HEAVY-TAILED NOISE DOES NOT EXPLAIN THE GAP BETWEEN SGD AND ADAM, BUT SIGN DESCENT MIGHT

Anonymous authors
Paper under double-blind review

ABSTRACT

The success of Adam on a wide array of architectures has made it the default in settings where stochastic gradient descent (SGD) performs poorly. However, our theoretical understanding of this discrepancy is lagging, preventing the development of significant improvements on either algorithm. Recent work advances the hypothesis that Adam and other heuristics like gradient clipping outperform SGD on language tasks because the distribution of the error induced by stochasticity has heavy tails. This hypothesis suggests that Adam is using a more robust gradient estimate, leading to the gap. We evaluate this hypothesis by varying the batch size, up to the entire dataset, to control for stochasticity. We present evidence that stochasticity and heavy-tailed noise are not major factors in the performance gap between SGD and Adam. Rather, Adam performs better as the batch size increases, while SGD does not seem to leverage this reduction in noise. This raises the question as to why Adam outperforms SGD in the full-batch setting. Through further investigation of simple normalized variants of SGD, we find that the behavior of Adam with large batches is similar to sign descent.

1 INTRODUCTION

Adam (Kingma and Ba, 2015) and its derivatives have been so successful in training deep learning models that they have become the default optimizer for some architectures. Adam often outperforms stochastic gradient descent (SGD) by such a margin that SGD is considered incapable of training certain models, to the point of being omitted from performance comparisons (e.g. Liu et al., 2020; Anil et al., 2019). Despite this success, we still do not understand why Adam works, much less why it outperforms SGD by such a wide margin. We have made progress understanding why it should not, as in the work of Reddi et al. (2018) who pointed out that Adam does not converge even on convex problems. Reddi et al. address this issue by proposing AMSGrad, which forces the step-sizes to decrease. While necessary for convergence, this does not answer the question as to why Adam outperforms SGD without this fix.

The limited effectiveness of standard theory. We normally analyse optimization algorithms under assumptions like Lipschitz function/gradient and convexity (e.g. Nesterov, 2018, Chapters 2–3). Many works have focused on improving the analysis of Adam and its variants under those same assumptions. But these assumptions are only models of how losses behave. They do not convey the complexity of the optimization process in complex architectures, and are limited to showing that Adam does not do much worse than gradient descent (Défossez et al., 2020; Alacaoglu et al., 2020). In the smooth setting, the progress is maximized by a step of gradient descent. Without additional assumptions, such analyses can only prove that other algorithms perform not-much-worse than gradient descent. Analyses in online learning also struggle to illuminate the gap. Starting solely from the Lipschitz continuity assumption requires decreasing step-sizes (e.g. Hazan, 2019, Thm 3.1), which decrease too quickly to perform well in practice. Our theoretical understanding is still limited in that we cannot describe the empirical behavior we observe—that Adam outperforms SGD in many settings.

As a result, there is a sentiment in the community that the success of these heuristics need not be due to robust theoretical underpinnings, but rather to social dynamics and a co-evolution of deep learning architectures and optimization heuristics that work together—a point made public by Orabona (2020) in blog form, but shared by others in talks and private conversations. These "adaptive" algorithms might actually be adapted to the type of problems where they outperform SGD. But this suggests that

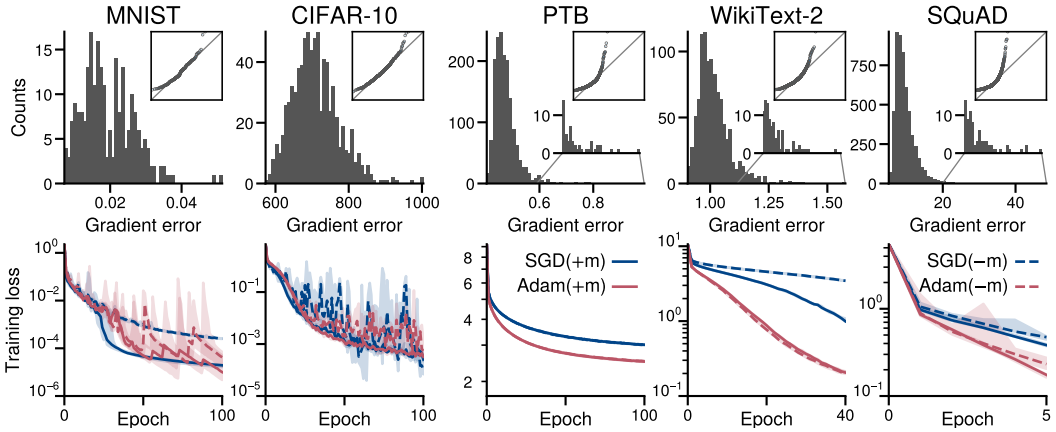

Figure 1: **The Heavy-Tail hypothesis: the gap between SGD and Adam is caused by a heavier tail in the distribution of the stochastic gradient error.** The performance gap between SGD and Adam is larger and more consistent on Transformer models on text data (right: PTB, Wikitext2, SQuAD) than on CNNs on image data (left: MNIST, CIFAR-10), which coincides with a heavier tail in the distribution of the stochastic gradient error. J. Zhang et al. (2020b) hypothesize that heavier tails might be the cause of this gap. **Top:** Distribution of errors in stochastic gradients at initialization, $\frac{1}{2}\|g - \tilde{g}\|^2$ where $\tilde{g}$ is stochastic and g is a full gradient, and QQ-plot comparison against a Gaussian. **Bottom:** Performance of SGD and Adam with (+m) and without momentum (−m) with a small batch size.

they are leveraging some problem structure that our current theory and theory-derived algorithms are missing. Understanding this structure may be key to developing better practical algorithms.

Bad starting assumptions can lead us astray. Standard assumptions in online or stochastic optimization such as bounded gradients or Hessians do not hold for neural networks. Since this theory does not apply to training deep learning models, this can lead to incorrect simplifications or conclusions on what happens during training. A recent example of the mismatch between optimization theory and the actual behavior of optimizers is the "edge of stability" phenomenon (Cohen et al., 2021). Theory, using smoothness, suggests using a step-size inversely proportional to the (largest eigenvalue of the) Hessian. Empirically however, Cohen et al. show that the reverse dependence holds; the Hessian instead evolves along the optimization trajectory to eventually reach the inverse of the step-size, a behavior analyzed by Arora et al. (2022) and Ahn et al. (2022). It is likely that other conclusions or "common knowledge" from optimization theory do not apply due to inappropriate assumptions.

The heavy-tailed assumption. Beyond smoothness or Lipschitzness, recent works have proposed alternative assumptions to model the behavior of optimizers on neural networks. One such assumption comes from J. Zhang et al. (2020b), who hypothesize that the gap in performance might arise from a *heavy-tailed distribution* of the error induced by stochasticity. They notice a larger and more consistent gap in the performance of SGD and Adam on language models than on image models, which coincides with a heavier tail in the distribution of the stochastic gradient error.[1] We reproduce their observation in Figure 1. The mechanism proposed by J. Zhang et al. is that heavy-tailed errors have a large impact on SGD than on Adam or clipped gradient descent, as the introduction of bias in the estimation of the gradient direction reduces the variance. This hypothesis suggests that a path to designing better algorithms is to focus on robustness to heavy-tailed noise, as in the work of Gorbunov et al. (2020) which combines acceleration and gradient clipping, or Srinivasan et al. (2021) which leverages statistical estimators tailored to heavy-tailed distributions.

Inconsistency with large batch results. While J. Zhang et al. (2020b) give a mechanism to link heavy-tailed errors to this gap, and note a correlation between heavy-tailedness and cases where Adam outperforms SGD, there is limited empirical evidence that stochasticity is the root cause of this

[1]In the sense that for a stochastic gradient $\tilde{g}$ and full gradient g, the distribution of $\|\tilde{g} - g\|^2$ is well approximated by a Gaussian for image models, but is closer to an α-stable distribution for language models.

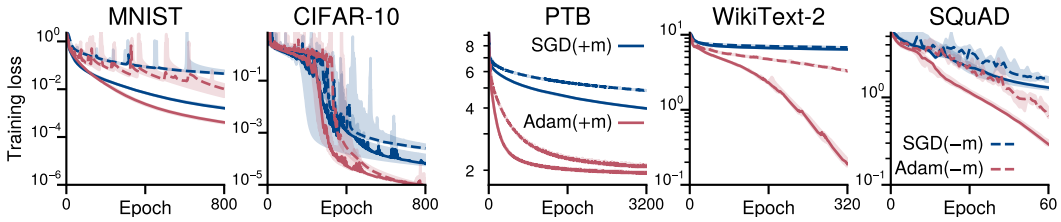

Figure 2: **The gap does not disappear when training in full batch.** Repeating the training procedure of Figure 1 in full batch reveals a similar—or larger—gap between SGD and Adam.

gap. In fact, there are reasons to believe noise might not be a major contributor. For example, the lack of variance in the behavior of the optimizers on language tasks in Figure 1 suggests they might be less sensitive to noise than on image tasks. Moreover, we would expect the gap to diminish as the noise is reduced in other ways, such as increasing the batch size. However, methods such as LAMB (You et al., 2020) or plain Adam find success in large batch settings (Nado et al., 2021), suggesting a competitive advantage when noise is removed. Studies of batch size scaling have also found that Adam scales better with batch size than SGD (G. Zhang et al., 2019).

Alternative views. These empirical results cast doubt on the idea that heavy-tailed noise is the primary factor behind the performance improvement of Adam over SGD. Hypotheses based on deterministic properties might provide better descriptions of the cause of this gap—we discuss them in more details in Section 5. Alternatively, we might look at other interpretations of Adam to gain insight on what algorithmic components enable the gap. One such interpretation is the view of Adam as a variant of *sign descent*, which was the main motivation of RMSprop (Tieleman and Hinton, 2012), as studied by Balles and Hennig (2018) and Bernstein et al. (2018). However, it remains unclear whether the performance of Adam can be explained by its similarities to a simpler algorithm, or if the additional changes needed to obtain Adam are the key drivers of its performance.

1.1 Contributions

Our goal is to empirically establish whether the performance gap between SGD and Adam on transformer models as studied by J. Zhang et al. (2020b) is primarily caused by heavy-tailed noise, or whether it can be explained by the behavior of the algorithms without noise, and to highlight the algorithmic differences that lead to this gap. Our experiments show that

1. **Noise is not the key contributor to the performance gap between SGD and Adam.** Removing noise by running in full batch does not reduce the gap, as shown in Figure 2.

2. **Plain SGD benefits less from a reduction in noise than Adam.** Increasing the batch size actually increases the gap between the methods when controlling for the step-size and number of iterations, as shown in Figure 3.

3. **Sign descent can close most of the gap between SGD and Adam in full batch.** While sign descent performs poorly with a small batch size, it improves drastically with larger batch sizes and can bridge most of the gap between gradient descent and Adam, as shown in Figures 4 and 5.

The fact that the performance gap between SGD and Adam grows when noise is removed suggests that the benefit of Adam over SGD is deterministic in nature. Studying the full batch setting might provide insights on this behavior. Our experiments further suggest that the performance of Adam and sign descent become closer as the batch size grows. While momentum improves the performance of both methods, they are similar when run with- or without momentum.

Despite sign descent being one of the key motivation behind RMSprop, and Adam by extension, the similarity to sign descent in full batch had not been shown empirically. Although this does not give an explanation for why Adam outperforms SGD, it might give an indication of how. As sign descent is a simpler algorithm, understanding its behavior in the deterministic setting might prove a useful tool to understand the success of Adam and the properties of the problem classes where it works well.

2 EXPERIMENTAL DESIGN

As experiments in large or full batch settings come with computational challenges, we touch on the key design decisions necessary to interpret the results. References and details are given in Appendix A.

Problems. We consider two vision and three language tasks with varying model sizes.

- Image classification on MNIST and CIFAR-10 using a small CNN and ResNet18.
- Language modeling on PTB and WikiText-2 using a small transformer and Transformer-XL.
- Question answering on SQuAD by fine-tuning a pre-trained DistillBERT model.

Focusing on the heavy-tail hypothesis of J. Zhang et al. (2020b), our attention is on the behavior of the optimizers on language tasks. The image tasks are included primarily for reference. We observe similar trends on both model classes, but the gap is larger in language tasks with transformer models.

Training performance. Our goal is to study optimization dynamics in cases where SGD fails to make sufficient progress. Our results focus on the training loss, and we use it to select hyperparameters. Results on hold-out data are given in Appendix C, but generalization at large batch sizes is known to require additional tuning and regularization (Shallue et al., 2019; Geiping et al., 2022).

Increasing batch sizes and iteration counts. To check if larger batch sizes and more accurate gradient estimates reduce the gap between SGD and Adam, we measure progress per iteration rather than per epoch. While keeping the number of optimization steps constant across batch sizes would be ideal, it is computationally infeasible given our budget.[2] As increasing the batch size can lead to better performance in fewer iterations, we run larger batch sizes for fewer steps. We set the number of epochs so that the number of iterations from one batch size to the next stays within a factor of 2–8$\times$. For example, if one setting runs 100 epochs with a batch size of 5% of the data, for 2000 iterations, and the next uses a batch size of 100%, we run it for $\approx$500 epochs/iterations. We observe a smooth transition in the performance of the optimizers across settings, and Adam with momentum reaches similar performance across batch sizes, suggesting the number of steps is appropriate.

