Which Algorithmic Choices Matter at Which Batch Sizes? Insights From a Noisy Quadratic Model

Guodong Zhang[1,2,3*], **Lala Li**[3], **Zachary Nado**[3], **James Martens**[4],
Sushant Sachdeva[1], **George E. Dahl**[3], **Christopher J. Shallue**[3], **Roger Grosse**[1,2]
[1]University of Toronto, [2]Vector Institute, [3]Google Research, Brain Team, [4]DeepMind

An optimizer is said to be preconditioned if it does not use the pure gradient, but changes the gradient based on a certain matrix. Examples: Adam, AdaGrad, RSMProp.

Abstract

Increasing the batch size is a popular way to speed up neural network training, but beyond some critical batch size, larger batch sizes yield diminishing returns. In this work, we study how the critical batch size changes based on properties of the optimization algorithm, including acceleration, preconditioning and averaging, through two different lenses: large scale experiments, and analysis of a simple noisy quadratic model (NQM). We experimentally demonstrate that optimization algorithms that employ preconditioning, specifically Adam and K-FAC, result in much larger critical batch sizes than stochastic gradient descent with momentum. We also demonstrate that the NQM captures many of the essential features of real neural network training, despite being drastically simpler to work with. The NQM predicts our results with preconditioned optimizers and exponential moving average, previous results with accelerated gradient descent, and other results around optimal learning rates and large batch training, making it a useful tool to generate testable predictions about neural network optimization.

1 Introduction

Increasing the batch size is one of the most appealing ways to accelerate neural network training on data parallel hardware. Larger batch sizes yield better gradient estimates and, up to a point, reduce the number of steps required for training, which reduces the training time. The importance of understanding the benefits of modern parallel hardware has motivated a lot of recent work on training neural networks with larger batch sizes [Goyal et al., 2017, Osawa et al., 2018, McCandlish et al., 2018, Shallue et al., 2018]. To date, the most comprehensive empirical study of the effects of batch size on neural network training is Shallue et al. [2018], who confirmed that increasing the batch size initially achieves perfect scaling (i.e. doubling the batch size halves the number of steps needed) up to a problem-dependent critical batch size, beyond which it yields diminishing returns [Balles et al., 2017, Goyal et al., 2017, Jastrzębski et al., 2018, McCandlish et al., 2018]. Shallue et al. [2018] also provided experimental evidence that the critical batch size depends on the optimization algorithm, the network architecture, and the data set. However, their experiments only covered plain SGD, SGD with (heavy-ball) momentum, and SGD with Nesterov momentum, leaving open the enticing possibility that other optimizers might extend perfect scaling to even larger batch sizes.

Empirical scaling curves like those in Shallue et al. [2018] are essential for understanding the effects of batch size, but generating such curves, even for a single optimizer on a single task, can be very expensive. On the other hand, existing theoretical analyses that attempt to analytically derive critical batch sizes (e.g. Ma et al. [2018], Yin et al. [2018], Jain et al. [2018]) do not answer our questions about which optimizers scale the best with batch size. They tend to make strong assumptions, produce parameter-dependent results that are difficult to apply, or are restricted to plain SGD. It would be

*Work done as part of the Google Student Researcher Program. Email: gdzhang@cs.toronto.edu

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

ideal to find a middle ground between a purely empirical investigation and theoretical analysis by building a model of neural network optimization problems that captures the essential behavior we see in real neural networks, while still being easy to understand. Additionally, we need to study optimizers beyond momentum SGD since they might provide us an approach to exploit speedups from the very largest batch sizes. In this work, we make the following contributions:

1. We show that a simple noisy quadratic model (NQM) is remarkably consistent with the batch size effects observed in real neural networks, while allowing us to run experiments in seconds, making it a great tool to generate testable predictions about neural network optimization.
2. We show that the NQM successfully predicts that momentum should speed up training relative to plain SGD at larger batch sizes, but have no benefit at small batch sizes.
3. Through large scale experiments with Adam [Kingma and Ba, 2014] and K-FAC [Martens and Grosse, 2015], we confirm that, as predicted by the NQM, preconditioning extends perfect batch size scaling to larger batch sizes than are possible with momentum SGD alone. Furthermore, unlike momentum, preconditioning can help at small batch sizes as well.
4. Lastly, we show that, as predicted by the NQM, exponential moving averages reduce the number of steps required for a specific batch size and can achieve the same acceleration with smaller batch sizes, thereby saving computation.

