mean field theory of cnns: How to train 10,000-layer vanilla convolutional neural networks. *arXiv preprint arXiv:1806.05393*, 2018.

[10] Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *arXiv preprint arXiv:1901.09321*, 2019.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[12] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[13] Jeffrey Pennington, Samuel Schoenholz, and Surya Ganguli. Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. In *NIPS*, 2017.

[14] Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. Character-level language modeling with deeper self-attention. In *AAAI*, volume 33, 2019.

[15] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *NIPS*, 2014.

[16] Jeffrey Pennington, Samuel S Schoenholz, and Surya Ganguli. The emergence of spectral universality in deep networks. *arXiv preprint arXiv:1802.09979*, 2018.

[17] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *NIPS*, 2010.

[18] Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

[19] Samuel S Schoenholz, Justin Gilmer, Surya Ganguli, and Jascha Sohl-Dickstein. Deep information propagation. *arXiv preprint arXiv:1611.01232*, 2016.

[20] Ge Yang and Samuel Schoenholz. Mean field residual networks: On the edge of chaos. In *NIPS*, 2017.

[21] Dar Gilboa, Bo Chang, Minmin Chen, Greg Yang, Samuel S Schoenholz, Ed H Chi, and Jeffrey Pennington. Dynamical isometry and a mean field theory of lstms and grus. *arXiv preprint arXiv:1901.08987*, 2019.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*. Springer, 2016.

[23] 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. *arXiv preprint arXiv:1706.02677*, 2017.

[24] Moritz Hardt and Tengyu Ma. Identity matters in deep learning. *arXiv preprint arXiv:1611.04231*, 2016.

[25] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of tricks for image classification with convolutional neural networks. In *CVPR*, 2019.

[26] Soham De and Samuel L Smith. Batch normalization biases deep residual networks towards shallow paths. *arXiv preprint arXiv:2002.10444*, 2020.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[28] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul), 2011.

[29] Pedro HP Savarese, Leonardo O Mazza, and Daniel R Figueiredo. Learning identity mappings with residual gates. *arXiv preprint arXiv:1611.01260*, 2016.

[30] Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, volume 11006, page 1100612. International Society for Optics and Photonics, 2019.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *NAACL*, 2019.

[32] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *ICLR*, 2017.

[33] Matt Mahoney. Large text compression benchmark, 2009.

[34] Toan Q. Nguyen and Julian Salazar. Transformers without tears: Improving the normalization of self-attention. *CoRR*, abs/1910.05895, 2019.

[35] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[36] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture. *arXiv preprint arXiv:2002.04745*, 2020.

[37] Microsoft. Turing-nlg: A 17-billion-parameter language model, 2020.

[38] Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tie-Yan Liu. Efficient training of BERT by progressively stacking. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *ICML*, volume 97 of *Proceedings of Machine Learning Research*, 2019.

[39] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher 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. 2020.

[40] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*. ACL, 2019.

[41] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *CoRR*, abs/1907.10597, 2019.

[42] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[43] Yang You, Jing Li, Jonathan Hseu, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. Reducing BERT pre-training time from 3 days to 76 minutes. *CoRR*, abs/1904.00962, 2019.

[44] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8), 1997.

A Vanishing singular values in Transformers

Two crucial components relevant to the signal propagation in the Transformer include LayerNorm [12] and (multi-head) self attention [27]. In this section, we argue that neither component by itself or in conjunction with a vanilla residual connection can satisfy dynamical isometry for all input signals.

A.1 LayerNorm

Layer normalization removes the mean and scales the variance over all neurons of a given layer and introduces learnable parameters γ and β to re-scale the variance and shift the mean according to

$$\text{LayerNorm}(\boldsymbol{x}) = \frac{\boldsymbol{x} - \text{E}(\boldsymbol{x})}{\sqrt{\text{Var}(x)}} \times \gamma + \beta \,. \tag{11}$$

It is clear from this definition that perturbing an input x by a transformation that purely shifts either its mean or variance will leave the output unchanged. These perturbations, therefore, give rise to two vanishing singular values of the input-output Jacobian. In the Transformer architecture [27], the norm is applied to each of the n elements of the input sentence, leading to a total of $2 \times n$ vanishing singular values of the Jacobian for each Transformer layer.

A.2 Self-Attention

Self-attention allows the model to relate content located across different positions by computing a weighted sum of an input sequence. Specifically, the $n \times d$ matrix $\boldsymbol{x}$ contains an input sequence of n rows containing d-dimensional embedding vectors, from which we can evaluate the query, key and value matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} = \boldsymbol{x} \cdot \boldsymbol{W}^{Q,K,V}$, where the $W^{Q,K,V}$ matrices are $d \times d$. The scaled dot-product attention then is given by

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\boldsymbol{Q} \cdot \boldsymbol{K}^\top / \sqrt{d}\right) \cdot \boldsymbol{V} \,. \tag{12}$$

In general, the singular value spectrum of the Jacobian of this attention process is complicated. Rather than studying it in full generality, we now merely argue that for some inputs $\boldsymbol{x}$ and weights $\boldsymbol{W}^{Q,K,V}$ the Jacobian has a large number of vanishing singular values (a claim we evaluate empirically in the paper). Consider weights or inputs such that each of the arguments of the softmax function is small compared to 1. The softmax function then simply returns a $n \times n$ dimensional matrix filled with entries that all approximate $1/n$. This means that the attention function projects all embedding vectors of the input sequence onto a single diagonal direction. This implies that out of the $n \times d$ Jacobian singular values only d are non-vanishing and hence much of the input signal is lost. A residual connection can restore some of the lost signals, but even then some perturbations are amplified while others are attenuated. This example demonstrates that self-attention is incompatible with dynamical isometry and unimpeded signal propagation in deep Transformer networks. It is easy to verify that the same conclusion holds for multi-head attention. A careful initialization of the weights might alleviate some of these issues, but we are not aware of any initialization scheme that would render a Transformer layer consistent with dynamical isometry.

B Convergence speed experimental hyperparameters

For all model variants in Section 6.2, we control the batch size to be 1080, number of layers to 12, feed-forward and attention dropout to 20%, hidden and embedding size to 512 units, context length to 512, the attention heads to 2, and GELU [42] activation in the point-wise feed-forward layer. To accommodate large batch training we use the LAMB optimizer [43] with a fixed learning rate of 0.016. Although learning rate schedules tend to improve performance [31], we omit them to simplify our training process. Training is performed on 8x V100 GPUs for at most a few days.

C Deep Transformers experimental hyperparameters

In Section 6.3, in order to examine whether our approach scales to deeper Transformers, we scale our 12 layer ReZero Transformer from Section 6.2 to 64 layers and 128 layers and compare it against the vanilla Transformer (*Post-Norm*). Due to memory constraints, we decreased the hidden size from 512 to 256 and reduced batch size to 304 and 144 for the 64 layer and 128 layer model respectively. Following guidelines from [43] we also adjusted the learning rate according to $0.0005 \times \sqrt{\text{batch size}}$. For all models in our experiments we limit training to a maximum of 100 training epochs. Training is performed on 8x V100 GPUs for at most a few days.

D Review of residual gates for deep signal propagation

In this section we give a chronological but light review of residual gates that are designed to preserve signals as they propagate deep into neural networks.

D.1 Highway Networks

Highway Networks [7], based on ideas from LSTM [44], were the first feedforward neural networks with hundreds of layers. This architecture employs gating units that learn to regulate signal flow through the network. Specifically, the authors define transform and carry gates $T[\boldsymbol{W}_T](\boldsymbol{x})$ and $C[\boldsymbol{W}_C](\boldsymbol{x})$, with weights $\boldsymbol{W}_{T,C}$ that act explicitly non-linearly on the signal $\boldsymbol{x}$. When combined with some block $F(\boldsymbol{x}_i)$ of a deep network, this gives the transformation

$$\boldsymbol{x}_{i+1} = C[\boldsymbol{W}_C](\boldsymbol{x}) \cdot \boldsymbol{x}_i + T[\boldsymbol{W}_T](\boldsymbol{x}) \cdot F(\boldsymbol{x}_i) \,. \tag{13}$$

The authors further propose to simplify the architecture according to $C \equiv 1 - T$, and using a simple Transform gate of the form $T[\boldsymbol{W}_T](\boldsymbol{x}) \equiv \sigma(\boldsymbol{W}_T^\top \cdot \boldsymbol{x} + \boldsymbol{b}_T)$, where σ denotes some activation function. The bias is initialized to be negative, as to bias the network towards carry behavior, e.g., C, but the network is not initialized as the identity map.

The proposal of Highway Networks lead to Gated ResNet [29], in which there exists a single additional parameter that parametrizes the gates as $\boldsymbol{W}_T = \boldsymbol{0}$, $\boldsymbol{b}_T = \alpha$, $C = 1 - T$.

Any feed-forward network could be written in the form (13), and ReZero corresponds to the simple choice $\boldsymbol{W}_T = \boldsymbol{W}_C = \boldsymbol{0}$, $\boldsymbol{b}_T = \alpha$, $\boldsymbol{b}_C = 1$. In contrast to Highway Networks, in ReZero the gates are independent of the input signal. We compare the performance of Gated ResNets to ReZero ResNets in Section 5.

D.2 ResNets

ResNets [2] introduced the simple residual connection

$$\boldsymbol{x}_{i+1} = \sigma\left(\boldsymbol{x}_i + F(\boldsymbol{x}_i)\right) , \tag{14}$$

that has been extremely successful in training deep networks. Just as Highway Networks, these residual connections are not initialized to give the identity map.

D.3 Pre-activation ResNets

Soon after the introduction of ResNets it was realized in [22] that applying the activation function σ prior to the residual connection allows for better performance. Schematically, we have the pre-activation connection

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + F(\boldsymbol{x}_i) , \tag{15}$$

where we absorbed the activation function into the block $F(\boldsymbol{x}_i)$. This finding of improved performance is consistent with improved signal propagation, since the residual connection is not modulated by the activation function.

D.4 Zero γ

Residual networks often contain normalization layers in which the signal is rescaled by learnable parameters γ which is referred to the Zero γ [23, 24, 25]. If the last element before a residual connection happens to be a normalization layer, then initializing these γ to zero has been found to improve convergence speed and accuracy. This method is in spirit very similar to the ReZero architecture. However, it potentially zero-initializes many parameters for each block, and is only applicable when a normalization layer exists.

D.5 FixUp

FixUp initialization [10] carefully rescales the initialization scheme in order to avoid vanishing or exploding gradients, without the use of normalization techniques. In particular, this scheme is implemented via the following Rules (verbatim from [10]):

1. Initialize the classification layer and the last layer of each residual branch to 0.
2. Initialize every other layer using a standard method (e.g., He et al. [6]), and scale only the weight layers inside residual branches by $L^{-1/(2m-2)}$.
3. Add a scalar multiplier (initialized at 1) in every branch and a scalar bias (initialized at 0) before each convolution, linear, and element-wise activation layer.

The authors emphasize that Rule 2 is the essential part. ReZero is simpler and similar to the first part of Rule 3, but the initialization differs.

D.6 SkipInit

[26] proposes to replace BatchNorm layers with a single scalar initialized at a small value. SkipInit is only applicable when normalization layers exist.

D.7 ReZero

ReZero is the simplest iteration achieving the goal of deep signal propagation. Schematically, the ReZero architecture is

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i F(\boldsymbol{x}_i)\,. \tag{16}$$

The rule to implement ReZero is

1. For every block add a scalar multiplier α (initialized at 0) and a residual connection.

E CIFAR-10 experiments

In this section we briefly describe the hyperparameters used in the image recognition experiments performed in §5. For all these experiments we used PyTorch version 1.2.0 (we have observed some inconsistencies in results with other PyTorch versions that may be due to different default initializations). CIFAR10 experiments tend to take less than an hour on a single RTX 2080TI GPU.

E.1 Step-down schedule

In Table 1 we compare the inference accuracy of different network architectures after training with identical hyperparameters a learning-rate schedule that decreases in three steps, as in [2]. The initial learning rate is 0.1 and decreases by a factor of 10 at 100 and 150 epochs. The models are trained for a total of 200 epochs. We use the SGD optimizer with a batch size of 128, weight decay of 5×10^{-4} and momentum 0.9.

E.2 Superconvergence schedule

To demonstrate superconvergence we use a one-cycle learning rate schedule, as described in [30] and closely follow the setup by Fast AI referenced in the text. In particular, the learning rate of the SGD optimizer evolves as follows over 45 epochs. The initial learning rate is 0.032 and linearly increases with each iteration to reach 1.2 after 10% of the total number of iterations has been reached. Then, the learning rate linearly decreases to return to 0.032 when 90% of the total steps. Thereafter, the learning rate linearly decays to a final value of 0.001 at the end of training. The SGD momentum varies between 0.85 and 0.95, mirroring the triangular learning rate, as is standard for the one-cycle policy in this setup [30]. Weight decay is set to 2×10^{-4} and the batch size is 512.

The residual weights cannot tolerate the extremely large learning rates required for the super-convergence phenomenon. For this reason we keep the learning rate of the residual weights at 0.1 throughout training.

F Datasets

We note that for all our experiments, we follow the official training, validation and test splits of the respective datasets.

On Empirical Comparisons of Optimizers for Deep Learning

Dami Choi [1 2] **Christopher J. Shallue** [1] **Zachary Nado** [1] **Jaehoon Lee** [1] **Chris J. Maddison** [3 4] **George E. Dahl** [1]

Abstract

Selecting an optimizer is a central step in the contemporary deep learning pipeline. In this paper, we demonstrate the sensitivity of optimizer comparisons to the hyperparameter tuning protocol. Our findings suggest that the hyperparameter search space may be the single most important factor explaining the rankings obtained by recent empirical comparisons in the literature. In fact, we show that these results can be contradicted when hyperparameter search spaces are changed. As tuning effort grows without bound, more general optimizers should never underperform the ones they can approximate (i.e., Adam should never perform worse than momentum), but recent attempts to compare optimizers either assume these inclusion relationships are not practically relevant or restrict the hyperparameters in ways that break the inclusions. In our experiments, we find that inclusion relationships between optimizers matter in practice and always predict optimizer comparisons. In particular, we find that the popular adaptive gradient methods never underperform momentum or gradient descent. We also report practical tips around tuning often ignored hyperparameters of adaptive gradient methods and raise concerns about fairly benchmarking optimizers for neural network training.

1. Introduction

The optimization algorithm chosen by a deep learning practitioner determines the training speed and the final predictive performance of their model. To date, there is no theory that adequately explains how to make this choice. Instead, our community relies on empirical studies (Wilson et al., 2017) and benchmarking (Schneider et al., 2019). Indeed, it is the de facto standard that papers introducing new optimizers report extensive comparisons across a large number of workloads. Therefore, to maximize scientific progress, we must have confidence in our ability to make empirical comparisons between optimization algorithms.

Although there is no theory guiding us when comparing optimizers, the popular first-order optimizers form a natural inclusion hierarchy. For example, ADAM (Kingma & Ba, 2015) and RMSPROP (Tieleman & Hinton, 2012) can approximately simulate MOMENTUM (Polyak, 1964) if the ϵ term in the denominator of their parameter updates is allowed to grow very large. However, these relationships may not matter in practice. For example, the settings of ADAM's hyperparameters that allow it to match the performance of MOMENTUM may be too difficult to find (for instance, they may be infinite).

In this paper, we demonstrate two important and interrelated points about empirical comparisons of neural network optimizers. First, we show that inclusion relationships between optimizers actually matter in practice; in our experiments, more general optimizers *never* underperform special cases. Despite conventional wisdom (Wilson et al., 2017; Balles & Hennig, 2017), we find that when carefully tuned, ADAM and other adaptive gradient methods never underperform MOMENTUM or SGD. Second, we demonstrate the sensitivity of optimizer comparisons to the hyperparameter tuning protocol. By comparing to previous experimental evaluations, we show how easy it is to change optimizer rankings on a given workload (model and dataset pair) by changing the hyperparameter tuning protocol, with optimizer rankings stabilizing according to inclusion relationships as we spend more and more effort tuning. Our findings raise serious questions about the practical relevance of conclusions drawn from these sorts of empirical comparisons.

The remainder of this paper is structured as follows. In Section 2, we review related work, focusing on papers that make explicit claims about optimizer comparisons in deep learning and application papers that provide evidence about the tuning protocols of practitioners. We develop our definition of first-order optimizers in Section 3 along with a notion of inclusion relationships between optimizers. We present our experimental results in Section 4. Despite thorny methodological issues over how to avoid biases in comparisons due to search spaces that favor one optimizer

[1]Google Research, Brain Team [2]Vector Institute and University of Toronto [3]DeepMind [4]Institute for Advanced Study, Princeton NJ. Correspondence to: Dami Choi <choidami@cs.toronto.edu>.

over another, we believe that our experimental methodology is an acceptable compromise and has substantial practical relevance. Among other results, we show that the inclusion hierarchy of update rules is almost entirely predictive of optimizer comparisons. In particular, NADAM (Dozat, 2016) achieves the best top-1 validation accuracy on ResNet-50 on ImageNet in our experiments. The 77.1% we obtain with NADAM, although not as good as the 77.6% obtained using learned data augmentation by Cubuk et al. (2018), is better than the best existing published results using any of the more standard pre-processing pipelines (76.5%, due to Goyal et al. (2017) using MOMENTUM).

2. Background and Related Work

Our work was inspired by the recent studies of neural network optimizers by Wilson et al. (2017) and Schneider et al. (2019). Wilson et al. (2017) constructed a simple classification problem in which adaptive gradient methods (e.g. ADAM) converge to provably worse solutions than standard gradient methods. However, crucially, their analysis ignored the ϵ parameter in the denominator of some adaptive gradient methods. Wilson et al. (2017) also presented experiments in which ADAM produced worse validation accuracy than SGD across *all* deep learning workloads considered. However, they only tuned over the learning rate and learning rate decay scheme in their experiments, leaving all other parameters of ADAM at fixed default values. Despite these findings, adaptive gradient methods continue to be popular since the work of Wilson et al. (2017). Schneider et al. (2019) presented a benchmark suite (DEEPOBS) for deep learning optimizers and reported that there was no single best optimizer across the workloads they considered. Yet Schneider et al. (2019) only tuned the learning rate of each optimizer and left all other hyperparameters at some fixed default values.

As we discuss in Section 4.3, the choices of hyperparameter tuning protocols in Wilson et al. (2017) and Schneider et al. (2019) may be the most important factor preventing their results from being relevant to practical choices about which optimizer to use. Hyperparameter tuning is a crucial step of the deep learning pipeline (Bergstra & Bengio, 2012; Snoek et al., 2012; Sutskever et al., 2013; Smith, 2018), so it is critical for papers studying optimizers to match as closely as possible the tuning protocols of an ideal practitioner. Yet, tuning protocols often differ between works studying neural network optimizers and works concerned with training neural networks to solve specific problems.

Recent papers that study or introduce optimization algorithms tend to compare to ADAM and RMSPROP without tuning their respective ϵ hyperparameters (see Table 1 for notations), presumably to simplify their experiments. It is standard to leave ϵ at the common default value of 10^{-8} for ADAM and 10^{-10} for RMSPROP (Tieleman & Hinton, 2012; Kingma & Ba, 2015; Dozat, 2016; Balles & Hennig, 2017; Loshchilov & Hutter, 2017; Zou & Shen, 2018; Ma & Yarats, 2018; Bernstein et al., 2018; Chen et al., 2019; Zou et al., 2019). Others do not even report the value of ϵ used (Balles & Hennig, 2017; Zhang & Mitliagkas, 2017; Keskar & Socher, 2017; Chen et al., 2018; Zhou et al., 2018; Aitchison, 2018; Reddi et al., 2019; Luo et al., 2019). There are exceptions. Zaheer et al. (2018) and Liu et al. (2019) considered ϵ values orders of magnitude larger than the standard default. However, the experiments in both papers gave only a limited consideration to ϵ, testing at most two values while tuning ADAM. De et al. (2018) is the only work we found that considered a broad range of values for ϵ. Both Zaheer et al. (2018) and De et al. (2018) found that non-default values of ϵ outperformed the default.

While it is also extremely common in applications to use a default value of ϵ, some notable papers tuned ϵ and selected values up to eight orders of magnitude away from the common defaults. Szegedy et al. (2016) used $\epsilon = 1$ for RMSPROP; Liu et al. (2019) reported that their results were sensitive to ϵ and set $\epsilon = 10^{-6}$ for ADAM; Tan et al. (2019) and Tan & Le (2019) set $\epsilon = 10^{-3}$ for RMSPROP, the latter achieving state-of-the-art ImageNet top-1 accuracy. In reinforcement learning, Hessel et al. (2017) set $\epsilon = 1.5 \times 10^{-4}$.

Despite being introduced solely to prevent division by zero[1] , ADAM's ϵ can be interpreted in ways that suggest the optimal choice is problem-dependent. If ADAM is interpreted as an empirical, diagonal approximation to natural gradient descent (Kingma & Ba, 2015), ϵ can be viewed as a multi-purpose damping term whose role is to improve the conditioning of the Fisher, in analogy to the approximate second-order method considered by Becker & Le Cun (1988). We can also view ϵ as setting a trust region radius (Martens & Grosse, 2015; Adolphs et al., 2019) and controlling an interpolation between momentum and diagonal natural gradient descent, by either diminishing or increasing the effect of v_t on the update direction. Under either interpretation, the best value for ϵ will be problem-dependent and likely benefit from tuning.

3. What is an optimizer?

Optimization algorithms are typically defined by their update rule, which is controlled by hyperparameters that determine its behavior (e.g. the learning rate). Consider a differentiable loss function $\ell : \mathbb{R}^d \to \mathbb{R}$ whose vector

[1]TensorFlow currently refers to ϵ as "a small constant for numerical stability"; https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/train/AdamOptimizer.

Table 1: Update rules considered in this work. SGD is due to (Robbins & Monro, 1951), MOMENTUM to (Polyak, 1964), NESTEROV to (Nesterov, 1983), RMSPROP to (Tieleman & Hinton, 2012), and NADAM to (Dozat, 2016). All operations are taken component-wise for vectors. In particular, for $x \in \mathbb{R}^d$, x^2 is a component-wise power function.

$\text{SGD}(H_t, \eta_t)$

$$\theta_{t+1} = \theta_t - \eta_t \nabla \ell(\theta_t)$$

$\text{MOMENTUM}(H_t, \eta_t, \gamma)$

$$v_0 = 0$$
$$v_{t+1} = \gamma v_t + \nabla \ell(\theta_t)$$
$$\theta_{t+1} = \theta_t - \eta_t v_{t+1}$$

$\text{NESTEROV}(H_t, \eta_t, \gamma)$

$$v_0 = 0$$
$$v_{t+1} = \gamma v_t + \nabla \ell(\theta_t)$$
$$\theta_{t+1} = \theta_t - \eta_t \left(\gamma v_{t+1} + \nabla \ell(\theta_t)\right)$$

$\text{RMSPROP}(H_t, \eta_t, \gamma, \rho, \epsilon)$

$$v_0 = 1, m_0 = 0$$
$$v_{t+1} = \rho v_t + (1-\rho)\nabla \ell(\theta_t)^2$$
$$m_{t+1} = \gamma m_t + \frac{\eta_t}{\sqrt{v_{t+1} + \epsilon}} \nabla \ell(\theta_t)$$
$$\theta_{t+1} = \theta_t - m_{t+1}$$

$\text{ADAM}(H_t, \alpha_t, \beta_1, \beta_2, \epsilon)$

$$m_0 = 0, v_0 = 0$$
$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla \ell(\theta_t)$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\nabla \ell(\theta_t)^2$$
$$b_{t+1} = \frac{\sqrt{1-\beta_2^{t+1}}}{1-\beta_1^{t+1}}$$
$$\theta_{t+1} = \theta_t - \alpha_t \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} b_{t+1}$$

$\text{NADAM}(H_t, \alpha_t, \beta_1, \beta_2, \epsilon)$

$$m_0 = 0, v_0 = 0$$
$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla \ell(\theta_t)$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\nabla \ell(\theta_t)^2$$
$$b_{t+1} = \frac{\sqrt{1-\beta_2^{t+1}}}{1-\beta_1^{t+1}}$$
$$\theta_{t+1} = \theta_t - \alpha_t \frac{\beta_1 m_{t+1} + (1-\beta_1)\nabla \ell(\theta_t)}{\sqrt{v_{t+1}} + \epsilon} b_{t+1}$$

of first partial derivatives is given by $\nabla \ell(\theta)$ (more generally, $\nabla \ell(\theta)$ might be a stochastic estimate of the true gradient). In our context, ℓ represents the loss function computed over an entire dataset by a neural network and $\theta \in \mathbb{R}^d$ represents the vector of model parameters. The optimization problem is to find a point that (at least locally) minimizes ℓ. First-order iterative methods for this problem (Nesterov, 2018) construct a sequence θ_t of iterates converging to a local minimum $\theta_\star$ using queries to ℓ and $\nabla \ell$. The sequence θ_t is constructed by an update rule $\mathcal{M}$, which determines the next iterate θ_{t+1} from the history H_t of previous iterates along with their function and gradient values, $H_t = \{\theta_s, \nabla \ell(\theta_s), \ell(\theta_s)\}_{s=0}^t$, and a setting of hyperparameters $\phi : \mathbb{N} \to \mathbb{R}^n$. Given an initial parameter value $\theta_0 \in \mathbb{R}^d$, the sequence of points visited by an optimizer with update rule $\mathcal{M}$ is given by,

$$\theta_{t+1} = \mathcal{M}(H_t, \phi_t).$$

The stochastic gradient descent algorithm (SGD; Robbins & Monro, 1951) is one of the simplest such methods used for training neural networks. SGD is initialized with $\theta_0 \in \mathbb{R}^d$, and its hyperparameter is a learning rate schedule $\eta : \mathbb{N} \to (0, \infty)$. The SGD update rule is given by $\text{SGD}(H_t, \eta_t) = \theta_t - \eta_t \nabla \ell(\theta_t)$. The MOMENTUM method due to Polyak (1964) generalizes the SGD method by linearly combining the gradient direction with a constant multiple of the previous parameter update. Its hyperparameters are a learning rate schedule $\eta : \mathbb{N} \to (0, \infty)$ and a momentum parameter $\gamma \in [0, \infty)$,

$$\text{MOMENTUM}(H_t, \eta_t, \gamma) = \theta_t - \eta_t \nabla \ell(\theta_t) + \gamma(\theta_t - \theta_{t-1}).$$

There has been an explosion of novel first-order methods in deep learning, all of which fall into this standard first-order scheme. In Table 1 we list the first-order update rules considered in this paper.

The difference between optimizers is entirely captured by the choice of update rule $\mathcal{M}$ and hyperparameters ϕ. Since the roles of optimizer hyperparameters on neural network loss functions are not well-understood, most practitioners tune a subset of the hyperparameters to maximize performance over a validation set, while leaving some hyperparameters at fixed default values. The choice of which hyperparameters to tune determines an *effective* family of update rules, and this family is the critical object from a practitioners perspective. Thus, in analogy to (overloaded) function declarations in C++, we define an *optimizer* by an update rule "signature," the update rule name together with the free hyperparameter arguments. For example, in this

definition MOMENTUM$(\cdot, \eta_t, \gamma)$ is not the same optimizer as MOMENTUM$(\cdot, \eta_t, 0.9)$, because the latter has two free hyperparameters while the former only has one. ADAM with the default ϵ is "different" from ADAM with tuned ϵ.

3.1. The taxonomy of first-order methods

The basic observation of this section is that some optimizers can approximately simulate others (i.e., optimizer A might be able to approximately simulate the trajectory of optimizer B for any particular setting of B's hyperparameters). This is important knowledge because, as a hyperparameter tuning protocol approaches optimality, a more expressive optimizer can never underperform any of its specializations. To capture the concept of one optimizer approximating another, we define the following inclusion relationship between optimizers.

Definition 1 (Inclusion relationship)**.** Let $\mathcal{M}, \mathcal{N}$ be update rules for use in a first-order optimization method. $\mathcal{M}$ is a subset or specialization of $\mathcal{N}$, if for all $\phi : \mathbb{N} \to \mathbb{R}^n$, there exists a sequence $\psi^i : \mathbb{N} \to \mathbb{R}^m$, such that for all $t \in [0, \infty)$ and histories H_t,

$$\lim_{i \to \infty} \mathcal{N}(H_t, \psi_t^i) = \mathcal{M}(H_t, \phi_t)$$

This is denoted $\mathcal{M} \subseteq \mathcal{N}$, with equality $\mathcal{M} = \mathcal{N}$ iff $\mathcal{M} \subseteq \mathcal{N}$ and $\mathcal{N} \subseteq \mathcal{M}$.

Evidently SGD $\subseteq$ MOMENTUM, since SGD$(H_t, \eta_t) =$ MOMENTUM$(H_t, \eta_t, 0)$. Many well-known optimizers fall naturally into this taxonomy. In particular, we consider RMSPROP with momentum (Tieleman & Hinton, 2012), ADAM (Kingma & Ba, 2015) and NADAM (Dozat, 2016) (see Table 1) and show the following inclusions in the appendix.

$$\text{SGD} \subseteq \text{MOMENTUM} \subseteq \text{RMSPROP}$$
$$\text{SGD} \subseteq \text{MOMENTUM} \subseteq \text{ADAM}$$
$$\text{SGD} \subseteq \text{NESTEROV} \subseteq \text{NADAM}$$

Note, some of these inclusions make use of the flexibility of hyperparameter schedules (dependence of ψ^i on t). In particular, to approximate MOMENTUM with ADAM, one needs to choose a learning rate schedule that accounts for ADAM's bias correction.

If two optimizers have an inclusion relationship, the more general optimizer can never be worse with respect to *any* metric of interest, provided the hyperparameters are sufficiently tuned to optimize that metric. Optimally-tuned MOMENTUM cannot underperform optimally-tuned SGD, because setting $\gamma = 0$ in MOMENTUM recovers SGD. However, optimizers with more hyperparameters might be more expensive to tune, so we should have a theoretical or experimental reason for using (or creating) a more general optimizer. For example, MOMENTUM improves local convergence rates over SGD on twice-differentiable functions that are smooth and strongly convex (Polyak, 1964), and NESTEROV has globally optimal convergence rates within the class of smooth and strongly convex functions (Nesterov, 1983; 2018).

At first glance, the taxonomy of optimizer inclusions appears to resolve many optimizer comparison questions. However, for a deep learning practitioner, there is no guarantee that the inclusion hierarchy is at all meaningful in practice. For example, the hyperparameters that allow ADAM to match or outperform MOMENTUM might not be easily accessible. They might exist only in the limit of very large values, or be so difficult to find that only practitioners with huge computational budgets can hope to discover them. Indeed, empirical studies and conventional wisdom hold that the inclusion hierarchy does not predict optimizer performance for many practical workloads (Wilson et al., 2017; Balles & Hennig, 2017; Schneider et al., 2019). Either these experimental investigations are too limited or the taxonomy of this section is of limited practical interest and provides no guidance about which optimizer to use on a real workload. In the following section we attempt to answer this question experimentally, and show that these inclusion relationships are meaningful in practice.

4. Experiments

An empirical comparison of optimizers should aim to inform a careful practitioner. Accordingly, we model our protocol on a practitioner that is allowed to vary all optimization hyperparameters for each optimizer (e.g. α_t, β_1, β_2, ϵ for ADAM) in addition to a parameterized learning rate decay schedule, in contrast to studies that fix a subset of the optimization hyperparameters to their default values (e.g. Wilson et al., 2017; Schneider et al., 2019). There is no standard method for selecting the values of these hyperparameters, but most practitioners tune at least a subset of the optimization hyperparameters by running a set of trials to maximize performance over the validation set. In our experiments, we run tens to hundreds of individual trials per workload. Given the variety of workloads we consider, this trial budget covers a wide range of computational budgets.

Selecting the hyperparameter search space for each optimizer is a key methodological choice for any empirical comparison of optimizers. Prior studies have attempted to treat each optimizer fairly by using the "same" search space for all optimizers (e.g. Wilson et al., 2017; Schneider et al., 2019). However, this requires the assumption that similarly-named hyperparameters should take similar values between optimizers, which is not generally true. For example, MOMENTUM and NESTEROV both have similar-looking momentum and learning rate hyperparameters,

Table 2: Summary of workloads used in experiments.

Task	Evaluation metric	Model	Dataset	Target error	Batch size	Budget
Image classification	Classification error	Simple CNN	Fashion MNIST	6.6%	256	10k steps
		ResNet-32	CIFAR-10	7%	256	50k steps
		CNN	CIFAR-100	–	256	350 epochs
		VGG-16	CIFAR-10	–	128	250 epochs
		ResNet-50	ImageNet	24%	1024	150k steps
Language modeling	Classification error	LSTM	War and Peace	–	50	200 epochs
	Cross entropy	Transformer	LM1B	3.45	256	750k steps

but NESTEROV tolerates larger values of its momentum (Sutskever et al., 2013). The situation worsens with less closely related optimizers: similarly-named hyperparameters could have totally different units, making it impossible to equate search volumes. Despite coming with its own set of challenges, it is most informative to compare optimizers assuming the practitioner is allowed to tune hyperparameters for different optimizers independently by way of optimizer-specific search spaces.

In our experiments, we chose the search space for each optimizer by running an initial set of experiments over a relatively large search space. In a typical case, we ran a single set of initial trials per optimizer to select the final search space. However, in some cases we chose one of the search spaces poorly, so we ran another set of experiments to select the final search space. We include these initial search spaces in Appendix D for the sake of transparency. The effort required to choose each search space cannot easily be quantified: our initial guesses were inevitably informed by prior experience with particular models and optimizers. This is true of all search spaces in the literature: tuning practices tend to be refined over many experiments and across many workloads, representing the sum total of our community's experience.

Following standard practice, we tuned the hyperparameters η_0, $1-\gamma$, $1-\beta_1$, and $1-\beta_2$ from Table 1 on a log scale by searching over the logarithms of their values. Additionally, we decoupled ϵ from the initial learning rate α_0 by searching over the logarithms of $(\epsilon, \alpha_0/\sqrt{\epsilon})$ for RMSPROP and $(\epsilon, \alpha_0/\epsilon)$ for ADAM and NADAM, instead of (ϵ, α_0). The fact that ϵ is coupled with the learning rate, with larger values of ϵ generally requiring larger learning rates, was shown for ADAM by Savarese et al. (2019). These search space transforms merely served to make our hyperparameter search more efficient; in principle, our results would be the same if we used a larger trial budget and naively searched all hyperparameters on a linear scale.

We validated our final search spaces by checking that that the optimal hyperparameter values were away from the search space boundaries for all optimizers in all experiments (see Figure 5 in Appendix E). We provide our final search spaces for all experiments in Appendix D. The fact that our final error rates compare favorably to prior published results – including reaching state-of-the-art for our particular configuration of ResNet-50 on ImageNet (see Section 4.2) – supports our claim that our methodology is highly competitive with expert tuning procedures.

4.1. Overview of Workloads and Experimental Details

We investigated the relative performance of optimizers across a variety of image classification and language modeling tasks. For image classification, we trained a simple convolutional neural network (Simple CNN) on Fashion MNIST (Xiao et al., 2017); ResNet-32 (He et al., 2016a) on CIFAR-10 (Krizhevsky, 2009); a CNN on CIFAR-100; VGG-16 (Simonyan & Zisserman, 2014) on CIFAR-10; and ResNet-50 on ImageNet (Russakovsky et al., 2015). For language modeling, we trained a 2-layer LSTM model (Hochreiter & Schmidhuber, 1997) on Tolstoy's *War and Peace*; and Transformer (Vaswani et al., 2017) on LM1B (Chelba et al., 2014). We used a linear learning rate decay schedule parameterized the same way as Shallue et al. (2019) for all workloads. We used a fixed batch size and a fixed budget of training steps for each workload independent of the optimizer. Table 2 summarizes these workloads and Appendix B provides the full details.

Given a search space, our tuning protocol sought to model a practitioner trying to achieve the best outcome with a fixed budget of trials (10, 50, or 100 depending on the workload).[2] A feasible trial is any trial that achieves finite train-

[2] In retrospect the best validation error across tuning trials converged quite quickly for our final search spaces, producing similar results with fewer than 20 trials in many cases. See Figures 6– 8 in Appendix E.

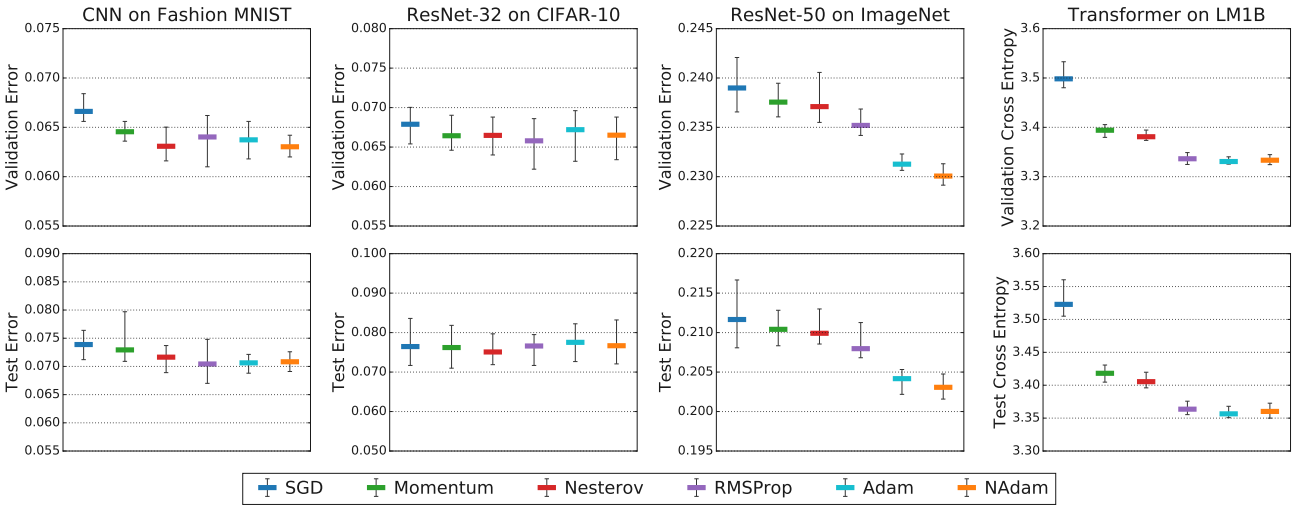


Figure 1: The relative performance of optimizers is consistent with the inclusion relationships, regardless of whether we compare final validation error (top) or test error (bottom). For all workloads, we tuned the hyperparameters of each optimizer separately, and selected the trial that achieved the lowest final validation error. Optimizers appear in the same order as the legend in all plots in this paper.

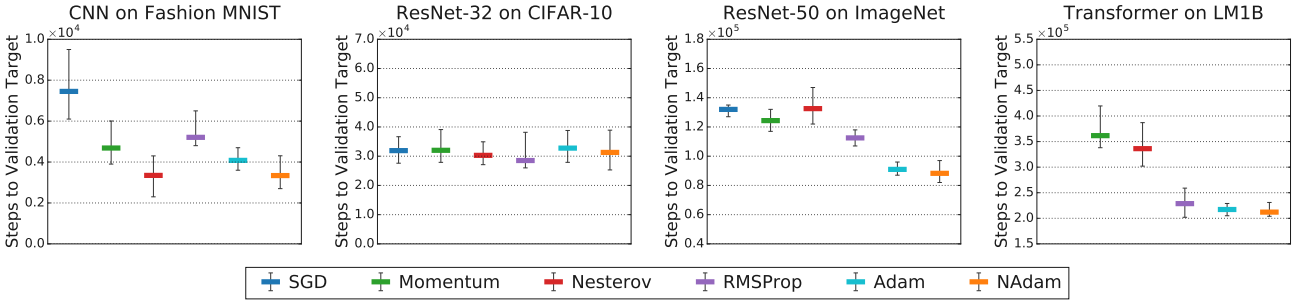


Figure 2: The relative training speed of optimizers is consistent with the inclusion relationships. We measured (idealized) training speed as the number of training steps required to reach a target validation error (see Table 2 for the error targets).

ing loss. We used quasi-random uniform search (Bousquet et al., 2017), and continued the search until we obtained a fixed number of feasible trials. From those trials we considered two statistics. The first, in order to characterize the best outcome, is a metric of interest (e.g. test accuracy) corresponding to the trial achieving the optimum of some other metric (e.g. validation accuracy). The second, in order to characterize the speed of training, is the number of steps required to reach a fixed validation target conditional on at least one trial in the search having reached that target. We chose the target for each workload based on initial experiments and known values from the literature (see Table 2). We estimated means and uncertainties using the bootstrap procedure described in Appendix C.

4.2. Inclusion relationships matter in practice

Figure 1 shows the final predictive performance of six optimizers on four different workloads after tuning hyperparameters to minimize validation error. Regardless of whether we compare final validation error or test error, the inclusion relationships hold in all cases – a more general optimizer never underperforms any of its specializations within the error bars. Similar results hold for training error (see Figure 9 in Appendix E). Training speed is also an important consideration, and Figure 2 demonstrates that the inclusion relationships also hold within error bars when we compare the number of steps required to reach a target validation error. Moreover, these results confirming the relevance of optimizer inclusion relationships do not depend on the exact step budgets or error targets we chose (see Figure 10 in Appendix E), although large changes to these values would require new experiments.

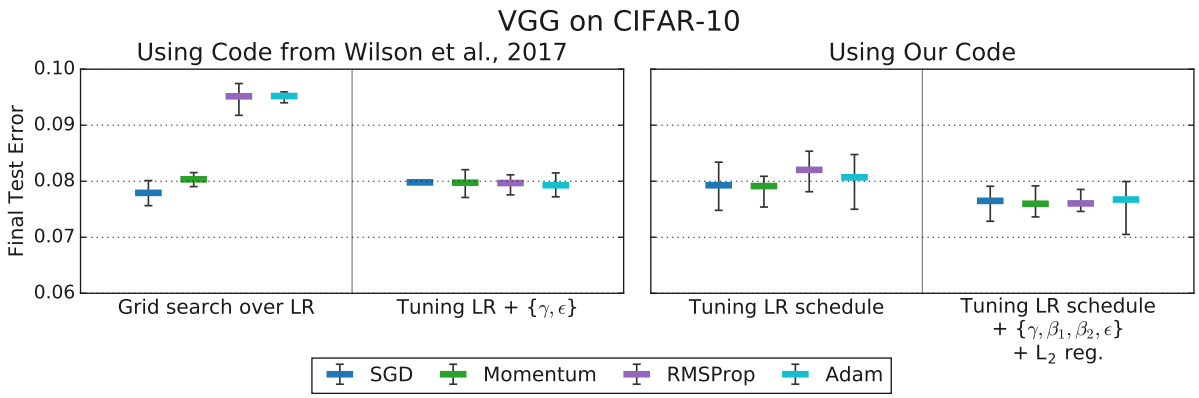


Figure 3: Tuning more hyperparameters removes the differences in test error between optimizers observed by Wilson et al. (2017). Tuning a subset of optimizer hyperparameters and the initial learning rate is sufficient to equalize performance between all optimizers (left). More extensive hyperparameter tuning in our setup, including the learning rate schedule, improves results for all optimizers and still does not produce any differences between optimizer performances (right).

Of course, just because a more general optimizer is no worse than any of its specializations doesn't mean the choice of optimizer makes a large difference on all workloads. For some workloads in Figures 1 and 2, all optimizers perform about the same, while other workloads have a clear ranking or even dramatic differences. For example, the choice of optimizer seems to make little difference for ResNet-32 on CIFAR-10; all optimizers achieve similar predictive performance and training speed. On the other hand, Transformer on LM1B exhibits a clear ranking in terms of predictive performance and training speed. For this workload, ADAM needs only 60% as many steps as MOMENTUM to reach our target error, and only 25% as many steps to get the same final result as SGD (see Figure 10 in the Appendix). These differences are clearly large enough to matter to a practitioner, and highlight the practical importance of choosing the right optimizer for some workloads.

The most general optimizers we considered were RMSPROP, ADAM, and NADAM, which do not include each other as special cases, and whose relative performance is not predicted by inclusion relationships. Across the workloads we considered, none of these optimizers emerged as the clear winner, although ADAM and NADAM generally seemed to have an edge over RMSPROP. For all of these optimizers, we sometimes had to set the ϵ parameter orders of magnitude larger than the default value in order to get good results. In particular, we achieved a validation accuracy of 77.1% for ResNet-50 on ImageNet using NADAM with $\epsilon = 9475$, a result that exceeds the 76.5% achieved by Goyal et al. (2017) using MOMENTUM. Across just these 4 workloads, the range of the optimal values of the ϵ parameter spanned 10 orders of magnitude.

4.3. Reconciling disagreements with previous work

In order to confirm that differences in hyperparameter tuning protocols explain the differences between our conclusions and those of Wilson et al. (2017) and Schneider et al. (2019), we reproduced a representative subset of their results and then inverted, or at least collapsed, the ranking over optimizers just by expanding the hyperparameter search space.

The left pane of Figure 3 shows our experiments on VGG on CIFAR-10 using code released by Wilson et al. (2017). When we match their protocol and perform their grid search over the initial learning rate and no other tuning, we reproduce their original result showing worse test error for RMSPROP and ADAM. However, when we tune the momentum parameter and ϵ with random search, all four optimizers reach nearly identical test error rates.[3] With our learning rate schedule search space, merely tuning the learning rate schedule was enough to make all optimizers reach the same test error within error bars. When we additionally tuned the optimization hyperparameters and weight decay in our setup we also get similar results for all optimizers, removing any evidence the inclusion relationships might be violated in practice.

Figure 4 shows our results with different tuning protocols for a CNN on CIFAR-100 and an LSTM language model trained on *War and Peace* to match the experiments in Schneider et al. (2019). As reported by Schneider et al. (2019), if we only tune the learning rate without tuning the decay schedule or other optimizer hyperparameters, ADAM

[3] Wilson et al. (2017) selected trials to minimize the training loss and then report test set results. As Figure 3 shows, removing this somewhat non-standard choice and tuning on a validation set and reporting test set results does not change anything.

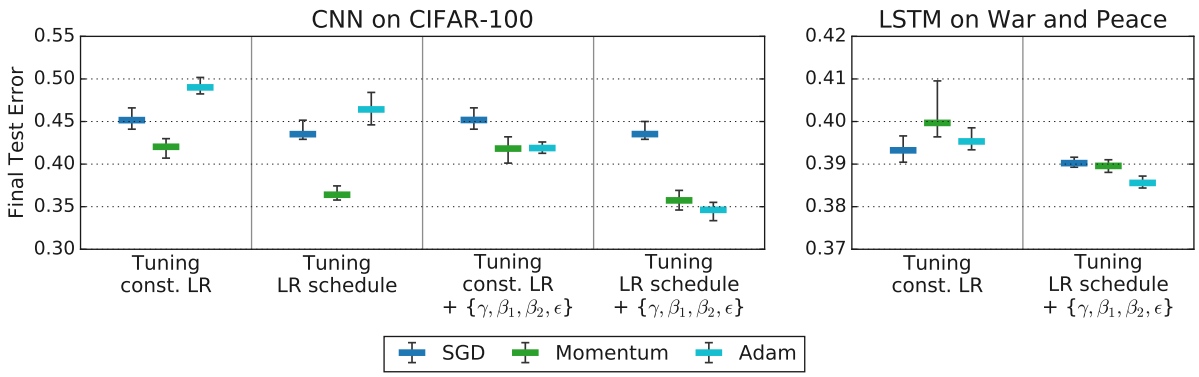


Figure 4: Tuning more hyperparameters changes optimizer rankings from Schneider et al. (2019) to rankings that are consistent with the inclusion relationships. The leftmost columns for each workload reproduce the rankings from Schneider et al. (2019), while the remaining columns tune over increasingly general search spaces. All columns use our random search tuning protocol.

does worse than MOMENTUM for the CNN and SGD performs slightly better than ADAM and MOMENTUM on the *War and Peace* dataset, although Schneider et al. (2019) found a larger advantage for SGD. However, once we tune the all the optimizer hyperparameters, ADAM does better than MOMENTUM which does better than SGD, as predicted by the inclusion relationships.

We conclude that the reason both Schneider et al. (2019) and Wilson et al. (2017) observed a ranking that, at first glance, contradicts the inclusion relationships is because they were not tuning enough of the hyperparameters. If we recast their results in our terminology where ADAM with default ϵ is a different optimizer than ADAM with ϵ tuned then there is no contradiction with our results and it becomes clear immediately that they do not consider the most interesting form of ADAM for practitioners.

5. Conclusions

Inspired by the recent efforts of Wilson et al. (2017) and Schneider et al. (2019), we set out to provide a detailed empirical characterization of the optimizer selection process in deep learning. Our central finding is that inclusion relationships between optimizers are meaningful in practice. When tuning all available hyperparameters under a realistic protocol at scales common in deep learning, we find that more general optimizers never underperform their special cases. In particular, we found that RMSPROP, ADAM, and NADAM never underperformed SGD, NESTEROV, or MOMENTUM under our most exhaustive tuning protocol. We did not find consistent trends when comparing optimizers that could not approximate each other. We also found workloads for which there was not a statistically significant separation in the optimizer ranking.

Our experiments have some important limitations and we should be careful not to overgeneralize from our results. The first major caveat is that we did not measure the effects of varying the batch size. Recent empirical work (Shallue et al., 2019; Zhang et al., 2019) has shown that increasing the batch size can increase the gaps between training times for different optimizers, with the gap from SGD to MOMENTUM (Shallue et al., 2019) and from MOMENTUM to ADAM (Zhang et al., 2019) increasing with the batch size. Nevertheless, we strongly suspect that the inclusion relations would be predictive at any batch size under a tuning protocol similar to the one we used. The second important caveat of our results is that they inevitably depend on the tuning protocol and workloads that we considered. Although we made every attempt to conduct realistic experiments, we should only expect our detailed findings to hold for similar workloads under similar protocols, namely uniform quasi-random tuning for tens to hundreds of trials, over hypercube search spaces, and with our specific learning rate schedule parameterization. Nevertheless, these caveats reinforce our central point: all empirical comparisons of neural network optimizers depend heavily on the hyperparameter tuning protocol, perhaps far more than we are used to with comparisons between model architectures.

If we were to extract "best practices" from our findings, then we suggest the following. If we can afford tens or more runs of our code, we should tune all of the hyperparameters of the popular adaptive gradient methods. Just because two hyperparameters have a similar role in two different update rules doesn't mean they should take similar values— optimization hyperparameters tend to be coupled and the optimal value for one may depend on how the others are set. Our results also confirm that the optimal value of Adam's ϵ is problem-dependent, so the onus is on empirical studies that fix $\epsilon = 10^{-8}$ to defend that choice. Finally, we should be skeptical of empirical comparisons of opti-

mizers in papers, especially if an optimizer underperforms any of its specializations. When we do inevitably compare optimizers, we should report search spaces and highlight decisions about what hyperparameters were tuned when interpreting results.

References

Adolphs, L., Kohler, J., and Lucchi, A. Ellipsoidal trust region methods and the marginal value of Hessian information for neural network training. *arXiv preprint arXiv:1905.09201*, 2019.

Aitchison, L. A unified theory of adaptive stochastic gradient descent as Bayesian filtering. *arXiv preprint arXiv:1807.07540*, 2018.

Balles, L. and Hennig, P. Dissecting Adam: The sign, magnitude and variance of stochastic gradients. *arXiv e-prints*, art. arXiv:1705.07774, May 2017.

Becker, S. and Le Cun, Y. Improving the convergence of the backpropagation learning with second order methods. *Morgan Koufmann, San Mateo, CA*, 1988.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

Bernstein, J., Wang, Y.-X., Azizzadenesheli, K., and Anandkumar, A. signSGD: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434*, 2018.

Bousquet, O., Gelly, S., Kurach, K., Teytaud, O., and Vincent, D. Critical hyper-parameters: no random, no cry. *arXiv preprint arXiv:1706.03200*, 2017.

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T. One billion word benchmark for measuring progress in statistical language modeling. In *Conference of the International Speech Communication Association*, 2014.

Chen, J., Zhao, L., Qiao, X., and Fu, Y. NAMSG: An efficient method for training neural networks. *arXiv preprint arXiv:1905.01422*, 2019.

Chen, X., Liu, S., Sun, R., and Hong, M. On the convergence of a class of Adam-type algorithms for non-convex optimization. *arXiv preprint arXiv:1808.02941*, 2018.

Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. Autoaugment: Learning augmentation policies from data. In *Conference on Computer Vision and Pattern Recognition*, 2018.

De, S., Mukherjee, A., and Ullah, E. Convergence guarantees for RMSProp and ADAM in non-convex optimization and an empirical comparison to Nesterov acceleration. *arXiv preprint arXiv:1807.06766*, 2018.

Dozat, T. Incorporating Nesterov momentum into Adam. In *ICLR Workshops*, 2016.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch SGD: training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition*, pp. 770–778. IEEE, 2016a.

He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pp. 630–645. Springer, 2016b.

Hessel, M., Modayil, J., and van Hasselt, H. Rainbow: Combining improvements in deep reinforcement learning. 2017. *arXiv preprint arXiv:1710.02298*, 2017.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Hoffer, E., Hubara, I., and Soudry, D. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pp. 1731–1741, 2017.

Ioffe, S. and Szegedy, C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pp. 448–456, 2015.

Keskar, N. S. and Socher, R. Improving generalization performance by switching from Adam to SGD. *arXiv preprint arXiv:1712.07628*, 2017.

Kingma, D. P. and Ba, J. Adam: a method for stochastic optimization. In *ICLR*, 2015.

Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., and Han, J. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. RoBERTa: a robustly optimized BERT pretraining approach. *arXiv e-prints*, art. arXiv:1907.11692, Jul 2019.

Loshchilov, I. and Hutter, F. Fixing weight decay regularization in Adam. *arXiv preprint arXiv:1711.05101*, 2017.

Luo, L., Xiong, Y., Liu, Y., and Sun, X. Adaptive gradient methods with dynamic bound of learning rate. *arXiv preprint arXiv:1902.09843*, 2019.

Ma, J. and Yarats, D. Quasi-hyperbolic momentum and Adam for deep learning. *arXiv preprint arXiv:1810.06801*, 2018.

Martens, J. and Grosse, R. Optimizing neural networks with Kronecker-factored approximate curvature. *arXiv preprint arXiv:1503.05671*, pp. 58, 2015.

Nesterov, Y. *Lectures on convex optimization*, volume 137. Springer, 2018.

Nesterov, Y. E. A method for solving the convex programming problem with convergence rate O(1/kˆ2). In *Dokl. akad. nauk Sssr*, volume 269, pp. 543–547, 1983.

Polyak, B. T. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

Reddi, S. J., Kale, S., and Kumar, S. On the convergence of Adam and beyond. In *ICLR*, 2019.

Robbins, H. and Monro, S. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3): 400–407, 1951.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3): 211–252, 2015.

Savarese, P., McAllester, D., Babu, S., and Maire, M. Domain-independent dominance of adaptive methods. *arXiv preprint arXiv:1912.01823*, 2019.

Schneider, F., Balles, L., and Hennig, P. DeepOBS: a deep learning optimizer benchmark suite. *arXiv preprint arXiv:1903.05499*, 2019.

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, 20(112):1–49, 2019.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Smith, L. N. A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay. *arXiv e-prints*, art. arXiv:1803.09820, Mar 2018.

Snoek, J., Larochelle, H., and Adams, R. P. Practical Bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pp. 2951–2959, USA, 2012. Curran Associates Inc.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. Striving for simplicity: the all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In *ICML*, 2013.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

Tan, M. and Le, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

Tieleman, T. and Hinton, G. Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems 30*, pp. 4148–4158. Curran Associates, Inc., 2017.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Zaheer, M., Reddi, S., Sachan, D., Kale, S., and Kumar, S. Adaptive methods for nonconvex optimization. In

Advances in Neural Information Processing Systems 31, pp. 9793–9803. Curran Associates, Inc., 2018.

Zhang, G., Li, L., Nado, Z., Martens, J., Sachdeva, S., Dahl, G. E., Shallue, C. J., and Grosse, R. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. *arXiv e-prints*, art. arXiv:1907.04164, Jul 2019.

Zhang, J. and Mitliagkas, I. Yellowfin and the art of momentum tuning. *arXiv preprint arXiv:1706.03471*, 2017.

Zhou, Z., Zhang, Q., Lu, G., Wang, H., Zhang, W., and Yu, Y. Adashift: Decorrelation and convergence of adaptive learning rate methods. *arXiv preprint arXiv:1810.00143*, 2018.

Zou, F. and Shen, L. On the convergence of weighted AdaGrad with momentum for training deep neural networks. *arXiv preprint arXiv:1808.03408*, 2018.

Zou, F., Shen, L., Jie, Z., Zhang, W., and Liu, W. A sufficient condition for convergences of Adam and RMSProp. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 11127–11135, 2019.

A. Optimizer inclusions

Table 1 summarizes the update rules for the optimizers we consider in this work. We assume update rules as implemented in TensorFlow r1.15. Here we prove their inclusion relationships, see Definition 1.

MOMENTUM can exactly implement SGD

MOMENTUM$(I_t, \eta_t, 0) = \text{SGD}(I_t, \eta_t)$, so SGD $\subseteq$ MOMENTUM.

NESTEROV can exactly implement SGD

NESTEROV$(I_t, \eta_t, 0) = \text{SGD}(I_t, \eta_t)$, so SGD $\subseteq$ NESTEROV.

RMSPROP with momentum can exactly implement MOMENTUM

Consider RMSPROP$(I_t, \eta_t, \gamma, \rho = 1, \epsilon = 0)$, so that

$$\begin{aligned} m_{t+1} &= \gamma m_t + \eta_t \nabla \ell(\theta_t)\,, \\ \theta_{t+1} &= \theta_t - m_{t+1}\,. \end{aligned}$$

This is equivalent to MOMENTUM, since

$$m_{t+1}^{(\text{RMSPROP})} \equiv \eta_t v_{t+1}^{(\text{MOMENTUM})}\,.$$

Thus RMSPROP$(I_t, \eta_t, \gamma, 1, 0) = $ MOMENTUM(I_t, η_t, γ), so MOMENTUM $\subseteq$ RMSPROP.

RMSTEROV can exactly implement NESTEROV

Consider RMSTEROV$(I_t, \eta_t, \gamma, \rho = 1, \epsilon = 0)$, so that

$$\begin{aligned} m_{t+1} &= \gamma m_t + \eta_t \nabla \ell(\theta_t)\,, \\ \theta_{t+1} &= \theta_t - \left[\gamma m_{t+1} + \eta_t \nabla \ell(\theta_t)\right]\,. \end{aligned}$$

This is equivalent to MOMENTUM, since

$$m_{t+1}^{(\text{RMSPROP})} \equiv \eta_t v_{t+1}^{(\text{NESTEROV})}\,.$$

Thus RMSTEROV$(I_t, \eta_t, \gamma, 1, 0) = $ MOMENTUM(I_t, η_t, γ), so MOMENTUM $\subseteq$ RMSTEROV.