Batch sizes. The batch sizes indicated by S, M, L and XL correspond to a $4\times$ relative increase. For the full batch runs, we remove a small percentage of the data ($\leq 0.5\%$) to ensure the dataset could be divided in even batches and run in full batch on the reduced data.

Simple training procedure. We use a constant step-size tuned by grid search to provide interpretable results. While better performance can be obtained with step-size schedules, our primary objective is to gain insight on the behavior of simple updates rather than reproduce state-of-the-art performance.

Hyperparameter tuning. Increasing the batch size can allow for larger step-sizes to be stable, and previous work has reported that the best step-size across batch size settings need not be monotonic (Shallue et al., 2019). To account for these effects, we tune the step-size for each setting individually by grid search. We start with a sparse grid of powers of 10, e.g. $10^{-4}, 10^{-3}, ..., 10^{0}$. After an initial run, we increase the density around the best step-size to half-powers, e.g. $10^{-4}, 10^{-3.5}, ..., 10^{0}$. We repeat those for 3 random seeds to get a measure of variation. Grid-search results are given in Appendix C. We use each method with and without momentum, indicated by ($+$m) and ($-$m). When using momentum, we fix it to 0.9 (β for SGD, β_1 for Adam). Other parameters are set to defaults.

3 BEHAVIOR OF SGD AND ADAM IN LARGE BATCH

Comparing SGD and Adam in small and full batch in Figures 1 and 2, we observe the following.

3.a **Adam outperforms SGD by a large margin on language tasks,** as expected from previous observations (G. Zhang et al., 2019),

3.b **Despite removing stochasticity, the gap between SGD and Adam is not reduced.** It even increases on some problems. On CIFAR-10 and PTB, the performance per iteration of Adam improves while the performance of SGD worsens on PTB and WikiText-2.

3.c **The importance of momentum is increased in full batch.** Momentum has a limited effect in small batch, especially for Adam and SGD on PTB or Adam on WikiText-2. The gap between

[2] For example, running Transformer-XL on WikiText-2 with a batch size of 20 for 40 epoch (32 600 iterations) takes 1–2 hours. Running 320 epochs in full batch (320 iterations) takes 8–10 hours. Doing 100$\times$ more iterations would take over a month, making hyperparameter tuning infeasible.

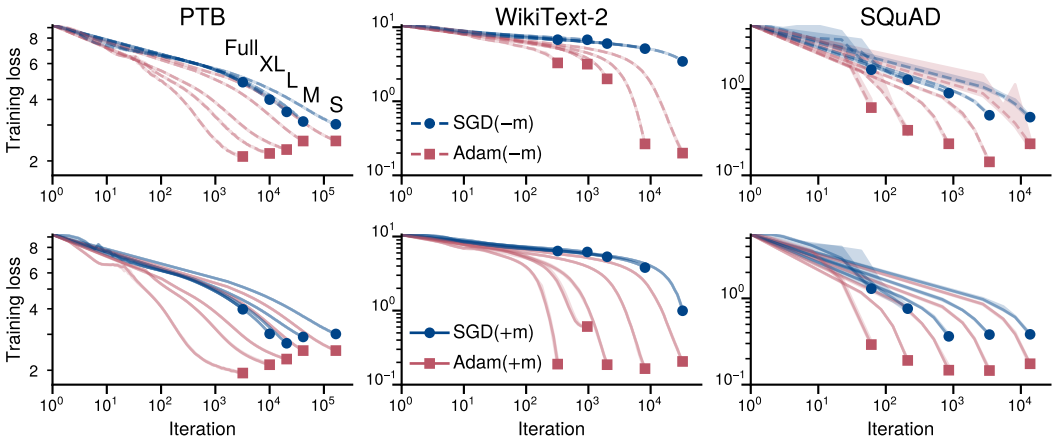

Figure 3: **SGD does not take advantage of the reduction in noise while Adam does,** and momentum helps both optimizers scale better with batch size. Adding intermediate batch size settings to interpolate between the small setting of Figure 1 and the full batch setting of Figure 2. Each line corresponds to one batch size setting and the marker (● , ■) highlights its performance at the end of the run. The lines correspond to a batch size of (left to right): Full, XL, L, M, S (see Appendix A.2). **Top/bottom:** results without/with momentum.

the same optimizer with and without momentum is larger in full batch. But it is not a sufficient solution: SGD struggles to make progress on WikiText-2 in full batch, with or without momentum.

4 Comparison across batch sizes

It is not clear from Figures 1 and 2 alone that the small and full batch setting are comparable. While Adam is still able to obtain a similar training loss (smaller on CIFAR-10, PTB), the number of iterations allocated to each optimizer remains a possible confounder. Although run for more epochs, the optimizers take far fewer steps in the full batch setting ($\approx 1/100$). To validate the results, we introduce intermediary batch sizes to interpolate the number of steps. To further support the findings of Figure 1–2, in Figure 3 we show the results of varying the batch size on a log-log scale to compare runs that differ in total number of iterations. We focus here on the language tasks; results on image tasks show similar trends, although less clearly, in Appendix B.

4.a **Plain SGD without momentum barely improves with batch-sizes.** Most batch size settings achieve a similar loss at a similar iteration counter. Without knowledge of the increase in batch size, this behavior would be consistent with running the same optimizer on the same batch-size, but terminating earlier. Adam, on the other hand, does improve with batch size, in the sense that it can reach a similar or even lower training loss in fewer iterations.

4.b **Momentum helps both SGD and Adam scale better to larger batch sizes.** This holds across optimizers, but its importance depends on the problem. It helps SGD scale with batch size better on PTB and SQuAD, and seems crucial for the performance of Adam on WikiText-2, althought the magnitude of the effect might be a side-effect of the visualization.[3]

These results give a more detailed view of how the gap between the two optimizers increases with batch size, and is greatest at full batch. SGD does not benefit from the more precise estimate of the gradient. Increasing the batch size but keeping the same epoch budget reduces the performance of SGD by cutting the number of iterations it can perform, suggesting that *stochasticity is not the limiting factor in the performance of SGD on these training problems.*

This is consistent with previous studies on batch size scaling, which show that momentum (Shallue et al., 2019) and Adam (G. Zhang et al., 2019) scale better to larger batch sizes. Our experiments

[3]We may not have run long enough on WikiText-2 to see the full improvement from large batch sizes for Adam without momentum. We expect the gap between SGD and Adam to grow even further as we run it longer, based on the other datasets and on the trajectories of the three large batches (L, XL, Full) stopping at a bend.

provide additional evidence on language tasks using simple to large architectures, and also show that these conclusions hold when considering only the training performance.

5 ALTERNATIVE INTERPRETATIONS AND DETERMINISTIC ASSUMPTIONS

The empirical behavior observed here and in previous studies does not seem consistent with the hypothesis that heavy-tailed noise impedes the progress of SGD, or that the better behavior of Adam comes from a more robust estimator of the descent direction. Rather, these results suggest that Adam is better than SGD in the deterministic setting and its performance degrades smoothly with added noise at smaller batch sizes. This suggests that we should look for algorithmic changes or assumptions that would show the benefits of Adam over SGD in the deterministic setting.

Algorithmic differences: sign-based approaches and normalization. A common view of Adam is that it is a form of smoothed sign descent (Balles and Hennig, 2018; Bernstein et al., 2018). Indeed, the motivation for RMSprop (Tieleman and Hinton, 2012) was to make sign descent more stable in the stochastic setting. The update of Adam and RMSprop also reduces to sign descent when setting the hyperparameters β_1, β_2, and ϵ to 0. But RMSprop originated as a comment in lecture notes. Why sign descent should work without stochasticity was not formally established, and the link from RMSprop to the original Rprop algorithm (Riedmiller and Braun, 1993) is also vague as Rprop maintains additional step-sizes based on subsequent signs. Whether the proposed modifications in RMSprop succeed in "stabilizing" sign descent has also not been explored. However, the idea that normalization across coordinates is supported by other studies of training dynamics in transformers. Liu et al. (2020, §3.2, Fig. 5) attribute the success of Adam to updates of the same magnitude across weights, despite large differences in the magnitude of the gradient across weights, which is consistent with the interpretation of Adam as sign descent.

Other approaches that have been found to help SGD such as clipping (Pascanu et al., 2013) have also been connected to Adam and sign-based methods. The element-wise variant of clipping is equivalent to taking the sign if all elements are larger than the clipping threshold (Crawshaw et al., 2022).

Theoretical justifications for clipping and normalized methods. A recent development for the analysis of such methods is the *relaxed smoothness assumption* of J. Zhang et al. (2020a) to explain the benefits of gradient clipping. Rather than assuming the Hessian is uniformly bounded ($\|\nabla^2 f\| \leq L$), they assume that the Hessian can grow with the gradient norm ($\|\nabla^2 f\| \leq L_0 + L_1\|\nabla f\|$) and provide empirical evidence that the Hessian and gradient norms can share a similar scale (J. Zhang et al., 2020a, Figure 1). This hypothesis provides a justification for normalizing large gradients by connecting the gradient norm to the Hessian.[4] This suggests that if the gradient is small, the function behaves like a smooth function and gradient descent will work. But if the gradient is large, the Hessian might be large. Dividing by the upper bound on the Hessian, the gradient norm, normalizes the step and leads to clipping. This view of gradient clipping has been further explored and the analysis improved by Qian et al. (2021) and B. Zhang et al. (2020).

Theoretical justifications for element-wise clipping and sign-based methods. Our results indicate that clipping alone is insufficient to bridge the gap between SGD and Adam. This is consistent with the previous work of J. Zhang et al. (2020a), who propose an adaptive, coordinate-wise clipping algorithm based on the assumption that the relaxed smoothness holds coordinate-wise. Crawshaw et al. (2022) build upon this assumption and propose a "generalized sign descent" algorithm as an alternative to clipping, a version of sign descent inspired by Adam which uses exponential moving averages to estimate the sign.

5.1 NORMALIZED GRADIENT DESCENT AND SIGN DESCENT

There interpretations raise the question of *whether improved performance of Adam in full batch can be attributed to a simpler algorithm like normalized gradient descent or sign descent.* Are the algorithmic differences between those methods, such as moving averages or bias-correction, mostly improving their performance in the stochastic setting? Or do they provide meaningful benefits even in the deterministic setting for deep learning models? To the best of our knowledge, the idea that RMSprop and Adam improve on sign descent by increasing its reliability to noise has not been

[4]This is reminiscent of previous assumptions in online learning that use gradient information to build preconditioners. One example is the Online Newton method of Hazan et al. (2006) for exp-concave function, where $f'' \geq \alpha(f')^2$ for any directional derivative, used as an upper- rather than lower-bound.

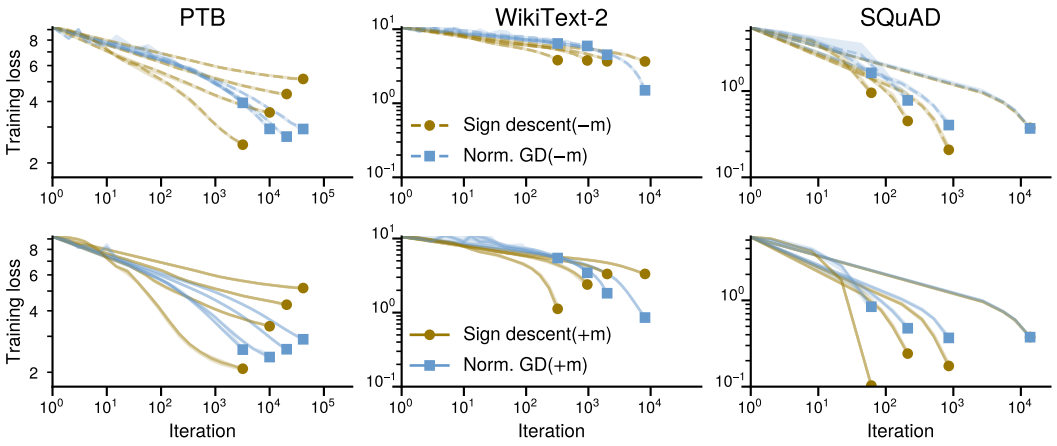

Figure 4: **Sign descent improves with batch size.** While normalized GD outperforms sign descent in small batch sizes, the relationship switches in full batch and sign descent outperforms normalized GD. Normalized gradient descent also scales better with batch size than plain gradient descent (compare Figure 3), but the method that improves most with batch size is sign descent. Each line corresponds to one batch size setting and the marker (● , ■) highlights its performance at the end of the run. The lines corresponds to a batch size of (left to right): Full, XL, L, M (see Appendix A.2). **Top/bottom:** results without/with momentum.

empirically verified. There is also limited evidence that sign descent alone performs well in the stochastic setting, as Adam is almost always preferred to it in deep learning, and it is rarely used in practice outside of the distributed setting (Bernstein et al., 2018).