2 Related Work

From the Jane Street post - since batch size is only relevant to test error as a proxy for temperature when learning rate is fixed, it means I can tune learning rate in order to achieve the largest possible batch size to benefit the most from data parallelism, speeding up training, while still achieving the same test loss.

In a classic paper, Bottou and Bousquet [2008] studied the asymptotics of stochastic optimization algorithms and found SGD to be competitive with fancier approaches. They showed that stochastic optimization involves fundamentally different tradeoffs from full-batch optimization. More recently, several studies have investigated the relationship between batch size and training time for neural networks. Chen et al. [2018] studied the effect of network width on the critical batch size, and showed experimentally that it depends on both the data set and network architecture. Golmant et al. [2018] studied how various heuristics for adjusting the learning rate as a function of batch size affect the relationship between batch size and training time. Shallue et al. [2018] conducted a comprehensive empirical study on the relationship between batch size and training time with different neural network architectures and data sets using plain SGD, heavy-ball momentum, and Nesterov momentum. Finally, McCandlish et al. [2018] used the average gradient noise over training to predict the critical batch size. All of these studies described a basic relationship between batch size and training steps to a fixed error goal, which is comprised of three regions: perfect scaling initially, then diminishing returns, and finally no benefit for all batch sizes greater than the critical batch size.

Other studies have attempted to characterize the critical batch size analytically in stochastic optimization. Under varying assumptions, Ma et al. [2018], Yin et al. [2018], Jain et al. [2018] all derived analytical notions of critical batch size, but to our knowledge, all for SGD.

Additionally, previous studies have shown that SGD and momentum SGD are equivalent for small learning rates (after appropriate rescaling), both for the continuous limit [Leen and Orr, 1994] and discrete settings Yuan et al. [2016]. However, they do not explain why momentum SGD (including heavy-ball and Nesterov momentum) sometimes outperforms plain SGD in mini-batch training (as observed by Kidambi et al. [2018] and Shallue et al. [2018]). Concurrently, Smith et al. [2019] showed that momentum outperforms plain SGD at large batch sizes.

Finally, there are a few works studying average of the iterates, rather than working with the last iterate. This is a classical idea in optimization, where it is known to provide improved convergence [Polyak and Juditsky, 1992, Bach and Moulines, 2013, Dieuleveut and Bach, 2016]. However, most of them focused on *tail averaging*, which you have to decide ahead of time the iteration to start accumulating the running averaging. More commonly (especially in deep learning), exponential moving average [Martens, 2014] is preferred for its simplicity and ability to handle non-convex landscape. However, no analysis was done especially when mini-batch is used.

3 Analysis of the Noisy Quadratic Model (NQM)

In this section, we work with a *noisy quadratic model* (NQM), a stochastic optimization problem whose dynamics can be simulated analytically, in order to reason about various phenomena en-

countered in training neural networks. In this highly simplified model, we first assume the loss function being optimized is a convex quadratic, with noisy observations of the gradient. For analytic tractability, we further assume the noise covariance is codiagonalizable with the Hessian. Because we are not interested in modeling overfitting effects, we focus on the online training setting, where the observations are drawn i.i.d. in every training iteration. Under these assumptions, we derive an analytic expression for the risk after any number of steps of SGD with a fixed step size, as well as a dynamic programming method to compute the risk following a given step size schedule.

Convex quadratics may appear an odd model for a complicated nonconvex optimization landscape. However, one obtains a convex quadratic objective by linearizing the network's function around a given weight vector and taking the second-order Taylor approximation to the loss function (assuming it is smooth and convex). Indeed, recent theoretical works [Jacot et al., 2018, Du et al., 2019, Zhang et al., 2019a] show that for wide enough networks, the weights stay close enough to the initialization for the linearized approximation to remain accurate. Empirically, linearized approximations closely match a variety of training phenomena for large but realistic networks [Lee et al., 2019].

3.1 Problem Setup

We now introduce the noisy quadratic model [Schaul et al., 2013, Martens, 2014, Wu et al., 2018], where the true function being optimized is a convex quadratic. Because we analyze rotation-invariant and translation-invariant optimizers such as SGD and heavy-ball momentum, we assume without loss of generality that the quadratic form is diagonal, and that the optimum is at the origin. Hence, our exact cost function decomposes as a sum of scalar quadratic functions for each coordinate:

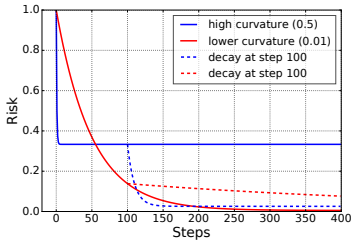


Figure 1: Cartoon of the evolution of risk for different coordinates with and without learning rate decay.