ADAM can approximate MOMENTUM for large ϵ

Consider ADAM$(I_t, \alpha_t = \epsilon\,\eta_t(1-\gamma^t), \beta_1 = \gamma, \beta_2 = 0, \epsilon)$, so that

$$\begin{aligned} m_{t+1} &= \gamma m_t + (1-\gamma)\nabla \ell(\theta_t)\,, \\ \theta_{t+1} &= \theta_t - \frac{\eta_t}{(1-\gamma)}\left[\frac{m_{t+1}}{|\nabla \ell(\theta_t)|/\epsilon + 1}\right]\,. \end{aligned}$$

If ϵ is large, so that $|\nabla \ell(\theta_t)|/\epsilon \ll 1$, then

$$\begin{aligned} m_{t+1} &= \gamma m_t + (1-\gamma)\nabla \ell(\theta_t)\,, \\ \theta_{t+1} &= \theta_t - \eta_t \frac{m_{t+1}}{1-\gamma}\,. \end{aligned}$$

This is equivalent to MOMENTUM, since

$$m_t^{(\text{ADAM})} \equiv (1-\gamma)\, v_t^{(\text{MOMENTUM})}.$$

Thus $\lim_{\epsilon\to\infty}$ ADAM$(I_t, \epsilon\,\eta_t(1-\gamma^t), \gamma, 0, \epsilon) = $ MOMENTUM(I_t, η_t, γ), so MOMENTUM $\subseteq$ ADAM

NADAM can approximate NESTEROV for large ϵ

Consider NADAM$(I_t, \alpha_t = \epsilon\,\eta_t(1-\gamma^t), \beta_1 = \gamma, \beta_2 = 0, \epsilon)$, so that

$$\begin{aligned} m_{t+1} &= \gamma m_t + (1-\gamma)\nabla \ell(\theta_t)\,, \\ \theta_{t+1} &= \theta_t - \frac{\eta_t}{(1-\gamma)}\left[\frac{\gamma m_{t+1} + (1-\gamma)\nabla \ell(\theta_t)}{|\nabla \ell(\theta_t)|/\epsilon + 1}\right]\,. \end{aligned}$$

If ϵ is large, so that $|\nabla \ell(\theta_t)|/\epsilon \ll 1$, then

$$m_{t+1} = \gamma m_t + (1-\gamma)\nabla \ell(\theta_t)\,,$$
$$\theta_{t+1} = \theta_t - \eta_t \left[\frac{\gamma m_{t+1}}{1-\gamma} + \nabla \ell(\theta_t)\right].$$

This is equivalent to NESTEROV, since

$$m_t^{(\text{NADAM})} \equiv (1-\gamma)\, v_t^{(\text{NESTEROV})}$$

Thus $\lim_{\epsilon\to\infty} \text{NADAM}(I_t, \epsilon\,\eta_t(1-\gamma^t), \gamma, 0, \epsilon) = \text{NESTEROV}(I_t, \eta_t, \gamma)$, so NESTEROV $\subseteq$ NADAM.

B. Workload details

This section details the datasets and models summarized in Table 2.

B.1. Dataset Descriptions

For Fashion MNIST, CIFAR-10, ImageNet, and LM1B, our setup was identical to Shallue et al. (2019) except for the image pre-processing details described below. For *War and Peace*, our setup was identical to the "Tolstoi" dataset of Schneider et al. (2019).

CIFAR-10/100: We pre-processed images by subtracting the average value across all pixels and channels and dividing by the standard deviation.[4] For experiments with the ResNet-32 and CNN models, we followed the standard data augmentation scheme used in He et al. (2016a): 4 pixels padded on each side with single random crop from padded image or its horizontal reflection. We did not use random cropping for experiments with VGG for consistency with Wilson et al. (2017).

ImageNet: We augmented images at training time by resizing each image, taking a random crop of 224×224 pixels, randomly horizontally reflecting the cropped images, and randomly distorting the image colors. At evaluation time, we performed a single central crop of 224×224 pixels. In both training and evaluation, we then subtracted the global mean RGB value from each pixel using the values computed by Simonyan & Zisserman (2014).[5]

B.2. Model Descriptions

Simple CNN is identical to the base model described in Shallue et al. (2019). It consists of 2 convolutional layers with max pooling followed by 1 fully connected layer. The convolutional layers use 5×5 filters with stride 1, "same" padding, and ReLU activation function. Max pooling uses a 2×2 window with stride 2. Convolutional layers have 32 and 64 filters each and the fully connected layer has 1024 units. It does not use batch normalization.

CNN is the "All-CNN-C" model from Springenberg et al. (2014), as used in Schneider et al. (2019). The model consists of 3 convolutional layer blocks with max pooling. The convolutional layers use 5×5 filters with stride 1, "same" padding, and ReLU activation function. Max pooling uses a 2×2 window with stride 2. Convolutional layer blocks have 96, 192 and 192 filters each. As in Schneider et al. (2019), we used L_2 regularization of 5×10^{-4}.

ResNet is described in He et al. (2016a). We used the improved residual block described in He et al. (2016b). We used batch normalization (Ioffe & Szegedy, 2015) with exponential moving average (EMA) decay of 0.997 for ResNet-32, and ghost batch normalization (Hoffer et al., 2017) with ghost batch size of 32 and EMA decay of 0.9 for ResNet-50.

VGG is based on "model C" from Simonyan & Zisserman (2014). It consists of 13 convolutional layers followed by 3 fully connected hidden layers. We followed the modification used by Wilson et al. (2017) with batch normalization layers.

LSTM is a two hidden-layer LSTM model (Hochreiter & Schmidhuber, 1997) identical to the model used in Schneider et al. (2019). It uses 128 embedding dimensions and 128 hidden units.

Transformer is the "base" model described in (Vaswani et al., 2017). We used it as an autoregressive language model by applying the decoder directly to the sequence of word embeddings for each sentence. Unlike the default implementation,

[4] We used the TensorFlow op `tf.image.per_image_standardization`.

[5] See `https://gist.github.com/ksimonyan/211839e770f7b538e2d8#description` for the mean RGB values used.

we removed dropout regularization and used separate weight matrices for the input embedding layer and the pre-softmax linear transformation, as we observed these choices led to better performing models.

C. Estimating trial outcomes via bootstrap

Our tuning protocol corresponds to running trials with quasi-random hyperparameter values sampled uniformly from the search space until K feasible trials are obtained, with K depending on the workload. We then select the best trial, based on our statistic of interest, over those K trials.

We used the following bootstrap procedure to estimate means and uncertainties of our tuning protocol. We ran $N > K$ trials, with N depending on the workload. Then, for each bootstrap sample, we resampled the dataset of N trials with replacement and computed our statistic on the first K trials of the resampled dataset. We collected 100 such bootstrap samples each time, and from those computed the means, 5^{th} percentiles, and 95^{th} percentiles of the bootstrap distribution. We used this procedure to generate the means and error bars for each plot.

Simple CNN on Fashion MNIST used $(K, N) = (100, 500)$; ResNet-32 on CIFAR-10 used $(K, N) = (100, 500)$; ResNet-50 on ImageNet used $(K, N) = (50, 250)$; Transformer on LM1B used $(K, N) = (50, 250)$; VGG on CIFAR-10 with our code used $(K, N) = (50, 250)$ for tuning the learning rate schedule and $(K, N) = (100, 500)$ for tuning the learning rate schedule, $\{\gamma, \beta_1, \beta_2, \epsilon\}$, and L_2 regularization; CNN on CIFAR-100 used $(K, N) = (100, 500)$; LSTM on *War and Peace* used $(K, N) = (10, 50)$ for tuning just the learning rate and $(K, N) = (100, 500)$ for tuning the learning rate schedule and $\{\gamma, \beta_1, \beta_2, \epsilon\}$.

The sole exceptions to this bootstrap procedure are the two left panels of Figure 3, for which we used a similar procedure to Wilson et al. (2017) to ensure comparability. For each optimizer, we selected the trial that minimized validation error in our final search space and ran the same hyperparameter values 5 times, reporting the mean, minimum, and maximum test error over those 5 runs in Figure 3. This is slightly different to Wilson et al. (2017), who chose the trial that minimized training error and reported validation error. When tuning the learning rate and $\{\gamma, \epsilon\}$, we used 24 trials per optimizer in the initial search space (which we used to select the final search space), and 16 trials per optimizer in the final search space.

D. Hyperparameter Search Spaces

Below we report the search spaces used for our experiments. We include both the initial search spaces used to refine the search spaces, and the final spaces used to generate the plots. When only one search space was used, we denote the initial space as final. η_0, α_0, $1 - \gamma$, $1 - \beta_1$, $1 - \beta_2$, ϵ, and combinations thereof are always tuned on a log scale. The number of samples from each search space is specified in Appendix C.

D.1. CNN on Fashion MNIST

We used linear learning rate decay for all experiments. We tuned the number of decay steps within $[0.5, 1.0]$ times the number of training steps and the learning rate decay factor within $\{10^{-3}, 10^{-2}, 10^{-1}\}$. We did not use L_2 regularization or weight decay.

	η_0
initial	$[10^{-2}, 10^{2}]$
final	$[10^{-2}, 1]$

Table 3: SGD

	η_0	$1-\gamma$
initial	$[10^{-4}, 10^{2}]$	$[10^{-4}, 1]$
final	$[10^{-3}, 10^{-1}]$	$[10^{-3}, 1]$

Table 4: MOMENTUM

	η_0	$1-\gamma$
initial	$[10^{-4}, 10^{2}]$	$[10^{-4}, 1]$
final	$[10^{-3}, 10^{-1}]$	$[10^{-3}, 1]$

Table 5: NESTEROV

	$\eta_0/\sqrt{\epsilon}$	$1-\gamma$	$1-\rho$	ϵ
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-10}, 10^{10}]$
final	$[10^{-2}, 1]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-10}, 10^{-6}]$

Table 6: RMSPROP

	α_0/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ
initial	$[10^{-2}, 10^{-4}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-10}, 10^{10}]$
final	$[10^{-1}, 10^{1}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-6}, 10^{-2}]$

Table 7: ADAM

	α_0/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-10}, 10^{10}]$
final	$[10^{-1}, 10^{1}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-6}, 10^{-2}]$

Table 8: NADAM

D.2. ResNet-32 on CIFAR-10

We used linear learning rate decay for all experiments. We tuned the number of decay steps within $[0.5, 1.0]$ times the number of training steps and the learning rate decay factor f within the values shown in the tables below. λ_{L_2} denotes the L_2 regularization coefficient.

	η_0	λ_{L_2}	f
initial	$[10^{-2}, 10^{2}]$	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$
final	$[10^{-1}, 10^{1}]$	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$

Table 9: SGD

	η_0	$1-\gamma$	λ_{L_2}	f
initial	$[10^{-4}, 10^{2}]$	$[10^{-3}, 1]$	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$
final	$[10^{-2}, 1]$	$[10^{-3}, 1]$	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$

Table 10: MOMENTUM

	η_0	$1-\gamma$	λ_{L_2}	f
initial	$[10^{-4}, 10^{2}]$	$[10^{-4}, 10^{1}]$	10^{-4}	$\{10^{-3}, 10^{-2}, 10^{-1}\}$
final	$[10^{-2}, 1]$	$[10^{-3}, 1]$	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$

Table 11: NESTEROV

	$\eta_0/\sqrt{\epsilon}$	$1-\gamma$	$1-\rho$	ϵ	λ_{L_2}	f
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-10}, 10^{10}]$	10^{-4}	$\{10^{-3}, 10^{-2}, 10^{-1}\}$
final	$[10^{-2}, 1]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-4}, 1]$	$\{10^{-5}, 10^{-4}$ $10^{-3}, 10^{-2}\}$	$\{10^{-4}, 10^{-3}$ $10^{-2}, 10^{-1}\}$

Table 12: RMSPROP

	α_0/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ	λ_{L_2}	f
initial	$[10^{-4}, 10^{-1}]$	$[10^{-3}, 5\times 10^{-1}]$	$[10^{-4}, 10^{-1}]$	$[10^{-9}, 10^{-5}]$	10^{-4}	$\{10^{-3}, 10^{-2}$ $10^{-1}\}$
final	$[10^{-1}, 10^{1}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-3}, 10^{1}]$	$\{10^{-5}, 10^{-4}$ $10^{-3}, 10^{-2}\}$	$\{10^{-4}, 10^{-3}$ $10^{-2}, 10^{-1}\}$

Table 13: ADAM

	α_0/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ	λ_{L_2}	f
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-10}, 10^{10}]$	$\{10^{-5}, 10^{-4}$ $10^{-3}, 10^{-2}\}$	$\{10^{-4}, 10^{-3}$ $10^{-2}, 10^{-1}\}$
final	$[10^{-2}, 1]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[1, 10^{4}]$	$\{10^{-5}, 10^{-4}$ $10^{-3}, 10^{-2}\}$	$\{10^{-4}, 10^{-3}$ $10^{-2}, 10^{-1}\}$

Table 14: NADAM

D.3. ResNet-50 on ImageNet

We used linear learning rate decay for all experiments. We tuned the number of decay steps within $[0.5, 1.0]$ times the number of training steps and the learning rate decay factor f within the values shown in the tables below. λ_{wd} denotes the weight decay coefficient and τ denotes the label smoothing coefficient.

	η_0	λ_{wd}	τ	f
initial	$[10^{-2}, 10^{1}]$	$[10^{-5}, 10^{-2}]$	$\{0, 10^{-2}, 10^{-1}\}$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$
final	$[1, 10^{2}]$	$[10^{-4}, 10^{-3}]$	10^{-1}	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$

Table 15: SGD

	η_0	$1-\gamma$	λ_{wd}	τ	f
initial	$[10^{-3}, 1]$	$[10^{-3}, 1]$	$[10^{-5}, 10^{-2}]$	$\{0, 10^{-2}, 10^{-1}\}$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$
final	$[10^{-2}, 1]$	$[10^{-2}, 1]$	$[10^{-4}, 10^{-3}]$	10^{-2}	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$

Table 16: MOMENTUM

	η_0	$1-\gamma$	λ_{wd}	τ	f
initial	$[10^{-3}, 1]$	$[10^{-3}, 1]$	$[10^{-5}, 10^{-2}]$	$\{0, 10^{-2}, 10^{-1}\}$	10^{-3}
final	$[10^{-2}, 1]$	$[10^{-3}, 1]$	$[10^{-4}, 10^{-3}]$	0	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$

Table 17: NESTEROV

	$\eta_0/\sqrt{\epsilon}$	$1-\gamma$	$1-\rho$	ϵ	λ_{wd}	τ	f
initial	$[10^{-2}, 10^{4}]$	0.1	$[10^{-4}, 1]$	$[10^{-10}, 10^{10}]$	$[10^{-5}, 10^{-2}]$	$\{0, 10^{-2}, 10^{-1}\}$	10^{-3}
final	$[10^{-2}, 1]$	0.1	$[10^{-2}, 1]$	$[10^{-8}, 10^{-3}]$	$[10^{-4}, 10^{-3}]$	0	$\{10^{-4}, 10^{-3}$ $10^{-2}, 10^{-1}\}$

Table 18: RMSPROP

	α_0/ϵ	$1-\beta_1$	ϵ	λ_{wd}	τ	f
initial	$[1, 10^{2}]$	$[10^{-3}, 1]$	$[1, 10^{4}]$	$[10^{-5}, 10^{-3}]$	$\{0, 10^{-2}, 10^{-1}\}$	10^{-3}
final	$[1, 10^{2}]$	$[10^{-2}, 1]$	$[10^{-2}, 10^{2}]$	10^{-4}	10^{-1}	$\{10^{-4}, 10^{-3}$ $10^{-2}, 10^{-1}\}$

Table 19: ADAM

	α_0/ϵ	$1-\beta_1$	ϵ	λ_{wd}	τ	f
initial	$[10^{-1}, 10^{3}]$	$[10^{-3}, 1]$	$[10^{-2}, 10^{10}]$	$[10^{-5}, 10^{-2}]$	$\{0, 10^{-2}, 10^{-1}\}$	10^{-3}
final	$[1, 10^{2}]$	$[10^{-3}, 1]$	$[10^{3}, 10^{7}]$	10^{-4}	10^{-1}	10^{-3}

Table 20: NADAM

D.4. Transformer on LM1B

We used linear learning rate decay for all experiments. We tuned the number of decay steps within $[0.5, 1.0]$ times the number of training steps and the learning rate decay factor within $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$.

	η_0
initial	$[10^{-4}, 10^{-1}]$
final	$[10^{-3}, 10^{-1}]$

Table 21: SGD

initial	$[10^{-4}, 10^{-1}]$	$[10^{-4}, 1]$
final	$[10^{-4}, 10^{-2}]$	$[10^{-3}, 1]$

Table 22: MOMENTUM

	η_0	$1-\gamma$
initial	$[10^{-4}, 10^{-1}]$	$[10^{-4}, 1]$
final	$[10^{-4}, 10^{-2}]$	$[10^{-3}, 1]$

Table 23: NESTEROV

	$\eta_0/\sqrt{\epsilon}$	$1-\gamma$	$1-\rho$	ϵ
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-10}, 10^{10}]$
final	$[10^{-1}, 10^{1}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-9}, 10^{-5}]$

Table 24: RMSPROP

	α_0/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-10}, 10^{10}]$
final	$[10^{1}, 10^{3}]$	$[10^{-3}, 1]$	$[10^{-6}, 10^{-2}]$	$[10^{-7}, 10^{-3}]$

Table 25: ADAM

	α_0/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-4}, 1]$	$[10^{-10}, 10^{10}]$
final	$[1, 10^{2}]$	$[10^{-3}, 1]$	$[10^{-6}, 10^{-2}]$	$[10^{-6}, 10^{-2}]$

Table 26: NADAM

D.5. VGG on CIFAR-10 Using Code from Wilson et al. (2017)

D.5.1. Grid Search over Learning Rate

We tuned over the same grid of initial learning rate values for each optimizer as Wilson et al. (2017). As in Wilson et al. (2017), we decayed the initial learning rate by a factor of 0.5 every 25 epochs and used a fixed L_2 regularization coefficient of 0.0005.

D.5.2. Tuning Learning Rate & $\{\gamma, \epsilon\}$

We used our quasi-random tuning protocol to tune over the initial learning rate, Momentum's γ, RMSProp's ϵ, and Adam's ϵ. As in Wilson et al. (2017), we decayed the initial learning rate by a factor of 0.5 every 25 epochs and used a fixed L_2 regularization coefficient of 0.0005.

	η_0
initial	$[10^{-3}, 1]$
final	$[10^{-1}, 10^{1}]$

Table 27: SGD

	η_0	$1-\gamma$
initial	$[10^{-3}, 1]$	$[10^{-3}, 1]$
final	$[10^{-1}, 10^{1}]$	$[10^{-1}, 1]$

Table 28: Momentum

	ϵ	$\alpha_0/\sqrt{\epsilon}$
initial	$[10^{-10}, 10^{10}]$	$[10^{-2}, 10^{4}]$
final	$[10^{-2}, 10^{2}]$	$[10^{-1}, 10^{1}]$

Table 29: RMSProp

	ϵ	α_0/ϵ
initial	$[10^{-10}, 10^{10}]$	$[10^{-2}, 10^{4}]$
final	$[10^{6}, 10^{10}]$	$[10^{-1}, 10^{1}]$

Table 30: Adam

D.6. VGG on CIFAR-10 Using our Code

We used linear learning rate decay for all experiments. We tuned the number of decay steps within $[0.5, 1.0]$ times the number of training steps and the learning rate decay factor within $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$.

D.6.1. Tuning Learning Rate Schedule

We fixed all optimizer hyperparameters excluding the learning rate to match those specified in Wilson et al. (2017). As in Wilson et al. (2017), we used a fixed L_2 regularization coefficient of 0.0005.

	η_0 (SGD)	η_0 (Momentum)	η_0 (RMSProp)	α_0 (Adam)
initial	$[10^{-3}, 10^{1}]$	$[10^{-3}, 10^{1}]$	$[10^{-5}, 10^{-1}]$	$[10^{-5}, 10^{-1}]$
final	1.0	$[10^{-2}, 1]$	$[10^{-4}, 10^{-2}]$	$[10^{-5}, 10^{-1}]$

Table 31: Learning rate search ranges.

D.6.2. Tuning Learning Rate Schedule & $\{\gamma, \beta_1, \beta_2, \epsilon, \lambda_{L_2}\}$

	η_0	λ_{L_2}
initial	$[10^{-3}, 10^{1}]$	$[10^{-5}, 10^{-2}]$
final	$[10^{-2}, 1]$	$[10^{-3}, 10^{-1}]$

Table 32: SGD

	η_0	$1-\gamma$	λ_{L_2}
initial	$[10^{-3}, 10^{1}]$	$[10^{-3}, 1]$	$[10^{-5}, 10^{-2}]$
final	$[10^{-2}, 1]$	$[10^{-1}, 1]$	$[10^{-3}, 10^{-1}]$

Table 33: Momentum

	$\alpha_0/\sqrt{\epsilon}$	$1-\gamma$	$1-\rho$	ϵ	λ_{L_2}
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-3}, 1]$	$[10^{-10}, 10^{10}]$	$[10^{-5}, 10^{-2}]$
final	$[10^{-2}, 1]$	$[10^{-1}, 1]$	$[10^{-3}, 10^{-2}]$	$[10^{2}, 10^{6}]$	$[10^{-3}, 10^{-1}]$

Table 34: RMSProp

	α_0/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ	λ_{L_2}
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-4}, 10^{-1}]$	$[10^{-10}, 10^{10}]$	$[10^{-5}, 10^{-2}]$
final	$[10^{-2}, 10^{1}]$	$[10^{-1}, 1]$	$[10^{-4}, 10^{-1}]$	$[10^{6}, 10^{10}]$	$[10^{-3}, 10^{-1}]$

Table 35: Adam

D.7. CNN on CIFAR-100

D.7.1. Tuning Constant Learning Rate

We fixed all optimizer hyperparameters excluding the learning rate to match those specified in Schneider et al. (2019).

	η (SGD)	η (Momentum)	α (Adam)
initial	$[10^{-2}, 1]$	$[10^{-4}, 1]$	$[10^{-5}, 10^{-2}]$
final	$[10^{-1}, 1]$	$[10^{-3}, 10^{-2}]$	$[10^{-4}, 10^{-3}]$

Table 36: Learning rate search ranges.

D.7.2. Tuning Learning Rate Schedule

We used linear learning rate decay, and tuned the number of decay steps within $[0.5, 1.0]$ times the number of training steps and the learning rate decay factor within $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$.

	η_0 (SGD)	η_0 (Momentum)	α_0 (Adam)
initial	$[10^{-2}, 1]$	$[10^{-4}, 1]$	$[10^{-5}, 10^{-2}]$
final	$[10^{-1}, 1]$	$[10^{-3}, 10^{-1}]$	$[10^{-4}, 10^{-3}]$

Table 37: Learning rate search ranges.

D.7.3. Tuning Constant Learning Rate & $\{\gamma, \beta_1, \beta_2, \epsilon\}$

For SGD, we reused the results from Appendix D.7.1, since there were no additional hyperparameters to tune.

	η	$1 - \gamma$
final	$[10^{-4}, 1]$	$[10^{-3}, 1]$

Table 38: Momentum

	α/ϵ	$1 - \beta_1$	$1 - \beta_2$	ϵ
initial	$[10^{-2}, 10^{-2}]$	$[10^{-3}, 1]$	$[10^{-4}, 10^{-1}]$	$[10^{-10}, 10^{-10}]$
final	$[10^{-1}, 10^{-1}]$	$[10^{-2}, 1]$	$[10^{-4}, 10^{-1}]$	$[10^{2}, 10^{6}]$

Table 39: Adam

D.7.4. Tuning Learning Rate Schedule & $\{\gamma, \beta_1, \beta_2, \epsilon\}$

We used linear learning rate decay, and tuned the number of decay steps within $[0.5, 1.0]$ times the number of training steps and the learning rate decay factor within $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$. For SGD, we reused the results from Appendix D.7.2, since there were no additional hyperparameters to tune.

	η_0	$1 - \gamma$
initial	$[10^{-4}, 1]$	$[10^{-2}, 1]$
final	$[10^{-3}, 10^{-1}]$	$[10^{-3}, 10^{-1}]$

Table 40: Momentum

	α_0/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ
initial	$[10^{-1}, 10^{1}]$	$[10^{-3}, 1]$	$[10^{-4}, 10^{-1}]$	$[10^{2}, 10^{6}]$
final	$[10^{-1}, 10^{1}]$	$[10^{-3}, 1]$	$[10^{-5}, 10^{-2}]$	$[10^{2}, 10^{6}]$

Table 41: ADAM

D.8. LSTM on War and Peace

D.8.1. Tuning constant Learning Rate

	η
final	$[10^{-2}, 10^{1}]$

Table 42: SGD

	η	$1-\gamma$
final	$[10^{-4}, 1]$	0.99

Table 43: Momentum

	α/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ
final	$[10^{-5}, 10^{-2}]$	0.9	0.999	10^{-8}

Table 44: Adam

D.8.2. Tuning Learning Rate schedule & $\{\gamma, \beta_1, \beta_2, \epsilon\}$

We used linear learning rate decay, and tuned the number of decay steps within $[0.5, 1.0]$ times the number of training steps and the learning rate decay factor f within the values shown in the tables below.

	η_0	f
initial	$[10^{-3}, 10^{1}]$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$
final	$[1, 10^{1}]$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$

Table 45: SGD

	η_0	$1-\gamma$	f
initial	$[10^{-4}, 1]$	$[10^{-3}, 1]$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$
final	$[10^{-1}, 10^{1}]$	$[10^{-2}, 1]$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$

Table 46: Momentum

	α_0/ϵ	$1-\beta_1$	$1-\beta_2$	ϵ	f
initial	$[10^{-2}, 10^{4}]$	$[10^{-3}, 1]$	$[10^{-4}, 10^{-1}]$	$[10^{-10}, 10^{10}]$	$\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$
final	$[1, 10^{2}]$	$[10^{-2}, 1]$	0.999	$[1, 10^{4}]$	10^{-3}

Table 47: Adam

E. Additional plots

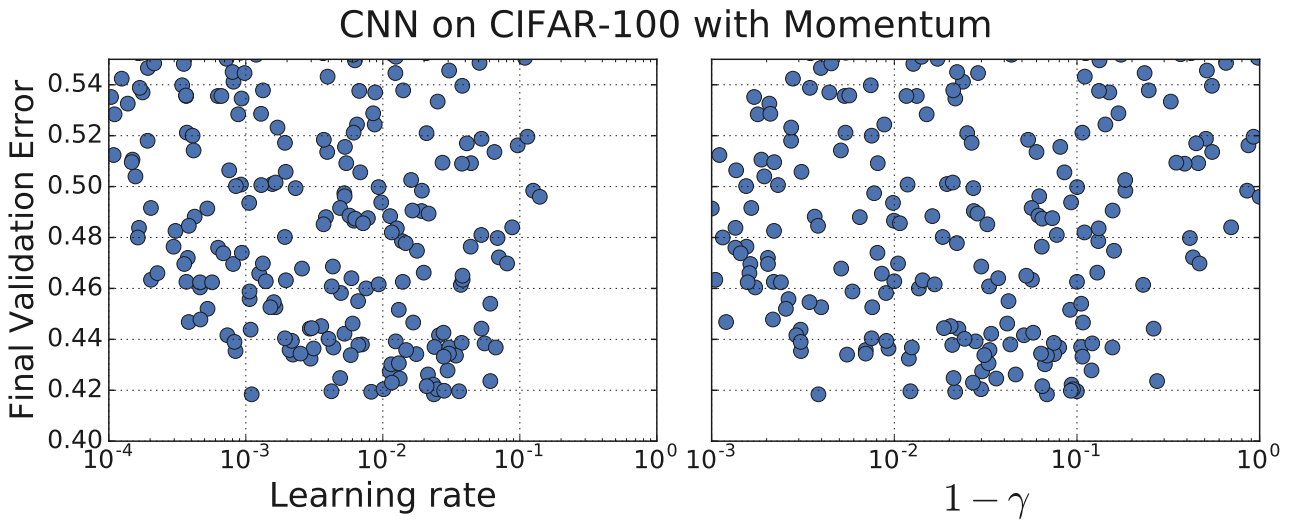


Figure 5: Example plot of final validation error projected onto the axes of the hyperparameter space. We consider this search space to be appropriate because the optimal values are away from the search space boundaries.

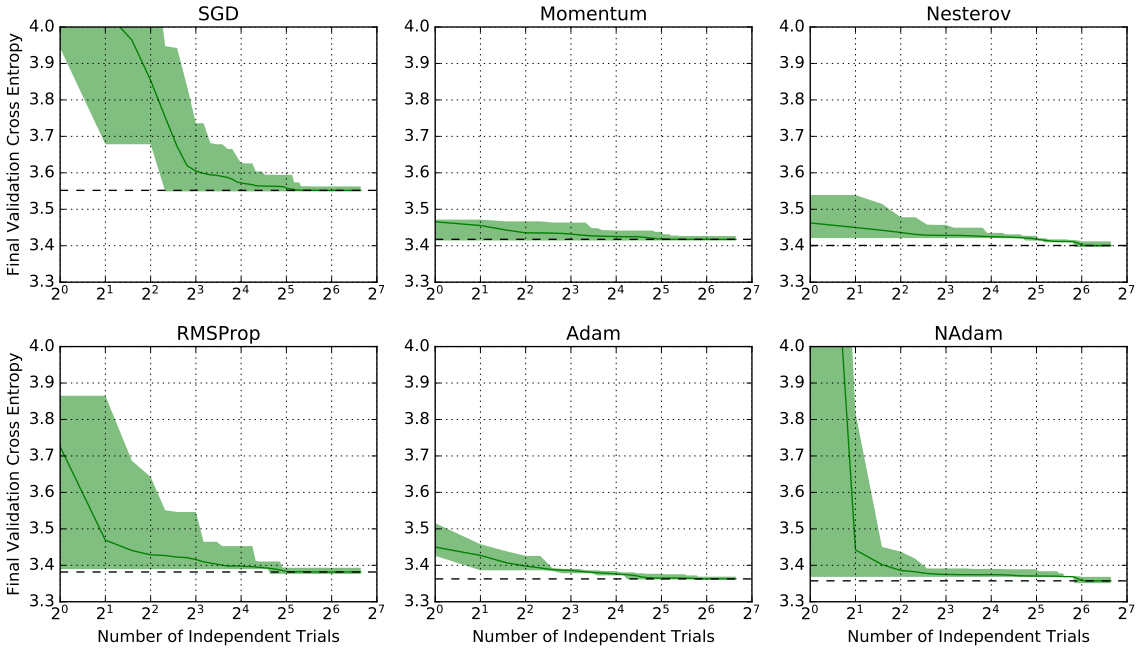


Figure 6: Validation performance of the best trial mostly converges with as few as 2^4 hyperparameter tuning trials for Transformer on LM1B. Shaded regions indicate 5^{th} and 95^{th} percentiles estimated with bootstrap sampling (see Appendix C). The search spaces can be found in Appendix D.4.

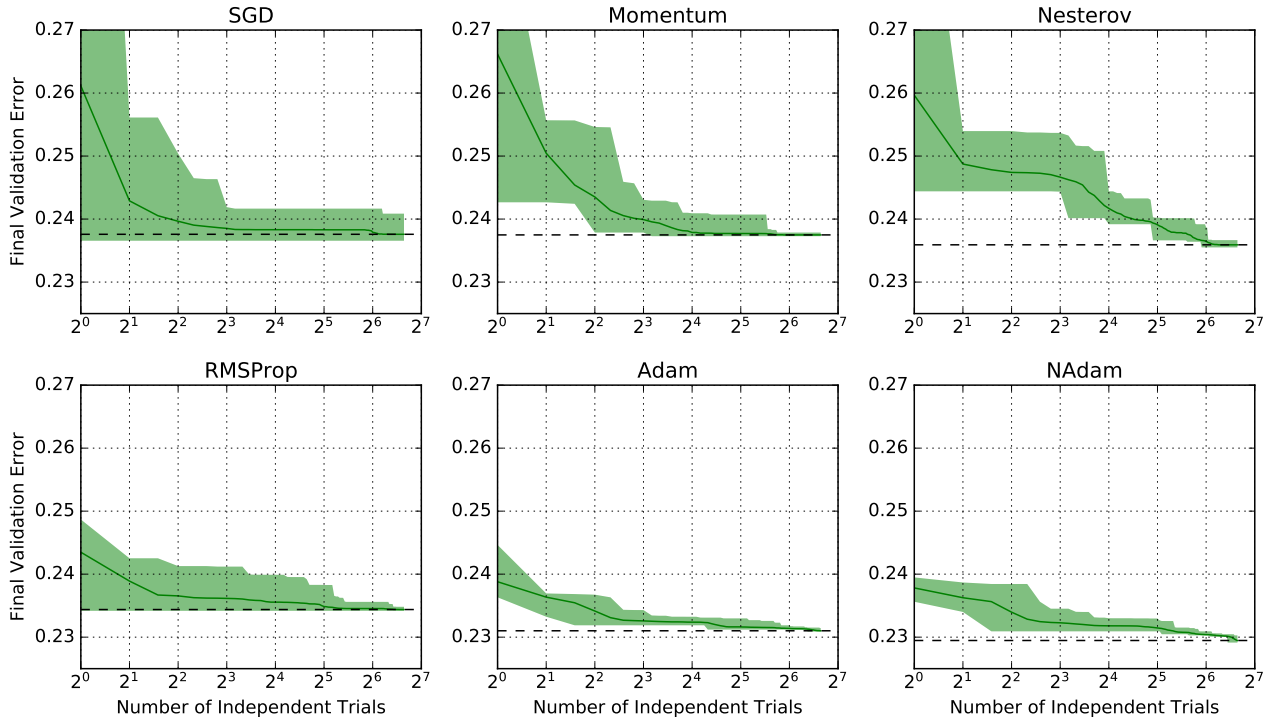


Figure 7: Validation performance of the best trial mostly converges with as few as 2^4 hyperparameter tuning trials for ResNet-50 in ImageNet. Shaded regions indicate 5^{th} and 95^{th} percentile estimated with bootstrap sampling (see Appendix C). The search spaces can be found in D.3.

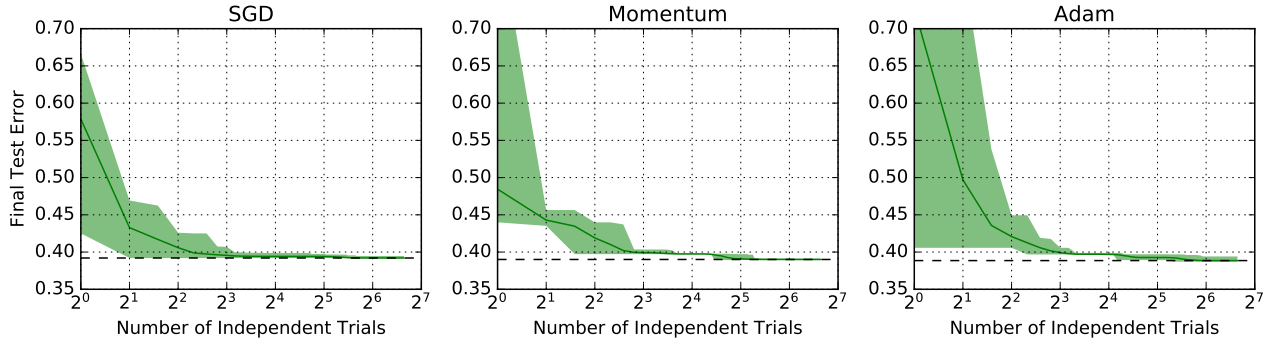


Figure 8: Test performance of the best trial mostly converges with as few as 2^3 hyperparameter tuning trials for a 2-layer LSTM on *War and Peace*. Shaded regions indicate 5^{th} and 95^{th} percentile estimated with bootstrap sampling (see Appendix C). The search spaces can be found in D.8.2.

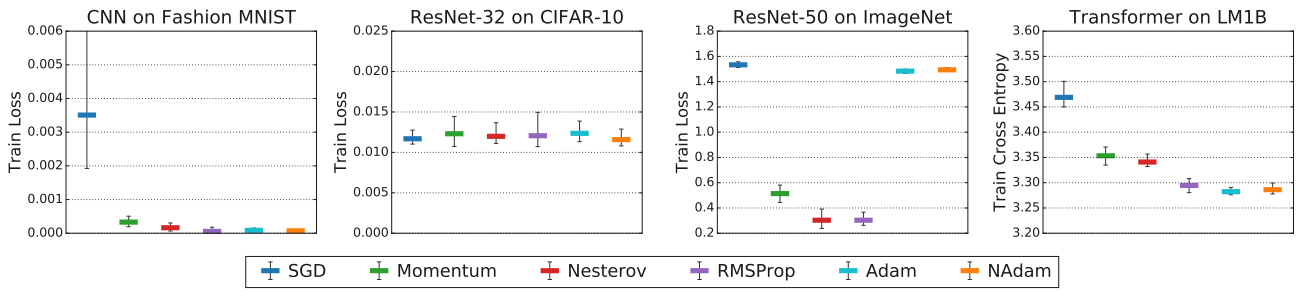


Figure 9: The relative performance of optimizers is consistent with the inclusion relationships when we select for lowest training loss. Note that SGD, ADAM, and NADAM for ResNet-50 on ImageNet used label smoothing in their final search spaces (see Section D.3), which makes their loss values incommensurate with the other optimizers. This is because their final search spaces were optimized to minimize validation error—if we had optimized their search spaces to minimize training error instead, we would not have used label smoothing, and we expect their training loss values would be consistent with the inclusion relationships.

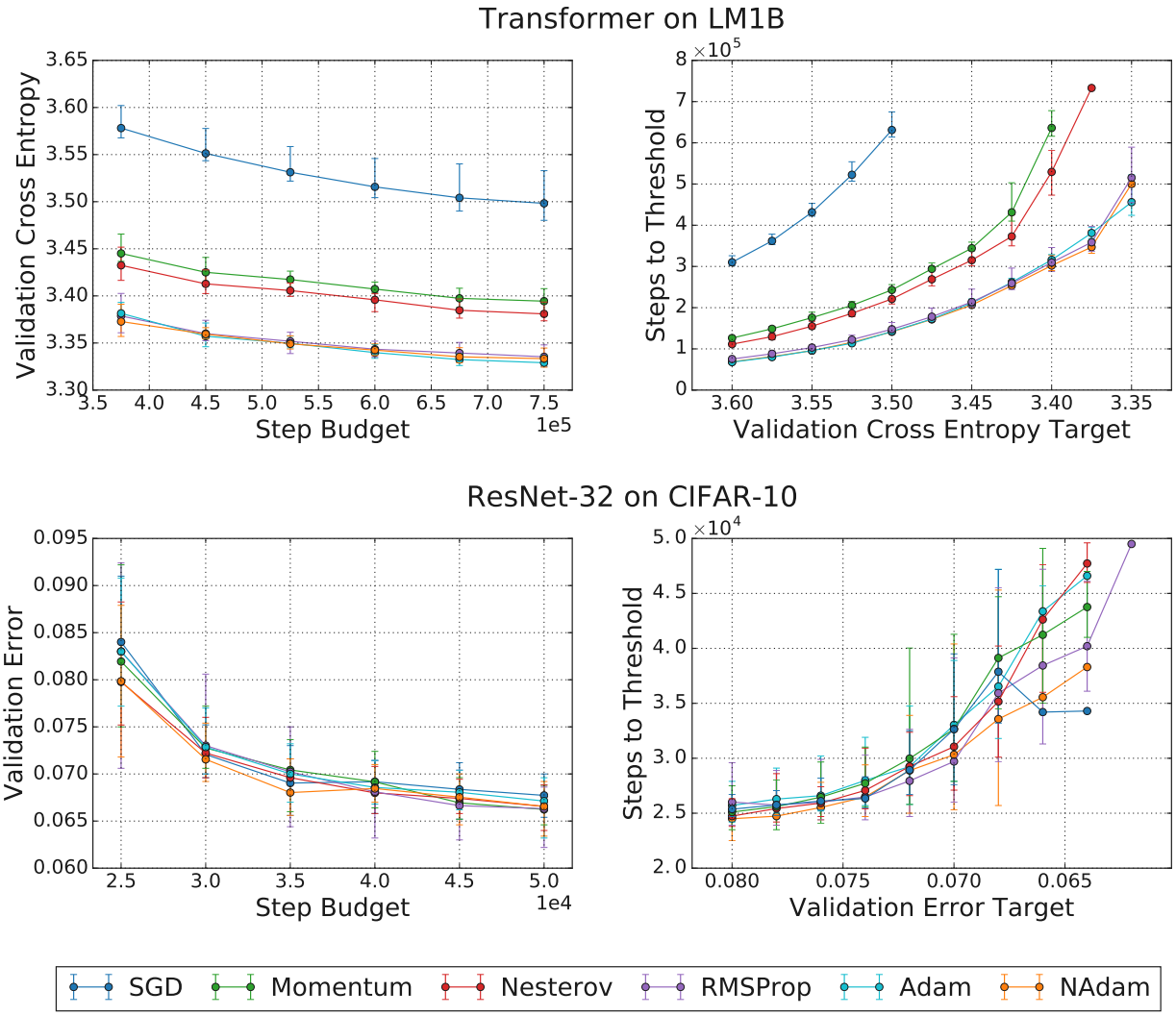


Figure 10: Our results confirming the relevance of optimizer inclusion relationships do not depend on the exact step budgets or error targets we chose.

Journal of Machine Learning Research 20 (2019) 1-49 Submitted 11/18; Published 7/19

Measuring the Effects of Data Parallelism on Neural Network Training

Christopher J. Shallue* SHALLUE@GOOGLE.COM
Jaehoon Lee*† JAEHLEE@GOOGLE.COM
Joseph Antognini† JOE.ANTOGNINI@GMAIL.COM
Jascha Sohl-Dickstein JASCHASD@GOOGLE.COM
Roy Frostig FROSTIG@GOOGLE.COM
George E. Dahl GDAHL@GOOGLE.COM

Google Brain
1600 Amphiteatre Parkway
Mountain View, CA, 94043, USA

Editor: Rob Fergus

Abstract

Recent hardware developments have dramatically increased the scale of data parallelism available for neural network training. Among the simplest ways to harness next-generation hardware is to increase the batch size in standard mini-batch neural network training algorithms. In this work, we aim to experimentally characterize the effects of increasing the batch size on training time, as measured by the number of steps necessary to reach a goal out-of-sample error. We study how this relationship varies with the training algorithm, model, and data set, and find extremely large variation between workloads. Along the way, we show that disagreements in the literature on how batch size affects model quality can largely be explained by differences in metaparameter tuning and compute budgets at different batch sizes. We find no evidence that larger batch sizes degrade out-of-sample performance. Finally, we discuss the implications of our results on efforts to train neural networks much faster in the future. Our experimental data is publicly available as a database of 71,638,836 loss measurements taken over the course of training for 168,160 individual models across 35 workloads.

Keywords: neural networks, stochastic gradient descent, data parallelism, batch size, deep learning

1. Introduction

Neural networks have become highly effective at a wide variety of prediction tasks, including image classification, machine translation, and speech recognition. The dramatic improvements in predictive performance over the past decade have partly been driven by advances in hardware for neural network training, which have enabled larger models to be trained on larger datasets than ever before. However, although modern GPUs and custom

∗. Both authors contributed equally.
†. Work done as a member of the Google AI Residency program (g.co/airesidency).

accelerators have made training neural networks orders of magnitude faster, training time still limits both the predictive performance of these techniques and how widely they can be applied. For many important problems, the best models are still improving at the end of training because practitioners cannot afford to wait until the performance saturates. In extreme cases, training must end before completing a single pass over the data (e.g. Anil et al., 2018). Techniques that speed up neural network training can significantly benefit many important application areas. Faster training can facilitate dramatic improvements in model quality by allowing practitioners to train on more data (Hestness et al., 2017), and by decreasing the experiment iteration time, allowing researchers to try new ideas and configurations more rapidly. Faster training can also allow neural networks to be deployed in settings where models have to be updated frequently, for instance when new models have to be produced when training data get added or removed.

Data parallelism is a straightforward and popular way to accelerate neural network training. For our purposes, data parallelism refers to distributing training examples across multiple processors to compute gradient updates (or higher-order derivative information) and then aggregating these locally computed updates. As long as the training objective decomposes into a sum over training examples, data parallelism is model-agnostic and applicable to any neural network architecture. In contrast, the maximum degree of *model parallelism* (distributing parameters and computation across different processors for the same training examples) depends on the model size and structure. Although data parallelism can be simpler to implement, ultimately, large scale systems should consider all types of parallelism at their disposal. In this work, we focus on the costs and benefits of data parallelism in the synchronous training setting.

Hardware development is trending towards increasing capacity for data parallelism in neural network training. Specialized systems using GPUs or custom ASICs (e.g. Jouppi et al., 2017) combined with high-performance interconnect technology are unlocking unprecedented scales of data parallelism where the costs and benefits have not yet been well studied. On the one hand, if data parallelism can provide a significant speedup at the limits of today's systems, we should build much bigger systems. On the other hand, if additional data parallelism comes with minimal benefits or significant costs, we might consider designing systems to maximize serial execution speed, exploit other types of parallelism, or even prioritize separate design goals such as power use or cost.

There is considerable debate in the literature about the costs and benefits of data parallelism in neural network training and several papers take seemingly contradictory positions. Some authors contend that large-scale data parallelism is harmful in a variety of ways, while others contend that it is beneficial. The range of conjectures, suggestive empirical results, and folk knowledge seems to cover most of the available hypothesis space. Answering these questions definitively has only recently become important (as increasing amounts of data parallelism have become practical), so it is perhaps unsurprising that the literature remains equivocal, especially in the absence of sufficiently comprehensive experimental data.

In this work, we attempt to provide the most rigorous and extensive experimental study on the effects of data parallelism on neural network training to date. In order to achieve this goal, we consider realistic workloads up to the current limits of data parallelism. We try to avoid making assumptions about how the optimal metaparameters vary as a function of batch size. Finally, in order to guide future work, we consider any remaining limitations in

our methodology, and we discuss what we see as the most interesting unanswered questions that arise from our experiments.

1.1 Scope

We restrict our attention to variants of mini-batch stochastic gradient descent (SGD), which are the dominant algorithms for training neural networks. These algorithms iteratively update the model's parameters using an estimate of the gradient of the training objective. The gradient is estimated at each step using a different subset, or *(mini-) batch*, of training examples. See Section 2.2 for a more detailed description of these algorithms. A data-parallel implementation computes gradients for different training examples in each batch in parallel, and so, in the context of mini-batch SGD and its variants, we equate the batch size with the amount of data parallelism.[1] We restrict our attention to synchronous SGD because of its popularity and advantages over asynchronous SGD (Chen et al., 2016).

Practitioners are primarily concerned with out-of-sample error and the cost they pay to achieve that error. Cost can be measured in a variety of ways, including training time and hardware costs. Training time can be decomposed into number of steps multiplied by average time per step, and hardware cost into number of steps multiplied by average hardware cost per step. The per-step time and hardware costs depend on the practitioner's hardware, but the number of training steps is hardware-agnostic and can be used to compute the total costs for any hardware given its per-step costs. Furthermore, in an idealized data-parallel system where the communication overhead between processors is negligible, training time depends only on the number of training steps (and not the batch size) because the time per step is independent of the number of examples processed. Indeed, this scenario is realistic today in systems like TPU pods[2], where there are a range of batch sizes for which the time per step is almost constant. Since we are primarily concerned with training time, we focus on number of training steps as our main measure of training cost.

An alternative hardware-agnostic measure of training cost is the number of training examples processed, or equivalently the number of passes (*epochs*) over the training data. This measure is suitable when the per-step costs are proportional to the number of examples processed (e.g. hardware costs proportional to the number of floating point operations). However, the number of epochs is not a suitable measure of training time in a data-parallel system—it is possible to reduce training time by using a larger batch size and processing *more* epochs of training data, provided the number of training steps decreases.

In light of practitioners' primary concerns of out-of-sample error and the resources needed to achieve it, we believe the following questions are the most important to study to understand the costs and benefits of data parallelism with mini-batch SGD and its variants:

1. What is the relationship between batch size and number of training steps to reach a goal out-of-sample error?
2. What governs this relationship?
3. Do large batch sizes incur a cost in out-of-sample error?

1. Mini-batch SGD can be implemented in a variety of ways, including data-serially, but a data-parallel implementation is always possible given appropriate hardware.
2. `https://www.blog.google/products/google-cloud/google-cloud-offer-tpus-machine-learning/`.

1.2 Contributions of This Work

1. We show that the relationship between batch size and number of training steps to reach a goal out-of-sample error has the same characteristic form across six different families of neural network, three training algorithms, and seven data sets.

   Specifically, for each workload (model, training algorithm, and data set), increasing the batch size initially decreases the required number of training steps proportionally, but eventually there are diminishing returns until finally increasing the batch size no longer changes the required number of training steps. To the best of our knowledge, we are the first to experimentally validate this relationship across models, training algorithms, and data sets while independently tuning the learning rate, momentum, and learning rate schedule (where applicable) for each batch size. Unlike prior work that made strong assumptions about these metaparameters, our results reveal a universal relationship that holds across all workloads we considered, across different error goals, and when considering either training error or out-of-sample error.

2. We show that the maximum useful batch size varies significantly between workloads and depends on properties of the model, training algorithm, and data set. Specifically, we show that:

   (a) SGD with momentum (as well as Nesterov momentum) can make use of much larger batch sizes than plain SGD, suggesting future work to study the batch size scaling properties of other algorithms.

   (b) Some models allow training to scale to much larger batch sizes than others. We include experimental data on the relationship between various model properties and the maximum useful batch size, demonstrating that the relationship is not as simple as one might hope from previous work (e.g. wider models do *not* always scale better to larger batch sizes).

   (c) The effect of the data set on the maximum useful batch size tends to be smaller than the effects of the model and training algorithm, and does not depend on data set size in a consistent way.

3. We show that the optimal values of training metaparameters do not consistently follow any simple relationships with the batch size. In particular, popular learning rate heuristics—such as linearly scaling the learning rate with the batch size— do not hold across all problems or across all batch sizes.

4. Finally, by reviewing the specifics of the experimental protocols used in prior work, we at least partially reconcile conflicting stances in the literature on whether increasing the batch size degrades model quality. Specifically, we show that assumptions about computational budgets and the procedures for selecting metaparameters at different batch sizes can explain many of the disagreements in the literature. We find no evidence that increasing the batch size necessarily degrades model quality, but additional regularization techniques may become important at larger batch sizes.

1.3 Experimental Data

We release our raw experimental data for any further analysis by the research community.[3] Our database contains 454 combinations of workload (model, data set, training algorithm) and batch size, each of which is associated with a metaparameter search space and a set of models trained with different configurations sampled from the search space. In total, our data contains 71,638,836 loss measurements taken over the course of training for 168,160 individual models. Together, these measurements make up the training curves of all of the individual models we trained, and can be used to reproduce all plots in this paper.[4]

2. Setup and Background

In this section we set up the basic definitions and background concepts used throughout the paper.

2.1 Learning

A *data distribution* is a probability distribution $\mathcal{D}$ over a data domain $\mathcal{Z}$. For example, we might consider a supervised learning task over a domain $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ is the set of 32-by-32-pixel color images and $\mathcal{Y}$ is the set of possible labels denoting what appears in the image. A *training set* $z_1, \ldots, z_n \in \mathcal{Z}$ is a collection of *examples* from the data domain, conventionally assumed to be drawn i.i.d. from the data distribution $\mathcal{D}$.

A machine learning *model* is a function that, given *parameters* θ from some set $\Theta \subset \mathbb{R}^d$, and given a data point $z \in \mathcal{Z}$, produces a prediction whose quality is measured by a differentiable non-negative scalar-valued loss function.[5] We denote by $\ell(\theta; z)$ the loss of a prediction made by the model, under parameters θ, on the data point z. We denote by L the *out-of-sample loss* or *expected loss*:

$$L(\theta) = \mathop{\mathbb{E}}_{z \sim \mathcal{D}} \left[\ell(\theta; z) \right], \tag{1}$$

and by $\hat{L}$ the *empirical average loss* under a data set $S = (z_1, \ldots, z_n)$:

$$\hat{L}(\theta; S) = \frac{1}{n} \sum_{i=1}^{n} \ell(\theta; z_i). \tag{2}$$

When S is the training set, we call $\hat{L}$ the *average training loss*. We will say that the data source $\mathcal{D}$, loss ℓ, and model with parameter set Θ together specify a learning *task*, in which our aim is to find parameters θ that achieve low out-of-sample loss (Equation 1), while given access only to n training examples. A common approach is to find parameters of low average training loss (Equation 2) as an estimate of the out-of-sample loss (Shalev-Shwartz and Ben-David, 2014).

When minimizing average training loss $\hat{L}$, it is common to add regularization penalties to the objective function. For a differentiable penalty $R : \Theta \to \mathbb{R}_+$, regularization weight

3. `https://github.com/google-research/google-research/tree/master/batch_science`
4. `https://colab.research.google.com/github/google-research/google-research/blob/master/batch_science/reproduce_paper_plots.ipynb`
5. Technically, the loss need only be sub-differentiable. Extending our setup to this end is straightforward.

$\lambda > 0$, and training set S, the training objective might be

$$J(\theta) = \hat{L}(\theta; S) + \lambda R(\theta). \tag{3}$$

In practice, we often approach a task by replacing its loss with another that is more amenable to training. For instance, in supervised classification, we might be tasked with learning under the 0/1 loss, which is an indicator of whether a prediction is correct (e.g. matches a ground-truth label), but we train by considering instead a surrogate loss (e.g. the logistic loss) that is more amenable to continuous optimization. When the surrogate loss bounds the original, achieving low loss under the surrogate implies low loss under the original. To distinguish the two, we say *error* to describe the original loss (e.g. 0/1), and we save *loss* to refer to the surrogate used in training.

2.2 Algorithms

The dominant algorithms for training neural networks are based on *mini-batch stochastic gradient descent* (SGD, Robbins and Monro, 1951; Kiefer et al., 1952; Rumelhart et al., 1986; Bottou and Bousquet, 2008; LeCun et al., 2015). Given an initial point $\theta_0 \in \Theta$, mini-batch SGD attempts to decrease the objective J via the sequence of iterates

$$\theta_t \leftarrow \theta_{t-1} - \eta_t g(\theta_{t-1}; B_t),$$

where each B_t is a random subset of training examples, the sequence $\{\eta_t\}$ of positive scalars is called the *learning rate*, and where, for any $\theta \in \Theta$ and $B \subset S$,

$$g(\theta; B) = \frac{1}{|B|} \sum_{z \in B} \nabla \ell(\theta; z) + \lambda \nabla R(\theta). \tag{4}$$

When the examples B are a uniformly random subset of training examples, $g(\theta; B)$ forms an unbiased estimate of the gradient of the objective J that we call a *stochastic gradient*. In our larger-scale experiments, when we sample subsequent batches B_t, we actually follow the common practice of cycling through permutations of the training set (Shamir, 2016). The result of mini-batch SGD can be any of the iterates θ_t for which we estimate that $L(\theta_t)$ is low using a validation data set.

Variants of SGD commonly used with neural networks include SGD with momentum (Polyak, 1964; Rumelhart et al., 1986; Sutskever et al., 2013), Nesterov momentum (Nesterov, 1983; Sutskever et al., 2013), RMSProp (Hinton et al., 2012), and Adam (Kingma and Ba, 2015). All of these optimization procedures, or *optimizers*, interact with the training examples only by repeatedly computing stochastic gradients (Equation 4), so they support the same notion of batch size that we equate with the scale of data parallelism. In this work, we focus on the SGD, SGD with momentum, and Nesterov momentum optimizers. The latter two optimizers are configured by a learning rate $\{\eta_t\}$ and a scalar $\gamma \in (0, 1)$ that we call *momentum*. They define the iterates[6]

SGD with momentum	Nesterov momentum
$v_{t+1} \leftarrow \gamma v_t + g(\theta_t; B_t)$	$v_{t+1} \leftarrow \gamma v_t + g(\theta_t; B_t)$
$\theta_{t+1} \leftarrow \theta_t - \eta_t v_{t+1}$	$\theta_{t+1} \leftarrow \theta_t - \eta_t g(\theta_t; B_t) - \eta_t \gamma v_{t+1}$,

6. These rules take slightly different forms across the literature and across library implementations. We present and use the update rules from the `MomentumOptimizer` class in TensorFlow (Abadi et al., 2016).

given $v_0 = 0$ and an initial θ_0. Note that plain SGD can be recovered from either optimizer by taking $\gamma = 0$. The outcome of using these optimizers should therefore be no worse if than SGD, in any experiment, the momentum γ is tuned across values including zero.

If we run SGD with momentum under a constant learning rate $\eta_t = \eta$, then, at a given iteration t, the algorithm computes

$$\theta_{t+1} = \theta_t - \eta v_{t+1} = \theta_0 - \eta \sum_{u=0}^{t} v_{u+1} = \theta_0 - \eta \sum_{u=0}^{t} \sum_{s=0}^{u} \gamma^{u-s} g(\theta_s; B_s).$$

For any fixed $\tau \in \{0, \ldots, t\}$, the coefficient accompanying the stochastic gradient $g(\theta_\tau; B_\tau)$ in the above update is $\eta \sum_{u=\tau}^{t} \gamma^{u-\tau}$. We define the *effective learning rate*, η^{eff} as the value of this coefficient at the end of training ($t = T$), in the limit of a large number of training steps ($T \to \infty$, while τ is held fixed):

$$\eta^{\text{eff}} = \lim_{T \to \infty} \sum_{u=\tau}^{T} \eta \gamma^{u-\tau} = \frac{\eta}{1-\gamma}.$$

Put intuitively, η^{eff} captures the contribution of a given mini-batch gradient to the parameter values at the end of training.

2.3 Additional Terminology in Experiments

A *data-parallel implementation* of mini-batch SGD (or one of its variants) computes the summands of Equation 4 in parallel and then synchronizes to coordinate their summation.

The models and algorithms in our experiments are modifiable by what we call *metaparameters*.[7] These include architectural choices, such as the number of layers in a neural network, and training parameters, such as learning rates $\{\eta_t\}$ and regularization weights λ. When we use the term *model*, we typically assume that all architectural metaparameters have been set. In our experiments, we *tune* the training metaparameters by selecting the values that yield the best performance on a validation set. We use the term *workload* to jointly refer to a data set, model, and training algorithm.

3. Related Work

In this section we review prior work related to our three main questions from Section 1.1. First we review studies that considered the relationship between batch size and number of training steps (Questions 1 and 2), and then we review studies that considered the effects of batch size on solution quality (Question 3).

3.1 Steps to Reach a Desired Out-Of-Sample Error

We broadly categorize the related work on this topic as either analytical or empirical in nature.

7. Sometimes called "hyperparameters," but we prefer a different name so as not to clash with the notion of hyperparameters in Bayesian statistics.

3.1.1 Analytical Studies

Convergence upper bounds from the theory of stochastic (convex) optimization can be specialized to involve terms dependent on batch size, so in this sense they comprise basic related work. These upper bounds arise from worst-case analysis, and moreover make convexity and regularity assumptions that are technically violated in neural network training, so whether they predict the actual observed behavior of our experimental workloads is an empirical question in its own right.