To gain insights on this question, we repeat the experiments performed on SGD and Adam on normalized variants of gradient descent. We include sign descent and $\|\cdot\|_2$-normalized gradient descent (normalized GD) to have one scheme that changes the magnitude of the update but not its direction. Writing $h(g)$ for their transformation of the gradient g,

$$\text{Normalized GD:} \quad h(g) = g/\|g\|_2, \qquad \text{and} \qquad \text{Sign descent:} \quad h(g) = \text{sign}(g),$$

We implement their updates with heavy-ball momentum by accumulating the transformed gradient;

$$m_{t+1} = \beta m_t + h(g_t), \qquad x_{t+1} = x_t - \alpha m_{t+1}.$$

As in the previous experiments, we do not tune the momentum parameter and present results with ($\beta = 0.9$) and without ($\beta = 0$) momentum. Reproducing the visualization of Figure 3 for the normalized variants in Figure 4, we observe the following;

5.a **Normalized GD without momentum has a similar scaling problem as SGD.** Contrary to the behavior observed in Adam, higher batch sizes do not lead to improved performance. As with other optimizers, momentum helps normalized GD scale better to larger batch sizes, and it scales better to full batch than SGD.

5.b **Sign descent performs poorly at small batch sizes but gets closer to Adam in full batch.** At small batch sizes, the performance of sign descent can be worse than that of plain SGD. However, the improvement from increasing batch sizes is greater than for other optimizers, and sign descent goes from being the worst optimizer to being competitive with Adam in full batch.

Similar trends can also be observed for image models, shown separately in Figure 6, Appendix B, where sign descent with momentum can even outperforms Adam in full batch. The performance improvement with increased batch size and the final performance of Adam resembles that of sign descent. Normalized GD alone is not sufficient to bridge this gap. To better illustrate that the performance of sign descent is poor in small batch but closes most of the gap between GD and Adam in full batch, we compare the behavior of all optimizers with momentum in a traditional loss per epoch format in Figure 5. Sign descent however still improves on SGD without momentum, as shown in in Figure 8, Appendix B.

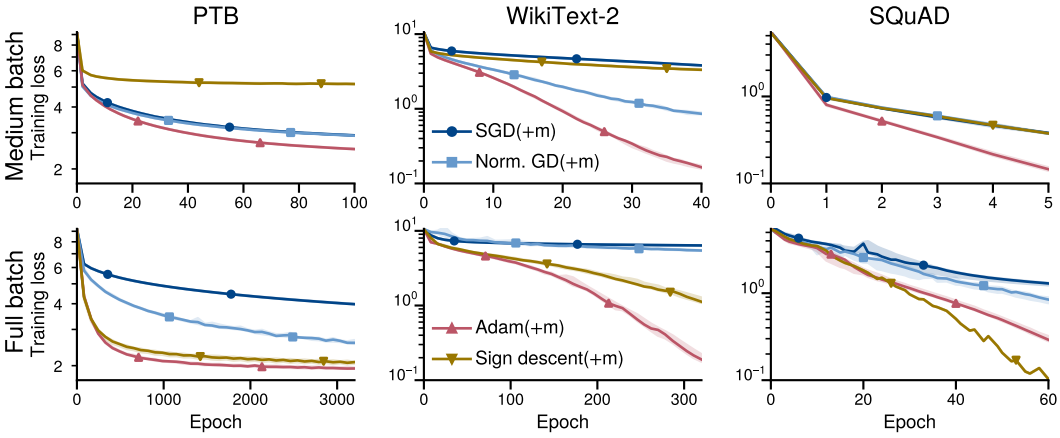

Figure 5: **Sign descent can close most of the gap between GD and Adam in full batch.** At small batch sizes, the performance of sign descent is worse than that of normalized SGD, and worse than plain SGD on PTB. However, the improvement when increasing the batch size is greater than for other optimizers, and sign descent goes from being the worst optimizer to being competitive with Adam in full batch on PTB, and closes half the gap on WikiText-2. Although all optimizers benefit from momentum, we observe a similar behavior of closing the gap with Adam in full batch without momentum, as shown in Figure 8, Appendix B. **Top/Bottom:** performance with a medium batch size/in full batch.

Sign descent and Adam share similar performance when both are run with- or without momentum on PTB, sign descent closes part of the gap on WikiText2, but the similarity in performance is limited to the non-momentum case on SQuAD (in Figure 8). While a gap still exists on some problem, this supports the hypothesis that the performance gap between Adam and gradient descent has its roots in a difference in the deterministic case, and that parts of the benefit of the second-moment estimation in Adam or RMSprop can be attributed to the difference between gradient descent and sign descent. The good scaling properties of Adam to large batch sizes are also shared with sign descent. This connection might provide a new avenue for analysis, by studying a simpler algorithm, to understand the behavior of Adam and the impact of noise. It also suggests that when training in a distributed system, sign descent might replace Adam with minimal loss in performance, while reducing the communication cost to one bit per dimension (Bernstein et al., 2018).

6 Discussion

We present experimental evidence that runs counter to the heavy-tailed hypothesis, which proposes that Adam outperforms SGD on transformers due to a more robust estimate of the descent direction. Rather, our experiments suggest that Adam outperforms gradient descent in the full batch setting, where gradient descent still performs poorly. We also show that the better scaling with large batch sizes exhibited by Adam, also observed by previous work (G. Zhang et al., 2019), is shared by sign descent. However, sign descent performs poorly in small batch sizes. This suggests that the benefits of the algorithmic differences between sign descent and Adam improve its behavior in small batch sizes. This validates the intuitive motivations for RMSprop, and by extension Adam, provided by Tieleman and Hinton (2012). From these observations, a promising direction to better understand those methods would be to separate the analyses of the algorithms along stochasticity; Why does sign descent work so well in the full batch setting on those models? How does Adam make it robust to noise in the small batch setting? We conclude by discussing the limitations of our approach and potential research questions that arise from our observations.

6.1 Limitations

A major limitation whenever probing the full batch setting is computation costs. As previously observed by large batch scaling studies (Shallue et al., 2019; G. Zhang et al., 2019) the critical batch size—the point at which increasing the batch size no longer leads to a linear speedup in training time—is much less than the total dataset size. It is inefficient to run those algorithms in such large

batch sizes for standard training purposes, and we include a comparison of training vs. time in Appendix B to illustrate this point. This might mean that compared to state-of-the-art workloads, our approach can only probe the "beginning" of the training procedure. Similarly, we used a constant step-size and did not explore step-size schedules. It is possible that the difficulties encountered by SGD on transformers might be mitigated by using a warm-up or decaying step-size schedule if the region of initialization has a very different landscape from the remainder of the optimization path.

We also do not attempt to optimize hyperparameters and stopping criteria for validation error. We instead focus on understanding the behavior of the optimizers on the training loss. For practical applications, where generalization is the focus, those results might be less applicable. However, the validation metrics for the runs selected by grid-search (available alongside our grid search validation in Appendix C) suggests that optimizers which perform better on training error achieve a lower testing loss faster (before over-fitting). Our conclusions might thus generalize to validation error by selecting hyperparameters based on validation error and using early stopping.

Finally, we wish to note that the clear difference between SGD and Adam on language tasks need not extrapolate to other architectures. On the two image tasks included to serve as a comparison point and the fine-tuning task starting from a pretrained model, the distinction between SGD and Adam and the trends are less clear. For example, SGD behaves differently on ResNet18/CIFAR10 in Figure 6, and improves with batch size as much as Adam, which might be due to this network using Batch Normalization. We might need different assumptions, analysis tools and possibly optimizers for different families of architectures that have qualitatively different behaviors.

6.2 Implications for future work

What architecture choices lead to the increased gap between SGD and Adam? Reducing the complexity of the models studied here to a minimal example, on the level of a linear regression, but still exhibiting a large gap between SGD and Adam, would be helpful to serve as a mental model and benchmark to test hypotheses or optimizers. For example, overparametrized linear models using logistic regression or exponential loss are known to converge faster with normalized gradient descent (Nacson et al., 2019) and satisfy the relaxed smoothness assumption of J. Zhang et al. (2020a). While those models do not explain why sign descent would be beneficial, they might still suggest a possible culprit for this behavior. Language models end with a prediction task over a large vocabulary—much larger than typical classification problems seen in image tasks—which might be enough to reproducible on toy data with a single attention layer.

For what function classes does sign descent work? The standard assumption for the convergence of sign descent is smoothness in $\|\cdot\|_\infty$ or $\|\cdot\|_2$ (Bernstein et al., 2018), and does not hold for neural networks. The relaxed smoothness assumption of J. Zhang et al. (2020a) provides a justification for clipping the magnitude of the gradient, and both J. Zhang et al. and Crawshaw et al. (2022) use a coordinate-wise version to analyse element-wise clipping or an exponential moving average form of sign descent. But it is not clear how helpful or realistic the coordinate-wise version of this assumption is. Checking this variant would require checking every weight in the network, and the convergence results then depends on many problem specific-constants, two per dimension of the model. A relaxed smoothness assumption in non-Euclidean norm, of the form $\|\nabla^2 f(x)\|_* \leq A + B\|\nabla f(x)\|$ as in the work of Balles et al. (2020) might be more interpretable.

batch size scaling laws for sign descent? In our experiments, sign descent exhibits the largest improvement with an increased batch size. This result is difficult to explain with existing models used to determine batch size scaling, such as the noisy quadratic model of G. Zhang et al. (2019). Their interpretation of Adam is as a preconditioned method similar to KFAC (Martens and Grosse, 2015), an approximate second-order method. But this interpretation does not apply to sign descent on quadratics. The noisy quadratic model cannot explain the similarity in the performance improvement observed by increasing the batch size for both Adam and sign descent. A model explaining both might be obtained by relating the gradient and the Hessian, as in the relaxed smoothness assumption of J. Zhang et al. (2020a) or similar assumptions used in online learning (Hazan et al., 2006).

Designing versions of sign descent that are more robust to small batch sizes? This also calls for a better understanding of how to reduce the variance of sign-based methods. Sign descent is often analysed in the distributed setting due to its natural compression ability, as it only requires 1 bit per dimension, such as in previous works by Bernstein et al. (2018) or Safaryan and Richtárik (2021) showing that variants of sign descent based on majority voting or momentum can improve on plain

sign descent. But it is unclear whether the standard assumptions used for the analysis of stochastic methods describe the type of noise encountered in deep learning models. A detailed look into the type of error induced by subsampling might help develop better approaches.

REFERENCES

Kwangjun Ahn, Jingzhao Zhang, and Suvrit Sra (2022). "Understanding the unstable convergence of gradient descent". In: *International Conference on Machine Learning, 2022*. Vol. 162. PMLR, pp. 247–257.

Ahmet Alacaoglu, Yura Malitsky, Panayotis Mertikopoulos, and Volkan Cevher (2020). "A new regret analysis for Adam-type algorithms". In: *International Conference on Machine Learning, 2020*. Vol. 119, pp. 202–210.

Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer (2019). "Memory Efficient Adaptive Optimization". In: *Advances in Neural Information Processing Systems 2019*, pp. 9746–9755.

Sanjeev Arora, Zhiyuan Li, and Abhishek Panigrahi (2022). "Understanding Gradient Descent on the Edge of Stability in Deep Learning". In: *International Conference on Machine Learning, 2022*. Vol. 162. PMLR, pp. 948–1024.

Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). *Layer Normalization*. Presented at the NeurIPS 2016 Deep Learning Symposium. arXiv/1607.06450.

Lukas Balles and Philipp Hennig (2018). "Dissecting Adam: The Sign, Magnitude and Variance of Stochastic Gradients". In: *International Conference on Machine Learning, 2018*. Vol. 80, pp. 413–422.

Lukas Balles, Fabian Pedregosa, and Nicolas Le Roux (2020). *The Geometry of Sign Gradient Descent*. Preprint. arXiv/2002.08056.

Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar (2018). "SIGNSGD: Compressed Optimisation for Non-Convex Problems". In: *International Conference on Machine Learning, 2018*. Vol. 80. PMLR, pp. 559–568.

Jeremy M. Cohen, Simran Kaur, Yuanzhi Li, J. Zico Kolter, and Ameet Talwalkar (2021). "Gradient Descent on Neural Networks Typically Occurs at the Edge of Stability". In: *International Conference on Learning Representations, 2021,*

Michael Crawshaw, Mingrui Liu, Francesco Orabona, Wei Zhang, and Zhenxun Zhuang (2022). *Robustness to Unbounded Smoothness of Generalized SignSGD*. Preprint. arXiv/2208.11195.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov (2019). "Transformer-XL: Attentive Language Models beyond a Fixed-Length Context". In: *Annual Meeting of the Association for Computational Linguistics 2019*, pp. 2978–2988.

Alexandre Défossez, Léon Bottou, Francis Bach, and Nicolas Usunier (2020). *A Simple Convergence Proof of Adam and Adagrad*. Preprint. arXiv/2003.02395.

John C. Duchi, Elad Hazan, and Yoram Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12, pp. 2121–2159.