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^\top \mathbf{H}\boldsymbol{\theta} = \frac{1}{2}\sum_{i=1}^{d} h_i \theta_i^2 \triangleq \sum_{i=1}^{d} \ell(\theta_i). \tag{1}$$

Without loss of generality, we assume $h_1 \geq h_2 \geq ... \geq h_d$. We consider a single gradient query to have the form $g(\boldsymbol{\theta}) = \mathbf{H}\boldsymbol{\theta} + \boldsymbol{\epsilon}$ where $\mathbb{E}[\boldsymbol{\epsilon}] = \mathbf{0}$ and $\mathrm{Cov}(\boldsymbol{\epsilon}) = \mathbf{C}$. To reduce the variance of gradient estimation, we can average over multiple independent queries, which corresponds to "mini-batch training" in neural network optimization. We denote the averaged gradient as $g_B(\boldsymbol{\theta})$ and the covariance $\mathrm{Cov}(g_B(\boldsymbol{\theta})) = \mathbf{C}/B$, where B is the number of queries (mini-batch size).

For analytical tractability, we make the nontrivial assumption that $\mathbf{H}$ and $\mathbf{C}$ are codiagonalizable. (Since $\mathbf{H}$ is diagonal, this implies that $\mathbf{C} = \mathrm{diag}(c_1, \ldots, c_d)$.) See Section 3.5 for justification of this assumption. Under gradient descent with fixed step size α, each dimension evolves independently as

$$\theta_i(t+1) = (1 - \alpha h_i)\theta_i(t) + \alpha\sqrt{c_i/B}\epsilon_i, \tag{2}$$

where α is the learning rate and ϵ_i is zero-mean unit variance iid noise. By treating θ_i as a random variable, we immediately obtain the dynamics of its mean and variance.

$$\mathbb{E}\left[\theta_i(t+1)\right] = (1-\alpha h_i)\mathbb{E}\left[\theta_i(t)\right], \; \mathbb{V}\left[\theta_i(t+1)\right] = (1-\alpha h_i)^2 \mathbb{V}\left[\theta_i(t)\right] + \frac{\alpha^2 c_i}{B}. \tag{3}$$

Based on eqn. (3), the expected risk after t steps in a given dimension i is

$$\mathbb{E}\left[\ell(\theta_i(t))\right] = \underbrace{(1-\alpha h_i)^{2t}}_{\text{convergence rate}} \mathbb{E}\left[\ell(\theta_i(0))\right] + \left(1 - (1-\alpha h_i)^{2t}\right) \underbrace{\frac{\alpha c_i}{2B(2-\alpha h_i)}}_{\text{steady state risk}}, \tag{4}$$

where we have assumed that $\alpha h_i \leq 2$. (Note that this can be seen as a special case of the convergence result derived for convex quadratics in Martens [2014].)

Remarkably, each dimension converges exponentially to a steady state risk. Unfortunately, there is a trade-off in the sense that higher learning rates (up to $1/h_i$) give faster convergence to the steady state risk, but also produce higher values of the steady-state risk. The steady state risk also decreases proportionally to increases in batch size; this is important to note because in the following subsections, we will show that traditional acceleration techniques (e.g., momentum and preconditioning) help improve the convergence rate at the expense of increasing the steady state risk. Therefore, the NQM implies that momentum and preconditioning would benefit more from large-batch training compared to plain SGD, as shown in later sections.

3.2 Momentum Accelerates Training at Large Batch Sizes

Applied to the same noisy quadratic model as before, the update equations for momentum SGD are:

$$\begin{aligned} m_i(t+1) &= \beta m_i(t) + h_i\theta_i(t) + \sqrt{c_i/B}\epsilon_i, \\ \theta_i(t+1) &= \theta_i(t) - \alpha m_i(t+1). \end{aligned} \tag{5}$$

We show in the following theorem (see Appendix C for proof) that momentum SGD performs similarly to plain SGD in the regime of small batch sizes but helps in the large-batch regime, which can be viewed as a near-deterministic optimization problem.