Given a sequence of examples drawn i.i.d. from a data source, an upper bound on the performance of SGD applied to L-Lipschitz convex losses is (Hazan, 2016; Shalev-Shwartz and Ben-David, 2014)

$$J(\theta_T) - J^\star \leq O\left(\sqrt{\frac{L^2}{T}}\right), \tag{5}$$

for any batch size. Here, J is the objective function, $J^\star$ is its value at the global optimum, and θ_T denotes the final output of the algorithm supposing it took T iterations.[8] Meanwhile, when losses are convex and the objective is H-smooth, accelerated parallel mini-batch SGD enjoys the bound (Lan, 2012)

$$J(\theta_T) - J^\star \leq O\left(\frac{H}{T^2} + \sqrt{\frac{L^2}{Tb}}\right), \tag{6}$$

where b is the batch size.

Compared to sequential processing without batching (i.e. a batch size of one), the bounds Equation 5 and Equation 6 offer two extremes, respectively:

1. **No benefit:** Increasing the batch size b does not change the number of steps to convergence, as per Equation 5.

2. **A b-fold benefit:** The term in Equation 6 proportional to $1/\sqrt{Tb}$ dominates the bound. Increasing the batch size b by a multiplicative factor decreases the number of steps T to a given achievable objective value by the same factor.

In other words, under these simplifications, batching cannot hurt the asymptotic guarantees of steps to convergence, but it could be wasteful of examples. The two extremes imply radically different guidance for practitioners, so the critical task of establishing a relationship between batch size and number of training steps remains one to resolve experimentally.

A few recent papers proposed analytical notions of a critical batch size: a point at which a transition occurs from a b-fold benefit to no benefit. Under assumptions including convexity, Ma et al. (2018) derived such a critical batch size, and argued that a batch size of one is optimal for minimizing the number of training epochs required to reach a given target error. Under different assumptions, Yin et al. (2018) established a critical batch size and a pathological loss function that together exhibit a transition from a b-fold benefit to no benefit. Although they ran experiments with neural networks, their experiments were designed to investigate the effect of data redundancy and do not provide enough

8. Not necessarily the T^{th} iterate, which may differ from θ_T if the algorithm averages its iterates.

information to reveal the empirical relationship between batch size and number of training steps. Focusing on linear least-squares regression, Jain et al. (2018) also derived a threshold batch size in terms of the operator norm of the objective's Hessian and a constant from a fourth-moment bound on example inputs.

To our knowledge, in all previous work that analytically derived a critical batch size, the thresholds defined are either (i) parameter-dependent, or (ii) specific to linear least-squares regression. A critical batch size that depends on model parameters can change over the course of optimization; it is not a problem-wide threshold that can be estimated efficiently *a priori*. Focusing on least-squares has issues as well: while it sheds intuitive light on how batching affects stochastic optimization locally, the quantities defined inherently cannot generalize to the non-linear optimization setting of neural network training, both because the objective's Hessian is not constant across the space of parameters as it is in a quadratic problem, and more broadly because it is unclear whether the Hessian of the objective is still the correct analogue to consider.

3.1.2 Empirical Studies

Wilson and Martinez (2003) investigated the relationship between batch size and training speed for plain mini-batch SGD. They found that a simple fully connected neural network took more epochs to converge with larger batch sizes on a data set of 20,000 examples, and also that using a batch size equal to the size of the training set took more epochs to converge than a batch size of one on several small data sets of size ≤ 600. However, their experimental protocol and assumptions limit the conclusions we can draw from their results. One issue is that training time was measured to different out-of-sample errors for different batch sizes on the same data set. To compare training speed fairly, the error goal should be fixed across all training runs being compared. Additionally, only four learning rates were tried for each data set, but quite often the best learning rate was at one of the two extremes and it appeared that a better learning rate might be found outside of the four possibilities allowed. Finally, despite the contention of the authors, their results do not imply slower training with larger batch sizes in data-parallel training: for the most part, their larger batch size experiments took fewer training steps than the corresponding batch size one experiments.

In the last few years, increasingly specialized computing systems have spurred practitioners to try much larger batch sizes than ever before, while increasingly promising results have driven hardware designers to create systems capable of even more data parallelism. Chen et al. (2016) used a pool of synchronized worker machines to increase the effective batch size of mini-batch SGD. They demonstrated speedups in both wall time and steps to convergence for an Inception model (Szegedy et al., 2016) on ImageNet (Russakovsky et al., 2015) by scaling the effective batch size from 1,600 to 6,400. More recently, Goyal et al. (2017) showed that the number of training epochs could be held constant across a range of batch sizes to achieve the same validation error for ResNet-50 (He et al., 2016a) on ImageNet. Holding the number of training epochs constant is equivalent to scaling the number of training steps inversely with the batch size, and this reduction in training steps with increasing batch size produced nearly proportional wall time speedups on their hardware. Although this hints at a b-fold benefit regime in which increasing the batch size reduces the

number of training steps by the same factor, the authors did not attempt to minimize the number of training steps (or epochs) required to reach the goal at each batch size separately. It is unclear whether any of the batch sizes that achieved the goal could do so in fewer steps than given, or how many steps the other batch sizes would have needed to achieve the same error goal.

Two studies performed concurrently with this work also investigated the relationship between batch size and training speed for neural networks. Chen et al. (2018) provide experimental evidence of a problem-dependent critical batch size after which a b-fold benefit is no longer achieved for plain mini-batch SGD. They contend that wider and shallower networks have larger critical batch sizes, and while their empirical results are equivocal for this particular claim, they show that the threshold batch size can depend on aspects of both the data set and the model. Additionally, Golmant et al. (2018) studied how three previously proposed heuristics for adjusting the learning rate as a function of batch size (linear scaling, square root scaling, and no scaling) affect the number of training steps required to reach a particular result. They found that if the learning rate is tuned for the the smallest batch size only, all three of these common scaling techniques break down for larger batch sizes and result in either (i) divergent training, or (ii) training that cannot reach the error goal within a fixed number of training epochs. They also describe a basic relationship between batch size and training steps to a fixed error goal, which is comprised of three regions: b-fold benefit initially, then diminishing returns, and finally no benefit for all batch sizes greater than a maximum useful batch size. However, their results are inconclusive because (i) not all model and data set pairs exhibit this basic relationship, (ii) it does not appear consistently across error goals, and (iii) the relationship is primarily evident in training error but not out-of-sample error. These inconsistent results may be due to suboptimal pre-determined learning rates arising from the scaling rules, especially at larger batch sizes. Finally, they also found that the maximum useful batch size depends on aspects of the model and the data set type, but not on the data set size. Since all their experiments use plain mini-batch SGD, their results are unable to reveal any effects from the choice of optimizer and might not generalize to other popular optimizers, such as SGD with momentum.

3.2 Solution Quality

The literature contains some seemingly conflicting claims about the effects of batch size on solution quality (out-of-sample error at the conclusion of training). Primarily, the debate centers on whether increasing the batch size incurs a cost in solution quality. Keskar et al. (2017) argue that large batch[9] training converges to so-called "sharp" minima with worse generalization properties. However, Dinh et al. (2017) show that a minimum with favorable generalization properties can be made, through reparameterization, arbitrarily sharp in the same sense. Le Cun et al. (1998) suggest that a batch size of one can result in better solutions because the noisier updates allow for the possibility of escaping from local minima in a descent algorithm. However, they also note that we usually stop training long before

9. The term "large batch" is inherently ambiguous, and in this case accompanies experiments in Keskar et al. (2017) that only compare two absolute batch sizes per data set, rather than charting out a curve to its apparent extremes.

reaching any sort of critical point. Hoffer et al. (2017) argue that increasing the batch size need not degrade out-of-sample error at all, assuming training has gone on long enough. Goyal et al. (2017), among others, tested batch sizes larger than those used in Keskar et al. (2017) without noticing any reduction in solution quality. Still, their results with yet larger batch sizes do not rule out the existence of a more sudden degradation once the batch size is large enough. Meanwhile, Goodfellow et al. (2016) state that small batches can provide a regularization effect such that they result in the best observed out-of-sample error, although in this case other regularization techniques might serve equally well.

Alas, the best possible out-of-sample error for a particular model and data set cannot be measured unconditionally due to practical limits on wall time and hardware resources, as well as practical limits on our ability to tune optimization metaparameters (e.g. the learning rate). An empirical study can only hope to measure solution quality subject to the budgets allowed for each model experiment, potentially with caveats due to limitations of the specific procedures for selecting the metaparameters. To the best of our knowledge, all published results handle the training budget issue in exactly one of three ways: by ignoring budgets (train to convergence, which is not always possible); by using a step budget (restrict the number of gradient descent updates performed); or by using an epoch budget (restrict number of training examples processed).[10] Furthermore, while some published results tune the learning rate anew for each batch size, others tune for only a single batch size and use a preordained heuristic to set the learning rate for the remaining batch sizes (the most common heuristics are constant, square root, and linear learning rate scaling rules). Tuning metaparameters at a single batch size and then heuristically adjusting them for others could clearly create a systematic advantage for trials at batch sizes near to the one tuned. All in all, the conclusions we can draw from previous studies depend on the budgets they assume and on how they select metaparameters across batch sizes. The following subsections attempt an investigation of their experimental procedures to this end.

3.2.1 Studies That Ignore Budgets

All studies in this section compared solution quality for different batch sizes after deeming their models to have converged. They determined training stopping time by using either manual inspection, convergence heuristics, or fixed compute budgets that they considered large enough to guarantee convergence.[11]

Keskar et al. (2017) trained several neural network architectures on MNIST and CIFAR-10, each with two batch sizes, using the Adam optimizer and without changing the learning rate between batch sizes. They found that the larger batch size consistently achieved worse out-of-sample error after training error had ceased to improve. However, all models used batch normalization (Ioffe and Szegedy, 2015) and presumably computed the batch nor-

10. Of course, there are budgets in between an epoch budget and a step budget that might allow the possibility of trading off time, computation, and/or solution quality. For example, it may be possible to increase the number of training epochs and still take fewer steps to reach the same quality solution. However, we are not aware of work that emphasizes these budgets.

11. As discussed further in Section 4.8, we find that millions of training steps for small batch sizes, or thousands of epochs for large batch sizes, are required to saturate performance even for data sets as small and simple as MNIST. In our experiments, this corresponded to more than 25 hours of wall-time for each metaparameter configuration.

malization statistics using the full batch size. For a fair comparison between batch sizes, batch normalization statistics should be computed over the same number of examples or else the training objective differs between batch sizes (Goyal et al., 2017). Indeed, Hoffer et al. (2017) found that computing batch normalization statistics over larger batches can degrade solution quality, which suggests an alternative explanation for the results of Keskar et al. (2017). Moreover, Keskar et al. (2017) reported that data augmentation eliminated the difference in solution quality between small and large batch experiments.

Smith and Le (2018) trained a small neural network on just 1,000 examples sampled from MNIST with two different batch sizes, using SGD with momentum and without changing the learning rate between batch sizes. They observed that the larger batch size overfit more than the small batch size resulting in worse out-of-sample error, but this gap was mitigated by applying L_2 regularization (Smith and Le, 2018, figures 3 and 8). They also compared a wider range of batch sizes in experiments that either (i) used a step budget without changing the learning rate for each batch size (Smith and Le, 2018, figures 4 and 6), or (ii) varied the learning rate and used a step budget that was a function of the learning rate (Smith and Le, 2018, figure 5). Instead, we focus on the case where the learning rate and batch size are chosen independently.

Breuel (2015a,b) trained a variety of neural network architectures on MNIST with a range of batch sizes, using the SGD and SGD with momentum optimizers with a range of learning rates and momentum values. They found that batch size had no effect on solution quality for LSTM networks (Breuel, 2015a), but found that larger batch sizes achieved worse solutions for fully connected and convolutional networks, and that the scale of the effect depended on the activation function in the hidden and output layers (Breuel, 2015b).

Finally, Chen et al. (2016) observed no difference in solution quality when scaling the batch size from 1,600 to 6,400 for an Inception model on ImageNet when using the RMSProp optimizer and a heuristic to set the learning rate for each batch size.

3.2.2 Studies with Step Budgets

Hoffer et al. (2017) trained neural networks with two different batch sizes on several image data sets. They found that, by computing batch normalization statistics over a fixed number of examples per iteration ("ghost batch normalization"), and by scaling the learning rate with the square root of the batch size instead of some other heuristic, the solution quality arising from the larger batch size was as good as or better than the smaller batch size. However, the largest batch size used was 4,096, which does not rule out an effect appearing at still larger batch sizes, as suggested by the work of Goyal et al. (2017). Moreover, it remains open whether their proposed learning rate heuristic extends to arbitrarily large batch sizes, or whether it eventually breaks down for batch sizes sufficiently far from the base batch size.

3.2.3 Studies with Epoch Budgets

An epoch budget corresponds to fixing the total number of per-example gradient computations, but, in an idealized data-parallel implementation of SGD, it also corresponds to a step (or even wall time) budget that scales inversely with the batch size. With an epoch budget, a larger batch size can only achieve the same solution quality as a smaller batch

size if it achieves perfect scaling efficiency (a b-fold reduction in steps from increasing the batch size, as described in Section 3.1.1).

Masters and Luschi (2018) show that after a critical batch size depending on the model and data set, solution quality degrades with increasing batch size *when using a fixed epoch budget*. Their results effectively show a limited region of b-fold benefit for those model and data set pairs when trained with SGD, although they did not investigate whether this critical batch size depends on the optimizer used, and they did not consider more than one epoch budget for each problem. We reproduced a subset of their experiments and discuss them in Section 5.

Goyal et al. (2017) recently popularized a linear learning rate scaling heuristic for training the ResNet-50 model using different batch sizes. Using this heuristic, a 90 epoch budget, and SGD with momentum without adjusting or tuning the momentum, they increased the batch size from 64 to 8,192 with no loss in accuracy. However, their learning rate heuristic broke down for even larger batch sizes. Inspired by these results, a sequence of follow-up studies applied additional techniques to further increase the batch size while still achieving the same accuracy and using the same 90 epoch budget. These follow-on studies (Codreanu et al., 2017; You et al., 2017; Akiba et al., 2017) confirm that the best solution quality for a given batch size will also depend on the exact optimization techniques used.

There are several additional papers (Lin et al., 2018; Devarakonda et al., 2017; Golmant et al., 2018) with experiments relevant to solution quality that used an epoch budget, tuned the learning rate for the smallest batch size, and then used a heuristic to choose the learning rate for all larger batch sizes. For instance, Devarakonda et al. (2017) and Lin et al. (2018) used linear learning rate scaling and Golmant et al. (2018) tried constant, square root, and linear learning rate scaling heuristics. All of them concluded that small batch sizes have superior solution quality to large batch sizes with a fixed epoch budget, for various notions of "small" and "large." This could just as easily be an artifact of the learning rate heuristics, and a possible alternative conclusion is that these heuristics are limited (as heuristics often are).

4. Experiments and Results

The primary quantity we measure is the number of steps needed to first reach a desired out-of-sample error, or *steps to result*. To measure steps to result, we used seven image and text data sets with training set sizes ranging from 45,000 to 26 billion examples. Table 1 summarizes these data sets and Appendix A provides the full details. We chose six families of neural network to train on these data sets. For MNIST and Fashion MNIST, we chose a simple fully connected neural network and a simple convolutional neural network (CNN). For CIFAR-10, we chose the ResNet-8 model without batch normalization, partly to compare our results to Masters and Luschi (2018), and partly to have a version of ResNet without batch normalization. For ImageNet, we chose ResNet-50, which uses batch normalization and residual connections, and VGG-11, which uses neither. For Open Images, we chose ResNet-50. For LM1B, we chose the Transformer model and an LSTM model. For Common Crawl, we chose the Transformer model. Table 2 summarizes these models and Appendix B provides the full details.

Data Set	Type	Task	Size	Evaluation Metric
MNIST	Image	Classification	55,000	Classification error
Fashion MNIST	Image	Classification	55,000	Classification error
CIFAR-10	Image	Classification	45,000	Classification error
ImageNet	Image	Classification	1,281,167	Classification error
Open Images	Image	Classification (multi-label)	4,526,492	Average precision
LM1B	Text	Language modeling	30,301,028	Cross entropy error
Common Crawl	Text	Language modeling	∼25.8 billion	Cross entropy error

Table 1: Summary of data sets. Size refers to the number of examples in the training set, which we measure in sentences for text data sets. See Appendix A for full details.

Model Class	Sizes	Optimizers	Data Sets	Learning rate schedule
Fully Connected	Various	SGD	MNIST	Constant
Simple CNN	Base Narrow Wide	SGD Momentum Nesterov mom.	MNIST Fashion MNIST	Constant
ResNet	ResNet-8	SGD Nesterov mom.	CIFAR-10	Linear decay
	ResNet-50	Nesterov mom.	ImageNet Open Images	Linear decay
VGG	VGG-11	Nesterov mom.	ImageNet	Linear decay
Transformer	Base Narrow and shallow Shallow Wide	SGD Momentum Nesterov mom.	LM1B Common crawl	Constant
LSTM	—	Nesterov mom.	LM1B	Constant

Table 2: Summary of models. See Appendix B for full details.

Measuring steps to result requires a particular value of out-of-sample error to be chosen as the goal. Ideally, we would select the best achievable error for each task and model, but since validation error is noisy, the best error is sometimes obtained unreliably. Moreover, for some workloads, the validation error continues to improve steadily beyond the maximum practical training time. Therefore, we generally tried to select the best validation error that we could achieve reliably within a practical training time.

Table 2 also shows the learning rate schedule we used for each model and data set. Learning rate schedules are often used to accelerate neural network training, but finding the best schedule is an optimization problem in its own right (Wu et al., 2018). Instead, researchers typically choose from a range of common learning rate functions based on validation performance and individual preference. While most schedules decay the learning rate monotonically over training, some researchers also "warm-up" the learning rate at the start of training (e.g. He et al., 2016a), particularly when training with large batch sizes (Goyal et al., 2017). We ran experiments with both constant learning rates and with learning rate decay. We used decay for ResNet-8, ResNet-50, and VGG-11, which significantly reduced

training time for those models. We selected our decay function by running an extensive set of experiments with ResNet-50 on ImageNet (see Appendix C for details). We chose linear decay because it performed at least as well as all other schedules we tried, while also being the simplest and requiring only two additional metaparameters. In experiments that used linear decay, we specified metaparameters (η_0, α, T) such that the learning rate decayed linearly from η_0 to $\eta_T = \alpha\eta_0$. That is, the learning rate at step t is given by

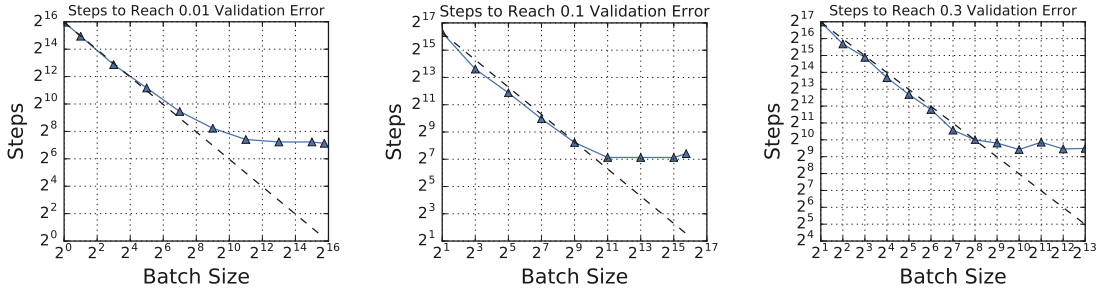
$$\eta_t = \begin{cases} \eta_0 - (1-\alpha)\eta_0 \frac{t}{T} & \text{if } t \leq T, \\ \alpha\eta_0 & \text{if } t > T. \end{cases}$$

Steps to result depends on the training metaparameters, and, for a given task and model, each batch size might have a different metaparameter configuration that minimizes steps to result. In all experiments, we independently tuned the metaparameters at each batch size, including the initial learning rate η_0 and, when learning rate decay was used, the decay schedule (α, T). Also, unless otherwise specified, we used the Nesterov momentum optimizer (Sutskever et al., 2013) and tuned the momentum γ.[12] Tuning anew for each batch size is extremely important since otherwise we would not be measuring steps to result as a function of batch size, rather we would be measuring steps to result as a function of batch size and the specific values of the learning rate and other metaparameters. We used quasi-random search (Bousquet et al., 2017) to tune the metaparameters with equal budgets of non-divergent[13] trials for different batch sizes. We selected metaparameter search spaces by hand based on preliminary experiments. The exact number of non-divergent trials needed to produce stable results depends on the search space, but 100 trials seemed to suffice in our experiments.[14] If the optimal trial occurred near the boundary of the search space, or if the goal validation error was not achieved within the search space, we repeated the search with a new search space. We measured steps to result for each batch size by selecting the metaparameter trial that reached the goal validation error in the fewest number of steps.

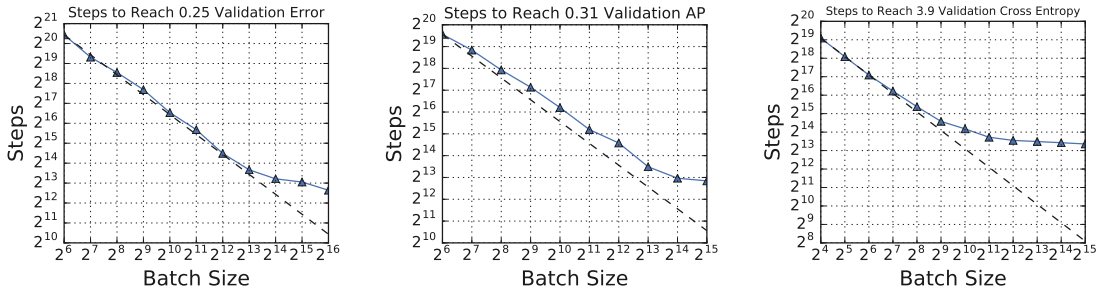
4.1 Steps to Result Depends on Batch Size in a Similar Way Across Problems

To get a sense of the basic empirical relationship, we measured the number of steps required to reach a goal validation error as a function of batch size across several different data sets and models (Figure 1). In all cases, as the batch size grows, there is an initial period of **perfect scaling** (b-fold benefit, indicated with a dashed line on the plots) where the steps needed to achieve the error goal halves for each doubling of the batch size. However, for all problems, this is followed by a region of **diminishing returns** that eventually leads to a regime of **maximal data parallelism** where additional parallelism provides no benefit whatsoever. In other words, for any given problem and without making strong assumptions about learning rates or other optimizer parameters, we can achieve both extremes suggested by theory (see Section 3.1.1). *A priori*, it is not obvious that every workload in our experiments should exhibit perfect scaling at the smallest batch sizes instead of immediately showing diminishing returns.

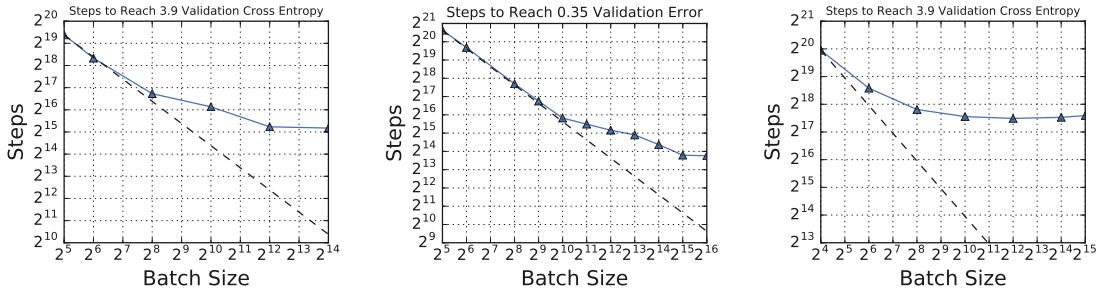
12. For LSTM for LM1B, we used a fixed value of $\gamma = 0.99$. We chose this value based on initial experiments and validated that tuning γ did not significantly affect the results for batch sizes 256, 1,024, or 4,096.
13. We discarded trials with a divergent training loss, which occurred when the learning rate was too high.
14. We used 100 non-divergent trials for all experiments except Transformer Shallow on LM1B with SGD, Transformer on Common Crawl, and LSTM on LM1B, for which we used 50 trials each.



(a) Simple CNN on MNIST (b) Simple CNN on Fashion MNIST (c) ResNet-8 on CIFAR-10

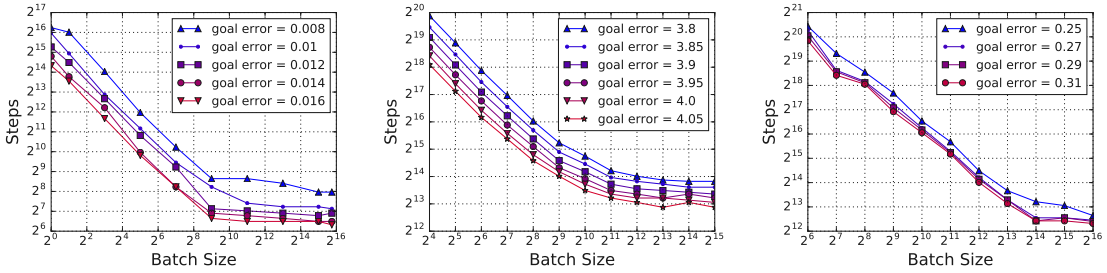


(d) ResNet-50 on ImageNet (e) ResNet-50 on Open Images (f) Transformer on LM1B



(g) Transformer on Common Crawl (h) VGG-11 on ImageNet (i) LSTM on LM1B

Figure 1: **The relationship between steps to result and batch size has the same characteristic form for all problems.** In all cases, as the batch size grows, there is an initial period of **perfect scaling** (indicated with a dashed line) where the steps needed to achieve the error goal halves for each doubling of the batch size. Then there is a region of **diminishing returns** that eventually leads to a region of **maximal data parallelism** where additional parallelism provides no benefit whatsoever. AP denotes average precision (see Appendix A).



(a) Simple CNN on MNIST (b) Transformer on LM1B (c) ResNet-50 on ImageNet

Figure 2: **Steps-to-result plots have a similar form for different (nearby) performance goals.** The transition points between the three regions (perfect scaling, diminishing returns, and maximal data parallelism) are nearly the same.

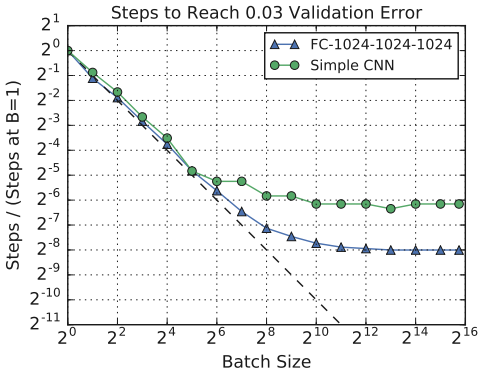
4.2 Validating Our Measurement Protocol

If the curves in Figure 1 were very sensitive to the goal validation error, then measuring the steps needed to reach our particular choice of the goal would not be a meaningful proxy for training speed. For small changes in the goal validation error, we do not care about vertical shifts as long as the transition points between the three scaling regions remain relatively unchanged. Figure 2 shows that varying the error goal only vertically shifts the steps-to-result curve, at least for modest variations centered around a good absolute validation error. Furthermore, although we ultimately care about out-of-sample error, if our plots looked very different when measuring the steps needed to reach a particular *training* error, then we would need to include both curves when presenting our results. However, switching to training error does not change the plots much at all (see Figure 12 in the Appendix).

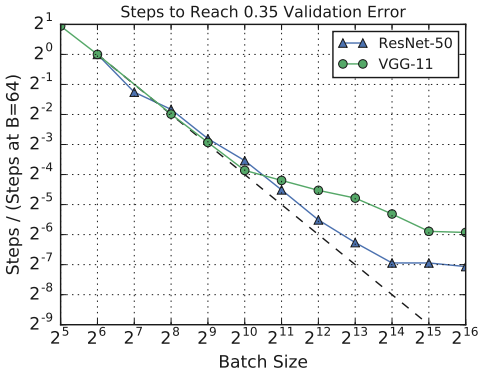
Our experiments depend on extensive metaparameter tuning for the learning rate, momentum, and, where applicable, the learning rate schedule. For each experiment, we verified our metaparameter search space by checking that the optimal trial was not too close to a boundary of the space. See Figures 13 and 14 in the Appendix for examples of how we verified our search spaces.

4.3 Some Models Can Exploit Much Larger Batch Sizes Than Others

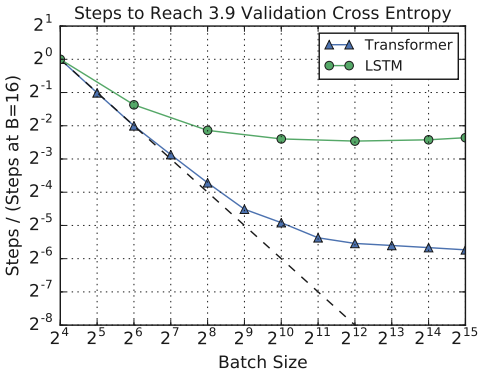
We investigated whether some models can make more use of larger batches than others by experimenting with different models while keeping the data set and optimizer fixed. We explored this question in two ways: (i) by testing completely different model architectures on the same data set, and (ii) by varying the size (width and depth) of a model within a particular model family. Since the absolute number of steps needed to reach a goal validation error depends on the model, the steps to result vs. batch size curves for each model generally appear at different vertical offsets from each other. Since we primarily care about the locations of the perfect scaling, diminishing returns, and maximal data parallelism regions, we normalized the y-axis of each plot by dividing by the number of steps needed to reach the goal for a particular batch size and data set. This normalization corresponds to a vertical shift of each curve (on log-scale plots), and makes it easier to compare different models. Appendix D contains all plots in this section without the y-axis normalized.



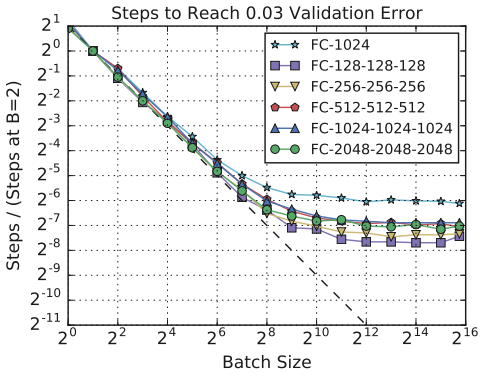
(a) Fully Connected vs Simple CNN on MNIST



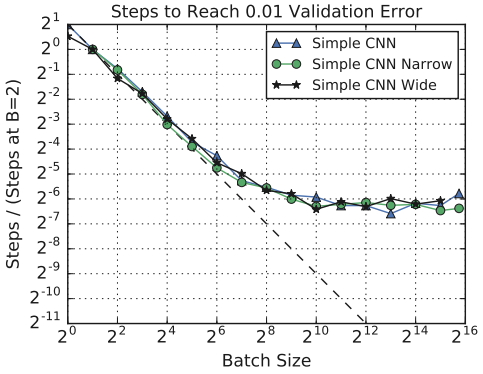
(b) ResNet-50 vs VGG-11 on ImageNet



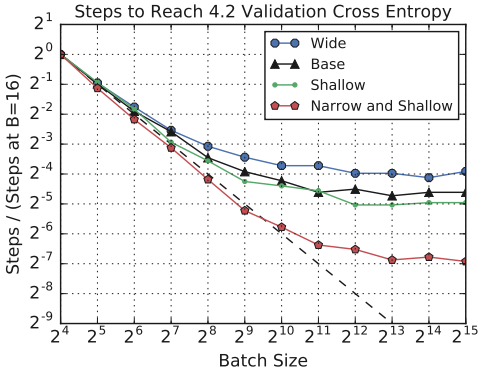
(c) Transformer vs LSTM on LM1B



(d) Fully Connected sizes on MNIST



(e) Simple CNN sizes on MNIST



(f) Transformer sizes on LM1B

Figure 3: **Some models can exploit much larger batch sizes than others.** Figures 3a-3c show that some model architectures can exploit much larger batch sizes than others on the same data set. Figures 3d-3f show that varying the depth and width can affect a model's ability to exploit larger batches, but not necessarily in a consistent way across different model architectures. All MNIST models in this Figure used plain mini-batch SGD, while all other models used Nesterov momentum. The goal validation error for each plot was chosen to allow all model variants to achieve that error. Figure 15 in the Appendix contains these plots without the y-axis normalized.

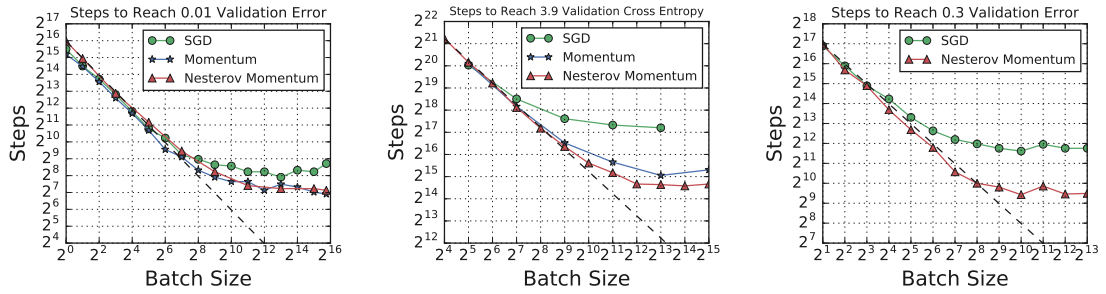
Figures 3a–3c show that the model architecture significantly affects the relationship between batch size and the number of steps needed to reach a goal validation error. In Figure 3a, the curve for the Fully Connected model flattens later than for the Simple CNN model on MNIST (although in this case the Simple CNN model can ultimately achieve better performance than the Fully Connected model). In Figure 3b, the curve for ResNet-50 flattens much later than the curve for VGG-11, indicating that ResNet-50 can make better use of large batch sizes on this data set. Unlike ResNet-50, VGG-11 does not use batch normalization or residual connections. Figure 3c shows that Transformer can make better use of large batch sizes than LSTM on LM1B.

Figures 3d–3f show that varying the depth and width can affect a model's ability to exploit larger batches, but not necessarily in a consistent way across different model architectures. In Figure 3d, the regions of perfect scaling, diminishing returns, and maximum useful batch size do not change much when the width is varied for the Fully Connected model on MNIST, although the shallower model seems less able to exploit larger batches than the deeper models. This contrasts with the findings of Chen et al. (2018), although they changed width and depth simultaneously while keeping the number of parameters fixed. For Simple CNN on MNIST, the relationship between batch size and steps to a goal validation error seems not to depend on width at all (Figure 15e in the Appendix shows that the curves are the same even when the y-axis is not normalized). However, in Figure 3f, the curves for *narrower* Transformer models on LM1B flatten later than for wider Transformer models, while the depth seems to have less of an effect. Thus, reducing width appears to allow Transformer to make more use of larger batch sizes on LM1B.

4.4 Momentum Extends Perfect Scaling to Larger Batch Sizes, but Matches Plain SGD at Small Batch Sizes

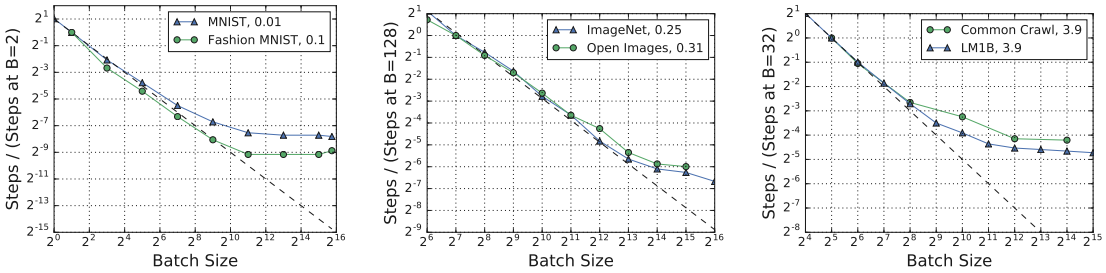
We investigated whether some optimizers can make better use of larger batches than others by experimenting with plain SGD, SGD with momentum, and Nesterov momentum on the same model and data set. Since plain SGD is a special case of both Nesterov momentum and SGD with momentum (with $\gamma = 0$ in each case), and since we tune γ in all experiments, we expect that experiments with either of these optimizers should do no worse than plain SGD at any batch size. However, it is not clear *a priori* whether momentum optimizers should outperform SGD, either by taking fewer training steps or by extending the perfect scaling region to larger batch sizes.

Figure 4 shows that Nesterov momentum and SGD with momentum can both extend the perfect scaling region beyond that achieved by SGD, and thus can significantly reduce the number of training steps required to reach a goal validation error at larger batch sizes. However, at batch sizes small enough that all optimizers are within their perfect scaling region, momentum optimizers perform identically to SGD without momentum. Though initially surprising, this identical performance at small batch sizes is consistent with observations made in Kidambi et al. (2018). In our experiments, we did not see a large difference between Nesterov momentum and SGD with momentum—Nesterov momentum appears to scale slightly better for Transformer on LM1B, but both perform about equally well for Simple CNN on MNIST.



(a) Simple CNN on MNIST (b) Transformer Shallow on LM1B (c) ResNet-8 on CIFAR-10

Figure 4: **Momentum extends perfect scaling to larger batch sizes, but matches plain SGD at small batch sizes.** Nesterov momentum and SGD with momentum can both significantly reduce the absolute number of training steps to reach a goal validation error, and also significantly extend the perfect scaling region and thus better exploit larger batches than plain mini-batch SGD.



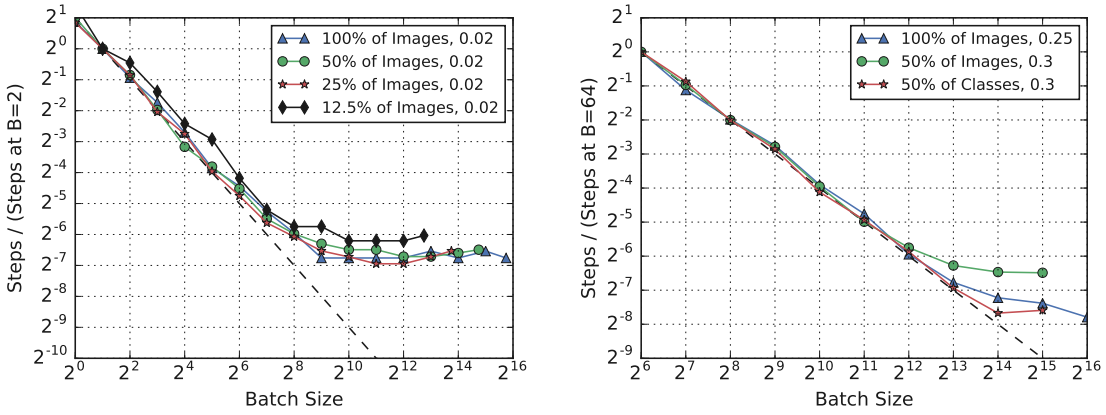
(a) Simple CNN on different data sets (b) ResNet-50 on different data sets (c) Transformer on different data sets

Figure 5: **The data set can influence the maximum useful batch size**. For the data sets shown in this plot, these differences are not simply as straightforward as larger data sets making larger batch sizes more valuable. Appendix A.2 describes the evaluation metric used for each data set, and the plot legends show the goal metric value for each task. Figure 16 in the Appendix contains these plots without the y-axis normalized.

4.5 The Data Set Matters, at Least Somewhat

We investigated whether properties of the data set make some problems able to exploit larger batch sizes than others by experimenting with different data sets while keeping the model and optimizer fixed. We approached this in two ways: (i) by testing the same model on completely different data sets, and (ii) by testing the same model on different subsets of the same data set. We normalized the y-axis of all plots in this section in the same way as Section 4.3. Appendix D contains all plots in this section without the y-axis normalized.

Figure 5 shows that changing the data set can affect the relationship between batch size and the number of steps needed to reach a goal validation error. Figure 5a shows that Fashion MNIST deviates from perfect scaling at a slightly larger batch size than MNIST for the Simple CNN model. Figure 5b shows that ImageNet and Open Images are extremely similar in how well ResNet-50 can make use of larger batch sizes, although, if anything, ImageNet might make slightly better use of larger batch sizes. Figure 5c shows that LM1B scales slightly better with increasing batch size than Common Crawl for Transformer. Since



(a) Simple CNN on MNIST subsets

(b) ResNet-50 on ImageNet subsets

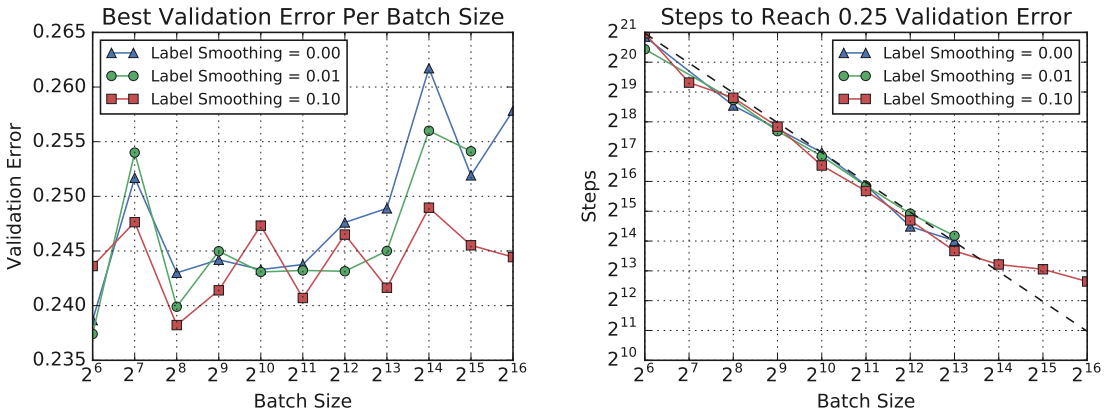
Figure 6: **Investigating the effect of data set size.** At least for MNIST, any effect of subset size on the maximum useful batch size is extremely small or nonexistent. For ImageNet, the random subset of half the images deviates from perfect scaling sooner than the full data set, but the curve for the subset with half the classes is very close to the curve for the full data set and, if anything, deviates from perfect scaling later. Appendix A.2 describes the evaluation metric used for each data set, and the plot legends show the goal metric value for each task. Figure 17 in the Appendix contains these plots without the y-axis normalized.

Fashion MNIST is the same size as MNIST, Open Images is larger than ImageNet, and Common Crawl is far larger than LM1B, these differences are not simply as straightforward as larger data sets making larger batch sizes more valuable.

To disentangle the effects from changes to the distribution and changes to the number of examples, we generated steps to result vs batch size plots for different random subsets of MNIST (Figure 6a) and ImageNet (Figure 6b). For MNIST, we selected subsets of different sizes, while for ImageNet, we selected a random subset of half the images and a similar sized subset that only includes images from half of the classes. At least on MNIST, any effect on the maximum useful batch size is extremely small or nonexistent. For ImageNet, Figure 6b shows that the random subset of half the images deviates from perfect scaling sooner than the full data set, but the curve for the subset with half the classes is very close to the curve for the full data set and, if anything, deviates from perfect scaling later, even though it contains roughly the same number of images as the random subset.

4.6 Regularization Can Be More Helpful at Some Batch Sizes Than Others

We used label smoothing (Szegedy et al., 2016) to regularize training in our experiments with ResNet-50 on ImageNet. Without label smoothing, we could not achieve our goal validation error rate of 0.25 with batch sizes greater than 2^{14} within our training budget. With a fixed compute budget for each batch size, label smoothing improved the error by as much as one percentage point at large batch sizes, while having no apparent effect at small batch sizes (Figure 7a). Meanwhile, if multiple choices for the label smoothing metaparameter achieved the goal within the training budget, then label smoothing did not change the number of steps needed (Figure 7b).



(a) Label smoothing benefits larger batch sizes, but has no apparent effect for smaller batch sizes.

(b) Label smoothing has no apparent effect on training speed, provided the goal error is achieved.

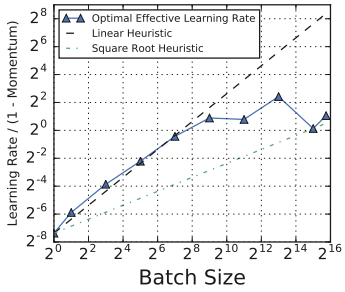
Figure 7: **Regularization can be more helpful at some batch sizes than others.** Plots are for ResNet-50 on ImageNet. Each point corresponds to a different metaparameter tuning trial, so the learning rate, Nesterov momentum, and learning rate schedule are independently chosen for each point. The training budget is fixed for each batch size, but varies between batch sizes.

We confirmed that label smoothing reduced overfitting at large batch sizes for ResNet-50 on ImageNet (see Figure 18 in the Appendix). This is consistent with the idea that noise from small batch training is a form of implicit regularization (e.g. Goodfellow et al., 2016). However, although our results show that *other forms of regularization can serve in place of this noise*, it might be difficult to select and tune other forms of regularization for large batch sizes. For example, we unsuccessfully tried to control overfitting with larger batch sizes by increasing the L_2 weight penalty and by applying additive Gaussian gradient noise before we obtained good results with label smoothing.

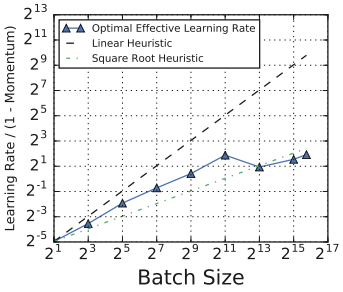
Finally, we also tried label smoothing with Simple CNN on MNIST and Fashion MNIST, and found that it generally helped *all* batch sizes, with no consistent trend of helping smaller or larger batch sizes more (see Figure 19 in the Appendix), perhaps because these data sets are sufficiently small and simple that overfitting is an issue at all batch sizes.

4.7 The Best Learning Rate and Momentum Vary with Batch Size

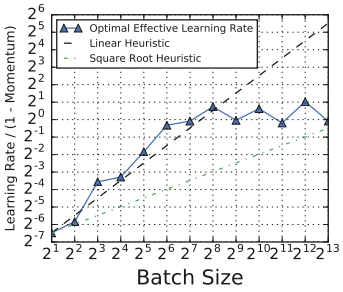
Across all problems we considered, the effective learning rate (η^{eff}; see Section 2.2) that minimized the number of training steps to a goal validation error tended to increase with increasing batch size (Figure 8). However, it did not always follow either a linear or square root scaling heuristic, despite the popularity of these rules of thumb. In some cases, the optimal effective learning rate even decreased for larger batch sizes. We also found that the best effective learning rate should be chosen by jointly tuning the learning rate and momentum, rather than tuning only the learning rate. For example, the optimal way to scale the effective learning rate for Transformer was to increase the momentum while decreasing the learning rate or holding it constant (see Figures 21 and 22 in the Appendix). This is a refinement to past prescriptions that only change the learning rate while keeping the momentum fixed.



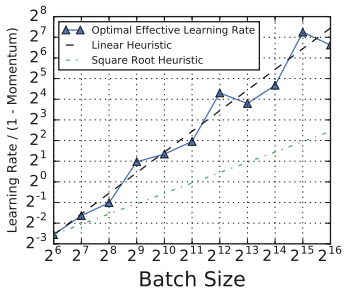
(a) Simple CNN on MNIST



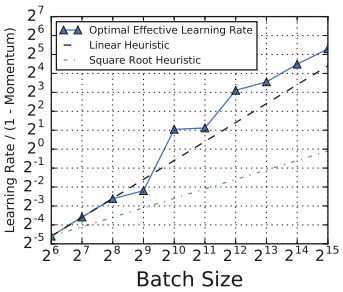
(b) Simple CNN on Fashion MNIST



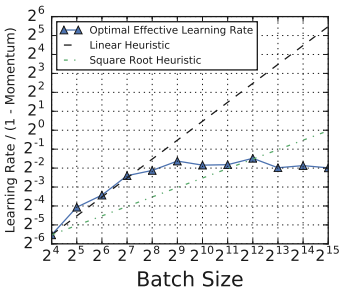
(c) ResNet-8 on CIFAR-10



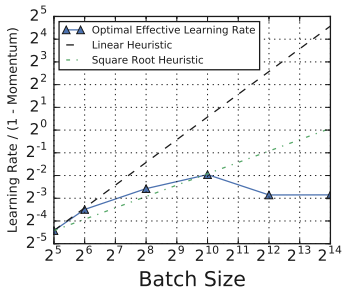
(d) ResNet-50 on ImageNet



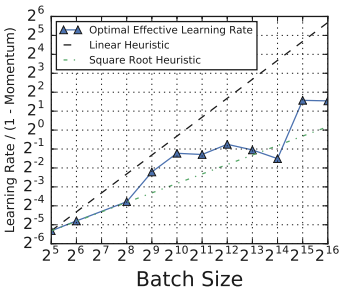
(e) ResNet-50 on Open Images



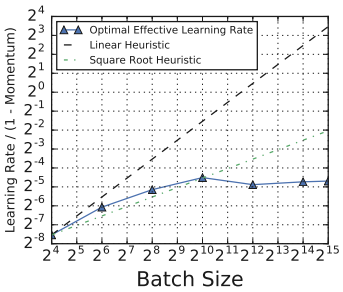
(f) Transformer on LM1B



(g) Transformer on Common Crawl

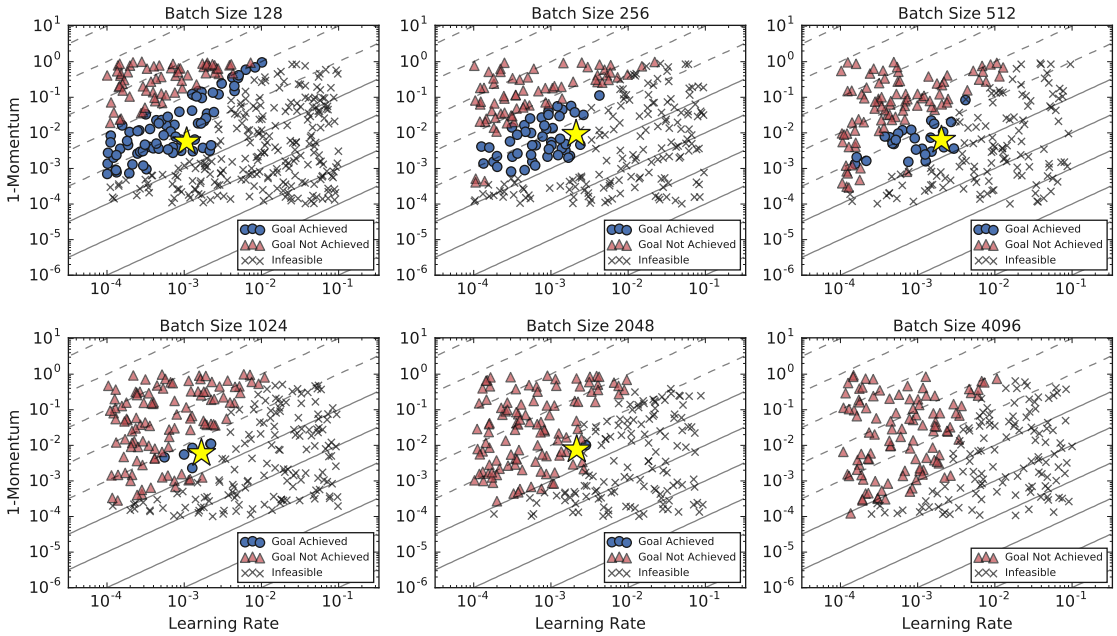


(h) VGG-11 on ImageNet

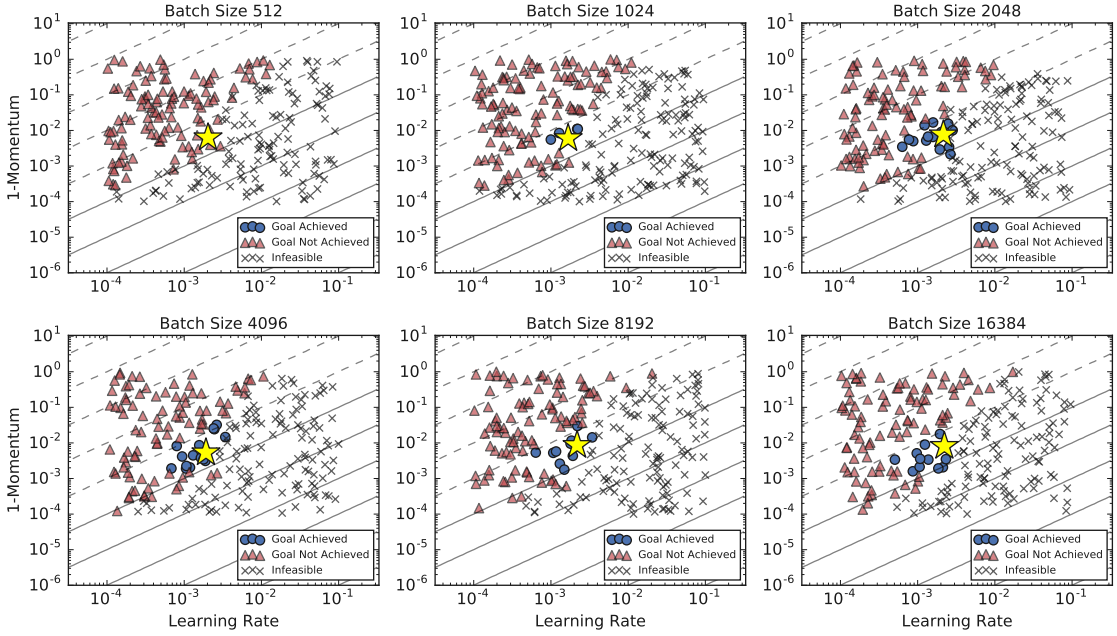


(i) LSTM on LM1B

Figure 8: **Optimal effective learning rates do not always follow linear or square root scaling heuristics.** Effective learning rates correspond to the trial that reached the goal validation error in the fewest training steps (see Figure 1). For models that used learning rate decay schedules (ResNet-8, ResNet-50, VGG-11), plots are based on the initial learning rate. See Figures 21 and 22 in the Appendix for separate plots of the optimal learning rate and momentum.



(a) Transformer on LM1B with a training budget of one epoch.



(b) Transformer on LM1B with a training budget of 25,000 steps.

Figure 9: **With increasing batch size, the region in metaparameter space corresponding to rapid training in terms of epochs becomes smaller, while the region in metaparameter space corresponding to rapid training in terms of step-count grows larger.** Yellow stars are the trials that achieved the goal in the fewest number of steps. Contours indicate the effective learning rate $\eta^{\text{eff}} = \frac{\eta}{1-\gamma}$. Infeasible trials are those that resulted in divergent training.

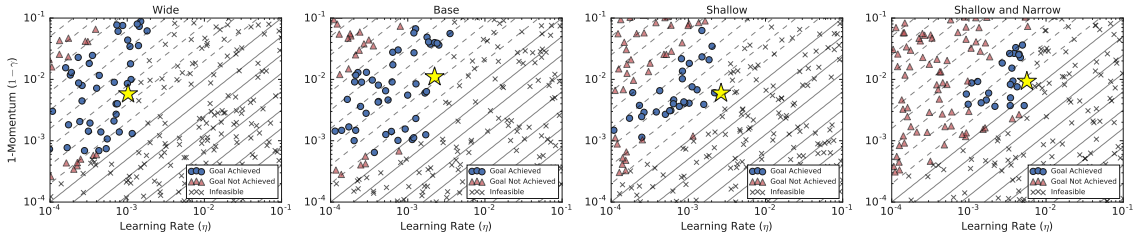


Figure 10: **Smaller models have larger stable learning rates for Transformer on LM1B.** Plots are for different sizes of Transformer on LM1B with a batch size of 1024, a goal validation cross entropy error of 4.2, and a training budget of 50,000 steps. Contours indicate the effective learning rate $\eta^{\text{eff}} = \frac{\eta}{1-\gamma}$. Infeasible trials are those that resulted in divergent training.

We further investigated the relationship between learning rate, momentum, and training speed by examining our metaparameter search spaces for different batch sizes and model sizes. For this analysis, we used Transformer on LM1B with Nesterov momentum because the metaparameter search spaces are consistent between all batch and model sizes, and can be easily visualized because they consist only of the constant learning rate η and the momentum γ. We observe the following behaviors:

- With increasing batch size, the region in metaparameter space corresponding to rapid training in terms of epochs becomes smaller (Figure 9a, consistent with the findings of Breuel, 2015b), while the region in metaparameter space corresponding to rapid training in terms of step-count grows larger (Figure 9b, although it eventually plateaus for batch sizes in the maximal data parallelism regime). Thus, with a fixed error goal and in a setting where training epochs are constrained (e.g. a compute budget), it may become more challenging to choose good values for the metaparameters with increasing batch size. Conversely, with a fixed error goal and in a setting where training steps are constrained (e.g. a wall-time budget), it may become easier to choose good values for the metaparameters with increasing batch size.

- The metaparameters yielding the fastest training are typically on the edge of the feasible region of the search space (Figure 9). In other words, small changes in the optimal metaparameters might make training diverge. This behavior may pose a challenge for metaparameter optimization techniques, such as Gaussian Process approaches, that assume a smooth relationship between metaparameter values and model performance. It could motivate techniques such as learning rate warm-up that enable stability at larger eventual learning rates, since the maximum stable learning rate depends on the current model parameters. We did not observe the same behavior for ResNet-50 on ImageNet. Figure 20 in the Appendix shows the results for a range of effective learning rates near the optimum for ResNet-50 on ImageNet and Transformer on LM1B.

- Smaller models have larger stable learning rates (Figure 10). This is consistent with recent work predicting that the largest stable learning rate is inversely proportional to layer width (Karakida et al., 2018).

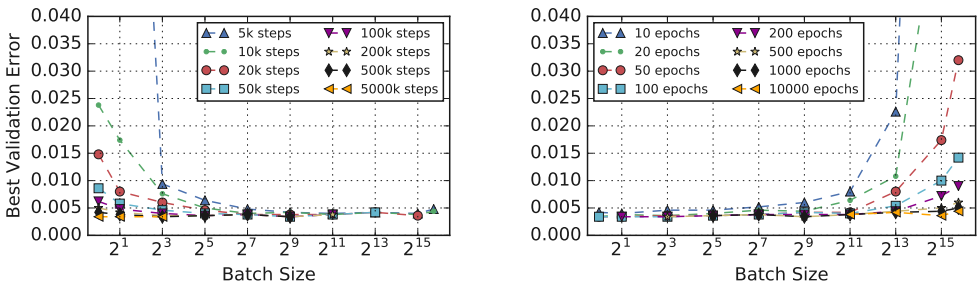
4.8 Solution Quality Depends on Compute Budget More Than Batch Size

We investigated the relationship between batch size and out-of-sample error for Simple CNN on MNIST and Fashion MNIST, and for two sizes of Transformer on LM1B. For each task, we ran a quasi-random metaparameter search over the constant learning rate η and Nesterov momentum γ. For MNIST and Fashion MNIST, we also added label smoothing and searched over the label smoothing parameter in $\{0, 0.1\}$ to mitigate any confounding effects of overfitting (see Section 4.6). We ran 100 metaparameter trials for each batch size with a large practical wall-time budget.