Jonas Geiping, Micah Goldblum, Phil Pope, Michael Moeller, and Tom Goldstein (2022). "Stochastic Training is Not Necessary for Generalization". In: *International Conference on Learning Representations, 2022*.

Eduard Gorbunov, Marina Danilova, and Alexander V. Gasnikov (2020). "Stochastic Optimization with Heavy-Tailed Noise via Accelerated Gradient Clipping". In: *Advances in Neural Information Processing Systems 2020*, pp. 15042–15053.

Elad Hazan (2019). *Introduction to Online Convex Optimization*.

Elad Hazan, Adam Kalai, Satyen Kale, and Amit Agarwal (2006). "Logarithmic regret algorithms for online convex optimization". In: *Learning theory*. Vol. 4005. Lecture Notes in Comput. Sci. Springer, Berlin, pp. 499–513.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016). "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition, 2016*, pp. 770–778.

Sergey Ioffe and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *International Conference on Machine Learning, 2015*. Vol. 37, pp. 448–456.

Diederik P. Kingma and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations, 2015*.

Alex Krizhevsky (May 2012). *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. University of Toronto.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998). "Gradient-Based Learning Applied to Document Recognition". In: *Proc. IEEE* 86.11, pp. 2278–2324.

Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han (2020). "Understanding the Difficulty of Training Transformers". In: *Conference on Empirical Methods in Natural Language Processing, 2020*, pp. 5747–5763.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz (1993). "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Computational Linguistics* 19.2, pp. 313–330.

James Martens and Roger B. Grosse (2015). "Optimizing Neural Networks with Kronecker-factored Approximate Curvature". In: *International Conference on Machine Learning, 2015*. Vol. 37, pp. 2408–2417.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher (2017). "Pointer Sentinel Mixture Models". In: *International Conference on Learning Representations, 2017*.

Mor Shpigel Nacson, Jason D. Lee, Suriya Gunasekar, Pedro Henrique Pamplona Savarese, Nathan Srebro, and Daniel Soudry (2019). "Convergence of Gradient Descent on Separable Data". In: *International Conference on Artificial Intelligence and Statistics, 2019*. Vol. 89. PMLR, pp. 3420–3428.

Zachary Nado, Justin Gilmer, Christopher J. Shallue, Rohan Anil, and George E. Dahl (2021). *A Large Batch Optimizer Reality Check: Traditional, Generic Optimizers Suffice Across Batch Sizes*. Preprint. arXiv/2102.06356.

Yurii Nesterov (2018). *Lectures on Convex Optimization*. Vol. 87. Springer Optimization and Its Applications. Springer.

Francesco Orabona (2019). *A Modern Introduction to Online Learning*. Lecture notes. arXiv/1912.13213v5.

Francesco Orabona (2020). *Neural Networks (maybe) Evolved to Make Adam the Best Optimizer*. Blog post. https://parameterfree.com/2020/12/06/.

Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio (2013). "On the difficulty of training recurrent neural networks". In: *International Conference on Machine Learning, 2013*. Vol. 28, pp. 1310–1318.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 2019*, pp. 8024–8035.

Jiang Qian, Yuren Wu, Bojin Zhuang, Shaojun Wang, and Jing Xiao (2021). "Understanding Gradient Clipping In Incremental Gradient Methods". In: *International Conference on Artificial Intelligence and Statistics, 2021*. Vol. 130. PMLR, pp. 1504–1512.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang (2016). "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Conference on Empirical Methods in Natural Language Processing, 2016*, pp. 2383–2392.

Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar (2018). "On the Convergence of Adam and Beyond". In: *International Conference on Learning Representations, 2018*.

Martin A. Riedmiller and Heinrich Braun (1993). "A direct adaptive method for faster backpropagation learning: the RPROP algorithm". In: *International Conference on Neural Networks(ICNN'88)*, pp. 586–591.

Mher Safaryan and Peter Richtárik (2021). "Stochastic Sign Descent Methods: New Algorithms and Better Theory". In: *International Conference on Machine Learning, 2021*. Vol. 139. PMLR, pp. 9224–9234.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf (2019). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. Presented at EMC^2, NeurIPS 2019. arXiv/1910.01108.

Christopher J. Shallue, Jaehoon Lee, Joseph M. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl (2019). "Measuring the Effects of Data Parallelism on Neural Network Training". In: *Journal of Machine Learning Research* 20, 112:1–112:49.

Vishwak Srinivasan, Adarsh Prasad, Sivaraman Balakrishnan, and Pradeep Kumar Ravikumar (2021). *Efficient Estimators for Heavy-Tailed Machine Learning*. Preprint. openreview:5K8ZG9twKY.

Tijmen Tieleman and Geoffrey Hinton (2012). *RMSPROP: Divide the gradient by a running average of its recent magnitude*. Lecture notes http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems 2017*, pp. 5998–6008.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush (2020). "Transformers: State-of-the-Art Natural Language Processing". In: *Conference on Empirical Methods in Natural Language Processing, 2020*, pp. 38–45.

Yang You, Jing Li, Sashank J. Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh (2020). "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes". In: *International Conference on Learning Representations, 2020*.

Bohang Zhang, Jikai Jin, Cong Fang, and Liwei Wang (2020). "Improved Analysis of Clipping Algorithms for Non-convex Optimization". In: *Advances in Neural Information Processing Systems 2020*, pp. 15511–15521.

Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George E. Dahl, Christopher J. Shallue, and Roger B. Grosse (2019). "Which Algorithmic Choices Matter at Which Batch Sizes? Insights From a Noisy Quadratic Model". In: *Advances in Neural Information Processing Systems 2019*, pp. 8194–8205.

Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie (2020a). "Why Gradient Clipping Accelerates Training: A Theoretical Justification for Adaptivity". In: *International Conference on Learning Representations, 2020*.

Jingzhao Zhang, Sai Praneeth Karimireddy, Andreas Veit, Seungyeon Kim, Sashank J. Reddi, Sanjiv Kumar, and Suvrit Sra (2020b). "Why are Adaptive Methods Good for Attention Models?" In: *Advances in Neural Information Processing Systems 2020*, pp. 15383–15393.

Supplementary material

Appendix A Experimental details
Appendix B Additional plots
Appendix C Grid-searches and performance of selected runs
Appendix D Additional figures

A Experimental details

Appendix A.1 Datasets and models
Appendix A.2 Batch sizes and epoch counts
Appendix A.3 Algorithms
Appendix A.4 Histograms
Appendix A.5 Grid search

The code to reproduce our experiments and data generated by each run will be made available.

A.1 Datasets and models

Problems in the main text are referred to by the dataset and corresponds to the following problems. For all problems, we use the default train/test split and report the performance on the test split as validation error.

MNIST, Modified LeNet5 (LeCun et al., 1998)

Image classification task using a modified LeNet5 architecture; a 3-convolution, 2-linear layer average pooling and tanh activations, using a linear rather than Gaussian output.

Dataset `yann.lecun.com/exdb/mnist/`

CIFAR10, ResNet18 (Krizhevsky, 2012; He et al., 2016)

PyTorch (Paszke et al., 2019) implementation of the 18-layer CNN with residual connections.

Dataset `www.cs.toronto.edu/~kriz/cifar.html`
Model `pytorch.org/vision/0.10/models.html#torchvision.models.resnet18`

PTB, Transformer (Marcus et al., 1993; Vaswani et al., 2017)

Word-level language modeling with sequences of 35 tokens using a simple Transformer model used as a tutorial example in PyTorch. The architecture consists of an 200-dimensional embedding layer, 2 transformer layers (2-head self attention, layer normalization, linear(200x200)-ReLU-linear(200x200), layer normalization) followed by a linear layer. Data processing uses the implementation of Dai et al. (2019).

Dataset `catalog.ldc.upenn.edu/LDC99T42`
Processing `github.com/kimiyoung/Transformer-XL`
Model `pytorch.org/tutorials/beginner/transformer_tutorial.html`

WikiText-2, Transformer-XL (Merity et al., 2017; Dai et al., 2019)

Word-level language modeling with sequences of length 128 with Transformer-XL, using the implementation of Dai et al. (2019) for the model and data preprocessing. Hyperparameters follow the ENWIK8 base experiment of Dai et al. (2019), except with the modifications of J. Zhang et al. (2020b), using 6 layers and a target length of 128.

Dataset `blog.salesforceairesearch.com/the-wikitext-long-term-dependency-language-modeling-dataset/`
Model `github.com/kimiyoung/Transformer-XL`

SQuAD v1.1, DistillBERT finetuning (Rajpurkar et al., 2016; Sanh et al., 2019)

Question-answering by fine-tuning a pretrained DistillBERT model using the HuggingFace (Wolf et al., 2020) implementation of the model and data processing.

Dataset `rajpurkar.github.io/SQuAD-explorer/explore/1.1/dev/`
Model `huggingface.co/transformers/v4.9.2/model_doc/distilbert.html`

Table 1: **Batch sizes used in experiments.** The batch size corresponds to the number of input-output samples. For image data, a sample correspond to an image-labels pair. For language models on PTB/WikiText-2, a sample corresponds to pairs of sequences of 35/135 tokens. For DistillBERT on SQuAD, a sample corresponds to a question/answer pair. The full batch runs that use $\geq 99.5\%$ of the entire dataset, drop 0.5% of the data and run in full batch on the remaining data at every epoch.

Dataset/Model		**Batch size**	**% data**	**Epochs**	**Iter./Epoch**	**Total Iter.**
MNIST	S	256	0.43%	100	234	23 400
Modified LeNet5	M	1 024	1.71%	100	58	5 800
	L	4 069	6.78%	200	14	2 800
	XL	16 384	27.31%	400	3	1 200
	Full	60 000	100.00%	800	1	800
CIFAR-10	S	64	0.13%	100	781	78 100
ResNet18	M	256	0.51%	100	195	19 500
	L	1 024	2.05%	100	48	4 800
	XL	4 096	8.19%	200	12	2 400
	Full	50 000	100.00%	800	1	800
PTB	S	16	0.06%	100	1 659	165 900
Transformer	M	64	0.24%	100	414	41 400
	L	256	0.96%	200	103	20 600
	XL	1 024	3.86%	400	25	10 000
	Full	26 520	99.85%	3 200	1	3 200
WikiText-2	S	20	0.12%	40	815	32 600
Transformer-XL	M	80	0.49%	40	203	8 120
	L	320	1.96%	40	50	2 000
	XL	1 280	7.84%	80	12	960
	Full	16 240	99.53%	320	1	320
SQuAD	S	32	0.04%	5	2 741	13 705
DistillBERT	M	128	0.15%	5	685	3 425
	L	512	0.58%	5	171	855
	XL	2 048	2.33%	10	42	420
	Full	87 680	99.96%	60	1	60

A.2 Batch sizes and epoch counts

Our goal is to compare the optimization performance while controlling for the number of iterations, but keeping the iteration count constant across batch sizes is computationally infeasible. For example, running Transformer-XL on WikiText-2 with a batch size of 20 for 40 epoch (32 600 iterations) takes us 1–2 hours. Running 320 epochs in full batch (320 iterations) takes 8–10 hours. Running $100\times$ more iterations would take over a month, making hyperparameter tuning infeasible. As increasing the batch size leads to better performance and convergence in fewer iterations, we run larger batch sizes for fewer iterations. To select the batch size and number of epochs, we start from a setting that achieves reasonable results using a constant learning rate at small batch sizes. We control how much the number of iterations changes from setting to setting to have comparable results. The batch sizes indicated by S, M, L and XL correspond to a relative increase of $4\times$ in batch size. We tune the number of epochs for each setting so that the number of total iterations from one batch size to the next stays within a factor of $\approx$2–8. The settings used in our experiments are described in Table 1.

For settings where computing the gradient would not fit into memory, we use gradient accumulation to reduce peak memory consumption (computing the gradients of smaller batches and averaging them). We use accumulation for Full batch runs, WikiText-2 ($\geq$ L) and SQuAD ($\geq$ M). Due to the batch normalization layers (Ioffe and Szegedy, 2015) in ResNet18, computing the full-batch gradient would require to pass the entire dataset through the network at once (the normalization makes it so that the average of the gradients of the minibatches is different from the full gradient). We use the average over the minibatch gradients with a batch size of 10 000. Transformer models use layer normalization (Ba et al., 2016) and are unaffected.

Algorithm 1 Gradient Descent and variants

Parameters: step-size α, momentum β.
Initialization: momentum buffer $m_0 = 0$.
At iteration t**:** given a stochastic gradient $\tilde{g}_t$,

$$d_t = \begin{cases} \tilde{g}_t & \text{Gradient Descent} \\ \tilde{g}_t / \|\tilde{g}_t\|_2 & \text{Normalized GD} \\ \operatorname{sign}(\tilde{g}_t) & \text{Sign Descent} \end{cases}$$

Update momentum m_t and iterates x_{t+1}

$$m_t = \beta m_{t-1} + d_t$$
$$x_{t+1} = x_t - \alpha m_t$$

Algorithm 2 Adam

Parameters: step-size α, momentum β_1, β_2.
Initialization: moments buffers $m_0, M_0 = 0$.
At iteration t**:** given a stochastic gradient $\tilde{g}_t$,

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\tilde{g}_t$$
$$M_t = \beta_2 m_{t-1} + (1 - \beta_2)\tilde{g}_t^2$$
$$x_{t+1} = x_t - \alpha \frac{m_t/(1 - \beta_1^t)}{\sqrt{\tilde{M}_t/(1 - \beta_2^t) + \epsilon}}$$

or, without bias correction,

$$x_{t+1} = x_t - \alpha \frac{m_t}{\sqrt{M_t + \epsilon}}$$

Using the default $\epsilon = 10^{-8}$.