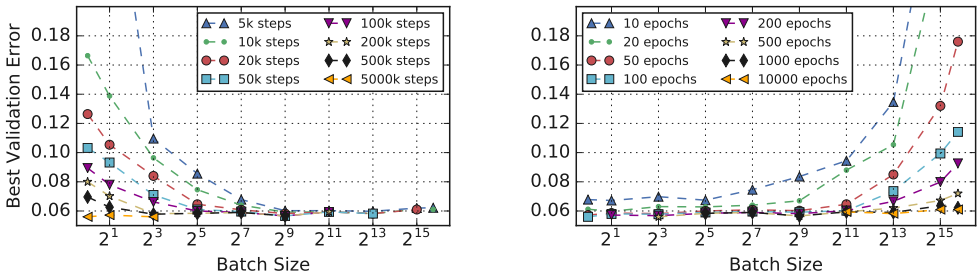
To disentangle the effects of the batch size from the compute budget, we compared batch sizes subject to budgets of either training steps or training epochs. For each batch size and compute budget, we found the model checkpoint that achieved the best validation accuracy across all metaparameter trials, and across all training steps that fell within the compute budget. Figure 11 shows the validation error for these best-validation-error checkpoints, as a function of batch size, for a range of compute budgets. We observe that, subject to a budget on training steps, larger batch sizes achieve better out-of-sample error than smaller batch sizes, but subject to a budget on training epochs, smaller batch sizes achieve better out-of-sample error than larger batch sizes. These observations are likely explained by the observations that, for a fixed number of training steps, larger batch sizes train on more data, while for a fixed number of epochs, smaller batch sizes perform more training steps.

The workloads in Figure 11 represent two distinct modes of neural network training. For the small MNIST and Fashion MNIST data sets, we used training budgets that would saturate (or almost saturate) performance at each batch size. In other words, out-of-sample error cannot be improved by simply increasing the budget, with caveats due to practical limitations on our ability to find optimal values for the metaparameters. Figures 11a and 11b show that differences in maximum performance between batch sizes on these data sets are very small (see Figures 23 and 24 in the Appendix for zoomed versions of these plots). We cannot rule out that any differences at this magnitude are due to noise from metaparameter choices and training stochasticity. Thus, for these workloads at least, the effect of batch size on solution quality is either very small or nonexistent. On the other hand, we cannot saturate performance with Transformer on LM1B within a practical training time. In this case, Figures 11c and 11d show that the best error is simply achieved by the largest compute budget.

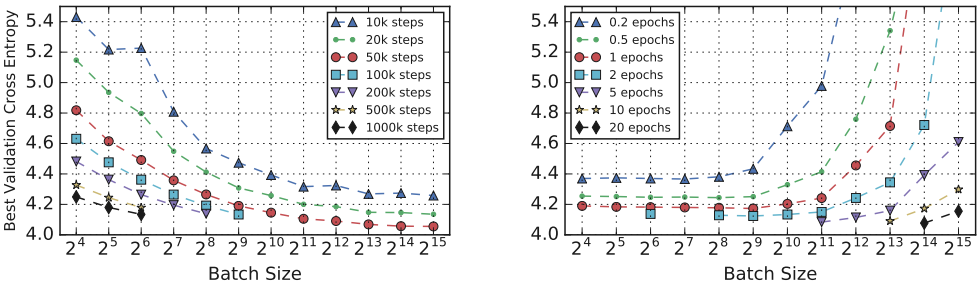
Taken together, these observations suggest that in practice *the relevant question is not which batch size leads to the best performance, but rather how compute budget varies as a function of batch size.* Although we tried our best to saturate performance with MNIST and Fashion MNIST, we found that it took millions of training steps for small batch sizes, and thousands of epochs for large batch sizes, even for data sets as small and simple as these. Indeed, despite sampling 100 metaparameter configurations per batch size and training for up to 25 hours per configuration, it is still not certain whether we truly saturated performance at the smallest and largest batch sizes (see Figures 23 and 24 in the Appendix). Thus, the regime of saturated performance is of limited practical concern for most workloads—the compute budget required to saturate performance is likely beyond what a practitioner would typically use. For realistic workloads, practitioners should be most concerned with identifying the batch size at which they can most efficiently apply their compute.



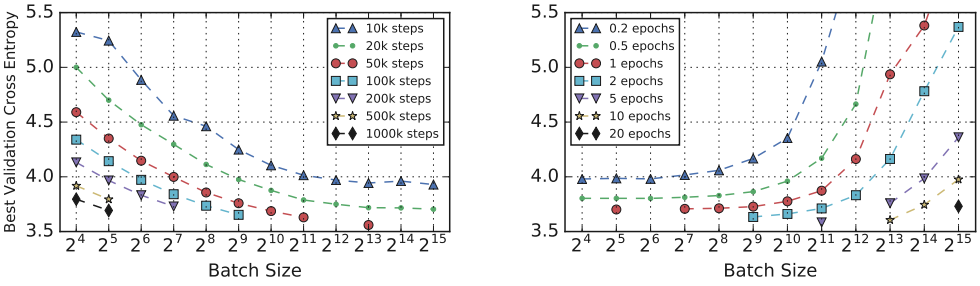
(a) Simple CNN on MNIST



(b) Simple CNN on Fashion MNIST



(c) Transformer (narrow and shallow) on LM1B



(d) Transformer (base) on LM1B

Figure 11: **Validation error depends on compute budget more than batch size.** Plots show the best validation error subject to budgets of training steps (left column) or training epochs (right column). Step budgets favor large batch sizes, while epoch budgets favor small batch sizes.

5. Discussion

Our goals in measuring the effects of data parallelism on neural network training were twofold: first, we hoped to produce actionable advice for practitioners, and second, we hoped to understand the utility of building systems capable of very high degrees of data parallelism. Our results indicate that, for idealized data parallel hardware, there is a universal relationship between training time and batch size, but there is dramatic variation in how well different workloads can make use of larger batch sizes. Across all our experiments, increasing the batch size initially reduced the number of training steps needed proportionally. However, depending on the workload, this perfect scaling regime ended anywhere from a batch size of 2^4 to a batch size of 2^{13}. As batch size increases beyond the perfect scaling regime, there are diminishing returns (where increasing the batch size by a factor of k only reduces the number of training steps needed by a factor less than k) that end with a maximum useful batch size (where increasing the batch size no longer changes the number of training steps needed). Once again, the maximum useful batch size is extremely problem-dependent and varied between roughly 2^9 and 2^{16} in our experiments. Other workloads may have the region of perfect scaling end at batch sizes even smaller or larger than the range we observed, as well as having even smaller or larger maximum useful batch sizes.

On the one hand, the possibility that perfect scaling can extend to batch sizes beyond 2^{13} for some workloads is good news for practitioners because it suggests that efficient data-parallel systems can provide extremely large speedups for neural network training. On the other hand, the wide variation in scaling behavior across workloads is bad news because any given workload might have a maximum useful batch size well below the limits of our hardware. Moreover, for a new workload, measuring the training steps needed as a function of batch size and confirming the boundaries of the three basic scaling regimes requires expensive experiments. In this work, we have only described how to retrospectively predict the scaling behavior by tuning the optimization metaparameters for every batch size. Although Golmant et al. (2018) also described the same basic scaling behavior we found, in their experiments the relationship did not appear consistently across problems, across error goals, or in out-of-sample error. In light of our own results, the heuristics they assumed for adjusting the learning rate as a function of batch size are the likely cause of these inconsistencies, but this explanation only drives home the inconvenience of having to carefully tune at every new batch size. We were unable to find reliable support for any of the previously proposed heuristics for adjusting the learning rate as a function of batch size. Thus we are forced to recommend that practitioners tune all optimization parameters anew when they change the batch size or they risk masking the true behavior of the training procedure.

If the scaling behavior of workloads with respect to batch size has a simple dependence on properties of the workload, then we might be able to predict the limits of perfect scaling (or the maximum useful batch size) before running extensive experiments. We could then prioritize workloads to run on specialized hardware or decide whether gaining access to specialized hardware would be useful for a given workload of interest. On the one hand, our results are bad news for practitioners because they show that accurate scaling predictions must depend on a combination of non-obvious properties of the model, optimizer, and data set. On the other hand, we have a lot of control over the choice of model and optimizer

and there is some indication that they might be responsible for the largest portion of the variation between workloads. Our results comparing SGD and SGD with momentum (or Nesterov momentum) show that, at least for the problems we tried, momentum can extend perfect scaling to much larger batch sizes, offering clear guidance for practitioners. Other optimizers, such as KFAC (Martens and Grosse, 2015; Grosse and Martens, 2016; Ba et al., 2017), or optimization techniques designed specifically for massively data parallel systems (e.g. Li et al., 2014), might allow perfect scaling to extend much further. Intuitively, it seems plausible that optimizers that estimate local curvature information might be able to benefit more from large batches than optimizers that only use gradients.

Although the model seems to have a large effect on the maximum useful batch size and the limit of perfect scaling, our results do not give definitive answers on exactly how to design models that scale better for a given optimizer and data set. Even when we kept the model family fixed, we observed somewhat inconsistent results from changing the model width and depth. Chen et al. (2018) suggested that wider models can exploit larger batch sizes than narrower models, but their theoretical arguments only apply to linear networks and fully connected networks with a single hidden layer. In contrast, we found that *narrower* variants of the Transformer model scaled better to larger batch sizes, although it is unclear if the same notion of "width" transfers between different types of neural networks.

Unlike the model and optimizer, we generally have much less control over the data set. Unfortunately, properties of the data set also affect how well training scales in practice. Our results are equivocal on whether the number of training examples has any effect, but changing the data set entirely can certainly change the scaling behavior with respect to batch size.

Finally, our results at least partially reconcile conflicting stances in the literature on whether increasing the batch size degrades model quality. Our experiments show that:

1. Any study that only tunes the learning rate for one batch size and then uses a heuristic to choose the learning rate for other batch sizes (Goyal et al., 2017; Keskar et al., 2017; Hoffer et al., 2017; Lin et al., 2018; Devarakonda et al., 2017; Golmant et al., 2018) gives a systematic advantage to the batch size used in tuning (as well as nearby batch sizes). Our results did not show a simple relationship between the optimal learning rate and batch size that scales indefinitely (see Figures 8 and 21), so the use of simple heuristics for batch sizes sufficiently far from the base batch size could very well explain the degraded solutions and divergent training reported in prior work. Similarly, the optimal values of other metaparameters, such as the momentum and learning rate decay schedule, should not be assumed to remain constant or scale in a simple way as the batch size increases.

2. Assuming an epoch budget when comparing solution quality between batch sizes (Masters and Luschi, 2018; Goyal et al., 2017; Lin et al., 2018; Devarakonda et al., 2017), in effect, limits an investigation to the perfect scaling region of the steps to result vs batch size curve (see Figure 1). This budget favors smaller batch sizes because they will perform more optimizer steps for the same number of training examples (see Section 4.8). Certainly, there are situations where an epoch budget is appropriate, but there may exist budgets just outside the perfect scaling region that can achieve the same quality solution, and those budgets may still represent a significant reduction

in the number of training steps required. Moreover, even for a fixed model and data set, simply changing the optimizer can significantly extend the perfect scaling regime to larger batch sizes. For example, Masters and Luschi (2018) found that test performance of ResNet-8 (without batch normalization) on CIFAR-10 with a fixed epoch budget degraded after batch size 16, but considered only plain mini-batch SGD. Our experiments confirmed that perfect scaling ends at batch size 16 with plain mini-batch SGD, but using Nesterov momentum extends the perfect scaling regime to batch size 256 (see Figure 1c).

3. Assuming a step budget when comparing solution quality between batch sizes (Hoffer et al., 2017) might favor larger batch sizes because they will see more training examples for the same number of gradient updates (see Section 4.8). A step budget is likely sufficient for a larger batch size to reach *at least* the same performance as a smaller batch size: we never saw the number of steps to reach a goal validation error increase when the batch size was increased (see Figure 1).

4. Increasing the batch size reduces noise in the gradient estimates (see Equation 4). However, the noise in updates due to small batches might, in some cases, provide a helpful regularization effect (Goodfellow et al., 2016; Smith and Le, 2018). Thankfully, other regularization techniques, such as label smoothing, can replace this effect (see Section 4.6). Others have also used regularization techniques, such as data augmentation (Keskar et al., 2017) and L_2 regularization (Smith and Le, 2018), to eliminate the "generalization gap" between two batch sizes.

5. Finally, although we do not believe there is an inherent degradation in solution quality associated with increasing the batch size, depending on the compute budget, it may become increasingly difficult to find good values for the metaparameters with larger batch sizes. Specifically, increasing the batch size may shrink the region in metaparameter space corresponding to rapid training in terms of epochs (see Figure 9a), as previously reported by Breuel (2015b). On the other hand, increasing the batch size may increase the region in metaparameter space corresponding to rapid training in terms of steps (see Figure 9b).

5.1 Limitations of our experimental protocol

When interpreting our results, one should keep in mind any limitations of our experimental protocol. We do not believe any of these limitations are debilitating, and we hope that describing these potential areas of concern will spur methodological innovation in future work.

Firstly, we were unable to avoid some amount of human judgment when tuning metaparameters. Although we did not tune metaparameters by hand, we specified the search spaces for automatic tuning by hand and they may not have been equally appropriate for all batch sizes, despite our best efforts. We are most confident in our search spaces that tuned the fewest metaparameters (such as in our experiments that only tuned learning rate and momentum). We found it quite difficult to be confident that our tuning was sufficient when we searched over learning rate decay schedules; readers should be aware that the steps to result measurement is generally quite sensitive to the learning rate schedule. Thus, we

may not have sampled enough trials at some batch sizes or, nearly equivalently, our search spaces may have been too wide at some batch sizes. Even though we verified that the best trial was not on the boundary of the search space, this by no means guarantees that we found the globally optimal metaparameters.

Smaller batch sizes typically had more opportunities to measure validation error and, when validation error was noisy, got more chances to sample a lucky validation error. Batch sizes (usually larger ones) that did not reach the goal validation error using the first search space used revised search spaces that gave them an extra bite of the apple, so to speak.

Finally, our analysis does not consider how robustly we can reach a goal error rate. For instance, we did not distinguish between batch sizes where all 100 trials achieved the goal validation error and batch sizes where only one of the 100 trials achieved the goal. The maximum or minimum value over a set of trials is not usually a very robust statistic, but something like the 50^{th} percentile trial mostly reveals information about the search space. We tried to strike a balance between studying realistic workloads and being able to repeat our experiments so many times that these uncertainty questions became trivial. Ultimately, we opted to study realistic workloads and simply report results for the optimal trials.

6. Conclusions and Future Work

Increasing the batch size is a simple way to produce valuable speedups across a range of workloads, but, for all workloads we tried, the benefits diminished well within the limits of current hardware. Unfortunately, blindly increasing the batch size to the hardware limit will not produce a large speedup for all workloads. However, our results suggest that some optimization algorithms may be able to consistently extend perfect scaling across many models and data sets. Future work should perform our same measurements with other optimizers, beyond the closely-related ones we tried, to see if any existing optimizer extends perfect scaling across many problems. Alternatively, if we only need speedups for specific, high-value problems, we could also consider designing models that extend perfect scaling to much larger batch sizes. However, unlike the optimizer, practitioners are likely to tailor their model architectures to the specific problems at hand. Therefore, instead of searching for model architectures that happen to scale extremely well, future work should try to uncover general principles for designing models that can scale perfectly to larger batch sizes. Even if such principles remain elusive, we would still benefit from methods to prospectively predict the scaling behavior of a given workload without requiring careful metaparameter tuning at several different batch sizes. Finally, the deep learning community can always benefit from methodical experiments designed to test hypotheses, characterize phenomena, and reduce confusion, to balance more exploratory work designed to generate new ideas for algorithms and models.

Acknowledgements

We thank Tomer Koren for helpful discussions. We also thank Justin Gilmer and Simon Kornblith for helpful suggestions and comments on the manuscript. Finally, we thank Matt J. Johnson for lending us some computing resources.

Appendix A. Data Set Details

This section contains details of the data sets summarized in Table 1.

A.1 Data Set Descriptions and Pre-Processing

MNIST (LeCun et al., 1998) is a classic handwritten digit image classification data set with 10 mutually exclusive classes. We split the original training set into 55,000 training images and 5,000 validation images, and used the official test set of 10,000 images. We did not use data augmentation.

Fashion MNIST (Xiao et al., 2017) is another reasonably simple image classification data set with 10 mutually exclusive classes. It was designed as a drop-in replacement for MNIST. We split the original training set into 55,000 training images and 5,000 validation images, and used the official test set of 10,000 images. We did not use data augmentation.

CIFAR-10 (Krizhevsky, 2009) is an image classification data set of 32×32 color images with 10 mutually exclusive classes. We split the original training set into 45,000 training images and 5,000 validation images. We used the official test set of 10,000 images. We pre-processed each image by subtracting the average value across all pixels and channels and dividing by the standard deviation.[15] We did not use data augmentation.

ImageNet (Russakovsky et al., 2015) is an image classification data set with 1,000 mutually exclusive classes. We split the official training set into 1,281,167 training images and 50,045 test images, and used the official validation set of 50,000 images. We pre-processed the images and performed data augmentation in a similar way to Simonyan and Zisserman (2014). Specifically, at training time, we sampled a random integer $S \in [256, 512]$, performed an aspect-preserving resize so that the smallest side had length S, and took a random crop of size $(224, 224)$. We randomly reflected the images horizonally, but unlike Simonyan and Zisserman (2014), we did not distort the colors. At evaluation time, we performed an aspect-preserving resize so that the smallest side had length 256, and took a central crop of size $(224, 224)$. In both training and evaluation, we then subtracted the global mean RGB value from each pixel using the values computed by Simonyan and Zisserman (2014).[16]

Open Images v4 (Krasin et al., 2017) is a data set of 9 million images that are annotated with image-level labels and object bounding boxes.[17] The image labels were generated by a computer vision model and then verified as either *positive* or *negative* labels by human annotators. We only considered the 7,186 "trainable" classes with at least 100 human-annotated positives in the training set. We filtered the official subsets by keeping only images with at least one positive trainable label, which produced training, validation and test sets of size 4,526,492; 41,225; and 124,293 images, respectively. On average, each image in the training set has 2.9 human-annotated positive labels, while each image in the validation and test sets have 8.4 human-annotated positive labels. We only considered the human-annotated positives and assumed all other classes were negative. We pre-processed the images and performed data augmentation identically to ImageNet.

15. We used the TensorFlow op `tf.image.per_image_standardization`.
16. See `https://gist.github.com/ksimonyan/211839e770f7b538e2d8#description` for the mean RGB values used.
17. Available at `https://storage.googleapis.com/openimages/web/index.html`.

LM1B (Chelba et al., 2014) is a text data set of English news articles.[18] We used the official training set and created validation and test sets using files `news.en.heldout-00000-of-00050` and `news.en.heldout-00001-of-00050`, respectively. These splits contain 30,301,028; 6,075; and 6,206 sentences, respectively. We used an invertable word tokenizer to split the text into sub-word tokens with a vocabulary of size 32,000.[19] On average, the training set contains around 20 tokens per sentence and the validation and test sets contain around 29 tokens per sentence. At training time, we clipped long sentences to the first 64 tokens, which affected only about 2% of sentences. We did not clip long sentences at evaluation time. The maximum sentence across the validation and test sets has 476 tokens.

Common Crawl is a repository of web data containing over 3 billion web pages.[20] We filtered and processed the data set identically to Anil et al. (2018).[21] The vocabulary contains 24,006 sub-word tokens. We randomly partitioned the sentences into a training set (99.98%) and a holdout set (0.02%). Our training set contains ∼25.8 billion sentences. We used the first 6,075 sentences of the holdout set as our validation set, which is the same number of sentences in our LM1B validation set. Some sentences are tens of thousands of tokens long. To maintain consistency with our LM1B processing, we clipped sentences to 64 tokens at training time and 476 at evaluation time.

A.2 Evaluation Metrics

We use **classification error** for MNIST, Fashion MNIST, CIFAR-10, and ImageNet. To compute this metric, we consider the model's classification for each image to be the class it assigns the highest probability. Then

$$\text{classification error} = \frac{\#\text{ incorrect classifications}}{\#\text{ classifications}}.$$

We use class-agnostic **average precision** (AP) for Open Images. To compute this metric, we first rank each image-class pair by the predicted likelihood of the class being a true positive for that image. Then

$$AP = \frac{1}{w}\sum_{k=1}^{nm}\text{Precision}(k)\cdot\text{Relevance}(k), \tag{7}$$

where $\text{Precision}(k)$ is the precision when considering the top k image-class pairs, $\text{Relevance}(k)$ is an indicator function equal to 1 if the k^{th} image-class pair is a verified positive and 0 otherwise, n is the number of images in the validation set, m is the number of classes, and w is the number of positive labels. Average precision was proposed for Open Images by Veit et al. (2017). Due to false negatives in the validation set, Veit et al. (2017) only computed AP over the the human-annotated classes in each image. However, on average, each image

18. Available at `http://www.statmt.org/lm-benchmark/`.
19. The code for processing the raw data and generating the vocabulary is available at `https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/data_generators/lm1b.py`
20. Available at `http://commoncrawl.org/2017/07/june-2017-crawl-archive-now-available/`.
21. See `https://github.com/google-research/google-research/tree/master/codistillation` for document IDs.

in the validation set only has 8.4 positive and 4 negative human-annotated classes, so each image is only evaluated over ∼12 classes out of 7,186. This yields misleadingly high values of AP. Instead, we compute AP over all classes in each image, which may underestimate the true AP due to false negatives in the validation set, but is more indicative of the true performance in our experience. We compute AP using an efficient approximation of the area under the discrete precision-recall curve.[22]

We use average per-token **cross entropy error** for LM1B and Common Crawl. For a single sentence $s = (w_1, ..., w_m)$, let $p(w_j|w_1, ..., w_{j-1})$ denote the model's predicted probability of the token w_j given all prior tokens in the sentence. Thus, the predicted log-probability of s is $\log p(s) = \sum_{j=1}^{m} \log p(w_j|w_1, ..., w_{j-1})$. We compute the average per-token cross entropy error over a data set $\{s_1, ..., s_n\}$ as

$$\text{cross entropy error} = \frac{\sum_{i=1}^{n} \log p(s_n)}{\sum_{i=1}^{n} \text{len}(s_n)},$$

where $\text{len}(s)$ denotes the number of tokens in s. This is the logarithm of the per-token perplexity.

Appendix B. Model Details

In this section we give the architectural details of the models summarized in Table 2. In addition to the descriptions below, each model has a task-specific output layer. Models trained on MNIST, Fashion MNIST, CIFAR-10, and ImageNet (classification with mutually exclusive labels) use a softmax output layer to model the probability distribution over classes. Models trained on Open Images (classification with multiple labels per image) use a sigmoid output layer to model the probability of each class. Models trained on LM1B and Common Crawl (language modeling) use a softmax output layer to model the probability of the next word in a sentence given all prior words in the sentence.

Fully Connected is a fully connected neural network with ReLU activation function. Hidden layers use dropout with probability 0.4 during training. We vary the number of layers and number of units per layer in different experiments to investigate the impact of model size. We use the notation FC-N_1-...-N_k to denote a fully connected neural network with k hidden layers and N_i units in the i^{th} layer.

Simple CNN consists of 2 convolutional layers with max-pooling followed by 1 fully connected hidden layer. The convolutional layers use 5×5 filters with stride length 1, "same" padding (Goodfellow et al., 2016), and ReLU activation function. Max pooling uses 2×2 windows with stride length 2. The fully connected layer uses dropout with probability 0.4 during training. We used three different model sizes: **base** has 32 and 64 filters in the convolutional layers and 1,024 units in the fully connected layer; **narrow** has 16 and 32 filters in the convolutional layers and 512 units in the fully connected layer; and **wide** has 64 and 128 filters in the convolutional layers and 2,048 units in the fully connected layer. We used the base model unless otherwise specified.

22. Equation 7 can be interpreted as a right Riemann sum of the discrete precision-recall curve $\{(r_i, p_i)|i = 1, ..., w\}$, where $r_i = i/w$ and p_i is the maximum precision among all values of precision with recall r_i (each value of recall may correspond to different values of precision at different classification thresholds). We use the TensorFlow op `tf.metrics.auc` with `curve="PR"`, `num_thresholds=200`, and `summation_method="careful_interpolation"`.

ResNet-8 consists of 7 convolutional layers with residual connections followed by 1 fully connected hidden layer. We used the model described in section 4.2 of He et al. (2016a) with $n = 1$, but with the improved residual block described by He et al. (2016b). We removed batch normalization, which is consistent with Masters and Luschi (2018).

ResNet-50 consists of 49 convolutional layers with residual connections followed by 1 fully connected hidden layer. We used the model described in section 4.1 of He et al. (2016a), but with the improved residual block described by (He et al., 2016b). We replaced batch normalization (Ioffe and Szegedy, 2015) with ghost batch normalization to keep the training objective fixed between batch sizes and to avoid possible negative effects from computing batch normalization statistics over a large number of examples (Hoffer et al., 2017). We used a ghost batch size of 32 for all experiments. We also applied label smoothing (Szegedy et al., 2016) to regularize the model at training time, which was helpful for larger batch sizes. The label smoothing coefficient was a metaparameter that we tuned in our experiments.

VGG-11 consists of 8 convolutional layers followed by 3 fully connected hidden layers. We used the model referred to as "model A" by Simonyan and Zisserman (2014).

LSTM is a one hidden-layer LSTM model (Hochreiter and Schmidhuber, 1997). It is a simpler variant of the LSTM-2048-512 model described by Jozefowicz et al. (2016), with 1,024 embedding dimensions, 2,048 hidden units, and 512 projection dimensions. We did not use bias parameters in the output layer because we found this improved performance in our preliminary experiments.

Transformer is a self-attention model that was originally presented for machine translation (Vaswani et al., 2017). We used it as an autoregressive language model by applying the decoder directly to the sequence of word embeddings for each sentence. We used four different sizes: the **base** model described by Vaswani et al. (2017); a **shallow** model that is identical to the base model except with only two hidden layers instead of six; a **narrow and shallow** model that is identical to the shallow model except with half as many hidden units and attention heads as well as half the filter size; and a **wide** model that is identical to the base model except with double the number of hidden units and attention heads as well as double the filter size. We used the base model unless otherwise specified.

Appendix C. Learning Rate Schedules

We chose our learning rate schedule by experimenting with a variety of different schedules for ResNet-50 on ImageNet. For each schedule, we specified the following metaparameters:

- η_0: initial learning rate
- α: decay factor ($\alpha > 0$)
- T: number of training steps until the learning rate decays from η_0 to $\alpha\eta_0$

Each schedule corresponds to a decay function $d(t)$, such that the learning rate at training step t is

$$\eta(t) = \begin{cases} d(t) \cdot \eta_0 & \text{if } t \leq T, \\ \alpha\eta_0 & \text{if } t > T. \end{cases}$$

We experimented with the following decay functions:

- **Constant**: $d(t) = 1$
- **Linear**: $d(t) = 1 - (1 - \alpha)\frac{t}{T}$
- **Cosine** (Loshchilov and Hutter, 2017): $d(t) = \alpha + \frac{(1-\alpha)}{2}\left(1 + \cos \pi \frac{t}{T}\right)$
- **Exponential Polynomial**: $d(t) = \alpha + (1 - \alpha)\left(1 - \frac{t}{T}\right)^{\lambda}$, where $\lambda > 0$
- **Inverse Exponential Polynomial**: $d(t) = \frac{\alpha}{\alpha + (1-\alpha)\left(\frac{t}{T}\right)^{\lambda}}$, where $\lambda > 0$
- **Exponential**: $d(t) = \alpha^{t/T}$

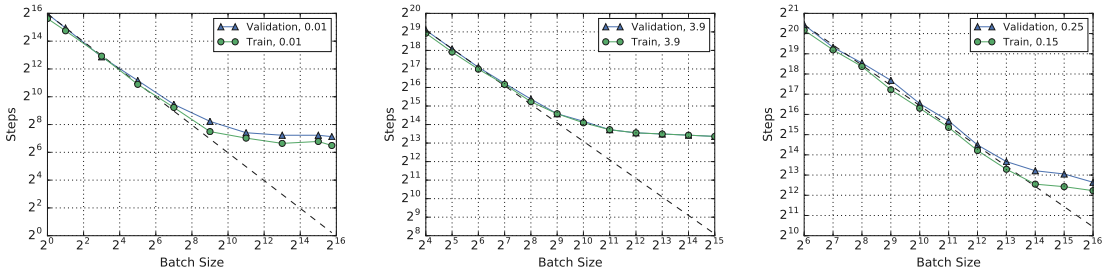
We also tried piecewise linear learning rate schedules. These schedules are specified by a sequence of pairs $\{(t_0, \eta_0), ..., (t_k, \eta_k)\}$, with $0 = t_0 < t_1 ... < t_k$, such that the learning rate at training step t is

$$\eta(t) = \begin{cases} \eta_i + \frac{\eta_{i+1} - \eta_i}{t_{i+1} - t_i}(t - t_i) & \text{if } t_i \leq t < t_{i+1}, \\ \eta_k & \text{if } t \geq t_k. \end{cases}$$

The schedules used by both He et al. (2016a) (piecewise constant) and Goyal et al. (2017) (linear warm-up followed by piecewise constant) for ResNet-50 on ImageNet can both be expressed as piecewise linear.

We ran experiments with ResNet-50 on ImageNet, using Nesterov momentum with batch size 1,024 for 150,000 training steps, while tuning the momentum and all metaparameters governing the learning rate schedule. We used quasi-random metaparameter search as described in Section 4. For piecewise linear schedules, we tried 1, 3, and 5 decay events. We found that it was possible to get good results with several of the schedules we tried, and it is likely that other schedules would also work well. Ultimately, we chose linear decay because it performed at least well as all other schedules we tried, while also being the simplest and requiring only two additional metaparameters.

Appendix D. Additional Plots



(a) Simple CNN on MNIST (b) Transformer on LM1B (c) ResNet-50 on ImageNet

Figure 12: **Steps to result on the training set is almost the same as on the validation set.** The evaluation metrics are described in Appendix A.2. Error goals are specified in the plot legends.

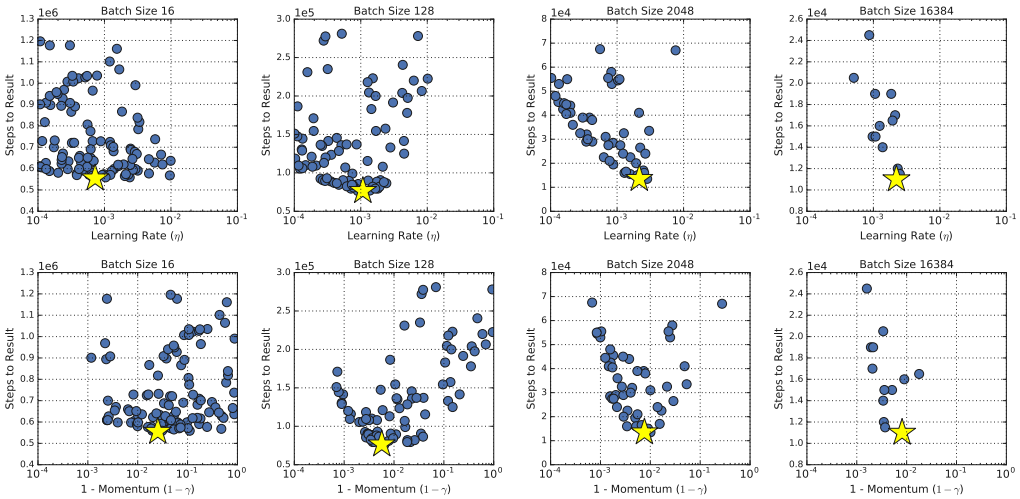


Figure 13: **Validating metaparameter search spaces for Transformer on LM1B.** Rows correspond to the metaparameters we tuned (learning rate η and momentum γ) and columns correspond to different batch sizes. The x-axis is the search range that was sampled by the quasi-random search algorithm. Blue dots represent trials that reached the goal of 3.9 validation cross entropy error, and yellow stars correspond to trials that achieved the goal in the fewest steps. We deem these search spaces appropriate because the yellow stars are not on the boundaries.

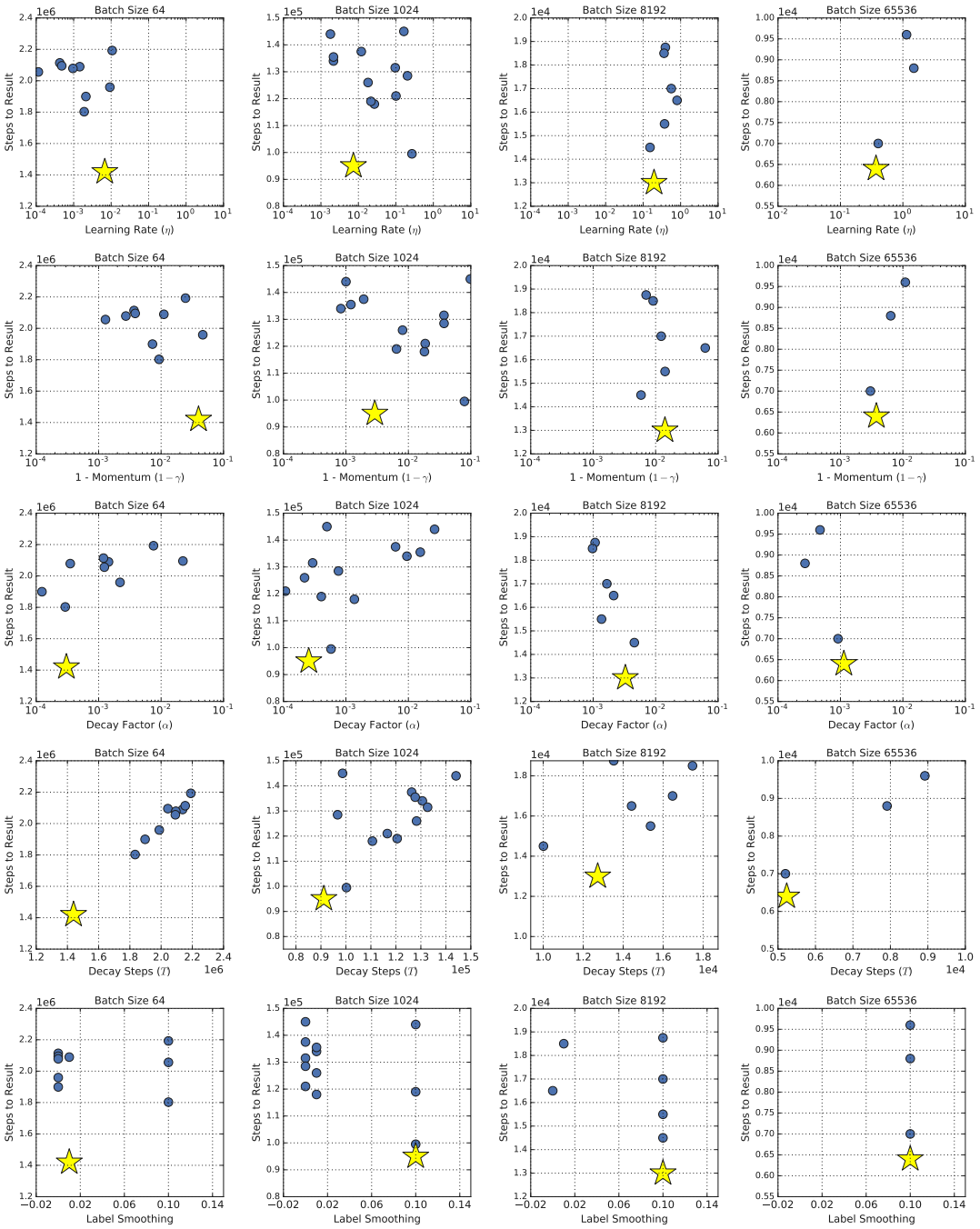
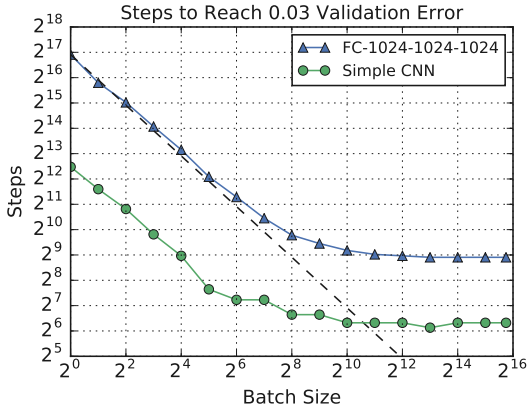
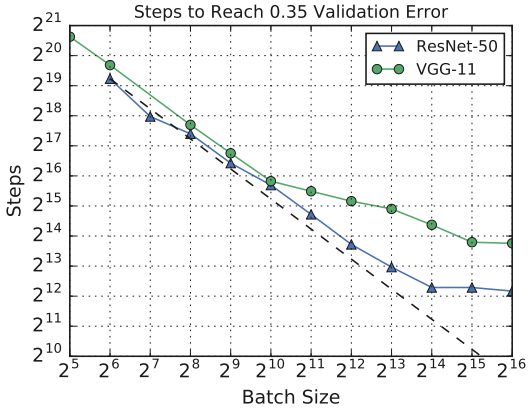


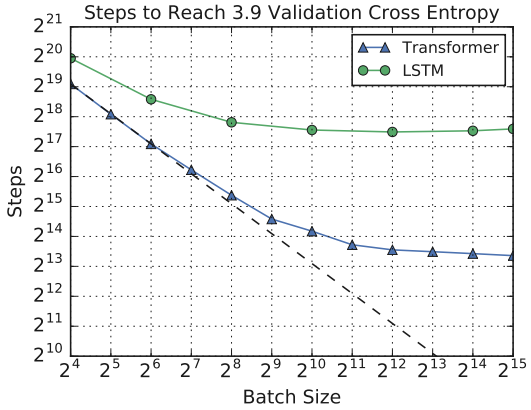
Figure 14: **Validating metaparameter search spaces for ResNet-50 on ImageNet.** Rows correspond to the metaparameters we tuned (initial learning rate η_0, momentum γ, learning rate decay parameters α, T, and label smoothing parameter) and columns correspond to different batch sizes. For all parameters except the label smoothing parameter, the x-axis is the search range sampled by the quasi-random search algorithm. The label smoothing parameter was sampled uniformly in $\{0, 0.01, 0.1\}$ for $b \leq 2^{14}$ and $\{0, 0.1\}$ for $b > 2^{14}$. Blue dots represent trials that reached the goal validation error rate of 0.25, and yellow stars correspond to trials that achieved the goal in the fewest steps. We deem these search spaces appropriate because the yellow stars are not on the boundaries.



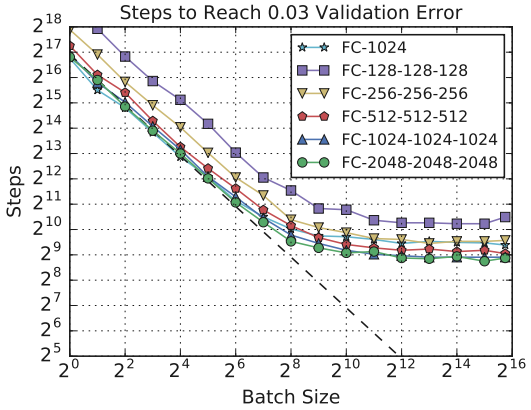
(a) Fully Connected vs Simple CNN on MNIST



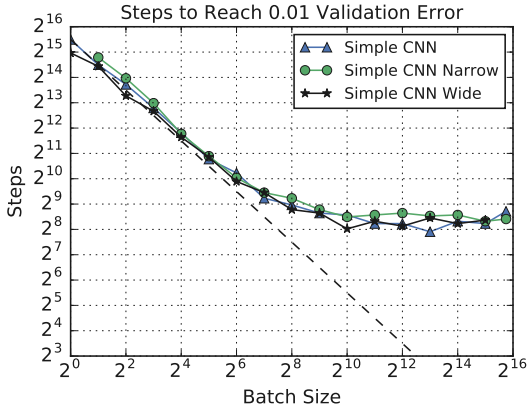
(b) ResNet-50 vs VGG-11 on ImageNet



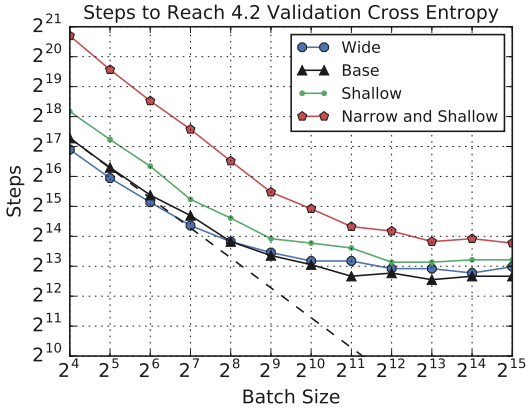
(c) Transformer vs LSTM on LM1B



(d) Fully Connected sizes on MNIST

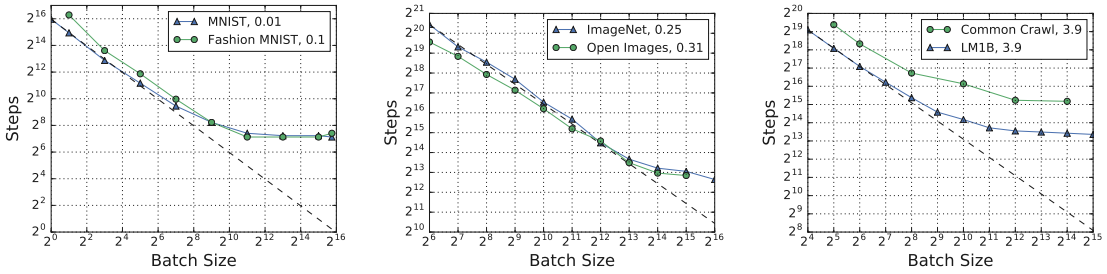


(e) Simple CNN sizes on MNIST



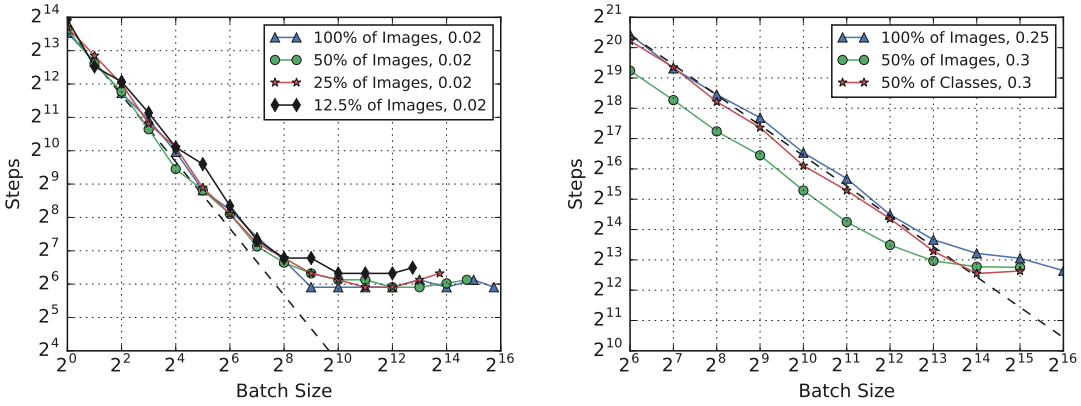
(f) Transformer sizes on LM1B

Figure 15: **Figure 3 without the y-axis normalized.**



(a) Simple CNN on different data sets (b) ResNet-50 on different data sets (c) Transformer on different data sets

Figure 16: **Figure 5 without the y-axis normalized.**



(a) Simple CNN on MNIST subsets (b) ResNet-50 on ImageNet subsets

Figure 17: **Figure 6 without the y-axis normalized.**

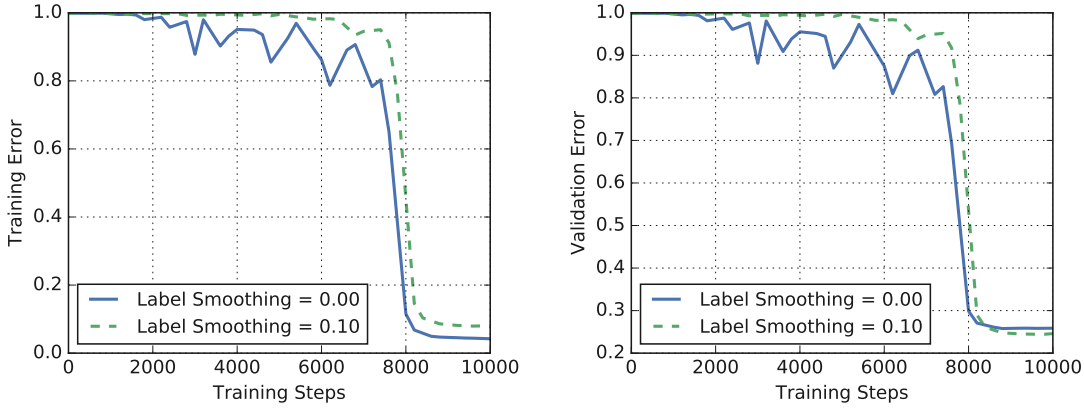
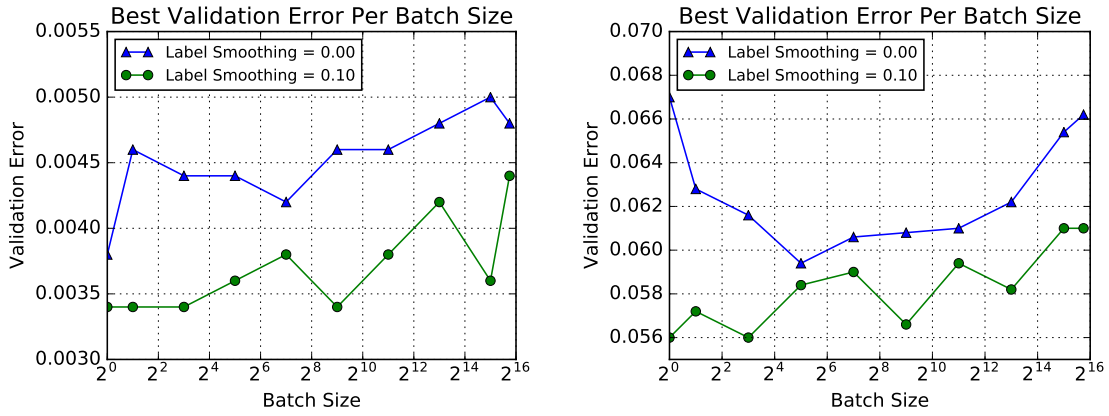


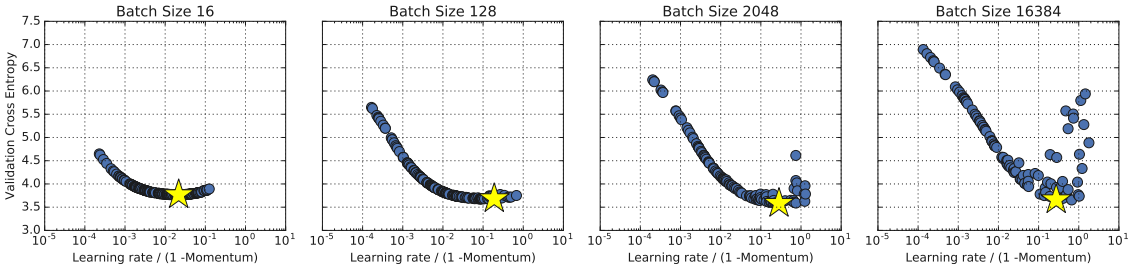
Figure 18: **Label smoothing reduces overfitting at large batch sizes.** Plots are training curves for the two best models with and without label smoothing for ResNet-50 on ImageNet with batch size 2^{16}. The two models correspond to different metaparameter tuning trials, so the learning rate, Nesterov momentum, and learning rate schedule were independently chosen for each trial. The two trials shown are those that reached the highest validation error at any point during training, for label smoothing equal to 0 and 0.1 respectively.



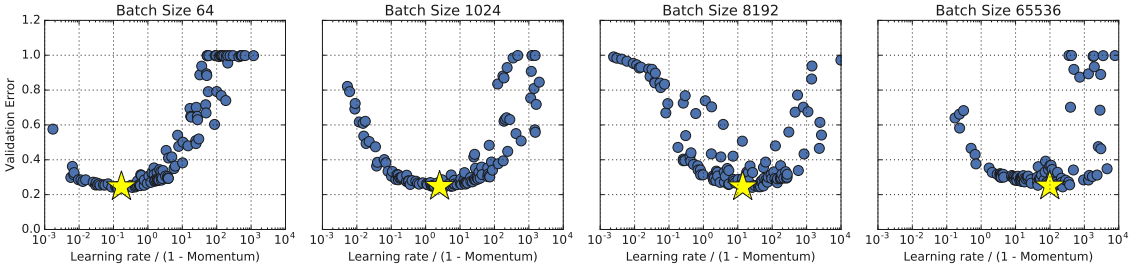
(a) Simple CNN on MNIST

(b) Simple CNN on Fashion MNIST

Figure 19: **Label smoothing helps all batch sizes for Simple CNN on MNIST and Fashion MNIST.** There is no consistent trend of label smoothing helping smaller or larger batch sizes more. Each point corresponds to a different metaparameter tuning trial, so the learning rate, Nesterov momentum, and learning rate schedule are independently chosen for each point. The training budget is fixed for each batch size, but varies between batch sizes.

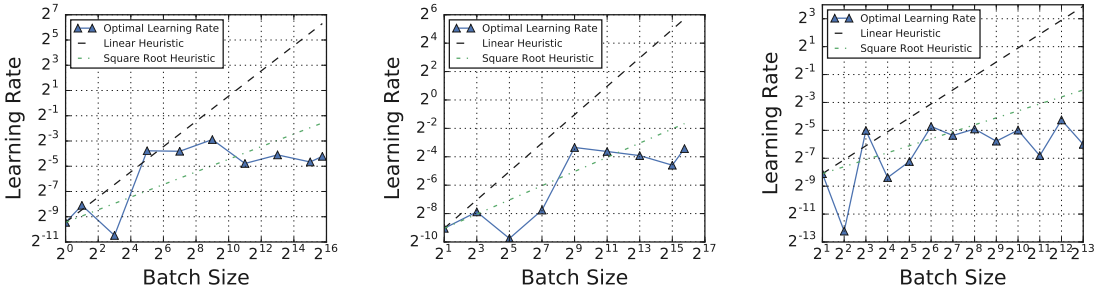


(a) Transformer on LM1B

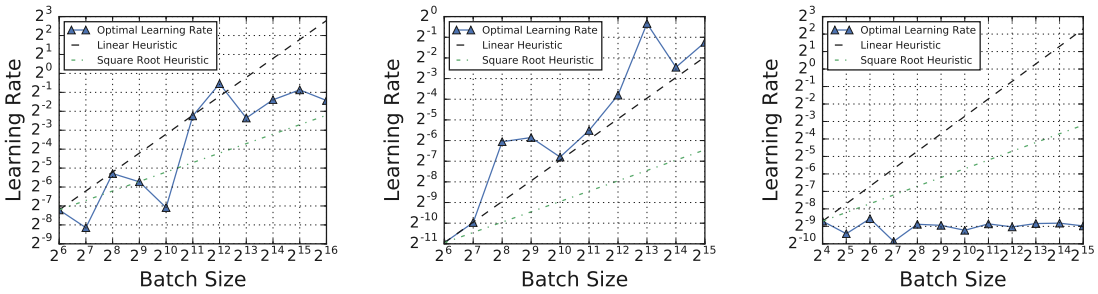


(b) ResNet-50 on ImageNet

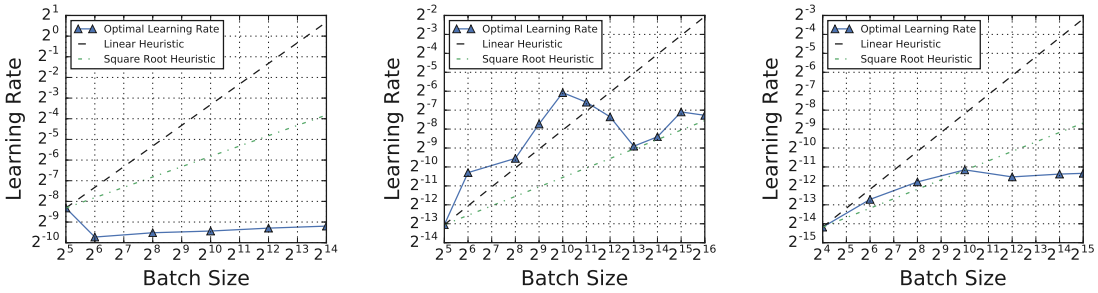
Figure 20: **Validation error vs effective learning rate.** Training budgets are consistent for each batch size, but not between batch sizes. These plots are projections of the entire metaparameter search space, which is 2-dimensional for Transformer on LM1B (see Figure 13) and 5-dimensional for ResNet-50 on ImageNet (see Figure 14).



(a) Simple CNN on MNIST (b) Simple CNN on Fashion MNIST (c) ResNet-8 on CIFAR-10

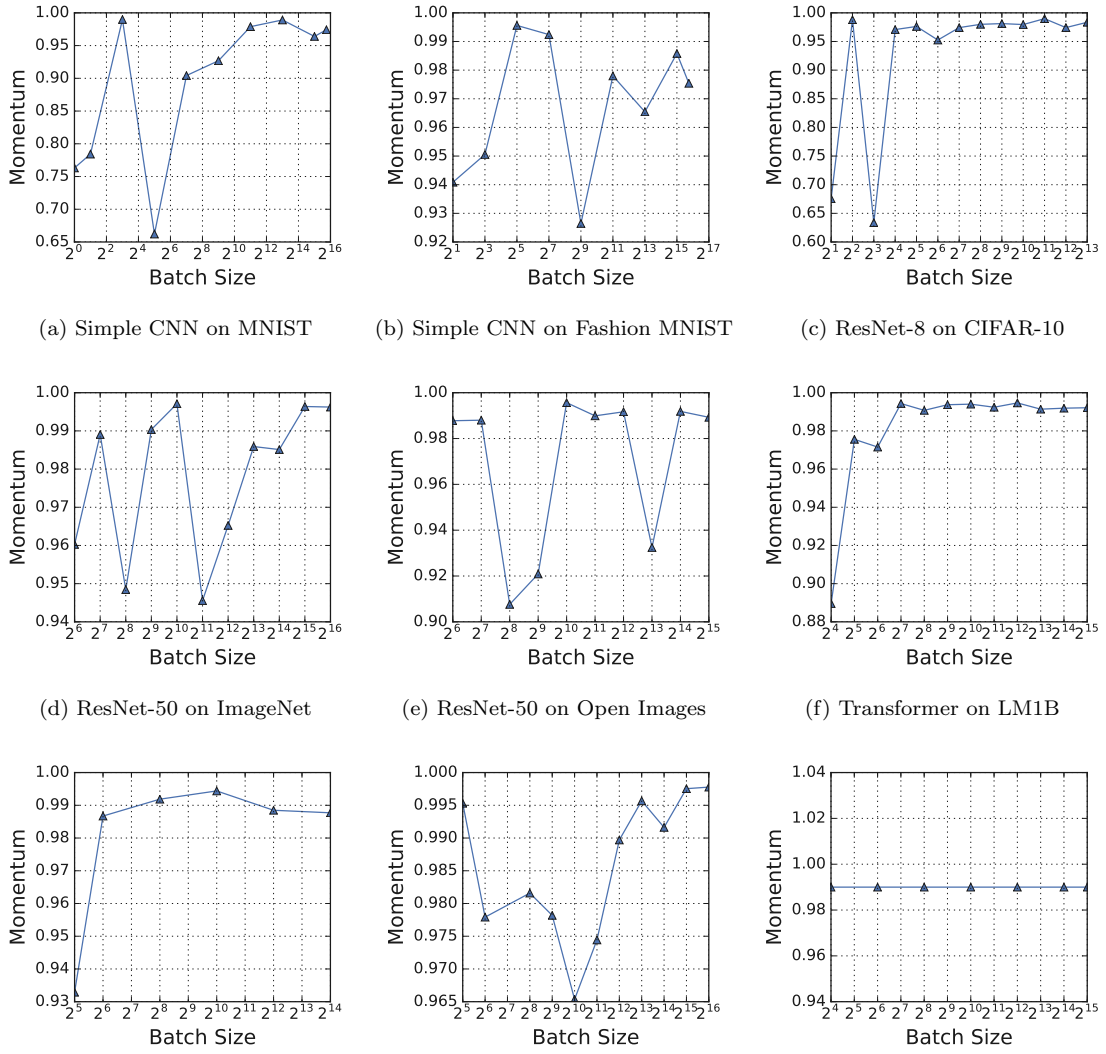


(d) ResNet-50 on ImageNet (e) ResNet-50 on Open Images (f) Transformer on LM1B



(g) Transformer on Common Crawl (h) VGG-11 on ImageNet (i) LSTM on LM1B

Figure 21: **Optimal learning rates do not always follow linear or square root scaling heuristics.** Learning rates correspond to the trial that reached the goal validation error in the fewest training steps (see Figure 1). For models using learning rate decay schedules (ResNet-8, ResNet-50, VGG-11), plots are based on the initial learning rate. See Figure 22 for the corresponding plot of optimal momentum, and Figure 8 for the corresponding plot of effective learning rate.

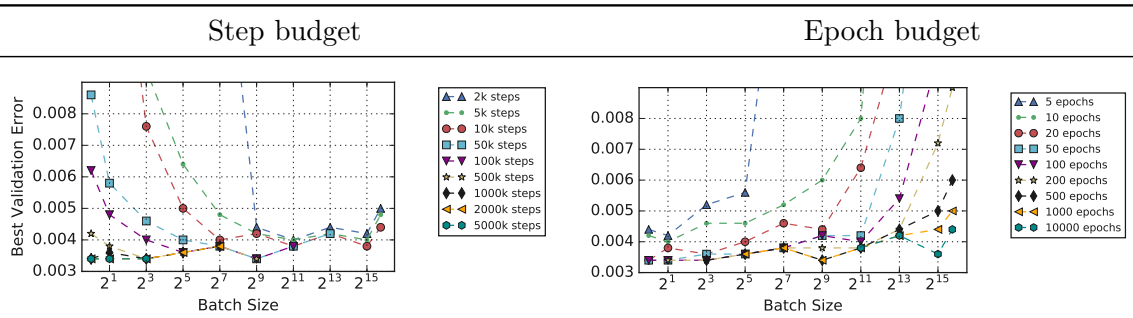


(a) Simple CNN on MNIST (b) Simple CNN on Fashion MNIST (c) ResNet-8 on CIFAR-10

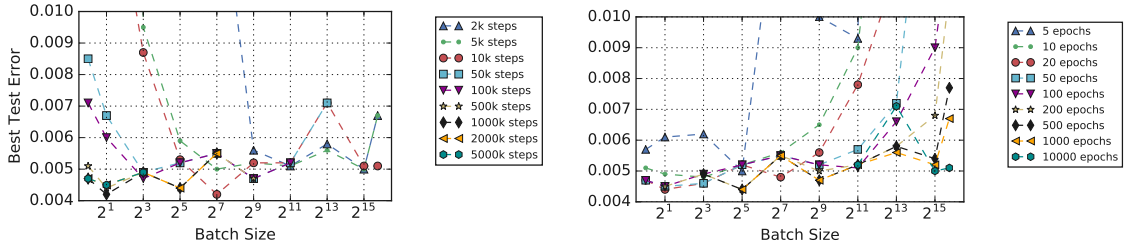
(d) ResNet-50 on ImageNet (e) ResNet-50 on Open Images (f) Transformer on LM1B

(g) Transformer on Common Crawl (h) VGG-11 on ImageNet (i) LSTM on LM1B*

Figure 22: **Optimal momentum has no consistent relationship with batch size.** Momentum corresponds to the trial that reached the goal validation error in the fewest training steps (see Figure 1). See Figure 21 for the corresponding plot of optimal learning rate, and Figure 8 for the corresponding plot of effective learning rate. *For LSTM on LM1B, we only tuned η with fixed $\gamma = 0.99$.

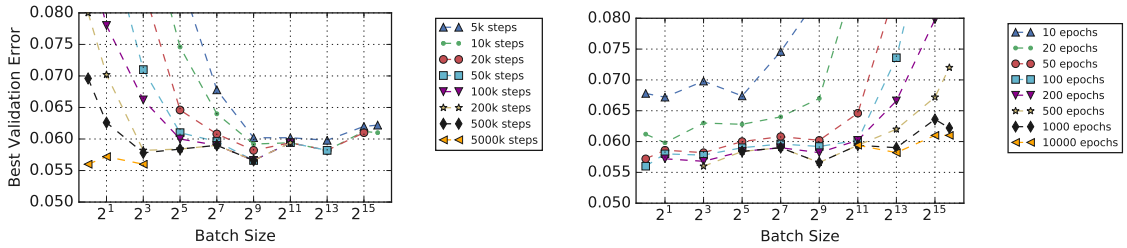


(a) Simple CNN on MNIST: Validation Error

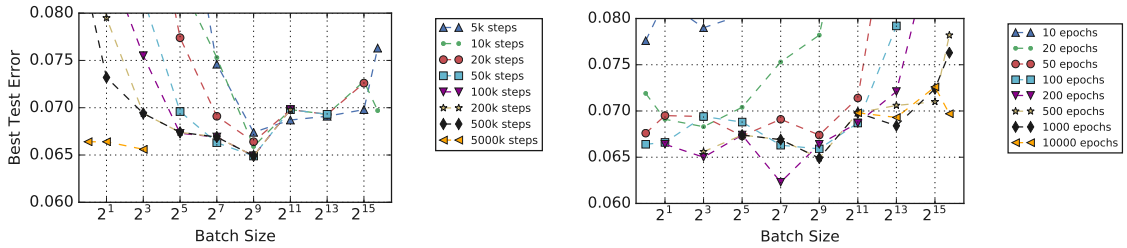


(b) Simple CNN on MNIST: Test Error

Figure 23: **Zoomed version of Figure 11a.**



(a) Simple CNN on Fashion MNIST: Validation Error



(b) Simple CNN on Fashion MNIST: Test Error

Figure 24: **Zoomed version of Figure 11b.**

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: a system for large-scale machine learning. In *Conference on Operating Systems Design and Implementation*, volume 16, pages 265–283. USENIX, 2016.

Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

Rohan Anil, Gabriel Pereyra, Alexandre Passos, Robert Ormandi, George E. Dahl, and Geoffrey E. Hinton. Large scale distributed neural network training through online distillation. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=rkr1UDeC-`.

Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using Kronecker-factored approximations. In *International Conference on Learning Representations*, 2017. URL `https://openreview.net/forum?id=SkkTMpjex`.

Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, pages 161–168, 2008.

Olivier Bousquet, Sylvain Gelly, Karol Kurach, Olivier Teytaud, and Damien Vincent. Critical hyper-parameters: No random, no cry. *arXiv preprint arXiv:1706.03200*, 2017.

Thomas M Breuel. Benchmarking of LSTM networks. *arXiv preprint arXiv:1508.02774*, 2015a.

Thomas M Breuel. The effects of hyperparameters on SGD training of neural networks. *arXiv preprint arXiv:1508.02788*, 2015b.

Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. In *Conference of the International Speech Communication Association*, 2014.

Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. In *International Conference on Learning Representations Workshop Track*, 2016. URL `https://openreview.net/forum?id=D1VDZ5kMAu5jEJ1zfEWL`.

Lingjiao Chen, Hongyi Wang, Jinman Zhao, Dimitris Papailiopoulos, and Paraschos Koutris. The effect of network width on the performance of large-batch training. *arXiv preprint arXiv:1806.03791*, 2018.

Valeriu Codreanu, Damian Podareanu, and Vikram Saletore. Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train. *arXiv preprint arXiv:1711.04291*, 2017.

Aditya Devarakonda, Maxim Naumov, and Michael Garland. AdaBatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. In *International Conference on Machine Learning*, pages 1019–1028, 2017.

Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint arXiv:1811.12941*, 2018.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL `http://www.deeplearningbook.org`.

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. *arXiv preprint arXiv:1706.02677*, 2017.

Roger Grosse and James Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. In *International Conference on Machine Learning*, pages 573–582, 2016.

Elad Hazan. Introduction to online convex optimization. *Foundations and Trends in Optimization*, 2(3-4):157–325, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition*, pages 770–778. IEEE, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016b.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Patwary, Mostofa Ali, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning, lecture 6a: overview of mini-batch gradient descent, 2012. URL `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pages 1731–1741, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

Prateek Jain, Sham M. Kakade, Rahul Kidambi, Praneeth Netrapalli, and Aaron Sidford. Parallelizing stochastic gradient descent for least squares regression: Mini-batching, averaging, and model misspecification. *Journal of Machine Learning Research*, 18(223):1–42, 2018. URL `http://jmlr.org/papers/v18/16-595.html`.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture*, pages 1–12. IEEE, 2017.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

Ryo Karakida, Shotaro Akaho, and Shun-ichi Amari. Universal statistics of Fisher information in deep neural networks: Mean field approach. *arXiv preprint arXiv:1806.01316*, 2018.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *International Conference on Learning Representations*, 2017. URL `https://openreview.net/forum?id=H1oyRlYgg`.

Rahul Kidambi, Praneeth Netrapalli, Prateek Jain, and Sham M. Kakade. On the insufficiency of existing momentum schemes for stochastic optimization. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=rJTutzbA-`.

Jack Kiefer, Jacob Wolfowitz, et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Shahab Kamali, Matteo Malloci, Jordi Pont-Tuset, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. OpenImages: A public dataset for large-scale multi-label and multi-class image classification., 2017. URL `https://storage.googleapis.com/openimages/web/index.html`.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL `http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`.

Guanghui Lan. An optimal method for stochastic composite optimization. *Mathematical Programming*, 133(1-2):365–397, 2012.

Yann Le Cun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524. Springer Verlag, 1998. URL `http://leon.bottou.org/papers/lecun-98x`.

Yann LeCun, Corinna Cortes, and CJ Burges. MNIST handwritten digit database, 1998. URL `http://yann.lecun.com/exdb/mnist`.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436, 2015.

Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *International Conference on Knowledge Discovery and Data Mining*, pages 661–670. ACM, 2014.

Tao Lin, Sebastian U Stich, and Martin Jaggi. Don't use large mini-batches, use local SGD. *arXiv preprint arXiv:1808.07217*, 2018.

Ilya Loshchilov and Frank Hutter. SGDR: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations*, 2017. URL `https://openreview.net/forum?id=Skq89Scxx`.

Siyuan Ma, Raef Bassily, and Mikhail Belkin. The power of interpolation: Understanding the effectiveness of SGD in modern over-parametrized learning. In *International Conference on Machine Learning*, pages 3331–3340, 2018.

James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning*, pages 2408–2417, 2015.

Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$. In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3): 211–252, 2015.

Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From foundations to algorithms*. Cambridge University Press, 2014. URL `https://books.google.com/books?id=OE9etAEACAAJ`.

Ohad Shamir. Without-replacement sampling for stochastic gradient methods. In *Advances in Neural Information Processing Systems*, pages 46–54, 2016.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Samuel L. Smith and Quoc V. Le. A Bayesian perspective on generalization and stochastic gradient descent. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=BJij4yg0Z`.

Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning*, pages 1139–1147, 2013.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception architecture for computer vision. In *Conference on Computer Vision and Pattern Recognition*, pages 2818–2826. IEEE, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

Andreas Veit, Neil Alldrin, Gal Chechik, Ivan Krasin, Abhinav Gupta, and Serge J Belongie. Learning from noisy large-scale datasets with minimal supervision. In *Conference on Computer Vision and Pattern Recognition*, pages 6575–6583. IEEE, 2017.

D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.

Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger Grosse. Understanding short-horizon bias in stochastic meta-optimization. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=H1MczcgR-`.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Dong Yin, Ashwin Pananjady, Max Lam, Dimitris Papailiopoulos, Kannan Ramchandran, and Peter Bartlett. Gradient diversity: a key ingredient for scalable distributed learning. In *International Conference on Artificial Intelligence and Statistics*, 2018. URL `http://proceedings.mlr.press/v84/yin18a.html`.

Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet training in minutes. *arXiv preprint arXiv:1709.05011*, 2017.

UNDERSTANDING SHORT-HORIZON BIAS IN STOCHASTIC META-OPTIMIZATION

Yuhuai Wu*, **Mengye Ren***, **Renjie Liao & Roger B. Grosse**
University of Toronto and Vector Institute
{ywu, mren, rjliao, rgrosse}@cs.toronto.edu

ABSTRACT

Careful tuning of the learning rate, or even schedules thereof, can be crucial to effective neural net training. There has been much recent interest in gradient-based meta-optimization, where one tunes hyperparameters, or even learns an optimizer, in order to minimize the expected loss when the training procedure is unrolled. But because the training procedure must be unrolled thousands of times, the meta-objective must be defined with an orders-of-magnitude shorter time horizon than is typical for neural net training. We show that such short-horizon meta-objectives cause a serious bias towards small step sizes, an effect we term short-horizon bias. We introduce a toy problem, a noisy quadratic cost function, on which we analyze short-horizon bias by deriving and comparing the optimal schedules for short and long time horizons. We then run meta-optimization experiments (both offline and online) on standard benchmark datasets, showing that meta-optimization chooses too small a learning rate by multiple orders of magnitude, even when run with a moderately long time horizon (100 steps) typical of work in the area. We believe short-horizon bias is a fundamental problem that needs to be addressed if meta-optimization is to scale to practical neural net training regimes.

1 INTRODUCTION

The learning rate is one of the most important and frustrating hyperparameters to tune in deep learning. Too small a value causes slow progress, while too large a value causes fluctuations or even divergence. While a fixed learning rate often works well for simpler problems, good performance on the ImageNet (Russakovsky et al., 2015) benchmark requires a carefully tuned schedule. A variety of decay schedules have been proposed for different architectures, including polynomial, exponential, staircase, etc. Learning rate decay is also required to achieve convergence guarantee for stochastic gradient methods under certain conditions (Bottou, 1998). Clever learning rate heuristics have resulted in large improvements in training efficiency (Goyal et al., 2017; Smith, 2017). A related hyperparameter is momentum; typically fixed to a reasonable value such as 0.9, careful tuning can also give significant performance gains (Sutskever et al., 2013). While optimizers such as Adam (Kingma & Ba, 2015) are often described as adapting coordinate-specific learning rates, in fact they also have global learning rate and momentum hyperparameters analogously to SGD, and tuning at least the learning rate can be important to good performance.

In light of this, it is not surprising that there have been many attempts to adapt learning rates, either online during optimization (Schraudolph, 1999; Schaul et al., 2013), or offline by fitting a learning rate schedule (Maclaurin et al., 2015). More ambitiously, others have attempted to learn an optimizer (Andrychowicz et al., 2016; Li & Malik, 2017; Finn et al., 2017; Lv et al., 2017; Wichrowska et al., 2017; Metz et al., 2017). All of these approaches are forms of meta-optimization, where one defines a meta-objective (typically the expected loss after some number of optimization steps) and tunes the hyperparameters to minimize this meta-objective. But because gradient-based meta-optimization can require thousands of updates, each of which unrolls the entire base-level optimization procedure, the meta-optimization is thousands of times more expensive than the base-level optimization. Therefore, the meta-objective must be defined with a much smaller time horizon

*Equal contribution.
Code available at https://github.com/renmengye/meta-optim-public

(e.g. hundreds of updates) than we are ordinarily interested in for large-scale optimization. The hope is that the learned hyperparameters or optimizer will generalize well to much longer time horizons. Unfortunately, we show that this is not achieved in this paper. This is because of a strong tradeoff between short-term and long-term performance, which we refer to as *short-horizon bias*.

In this work, we investigate the short-horizon bias both mathematically and empirically. First, we analyze a quadratic cost function with noisy gradients based on Schaul et al. (2013). We consider this a good proxy for neural net training because second-order optimization algorithms have been shown to train neural networks in orders-of-magnitude fewer iterations (Martens, 2010), suggesting that much of the difficulty of SGD training can be explained by quadratic approximations to the cost. In our noisy quadratic problem, the dynamics of SGD with momentum can be analyzed exactly, allowing us to derive the greedy-optimal (i.e. 1-step horizon) learning rate and momentum in closed form, as well as to (locally) minimize the long-horizon loss using gradient descent. We analyze the differences between the short-horizon and long-horizon schedules.

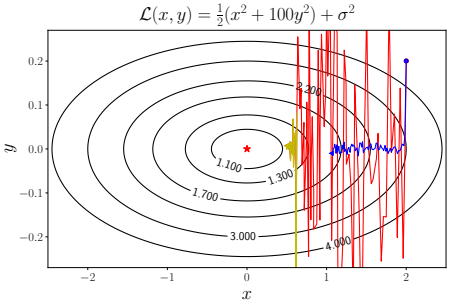


Figure 1: Aggressive learning rate (red) followed by a decay schedule (yellow) wins over conservative learning rate (blue) by making more progress along the low curvature direction (x direction).

Interestingly, when the noisy quadratic problem is *either* deterministic or spherical, greedy schedules are optimal. However, when the problem is *both* stochastic and badly conditioned (as is most neural net training), the greedy schedules decay the learning rate far too quickly, leading to slow convergence towards the optimum. This is because reducing the learning rate dampens the fluctuations along high curvature directions, giving it a large immediate reduction in loss. But this comes at the expense of long-run performance, because the optimizer fails to make progress along low curvature directions. This phenomenon is illustrated in Figure 1, a noisy quadratic problem in 2 dimensions, in which two learning rate schedule are compared: a small fixed learning rate (blue), versus a larger fixed learning rate (red) followed by exponential decay (yellow). The latter schedule initially has higher loss, but it makes more progress towards the optimum, such that it achieves an even smaller loss once the learning rate is decayed.

Figure 2 shows this effect quantitatively for a noisy quadratic problem in 1000 dimensions (defined in Section 2.3). The solid lines show the loss after various numbers of steps of lookahead with a fixed learning rate; if this is used as the meta-objective, it favors small learning rates. The dashed curves show the loss if the same trajectories are followed by 50 steps with an exponentially decayed learning rate; these curves favor higher learning rates, and bear little obvious relationship to the solid ones. This illustrates the difficulty of selecting learning rates based on short-horizon information.

The second part of our paper empirically investigates gradient-based meta-optimization for neural net training. We consider two idealized meta-optimization algorithms: an offline algorithm which fits a learning rate decay schedule by running optimization many times from scratch, and an online algorithm which adapts the learning rate during training. Since our interest is in studying the effect of the meta-objective itself rather than failures of meta-optimization, we give the meta-optimizers sufficient time to optimize their meta-objectives well. We show that short-horizon meta-optimizers, both online and offline, dramatically underperform a hand-tuned fixed learning rate, and sometimes cause the base-level optimization progress to slow to a crawl, even with moderately long time horizons (e.g. 100 or 1000 steps) similar to those used in prior work on gradient-based meta-optimization.

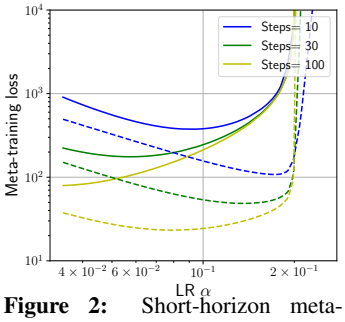


Figure 2: Short-horizon meta-objectives for the noisy quadratic problem. Solid: loss after k updates with fixed learning rate. Dashed: loss after k updates with fixed learning rate, followed by exponential decay.

In short, we expect that any meta-objective which does not correct for short-horizon bias will probably fail when run for a much longer time horizon than it was trained on. There are applications

where short-horizon meta-optimization is directly useful, such as few-shot learning (Santoro et al., 2016; Ravi & Larochelle, 2017). In those settings, short-horizon bias is by definition not an issue. But much of the appeal of meta-optimization comes from the possibility of using it to speed up or simplify the training of large neural networks. In such settings, short-horizon bias is a fundamental obstacle that must be addressed for meta-optimization to be practically useful.

2 NOISY QUADRATIC PROBLEM

In this section, we consider a toy problem which demonstrates the short-horizon bias and can be analyzed analytically. In particular, we borrow the noisy quadratic model of Schaul et al. (2013); the true function being optimized is a quadratic, but in each iteration we observe a noisy version with the correct curvature but a perturbed minimum. This can be equivalently viewed as noisy observations of the gradient, which are intended to capture the stochasticity of a mini-batch-based optimizer. We analyze the dynamics of SGD with momentum on this example, and compare the long-horizon-optimized and greedy-optimal learning rate schedules.

2.1 BACKGROUND

Approximating the cost surface of a neural network with a quadratic function has led to powerful insights and algorithms. Second-order optimization methods such as Newton-Raphson and natural gradient (Amari, 1998) iteratively minimize a quadratic approximation to the cost function. Hessian-free (H-F) optimization (Martens, 2010) is an approximate natural gradient method which tries to minimize a quadratic approximation using conjugate gradient. It can often fit deep neural networks in orders-of-magnitude fewer updates than SGD, suggesting that much of the difficulty of neural net optimization is captured by quadratic models. In the setting of Bayesian neural networks, quadratic approximations to the log-likelihood motivated the Laplace approximation (MacKay, 1992) and variational inference (Graves, 2011; Zhang et al., 2017). Koh & Liang (2017) used quadratic approximations to analyze the sensitivity of a neural network's predictions to particular training labels, thereby yielding insight into adversarial examples.

Such quadratic approximations to the cost function have also provided insights into learning rate and momentum adaptation. In a deterministic setting, under certain conditions, second-order optimization algorithms can be run with a learning rate of 1; for this reason, H-F was able to eliminate the need to tune learning rate or momentum hyperparameters. Martens & Grosse (2015) observed that for a deterministic quadratic cost function, greedily choosing the learning rate and momentum to minimize the error on the next step is equivalent to conjugate gradient (CG). Since CG achieves the minimum possible loss of any gradient-based optimizer on each iteration, the greedily chosen learning rates and momenta are optimal, in the sense that the greedy sequence achieves the minimum possible loss value of *any* sequence of learning rates and momenta. This property fails to hold in the stochastic setting, however, and as we show in this section, the greedy choice of learning rate and momentum can do considerably worse than optimal.

Our primary interest in this work is to adapt scalar learning rate and momentum hyperparameters shared across all dimensions. Some optimizers based on diagonal curvature approximations (Kingma & Ba, 2015) have been motivated in terms of adapting dimension-specific learning rates, but in practice, one still needs to tune scalar learning rate and momentum hyperparameters. Even K-FAC (Martens & Grosse, 2015), which is based on more powerful curvature approximations, has scalar learning rate and momentum hyperparameters. Our analysis applies to all of these methods since they can be viewed as performing SGD in a preconditioned space.

2.2 ANALYSIS

2.2.1 NOTATIONS

We will primarily focus on the SGD with momentum algorithm in this paper. The update is written as follows:

$$\mathbf{v}^{(t+1)} = \mu^{(t)}\mathbf{v}^{(t)} - \alpha^{(t)}\nabla_{\boldsymbol{\theta}^{(t)}}\mathcal{L}, \tag{1}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \mathbf{v}^{(t+1)}, \tag{2}$$

where $\mathcal{L}$ is the loss function, t is the training step, and $\alpha^{(t)}$ is the learning rate. We call the gradient trace $\mathbf{v}^{(t)}$ "velocity", and its decay constant $\mu^{(t)}$ "momentum". We denote the ith coordinate of a vector $\mathbf{v}$ as v_i. When we focus on a single dimension, we sometimes drop the dimension subscripts. We also denote $A(\cdot) = \mathbb{E}[\cdot]^2 + \mathbb{V}[\cdot]$, where $\mathbb{E}$ and $\mathbb{V}$ denote expectation and variance respectively.

2.2.2 PROBLEM FORMULATION

We now define the noisy quadratic model, where in each iteration, the optimizer is given the gradient for a noisy version of a quadratic cost function, where the curvature is correct but the minimum is sampled stochastically from a Gaussian distribution. We assume WLOG that the Hessian is diagonal because SGD is a rotation invariant algorithm, and therefore the dynamics can be analyzed in a coordinate system corresponding to the eigenvectors of the Hessian. We make the further (nontrivial) assumption that the noise covariance is also diagonal.[1] Mathematically, the stochastic cost function is written as:

$$\hat{\mathcal{L}}(\boldsymbol{\theta}) = \frac{1}{2} \sum_i h_i (\theta_i - c_i)^2, \tag{3}$$

where $\mathbf{c}$ is the stochastic minimum, and each c_i follows a Gaussian distribution with mean θ_i^* and variance σ_i^2. The expected loss is given by:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}\left[\hat{\mathcal{L}}(\boldsymbol{\theta})\right] = \frac{1}{2} \sum_i h_i \left((\theta_i - \theta_i^*)^2 + \sigma_i^2 \right). \tag{4}$$

The optimum of $\mathcal{L}$ is given by $\boldsymbol{\theta}^* = \mathbb{E}[\mathbf{c}]$; we assume WLOG that $\boldsymbol{\theta}^* = \mathbf{0}$. The stochastic gradient is given by $\frac{\partial \hat{\mathcal{L}}}{\partial \theta_i} = h_i(\theta_i - c_i)$. Since the deterministic gradient is given by $\frac{\partial \mathcal{L}}{\partial \theta_i} = h_i \theta_i$, the stochastic gradient can be viewed as a noisy Gaussian observation of the deterministic gradient with variance $h_i^2 \sigma_i^2$. This interpretation motivates the use of this noisy quadratic problem as a model of SGD dynamics.

We treat the iterate $\boldsymbol{\theta}^{(t)}$ as a random variable (where the randomness comes from the sampled $\mathbf{c}$'s); the expected loss in each iteration is given by

$$\mathbb{E}\left[\mathcal{L}(\boldsymbol{\theta}^{(t)})\right] = \mathbb{E}\left[\frac{1}{2} \sum_i h_i \left((\theta_i^{(t)})^2 + \sigma_i^2 \right)\right] \tag{5}$$

$$= \frac{1}{2} \sum_i h_i \left(\mathbb{E}\left[\theta_i^{(t)}\right]^2 + \mathbb{V}\left[\theta_i^{(t)}\right] + \sigma_i^2 \right). \tag{6}$$

2.2.3 OPTIMIZED AND GREEDY-OPTIMAL SCHEDULES

We are interested in adapting a global learning rate $\alpha^{(t)}$ and a global momentum decay parameter $\mu^{(t)}$ for each time step t. We first derive a recursive formula for the mean and variance of the iterates at each step, and then analyze the greedy-optimal schedule for $\alpha^{(t)}$ and $\mu^{(t)}$.

Several observations allow us to compactly model the dynamics of SGD with momentum on the noisy quadratic model. First, $\mathbb{E}[\mathcal{L}(\boldsymbol{\theta}^{(t)})]$ can be expressed in terms of $\mathbb{E}[\theta_i]$ and $\mathbb{V}[\theta_i]$ using Eqn. 5. Second, due to the diagonality of the Hessian and the noise covariance matrix, each coordinate evolves independently of the others. Third, the means and variances of the parameters $\theta_i^{(t)}$ and the velocity $v_i^{(t)}$ are functions of those statistics at the previous step.

Because each dimension evolves independently, we now drop the dimension subscripts. Combining these observations, we model the dynamics of SGD with momentum as a *deterministic* recurrence relation with sufficient statistics $\mathbb{E}[\theta^{(t)}]$, $\mathbb{E}[v^{(t)}]$, $\mathbb{V}[\theta^{(t)}]$, $\mathbb{V}[v^{(t)}]$, and $\Sigma_{\theta,v}^{(t)} = \mathrm{Cov}(\theta^{(t)}, v^{(t)})$. The dynamics are as follows:

[1] This amounts to assuming that the Hessian and the noise covariance are codiagonalizable. One heuristic justification for this assumption in the context of neural net optimization is that under certain assumptions, the covariance of the gradients is proportional to the Fisher information matrix, which is close to the Hessian (Martens, 2014).

Theorem 1 (Mean and variance dynamics)**.** *The expectations of the parameter θ and the velocity v are updated as,*

$$\mathbb{E}\left[v^{(t+1)}\right] = \mu^{(t)}\mathbb{E}\left[v^{(t)}\right] - (\alpha^{(t)}h)\mathbb{E}\left[\theta^{(t)}\right],$$
$$\mathbb{E}\left[\theta^{(t+1)}\right] = \mathbb{E}\left[\theta^{(t)}\right] + \mathbb{E}\left[v^{(t+1)}\right].$$

The variances of the parameter θ and the velocity v are updated as

$$\mathbb{V}\left[v^{(t+1)}\right] = \left(\mu^{(t)}\right)^2 \mathbb{V}\left[v^{(t)}\right] + \left(\alpha^{(t)}h\right)^2 \mathbb{V}\left[\theta^{(t)}\right] - 2\mu^{(t)}\alpha^{(t)}h\Sigma^{(t)}_{\theta,v} + \left(\alpha^{(t)}h\sigma\right)^2,$$
$$\mathbb{V}\left[\theta^{(t+1)}\right] = \left(1 - 2\alpha^{(t)}h\right)\mathbb{V}\left[\theta^{(t)}\right] + \mathbb{V}\left[v^{(t+1)}\right] + 2\mu^{(t)}\Sigma^{(t)}_{\theta,v},$$
$$\Sigma^{(t+1)}_{\theta,v} = \mu^{(t)}\Sigma^{(t)}_{\theta,v} - \alpha^{(t)}h\mathbb{V}\left[\theta^{(t)}\right] + \mathbb{V}\left[v^{(t+1)}\right].$$

By applying Theorem 1 recursively, we can obtain $\mathbb{E}[\boldsymbol{\theta}^{(t)}]$ and $\mathbb{V}[\boldsymbol{\theta}^{(t)}]$, and hence $\mathbb{E}[\mathcal{L}(\boldsymbol{\theta}^{(t)})]$, for every t. Therefore, using gradient-based optimization, we can fit a locally optimal learning rate and momentum schedule, i.e. a sequence of values $\{(\alpha^{(t)}, \mu^{(t)})\}_{t=1}^{T}$ which locally minimizes $\mathbb{E}[\mathcal{L}(\boldsymbol{\theta}^{(t)})]$ at some particular time T. We refer to this as the *optimized* schedule.

Furthermore, there is a closed-form solution for one-step lookahead, i.e., we can solve for the optimal learning rate $\alpha^{(t)*}$ and momentum $\mu^{(t)*}$ that minimizes $\mathbb{E}[\mathcal{L}(\boldsymbol{\theta}^{(t+1)})]$ given the statistics at time t. We call this as the *greedy-optimal* schedule.

Theorem 2 (Greedy-optimal learning rate and momentum)**.** *The greedy-optimal learning rate and momentum schedule is given by*

$$\alpha^{(t)*} = \frac{\sum_i h_i^2 A\left(\theta_i^{(t)}\right)\left[\sum_j h_j A\left(v_j^{(t)}\right)\right] - \left(\sum_j h_j \mathbb{E}\left[\theta_j^{(t)} v_j^{(t)}\right]\right) h_i^2\, \mathbb{E}\left[\theta_i^{(t)} v_i^{(t)}\right]}{\sum_i h_i^3\left[A\left(\theta_i^{(t)}\right) + \sigma_i^2\right]\left[\sum_j h_j A\left(v_j^{(t)}\right)\right] - \left(\sum_j h_j^2\, \mathbb{E}\left[\theta_j^{(t)} v_j^{(t)}\right]\right) h_i^2\, \mathbb{E}\left[\theta_i^{(t)} v_i^{(t)}\right]},$$
$$\mu^{(t)*} = -\frac{\sum_i h_i\left(1 - \alpha^{(t)*} h_i\right)\mathbb{E}\left[\theta_i^{(t)} v_i^{(t)}\right]}{\sum_i h_i A\left(v_i^{(t)}\right)}.$$

Note that Schaul et al. (2013) derived the greedy optimal learning rate for SGD, and Theorem 2 extends it to the greedy optimal learning rate and momentum for SGD with momentum.

2.2.4 Univariate and Spherical Cases

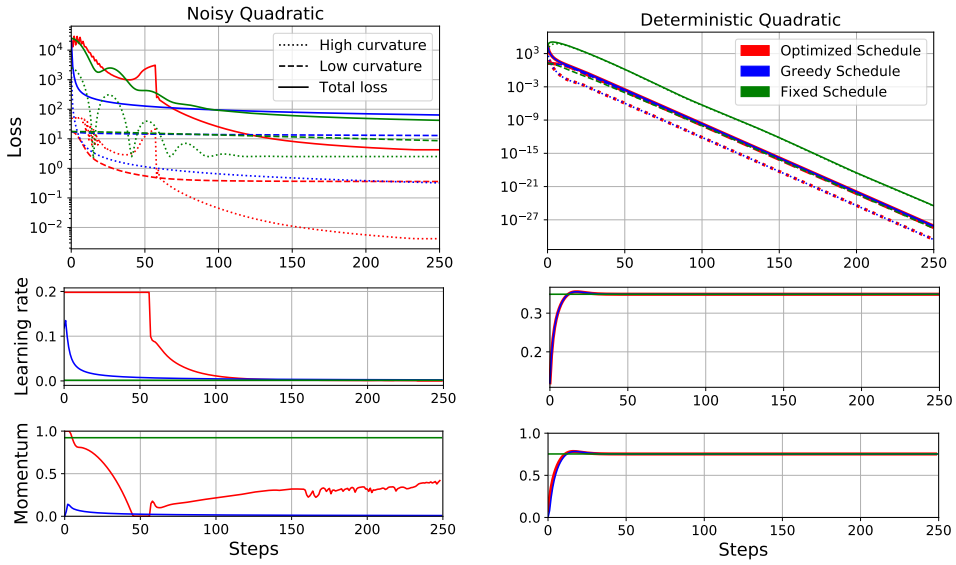
As noted in Section 2.1, Martens & Grosse (2015) found the greedy choice of α and μ to be optimal for gradient descent on deterministic quadratic objectives. We now show that the greedy schedule is also optimal for SGD *without* momentum in the case of univariate noisy quadratics, and hence also for multivariate ones with spherical Hessians and gradient covariances. In particular, the following holds for SGD without momentum on a univariate noisy quadratic:

Theorem 3 (Optimal learning rate, univariate)**.** *For all $T \in \mathbb{N}$, the sequence of learning rates $\{\alpha^{(t)*}\}_{t=1}^{T-1}$ that minimizes $\mathcal{L}(\theta^{(T)})$ is given by*

$$\alpha^{(t)*} = \frac{A\left(\theta^{(t)}\right)}{h\left(A\left(\theta^{(t)}\right) + \sigma^2\right)}. \tag{7}$$

Moreover, this agrees with the greedy-optimal learning rate schedule as derived by Schaul et al. (2013).

If the Hessian and the gradient covariance are both spherical, then each dimension evolves identically and independently according to the univariate dynamics. Of course, one is unlikely to encounter an optimization problem where both are exactly spherical. But some approximate second-order optimizers, such as K-FAC, can be viewed as preconditioned SGD, i.e. SGD in a transformed



(a) (b)

Figure 3: Comparisons of the optimized learning rates and momenta trained by gradient descent (red), greedy learning rates and momenta (blue), and the optimized fixed learning rate and momentum (green) in both noisy (a) and deterministic (b) quadratic settings. In the deterministic case, our optimized schedule matched the greedy one, just as the theory predicts.

space where the Hessian and the gradient covariance are better conditioned (Martens & Grosse, 2015). In principle, with a good enough preconditioner, the Hessian and the gradient covariance would be close enough to spherical that a greedy choice of α and μ would perform well. It will be interesting to investigate whether any practical optimization algorithms demonstrate this behavior.

2.3 Experiments

In this section, we compare the optimized and greedy-optimal schedules on a noisy quadratic problem. We chose a 1000 dimensional quadratic cost function with the curvature distribution from Li (2005), on which CG achieves its worst-case convergence rate. We assume that $h_i = \mathbb{V}[\frac{\partial \mathcal{L}}{\partial \theta_i}]$, and hence $\sigma_i^2 = \frac{1}{h_i}$; this choice is motivated by the observations that under certain assumptions, the Fisher information matrix is a good approximation to the Hessian matrix, but also reflects the covariance structure of the gradient noise (Martens, 2014). We computed the greedy-optimal schedules using Theorem 3. For the optimized schedules, we minimized the expected loss at time $T = 250$ using Adam using Adam (Kingma & Ba, 2015), with a learning rate 0.003 and 500 steps. We set an upper bound for the learning rate which prevented the loss component for any dimension from becoming larger than its initial value; this was needed because otherwise the optimized schedule allowed the loss to temporarily grow very large, a pathological solution which would be unstable on realistic problems. We also considered fixed learning rate and momentum, with the two hyperparameters fit using Adam. The training curves and the corresponding learning rates and momenta are shown in Figure 3(a). The optimized schedule achieved a much lower final expected loss value (4.25) than was obtained by the greedy-optimal schedule (63.86) or fixed schedule (42.19).

We also show the sums of the losses along the 50 highest curvature directions and 50 lowest curvature directions. We find that under the optimized schedule, the losses along the high curvature directions hardly decrease initially. However, because it maintains a high learning rate, the losses along the low curvature directions decrease significantly. After 50 iterations, it begins decaying the learning rate, at which point it achieves a large drop in both the high-curvature and total losses. On the other hand, under the greedy-optimal schedule, the learning rates and momenta become small very early on, which immediately reduces the losses on the high curvature directions, and hence also the total loss. However, in the long term, since the learning rates are too small to make substantial progress along the low curvature directions, the total loss converged to a much higher value in the

end. This gives valuable insight into the nature of the short-horizon bias in meta-optimization: short-horizon objectives will often encourage the learning rate and momentum to decay quickly, so as to achieve the largest gain in the short term, but at the expense of long-run performance.

It is interesting to compare this behavior with the deterministic case. We repeated the above experiment for a *deterministic* quadratic cost function (i.e. $\sigma_i^2 = 0$) with the same Hessian; results are shown in Figure 3(b). The greedy schedule matches the optimized one, as predicted by the analysis of Martens & Grosse (2015). This result illustrates that stochasticity is necessary for short-horizon bias to manifest. Interestingly, the learning rate and momentum schedules in the deterministic case are nearly flat, while the optimized schedules for the stochastic case are much more complex, suggesting that stochastic optimization raises a different set of issues for hyperparameter adaptation.

3 GRADIENT-BASED META-OPTIMIZATION

We now turn our attention to gradient-based hyperparameter optimization. A variety of approaches have been proposed which tune hyperparameters by doing gradient descent on a meta-objective (Schraudolph, 1999; Maclaurin et al., 2015; Andrychowicz et al., 2016). We empirically analyze an idealized version of a gradient-based meta-optimization algorithm called stochastic meta-descent (SMD) (Schraudolph, 1999). Our version of SMD is idealized in two ways: first, we drop the algorithmic tricks used in prior work, and instead allow the meta-optimizer more memory and computation than would be economical in practice. Second, we limit the representational power of our meta-model: whereas Andrychowicz et al. (2016) aimed to learn a full optimization algorithm, we focus on the much simpler problem of adapting learning rate and momentum hyperparameters, or schedules thereof. The aim of these two simplifications is that we would like to do a good enough job of optimizing the meta-objective that any base-level optimization failures can be attributed to deficiencies in the meta-objective itself (such as short-horizon bias) rather than incomplete meta-optimization.

Despite these simplifications, we believe our experiments are relevant to practical meta-optimization algorithms which optimize the meta-objective less thoroughly. Since the goal of the meta-optimizer is to adapt two hyperparameters, it's possible that poor meta-optimization could cause the hyperparameters to get stuck in regions that happen to perform well; indeed, we observed this phenomenon in some of our early explorations. But it would be dangerous to rely on poor meta-optimization, since improved meta-optimization methods would then lead to worse base-level performance, and tuning the meta-optimizer could become a roundabout way of tuning learning rates and momenta.

We also believe our experiments are relevant to meta-optimization methods which aim to learn entire algorithms. Even if the learned algorithms don't have explicit learning rate parameters, it's possible for a learning rate schedule to be encoded into an algorithm itself; for instance, Adagrad (Duchi et al., 2011) implicitly uses a polynomial decay schedule because it sums rather than averages the squared derivatives in the denominator. Hence, one would need to worry about whether the meta-optimizer is implicitly fitting a learning rate schedule that's optimized for short-term performance.

3.1 BACKGROUND: STOCHASTIC META-DESCENT

The high-level idea of stochastic meta-descent (SMD) (Schraudolph, 1999) is to perform gradient descent on the learning rate, or any other differentiable hyperparameters. This is feasible since any gradient based optimization algorithm can be unrolled as a computation graph (see Figure 4), and automatic differentiation is readily available in most deep learning libraries.

There are two basic types of automatic differentiation (autodiff) methods: *forward mode* and *reverse mode*. In forward mode autodiff, directional derivatives are computed alongside the forward computation. In contrast, reverse mode autodiff (a.k.a. backpropagation) computes the gradients moving backwards through the computation graph. Meta-optimization using reverse mode can be computationally demanding due to memory constraints, since the parameters need to be stored at every step. Maclaurin et al. (2015) got around this by cleverly exploiting approximate reversibility to minimize the memory cost of activations. Since we are optimizing only two hyperparameters, however, forward mode autodiff can be done cheaply. Here, we provide the forward differentiation

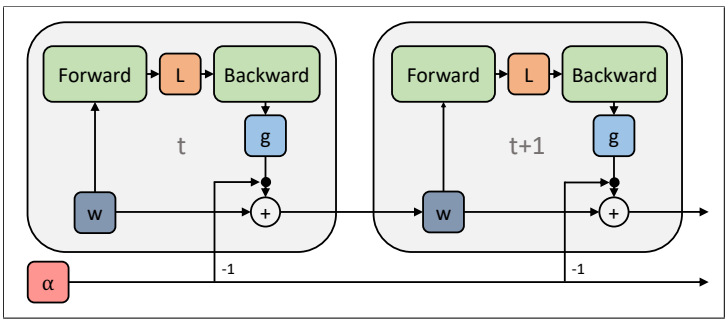


Figure 4: Regular SGD in the form of a computation graph. The learning rate parameter α is part of the differentiable computations.

equations for obtaining the gradient of vanilla SGD learning rate. Let $\frac{d\boldsymbol{\theta}_t}{d\alpha}$ be $\boldsymbol{u}_t$, and $\frac{d\mathcal{L}_t}{d\alpha}$ be α', and the Hessian at step t to be H_t. By chain rule, we get,

$$\alpha' = \boldsymbol{g}_t \cdot \boldsymbol{u}_{t-1}, \tag{8}$$

$$\boldsymbol{u}_t = \boldsymbol{u}_{t-1} - \boldsymbol{g}_t - \alpha H_t \boldsymbol{u}_{t-1}. \tag{9}$$

While the Hessian is infeasible to construct explicitly, the Hessian-vector product in Equation 9 can be computed efficiently using reverse-on-reverse (Werbos, 1988) or forward-on-reverse automatic differentiation (Pearlmutter, 1994), in time linear in the cost of the forward pass. See Schraudolph (2002) for more details.

Using the gradients with respect to hyperparameters, as given in Eq. 9, we can apply gradient based meta-optimization, just like optimizing regular parameters. It is worth noting that, although SMD was originally proposed for optimizing vanilla SGD, in practice it can be applied to other optimization algorithms such as SGD with momentum or Adam (Kingma & Ba, 2015). Moreover, gradient-based optimizers other than SGD can be used for the meta-optimization as well.

Algorithm 1: Stochastic Meta-Descent

Input: $\alpha_0, \eta, \boldsymbol{\theta}, T, M$
Output: α
$\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}$;
$\alpha \leftarrow \alpha_0$;
for $m \leftarrow 1 \ldots M$ **do**
  $\boldsymbol{u} \leftarrow \mathbf{0}$;
  for $t \leftarrow 1 \ldots T$ **do**
    $X, \boldsymbol{y} \leftarrow$ `GetMiniBatch()`;
    $\boldsymbol{g} \leftarrow$ `BGrad`$(L(X, \boldsymbol{y}, \boldsymbol{\theta}), \boldsymbol{\theta}, 1)$;
    $\boldsymbol{\theta}_{new} \leftarrow$ `Step`$(\boldsymbol{\theta}, \boldsymbol{g}, \alpha)$;
    $\alpha' \leftarrow \boldsymbol{g} \cdot \boldsymbol{u}$;
    $\boldsymbol{u} \leftarrow$ `FGrad`$(\boldsymbol{\theta}_{new}, [\alpha, \boldsymbol{\theta}], [1, \boldsymbol{u}])$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{new}$;
  $\alpha \leftarrow$ `MetaStep`(α, α', η);
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0$
return α

The basic SMD algorithm is given as Algorithm 1. Here, α is a set of hyperparameters (e.g. learning rate), and α_0 are inital hyperparameter values; $\boldsymbol{\theta}$ is a set of optimization intermediate variables, such as weights and velocities; η is a set of meta-optimizer hyperparameters (e.g. meta learning rate). $\texttt{BGrad}(y, x, dy)$ is the backward gradient function that computes the gradients of the loss function wrt. $\boldsymbol{\theta}$, and $\texttt{FGrad}(y, x, dx)$ is the forward gradient function that accumulates the gradients of $\boldsymbol{\theta}$ with respect to α. `Step` and `MetaStep` optimize regular parameters and hyperparameters, respectively, for one step using gradient-based methods. Additionally, T is the lookahead window size, and M is the number of meta updates.

Simplifications from the original SMD algorithm. The original SMD algorithm (Schraudolph, 1999) fit coordinate-wise adaptive learning rates with intermediate gradients ($\boldsymbol{u}_t$) accumulated throughout the process of training. Since computing separate directional derivatives for each coordinate using forward mode autodiff is computationally prohibitive, the algorithm used approximate updates. Both features introduced bias into the meta-gradients. We make several changes to the original algorithm. First, we tune only a global learning rate parameter. Second, we use exact forward mode accumulation because this is feasible for a single learning rate. Third, rather than accumulate directional derivatives during training, we compute the meta-updates on separate SGD

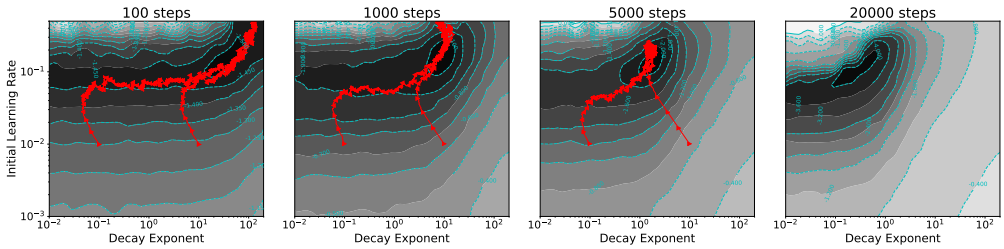


Figure 5: Meta-objective surfaces and SMD trajectories (red) optimizing initial effective learning rate and decay exponent with horizons of {100, 1k, 5k, 20k} steps[2]. 2.5k random samples with Gaussian interpolation are used to illustrate the meta-objective surface.

trajectories simulated using fixed network parameters. Finally, we compute multiple meta-updates in order to ensure that the meta-objective is optimized sufficiently well. Together, these changes ensure unbiased meta-gradients, as well as careful optimization of the meta-objective, at the cost of high computational overhead. We do not recommend this approach as a practical SMD implementation, but rather as a way of understanding the biases in the meta-objective itself.

3.2 Offline meta-optimization

To understand the sensitivity of the optimized hyperparameters to the horizon, we first carried out an offline experiment on a multi-layered perceptron (MLP) on MNIST (LeCun et al., 1998). Specifically, we fit learning rate decay schedules offline by repeatedly training the network, and a single meta-gradient was obtained from each training run.

Learnable decay schedule. We used a parametric learning rate decay schedule known as *inverse time decay* (Welling & Teh, 2011): $\alpha_t = \frac{\alpha_0}{(1+\frac{t}{K})^\beta}$, where α_0 is the initial learning rate, t is the number of training steps, β is the learning rate decay exponent, and K is the time constant. We jointly optimized α_0 and β. We fixed $\mu = 0.9$, $K = 5000$ for simplicity.

Experimental details. The network had two layers of 100 hidden units, with ReLU activations. Weights were initialized with a zero-mean Gaussian with standard deviation 0.1. We used a warm start from a network trained for 50 SGD with momentum steps, using $\alpha = 0.1, \mu = 0.9$. (We used a warm start because the dynamics are generally different at the very start of training.) For SMD optimization, we trained all hyperparameters in log space using Adam optimizer, with 5k meta steps.

Figure 5 shows SMD optimization trajectories on the meta-objective surfaces, initialized with multiple random hyperparameter settings. The SMD trajectories appear to have converged to the global optimum.

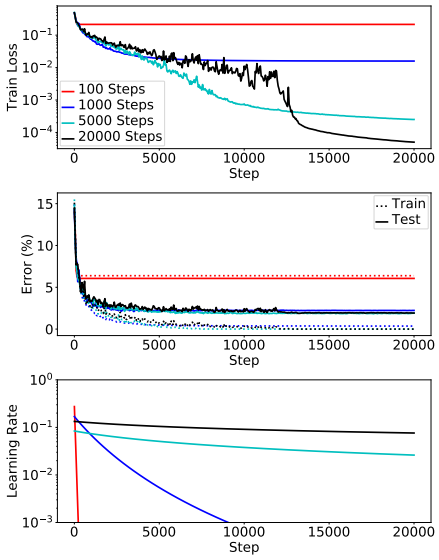


Figure 6: Training curves with best learning rate schedules from meta-objective surfaces with {100, 1k, 5k, 20k} step horizons.

Importantly, the meta-objectives with longer horizons favored a much smaller learning rate decay exponent β, leading to a more gradual decay schedule. The meta-objective surfaces were very different depending on the time horizon, and the final β value differed by over two orders of magnitude between 100 and 20k step horizons.

We picked the best learning rate schedules from meta-objective surfaces (in Figure 5), and obtained the training curves of a network shown in Figure 6. The resulting training loss at 20k steps with

[2]We encountered some optimization difficulties for SMD with horizon of 20k steps. Since those are not the focus of this paper, we left out the trajectories of 20k steps to avoid confusions.

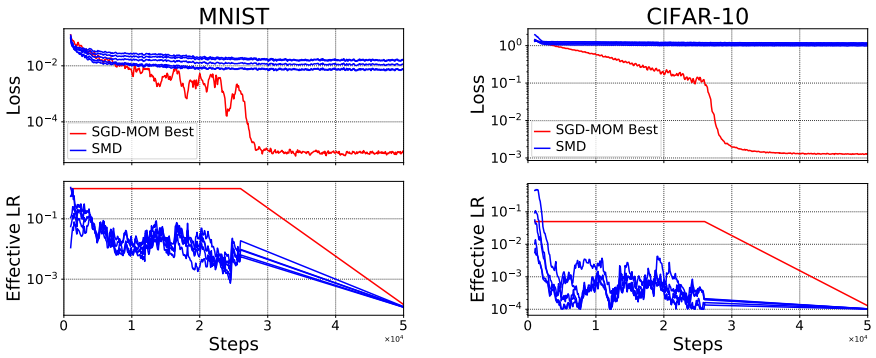


Figure 7: Training curves and learning rates from online SMD with lookahead of 5 steps (blue), and hand-tuned fixed learning rate (red). Each blue curve corresponds to a different initial learning rate.

the 100 step horizon was over *three* orders of magnitude larger than with the 20k step horizon. In general, short horizons gave better performance initially, but were surpassed by longer horizons. The differences in *error* were less drastic, but we see that the 100 step network was severely undertrained, and the 1k step network achieved noticeably worse test error than the longer-horizon ones.

3.3 Online Meta-Optimization

In this section, we study whether online adaptation also suffers from short-horizon bias. Specifically, we used Algorithm 1) to adapt the learning rate and momentum hyperparameters online while a network is trained. We experimented with an MLP on MNIST and a CNN on CIFAR-10 (Krizhevsky, 2009).

Experimental details. For the MNIST experiments, we used an MLP network with two hidden layers of 100 units, with ReLU activations. Weights were initialized with a zero-mean Gaussian with standard deviation 0.1. For CIFAR-10 experiments, we used a CNN network adapted from Caffe (Jia et al., 2014), with 3 convolutional layers of filter size 3×3 and depth [32, 32, 64], and 2×2 max pooling with stride 2 after every convolution layer, and follwed by a fully connected hidden layer of 100 units. Meta-optimization was done with 100 steps of Adam for every 10 steps of regular training. We adapted the learning rate α and momentum μ. After 25k steps, adaptation was stopped, and we trained for another 25k steps with an exponentially decaying learning rate such that it reached 1e-4 on the last time step. We re-parameterized the learning rate with the effective learning rate $\alpha_{\text{eff}} = \frac{\alpha}{1-\mu}$, and the momentum with $1 - \mu$, so that they can be optimized more smoothly in the log space.

Figure 7 shows training curves both with online SMD and with hand-tuned fixed learning rate and momentum hyperparameters. We show several SMD runs initialized from widely varying hyperparameters; all the SMD runs behaved similarly, suggesting it optimized the meta-objective efficiently enough. Under SMD, learning rates were quickly decreased to very small values, leading to slow progress in the long term, consistent with the noisy quadratic and offline adaptation experiments.

As online SMD can be too conservative in the choice of learning rate, it is natural to ask whether removing the stochasticity in the lookahead sequence can fix the problem. We therefore considered online SMD where the entire lookahead trajectory used a *single* mini-batch, hence removing the stochasticity. As shown in Figure 8, this deterministic lookahead scheme led to the opposite problem: the adapted learning rates were very large, leading to instability. We conclude that the stochasticity of mini-batch training cannot be simply ignored in meta-optimization.

4 Conclusion

In this paper, we analyzed the problem of short-horizon bias in meta-optimization. We presented a noisy quadratic toy problem which we analyzed mathematically, and observed that the optimal

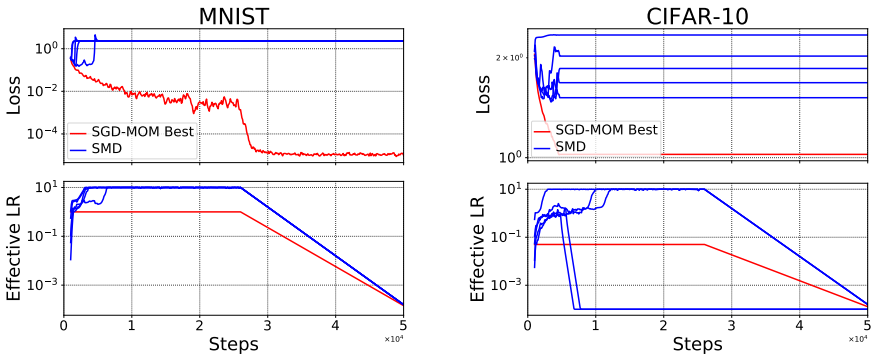


Figure 8: Online SMD with deterministic lookahead of 5 steps (blue), compared with a manually tuned fixed learning rate (red). Other settings are the same as Figure 7.

learning rate schedule differs greatly from a greedy schedule that minimizes training loss one step ahead. While the greedy schedule tends to decay the learning rate drastically to reduce the loss on high curvature directions, the optimal schedule keeps a high learning rate in order to make steady progress on low curvature directions, and eventually achieves far lower loss. We showed that this bias stems from the combination of stochasticity and ill-conditioning: when the problem is *either* deterministic or spherical, the greedy learning rate schedule is globally optimal; however, when the problem is both stochastic and ill-conditioned (as is most neural net training), the greedy schedule performs poorly. We empirially verified the short-horizon bias in the context of neural net training by applying gradient based meta-optimization, both offline and online. We found the same pathological behaviors as in the noisy quadratic problem — a fast learning rate decay and poor long-run performance.

While our results suggest that meta-optimization should not be applied blindly, our noisy quadratic analysis also provides grounds for optimism: by removing ill-conditioning (by using a good preconditioner) and/or stochasticity (with large batch sizes or variance reduction techniques), it may be possible to enter the regime where short-horizon meta-optimization works well. It remains to be seen whether this is achievable with existing optimization algorithms.

Acknowledgement YW is supported by a Google PhD Fellowship. RL is supported by Connaught International Scholarships.

REFERENCES

Shun Ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998. ISSN 0899-7667.

Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 3981–3989, 2016.

Léon Bottou. On-line learning in neural networks. chapter On-line Learning and Stochastic Approximations, pp. 9–42. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-65263-4.

John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, 2017.

Priya Goyal, Piotr Dollar, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. Technical report, FAIR, 2017.

Alex Graves. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pp. 2348–2356, 2011.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pp. 675–678, 2014. ISBN 978-1-4503-3063-3.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3th International Conference on Learning Representations*, 2015.

P. W. Koh and P. Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning (ICML)*, 2017.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

Ke Li and Jitendra Malik. Learning to optimize. In *5th International Conference on Learning Representations*, 2017.

Ren-cang Li. Sharpness in rates of convergence for CG and symmetric lanczos methods. Technical report, 2005.

Kaifeng Lv, Shunhua Jiang, and Jian Li. Learning gradient descent: Better generalization and longer horizons. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, pp. 2247–2255, 2017.

David J. C. MacKay. A practical bayesian framework for backpropagation networks. *Neural Comput.*, 4(3):448–472, May 1992. ISSN 0899-7667.

Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32nd International Conference on Machine Learning*, July 2015.

James Martens. Deep learning via Hessian-free optimization. In *ICML-10*, 2010.

James Martens. New insights and perspectives on the natural gradient method. arXiv:1412.1193, 2014.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *ICML*, 2015.

Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks. In *5th International Conference on Learning Representations*, 2017.

Barak A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6(1):147–160, January 1994. ISSN 0899-7667. doi: 10.1162/neco.1994.6.1.147.

Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In *5th International Conference on Learning Representations*, 2017.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy P. Lillicrap. Meta-learning with memory-augmented neural networks. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pp. 1842–1850, 2016.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pp. 343–351, 2013.

Nicol N. Schraudolph. Local gain adaptation in stochastic gradient descent. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pp. 569–574 vol.2, 1999. doi: 10.1049/cp:19991170.

Nicol N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7):1723–1738, July 2002. ISSN 0899-7667. doi: 10.1162/08997660260028683.

L. N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464–472, March 2017. doi: 10.1109/WACV.2017.58.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, 2013.

Max Welling and Yee Whye Teh. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pp. 681–688, 2011.

P. J. Werbos. Backpropagation: past and future. *IEEE 1988 International Conference on Neural Networks*, pp. 343–353 vol.1, 1988.

Olga Wichrowska, Niru Maheswaranathan, Matthew W. Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando de Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, pp. 3751–3760, 2017.

Guodong Zhang, Shengyang Sun, David K. Duvenaud, and Roger B. Grosse. Noisy natural gradient as variational inference. *CoRR*, abs/1712.02390, 2017.

A Proofs of Theorems

The proofs are organized as follows; we provide a proof to Theorem 1 in A.1, a proof to Theorem 2 in A.2 and a proof to Theorem 3 in A.3.

A.1 Model Dynamics

Recall the stochastic gradient descent with momentum is defined as follows,

$$\begin{aligned} v^{(t+1)} &= \mu^{(t)} v^{(t)} - \alpha^{(t)}(h\theta^{(t)} + h\sigma\xi), \quad \xi \sim \mathcal{N}(0,1) \\ \theta^{(t+1)} &= \theta^{(t)} + v^{(t+1)} = \theta^{(t)} + \mu^{(t)} v^{(t)} - \alpha^{(t)}(h\theta^{(t)} + h\sigma\xi) \\ &= (1 - \alpha^{(t)} h)\theta^{(t)} + \mu^{(t)} v^{(t)} + h\sigma\xi. \end{aligned}$$

A.1.1 Dynamics of the expectation

We calculate the mean of the velocity $v^{(t+1)}$,

$$\begin{aligned} \mathbb{E}\left[v^{(t+1)}\right] &= \mathbb{E}\left[\mu^{(t)} v^{(t)} - \alpha^{(t)} h\theta^{(t)}\right] \\ &= \mu^{(t)} \mathbb{E}\left[v^{(t)}\right] - \alpha^{(t)} h \mathbb{E}\left[\theta^{(t)}\right]. \end{aligned} \tag{10}$$

We calculate the mean of the parameter $\theta^{(t+1)}$,

$$\mathbb{E}\left[\theta^{(t+1)}\right] = \mathbb{E}\left[\theta^{(t)}\right] + \mathbb{E}\left[v^{(t+1)}\right]. \tag{11}$$

Let's assume the following initial conditions:

$$\mathbb{E}\left[v^{(0)}\right] = 0$$
$$\mathbb{E}\left[\theta^{(0)}\right] = E_0.$$

Then Eq.(10) and Eq.(11) describes how $\mathbb{E}\left[\theta^{(t)}\right], \mathbb{E}\left[v^{(t)}\right]$ changes over time t.