A.3 Algorithms

We use the PyTorch (Paszke et al., 2019) implementation of SGD and Adam, and implement of the normalized variants (sign descent, normalized GD) with heavy-ball momentum following the pseudo-code in Algorithm 1. The pseudo-code of Adam is given in Algorithm 2.

For every algorithm, we tune the main step-size by grid search. The momentum parameters β in gradient descent variants and β_1 in Adam is set either to 0 or 0.9. This is indicated in the main text and figure legends by the suffix (+m) (with momentum) or (−m) (without). We leave the remaining hyperparameters of Adam to their default ($\beta_2 = 0.999, \epsilon = 10^{-8}$).

A.4 Histograms

The batch sizes used are in Table 1. The histograms are generated with a batch size of

MNIST	CIFAR-10	PTB	WikiText-2	SQuAD
256	64	16	16	16

As ResNet18 uses batch normalization, it does not have a well-defined "full" gradient (the average of minibatch gradients is not the gradient obtained by passing the entire dataset through the network). We use the average over the minibatch gradients. Other models are unaffected.

A.5 Grid search

We start with a sparse grid of integer powers of 10 (eg. $[10^{-5}, 10^{-4}, \ldots, 10^{0}]$). After an initial run to identify a reasonable region, we increase the density to include half-powers (eg. $[10^{-3}, 10^{-2.5}, 10^{-2}, 10^{-1.5}, 10^{-1}]$). The density of the grid is the same for all problems. We run those step-sizes for 3 random seeds determining the data ordering and initialization (except for DistillBERT on SQuAD, which is pretrained). To select the step-size, we minimize the (maximum over seeds of the) training loss at the end of training.

The final performance as a function of the step-size for the training loss and accuracy/PPL/F1-score (including on holdout data) are given in Appendix C

B ADDITIONAL PLOTS

B.1 VERSION OF FIGURE 3 INCLUDING IMAGE TASKS.

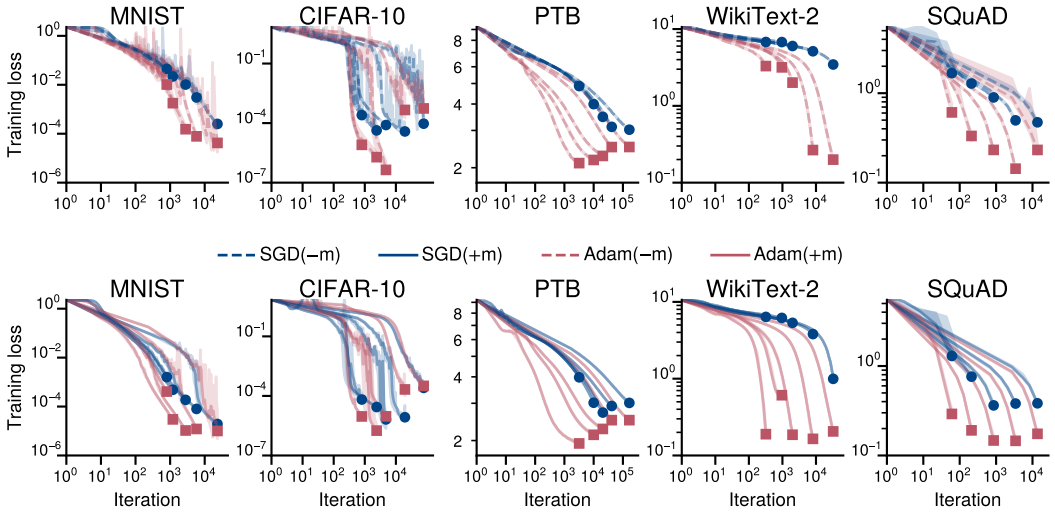

Figure 6: **SGD does not take advantage of the reduction in noise while Adam does,** and momentum helps both optimizers scale better with batch size. However, the trends are not as clear on MNIST and CIFAR-10. SGD performs better on CIFAR-10 than on other problems, which might be attributable to the presence of Batch Normalization layers. The lines correspond to a batch size of (left to right): Full, XL, L, M, S (see Appendix A.2). **Top/bottom:** results without/with momentum.

B.2 VERSION OF FIGURE 4 INCLUDING IMAGE TASKS.

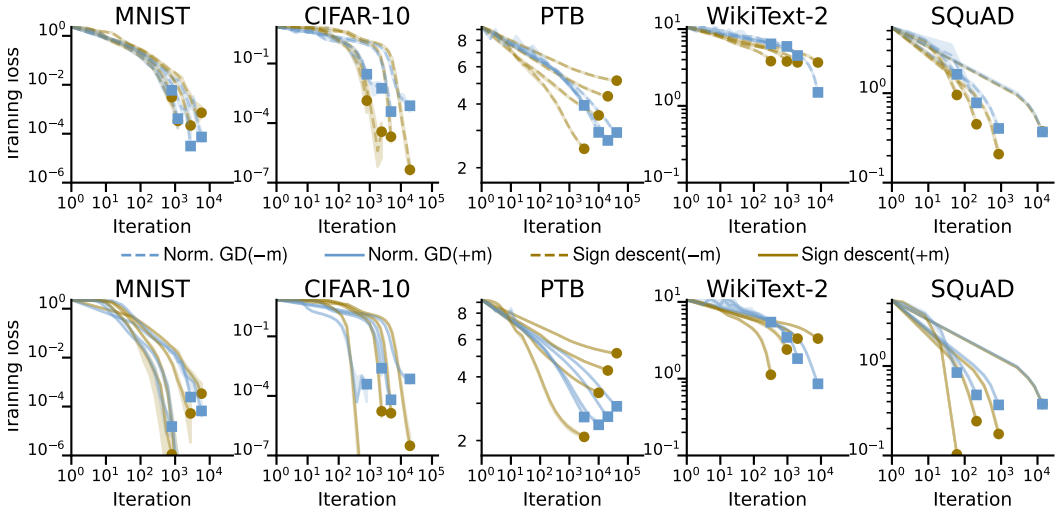

Figure 7: **Sign descent improves with batch size.** Normalized gradient descent also scales better with batch size than plain gradient descent (compare Figure 3), but less so than sign descent. The use of momentum seems more important on image tasks. Each line corresponds to one batch size setting and the marker (● , ■) highlights its performance at the end of the run. The lines correspond to a batch size of (left to right): Full, L, M (see Appendix A.2). **Top/bottom:** results without/with momentum.

B.3 VERSION OF FIGURE 5 WITHOUT MOMENTUM

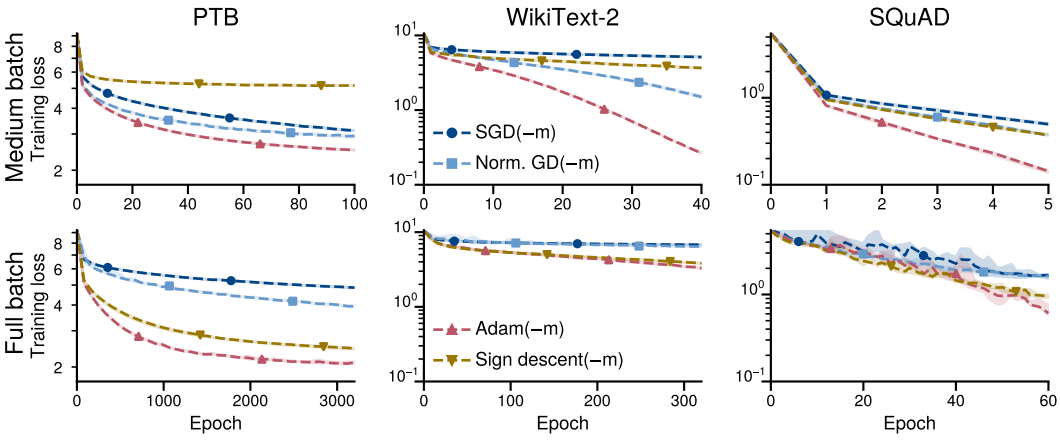

Figure 8: **Sign descent performs poorly with small batch sizes but closes most of the gap between GD and Adam in full batch.** The observation made on Figure 5 also hold without momentum. Although all optimizers suffer from the lack of momentum, especially on WikiText-2, sign descent closes most of the gap between SGD and Adam. **Top/Bottom:** performance with a medium batch size/in full batch.

B.4 COMPARISON OF TRAINING PERFORMANCE IN TIME

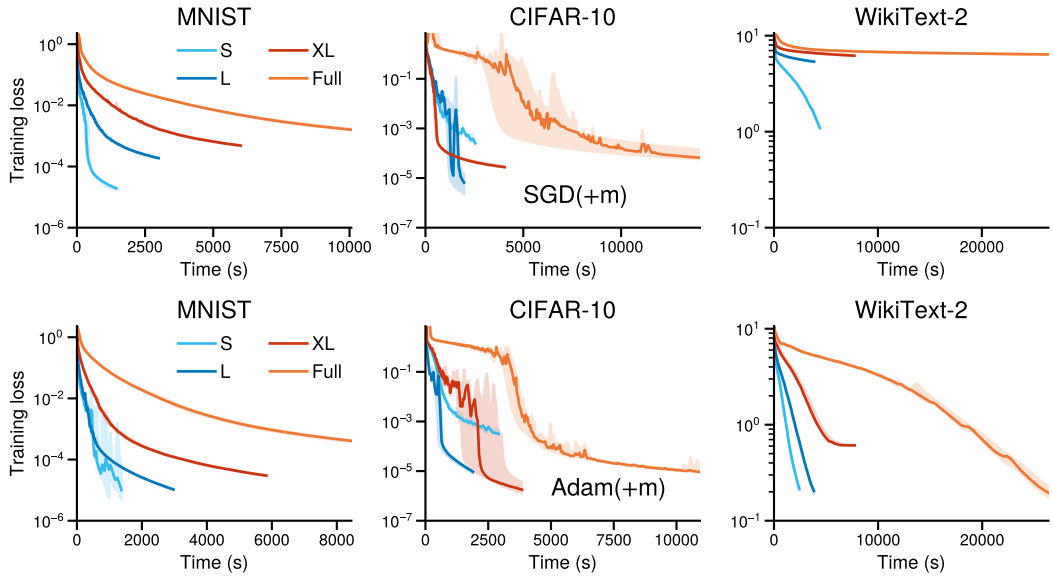

Figure 9: **Increasing the batch sizes need not lead to better progress per unit of time.** Plots for the performance of SGD (top) and Adam (bottom) with momentum.

C GRID-SEARCHES AND PERFORMANCE OF SELECTED RUNS

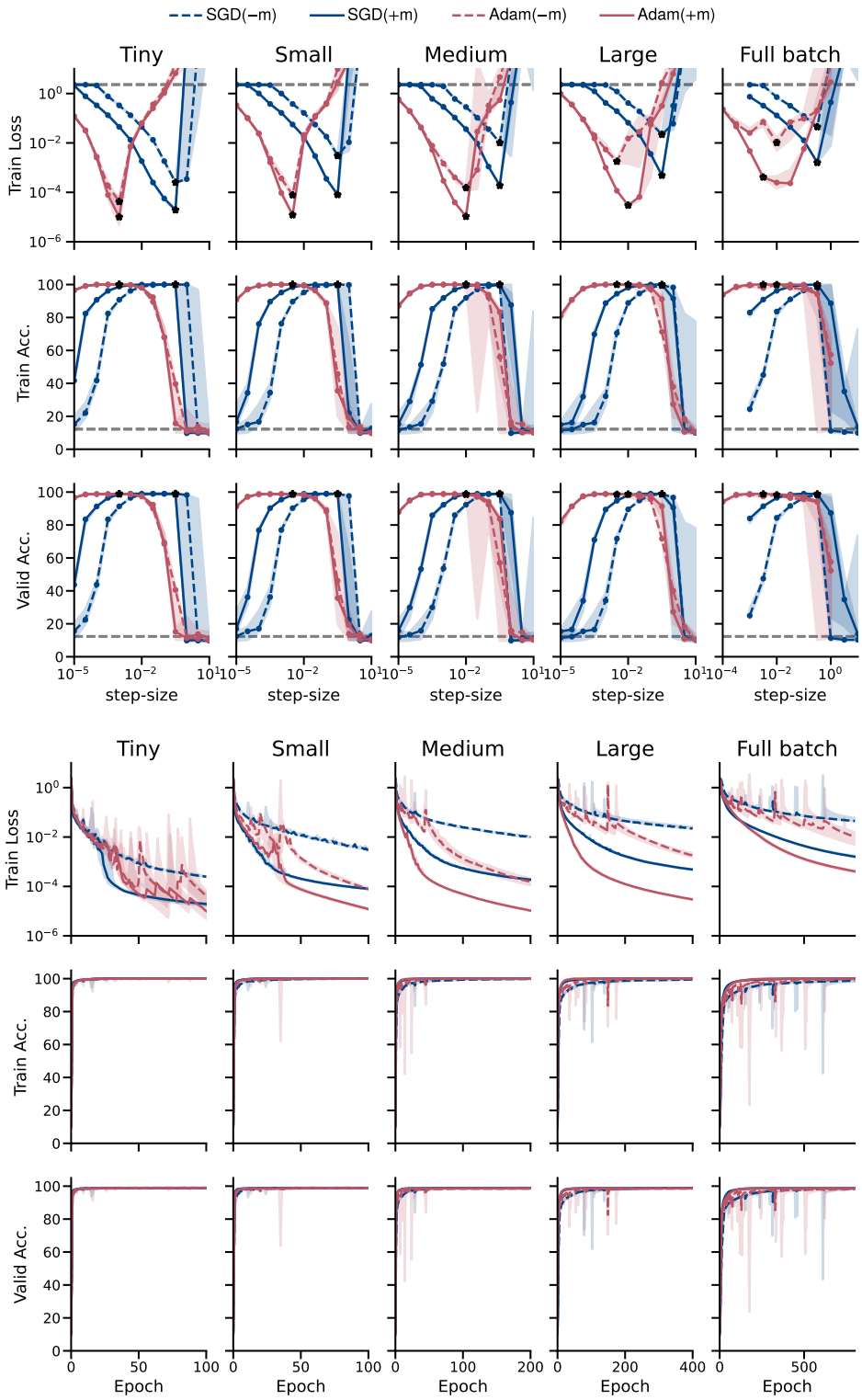

Figure 10: **Grid search and selected runs for modified LeNet5 on MNIST**.