A.1.2 Dynamics of the variance

We calculate the variance of the velocity $v^{(t+1)}$,

$$\begin{aligned}\mathbb{V}\left[v^{(t+1)}\right] &= \mathbb{V}\left[\mu^{(t)}v^{(t)} - \alpha^{(t)}h\theta^{(t)}\right] + (\alpha^{(t)}h\sigma)^2 \\ &= (\mu^{(t)})^2\mathbb{V}\left[v^{(t)}\right] + (\alpha^{(t)}h)^2\mathbb{V}\left[\theta^{(t)}\right] - 2\mu^{(t)}\alpha^{(t)}h \cdot \mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right) + (\alpha^{(t)}h\sigma)^2.\end{aligned} \tag{12}$$

The variance of the parameter $\theta^{(t+1)}$ is given by,

$$\mathbb{V}\left[\theta^{(t+1)}\right] = \mathbb{V}\left[\theta^{(t)}\right] + \mathbb{V}\left[v^{(t+1)}\right] + 2\left(\mu^{(t)}\mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right) - \alpha^{(t)}h\mathbb{V}\left[\theta^{(t)}\right]\right). \tag{13}$$

We also need to derive how the covariance of θ and v changes over time:

$$\begin{aligned}\mathrm{Cov}\left(\theta^{(t+1)}, v^{(t+1)}\right) &= \mathrm{Cov}\left((\theta^{(t)} + v^{(t+1)}), v^{(t+1)}\right) \\ &= \mathrm{Cov}\left(\theta^{(t)}, v^{(t+1)}\right) + \mathbb{V}\left[v^{(t+1)}\right] \\ &= \mu^{(t)}\mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right) - \alpha^{(t)}h\mathbb{V}\left[\theta^{(t)}\right] + \mathbb{V}\left[v^{(t+1)}\right].\end{aligned} \tag{14}$$

Let's assume the following initial conditions:

$$\mathbb{V}\left[v^{(0)}\right] = 0$$
$$\mathbb{V}\left[\theta^{(0)}\right] = V_0$$
$$\mathrm{Cov}\left(\theta^{(0)}, v^{(0)}\right) = 0.$$

Combining Eq.(12-14), we obtain the following dynamics (from $t = 0, \ldots, T-1$):

$$\begin{aligned}\mathbb{V}\left[v^{(t+1)}\right] &= (\mu^{(t)})^2\mathbb{V}\left[v^{(t)}\right] + (\alpha^{(t)}h)^2\mathbb{V}\left[\theta^{(t)}\right] - 2\mu^{(t)}\alpha^{(t)}h \cdot \mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right) + (\alpha^{(t)}h\sigma)^2 \\ \mathbb{V}\left[\theta^{(t+1)}\right] &= \mathbb{V}\left[\theta^{(t)}\right] + \mathbb{V}\left[v^{(t+1)}\right] + 2\left(\mu^{(t)}\mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right) - \alpha^{(t)}h\mathbb{V}\left[\theta^{(t)}\right]\right) \\ \mathrm{Cov}\left(\theta^{(t+1)}, v^{(t+1)}\right) &= \mu^{(t)}\mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right) - \alpha^{(t)}h\mathbb{V}\left[\theta^{(t)}\right] + \mathbb{V}\left[v^{(t+1)}\right].\end{aligned}$$

A.2 GREEDY OPTIMALITY

A.2.1 UNIVARIATE CASE

The loss at time step t is,

$$\begin{aligned}
\mathcal{L}^{(t+1)} &= \frac{1}{2}h\left(\mathbb{E}\left[\theta^{(t+1)}\right]^2 + \mathbb{V}\left[\theta^{(t+1)}\right]\right) \\
&= \frac{1}{2}h\Big[\left(\mathbb{E}\left[\theta^{(t)}\right] + \mu^{(t)}\mathbb{E}\left[v^{(t)}\right] - (\alpha^{(t)}h)\mathbb{E}\left[\theta^{(t)}\right]\right)^2 + \mathbb{V}\left[\theta^{(t)}\right] + (\mu^{(t)})^2\mathbb{V}\left[v^{(t)}\right] + (\alpha^{(t)}h)^2\mathbb{V}\left[\theta^{(t)}\right] \\
&\quad - 2\mu^{(t)}\alpha^{(t)}h\cdot\mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right) + (\alpha^{(t)}h\sigma)^2 + 2\left(\mu^{(t)}\mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right) - \alpha^{(t)}h\mathbb{V}\left[\theta^{(t)}\right]\right)\Big] \\
&= \frac{1}{2}h\Big[\left((1-\alpha^{(t)}h)\mathbb{E}\left[\theta^{(t)}\right] + \mu^{(t)}\mathbb{E}\left[v^{(t)}\right]\right)^2 + (1-\alpha^{(t)}h)^2\mathbb{V}\left[\theta^{(t)}\right] + (\mu^{(t)})^2\mathbb{V}\left[v^{(t)}\right] \\
&\quad + 2\mu^{(t)}(1-\alpha^{(t)}h)\mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right) + (\alpha^{(t)}h\sigma)^2\Big] \\
&= \frac{1}{2}h\Big[(1-\alpha^{(t)}h)^2\left(\mathbb{E}\left[\theta^{(t)}\right]^2 + \mathbb{V}\left[\theta^{(t)}\right]\right) + (\mu^{(t)})^2\left(\mathbb{E}\left[v^{(t)}\right]^2 + \mathbb{V}\left[v^{(t)}\right]\right) \\
&\quad + 2\mu^{(t)}(1-\alpha^{(t)}h)\left(\mathbb{E}\left[\theta^{(t)}\right]\mathbb{E}\left[v^{(t)}\right] + \mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right)\right) + (\alpha^{(t)}h\sigma)^2\Big].
\end{aligned}$$

For simplicity, we denote $A(\cdot) = \mathbb{E}\left[\cdot\right]^2 + \mathbb{V}\left[\cdot\right]$, and notice that $\mathbb{E}\left[\theta^{(t)}v^{(t)}\right] = \mathbb{E}\left[\theta^{(t)}\right]\mathbb{E}\left[v^{(t)}\right] + \mathrm{Cov}\left(\theta^{(t)}, v^{(t)}\right)$, hence,

$$\mathcal{L}^{(t+1)} = \frac{1}{2}h\Big[(1-\alpha^{(t)}h)^2A(\theta^{(t)}) + (\mu^{(t)})^2A(v^{(t)}) + 2\mu^{(t)}(1-\alpha^{(t)}h)\mathbb{E}\left[\theta^{(t)}v^{(t)}\right] + (\alpha^{(t)}h\sigma)^2\Big]. \tag{15}$$

In order to find the optimal learning rate and momentum for minimizing $\mathcal{L}^{(t+1)}$, we take the derivative with respect to $\alpha^{(t)}$ and $\mu^{(t)}$, and set it to 0:

$$\begin{aligned}
\nabla_{\alpha^{(t)}}\mathcal{L}^{(t+1)} &= (1-\alpha^{(t)}h)A(\theta^{(t)})\cdot(-h) - \mu^{(t)}h\mathbb{E}\left[\theta^{(t)}v^{(t)}\right] + \alpha^{(t)}(h\sigma)^2 = 0 \\
\alpha^{(t)}h(A(\theta^{(t)}) + \sigma^2) &= A(\theta^{(t)}) + \mu^{(t)}\mathbb{E}\left[\theta^{(t)}v^{(t)}\right] \\
\nabla_{\mu^{(t)}}\mathcal{L}^{(t+1)} &= \mu^{(t)}A(v^{(t)}) + (1-\alpha^{(t)}h)\mathbb{E}\left[\theta^{(t)}v^{(t)}\right] = 0 \\
\mu^{(t)*} &= -\frac{(1-\alpha^{(t)}h)\mathbb{E}\left[\theta^{(t)}v^{(t)}\right]}{A(v^{(t)})} \\
\alpha^{(t)}h(A(\theta^{(t)}) + \sigma^2) &= A(\theta^{(t)}) - \frac{(1-\alpha^{(t)}h)\mathbb{E}\left[\theta^{(t)}v^{(t)}\right]}{A(v^{(t)})}\mathbb{E}\left[\theta^{(t)}v^{(t)}\right] \\
\alpha^{(t)*} &= \frac{A(\theta^{(t)})A(v^{(t)}) - \mathbb{E}\left[\theta^{(t)}v^{(t)}\right]^2}{h\left(A(v^{(t)})(A(\theta^{(t)}) + \sigma^2) - \mathbb{E}\left[\theta^{(t)}v^{(t)}\right]^2\right)}.
\end{aligned}$$

A.2.2 HIGH DIMENSION CASE

The loss is the sum of losses along all directions:

$$\mathcal{L}^{(t+1)} = \sum_i \frac{1}{2}h_i\Big[(1-\alpha^{(t)}h_i)^2A(\theta_i^{(t)}) + (\mu^{(t)})^2A(v_i^{(t)}) + 2\mu^{(t)}(1-\alpha^{(t)}h_i)\mathbb{E}\left[\theta_i^{(t)}v_i^{(t)}\right] + (\alpha^{(t)}h_i\sigma_i)^2\Big]$$

Now we obtain optimal learning rate and momentum by setting the derivative to 0,

$$\nabla_{\alpha^{(t)}}\mathcal{L}^{(t+1)} = \sum_i h_i\Big[(1-\alpha^{(t)}h_i)A(\theta_i^{(t)})\cdot(-h_i) - \mu^{(t)}h_i\mathbb{E}\left[\theta_i^{(t)}v_i^{(t)}\right] + \alpha^{(t)}(h_i\sigma_i)^2\Big] = 0$$

$$\alpha^{(t)}\sum_i\Big((h_i)^3(A(\theta_i^{(t)}) + (\sigma_i)^2)\Big) = \sum_i\Big((h_i)^2A(\theta_i^{(t)}) + \mu^{(t)}(h_i)^2\mathbb{E}\left[\theta_i^{(t)}v_i^{(t)}\right]\Big)$$

$$\nabla_{\mu^{(t)}}\mathcal{L}^{(t+1)} = \sum_i h_i\mu^{(t)}A(v_i^{(t)}) + h_i(1-\alpha^{(t)}h_i)\mathbb{E}\left[\theta_i^{(t)}v_i^{(t)}\right] = 0$$

$$\mu^{(t)*} = -\frac{\sum_i h_i(1-\alpha^{(t)}h_i)\mathbb{E}\left[\theta_i^{(t)}v_i^{(t)}\right]}{\sum_i h_iA(v_i^{(t)})}$$

$$\alpha^{(t)}\sum_i\left(\Big((h_i)^3(A(\theta_i^{(t)}) + (\sigma_i)^2)\Big)\Big(\sum_j h_jA(v_j^{(t)})\Big) - \Big(\sum_j(h_j)^2\mathbb{E}\left[\theta_j^{(t)}v_j^{(t)}\right]\Big)(h_i)^2\mathbb{E}\left[\theta_i^{(t)}v_i^{(t)}\right]\right) =$$

$$\sum_i\left((h_i)^2A(\theta_i^{(t)})\Big(\sum_j h_jA(v_j^{(t)})\Big) - \Big(\sum_j h_j\mathbb{E}\left[\theta_j^{(t)}v_j^{(t)}\right]\Big)(h_i)^2\mathbb{E}\left[\theta_i^{(t)}v_i^{(t)}\right]\right)$$

$$\alpha^{(t)*} = \frac{\sum_i\left((h_i)^2A(\theta_i^{(t)})\Big(\sum_j h_jA(v_j^{(t)})\Big) - \Big(\sum_j h_j\mathbb{E}\left[\theta_j^{(t)}v_j^{(t)}\right]\Big)(h_i)^2\mathbb{E}\left[\theta_i^{(t)}v_i^{(t)}\right]\right)}{\sum_i\left(\Big((h_i)^3(A(\theta_i^{(t)}) + (\sigma_i)^2)\Big)\Big(\sum_j h_jA(v_j^{(t)})\Big) - \Big(\sum_j(h_j)^2\mathbb{E}\left[\theta_j^{(t)}v_j^{(t)}\right]\Big)(h_i)^2\mathbb{E}\left[\theta_i^{(t)}v_i^{(t)}\right]\right)}.$$

A.3 Univariate Optimality in SGD

We now consider a dynamic programming approach to solve the problem. We formalize the optimization problem of $\{\alpha_i\}$ as follows. We first denote $\mathcal{L}_{\min}$ as the minimum expected loss at the last time step T (i.e., under the optimal learning rate).

$$\mathcal{L}_{\min} = \min_{\alpha^{(t)},\alpha^{(t+1)},\ldots,\alpha^{(T-1)}}\mathbb{E}_{\xi^{(t)},\xi^{(t+1)},\ldots,\xi^{(T-1)}}\left[\mathcal{L}(\theta^{(T)})\right].$$

Recall that the loss can be expressed in terms of the expectation and variance of θ. Denote $A^{(t)} = (\mathbb{E}\left[\theta^{(t)}\right])^2 + \mathbb{V}\left[\theta^{(t)}\right]$. The final loss can be expressed in terms of $A_{\min}^{(T)}$, obtained by using optimal learning rate schedule:

$$\mathcal{L}_{\min} = \frac{1}{2}hA_{\min}^{(T)} + \sigma^2.$$

As in Theorem 1, we derive the dynamics for SGD *without* momentum:

$$\theta^{(t)} = (1-\alpha^{(t-1)}h)\theta^{(t-1)} + \alpha^{(t-1)}h\sigma\xi^{(t-1)}$$

$$\Rightarrow\left(\mathbb{E}\left[\theta^{(t)}\right], \mathbb{V}\left[\theta^{(t)}\right]\right) = \left((1-\alpha^{(t-1)}h)\mathbb{E}\left[\theta^{(t-1)}\right], (1-\alpha^{(t-1)}h)^2\mathbb{V}\left[\theta^{(t-1)}\right] + (\alpha^{(t-1)}h\sigma)^2\right).$$

Thus, we can find a recurrence relation of the sequence $A^{(t)}$:

$$A^{(t)} = (1-\alpha^{(t-1)}h)^2\left((\mathbb{E}\left[\theta^{(t-1)}\right])^2 + \mathbb{V}\left[\theta^{(t-1)}\right]^2\right) + (\alpha^{(t-1)}h\sigma)^2 = (1-\alpha^{(t-1)}h)^2A_{\min}^{(T-1)} + (\alpha^{(t-1)}h\sigma)^2.$$

Since $A_{\min}^{(T)}$ is a function of $\alpha^{(T-1)*}$. We can obtain optimal learning rate $\alpha^{(T-1)*}$ by taking the derivative of $\mathcal{L}_{\min}$ w.r.t. $\alpha^{(T-1)}$ and setting it to zero:

$$\begin{aligned}\frac{d\mathcal{L}_{\min}}{d\alpha^{(T-1)}} = \frac{1}{2}h\frac{dA_{\min}^{(T-1)}}{d\alpha^{(T-2)}} &= 0\\ \Rightarrow\quad \frac{dA_{\min}^{(T)}}{d\alpha^{(T-1)}} &= 0\\ \Rightarrow\quad \alpha^{(T-1)*} &= \frac{A_{\min}^{(T-1)}}{h(A_{\min}^{(T-1)} + \sigma^2)}.\end{aligned}$$

Thus we can write $A_{\min}^{(T)}$ in terms of $A_{\min}^{(T-1)}$ and the optimal $\alpha^{(T-1)*}$:

$$\begin{aligned} A_{\min}^{(T)} &= \left(1 - \frac{A_{\min}^{(T-1)}}{A_{\min}^{(T-1)} + \sigma^2}\right)^2 A_{\min}^{(T-1)} + \left(\frac{A_{\min}^{(T-1)}}{A_{\min}^{(T-1)} + \sigma^2}\sigma\right)^2 \\ &= \left(\frac{\sigma^2}{A_{\min}^{(T-1)} + \sigma^2}\right)^2 A_{\min}^{(T-1)} + \left(\frac{A_{\min}^{(T-1)}}{A_{\min}^{(T-1)} + \sigma^2}\sigma\right)^2 \\ &= \frac{A_{\min}^{(T-1)}\sigma^2}{A_{\min}^{(T-1)} + \sigma^2}. \end{aligned}$$

Therefore,

$$\begin{aligned} \mathcal{L}_{\min} &= \frac{1}{2}hA_{\min}^{(T)} + \sigma^2 \\ &= \frac{1}{2}h\left(\frac{A_{\min}^{(T-1)}\sigma^2}{A_{\min}^{(T-1)} + \sigma^2}\right) + \sigma^2. \end{aligned}$$

We now generalize the above derivation. First rewrite $\mathcal{L}_{\min}$ in terms of $A_{\min}^{T-k}$ and calculate the optimal learning rate at time step $T-k$.

Theorem 4. *For all $T \in \mathbb{N}$, and $k \in \mathbb{N}$, $1 \leq k \leq T$, we have,*

$$\mathcal{L}_{\min} = \frac{1}{2}h\left(\frac{A_{\min}^{(T-k)}\sigma^2}{kA_{\min}^{(T-k)} + \sigma^2}\right) + \sigma^2. \tag{16}$$

Therefore, the optimal learning $\alpha^{(t)}$ at timestep t is given as,

$$\alpha^{(t)*} = \frac{A^{(t)}}{h(A^{(t)} + \sigma^2)}. \tag{17}$$

Proof. The form of $\mathcal{L}_{\min}$ can be easily proven by induction on k, and use the identity that,

$$\frac{(\frac{ab}{a+b})b}{k(\frac{ab}{a+b}) + b} = \frac{ab}{(k+1)a + b}.$$

The optimal learning rate then follows immediately by taking the derivative of $\mathcal{L}_{\min}$ w.r.t. $\alpha^{(T-k-1)}$ and setting it to zero. Note that the subscript $\min$ is omitted from $A^{(t)}$ in Eq.(17) as we assume all $A^{(t)}$ are obtained using optimal α^*, and hence minimum. □

Critical Hyper-Parameters: No Random, No Cry

Olivier Bousquet, Sylvain Gelly, Karol Kurach,
Olivier Teytaud, Damien Vincent, Google Brain, Zürich

Abstract

The selection of hyper-parameters is critical in Deep Learning. Because of the long training time of complex models and the availability of compute resources in the cloud, "one-shot" optimization schemes – where the sets of hyper-parameters are selected in advance (e.g. on a grid or in a random manner) and the training is executed in parallel – are commonly used. [1] show that grid search is sub-optimal, especially when only a few critical parameters matter, and suggest to use random search instead. Yet, random search can be "unlucky" and produce sets of values that leave some part of the domain unexplored. Quasi-random methods, such as Low Discrepancy Sequences (LDS) avoid these issues. We show that such methods have theoretical properties that make them appealing for performing hyperparameter search, and demonstrate that, when applied to the selection of hyperparameters of complex Deep Learning models (such as state-of-the-art LSTM language models and image classification models), they yield suitable hyperparameters values with much fewer runs than random search. We propose a particularly simple LDS method which can be used as a drop-in replacement for grid/random search in any Deep Learning pipeline, both as a fully one-shot hyperparameter search or as an initializer in iterative batch optimization.

1 Introduction

Hyperparameter (HP) optimization can be interpreted as the black-box search of an x, such that, for a given function $f : S \subset \mathcal{R}^d \to \mathcal{R}$, the value $f(x)$ is small, where the function f can be stochastic. This captures the situation where one is looking for the best setting of the hyper-parameters of a (possibly randomized) machine learning algorithm by trying several values of these parameters and picking the value yielding the best validation error. Non-linear optimization in general is a well-developed area [18]. However, hyper-parameter optimization in the context of Deep Learning has several specific features that need to be taken into account to develop appropriate optimization techniques: function evaluation is very expensive, computations can be parallelized, derivatives are not easily accessible, and there is a discrepancy between training, validation and test errors. Also, there can be very deep and narrow optima. Published methods for hyper-parameters search include evolution strategies [2], Gaussian processes [10, 23], pattern search [4], grid sampling and random sampling [1]. In this work we explore one-shot optimization, hence focusing on non-iterative methods like random search[1]. This type of optimization is popular - extremely scalable and easy to implement. Among one-shot optimization methods, grid sampling is sub-optimal if only a few critical parameters matter because the same values of these parameters will be explored many times. Another very popular approach, random sampling, can suffer from unlucky draws, and leave some part of the space unexplored. To address those issues, we investigate Low Discrepancy Sequences (LDS) to see if they can improve upon currently used methods, an approach also suggested by [1] as future work (although they already proposed the use of LDS and did preliminary experiments on artificial

[1]Note that most of the improvements on one-shot optimization can be applied to batch iterative methods as well, as shown by Section E.7.

objective functions, they did not run experiments on HP tuning for Deep Learning as we do here). LDS is a rich family of different methods to produce sequences well spread in $[0,1]^d$. Based on a theoretical analysis and empirical evaluation, we suggest Randomly Scrambled Hammersley with random shift as a robust one-shot optimization method for HP search in Deep Learning.

2 One-shot optimization: formalization & algorithms

In this work, we focus on so-called one-shot optimization methods where one chooses a priori the points at which the function will be evaluated and does not alter this choice based on observed values of the function. This reflects the parallel tuning of HPs where one runs the training of a learning algorithm with several choices of HPs and simply picks the best choice (based on the validation error). We do not consider the possibility of saving computational resources by stopping some runs early based on an estimate of their final performance, as is done by methods such as Sequential Halving [11] and Hyperband [14] or validation curves modeling as in [5]. However, the techniques presented here can easily be combined with such stopping methods.

2.1 Formalization

We consider a function f defined on $[0,1]^d$ and are interested in finding its infimum. To simplify the analysis, we will assume that the infimum of f is reached at some point $x^* \in [0,1]^d$. A one-shot optimization algorithm is a (possibly randomized) algorithm that produces a sequence $x_1, \ldots, x_n$ of elements of $[0,1]^d$ and, given any function f, returns the value $\min_i f(x_i)$. Since the choice of the sequence is independent of the function f and its values, we can simply see such an algorithm as producing a distribution P over sets $\{x_1, \ldots, x_n\}$. The performance of such an algorithm is then measured via the *optimization error*: $|\min_i f(x_i) - \inf_{x\in[0,1]^d} f(x)| = |\min_i f(x_i) - f(x^*)|$. This is a random variable and we are thus interested in its quantiles. We would like algorithms that make this quantity small with high probability. Since we have to choose the sequence before having any information about the function, it is natural to try to have a good coverage of the domain. In particular, by ensuring that any point of the domain has a close neighbor in the sequence, we can get a control on the optimization error (provided we have some knowledge of how the function behaves with respect to the distance between points). In order to formalize this, we first introduce several notions of how "spread" a particular sequence is.

Definition 1 (Volume Dispersion) *The* volume dispersion *of* $S = \{x_1, \ldots, x_n\} \subset [0,1]^d$ *with respect to a family* $\mathcal{R}$ *of subsets of* $[0,1]^d$ *is* $vdisp(S, \mathcal{R}) := \sup\{\mu(R) : R \in \mathcal{R},\ R \cap S = \emptyset\}$.

In particular, we will be interested in the case where the family $\mathcal{R}$ is the set $\mathbb{B}$ of all balls $B(x, \epsilon)$, in which case instead of considering the volume we can consider directly the radius of the ball.

Definition 2 (Dispersion) *The* dispersion *of a set* $S = \{x_1, \ldots, x_n\} \subset [0,1]^d$ *is defined as* $disp(S) := \sup\{\epsilon : x \in [0,1]^d,\ B(x,\epsilon) \cap S = \emptyset\}$.

Since we are interested in sequences that are stochastic, we introduce a more general notion of dispersion. Imagine that the sets S are generated by sampling from a distribution P.

Definition 3 (Stochastic Dispersion) *The* stochastic dispersion *of a distribution* P *over sets* $S = \{x_1, \ldots, x_n\} \subset [0,1]^d$ *at confidence* $\delta \in [0,1]$ *is defined as*

$$sdisp(P, \delta) := \sup_{x\in[0,1]^d} \sup\{\epsilon : P(\min_i \|x_i - x\| > \epsilon) \geq 1 - \delta\}, \tag{1}$$

$$sdisp'(P, \delta) := \sup_{x\in[0,1]^d} \sup\{\epsilon : P(\min_i \|x_i - x\|' > \epsilon) \geq 1 - \delta\}. \tag{2}$$

where $\|.\|'$ is the torus distance in $[0,1]^d$. We present a simple result that connects the (stochastic) dispersion to the optimization error when the function is well behaved around its infimum.

Lemma 1 *Let* $\omega(f, x^*\delta)$ *be the modulus of continuity of* f *around* x^*, $\omega(f, x^*, \delta) = \sup_{y:\|x^*-y\|\leq\delta} |f(x^*) - f(y)|$. *Then for any fixed sequence* S, $|\min_{x\in S} f(x) - f(x^*)| \leq \omega(f, x^*, disp(S))$, *and for any distribution* P *over sequences, with probability at least* $1 - \delta$ *(when* S *is sampled according to* P*),* $|\min_{x\in S} f(x) - f(x^*)| \leq \omega(f, x^*, sdisp(P, \delta))$.

In particular, if the function is known to be Lipschitz then its modulus of continuity is a linear function $\omega(f, x^*, \delta) \leq L(f)\delta$. In view of the above lemma, the (stochastic) dispersion gives a direct control on the optimization error (provided one has some knowledge about the behavior of the function). We will thus present our results in terms of the stochastic dispersion for various algorithms.

2.2 Sampling Algorithms

2.2.1 Grid and Random

Grid and random are the two most widely used algorithms to choose HPs in Deep Learning. Let us assume that $n = k^d$. Then **Grid** sampling consists in choosing k values for each axis and then taking all k^d points which can be obtained with these k values per axis. The value k does not have to be the same for all axes; this has no impact on the present discussion. There are various tools for choosing the k values per axis; evenly spaced, or purely randomly, or in a stratified manner. **Random** sampling consists in picking n independently and uniformly sampled vectors in $[0, 1]^d$.

2.2.2 Low Discrepancy Sequences

Low-discrepancy sequences have been heavily used in numerical integration but seldom in one-shot optimization. In the case of numerical integration there exists a tight connection between the integration error and a measure of "spread" of the points called discrepancy.

Definition 4 (Discrepancy) *The* discrepancy *of a set $S = \{x_1, \ldots, x_n\} \subset [0, 1]^d$ with respect to a family $\mathcal{R}$ of subsets of $[0, 1]^d$ is defined as $disc(S, \mathcal{R}) := \sup_{R \in \mathcal{R}} |\mu_S(R) - \mu(R)|$, where $\mu(R)$ is the Lebesgue measure of R and $\mu_S(R)$ is the fraction of elements of S that belong to R.*

LDS refer to algorithms constructing S of size n such that $disc(S, \mathcal{H}_0) = O(log(n)^d/n)$. One common way of generating such sequences is as follows: pick d coprime integers $q_1, \ldots, q_d$. For an integer k, if $k = \sum_{j \geq 0} b_j q^j$ is its q-ary representation, then we define $\gamma_q(k) := \sum_{j \geq 0} b_j q^{-j-1}$ which corresponds to the point in $[0, 1]$ whose q-ary representation is the reverse of that of k. Then the **Halton** sequence [7] is defined as $\{(\gamma_{q_1}(k), \ldots, \gamma_{q_d}(k)) : 1 \leq k \leq n\}$ and the **Hammersley** sequence is defined as $\{((k - \frac{1}{2})/n, \gamma_{q_1}(k), \ldots, \gamma_{q_{d-1}}(k)) : 1 \leq k \leq n\}$ One can also define scrambled versions of those sequences by randomly permuting the digits in the q-ary expansion (with a fixed permutation). The **Sobol** sequence [19] is also a popular choice and can be constructed using Gray codes [3]. It has the property that (for $n = 2^d$) all hypercubes obtained by splitting each axis into two equal parts contain exactly one point. We use the publicly available implementation of Sobol by John Burkardt (2009).

It is often desirable to randomize LDS. This improves their robustness and permits repeated distinct runs. Some LDS are randomized by nature, e.g. when the scrambling is randomized, as in Halton scrambling. Another randomization consists in discarding the first k points, with k randomly chosen. A simple and generic randomization technique, called **Random Shifting**, consists in shifting by a random vector in the unit box; i.e. with $x = (x_1, \ldots, x_n)$ a sampling in $[0, 1]^d$, we randomly draw a uniformly in $[0, 1]^d$, and the randomly shifted counterpart of x is $mod(x_1 + a, 1), mod(x_2 + a, 1), \ldots, mod(x_n + a, 1)$ (where mod is the coordinate-wise modulo operator). Random shifting does not change the discrepancy (asymptotically) and provides low variance numerical integration [22]. The shifted versions of scrambled Halton and scrambled Hammersley are denoted by S-Ha and S-SH respectively.

2.2.3 Latin Hypercube Sampling (LHS)

LHS [6, 16] construct a sequence as follows: for a given n, consider the partition of $[0, 1]^d$ into a regular grid of n^d cells and then define the index of a cell as its position in the corresponding n^d grid; choose d random permutations $\sigma_1, \ldots, \sigma_d$ of $\{1, \ldots, n\}$ and choose for each $k = 1, \ldots, n$, the cell whose index is $\sigma_1(k), \ldots, \sigma_d(k)$, and choose x_k uniformly at random in this cell. LHS ensures that all marginal projections on one axis have one point in each of the n regular intervals partitioning $[0, 1]$. On the other hand, LHS can be unlucky; if $\sigma_j(i) = i$ for each $j \in \{1, 2, \ldots, d\}$ and $i \in \{1, 2, \ldots, n\}$, then all points will be on the diagonal cells. Several variants of LHS exist that prevent such issues, in particular orthogonal sampling [21].

3 Theoretical Analysis

Due to length constraints all proofs are reported to the supplementary material.

Desirable properties. Here are some properties that are particularly relevant in the setup of parallel HP optimization: **No bad sequence**: When using randomized algorithms to generate sequences, one desirable property is that the probability of getting a bad sequence (and thus missing the optimal setting of the HPs by a large amount) should be as low as possible. In other words, one would want to have some way to avoid being unlucky. **Robustness w.r.t. irrelevant parameters**: the performance of the search procedure should not be affected by the addition of irrelevant parameters. Indeed, it is often the case that one develops algorithms with many HPs, some of which do not affect the performance of the algorithm, or affect it only mildly. Otherwise stated, when projecting to a subset of critical variables, we get a point set with similar properties on this lower dimensional subspace. As argued in [1], this concept is critical in HP optimization. **Consistency**: The produced sequence should be such that the optimization error converges to zero as n goes to ∞. **Optimal Dispersion**: it is known that the best possible dispersion for a sequence of n points is of order $1/n^{1/d}$, so it is desirable to have a sequence whose dispersion is within a constant factor of this optimal rate. We first mention a result from [17] which gives an estimate (tight up to constant factors) of the dispersion (with respect to hyperrectangles) of some known LDS.

Theorem 1 ([17]) *Consider S_n a set of n Halton points (resp. n Hammersley points, resp. n (t, m, s)-net points) in dimension d, then $vdisp(S_n, \mathcal{H}) = \Theta(1/n)$*

Let V_d be the volume of the unit ball in dimension d, V'_d the volume of the orthant of this ball intersecting $[0, \infty)^d$ and K_d be the volume of the largest hypercube included in this orthant. The following lemma gives relations between the different measures of spread we have introduced, showing that the discrepancy upper bounds all the others.

Lemma 2 (Relations between measures) *For any distribution P and any $\delta \geq 0$, $sdisp(P, \delta) \leq sdisp(P, 0)$. And if P generates only one sequence S, then $sdisp(P, 0) = disp(S) \leq (vdisp(S, \mathbb{B})/V'_d)^{1/d} \leq (K_d vdisp(S, \mathcal{H})/V'_d)^{1/d}$. Also we have for any family $\mathcal{R}$, $vdisp(S, \mathcal{R}) \leq disc(S, \mathcal{R})$.*

3.1 Dispersion of Sampling Algorithms & Projection to Critical Variables

The first observation is that if we compare (shifted) Grid and (shifted) LDS such as considered in Theorem 1 their stochastic dispersion is of the same asymptotic order $1/n^{1/d}$.

Theorem 2 (Asymptotic Rate) *Consider Random, (shifted) Grid, or a shifted LDS such as those considered in Theorem 1. For any fixed δ, there exists a constant c_δ such that for n large enough, $sdisp(P, \delta) \leq K \left(\frac{c_\delta}{n}\right)^{1/d}$.*

Remark 1 *One natural question to ask is whether the dependence on δ is different for different sequences. We can prove a slightly better bound on the stochastic dispersion for shifted LDS or shifted grid than for random. In particular we can show that, for all $0 < c < \frac{1}{2}$, for δ close enough to 1, $sdisp'(P, \delta) \leq K \left(\frac{1-\delta}{n}\right)^{1/d}$. The constant K depends on the considered sequence and on δ; and for δ close enough to 1 the constant is better for S-SH or a randomly shifted grid than for random. This follows from noticing that when the points of the sequence are at least 2ϵ-separated, then the probability of x^* to be in the ϵ-neighborhood (for the torus distance) of any point of the sequence is $n\epsilon^d V_d$.*

Next we observe that the Random sequences have one major drawback which is that they cannot guarantee a small dispersion, while for Grid or LDS, we can get a small dispersion with probability 1. The following theorem follows from the proof of theorem 2:

Theorem 3 (Guaranteed Success) *For Random, $\lim_{\delta \to 0} sdisp(P, \delta) = 1$. However, for Grid and Halton/Hammersley, $sdisp(P, 0) = O(1/n^{1/d})$.*

Finally, we show that when the function f depends only on a subset of the variables, LDS and Random provide lower dispersion while this is not the case for Grid. Assume $f(x)$ depends only on

$(x_i)_{i \in I}$ for some subset I of $\{1, \dots, d\}$. Then the optimization error is controlled by the stochastic dispersion of the projection of the sequence on the coordinates in I. Given a sequence S, we define S_I as the sequence of projections on coordinates I of the elements of S, and we define P_I as the distribution of S_I when S is distributed according to P.

Theorem 4 (Dependence on Critical Variables) *We have the following bounds on the stochastic dispersion of projections: (i) For Random, Halton or Hammersley* $sdisp(P_I, \delta) = O(1/n^{1/|I|})$*, (ii) for Grid* $sdisp(P_I, \delta) = \Theta(1/n^{1/d})$.

Remark 2 (Importance of ranking variables properly) *While the above result suggests that the dispersion of the projection only depends on the number of important variables, the performance in the non-asymptotic regime actually depends on the order of the important variables among all coordinates. Indeed, for Halton and Hammersley, the distribution on any given coordinate will become uniform only when* n *is larger than the corresponding* q_i*. Since the* q_i *have to be coprime, this means that assuming the* q_i *are sorted, they will each be not smaller than the* i*-th prime number. So what will determine the quality of the sequence when only variables in* I *matter is the value* $\max_{i \in I} q_i$*. Hence it is preferable to assign small* q_i *to the important variables.*

The conclusion of this section, as illustrated by Fig. 1 is that among the various algorithms considered here, only the scrambled/shifted variants of Halton and Hammersley have all the desirable properties from Section 3. We will see in Section 4 that, besides having the desirable theoretical properties, the Halton/Hammersley variants also reach the best empirical performance.

3.2 Pathological Functions

As we have seen above, the dispersion can give a characterization of the optimization error in cases where the modulus of continuity around the optimum is well behaved. Hence convergence to zero of the dispersion is a sufficient condition for consistency. However, one can construct pathological functions for which the considered algorithms fail to give a low optimization error.

Deterministic sequences are inconsistent: In particular, if the sequence is deterministic one can always construct a function on which the optimization will not converge at all. Indeed, given a sequence $x_1, \dots, x_n, \dots$, one can construct a function such that $f(x_i) = 1$ for any i and $f(x) = 0$ otherwise except in the neighborhood of the x_i. Obviously the optimization error will be 1 for any n, meaning that the optimization procedure is not even consistent (i.e. fails to converge to the essential infimum of the function in the limit of $n \to \infty$).

Shifted sequences are consistent but can be worse than Random: One obvious fix for this issue is to use non-deterministic sequences by adding a random shift. This will guarantee that the optimization is consistent with probability 1. However, since we are shifting by the same vector every point in the deterministic sequence, we can still construct a pathological function which will be such that the optimization with the shifted sequence performs significantly worse that with a pure Random strategy. Indeed, imagine a function which is equal to 1 on balls $B(x_i, \epsilon)$ (for torus distance) for some small enough ϵ and which is equal to 0 everywhere else (ϵ has to be small enough for the balls not to cover the whole space, so ϵ depends on n). In this case, the probability that the shifted sequence gives an optimization error of 1 is proportional to ϵV_d, while it is less than $(n\epsilon V_d)^n$ (which is much smaller for ϵ small enough) for Random. This proof does not cover stochastic LDS; but it can be extended to random shifts of any possibly stochastic point set which has a positive probability for at least one fixed set of values - this covers all usual LDS, stochastic or not.

Other pathological examples: If we compare the various algorithms to Random, it is possible to construct pathological functions which will lead to worse performance than Random as illustrated on Figure 1 (right).

4 Experiments

We perform extensive empirical evaluation of LDS. We first validated the theory using a set of simple toy optimization problems, these fast problems provide an extensive validation with negligible p-values. Results (given in Section C) illustrate each claim - the performance of LDS for one-shot optimization (compared to random and LHS), the positive effect of Hammersley (compared to Halton)

Property	Grid	Rand	LHS	S-Ha	S-SH
No Bad Sequence	✓	✗	✗	✓	✓
Robust Irrelevant Params	✗	✓	✓	✓	✓
Consistency	✓	✓	✓	✓	✓
Optimal dispersion	✓	✓	✓	✓	✓

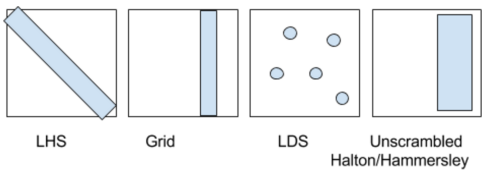


Figure 1: Left: Summary of the properties for some of the considered sampling methods. S-SH and S-Ha have all the desired properties. Right: Pathological examples where various sampling algorithms will perform worse than Random. LHS can produce sequences that are aligned with the diagonal of the domain or that are completely off-diagonal with a higher probability than Random. This can be exploited by a function with high values on the diagonal and low values everywhere else. Grid: when the area with low values is thin and depends on one axis only, Grid is more likely to fail than Random. LDS (with or without random shift): as explained in Section 3.2, if the function values are high around the points in the sequence and low otherwise, even with random shift, the performance will be worse than Random. Halton or Hammersley without scrambling: due to the sequential nature of the function $\gamma_q(k)$, the left part of the domain (lower values) is sampled more frequently than the right hand side.

and of scrambling, the additional improvement when variables are ranked by decreasing importance, and the existence of counter-examples.

We perform some real-world deep learning experiments to further confirm the good properties of S-SH for hyper-parameter optimization. We finally broaden the use of LDS as a first-step initialization in the context of Bayesian optimization. Some additional experiments are provided in Section E, the additional results are consistent with the claims.

4.1 Deep Learning tasks

Metrics We need to have a consistent and robust way to compare the sampling algorithms which are inherently stochastic. For this purpose, we can simply measure the **probability** p that a sampling algorithm S performs better than random search, when both use the same budget b of hyper-parameters trials. From this "win rate" probability p, we can simply define a **speed-up** s as $s = \frac{2p-1}{1-p}$. Given two instances of random search R_1 and R_2, with different budgets b_1 and b_2, s refers to the additional budget needed for R_1 to be better than R_2 with probability p. That is, if $b_1 = (1+s)b_2$, then $p = \frac{1+s}{2+s}$ or equivalently $s = \frac{2p-1}{1-p}$.

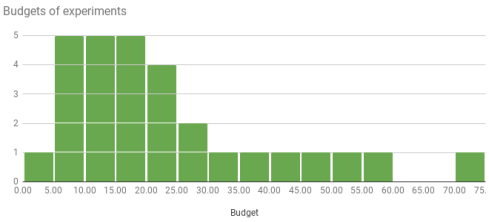
In addition to the speed-up s, we report p and, when we have enough experiments on a single setup for having meaningful such statistics, we also report the raw improvements in validation score when using the same budget.

Language Modeling We use language modeling as one of the challenging domain. Our datasets include Penn Tree Bank (PTB) [15], using both a word level representation and a byte level representation, and a variant UB-PTB that we created from PTB by randomly permuting blocks of 200 lines, to avoid systematic bias between train/validation/test[2]. Additionally, we use a subset of the Enwik8 [9] dataset, about 6%, which we name MiniWiki, and the corresponding shuffled version MiniUBWiki.

We use close to state-of-the-art LSTM models as language models, and measure the perplexity (or bit-per-byte) as the target loss. We compare S-SH and LHS with random search on all those datasets, tuning 5 HPs. More details can be found in *setup A* of Section B.

The results can be found in Fig. 2. Using that setup, both LHS and S-SH outperform random on all considered metrics, and for all budgets. LHS outperforms S-SH for small budgets (<11 values explored), while S-SH is best for larger budgets.

[2]PTB and Enwik8 have systematic differences between the distributions of the train and the validation/test parts, because it is split in the order of the text. That can create systematic biases when an algorithm is more heavily tuned on the validation set.



	budget < 11	budget$\in \{11, \ldots, 28\}$	budget $> 28)$	All budgets
S-SH win rate	0.529 ± 0.04	$\mathbf{0.638 \pm 0.03}$	$0.513\% \pm 0.03$	$\mathbf{0.569 \pm 0.03}$
S-SH speed-up	+12.3%	**+76.6%**	+5.3%	**+32%**
S-SH bit-per-byte	/	**-1.1%**	/	-0.41%
LHS win rate	$\mathbf{0.686 \pm 0.06}$	0.546 ± 0.04	0.50 ± 0.07	0.555 ± 0.05
LHS speed-up	**+118%**	+20%	+0%	+24.7%
LHS bit-per-byte	/	-0.08%	/	-0.48%

Figure 2: Experiments on real-world language modeling, depending on the budget: we provide frequencies at which S-SH (resp. LHS) outperforms random and speedup interpretations. Left: Histogram of budgets used for comparing LHS, S-SH and Random. Right: Winning rate of S-SH and LHS compared to random, on language modeling tasks (PTB, UBPTB, MiniWiki) with various budgets.

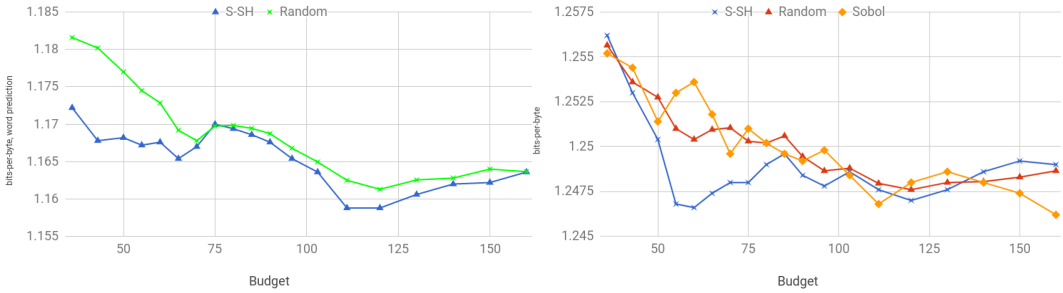


Figure 3: Comparison between the average loss (bits per byte) for S-SH and for random on PTB-words (left) and PTB-words (right). Moving average (5 successives values) of the performance for setup C in Section B (3 HPs).

We then perform additional experiments, with larger numbers of epochs and bigger nets, on language modeling. Results in Fig. 3 confirm that S-SH can provide a significant improvement, especially for budgets 30-60 for the word-PTB model and for budgets greater than 60 for the byte-PTB model. Detailed p-values for that experiment are given in Section E.

Image classification experiments In addition to language modeling, image classification is representative of current Deep Learning challenges. We use a classical CNN model trained on Cifar10 [12]. The chosen architecture is not the state-of-the-art but is representative of the mainstream ones. Those experiments are tuning 6 HPs. More details can be found in *setup B* in Section B.

Results are presented in Fig. 4. Except for very small budgets (< 5, for which S-SH fails and for which we might recommend LHS rather than LDS; see also Fig. 2 on this LHS/S-SH comparison for small budget) we usually get better results with S-SH.

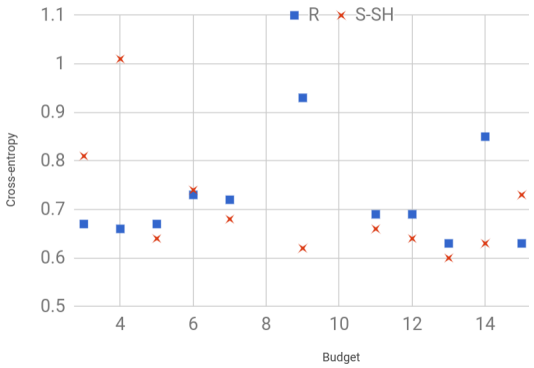


Figure 4: Cifar10: test loss for random and S-SH as a function of the budget.

Dimension 12

	Rastrigin	Sphere	Ellip.	Styb.	Beale	Branin	6Hump
1 batch							
Pessimistic F.	1.00	1.05	1.03	1.00	0.98	1.83	1.18
Random init.	1.02	1.30	1.47	1.00	1.07	1.22	1.31
LHS	1.00	1.04	1.09	0.99	0.99	1.72	1.17
3 batches							
Pessimistic F.	1.01	1.06	1.26	1.00	0.83	1.20	0.91
Random init.	1.02	1.21	1.20	1.00	0.84	1.08	0.90
LHS	1.03	0.99	1.17	1.00	0.93	1.16	0.99
5 batches							
Pessimistic F.	0.98	1.30	1.47	1.00	1.25	0.89	0.97
Random init.	1.01	1.54	1.29	1.00	1.12	0.67	0.83
LHS	0.98	1.44	1.17	1.00	1.34	1.01	1.03

Dimension 2

	Rastrigin	Sphere	Ellip.	Styb.	Beale	Branin	6Hump
1 batch							
Pessimistic F.	0.96	1.14	0.95	0.90	0.81	0.95	1.22
Random init.	1.01	1.24	1.43	0.95	1.14	0.88	1.13
LHS	1.02	1.17	1.04	0.91	0.84	1.01	1.25
3 batches							
Pessimistic F.	1.12	1.38	1.24	0.03	0.82	1.47	.88
Random init.	1.07	0.83	1.00	0.41	0.89	3.14	1.38
LHS	1.01	1.45	1.48	0.06	0.73	0.78	1.19
5 batches							
Pessimistic F.	0.93	1.27	1.10	0.71	1.94	1.61	0.68
Random init.	1.01	1.08	1.09	0.58	5.01	1.60	1.14
LHS	1.05	2.09	0.33	0.87	2.56	1.62	0.63

Table 1: Ratio "loss of a method / loss of LDS-BO method" (i.e. > 1 means LDS-BO wins) for 64 runs per batch. In dimension 12 for small number of batches LDS-BO clearly wins against competitors; for larger budgets the pessimistic fantasizing can compete (LDS-BO still usually better); in dimension 2 pessimistic fantasizing and LDS-BO perform best, with a significant advantage compared to LHS-BO and random. Fig. 4 (supplementary material) shows that on the other hand, and consistently with known bounds on discrepancy, LHS-BO outperforms LDS-BO for small budget.

4.2 LDS as a first-step in a Bayesian optimization framework.

LDS (here S-Ha) also succeeds in the context of sampling the first batch in a Bayesian optimization run (Table 1), LDS outperforms (a) random sampling, (b) LHS, and (c) methods based on pessimistically fantasizing the objective value of sampled vectors[8]. For rare cases Bayesian optimization based on LHS or pessimistic fantasizing outperform our LDS-based Bayesian optimization even for one single batch - but in dimension 12 LDS-BO (LDS-initialized Bayesian optimization) still performed best for most numbers of batches in $\{1, 3, 5\}$ and most functions in $\{Branin, Beale, SixHump, Rastrigin, Ellipsoidal, Sphere\}$. Results were mixed in dimension 2.

5 Conclusion

We analyzed theoretically and experimentally the performance of LDS for hyper-parameter optimization. The convergence rate results show that LDS are strictly better than random search both for high confidence and low confidence quantiles (Theorem 3 and Remark 1). In particular, with enough points, LDS can find the optimum within ϵ distance with probability exactly 1, due to the absence of unlucky cases, unlike random search. For intermediate quantiles the theoretical rate bounds are equivalent between LDS and random search but we could not prove that the constant is better except in dimension 1. In the common case where parameters have different impacts, and only a few really matter, we show that parameters should be ranked (Remark 2). Further we prove that LDS outperforms grid search (in case some variables are critically important) and the best LDS sequences are robust against bad parameter ranking (Theorem 4). The experiments confirmed and extended our theoretical results: LDS is consistently outperforming random search, as well as LHS except for small budget. Importantly, while the theory covers only the one-shot case, we get great performance for LDS as a first step in Bayesian optimization[10], even compared to entropy search[8] or LHS-initialized Bayesian optimization.

As a summary of this study, we suggest Randomly Scrambled Hammersley with random shift as a robust one-shot optimization method. Replacing random search with this sampling method is a very simple change in a Deep Learning training pipeline, can bring some significant speedup and we believe should be adopted by most practitioners - though for budgets < 10 we might prefer LHS. To spread the use of LDS as a way to optimize HPs, we will open-source a small library at https://AnonymizedUrl. Our results also support the application of LDS for the first batch of Bayesian optimization. Importantly, not all LDS perform equally - we were never disappointed by S-SH, but Sobol (used in [13]) or unscrambled variants of Halton are risky alternatives.

References

[1] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, Feb. 2012.

[2] H. Beyer. *The theory of evolution strategies*. Natural computing series. Springer, Berlin, 2001.

[3] P. Bratley and B. L. Fox. Algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 14(1):88–100, Mar. 1988.

[4] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[5] T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3460–3468. AAAI Press, 2015.

[6] V. Eglajs and P. Audze. New approach to the design of multifactor experiments. *Problems of Dynamics and Strengths*, 35:104–107, 1977.

[7] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, Dec. 1964.

[8] P. Hennig and C. J. Schuler. Entropy search for information-efficient global optimization. *CoRR*, abs/1112.1217, 2011.

[9] M. Hutter. 50000 euros prize for compressing human knowledge. `http://prize.hutter1.net/`, 2006.

[10] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *J. of Global Optimization*, 13(4):455–492, Dec. 1998.

[11] Z. S. Karnin, T. Koren, and O. Somekh. Almost Optimal Exploration in Multi-Armed Bandits. In *Proceedings of the 30th International Conference on Machine Learning, Cycle 3*, volume 28 of *JMLR Proceedings*, pages 1238–1246. JMLR.org, 2013.

[12] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research). 2009.

[13] M. J. Kusner, J. R. Gardner, R. Garnett, and K. Q. Weinberger. Differentially private bayesian optimization. In F. R. Bach and D. M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 918–927. JMLR.org, 2015.

[14] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016.

[15] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330, June 1993.

[16] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61, Feb. 2000.

[17] G. Rote and R. F. Tichy. Quasi-monte-carlo methods and the dispersion of point sequences. *Math. Comput. Model.*, 23(8-9):9–23, Apr. 1996.

[18] A. Ruszczynski. *Nonlinear Optimization*. Princeton University Press, Princeton, NJ, USA, 2006.

[19] I. Sobol. Uniformly distributed sequences with an additional uniform property. *U.S.S.R. Comput. Maths. Math. Phys.*, 16:236–242, 1976.

[20] A. Sukharev. Optimal strategies of the search for an extremum. *USSR Computational Mathematics and Mathematical Physics*, 11(4):119 – 137, 1971.

[21] B. Tang. Orthogonal array-based latin hypercubes. *Journal of the American Statistical Association*, 88:1392–1397, 1993.

[22] B. Tuffin. On the use of low discrepancy sequences in monte carlo methods. *Monte Carlo Methods and Applications*, 2(4):295–320, 1996.

[23] J. Villemonteix, E. Vazquez, and E. Walter. An informational approach to the global optimization of expensive-to-evaluate functions. *J. of Global Optimization*, 44(4):509–534, Aug. 2009.

[24] C. Wang, Q. Duan, W. Gong, A. Ye, Z. Di, and C. Miao. An evaluation of adaptive surrogate modeling based optimization with two benchmark problems. *Environmental Modelling & Software*, 60:167 – 179, 2014.

Supplementary material — critical hyperparameters: no random, no cry.

A Proofs

Proof [Lemma 2] A ball of radius ϵ has volume $V_d\epsilon^d$, so if v is the volume of a ball, its radius is $(v/V_d)^{1/d}$. If v is the largest volume of a ball not containing any point of S, then the largest hypercube contained in this ball, which has volume $v.K_d$ will also be empty, which shows that $vdisp(S, \mathbb{B}) \leq K_d vdisp(S, \mathcal{H})$. Finally it is easy to see that the volume dispersion is upper bounded by the discrepancy as it follows from their definition. ∎

Proof [Theorem 2] For Grid this is a known result[20]. For Random, the probability of n randomly picked balls of radius ϵ to contain any particular point x^* is upper bounded by $1 - (1 - V'_d\epsilon^d)^n$, hence this gives an upper bound of $\left(\frac{\log 1/\delta}{n}\right)^{1/d}$ for n large with respect to $\log 1/\delta$.

For the LDS sequences, this follows from combining Lemma 2 with Theorem 1. ∎

Proof [Theorem 4] The projection of Random (resp. Halton) sequences are Random (resp. Halton) sequences. For Hammersley the first coordinate is uniform, so the projection onto I is Hammersley if $1 \in I$ or is Halton otherwise. Theorem 2 allows to conclude. The result for Grid is obvious. ∎

B Detailed experiment setups

Setup A Language model with 3 layers of LSTM with 250 units trained for 6 epochs. In this setup, we have 5 hyper-parameters: weight init scale $[0.02, 1]$, Adam's ϵ parameter $[0.001, 2]$, clipping gradient norm $[0.02, 1]$, learning rate $[0.5, 30]$, dropout keep probability $[0.6, 1]$.

Setup B Image classification model with 3 convolutional layers, with filter size 7x7, max-pooling, stride 1, 512 chanels; then one convolutional layer with filter size 5x5 and 64 chanels; then two fully connected layers with 384 and 192 units respectively. We apply stochastic gradient descent with batch size 64, weight init scale in $[1, 30]$ for feedforward layers and $[0.005, 0.03]$ for convolutional layers, learning rate $[0.00018, 0.002]$, epoch for starting the learning rate decay $[2, 350]$, learning rate decay in $[0.5, 0.99999]$ (sampled logarithmically around 1), dropout keep probability in $[0.995, 1.]$. Trained for 30 epochs.

Setup C Language model with 2 stacked LSTM with 650 units, with the following hyper-parameters: learning rate $[0.5, 30]$, gradient clipping $[0.02, 1]$, dropout keep probability $[0.6, 1]$. Model trained for 45 epochs.

C Toy optimization problems

We conducted a set of experiments on multiple toy problems to quickly validate our assumptions. Compared methods are one-shot optimization algorithms based on the following samplings: Random, LHS, Sobol, Hammersley, Halton, S-SH. We loop over dimensions 2, 4, 8, and 16; we check three objective functions (l2-norm $f(x) = ||x - x^*||$, illcond $f(x) = \sum_{i=1}^{d}(d-i)^3(x_i - x_i^*)^2$, reverseIllcond $f(x) = \sum_{i=1}^{d}(1+i)^3(x_i - x_i^*)^2$). The budget is $n = 37$ in all cases. We used antithetic variables, thanks to mirroring w.r.t the 3 first axes (hence 8 symmetries). Each method is tested with and without this 3D mirroring. 3D mirroring deals conveniently with multiples of 8; additional points are generated in a pure random manner. x^* is randomly drawn uniformly in the domain. Each of these 12 experiments is reproduced 1221 times.

In each of these 12 cases (4 different dimensions $\times$ 3 different test functions), on average over the 1221 runs, Sobol and Scrambled-Hammersley performed better than Random in terms of simple

regret. This validates, on these artificial problems, both Sobol and Scrambled-Hammersley, in terms of one-shot optimization and face to random search, with **p-value 0.0002**.

Unsurprisingly, for illcond, Halton and Hammersley outperform random, whereas it is the opposite for reverseIllcond at least in dimension 8 and 16, i.e. it matters to have the most important variables first for these sequences. Scrambled versions (both Halton and Hammersley) resist much better and still outperform random - this validates scrambling.

Scrambled Hammersley performs best 6 times, scrambled Halton and Hammersley 2 times each, Sobol and Halton once each; none of the 3d mirrored tools ever performed best. This invalidates mirroring, and confirm the good behavior of scrambled Hammersley.

These intensive toy experiments are in accordance with the known results and our theoretical analysis (Section 3). Based on the results, the randomly shifted Scrambled-Hammersley is our main LDS for the evaluation on real Deep Learning tasks.

D Artificial datasets

Our artificial datasets are indexed with one string (AN or ANBN or .N or others, detailed below) and 4 numbers. For the artificial dataset C, the 4 numbers are the vocabulary size (the number of letters), maximum word size (n or N), vocabulary growth and depth - unless specified otherwise, vocabulary growth is 10 and depth is 3. There are also four parameters for ANBN, AN, .N and anbn; but the two last parameters are different: they are "fixed size" (True for fixed size, False otherwise) and "size gaps" (impact of size gaps equal to True detailed below). For example, artificial(anbn,26,10,0,1) means that n is randomly drawn between 1 and 10, and that there are size gaps; whereas artificial(anbn,26,10,1,0) means that n is fixed equal to 10 and there is no size gap.

- AN is a artificial dataset with, as word, a single letter randomly drawn (once for each sequence) and repeated a fixed number of times (same number of times for different sequences, but different letter). For examples, the first sequence might be "qqqqqq qqqqqq qqqqqq" and the second one "pppppp pppppp pppppp".
- The "ANBN" testbed is made of words built by concatenating N copies of a given randomly drawn letter, followed by N copies of another randomly drawn letter. The words are repeated until the end of the sequence. For different sequences, we have different letters, but the same number N.
- In the "ABNA" dataset, a word is one letter (randomly drawn, termed A), then N copies of another letter (randomly drawn and termed B), and then the first letter again.
- We also use the ".N" testbed, where the "language" to be modelled is made of sequences, each of them containing only one word (made of N randomly independently drawn letters) repeated until the end of the sequence. The first sequence might be "bridereix bridereix bridereix" and the second sequence "dunlepale dunlepale dunlepale".
- We also have "anbn" as a testbed: compared to ANBN, the number of letters vary for each word in a same sequence, and the letters vary even inside a sequence. The first sequence might be "aaabbb ddddcccc db".
- Finally, we use the "C" testbed, in which there are typically 26 letters (i.e. the vocabulary size is 26 unless stated otherwise), words are randomly drawn combinations of letters and there exists $V \times 26$ words of e.g. 10 (word size) letters; and there exists $V^2 \times 26$ groups (we might say "sentences") of 10 words, where V is the "vocabulary growth"; there are 3 levels (letters, words, groups of words) when the depth is 3. For example $artificial(C, 26, 10, 7, 3)$ contains 26 letters, 26×7 words of length 10, and 26×7^2 groups of 10 words.

For each sequence of these artificial datasets, the word size N is randomly chosen uniformly between 1 and the maximum word size, except when fixed size is 1 - in which case the word size is always the maximum word size.

If size gaps is equal to True, then the test sizes used in test and validation are guaranteed to not have been seen in training; 4 word sizes are randomly chosen for valid and for test, and the other 16 are used in training. There are 10 000 training sequences, 1000 validation sequences, 1000 test sequences.

	Untuned	Half-tuned	Tuned
Learning rate	$[0.001, 3000]$	$[0.1, 200]$	$[0.5, 30]$
Max grad. norm	$[0.002, 100]$	$[0.006, 30]$	$[0.02, 1.0]$
Weight init scale	$[0.02, 1]$	$[0.02, 1]$	$[0.02, 1]$

Table 2: Range of hyper-parameters.