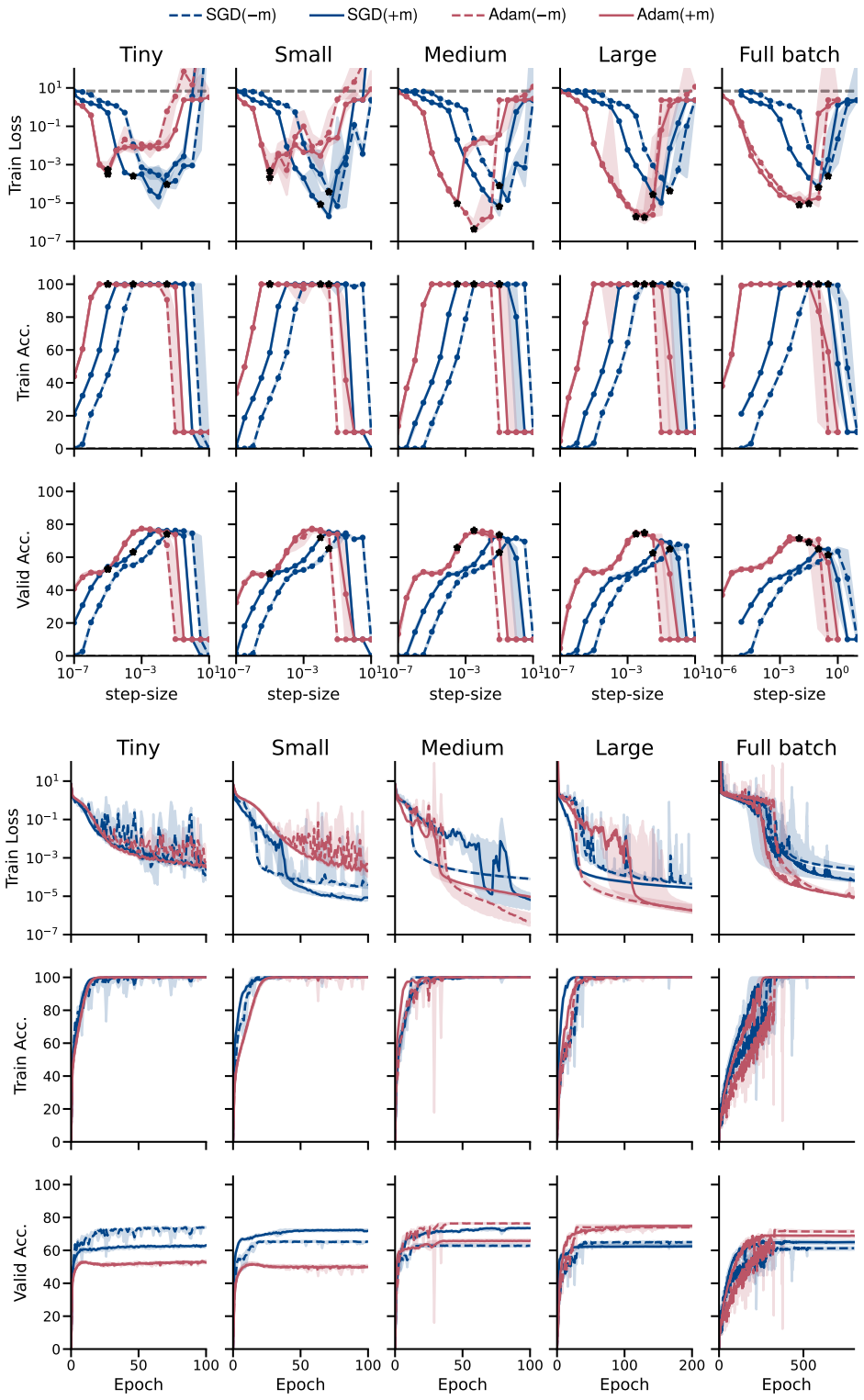

Figure 11: **Grid search and selected runs for ResNet18 on Cifar10**.

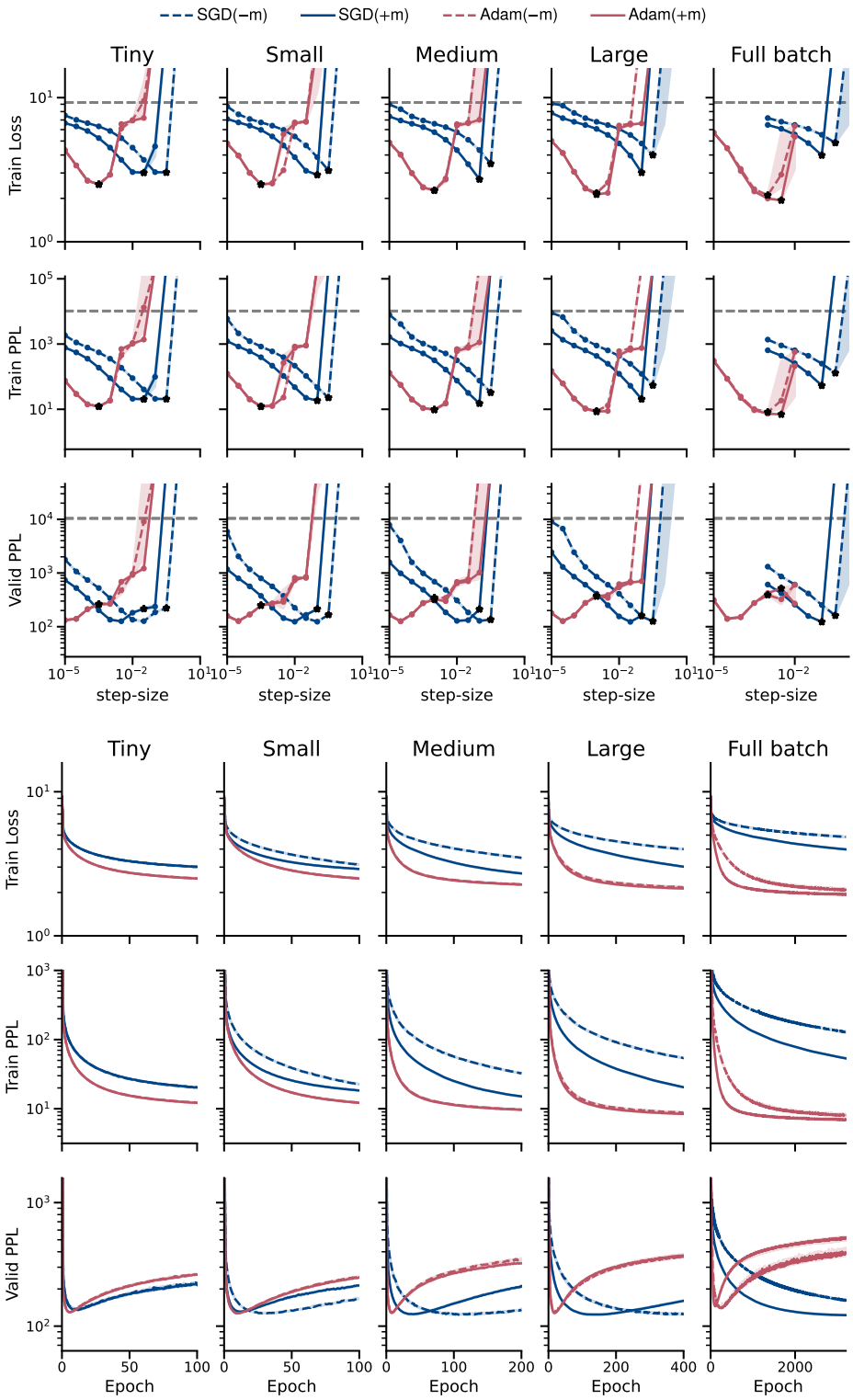

Figure 12: **Grid search and selected runs for Small Transformer on PTB**.

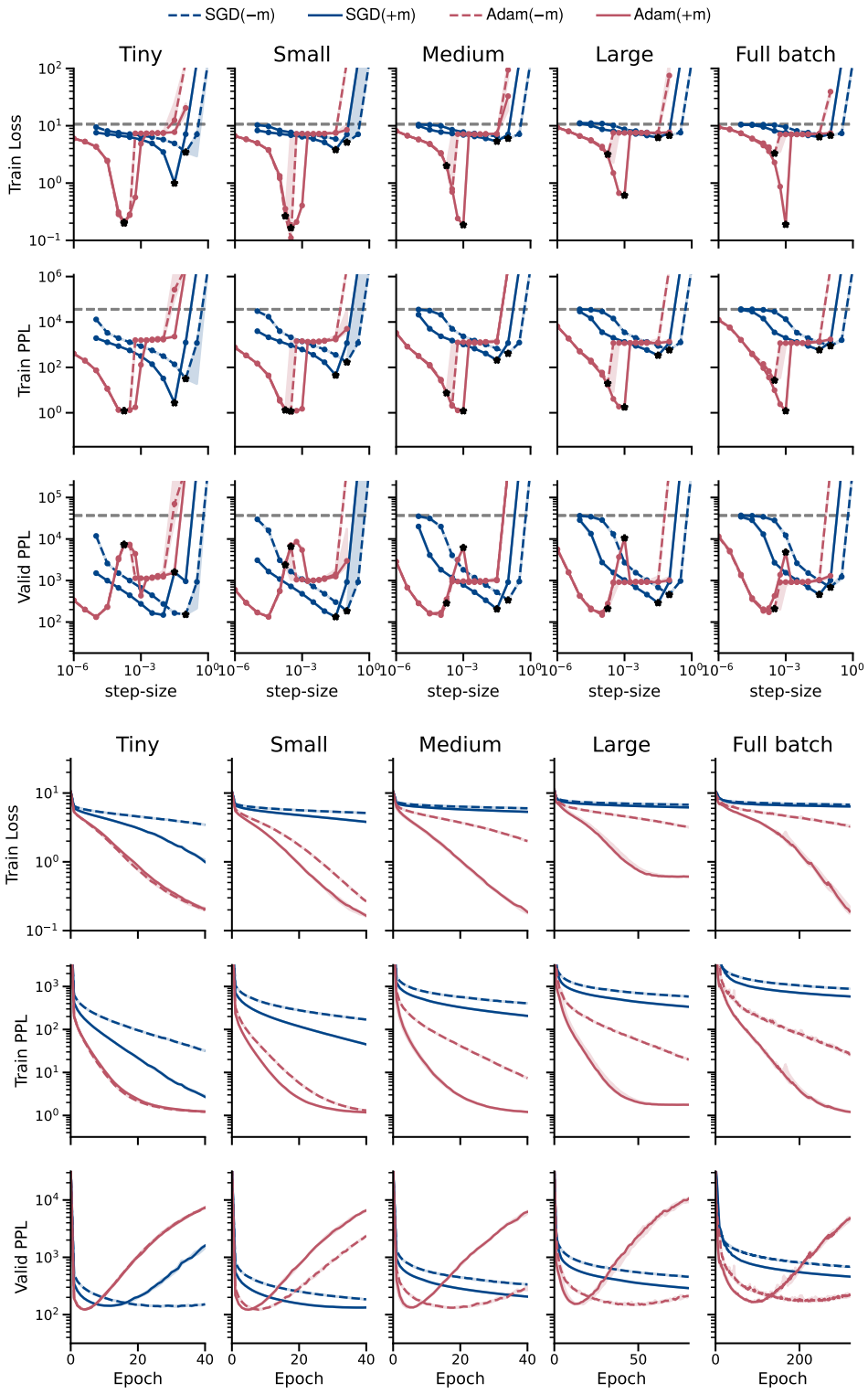

Figure 13: **Grid search and selected runs for Transformer-XL on WikiText-2**.