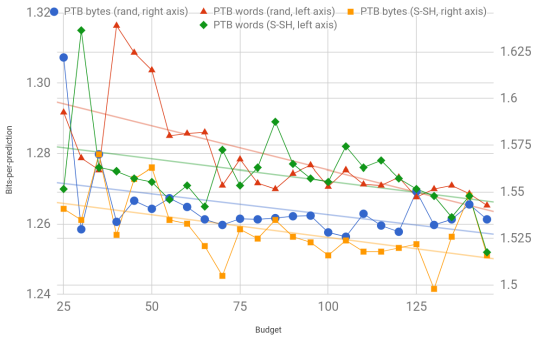


Figure 5: Language modeling with moderate networks, (x) in Table 3.

Each sequence is made of 50 words, except for C for which a sequence is one group of the maximum level.

In all artificial sequences, letters which are not predictable given the type of sequence have a weight 0 (i.e. are not taken into account when computing the loss). In all cases, the loss functions are in bits-per-byte.

E Additional experiments

E.1 Robustness speed-up

In addition to the speed-up defined in Section 4.1, we here define a **robustness speed-up**.

Speed-up was defined as such because $(1+s)/(2+s)$ is the frequency at which a random search with budget $b \times (1+s)$ wins against a random search with budget b. We can define the speed-up similarly when S competes with several (say K) instances of random search with the same budget b: $p = (1+s)/(K+1+s)$. Note that when $K > 1$, we could define another type of speed-up: instead of the frequency at which S is *better than all* K independent copies of random search, we can consider the frequency at which it performs *worse than all* these K instances. We call it **robustness speed-up**; this speed-up is positive when the algorithm is robust and avoids bad cases. When ambiguous, we refer to the initial one as **optimistic speed-up**, otherwise simply **speed-up**.

E.2 Average speed-ups over many budgets and testbeds: LDS outperforms random in a stable manner

We consider tuning three commonly used hyper-parameters ranked in the following way: learning rate, weight init scale, max gradient norm in three different settings – untuned, half-tuned and tuned – which corresponds to some degree of prior on the expected range of those hyper parameters (Table 2).

S-SH performs clearly better than random (Table 3): all statistically significant results are in favor of S-SH compared to random. In cases in which the speed-up was negative, this was not statistically significant; and we checked, in all cases in which the speed-up was less than +200%, that the robustness speed-up is still positive.

Setting	Budgets	Optimistic speed-up	Robustness speed-up
Untuned* (3 HP, 7 epochs)	5:12	+350%^{o}	
Untuned* (3 HP, 7 epochs)	5:15, 17, 19	+237%^{o}	
Untuned* (3 HP, 7 epochs)	25:5:145	+12.8 %^{o} (x)	+26%
Half-tuned* (3 HP, 7 epochs)	40:10:120 and 140:20:200	+50.0 %^{o}	+25%

Setting	Budgets	Optimistic speed-up	Robustness speed-up
Tuned (3 HP, 7 epochs)*	5:47	-51%	+28%
Tuned* (3 HP, 7 epochs)	48:73	+87%	+50%
Tuned* (3 HP, 4 epochs)	20:5:30	+160%^{o}	+160%
Leaky** (5 HP, 30 epochs)	10:5:100	+62 %	+40%
PTB Bytes/Words*** (7 HP, 17 epochs) (36 xps)	budget 20 budget 30	+57% +0%	+57% +0%

Table 3: Performance of Shifted Scrambled-Hammersley versus random search in various setups. *: PTB bytes and words, Toy sequences (see Section D) C, AN, anbn and ABNA, no size gap, no fixed size, 3 instances of random vs S-SH, 200 units, 11 epochs. **: leaky setup as defined in Section E.3, toy sequences .N, AN, anbn with 26 letters and max word size 5 (resp. 10), 2 Lstm, 100 units, with size gaps, variable word size, 3 instances of random vs S-SH. ***: learning rate in $[0.5, 30]$, max gradient norm $[2e-3, 1]$, Adam's epsilon $[0.001, 2]$, weight init scale $[2e-3, 10]$, number of epochs before learning rate decay $[5, 15]$, dropout keep probability $[0.2, 1]$, 17 epochs, 2 LSTM with 650 units, S-SH vs one instance of random. ^{o} denote p-values < 0.05; there are cases in which the speed-up was negative but these cases are not statistically significant; and when the speed-up was less than $+200\%$ we also checked the robustness speed-up and it was positive in all cases - this illustrates the robustness property emphasized in Section 3. (x): Fig. 5 suggests a speed-up $> 100\%$ for budget $\simeq 20-70$ decreasing to zero for large budget.

E.3 Results on toy language modeling problems with additional model hyper-parameters.

We use a modified version of the LSTM cell using 100 leaky state units; each state unit includes 2 additional parameters, sampled at initialization time according to a gaussian distribution, whose mean and variance are the 2 additional HPs of the model. Whereas we use, in all experiments (including the present ones), learning rate and gradient clipping norm as the two first parameters, a posteriori analysis shows that the important parameters are these two specific parameters and therefore our prior (on the ranking of HP) was wrong - this is a common scenario where we do not have any prior on the relative importance of the hyper-parameters. The model was trained on 6 of the artificial datasets described in Appendix D, namely toy(.N26,5,0,1), toy(.N,26,10,0,1), toy(AN,26,5,0,1), toy(AN,26,10,0,1), toy(anbn,26,5,0,1), toy(anbn,26,10,0,1). We use learning rate $\in [5, 100]$, weight init scale $\in [0.05, 1]$, max gradient norm $\in [0.05, 1]$, and the two new parameters (mean/std) are respectively $\in [-9, 9]$ and $\in [0.01, 10]$. We check budget $10, 15, 20, 25, \ldots, 100$. Fig. 6 shows the rank of S-SH, among S-SH and 3 instances of random search; this rank is between 1 and 4; after normalization to $[0, 1]$ we get 0.404$\pm$ 0.097, 0.456 $\pm$ 0.084, 0.281$\pm$ 0.084, 0.491$\pm$0.080, 0.544$\pm$ 0.092, 0.316$\pm$ 0.081 on these 6 tasks respectively; all but one are in favor of S-SH (i.e. rank < 0.5), 2 are statistically significant, and when agregating over all these runs we get an average rank 0.415 at ≥ 3 standard deviations from .5, hence clearly significant.

E.4 Other sampling algorithms: S-SH performs best

E.4.1 Validating scrambling/Hammersley on PTB: not all LDS are equivalent

Fig. 7 compares pure random, Halton, and scrambled Halton on PTB-Bytes and PTB-Words. The setting is as follows: 7 HPs (dropout keep probability in $[0.2, 1]$, learning rate in $[0.05, 300]$, gradient clipping norm in $[0.002, 1]$, Adam's epsilon parameter in $[0.001, 2]$, weight initialization scale in $[0.002, 10]$, epoch index for starting the exponential decay of learning rate in $[5, 15]$, learning rate decay in $[0.1, 1]$); 17 training epochs, 2 stacked LSTM; we perform experiments for a number of units ranging from 12 to 29. The budget for the randomly drawn HPs is 20. Overall, there are 36 comparisons (18 on PTB-Bytes, corresponding to 18 different numbers of units, and 18 on PTB-Words);

- Scrambled-Hammersley outperforms Halton 25 times (p-value **0.036**);
- Scrambled-Hammersley outperforms random 22 times (p-value 0.12);
- Halton outperforms random 19 times (p-value 0.43).

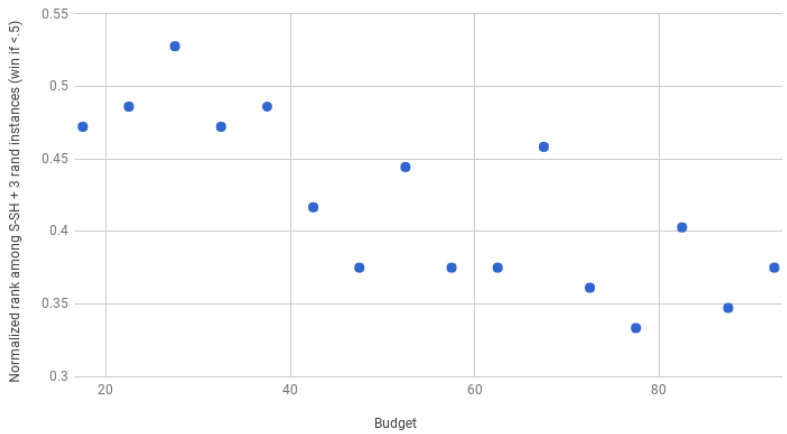


Figure 6: Rank of S-SH among S-SH and 3 random instances, normalized to $[0, 1]$, on toy language modelling testbeds and with leaky LSTM; < 0.5 in most cases i.e. success for S-SH. Points are moving averages over 4 independent successive budgets hence points 4 values apart are independent. This corresponds to an overall speed-up of +62%, i.e. we get with 50 runs the same best performance as random with 81 runs.

The experiment was reproduced a second time, with budget $= 30$ random tries for the HPs. Results are presented in Fig. 8 - no statistically significant improvement.

E.5 Robustness vs peak performance

We have seen in Section E.2 that S-SH was always beneficial in terms of robustness; and most often in terms of peak performance (optimistic speed-up). We develop this point by performing additional experiments. We prefer challenges, so we focus on the so-called tuned-setting in which our results in Section E.2 were least positive for S-SH. We performed experiments with many different values of the budget (number of HPs vectors sampled) and tested if S-SH performs better than random in both low budget regime (≤ 48) and large budget > 48). Each budget from 5 to 73 was tested. For large budget regime, S-SH performed excellently, both in terms of best performance (speed-up $+87\%$, p-value 0.09) and worst performance (robustness speed-up $+50\%$, p-value 0.18). For budget ≤ 48, random was performing better than S-SH for the optimistic speed-up (-51%, p-value 0.18); but worse than S-SH for the robustness speed-up ($+28\%$, p-value 0.09). Importantly, this section focuses on the only setting in which S-SH was not beneficial in our diverse experiments (Section E.2); we have, on purpose, developped precisely the case in which things were going wrong in order to clarify to which extent replacing random by S-SH can be detrimental.

E.6 Performance as a function of the number of epochs

We consider the untuned setting, on the same 6 problems as (*) in Fig. 3. We have a number of epochs ranging from 7 to 36, and we consider moving averages of the ranks over the 6 datasets and 4 successive numbers of epochs. Fig. 9 presents the impact of the number of epochs; S-SH performs best overall.

E.7 Results as an initialization for GP

In this section we leave the one-shot setting; we consider several batches, and a Gaussian processes (GP) based Bayesian optimization. Results in [24] already advocated LDS for the initialization of GP, pointing out that this outperforms Latin Hypercube Sampling; in the present section, we confirm those results in the case of deep learning and show that we also outperform typical pessimistic fantasizing as used in batch entropy search. The pessimistic fantasizing method for GP is based on (i) for the first point of a batch, use optimistic estimates on the value as a criterion for selecting candidates (ii) for the next points in the same batch, fantasizing the values of the previous points in the current batch in a pessimistic manner; and apply the same criterion as above, assuming these pessimistic values for previous values of the batch. For the first batch, this leads to regular patterns as

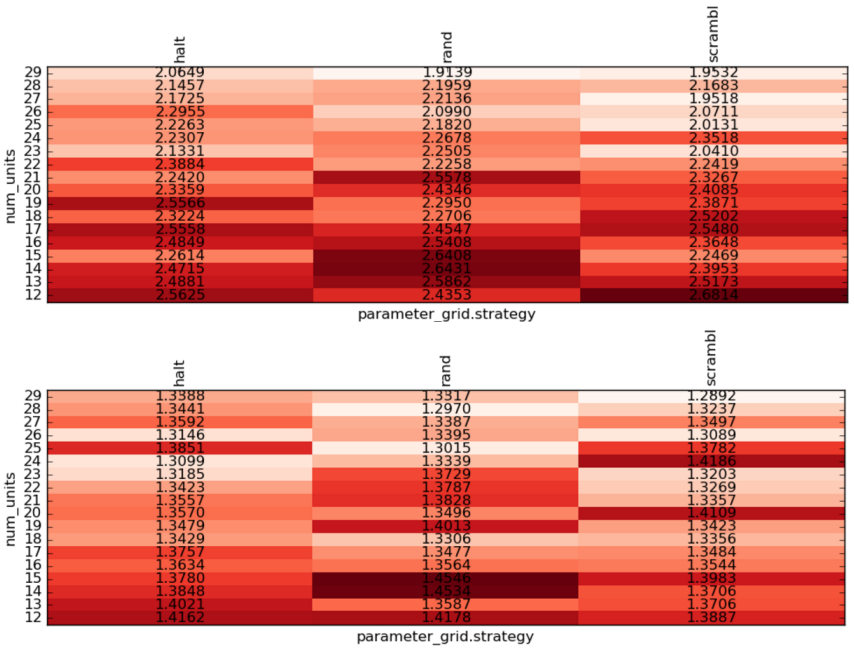


Figure 7: Bits-per-byte loss for various settings. Comparing a sophisticated Halton to a naive Halton and to random with 17 learning epochs. Halton refers to the original Halton sequence. Scramb. refers to Hammersley with scrambling - which has the best known discrepancy bounds (more precisely the best proved bounds are obtained by Atanassov's scrambling; for the random scrambling we use, as previously mentioned, it is conjectured that the performance is the same). Top: PTB-Bytes (bits per byte). Bottom: PTB-Words (bits per word). Scrambled-Hammersley outperforms the simple Halton and pure random (see text).

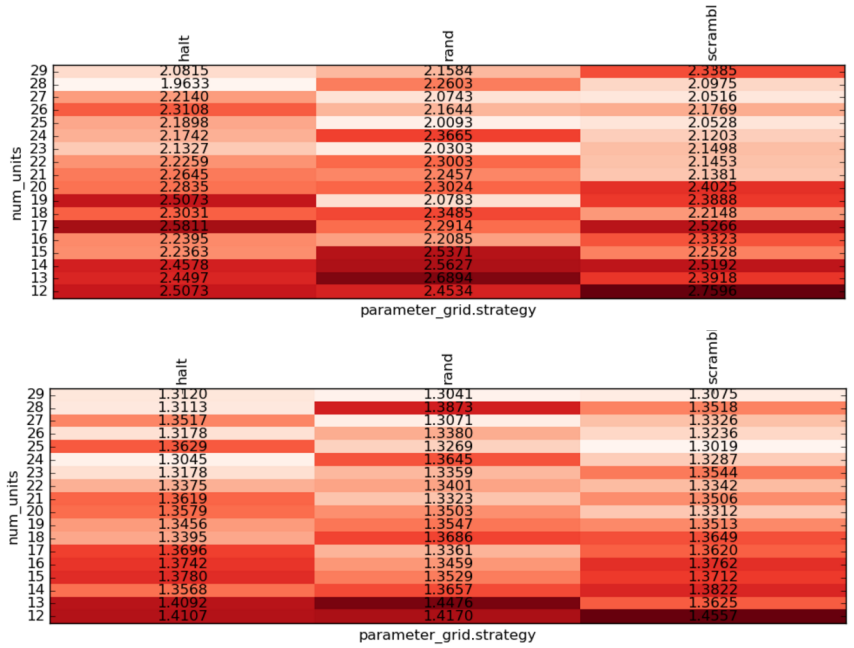


Figure 8: Same as Fig. 7 but with budget 30 instead of 20, still 17 learning epochs. Top: PTB-Bytes. Bottom: PTB-Words. No statistically significant difference neither for bytes (9 wins for 18 test cases, 9/18 and 10/18 for S-SH vs random, for Halton vs rand, and for S-SH vs Halton respectively) or words (9/18, 10/18 and 9/18 respectively).

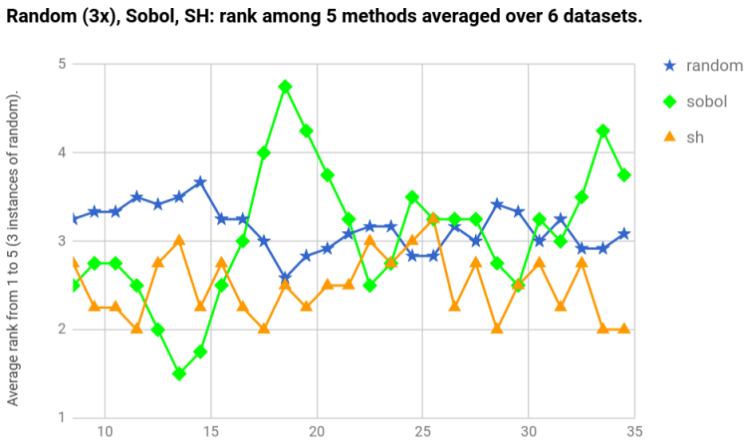


Figure 9: Untuned setting; rank of each method among 5 methods (3 instances of random, plus Sobol and Scrambled-Hammersley), averaged over 6 datasets and 4 successive numbers of epochs (i.e. each point is the average of 24 results and results 3 points apart are independent). This figure corresponds to a budget of 20 vectors of HPs; Scrambled Hammersley outperforms random; Sobol did not provide convincing results. We also tested a budget 10 and did not get any clear difference among methods.

in grids. We keep the same method for further iterations but replace this first batch by (a) random sampling (b) low-discrepancy sampling. Results are presented in Fig. 10. We randomly translate the optimum in the domain but do not rotate the space of functions - which would destroy the concept of critical variables. We consider artificial test functions, namely Sphere, Ellipsoid, Branin, Rastrigin, Six-Hump, Styblinski, Beale. We work in dimension 12, with 64 vectors of HPs per batch. We average results over 750, 100 and 50 runs for 1, 3 and 5 batches respectively.

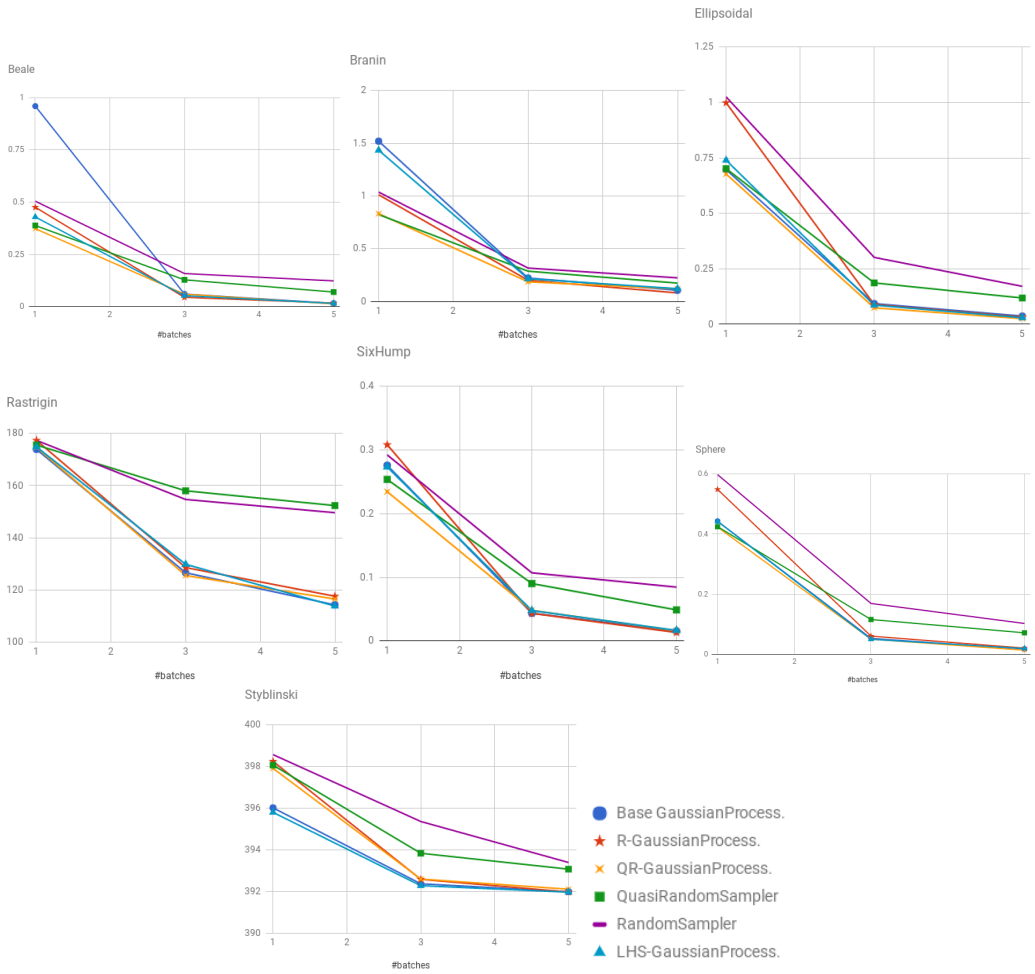


Figure 10: Experiments comparing 1, 3 and 5 batches of 64 points for Bayesian optimization (four flavors: base (i.e. pessimistic-fantasizing of unseen values as in Entropy search[8]), random, LDS (here Scrambled Halton)) and randomly shifted Scrambled-Halton and random on seven test functions. We note that LDS-based Bayesian optimization usually outperformed all other methods.

Dimension 12, budget 4

1 batch	Rastrigin	Sphere	Ellipsoidal	Styblinski	Beale	Branin	SixHump
Pessimistic F.	0.98	1.07	0.93	1.00	1.14	1.02	1.01
Random init	1.03	1.06	1.11	1.00	2.23	0.65	1.35
LDSBO	0.99	1.10	1.01	1.02	2.56	1.79	1.71
3 batches	Rastrigin	Sphere	Ellipsoidal	Styblinski	Beale	Branin	SixHump
Pessimistic F.	0.97	1.36	1.21	1.00	1.29	1.05	0.93
Random init	1.03	1.47	1.24	1.01	2.09	0.97	1.23
LDSBO	1.01	1.31	0.95	1.00	1.48	1.28	1.04
5 batches	Rastrigin	Sphere	Ellipsoidal	Styblinski	Beale	Branin	SixHump
Pessimistic F.	1.04	0.79	1.08	1.00	0.94	1.01	1.06
Random init	1.02	0.70	1.15	1.00	1.83	0.92	1.11
LDSBO	1.01	1.03	1.03	1.00	0.72	1.67	0.99

Dimension 12, budget 16

1 batch	Rastrigin	Sphere	Ellipsoidal	Styblinski	Beale	Branin	SixHump
Pessimistic F.	0.99	0.99	0.89	1.00	0.97	1.02	0.98
Random init	1.01	1.09	1.07	0.99	1.21	0.96	1.13
LDSBO	0.99	1.13	1.11	0.99	1.44	1.20	0.91
3 batches	Rastrigin	Sphere	Ellipsoidal	Styblinski	Beale	Branin	SixHump
Pessimistic F.	0.97	0.90	1.04	1.01	0.74	1.09	1.00
Random init	0.96	1.32	1.01	0.99	1.11	1.00	1.18
LDSBO	0.99	1.16	1.05	1.01	1.57	0.94	0.98
5 batches	Rastrigin	Sphere	Ellipsoidal	Styblinski	Beale	Branin	SixHump
Pessimistic F.	1.00	0.90	1.15	1.00	0.90	1.07	0.78
Random init	0.99	1.10	0.86	0.98	1.73	1.11	1.39
LDSBO	1.01	0.79	1.06	0.99	1.65	1.30	1.00

Table 4: Experiments with lower budget, in which case LHS (which has excellent discrepancy for low budget) performs well. Ratio "loss of a method / loss of LHS-BO method" (i.e. > 1 means LHS-BO wins) in dimension 12. We see that LHS-BO outperforms LDS-BO for the small budget 4 (i.e. most values are > 1). For budget 16, pessimistic fantasizing can compete. BO with randomized initialization is always weak compared to LHS-BO.

Train longer, generalize better: closing the generalization gap in large batch training of neural networks

Elad Hoffer[*] **Itay Hubara**[*] **Daniel Soudry**
Technion - Israel Institute of Technology, Haifa, Israel
{elad.hoffer, itayhubara, daniel.soudry}@gmail.com

Abstract

Background: Deep learning models are typically trained using stochastic gradient descent or one of its variants. These methods update the weights using their gradient, estimated from a small fraction of the training data. It has been observed that when using large batch sizes there is a persistent degradation in generalization performance - known as the "generalization gap" phenomenon. Identifying the origin of this gap and closing it had remained an open problem.

Contributions: We examine the initial high learning rate training phase. We find that the weight distance from its initialization grows logarithmically with the number of weight updates. We therefore propose a "random walk on a random landscape" statistical model which is known to exhibit similar "ultra-slow" diffusion behavior. Following this hypothesis we conducted experiments to show empirically that the "generalization gap" stems from the relatively small number of updates rather than the batch size, and can be completely eliminated by adapting the training regime used. We further investigate different techniques to train models in the large-batch regime and present a novel algorithm named "Ghost Batch Normalization" which enables significant decrease in the generalization gap without increasing the number of updates. To validate our findings we conduct several additional experiments on MNIST, CIFAR-10, CIFAR-100 and ImageNet. Finally, we reassess common practices and beliefs concerning training of deep models and suggest they may not be optimal to achieve good generalization.

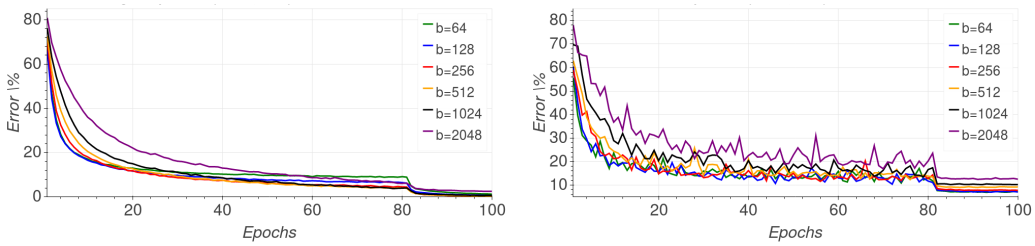
1 Introduction

For quite a few years, deep neural networks (DNNs) have persistently enabled significant improvements in many application domains, such as object recognition from images (He et al., 2016); speech recognition (Amodei et al., 2015); natural language processing (Luong et al., 2015) and computer games control using reinforcement learning (Silver et al., 2016; Mnih et al., 2015).

The optimization method of choice for training highly complex and non-convex DNNs, is typically stochastic gradient decent (SGD) or some variant of it. Since SGD, at best, finds a local minimum of the non-convex objective function, substantial research efforts are invested to explain DNNs ground breaking results. It has been argued that saddle-points can be avoided (Ge et al., 2015) and that "bad" local minima in the training error vanish exponentially (Dauphin et al., 2014; Choromanska et al., 2015; Soudry & Hoffer, 2017). However, it is still unclear why these complex models tend to generalize well to unseen data despite being heavily over-parameterized (Zhang et al., 2017).

A specific aspect of generalization has recently attracted much interest. Keskar et al. (2017) focused on a long observed phenomenon (LeCun et al., 1998a) – that when a large batch size is used while training DNNs, the trained models appear to generalize less well. This remained true even when the

[*]Equal contribution



(a) Training error

(b) Validation error

Figure 1: Impact of batch size on classification error

models were trained "without any budget or limits, until the loss function ceased to improve" (Keskar et al., 2017). This decrease in performance has been named the "generalization gap".

Understanding the origin of the generalization gap, and moreover, finding ways to decrease it, may have a significant practical importance. Training with large batch size immediately increases parallelization, thus has the potential to decrease learning time. Many efforts have been made to parallelize SGD for Deep Learning (Dean et al., 2012; Das et al., 2016; Zhang et al., 2015), yet the speed-ups and scale-out are still limited by the batch size.

In this study we suggest a first attempt to tackle this issue.
First,

- We propose that the initial learning phase can be described using a high-dimensional "random walk on a random potential" process, with an "ultra-slow" logarithmic increase in the distance of the weights from their initialization, as we observe empirically.

Inspired by this hypothesis, we find that

- By simply adjusting the learning rate and batch normalization the generalization gap can be significantly decreased (for example, from 5% to $1\% - 2\%$).
- In contrast to common practices (Montavon et al., 2012) and theoretical recommendations (Hardt et al., 2016), generalization keeps improving for a long time at the initial high learning rate, even without any observable changes in training or validation errors. However, this improvement seems to be related to the distance of the weights from their initialization.
- There is no inherent "generalization gap": large-batch training can generalize as well as small batch training by adapting the number of iterations.

2 Training with a large batch

Training method. A common practice of training deep neural networks is to follow an optimization "regime" in which the objective is minimized using gradient steps with a fixed learning rate and a momentum term (Sutskever et al., 2013). The learning rate is annealed over time, usually with an exponential decrease every few epochs of training data. An alternative to this regime is to use an adaptive per-parameter learning method such as Adam (Kingma & Ba, 2014), Rmsprop (Dauphin et al.) or Adagrad (Duchi et al., 2011). These methods are known to benefit the convergence rate of SGD based optimization. Yet, many current studies still use simple variants of SGD (Ruder, 2016) for all or part of the optimization process (Wu et al., 2016), due to the tendency of these methods to converge to a lower test error and better generalization.

Thus, we focused on momentum SGD, with a fixed learning rate that decreases exponentially every few epochs, similarly to the regime employed by He et al. (2016). The convergence of SGD is also known to be affected by the batch size (Li et al., 2014), but in this work we will focus on generalization. Most of our results were conducted on the Resnet44 topology, introduced by He et al. (2016). We strengthen our findings with additional empirical results in section 6.

Empirical observations of previous work. Previous work by Keskar et al. (2017) studied the performance and properties of models which were trained with relatively large batches and reported the following observations:

- Training models with large batch size increase the generalization error (see Figure 1).
- This "generalization gap" seemed to remain even when the models were trained without limits, until the loss function ceased to improve.
- Low generalization was correlated with "sharp" minima[2] (strong positive curvature), while good generalization was correlated with "flat" minima (weak positive curvature).
- Small-batch regimes were briefly noted to produce weights that are farther away from the initial point, in comparison with the weights produced in a large-batch regime.

Their hypothesis was that a large estimation noise (originated by the use of mini-batch rather than full batch) in small mini-batches encourages the weights to exit out of the basins of attraction of sharp minima, and towards flatter minima which have better generalization.In the next section we provide an analysis that suggest a somewhat different explanation.

3 Theoretical analysis

Notation. In this paper we examine Stochastic Gradient Descent (SGD) based training of a Deep Neural Network (DNN). The DNN is trained on a finite training set of N samples. We define $\mathbf{w}$ as the vector of the neural network parameters, and $L_n(\mathbf{w})$ as loss function on sample n. We find $\mathbf{w}$ by minimizing the training loss.

$$L(\mathbf{w}) \triangleq \frac{1}{N}\sum_{n=1}^{N} L_n(\mathbf{w}) \ ,$$

using SGD. Minimizing $L(\mathbf{w})$ requires an estimate of the gradient of the negative loss.

$$\mathbf{g} \triangleq \frac{1}{N}\sum_{n=1}^{N} \mathbf{g}_n \triangleq -\frac{1}{N}\sum_{n=1}^{N} \nabla L_n(\mathbf{w})$$

where $\mathbf{g}$ is the true gradient, and $\mathbf{g}_n$ is the per-sample gradient. During training we increment the parameter vector $\mathbf{w}$ using only the mean gradient $\hat{\mathbf{g}}$ computed on some mini-batch B – a set of M randomly selected sample indices.

$$\hat{\mathbf{g}} \triangleq \frac{1}{M}\sum_{n\in B} \mathbf{g}_n \, .$$

In order to gain a better insight into the optimization process and the empirical results, we first examine simple SGD training, in which the weights at update step t are incremented according to the mini-batch gradient $\Delta\mathbf{w}_t = \eta\hat{\mathbf{g}}_t$. With respect to the randomness of SGD,

$$\mathbb{E}\hat{\mathbf{g}}_t = \mathbf{g} = -\nabla L(\mathbf{w}_t) \, ,$$

and the increments are uncorrelated between different mini-batches[3]. For physical intuition, one can think of the weight vector $\mathbf{w}_t$ as a particle performing a random walk on the loss ("potential") landscape $L(\mathbf{w}_t)$. Thus, for example, adding momentum term to the increment is similar to adding inertia to the particle.

Motivation. In complex systems (such as DNNs) where we do not know the exact shape of the loss, statistical physics models commonly assume a simpler description of the potential as a random process. For example, Dauphin et al. (2014) explained the observation that local minima tend to have

[2]It was later pointed out (Dinh et al., 2017) that certain "degenerate" directions, in which the parameters can be changed without affecting the loss, must be excluded from this explanation. For example, for any $c > 0$ and any neuron, we can multiply all input weights by c and divide the output weights by c: this does not affect the loss, but can generate arbitrarily strong positive curvature.

[3]Either exactly (with replacement) or approximately (without replacement): see appendix section A.

low error using an analogy between $L(\mathbf{w})$, the DNN loss surface, and the high-dimensional Gaussian random field analyzed in Bray & Dean (2007), which has zero mean and auto-covariance

$$\mathbb{E}\left(L\left(\mathbf{w}_1\right)L\left(\mathbf{w}_2\right)\right) = f\left(\left\|\mathbf{w}_1 - \mathbf{w}_2\right\|^2\right) \tag{1}$$

for some function f, where the expectation now is over the randomness of the loss. This analogy resulted with the hypothesis that in DNNs, local minima with high loss are indeed exponentially vanishing, as in Bray & Dean (2007). Only recently, similar results are starting to be proved for realistic neural network models (Soudry & Hoffer, 2017). Thus, a similar statistical model of the loss might also give useful insights for our empirical observations.

Model: Random walk on a random potential. Fortunately, the high dimensional case of a particle doing a "random walk on a random potential" was extensively investigated already decades ago (Bouchaud & Georges, 1990). The main result of that investigation was that the asymptotic behavior of the auto-covariance of a random potential[4],

$$\mathbb{E}\left(L\left(\mathbf{w}_1\right)L\left(\mathbf{w}_2\right)\right) \sim \left\|\mathbf{w}_1 - \mathbf{w}_2\right\|^{\alpha} \ , \ \alpha > 0 \tag{2}$$

in a certain range, determines the asymptotic behavior of the random walker in that range:

$$\mathbb{E}\left\|\mathbf{w}_t - \mathbf{w}_0\right\|^2 \sim \left(\log t\right)^{\frac{4}{\alpha}} \ . \tag{3}$$

This is called an "ultra-slow diffusion" in which, typically $\left\|\mathbf{w}_t - \mathbf{w}_0\right\| \sim \left(\log t\right)^{2/\alpha}$, in contrast to standard diffusion (on a flat potential), in which we have $\left\|\mathbf{w}_t - \mathbf{w}_0\right\| \sim \sqrt{t}$. The informal reason for this behavior (for any $\alpha > 0$), is that for a particle to move a distance d, it has to pass potential barriers of height $\sim d^{\alpha/2}$, from eq. (2). Then, to climb (or go around) each barrier takes exponentially long time in the height of the barrier: $t \sim \exp(d^{\alpha/2})$. Inverting this relation, we get eq. $d \sim (\log(t))^{2/\alpha}$. In the high-dimensional case, this type of behavior was first shown numerically and explained heuristically by Marinari et al. (1983), then rigorously proven for the case of a discrete lattice by Durrett (1986), and explained in the continuous case by Bouchaud & Comtet (1987).

3.1 Comparison with empirical results and implications

To examine this prediction of ultra slow diffusion and find the value of α, in Figure 2a, we examine $\left\|\mathbf{w}_t - \mathbf{w}_0\right\|$ during the initial training phase over the experiment shown in Figure 1. We found that the weight distance from initialization point increases logarithmically with the number of training iterations (weight updates), which matches our model with $\alpha = 2$:

$$\left\|\mathbf{w}_t - \mathbf{w}_0\right\| \sim \log t \, . \tag{4}$$

Interestingly, the value of $\alpha = 2$ matches the statistics of the loss estimated in appendix section B.

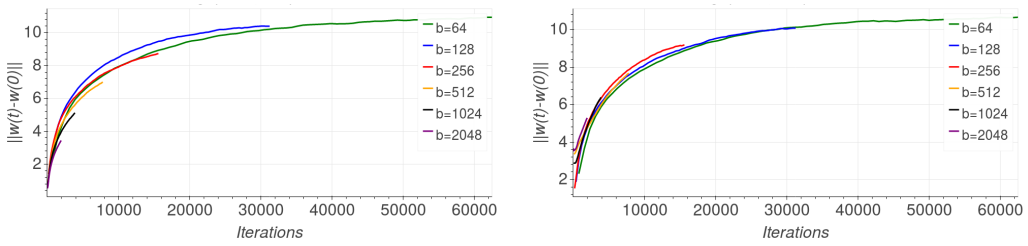
Moreover, in Figure 2a, we find that a very similar logarithmic graph is observed for all batch sizes. Yet, there are two main differences. First, each graph seems to have a somewhat different slope (i.e., it is multiplied by different positive constant), which peaks at $M = 128$ and then decreases with the mini-batch size. This indicates a somewhat different diffusion rate for different batch sizes. Second, since we trained all models for a constant number of epochs, smaller batch sizes entail more training iterations in total. Thus, there is a significant difference in the number of iterations and the corresponding weight distance reached at the end of the initial learning phase.

This leads to the following informal argument (which assumes flat minima are indeed important for generalization). During the initial training phase, to reach a minima of "width" d the weight vector $\mathbf{w}_t$ has to travel at least a distance d, and this takes a long time – about $\exp(d)$ iterations. Thus, to reach wide ("flat") minima we need to have the highest possible diffusion rates (which do not result in numerical instability) and a large number of training iterations. In the next sections we will implement these conclusions in practice.

4 Matching weight increment statistics for different mini-batch sizes

First, to correct the different diffusion rates observed for different batch sizes, we will aim to match the statistics of the weights increments to that of a small batch size.

[4]Note that this form is consistent with eq. (1), if $f(x) = x^{\alpha/2}$.



(a) Before learning rate adjustment and GBN (b) After learning rate adjustment and GBN

Figure 2: Euclidean distance of weight vector from initialization

Learning rate. Recall that in this paper we investigate SGD, possibly with momentum, where the weight updates are proportional to the estimated gradient.

$$\Delta \mathbf{w} \propto \eta \hat{\mathbf{g}} \,, \tag{5}$$

where η is the learning rate, and we ignore for now the effect of batch normalization.

In appendix section A, we show that the covariance matrix of the parameters update step $\Delta \mathbf{w}$ is,

$$\mathrm{cov}\left(\Delta \mathbf{w}, \Delta \mathbf{w}\right) \approx \frac{\eta^2}{M} \left(\frac{1}{N} \sum_{n=1}^{N} \mathbf{g}_n \mathbf{g}_n^\top \right) \tag{6}$$

in the case of uniform sampling of the mini-batch indices (with or without replacement), when $M \ll N$. Therefore, a simple way to make sure that the covariance matrix stays the same for all mini-batch sizes is to choose

$$\eta \propto \sqrt{M} \,, \tag{7}$$

i.e., we should increase the learning rate by the square root of the mini-batch size.

We note that Krizhevsky (2014) suggested a similar learning rate scaling in order to keep the variance in the gradient expectation constant, but chose to use a linear scaling heuristics as it reached better empirical result in his setting. Later on, Li (2017) suggested the same.

Naturally, such an increase in the learning rate also increases the mean steps $\mathbb{E}\left[\Delta \mathbf{w}\right]$. However, we found that this effect is negligible since $\mathbb{E}\left[\Delta \mathbf{w}\right]$ is typically orders of magnitude lower than the standard deviation.

Furthermore, we can match both the first and second order statistics by adding multiplicative noise to the gradient estimate as follows:

$$\hat{\mathbf{g}} = \frac{1}{M} \sum_{n \in B}^{N} \mathbf{g}_n z_n \,,$$

where $z_n \sim \mathcal{N}\left(1, \sigma^2\right)$ are independent random Gaussian variables for which $\sigma^2 \propto M$. This can be verified by using similar calculation as in appendix section A. This method keeps the covariance constant when we change the batch size, yet does not change the mean steps $\mathbb{E}\left[\Delta \mathbf{w}\right]$.

In both cases, for the first few iterations, we had to clip or normalize the gradients to prevent divergence. Since both methods yielded similar performance [5] (due the negligible effect of the first order statistics), we preferred to use the simpler learning rate method.

It is important to note that other types of noise (e.g., dropout (Srivastava et al., 2014), dropconnect (Wan et al., 2013), label noise (Szegedy et al., 2016)) change the structure of the covariance matrix and not just its scale, thus the second order statistics of the small batch increment cannot be accurately matched. Accordingly, we did not find that these types of noise helped to reduce the generalization gap for large batch sizes.

Lastly, note that in our discussion above (and the derivations provided in appendix section A) we assumed each per-sample gradient $\mathbf{g}_n$ does not depend on the selected mini-batch. However, this ignores the influence of batch normalization. We take this into consideration in the next subsection.

[5] a simple comparison can be seen in appendix (figure 3)

Ghost Batch Normalization. Batch Normalization (BN) (Ioffe & Szegedy, 2015), is known to accelerate the training, increase the robustness of neural network to different initialization schemes and improve generalization. Nonetheless, since BN uses the batch statistics it is bounded to depend on the choosen batch size. We study this dependency and observe that by acquiring the statistics on small virtual ("ghost") batches instead of the real large batch we can reduce the generalization error. In our experiments we found out that it is important to use the full batch statistic as suggested by (Ioffe & Szegedy, 2015) for the inference phase. Full details are given in Algorithm 1. This modification by itself reduce the generalization error substantially.

Algorithm 1: Ghost Batch Normalization (GBN), applied to activation x over a large batch B_L with virtual mini-batch B_S. Where $B_S < B_L$.

Require: Values of x over a large-batch: $B_L = \{x_{1...m}\}$ size of virtual batch $|B_S|$; Parameters to be learned: γ, β, momentum η

Training Phase:

Scatter B_L to $\{X^1, X^2, ...X^{|B_L|/|B_S|}\} = \{x_{1...|B_S|}, x_{|B_S|+1...2|B_S|}...x_{|B_L|-|B_S|...m}\}$

$\mu_B^l \leftarrow \frac{1}{|B_S|}\sum_{i=1}^{|B_S|} X_i^l \quad$ for $l = 1, 2, 3 \ldots \quad$ {calculate ghost mini-batches means}

$\sigma_B^l \leftarrow \sqrt{\frac{1}{|B_S|}\sum_{i=1}^{|B_S|}(X_i^l - \mu_B)^2 + \epsilon} \quad$ for $l = 1, 2, 3 \ldots \quad$ {calculate ghost mini-batches std}

$\mu_{run} = (1-\eta)^{|B_S|}\mu_{run} + \sum_{i=1}^{|B_L|/|B_S|}(1-\eta)^i \cdot \eta \cdot \mu_B^l$

$\sigma_{run} = (1-\eta)^{|B_S|}\sigma_{run} + \sum_{i=1}^{|B_L|/|B_S|}(1-\eta)^i \cdot \eta \cdot \sigma_B^l$

return $\gamma\frac{X^l - \mu_B^l}{\sigma_B^l} + \beta$

Test Phase:

return $\gamma\frac{X - \mu_{run}^l}{\sigma_{run}} + \beta$ {scale and shift}

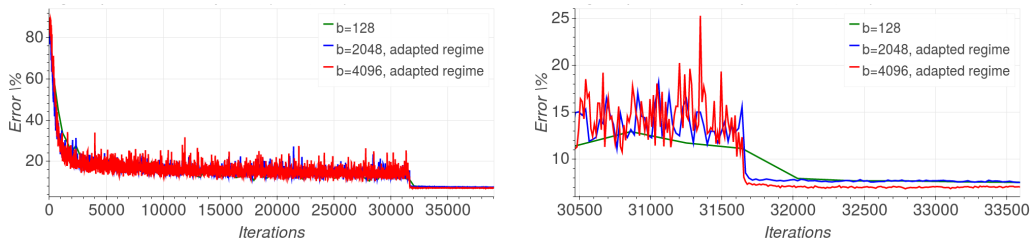
We note that in a multi-device distributed setting, some of the benefits of "Ghost BN" may already occur, since batch-normalization is often preformed on each device separately to avoid additional communication cost. Thus, each device computes the batch norm statistics using only its samples (i.e., part of the whole mini-batch). It is a known fact, yet unpublished, to the best of the authors knowledge, that this form of batch norm update helps generalization and yields better results than computing the batch-norm statistics over the entire batch. Note that GBN enables flexibility in the small (virtual) batch size which is not provided by the commercial frameworks (e.g., TensorFlow, PyTorch) in which the batch statistics is calculated on the entire, per-device, batch. Moreover, in those commercial frameworks, the running statistics are usually computed differently from "Ghost BN", by weighting each update part equally. In our experiments we found it to worsen the generalization performance.

Implementing both the learning rate and GBN adjustments seem to improve generalization performance, as we shall see in section 6. Additionally, as can be seen in Figure 6, the slopes of the logarithmic weight distance graphs seem to better matched, indicating similar diffusion rates. We also observe some constant shift, which we believe is related to the gradient clipping. Since this shift only increased the weight distances, we assume it does not harm the performance.

5 Adapting number of weight updates eliminates generalization gap

According to our conclusions in section 3, the initial high-learning rate training phase enables the model to reach farther locations in the parameter space, which may be necessary to find wider local minima and better generalization. Examining figure 2b, the next obvious step to match the graphs for different batch sizes is to increase the number of training iterations in the initial high learning rate regime. And indeed we noticed that the distance between the current weight and the initialization point can be a good measure to decide upon when to decrease the learning rate.

Note that this is different from common practices. Usually, practitioners decrease the learning rate after validation error appears to reach a plateau. This practice is due to the long-held belief that the optimization process should not be allowed to decrease the training error when validation error "flatlines", for fear of overfitting (Girosi et al., 1995). However, we observed that substantial improvement to the final accuracy can be obtained by continuing the optimization using the same



(a) Validation error

(b) Validation error - zoomed

Figure 3: Comparing generalization of large-batch regimes, adapted to match performance of small-batch training.

learning rate even if the training error decreases while the validation plateaus. Subsequent learning rate drops resulted with a sharp validation error decrease, and better generalization for the final model.

These observations led us to believe that "generalization gap" phenomenon stems from the relatively small number of updates rather than the batch size. Specifically, using the insights from Figure 2 and our model, we adapted the training regime to better suit the usage of large mini-batch. We "stretched" the time-frame of the optimization process, where each time period of e epochs in the original regime, will be transformed to $\frac{|B_L|}{|B_S|}e$ epochs according to the mini-batch size used. This modification ensures that the number of optimization steps taken is identical to those performed in the small batch regime. As can be seen in Figure 3, combining this modification with learning rate adjustment completely eliminates the generalization gap observed earlier [6].

6 Experiments

Experimental setting. We experimented with a set of popular image classification tasks:

- MNIST (LeCun et al., 1998b) - Consists of a training set of $60K$ and a test set of $10K$ 28×28 gray-scale images representing digits ranging from 0 to 9.
- CIFAR-10 and CIFAR-100 (Krizhevsky, 2009) - Each consists of a training set of size 50K and a test set of size $10K$. Instance are 32×32 color images representing 10 or 100 classes.
- ImageNet classification task Deng et al. (2009) - Consists of a training set of size 1.2M samples and test set of size 50K. Each instance is labeled with one of 1000 categories.

To validate our findings, we used a representative choice of neural network models. We used the fully-connected model, F1, as well as shallow convolutional models C1 and C3 suggested by Keskar et al. (2017). As a demonstration of more current architectures, we used the models: VGG (Simonyan, 2014) and Resnet44 (He et al., 2016) for CIFAR10 dataset, Wide-Resnet16-4 (Zagoruyko, 2016) for CIFAR100 dataset and Alexnet (Krizhevsky, 2014) for ImageNet dataset.
In each of the experiments, we used the training regime suggested by the original work, together with a momentum SGD optimizer. We use a batch of 4096 samples as "large batch" (LB) and a small batch (SB) of either 128 (F1,C1,VGG,Resnet44,C3,Alexnet) or 256 (WResnet). We compare the original training baseline for small and large batch, as well as the following methods[7]:

- Learning rate tuning (LB+LR): Using a large batch, while adapting the learning rate to be larger so that $\eta_L = \sqrt{\frac{|B_L|}{|B_S|}}\eta_S$ where η_S is the original learning rate used for small batch, η_L is the adapted learning rate and $|B_L|, |B_S|$ are the large and small batch sizes, respectively.
- Ghost batch norm (LB+LR+GBN): Additionally using the "Ghost batch normalization" method in our training procedure. The "ghost batch size" used is 128.
- Regime adaptation: Using the tuned learning rate as well as ghost batch-norm, but with an adapted training regime. The training regime is modified to have the same number of

[6] Additional graphs, including comparison to non-adapted regime, are available in appendix (figure 2).
[7] Code is available at `https://github.com/eladhoffer/bigBatch`.

iterations for each batch size used - effectively multiplying the number of epochs by the relative size of the large batch.

Results. Following our experiments, we can establish an empirical basis to our claims. Observing the final validation accuracy displayed in Table 1, we can see that in accordance with previous works the move from a small-batch (SB) to a large-batch (LB) indeed incurs a substantial generalization gap. However, modifying the learning-rate used for large-batch (+LR) causes much of this gap to diminish, following with an additional improvement by using the Ghost-BN method (+GBN). Finally, we can see that the generalization gap completely disappears when the training regime is adapted (+RA), yielding validation accuracy that is good-as or better than the one obtained using a small batch.
We additionally display results obtained on the more challenging ImageNet dataset in Table 2 which shows similar impact for our methods.

Table 1: Validation accuracy results, SB/LB represent small and large batch respectively. GBN stands for Ghost-BN, and RA stands for regime adaptation

Network	Dataset	SB	LB	+LR	+GBN	+RA
F1 (Keskar et al., 2017)	MNIST	98.27%	97.05%	97.55%	97.60%	98.53%
C1 (Keskar et al., 2017)	Cifar10	87.80%	83.95%	86.15%	86.4%	88.20%
Resnet44 (He et al., 2016)	Cifar10	92.83%	86.10%	89.30%	90.50%	93.07%
VGG (Simonyan, 2014)	Cifar10	92.30%	84.1%	88.6%	91.50%	93.03%
C3 (Keskar et al., 2017)	Cifar100	61.25%	51.50%	57.38%	57.5%	63.20%
WResnet16-4 (Zagoruyko, 2016)	Cifar100	73.70%	68.15%	69.05%	71.20%	73.57%

Table 2: ImageNet top-1 results using Alexnet topology (Krizhevsky, 2014), notation as in Table 1.

Network	LB size	Dataset	SB	LB[8]	+LR[8]	+GBN	+RA
Alexnet	4096	ImageNet	57.10%	41.23%	53.25%	54.92%	59.5%
Alexnet	8192	ImageNet	57.10%	41.23%	53.25%	53.93%	59.5%

7 Discussion

There are two important issues regarding the use of large batch sizes. First, why do we get worse generalization with a larger batch, and how do we avoid this behaviour? Second, can we decrease the training wall clock time by using a larger batch (exploiting parallelization), while retaining the same generalization performance as in small batch?

This work tackles the first issue by investigating the random walk behaviour of SGD and the relationship of its diffusion rate to the size of a batch. Based on this and empirical observations, we propose simple set of remedies to close down the generalization gap between the small and large batch training strategies: (1) Use SGD with momentum, gradient clipping, and a decreasing learning rate schedule; (2) adapt the learning rate with batch size (we used a square root scaling); (3) compute batch-norm statistics over several partitions ("ghost batch-norm"); and (4) use a sufficient number of high learning rate training iterations.

Thus, the main point arising from our results is that, in contrast to previous conception, there is no inherent generalization problem with training using large mini batches. That is, model training using large mini-batches can generalize as well as models trained using small mini-batches. Though this answers the first issues, the second issue remained open: can we speed up training by using large batch sizes?

Not long after our paper first appeared, this issue was also answered. Using a Resnet model on Imagenet Goyal et al. (2017) showed that, indeed, significant speedups in training could be achieved using a large batch size. This further highlights the ideas brought in this work and their importance to future scale-up, especially since Goyal et al. (2017) used similar training practices to those we

[8] Due to memory limitation those experiments were conducted with batch of 2048.

described above. The main difference between our works is the use of a linear scaling of the learning rate[9], similarly to Krizhevsky (2014), and as suggested by Bottou (2010). However, we found that linear scaling works less well on CIFAR10, and later work found that linear scaling rules work less well for other architectures on ImageNet (You et al., 2017).

We also note that current "rules of thumb" regarding optimization regime and explicitly learning rate annealing schedule may be misguided. We showed that good generalization can result from extensive amount of gradient updates in which there is no apparent validation error change and training error continues to drop, in contrast to common practice. After our work appeared, Soudry et al. (2017) suggested an explanation to this, and to the logarithmic increase in the weight distance observed in Figure 2. We show this behavior happens even in simple logistic regression problems with separable data. In this case, we exactly solve the asymptotic dynamics and prove that $\mathbf{w}(t) = \log(t)\hat{\mathbf{w}} + O(1)$ where $\hat{\mathbf{w}}$ is to the L_2 maximum margin separator. Therefore, the margin (affecting generalization) improves slowly (as $O(1/\log(t))$, even while the training error is very low. Future work, based on this, may be focused on finding when and how the learning rate should be decreased while training.

Conclusion. In this work we make a first attempt to tackle the "generalization gap" phenomenon. We argue that the initial learning phase can be described using a high-dimensional "random walk on a random potential" process, with a an "ultra-slow" logarithmic increase in the distance of the weights from their initialization, as we observe empirically. Following this observation we suggest several techniques which enable training with large batch without suffering from performance degradation. This implies that the problem is not related to the batch size but rather to the amount of updates. Moreover we introduce a simple yet efficient algorithm "Ghost-BN" which improves the generalization performance significantly while keeping the training time intact.

Acknowledgments

We wish to thank Nir Ailon, Dar Gilboa, Kfir Levy and Igor Berman for their feedback on the initial manuscript. The research leading to these results has received funding from the European Research Council under European Unions Horizon 2020 Program, ERC Grant agreement no. 682203 "Speed-InfTradeoff". The research was partially supported by the Taub Foundation, and the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior/ Interior Business Center (DoI/IBC) contract number D16PC00003. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/IBC, or the U.S. Government.

References

Amodei, D., Anubhai, R., Battenberg, E., et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

Bottou, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pp. 177–186. Springer, 2010.

Bouchaud, J. P. and Georges, A. Anomalous diffusion in disordered media: statistical mechanisms, models and physical applications. *Physics reports*, 195:127–293, 1990.

Bouchaud, J. P. and Comtet, A. Anomalous diffusion in random media of any dimensionality. *J. Physique*, 48: 1445–1450, 1987.

Bray, A. J. and Dean, D. S. Statistics of critical points of Gaussian fields on large-dimensional spaces. *Physical Review Letters*, 98(15):1–5, 2007.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. The Loss Surfaces of Multilayer Networks. *AISTATS15*, 38, 2015.

Das, D., Avancha, S., Mudigere, D., et al. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

Dauphin, Y., de Vries, H., Chung, J., and Bengio, Y. Rmsprop and equilibrated adaptive learning rates for non-convex optimization. corr abs/1502.04390 (2015).

Dauphin, Y., Pascanu, R., and Gulcehre, C. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS*, pp. 1–9, 2014.

Dean, J., Corrado, G., Monga, R., et al. Large scale distributed deep networks. In *NIPS*, pp. 1223–1231, 2012.

[9] e.g., Goyal et al. (2017) also used an initial warm-phase for the learning rate, however, this has a similar effect to the gradient clipping we used, since this clipping was mostly active during the initial steps of training.

Deng, J., Dong, W., Socher, R., et al. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

Dinh, L., Pascanu, R., Bengio, S., and Bengio, Y. Sharp minima can generalize for deep nets. *arXiv preprint arXiv:1703.04933*, 2017.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Durrett, R. Multidimensional random walks in random environments with subclassical limiting behavior. *Communications in Mathematical Physics*, 104(1):87–102, 1986.

Ge, R., Huang, F., Jin, C., and Yuan, Y. Escaping from saddle points-online stochastic gradient for tensor decomposition. In *COLT*, pp. 797–842, 2015.

Girosi, F., Jones, M., and Poggio, T. Regularization theory and neural networks architectures. *Neural computation*, 7(2):219–269, 1995.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pp. 249–256, 2010.

Goyal, P., Dollár, P., Girshick, R., et al. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Hardt, M., Recht, B., and Singer, Y. Train faster, generalize better: Stability of stochastic gradient descent. *ICML*, pp. 1–24, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, A. Learning multiple layers of features from tiny images. 2009.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

LeCun, Y., Bottou, L., and Orr, G. Efficient backprop in neural networks: Tricks of the trade (orr, g. and müller, k., eds.). *Lecture Notes in Computer Science*, 1524, 1998a.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998b.

Li, M. *Scaling Distributed Machine Learning with System and Algorithm Co-design*. PhD thesis, Intel, 2017.

Li, M., Zhang, T., Chen, Y., and Smola, A. J. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 661–670. ACM, 2014.

Luong, M.-T., Pham, H., and Manning, C. D. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Marinari, E., Parisi, G., Ruelle, D., and Windey, P. Random Walk in a Random Environment and 1f Noise. *Physical Review Letters*, 50(1):1223–1225, 1983.

Mnih, V., Kavukcuoglu, K., Silver, D., et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Montavon, G., Orr, G., and Müller, K.-R. *Neural Networks: Tricks of the Trade*. 2 edition, 2012. ISBN 978-3-642-35288-1.

Ruder, S. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

Silver, D., Huang, A., Maddison, C. J., et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Simonyan, K. e. a. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Soudry, D., Hoffer, E., and Srebro, N. The Implicit Bias of Gradient Descent on Separable Data. *ArXiv e-prints*, October 2017.

Soudry, D. and Hoffer, E. Exponentially vanishing sub-optimal local minima in multilayer neural networks. *arXiv preprint arXiv:1702.05777*, 2017.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pp. 1139–1147, 2013.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818–2826, 2016.

Wan, L., Zeiler, M., Zhang, S., LeCun, Y., and Fergus, R. Regularization of neural networks using dropconnect. ICML'13, pp. III–1058–III–1066. JMLR.org, 2013.

Wu, Y., Schuster, M., Chen, Z., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

You, Y., Gitman, I., and Ginsburg, B. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 2017.

Zagoruyko, K. Wide residual networks. In *BMVC*, 2016.
Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. Understanding deep learning requires rethinking generalization. In *ICLR*, 2017.
Zhang, S., Choromanska, A. E., and LeCun, Y. Deep learning with elastic averaging sgd. In *NIPS*, pp. 685–693, 2015.

Appendix

A Derivation of eq. (6)

Note that we can write the mini-batch gradient as

$$\hat{\mathbf{g}} = \frac{1}{M}\sum_{n=1}^{N} \mathbf{g}_n s_n \text{ with } s_n \triangleq \begin{cases} 1 & \text{, if } n \in B \\ 0 & \text{, if } n \notin B \end{cases}$$

Clearly, $\hat{\mathbf{g}}$ is an unbiased estimator of $\mathbf{g}$, if

$$\mathbb{E}s_n = P\left(s_n = 1\right) = \frac{M}{N}\,.$$

since then

$$\mathbb{E}\hat{\mathbf{g}} = \frac{1}{M}\sum_{n=1}^{N} \mathbf{g}_n \mathbb{E}s_n = \frac{1}{N}\sum_{n=1}^{N} \mathbf{g}_n = \mathbf{g}\,.$$

First, we consider the simpler case of sampling with replacement. In this case it easy to see that different minibatches are uncorrelated, and we have

$$\begin{aligned}\mathbb{E}\left[s_n s_{n'}\right] &= P\left(s_n = 1\right)\delta_{nn'} + P\left(s_n = 1, s_{n'} = 1\right)\left(1 - \delta_{nn'}\right) \\ &= \frac{M}{N}\delta_{nn'} + \frac{M^2}{N^2}\left(1 - \delta_{nn'}\right)\,.\end{aligned}$$

and therefore

$$\begin{aligned}\mathrm{cov}\left(\hat{\mathbf{g}}, \hat{\mathbf{g}}\right) &= \mathbb{E}\left[\hat{\mathbf{g}}\hat{\mathbf{g}}^\top\right] - \mathbb{E}\hat{\mathbf{g}}\mathbb{E}\hat{\mathbf{g}}^\top \\ &= \frac{1}{M^2}\sum_{n=1}^{N}\sum_{n'=1}^{N}\mathbb{E}\left[s_n s_{n'}\right]\mathbf{g}_n\mathbf{g}_{n'}^\top - \mathbf{g}\mathbf{g}^\top \\ &= \frac{1}{M^2}\sum_{n=1}^{N}\sum_{n'=1}^{N}\left[\frac{M}{N}\delta_{nn'} + \frac{M^2}{N^2}\left(1 - \delta_{nn'}\right)\right]\mathbf{g}_n\mathbf{g}_{n'}^\top - \mathbf{g}\mathbf{g}^\top \\ &= \left(\frac{1}{M} - \frac{1}{N}\right)\frac{1}{N}\sum_{n=1}^{N}\mathbf{g}_n\mathbf{g}_n^\top,\end{aligned}$$

which confirms eq. (6).

Next, we consider the case of sampling without replacement. In this case the selector variables are now different and correlated between different mini-batches (*e.g.*, with indices t and $t+k$), since we cannot select previous samples. Thus, these variables s_n^t and s_n^{t+k} have the following second-order statistics

$$\begin{aligned}\mathbb{E}\left[s_n^t s_{n'}^{t+k}\right] &= P\left(s_n^t = 1, s_{n'}^{t+k} = 1\right) \\ &= \frac{M}{N}\delta_{nn'}\delta_{k0} + \frac{M}{N}\frac{M}{N-1}\left(1 - \delta_{nn'}\delta_{k0}\right)\,.\end{aligned}$$

This implies

$$\begin{aligned}&\mathbb{E}\left[\hat{\mathbf{g}}_t\hat{\mathbf{g}}_{t+k}^\top\right] - \mathbb{E}\hat{\mathbf{g}}_t\mathbb{E}\hat{\mathbf{g}}_{t+k}^\top \\ &= \mathbb{E}\left[\left(\frac{1}{M}\sum_{n=1}^{N}s_n^t\mathbf{g}_n\right)\left(\frac{1}{M}\sum_{n'=1}^{N}s_{n'}^{t+k}\mathbf{g}_{n'}^\top\right)\right] - \mathbf{g}\mathbf{g}^\top \\ &= \frac{1}{M^2}\sum_{n=1}^{N}\sum_{n'=1}^{N}\mathbb{E}\left[s_n^t s_{n'}^{t+k}\right]\mathbf{g}_n\mathbf{g}_{n'}^\top - \mathbf{g}\mathbf{g}^\top \\ &= \frac{1}{M^2}\sum_{n=1}^{N}\sum_{n'=1}^{N}\left[\left(\frac{M}{N} - \frac{M^2}{N^2 - N}\right)\delta_{nn'}\delta_{k0} - \frac{M^2}{N^2 - N}\right]\mathbf{g}_n\mathbf{g}_{n'}^\top - \mathbf{g}\mathbf{g}^\top\end{aligned}$$

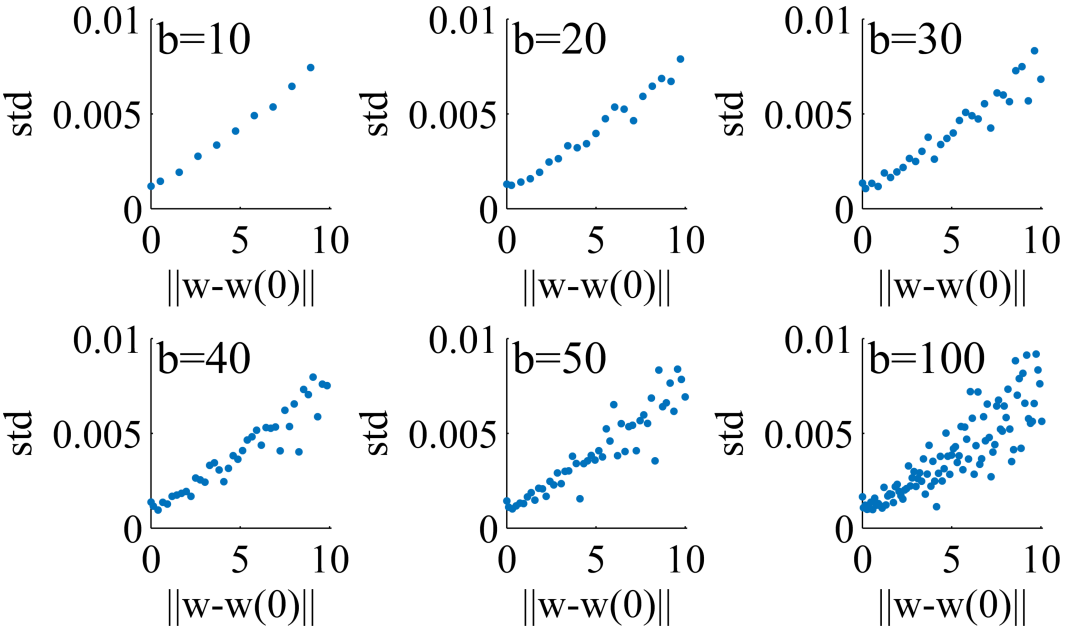


Figure 4: **The standard deviation of the loss shows linear dependence on weight distance (eq. 8) as predicted by the "random walk on a random potential" model with $\alpha = 2$ we found in the main paper.** To approximate the ensemble average in eq. 8 we divided the x-axis to b bins and calculated the empiric average in each bin. Each panel shows the resulting graph for a different value of b.

so, if $k = 0$ the covariance is

$$\mathbb{E}\left[\hat{\mathbf{g}}_t\hat{\mathbf{g}}_t^\top\right] - \mathbb{E}\hat{\mathbf{g}}_t\mathbb{E}\hat{\mathbf{g}}_t^\top = \left(\frac{1}{M} - \frac{1}{N-1}\right)\frac{1}{N}\sum_{n=1}^{N}\mathbf{g}_n\mathbf{g}_n^\top + \frac{1}{N-1}\mathbf{g}\mathbf{g}^\top$$
$$\overset{M\ll N}{\approx} \frac{1}{M}\left(\frac{1}{N}\sum_{n=1}^{N}\mathbf{g}_n\mathbf{g}_n^\top\right)$$

while the covariance between different minibatches ($k \neq 0$) is much smaller for $M \ll N$

$$\mathbb{E}\left[\hat{\mathbf{g}}_t\hat{\mathbf{g}}_{t+k}^\top\right] - \mathbb{E}\hat{\mathbf{g}}_t\mathbb{E}\hat{\mathbf{g}}_{t+k}^\top = \frac{1}{N-1}\mathbf{g}\mathbf{g}^\top$$

this again confirms eq. (6).

B Estimating α from random potential

The logarithmic increase in weight distance (Figure 2 in the paper) matches a "random walk on a random potential" model with $\alpha = 2$. In such a model the loss auto-covariance asymptotically increases with the square of the weight distance, or, equivalently (Marinari et al., 1983), the standard deviation of the loss difference asymptotically increases linearly with the weight distance

$$\text{std} \triangleq \sqrt{\mathbb{E}\left(L\left(\mathbf{w}\right) - L\left(\mathbf{w}_0\right)\right)^2} \sim \left\|\mathbf{w} - \mathbf{w}_0\right\| . \tag{8}$$

In this section we examine this behavior: in Figure we indeed find such a linear behavior, confirming the prediction of our model with $\alpha = 2$.

To obtain the relevant statistics to plot eq. 8 we conducted the following experiment on Resnet44 model (He et al., 2016). We initialized the model weights, $\mathbf{w}_0$, according to Glorot & Bengio (2010), and repeated the following steps a 1000 times, given some parameter c:

- Sample a random direction $\mathbf{v}$ with norm one.
- Sample a scalar z uniformly in some range $[0, c]$.
- Choose $\mathbf{w} = \mathbf{w}_0 + z\mathbf{v}$.
- Save $||\mathbf{w} - \mathbf{w}_0||$ and $L(\mathbf{w})$.

We have set the parameter c so that the maximum weight distance from initialization $||\mathbf{w} - \mathbf{w}_0||$ is equal to the same maximal distance in Figure 2 in the paper, *i.e.,* $c \approx 10$.

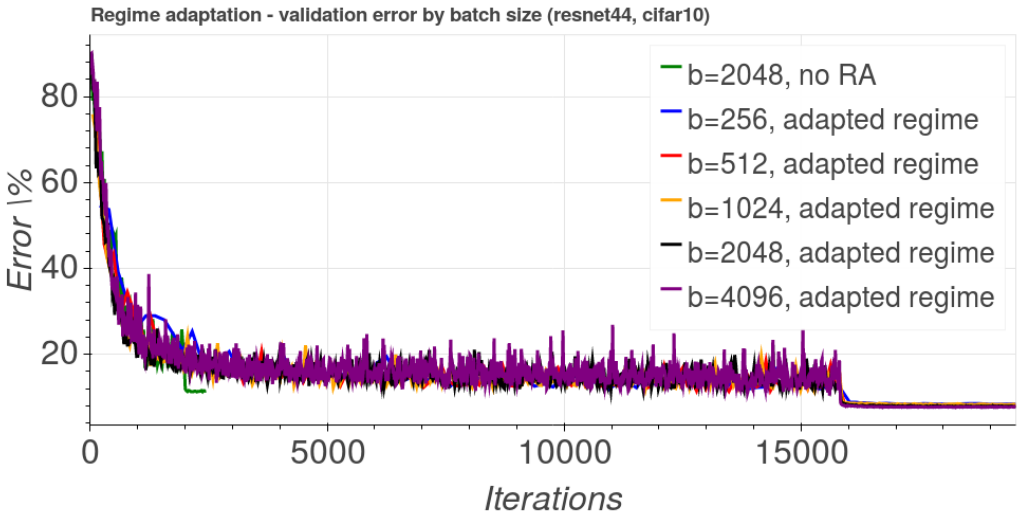


Figure 5: Comparing regime adapted large batch training vs. a 2048 batch with no adaptation.

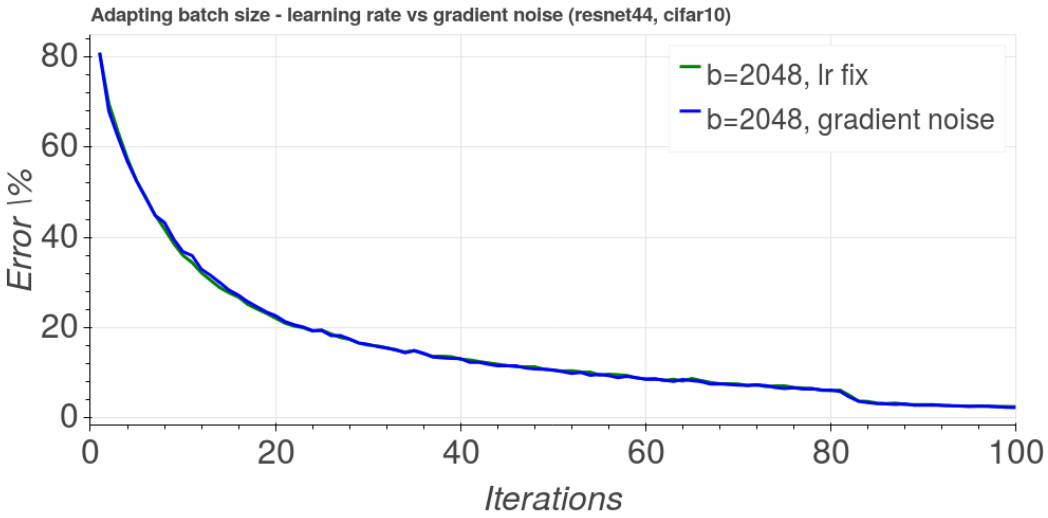


Figure 6: Comparing a learning scale fix for a 2048 batch, to a multiplicative noise to the gradient of the same scale

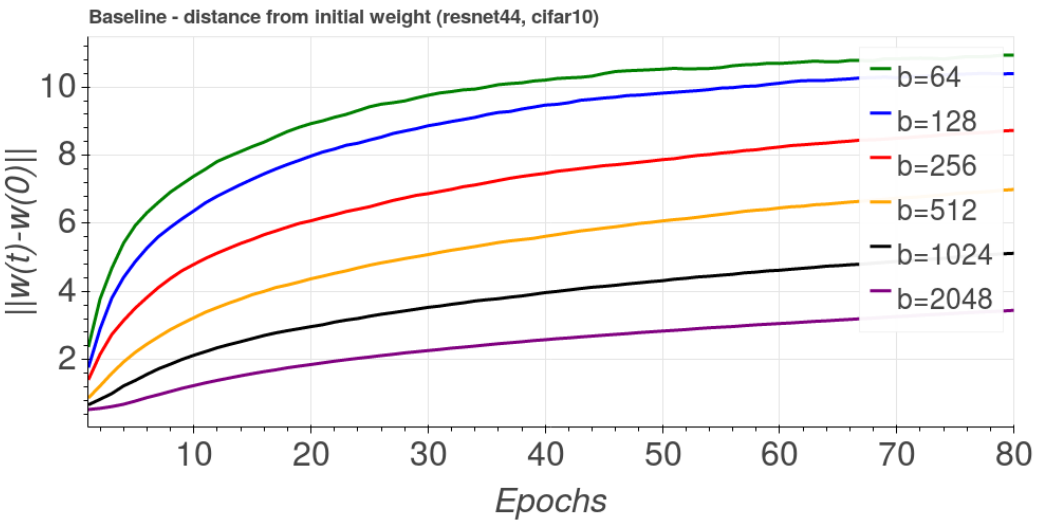


Figure 7: Comparing L_2 distance from initial weight for different batch sizes

Bayesian Optimization with Unknown Constraints

Michael A. Gelbart, Jasper Snoek, and Ryan P. Adams

School of Engineering and Applied Sciences, Harvard University

{mgelbart, jsnoek, rpa} @ seas.harvard.edu

Abstract

Recent work on Bayesian optimization has shown its effectiveness in global optimization of difficult black-box objective functions. Many real-world optimization problems of interest also have constraints which are unknown *a priori*. In this paper, we study Bayesian optimization for constrained problems in the general case that noise may be present in the constraint functions, and the objective and constraints may be evaluated independently. We provide motivating practical examples, and present a general framework to solve such problems. We demonstrate the effectiveness of our approach on optimizing the performance of online latent Dirichlet allocation subject to topic sparsity constraints, tuning a neural network given test-time memory constraints, and optimizing Hamiltonian Monte Carlo to achieve maximal effectiveness in a fixed time, subject to passing standard convergence diagnostics.

1 Introduction

Bayesian optimization (Mockus et al., 1978) is a method for performing global optimization of unknown "black box" objectives that is particularly appropriate when objective function evaluations are expensive (in any sense, such as time or money). For example, consider a food company trying to design a low-calorie variant of a popular cookie. In this case, the design space is the space of possible recipes and might include several key parameters such as quantities of various ingredients and baking times. Each evaluation of a recipe entails computing (or perhaps actually measuring) the number of calories in the proposed cookie. Bayesian optimization can be used to propose new candidate recipes such that good results are found with few evaluations.

Now suppose the company also wants to ensure the *taste* of the cookie is not compromised when calories are reduced. Therefore, for each proposed low-calorie recipe, they perform a taste test with sample customers. Because different people, or the same people at different times, have differing opinions about the taste of cookies, the company decides to require that at least 95% of test subjects must like the new cookie. This is a constrained optimization problem:

$$\min_{\mathbf{x}} c(\mathbf{x}) \text{ s.t. } \rho(\mathbf{x}) \geq 1 - \epsilon \ ,$$

where $\mathbf{x}$ is a real-valued vector representing a recipe, $c(\mathbf{x})$ is the number of calories in recipe $\mathbf{x}$, $\rho(\mathbf{x})$ is the fraction of test subjects that like recipe $\mathbf{x}$, and $1 - \epsilon$ is the minimum acceptable fraction, i.e., 95%.

This paper presents a general formulation of constrained Bayesian optimization that is suitable for a large class of problems such as this one. Other examples might include tuning speech recognition performance on a smart phone such that the user's speech is transcribed within some acceptable time limit, or minimizing the cost of materials for a new bridge, subject to the constraint that all safety margins are met.

Another use of constraints arises when the search space is known *a priori* but occupies a complicated volume that cannot be expressed as simple coordinate-wise bounds on the search variables. For example, in a chemical synthesis experiment, it may be known that certain combinations of reagents cause an explosion to occur. This constraint is not unknown in the sense of being a discovered property of the environment as in the examples above—we do not want to discover the constraint boundary by trial and error explosions of our laboratory. Rather, we would like to specify this constraint using a boolean noise-free oracle function that declares input vectors as valid or invalid. Our formulation of constrained Bayesian optimization naturally encapsulates such constraints.

1.1 Bayesian Optimization

Bayesian optimization proceeds by iteratively developing a global statistical model of the unknown objective function. Starting with a prior over functions and a likelihood, at each iteration a posterior distribution is computed by conditioning on the previous evaluations of the objective function, treating them as observations in a Bayesian nonlinear regression. An *acquisition function* is used to map beliefs about the objective function to a measure of how promising each location in input space is, if it were to be evaluated next. The goal is then to find the input that maximizes the acquisition function, and submit it for function evaluation.

Maximizing the acquisition function is ideally a relatively easy proxy optimization problem: evaluations of the acquisition function are often inexpensive, do not require the objective to be queried, and may have gradient information available. Under the assumption that evaluating the objective function is expensive, the time spent computing the best next evaluation via this inner optimization problem is well spent. Once a new result is obtained, the model is updated, the acquisition function is recomputed, and a new input is chosen for evaluation. This completes one iteration of the Bayesian optimization loop.

For an in-depth discussion of Bayesian optimization, see Brochu et al. (2010b) or Lizotte (2008). Recent work has extended Bayesian optimization to multiple tasks and objectives (Krause and Ong, 2011; Swersky et al., 2013; Zuluaga et al., 2013) and high dimensional problems (Wang et al., 2013; Djolonga et al., 2013). Strong theoretical results have also been developed (Srinivas et al., 2010; Bull, 2011; de Freitas et al., 2012). Bayesian optimization has been shown to be a powerful method for the meta-optimization of machine learning algorithms (Snoek et al., 2012; Bergstra et al., 2011) and algorithm configuration (Hutter et al., 2011).

1.2 Expected Improvement

An acquisition function for Bayesian optimization should address the exploitation vs. exploration tradeoff: the idea that we are interested both in regions where the model believes the objective function is low ("exploitation") and regions where uncertainty is high ("exploration"). One such choice is the Expected Improvement (EI) criterion (Mockus et al., 1978), an acquisition function shown to have strong theoretical guarantees (Bull, 2011) and empirical effectiveness (e.g., Snoek et al., 2012). The expected improvement, $\mathrm{EI}(\mathbf{x})$, is defined as the expected amount of improvement over some target t, if we were to evaluate the objective function at $\mathbf{x}$:

$$\mathrm{EI}(\mathbf{x}) = \mathbb{E}[(t-y)_+] = \int_{-\infty}^{\infty} (t-y)_+ p(y \,|\, \mathbf{x}) \, \mathrm{d}y \;, \tag{1}$$

where $p(y \,|\, \mathbf{x})$ is the predictive marginal density of the objective function at $\mathbf{x}$, and $(t-y)_+ \equiv \max(0, t-y)$ is the improvement (in the case of minimization) over the target t. EI encourages both exploitation and exploration because it is large for inputs with a low predictive mean (exploitation) and/or a high predictive variance (exploration). Often, t is set to be the minimum over previous observations (e.g., Snoek et al., 2012), or the minimum of the expected value of the objective (Brochu et al., 2010a). Following our formulation of the problem, we use the minimum expected value of the objective such that the probabilistic constraints are satisfied (see Section 1.5, Eq., 6).

When the predictive distribution under the model is Gaussian, EI has a closed-form expression (Jones, 2001):

$$\mathrm{EI}(\mathbf{x}) = \sigma(\mathbf{x}) \left(z(\mathbf{x})\Phi\left(z(\mathbf{x})\right) + \phi\left(z(\mathbf{x})\right)\right) \tag{2}$$

where $z(\mathbf{x}) \equiv \frac{t-\mu(\mathbf{x})}{\sigma(\mathbf{x})}$, $\mu(\mathbf{x})$ is the predictive mean at $\mathbf{x}$, $\sigma^2(\mathbf{x})$ is the predictive variance at $\mathbf{x}$, $\Phi(\cdot)$ is the standard normal CDF, and $\phi(\cdot)$ is the standard normal PDF. This function is differentiable and fast to compute, and can therefore be maximized with a standard gradient-based optimizer. In Section 3 we present an acquisition function for constrained Bayesian optimization based on EI.

1.3 Our Contributions

The main contribution of this paper is a general formulation for constrained Bayesian optimization, along with an acquisition function that enables efficient optimization of such problems. Our formulation is suitable for addressing a large class of constrained problems, including those considered in previous work. The specific improvements are enumerated below.

First, our formulation allows the user to manage uncertainty when constraint observations are noisy. By reformulating the problem with probabilistic constraints, the user can directly address this uncertainty by specifying the required confidence that constraints are satisfied.

Second, we consider the class of problems for which the objective function and constraint function need not be evaluated jointly. In the cookie example, the number of calories might be predicted very cheaply with a simple calculation, while evaluating the taste is a large undertaking requiring human trials. Previous methods, which assume joint evaluations, might query a particular recipe only to discover that the objective (calorie) function for that recipe is highly unfavorable. The resources spent simultaneously evaluating the constraint (taste) function would then be very poorly spent. We present an acquisition function for such problems, which incorporates this user-specified cost information.

Third, our framework, which supports an arbitrary number of constraints, provides an expressive language for specifying arbitrarily complicated restrictions on the parameter search spaces. For example if the total memory usage of a neural network must be within some bound, this restriction could be encoded as a separate, noise-free constraint with very low cost. As described above, evaluating this low-cost constraint would take priority over the more expensive constraints and/or objective function.

1.4 Prior Work

There has been some previous work on constrained Bayesian optimization. Gramacy and Lee (2010) propose an acquisition function called the integrated expected conditional improvement (IECI), defined as

$$\mathrm{IECI}(\mathbf{x}) = \int_{\mathcal{X}} \left[\mathrm{EI}(\mathbf{x}') - \mathrm{EI}(\mathbf{x}'|\mathbf{x})\right] h(\mathbf{x}')\mathrm{d}\mathbf{x}' \tag{3}$$

In the above, $\mathrm{EI}(\mathbf{x}')$ is the expected improvement at $\mathbf{x}'$, $\mathrm{EI}(\mathbf{x}'|\mathbf{x})$ is the expected improvement at $\mathbf{x}'$ given that the objective has been observed at $\mathbf{x}$ (but without making any assumptions about the observed value), and $h(\mathbf{x}')$ is an arbitrary density over $\mathbf{x}'$. In words, the IECI at $\mathbf{x}$ is the expected reduction in EI at $\mathbf{x}'$, under the density $h(\mathbf{x}')$, caused by observing the objective at $\mathbf{x}$. Gramacy and Lee use IECI for constrained Bayesian optimization by setting $h(\mathbf{x}')$ to the probability of satisfying the constraint. This formulation encourages evaluations that inform the model in places that are likely to satisfy the constraint.

Zuluaga et al. (2013) propose the Pareto Active Learning (PAL) method for finding Pareto-optimal solutions when multiple objective functions are present and the input space is a discrete set. Their algorithm classifies each design candidate as either Pareto-optimal or not, and proceeds iteratively until all inputs are classified. The user may specify a confidence parameter determining the tradeoff between the number of function evaluations and prediction accuracy. Constrained optimization can

be considered a special case of multi-objective optimization in which the user's utility function for the "constraint objectives" is an infinite step function: constant over the feasible region and negative infinity elsewhere. However, PAL solves different problems than those we intend to solve, because it is limited to discrete sets and aims to classify each point in the set versus finding a single optimal solution.

Snoek (2013) discusses constrained Bayesian optimization for cases in which constraint violations arise from a failure mode of the objective function, such as a simulation crashing or failing to terminate. The thesis introduces the weighted expected improvement acquisition function, namely expected improvement weighted by the predictive probability that the constraint is satisfied at that input.

1.5 Formalizing the Problem

In Bayesian optimization, the objective and constraint functions are in general unknown for two reasons. First, the functions have not been observed everywhere, and therefore we must interpolate or extrapolate their values to new inputs. Second, our observations may be noisy; even after multiple observations at the same input, the true function is not known. Accounting for this uncertainty is the role of the model, see Section 2.

However, before solving the problem, we must first define it. Returning to the cookie example, each taste test yields an estimate of $\rho(\mathbf{x})$, the fraction of test subjects that like recipe $\mathbf{x}$. But uncertainty is always present, even after many measurements. Therefore, it is impossible to be certain that the constraint $\rho(\mathbf{x}) \geq 1 - \epsilon$ is satisfied for any $\mathbf{x}$. Likewise, the objective function can only be evaluated point-wise and, if noise is present, it may never be determined with certainty.

This is a stochastic programming problem: namely, an optimization problem in which the objective and/or constraints contain uncertain quantities whose probability distributions are known or can be estimated (see e.g., Shapiro et al., 2009). A natural formulation of these problems is to minimize the objective function *in expectation*, while satisfying the constraints *with high probability*. The condition that the constraint be satisfied with high probability is called a *probabilistic constraint*. This concept is formalized below.

Let $f(\mathbf{x})$ represent the objective function. Let $\mathcal{C}(\mathbf{x})$ represent the the *constraint condition*, namely the boolean function indicating whether or not the constraint is satisfied for input $\mathbf{x}$. For example, in the cookie problem, $\mathcal{C}(\mathbf{x}) \iff \rho(\mathbf{x}) \geq 1 - \epsilon$. Then, our probabilistic constraint is

$$\Pr(\mathcal{C}(\mathbf{x})) \geq 1 - \delta\,, \tag{4}$$

for some user-specified minimum confidence $1 - \delta$.

If K constraints are present, for each constraint $k \in (1, \ldots, K)$ we define $\mathcal{C}_k(\mathbf{x})$ to be the constraint condition for constraint k. Each constraint may also have its own tolerance δ_k, so we have K probabilistic constraints of the form

$$\Pr(\mathcal{C}_k(\mathbf{x})) \geq 1 - \delta_k\,. \tag{5}$$

All K probabilistic constraints must ultimately be satisfied at a solution to the optimization problem.[1]

Given these definitions, a general class of constrained Bayesian optimization problems can be formulated as

$$\min_{\mathbf{x}} \mathbb{E}[f(\mathbf{x})] \text{ s.t. } \forall k \;\; \Pr(\mathcal{C}_k(\mathbf{x})) \geq 1 - \delta_k\,. \tag{6}$$

The remainder of this paper proposes methods for solving problems in this class using Bayesian optimization. Two key ingredients are needed: a model of the objective and constraint functions (Section 2), and an acquisition function that determines which input $\mathbf{x}$ would be most beneficial to observe next (Section 3).

[1] Note: this formulation is based on individual constraint satisfaction for all constraints. Another reasonable formulation requires the (joint) probability that *all* constraints are satisfied to be above some single threshold.

2 Modeling the Constraints

2.1 Gaussian Processes

We use Gaussian processes (GPs) to model both the objective function $f(\mathbf{x})$ and the constraint functions. A GP is a generalization of the multivariate normal distribution to arbitrary index sets, including infinite length vectors or functions, and is specified by its positive definite covariance kernel function $K(\mathbf{x}, \mathbf{x}')$. GPs allow us to condition on observed data and tractably compute the posterior distribution of the model for any finite number of query points. A consequence of this property is that the marginal distribution at any single point is univariate Gaussian with a known mean and variance. See Rasmussen and Williams (2006) for an in-depth treatment of GPs for machine learning.

We assume the objective and all constraints are independent and model them with independent GPs. Note that since the objective and constraints are all modeled independently, they need not all be modeled with GPs or even with the same types of models as each other. Any combination of models suffices, so long as each one represents its uncertainty about the true function values.

2.2 The latent constraint function, $g(\mathbf{x})$

In order to model constraint conditions $\mathcal{C}_k(\mathbf{x})$, we introduce real-valued latent *constraint functions* $g_k(\mathbf{x})$ such that for each constraint k, the constraint condition $\mathcal{C}_k(\mathbf{x})$ is satisfied if and only if $g_k(\mathbf{x}) \geq 0$.[2] Different observation models lead to different likelihoods on $g(\mathbf{x})$, as discussed below. By computing the posterior distribution of $g_k(\mathbf{x})$ for each constraint, we can then compute $\Pr(\mathcal{C}_k(\mathbf{x})) = \Pr(g_k(\mathbf{x}) \geq 0)$ by simply evaluating the Gaussian CDF using the predictive marginal mean and variance of the GP at $\mathbf{x}$.

Different constraints require different definitions of the constraint function $g(\mathbf{x})$. When the nature of the problem permits constraint observations to be modeled with a Gaussian likelihood, the posterior distribution of $g(\mathbf{x})$ can be computed in closed form. If not, approximations or sampling methods are needed (see Rasmussen and Williams, 2006, p. 41-75). We discuss two examples below, one of each type, respectively.

2.3 Example I: bounded running time

Consider optimizing some property of a computer program such that its running time $\tau(\mathbf{x})$ must not exceed some value $\tau_{\max}$. Because $\tau(\mathbf{x})$ is a measure of time, it is nonnegative for all $\mathbf{x}$ and thus not well-modeled by a GP prior. We therefore choose to model time in logarithmic units. In particular, we define $g(\mathbf{x}) = \log \tau_{\max} - \log \tau$, so that the condition $g(\mathbf{x}) \geq 0$ corresponds to our constraint condition $\tau \leq \tau_{\max}$, and place a GP prior on $g(\mathbf{x})$. For every problem, this transformation implies a particular prior on the original variables; in this case, the implied prior on $\tau(\mathbf{x})$ is the log-normal distribution. In this problem we may also posit a Gaussian likelihood for observations of $g(\mathbf{x})$. This corresponds to the generative model that constraint observations are generated by some true latent function corrupted with i.i.d. Gaussian noise. As with the prior, this choice implies something about the original function $\tau(\mathbf{x})$, in this case a log-normal likelihood. The basis for these choices is their computational convenience. Given a Gaussian prior and likelihood, the posterior distribution is also Gaussian and can be computed in closed form using the standard GP predictive equations.

2.4 Example II: modeling cookie tastiness

Recall the cookie optimization, and let us assume that constraint observations arrive as a set of counts indicating the numbers of people who did and did not like the cookies. We call these *binomial*

[2] Any inequality constraint $g(\mathbf{x}) \leq g_0$ or $g(\mathbf{x}) \geq g_1$ can be represented this way by transforming to a new variable $\hat{g}(\mathbf{x}) \equiv g_0 - g(\mathbf{x}) \geq 0$ or $\hat{g}(\mathbf{x}) \equiv g(\mathbf{x}) - g_1 \geq 0$, respectively, so we set the right-hand side to zero without loss of generality.

constraint observations. Because these observations are discrete, they are not modeled well by a GP prior. Instead, we model the (unknown) binomial probability $\rho(\mathbf{x})$ that a test subject likes cookie $\mathbf{x}$, which is linked to the observations through a binomial likelihood.[3] In Section 1.5, we selected the constraint condition $\rho(\mathbf{x}) \geq 1 - \epsilon$, where $1 - \epsilon$ is the user-specified threshold representing the minimum allowable probability that a test subject likes the new cookie. Because $\rho(\mathbf{x}) \in (0, 1)$ and $g(\mathbf{x}) \in \mathbb{R}$, we define $g(\mathbf{x}) = s^{-1}(\rho(\mathbf{x}))$, where $s(\cdot)$ is a monotonically increasing sigmoid function mapping $\mathbb{R} \to (0, 1)$ as in logistic or probit regression.[4] In our implementation, we use $s(z) = \Phi(z)$, the Gaussian CDF. The likelihood of $g(\mathbf{x})$ given the binomial observations is then the binomial likelihood composed with s^{-1}. Because this likelihood is non-Gaussian, the posterior distribution cannot be computed in closed form, and therefore approximation or sampling methods are needed.

2.5 Integrating out the GP hyperparameters

Following Snoek et al. (2012), we use the Matérn 5/2 kernel for the Gaussian process prior, which corresponds to the assumption that the function being modeled is twice differentiable. This kernel has $D + 1$ hyperparameters in D dimensions: one characteristic length scale per dimension, and an overall amplitude. Again following Snoek et al. (2012), we perform a fully-Bayesian treatment by integrating out these kernel hyperparameters with Markov chain Monte Carlo (MCMC) via slice sampling (Neal, 2000).

When the posterior distribution cannot be computed in closed form due to a non-Gaussian likelihood, we use elliptical slice sampling (Murray et al., 2010) to sample $g(\mathbf{x})$. We also use the prior whitening procedure described in Murray and Adams (2010) to avoid poor mixing due to the strong coupling of the latent values and the kernel hyperparameters.

3 Acquisition Functions

3.1 Constraint weighted expected improvement

Given the probabilistic constraints and the model for a particular problem, it remains to specify an acquisition function that leads to efficient optimization. Here, we present an acquisition function for constrained Bayesian optimization under the Expected Improvement (EI) criterion (Section 1.2). However, the general framework presented here does not depend on this specific choice and can be used in conjunction with any improvement criterion.

Because improvement is not possible when the constraint is violated, we can define an acquisition function for constrained Bayesian optimization by extending the expectation in Eq. 1 to include the additional constraint uncertainty. This results in a constraint-weighted expected improvement criterion, $a(\mathbf{x})$:

$$a(\mathbf{x}) = \mathrm{EI}(\mathbf{x}) \Pr(\mathcal{C}(\mathbf{x})) \tag{7}$$

$$= \mathrm{EI}(\mathbf{x}) \prod_{k=1}^{K} \Pr(\mathcal{C}_k(\mathbf{x})) \tag{8}$$

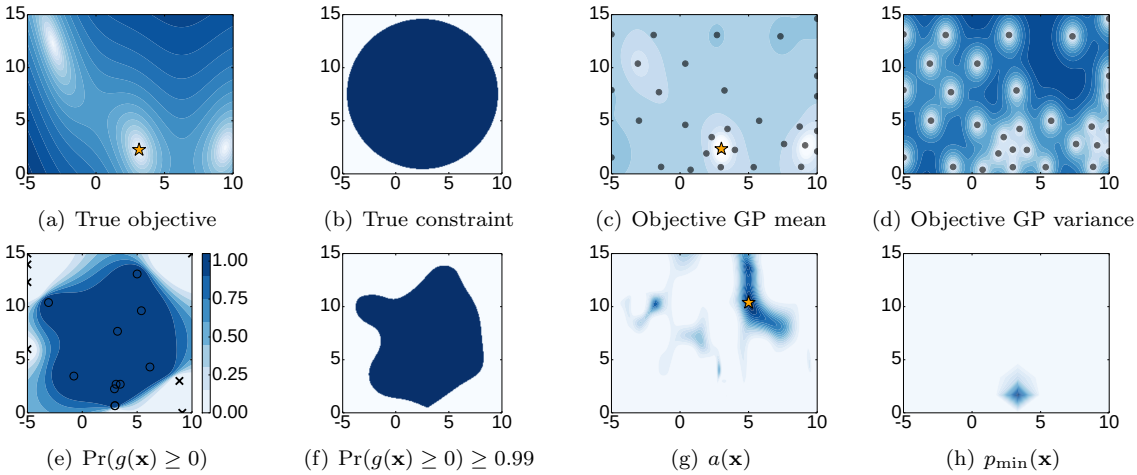
where the second line follows from the assumed independence of the constraints.

Then, the full acquisition function $a(\mathbf{x})$, after integrating out the GP hyperparameters, is given by

$$a(\mathbf{x}) = \int \mathrm{EI}(\mathbf{x}|\theta) p(\theta|\mathbf{D}) p(\mathcal{C}(\mathbf{x})|\mathbf{x}, \mathbf{D}', \omega) p(\omega|\mathbf{D}') \mathrm{d}\theta \mathrm{d}\omega,$$

[3] We use the notation $\rho(\mathbf{x})$ both for the fraction of test subjects who like recipe $\mathbf{x}$ and for its generative interpretation as the probability that a subject likes recipe $\mathbf{x}$.

[4] When the number of binomial trials is one, this model is called Gaussian Process Classification.



(a) True objective (b) True constraint (c) Objective GP mean (d) Objective GP variance

(e) $\Pr(g(\mathbf{x}) \geq 0)$ (f) $\Pr(g(\mathbf{x}) \geq 0) \geq 0.99$ (g) $a(\mathbf{x})$ (h) $p_{\min}(\mathbf{x})$

Figure 1: Constrained Bayesian optimization on the 2D Branin-Hoo function with a disk constraint, after 50 iterations (33 objective evaluations and 17 constraint evaluations): (a) Branin-Hoo function, (b) true constraint, (c) mean of objective function GP, (d) variance of objective function GP, (e) probability of constraint satisfaction, (f) probabilistic constraint, $\Pr(g(\mathbf{x}) \geq 0) \geq 0.99$, (g) acquisition function, $a(\mathbf{x})$, and (h) probability distribution over the location of the minimum, $p_{\min}(\mathbf{x})$. Lighter colors indicate lower values. Objective function observations are indicated with black circles in (c) and (d). Constraint observations are indicated with black $\times$'s (violations) and o's (satisfactions) in (e). Orange stars: (a) unique true minimum of the constrained problem, (c) best solution found by Bayesian optimization, (g) input chosen for the next evaluation, in this case an objective evaluation because $\Delta S_o(\mathbf{x}) > \Delta S_c(\mathbf{x})$ at the next observation location $\mathbf{x}$.

where θ is the set of GP hyperparameters for the objective function model, ω is the set of GP hyperparameters for the constraint model(s), $\mathbf{D} = \{\mathbf{x}_n, y_n\}_{n=1}^N$ are the previous objective function observations, and $\mathbf{D}'$ are the constraint function observations.

3.2 Finding the feasible region

The acquisition function given above is not defined when at least one probabilistic constraint is violated for all $\mathbf{x}$, because in this case the EI target does not exist and therefore EI cannot be computed. In this case we take the acquisition function to include only the second factor,

$$a(\mathbf{x}) = \prod_{k=1}^{K} \Pr(g_k(\mathbf{x}) \geq 0) \tag{9}$$

Intuitively, if the probabilistic constraint is violated everywhere, we ignore the objective function and try to satisfy the probabilistic constraint until it is satisfied somewhere. This acquisition function may also be used if no objective function exists, i.e., if the problem is just to search for any feasible input. This feasibility search is purely exploitative: it searches where the probability of satisfying the constraints is highest. This is possible because the true probability of constraint satisfaction is either zero or one. Therefore, as the algorithm continues to probe a particular region, it will either discover that the region is feasible, or the probability will drop and it will move on to a more promising region.

3.3 Acquisition function for decoupled observations

In some problems, the objective and constraint functions may be evaluated independently. We call this property the *decoupling* of the objective and constraint functions. In decoupled problems, we must choose to evaluate either the objective function or one of the constraint functions at each iteration of Bayesian optimization. As discussed in Section 1.3, it is important to identify problems with this decoupled structure, because often some of the functions are much more expensive to evaluate than others. Bayesian optimization with decoupled constraints is a form of multi-task Bayesian optimization (Swersky et al., 2013), in which the different black-boxes or *tasks* are the objective and decoupled constraint(s), represented by the set $\{\text{objective}, 1, 2, \ldots, K\}$ for K constraints.

3.3.1 Chicken and Egg Pathology

One possible acquisition function for decoupled constraints is the expected improvement of individually evaluating each task. However, the myopic nature of the EI criterion causes a pathology in this formulation that prevents exploration of the design space. Consider a situation, with a single constraint, in which some feasible region has been identified and thus the current best input is defined, but a large unexplored region remains. Evaluating only the objective in this region could not cause improvement as our belief about $\Pr(g(\mathbf{x}) \geq 0)$ will follow the prior and not exceed the threshold $1 - \delta$. Likewise, evaluating only the constraint would not cause improvement because our belief about the objective will follow the prior and is unlikely to become the new best. This is a causality dilemma: we must learn that *both* the objective and the constraint are favorable for improvement to occur, but this is not possible when only a single task is observed. This difficulty suggests a non-myopic aquisition function which assesses the improvement after a sequence of objective and constraint observations. However, such a multi-step acquisition function is intractable in general (Ginsbourger and Riche, 2010).

Instead, to address this pathology, we propose to use the coupled acquisition function (Eq. 7) to select an input $\mathbf{x}$ for observation, followed by a second step to determine which task will be evaluated at $\mathbf{x}$. Following Swersky et al. (2013), we use the entropy search criterion (Hennig and Schuler, 2012) to select a task. However, our framework does not depend on this choice.

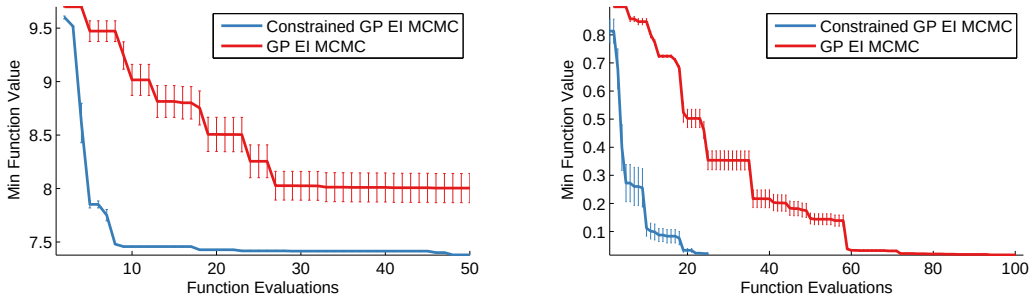
3.3.2 Entropy Search Criterion

Entropy search works by considering $p_{\min}(\mathbf{x})$, the probability distribution over the location of the minimum of the objective function. Here, we extend the definition of $p_{\min}$ to be the probability distribution over the location of the solution to the constrained problem. Entropy search seeks the action that, in expectation, most reduces the relative entropy between $p_{\min}(\mathbf{x})$ and an uninformative base distribution such as the uniform distribution. Intuitively speaking, we want to reduce our uncertainty about $p_{\min}$ as much as possible at each step, or, in other words, maximize our information gain at each step. Following Hennig and Schuler (2012), we choose $b(\mathbf{x})$ to be the uniform distribution on the input space. Given this choice, the relative entropy of $p_{\min}$ and b is the differential entropy of $p_{\min}$ up to a constant that does not affect the choice of task. Our decision criterion is then

$$T^* = \arg\min_T \mathbb{E}_y \left[S\left(p_{\min}^{(y_T)}\right) - S(p_{\min}) \right], \tag{10}$$

where T is one of the tasks in $\{\text{objective}, 1, 2, \ldots, K\}$, T^* is the selected task, $S(\cdot)$ is the differential entropy functional, and $p_{\min}^{(y_T)}$ is $p_{\min}$ conditioned on observing the value y_T for task T. When integrating out the GP covariance hyperparameters, the full form is

$$T^* = \arg\min_T \int S\left(p_{\min}^{(y_T)}\right) p\left(y_T|\theta,\omega\right) \mathrm{d}y_T \, \mathrm{d}\theta \, \mathrm{d}\omega \tag{11}$$



(a) Online LDA (b) Neural Network

Figure 2: Empirical performance of constrained Bayesian optimization for (a) Online Latent Dirichlet Allocation and (b) turning a deep neural network. Blue curve: our method. Red curve: unconstrained Bayesian optimization with constraint violations as large values. Errors bars indicate standard error from 5 independent runs.

where y_T is a possible observed outcome of selecting task T and θ and ω are the objective and constraint GP hyperparameters respectively.[5]

3.3.3 Entropy Search in Practice

Solving Eq. 11 poses several practical difficulties, which we address here in turn. First, estimating $p_{\min}(\mathbf{x})$ requires a discretization of the space. In the spirit of Hennig and Schuler (2012), we form a discretization of N_d points by taking the top N_d points according to the weighted expected improvement criterion. Second, $p_{\min}$ cannot be computed in closed form and must be either estimated or approximated. Swersky et al. (2013) use Monte Carlo sampling to estimate $p_{\min}$ by drawing samples from the GP on the discretization set and finding the minimum. We use the analogous method for constrained optimization: we sample from the objective function GP and all K constraint GPs, and then find the minimum of the objective for which the constraint is satisfied for all K constraint samples.

3.3.4 Incorporating cost information

Following Swersky et al. (2013), we incorporate information about the relative cost of the tasks by simply scaling the acquisition functions by these costs (provided by the user). In doing so, we pick the task with the most information gain per unit cost. If λ_A is the cost of observing task A, then Eq. 10 becomes

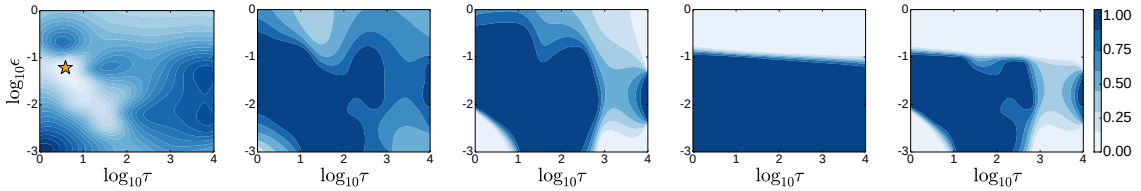
$$A^* = \arg\min_A \frac{1}{\lambda_A} \mathbb{E}_y \left[S\left(p_{\min}^{(y_A)}\right) - S(p_{\min}) \right] . \tag{12}$$

4 Experiments

4.1 Branin-Hoo function

We first illustrate constrained Bayesian optimization on the Branin-Hoo function, a 2D function with three global minima (Fig. 1(a)). We add a decoupled disk constraint $(\mathbf{x}_1 - 2.5)^2 + (\mathbf{x}_2 - 7.5)^2) \leq 50$, shown in Fig. 1(b). This constraint eliminates the upper-left and lower-right solutions, leaving a

[5] For brevity, we have omitted the base entropy term (which does not affect the decision T^*) and the explicit dependence of $p_{\min}$ on θ and ω.



(a) Objective Function (b) Geweke (c) Gelman-Rubin (d) Stability (e) Overall

Figure 3: Tuning Hamiltonian Monte Carlo with constrained Bayesian optimization: (a) objective function model, (b-e) constraint satisfaction probability surfaces for (b) Geweke test, (c) Gelman-Rubin test, (d) stability of the numerical integration, (d) overall, which is the product of the preceding three probability surfaces. In (a), lighter colors correspond to more effective samples, circles indicate function evaluations, and the orange star indicates the best solution. In (b-e), constraint observations are indicated with black $\times$'s (violations) and o's (satisfactions). Vertical axis label at left is for all subplots. Probability colormap at right is for (b-d).

unique global minimum at $\mathbf{x} = (\pi, 2.275)$, indicated by the orange star in Fig. 1(a). After 33 objective function evaluations and 17 constraint evaluations, the best solution is $(3.01, 2.36)$, which satisfies the constraint and has value 0.48 (true best value = 0.40).

4.2 Online LDA with sparse topics

Online Latent Dirichlet Allocation (LDA, Hoffman et al., 2010) is an efficient variational formulation of a popular topic model for learning topics and corresponding word distributions given a corpus of documents. In order for topics to have meaningful semantic interpretations, it is desirable for the word distributions to exhibit sparsity. In this experiment we optimize the hyperparameters of online LDA subject to the constraint that the entropy of the per-topic word distribution averaged over topics is less than $\log_2 200$ bits, which is achieved, for example by allocating uniform density over 200 words. We used the online LDA implementation from Agarwal et al. (2011) and optimized five hyperparameters corresponding to the number of topics (from 2 to 100), two Dirichlet distribution prior base measures (from 0 to 2), and two learning rate parameters (rate from 0.1 to 1, decay from 10^{-5} to 1). As a baseline, we compare with unconstrained Bayesian optimization in which constraint violations are set to the worst possible value for this LDA problem. Fig. 2(a) shows that constrained Bayesian optimization significantly outperforms the baseline. Intuitively, the baseline is poor because the GP has difficulty modeling the sharp discontinuities caused by the large values.

Table 1: Tuning Hamiltonian Monte Carlo.

Experiment	burn-in	τ	ϵ	mass	# samples	accept	**effective samples**
Baseline	10%	100	0.047	1	8.3×10^3	85%	1.1×10^3
BayesOpt	3.8%	2	0.048	1.55	3.3×10^5	70%	9.7×10^4

4.3 Memory-limited neural net

In the final experiment, we optimize the hyperparameters of a deep neural network on the MNIST handwritten digit classification task in a memory-constrained scenario. We optimize over 11 parameters: 1 learning rate, 2 momentum parameters (initial and final), the number of hidden units per

layer (2 layers), the maximum norm on model weights (for 3 sets of weights), and the dropout regularization probabilities (for the inputs and 2 hidden layers). We optimize the classification error on a withheld validation set under the constraint that the total number of model parameters (weights in the network) must be less than one million. This constraint is decoupled from the objective and inexpensive to evaluate, because the number of weights can be calculated directly from the parameters, without training the network. We train the neural network using momentum-based stochastic gradient descent which is notoriously difficult to tune as training can diverge under various combinations of the momentum and learning rate. When training diverges, the objective function cannot be measured. Reporting the constraint violation as a large objective value performs poorly because it introduces sharp discontinuities that are hard to model (Fig. 2). This necessitates a second noisy, binary constraint which is violated when training diverges, for example when the both the learning rate and momentum are too large. The network is trained[6] for 25,000 weight updates and the objective is reported as classification error on the standard validation set. Our Bayesian optimization routine can thus choose between two decoupled tasks, evaluating the memory constraint or the validation error after a full training run. Evaluating the validation error can still cause a constraint violation when the training diverges, which is treated as a binary constraint in our model. Fig. 2(b) shows a comparison of our constrained Bayesian optimization against a baseline standard Bayesian optimization where constraint violations are treated as resulting in a random classifier (90% error). Only the objective evaluations are presented, since constraint evaluations are extremely inexpensive compared to an entire training run. In the event that training diverges on an objective evaluation, we report 90% error. The optimized net has a learning rate of 0.1, dropout probabilities of 0.17 (inputs), 0.30 (first layer), and 0 (second layer), initial momentum 0.86, and final momentum 0.81. Interestingly, the optimization chooses a small first layer (size 312) and a large second layer (size 1772).

4.4 Tuning Markov chain Monte Carlo

Hamiltonian Monte Carlo (HMC) is a popular MCMC sampling technique that takes advantage of gradient information for rapid mixing. However, HMC contains several parameters that require careful tuning. The two basic parameters are the number of leapfrog steps τ, and the step size ϵ. HMC may also include a mass matrix which introduces $\mathcal{O}(D^2)$ additional parameters in D dimensions, although the matrix is often chosen to be diagonal (D parameters) or a multiple of the identity matrix (1 parameter) (Neal, 2011). In this experiment, we optimize the performance of HMC using Bayesian optimization; see Mahendran et al. (2012) for a similar approach. We optimize the following parameters: τ, ϵ, a mass parameter, and the fraction of the allotted computation time spent burning in the chain.

Our experiment measures the number of effective samples (ES) in a fixed computation time; this corresponds to finding chains that minimize estimator variance. We impose the constraints that the generated samples must pass the Geweke (Geweke, 1992) and Gelman-Rubin (Gelman and Rubin, 1992) convergence diagnostics. In particular, we require the worst (largest absolute value) Geweke test score across all variables and chains to be at most 2.0, and the worst (largest) Gelman-Rubin score between chains and across all variables to be at most 1.2. We use PyMC (Patil et al., 2010) for the convergence diagnostics and the LaplacesDemon R package to compute effective sample size. The chosen thresholds for the convergence diagnostics are based on the PyMC and LaplacesDemon documentation. The HMC integration may also diverge for large values of ϵ; we treat this as an additional constraint, and set $\delta = 0.05$ for all constraints. We optimize HMC sampling from the posterior of a logistic regression binary classification problem using the German credit data set from the UCI repository (Frank and Asuncion, 2010). The data set contains 1000 data points, and is normalized to have unit variance. We initialize each chain randomly with D independent draws from a Gaussian distribution with mean zero and standard deviation 10^{-3}. For each set of inputs, we compute two chains, each with 5 minutes of computation time on a single core of a compute node.

[6] We use the Deepnet package: https://github.com/nitishsrivastava/deepnet.

Fig. 3 shows the constraint surfaces discovered by Bayesian optimization for a simpler experiment in which only τ and ϵ are varied; burn-in is fixed at 10% and the mass is fixed at 1. These diagrams yield interpretations of the feasible region; for example, Fig. 3(d) shows that the numerical integration diverges for values of ϵ above $\approx 10^{-1}$. Table 1 shows the results of our 4-parameter optimization after 50 iterations, compared with a baseline that is reflective of a typical HMC configuration: 10% burn in, 100 leapfrog steps, and the step size chosen to yield an 85% proposal accept rate. Each row in the table was produced by averaging 5 independent runs with the given parameters. The optimization chooses to perform very few ($\tau = 2$) leapfrog steps and spend relatively little time (3.8%) burning in the chain, and chooses an acceptance rate of 70%. In contrast, the baseline spends much more time generating each proposal ($\tau = 100$), which produces many fewer total samples and, correspondingly, significantly fewer effective samples.

5 Conclusion

In this paper we extended Bayesian optimization to constrained optimization problems. Because constraint observations may be noisy, we formulate the problem using probabilistic constraints, allowing the user to directly express the tradeoff between cost and risk by specifying the confidence parameter δ. We then propose an acquisition function to perform constrained Bayesian optimization, including the case where the objective and constraint(s) may be observed independently. We demonstrate the effectiveness of our system on the meta-optimization of machine learning algorithms and sampling techniques. Constrained optimization is a ubiquitous problem and we believe this work has applications in areas such as product design (e.g. designing a low-calorie cookie), machine learning meta-optimization (as in our experiments), real-time systems (such as a speech recognition system on a mobile device with speed, memory, and/or energy usage constraints), or any optimization problem in which the objective function and/or constraints are expensive to evaluate and possibly noisy.

Acknowledgements

The authors would like to thank Geoffrey Hinton, George Dahl, and Oren Rippel for helpful discussions, and Robert Nishihara for help with the experiments. This work was partially funded by DARPA Young Faculty Award N66001-12-1-4219. Jasper Snoek is a fellow in the Harvard Center for Research on Computation and Society.

References

Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system, 2011. arXiv: 1110.4198 [cs.LG].

James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Bálázs Kégl. Algorithms for hyper-parameter optimization. In *NIPS*. 2011.

Eric Brochu, Tyson Brochu, and Nando de Freitas. A Bayesian interactive optimization approach to procedural animation design. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2010a.

Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on Bayesian optimization of expensive cost functions, 2010b. arXiv:1012.2599 [cs.LG].

Adam D. Bull. Convergence rates of efficient global optimization algorithms. *JMLR*, (3-4):2879–2904, 2011.

Nando de Freitas, Alex Smola, and Masrour Zoghi. Exponential regret bounds for Gaussian process bandits with deterministic observations. In *ICML*, 2012.

Josip Djolonga, Andreas Krause, and Volkan Cevher. High dimensional Gaussian Process bandits. In *NIPS*, 2013.

Andrew Frank and Arthur Asuncion. UCI machine learning repository, 2010.

Andrew Gelman and Donald R. Rubin. A single series from the Gibbs sampler provides a false sense of security. In *Bayesian Statistics*, pages 625–32. Oxford University Press, 1992.

John Geweke. Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments. In *Bayesian Statistics*, pages 169–193. University Press, 1992.

David Ginsbourger and Rodolphe Riche. Towards Gaussian process-based optimization with finite time horizon. In *Advances in Model-Oriented Design and Analysis*. Physica-Verlag HD, 2010.

Robert B. Gramacy and Herbert K. H. Lee. Optimization under unknown constraints, 2010. arXiv:1004.4027 [stat.ME].

Philipp Hennig and Christian J. Schuler. Entropy search for information-efficient global optimization. *JMLR*, 13, 2012.

Matthew Hoffman, David M. Blei, and Francis Bach. Online learning for latent Dirichlet allocation. In *NIPS*, 2010.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION*, 2011.

Donald R. Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of Global Optimization*, 21, 2001.

Andreas Krause and Cheng Soon Ong. Contextual Gaussian Process bandit optimization. In *NIPS*, 2011.

Dan Lizotte. *Practical Bayesian Optimization*. PhD thesis, University of Alberta, Edmonton, Alberta, 2008.

Nimalan Mahendran, Ziyu Wang, Firas Hamze, and Nando de Freitas. Adaptive MCMC with Bayesian optimization. In *AISTATS*, 2012.

Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of Bayesian methods for seeking the extremum. *Towards Global Optimization*, 2, 1978.

Iain Murray and Ryan P. Adams. Slice sampling covariance hyperparameters of latent Gaussian models. In *NIPS*, 2010.

Iain Murray, Ryan P. Adams, and David J.C. MacKay. Elliptical slice sampling. *JMLR*, 9:541–548, 2010.

Radford Neal. Slice sampling. *Annals of Statistics*, 31:705–767, 2000.

Radford Neal. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*. Chapman and Hall/CRC, 2011.

Anand Patil, David Huard, and Christopher Fonnesbeck. PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software*, 2010.

Carl Rasmussen and Christopher Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

A. Shapiro, D. Dentcheva, and A. Ruszczynski. *Lectures on stochastic programming: modeling and theory*. MPS-SIAM Series on Optimization, Philadelphia, USA, 2009.

Jasper Snoek. *Bayesian Optimization and Semiparametric Models with Applications to Assistive Technology*. PhD thesis, University of Toronto, Toronto, Canada, 2013.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: no regret and experimental design. In *ICML*, 2010.

Kevin Swersky, Jasper Snoek, and Ryan P. Adams. Multi-task Bayesian optimization. In *NIPS*, 2013.

Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, and Nando de Freitas. Bayesian optimization in high dimensions via random embeddings. In *IJCAI*, 2013.

Marcela Zuluaga, Andreas Krause, Guillaume Sergent, and Markus Püschel. Active learning for multi-objective optimization. In *ICML*, 2013.