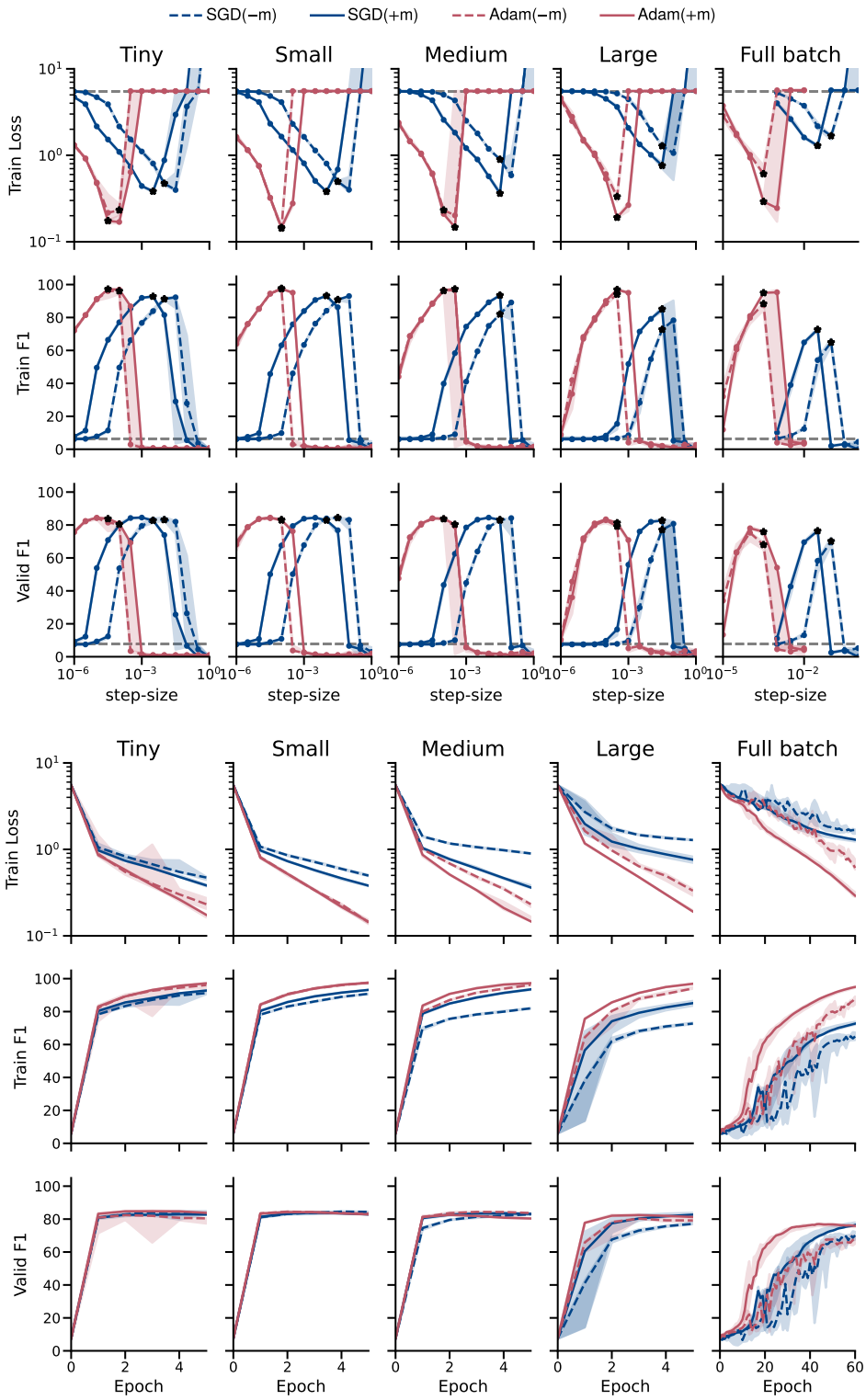

Figure 14: **Grid search and selected runs for DistillBERT finetuning on SQuAD**.

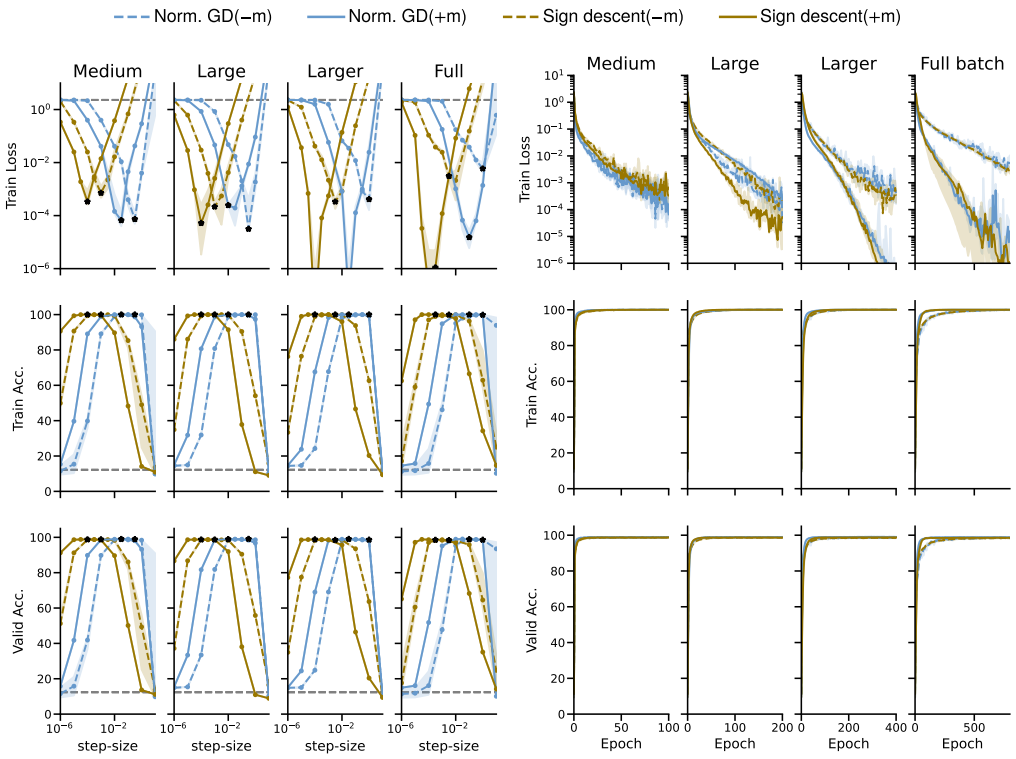

Figure 15: **Grid search and selected runs for modified LeNet5 on MNIST**.

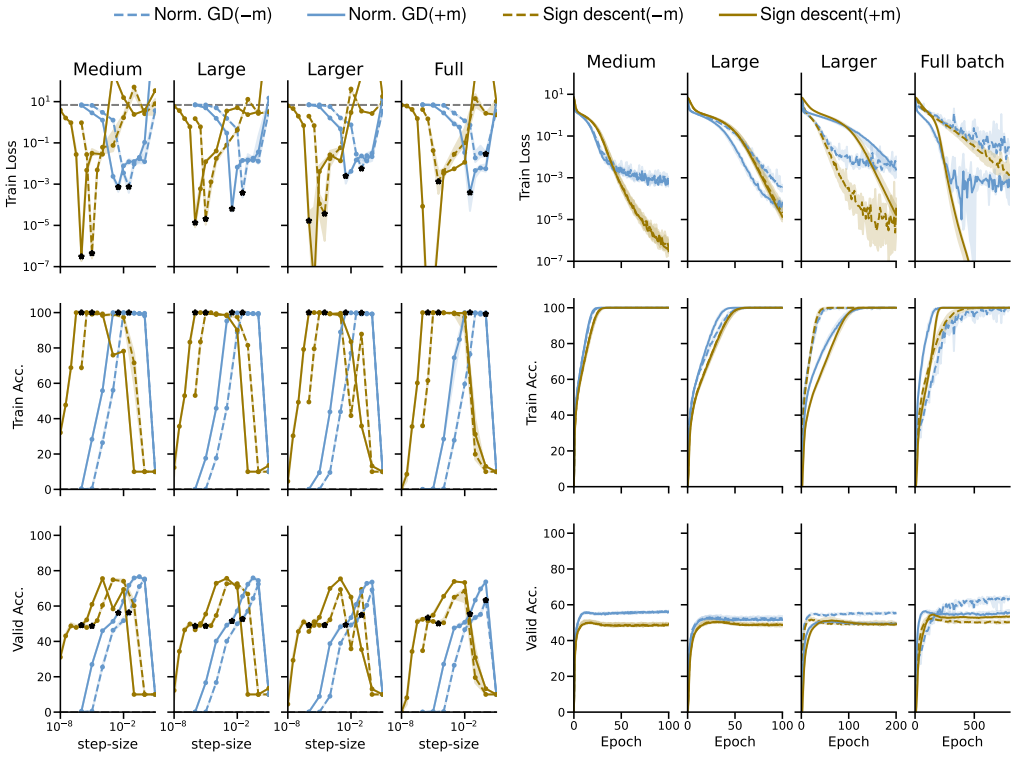

Figure 16: **Grid search and selected runs for ResNet18 on Cifar10**.

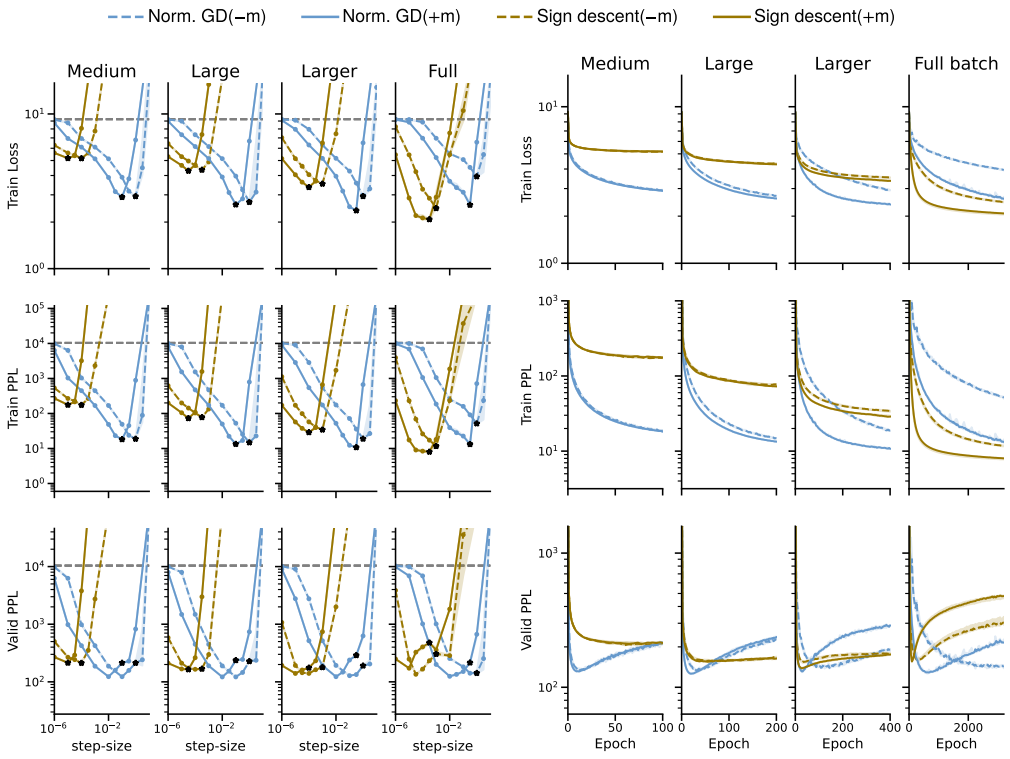

Figure 17: **Grid search and selected runs for Small Transformer on PTB**.

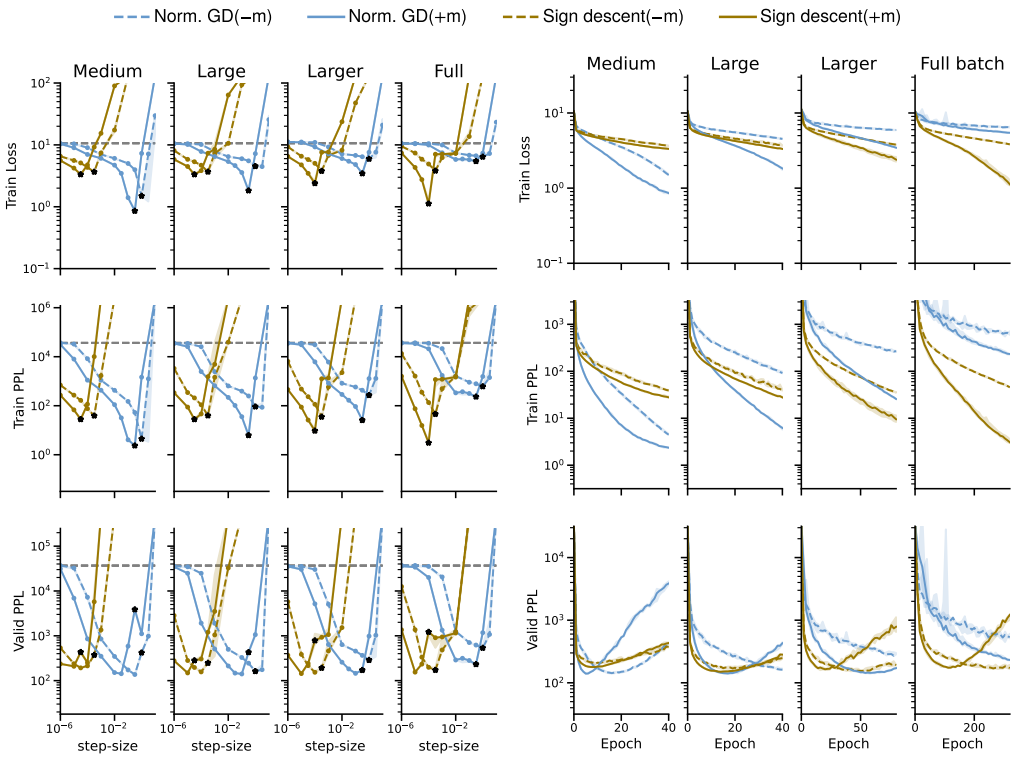

Figure 18: **Grid search and selected runs for Transformer-XL on WikiText-2**.

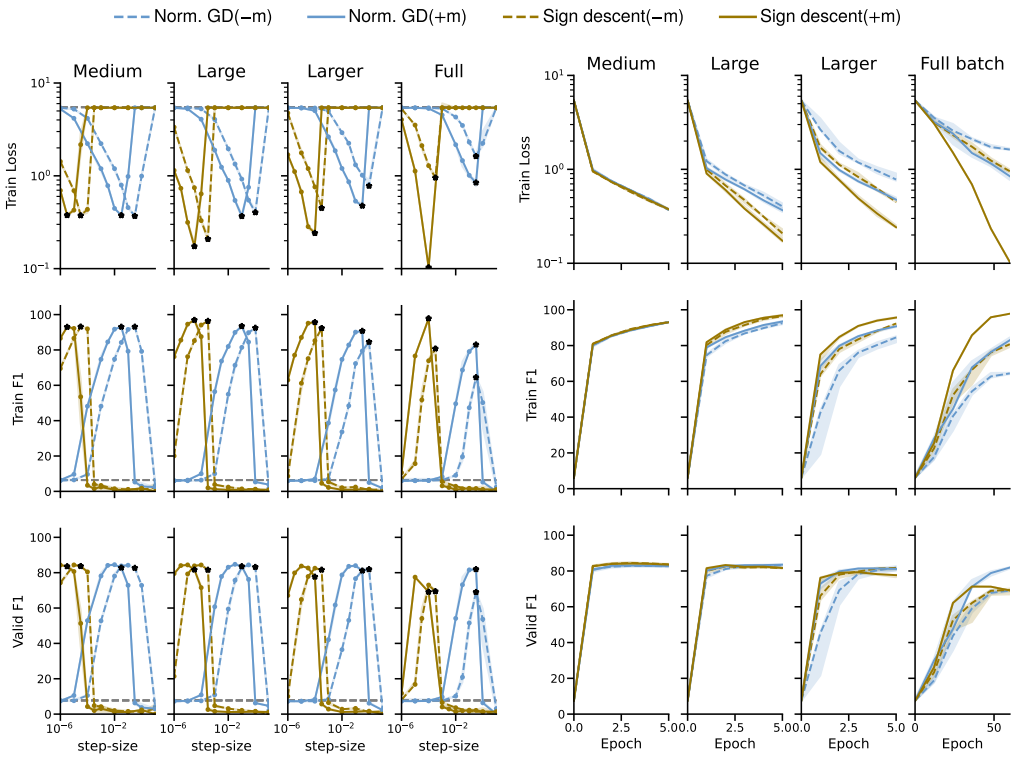

Figure 19: **Grid search and selected runs for DistillBERT finetuning on SQuAD**.

D ADDITIONAL FIGURES

D.1 FINAL PERFORMANCE VS. BATCH SIZE

Figure 20 presents an alternative visualization of the gap between Adam and SGD, looking at the final performance as a function of batch size. Each batch size has a run for a different number of iterations, with large batches doing fewer parameter updates.

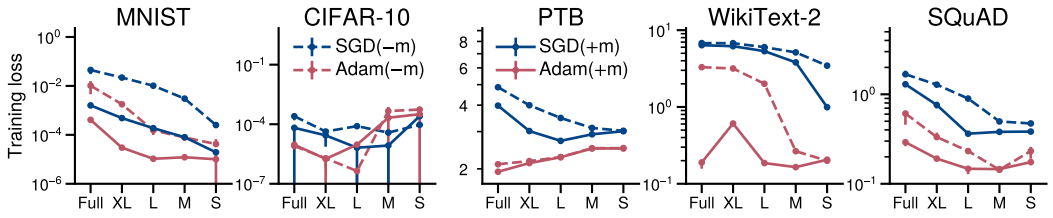

Figure 20: **Final performance vs. Batch size.** Simpler form of the visualization provided in Figure 3. We can see the increase in the gap of the performance between SGD and Adam (both with momentum) on PTB, WikiText2 and SQuAD. However, looking at the final performance in isolation hides the confounder factor that the runs heavily depends on the number of iterations. Figure 3 highlights that the performance across iterations is very stable for SGD without momentum across batch sizes while Adam improves with batch size.

D.2 REMOVING DROPOUT

The increase of batch size to "Full batch" drives noise to a floor, but does not remove it entirely due to the use of Dropout in the transformer models. Figure 21 rerun those models after disabling dropout to verify that the same trends holds, showing the performance of the optimizer in full batch, as in the bottom row of Figure 5 (momentum), Figure 8 (without momentum). We note that the models achieve lower training error, but the overall trend of Sign Descent closing most of the gap between SGD and Adam in full batch is preserved. The grid-search validation results are available in Figure 21.

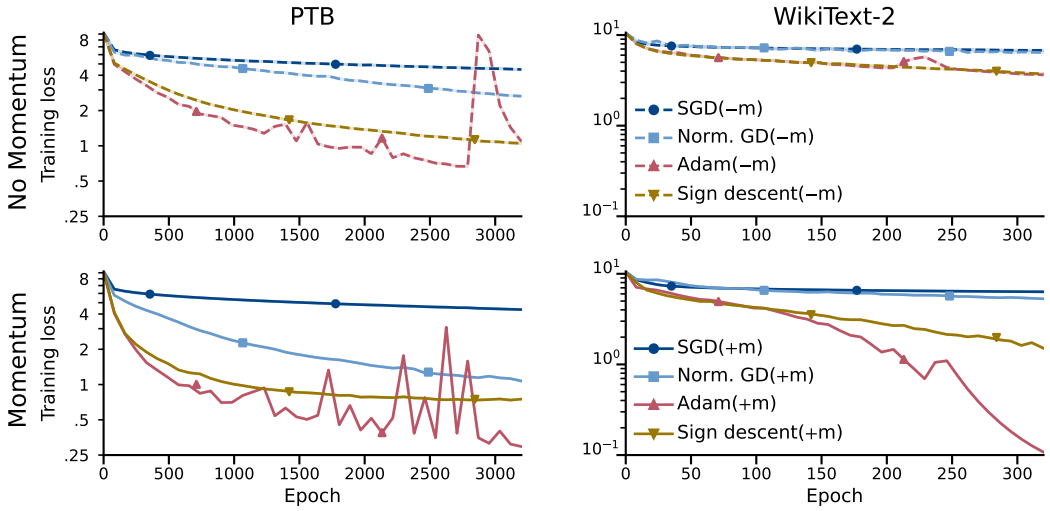

Figure 21: **Checking that momentum is not a confounder.** The increase of batch size to "Full batch" drives noise to a floor, but does not remove it entirely due to the use of Dropout in the transformer models. rerun those models after disabling dropout to verify that the same trends holds. We note that the models achieve lower training error, but the overall trend of Sign Descent closing most of the gap between SGD and Adam in full batch is preserved.

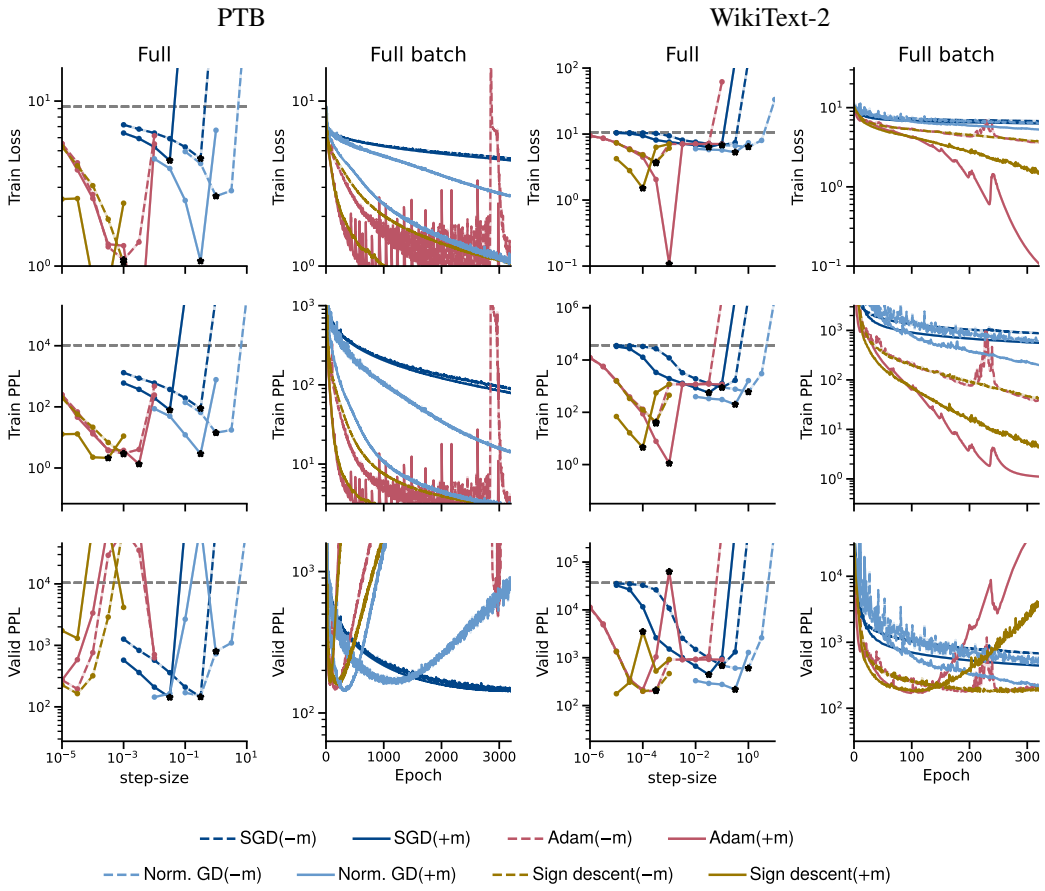

Figure 22: **Grid search and selected runs for Small Transformer on PTB and Transformer-XL on WikiText-2 when Dropout is disabled.**

D.3 RUNNING SQUAD IN FULL BATCH

The runs on SQuAD had the following issue, initially noted in Footnote 3 and detailled in the last Appendix; instead of dropping dropping 0.5% of the dataset and running in full batch on the remaining data, the code used a batch size of 99.5% of the data. We have re-run those results and our conclusions are unchanged. We will update the main plots for the camera ready version and temporarily include those results separately in Figure 23.

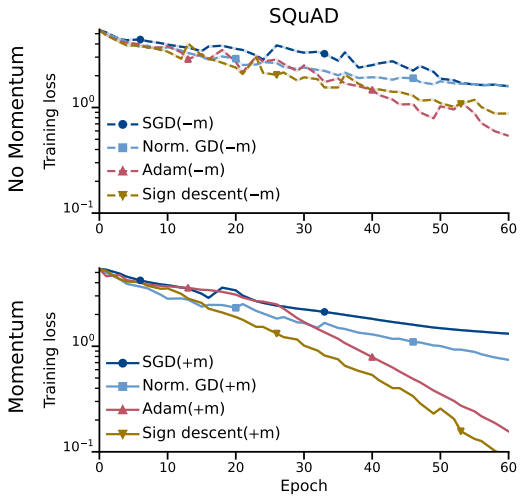

Figure 23: **Repeating the SQuAD runs after fixing the full batch size issue.** The runs on SQuAD had the following issue, initially noted in Footnote 3 and detailled in the last Appendix; instead of dropping dropping 0.5% of the dataset and running in full batch on the remaining data, the code used a batch size of 99.5% of the data. We have re-run those results and our conclusions are unchanged.