Theorem 1. *Given a dimension index i, and $0 \le \beta < 1$ with $\beta \neq (1-\sqrt{\alpha h_i})^2$, the expected risk at time t associated with that dimension satisfies the upper bound*

$$\mathbb{E}\left[\ell(\theta_i(t))\right] \le \left(\frac{(r_1^{t+1} - r_2^{t+1}) - \beta(r_1^t - r_2^t))}{r_1 - r_2}\right)^2 \mathbb{E}\left[\ell(\theta_i(0))\right] + \frac{(1+\beta)\alpha c_i}{2B(2\beta + 2 - \alpha h_i)(1-\beta)}, \tag{6}$$

where r_1 and r_2 (with $r_1 \ge r_2$) are the two roots of the quadratic equation $x^2 - (1 - \alpha h_i + \beta)x + \beta = 0$.

As with plain SGD (c.f. eqn. (4)), the loss associated with each dimension can be expressed as the sum of two terms, where the first one decays exponentially and corresponds to the behavior of the deterministic version of the algorithm, and the second remains constant.

Following the existing treatment of the deterministic version of the algorithm [Chiang, 1974, Qian, 1999, Yang et al., 2018, Goh, 2017], we divide our analysis two cases: *overdamping* and *underdamping*. In the case of overdamping, where $\beta < (1-\sqrt{\alpha h_i})^2$, both roots r_1 and r_2 are real and therefore the convergence rate is determined by the larger one (i.e. r_1), which has the value

$$r_1 = \frac{1 - \alpha h_i + \beta + \sqrt{(1-\beta)^2 - 2(1+\beta)\alpha h_i + \alpha^2 h_i^2}}{2} \tag{7}$$

With a fixed learning rate, the steady state risk will be constant, and the best achievable expected risk will be lower bounded by it. Thus, to achieve a certain target loss we must either drive the learning rate down, or the batch size up. Assuming a small batch size and a low target risk, we are forced to pick a small learning rate, in which case one can show[2] that $r_1 \approx 1 - \alpha h/1-\beta$. In Figure 2 we plot the convergence rate as a function of β, and we indeed observe that the convergence rate closely matches $1 - \alpha h/1-\beta$, assuming a relative small learning rate. We further note that the convergence rate and steady state risk of eqn. (6) are the same as the ones in plain SGD (eqn. (4)), except that they use an "effective learning rate" of $\alpha/1-\beta$. To help validate these predictions, in Appendix E.3 we provide a comparison of momentum SGD with plain SGD using the effective learning rate.

In the case of underdamping where $\beta > (1-\sqrt{\alpha h_i})^2$, both r_1 and r_2 will be complex and have norm $\sqrt{\beta}$. We note that the optimal β should be equal to or smaller than $(1-\sqrt{\alpha h_d})^2$, since otherwise all dimensions are underdamped, and we can easily improve the convergence rate and steady state risk by reducing β.

Next we observe that the convergence of the total loss will *eventually* be dominated by the slowest converging dimension (which corresponds to the smallest curvature h_d), and this will be in the overdamping regime as argued above. By our analysis of the overdamping case, we can achieve the same convergence rate for this dimension by simply replacing the learning rate α in the bound for plain SGD (eqn. (4)) with the effective learning rate $\alpha/1-\beta$.

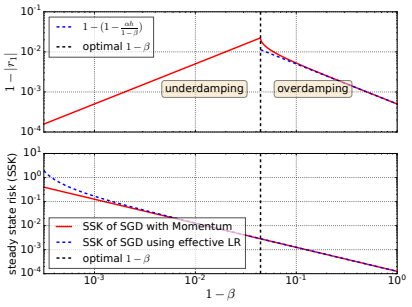


Figure 2: Convergence rate and steady state risk (SSK) as a function of momentum for a single dimension with $\alpha h = 0.0005$ and batch size $B = 1$.

So while momentum gives no long-term training acceleration for very low fixed learning rates (which we are forced to use when the batch size is small), we note that it can help in large-batch training. With $\beta > 0$, the steady state risk roughly amplifies by a factor of $1/1-\beta$, and we note that steady state risk also decreases proportionally to increases in batch size. Therefore, we expect momentum SGD to exhibit perfect scaling up to larger batch sizes than plain SGD.

[2] To see this, note that the term in the square root of eqn. (7) for r_1 can be written as $(1-\beta-(1+\beta)\alpha h_i/1-\beta)^2 + \mathcal{O}(\alpha^2 h_i^2)$. Dropping the $\mathcal{O}(\alpha^2 h_i^2)$ term and simplifying gives the claimed expression for r_1.

3.3 Preconditioning Further Extends Perfect Scaling to Larger Batch Sizes

Many optimizers, such as Adam and K-FAC, can be viewed as preconditioned gradient descent methods. In each update, the gradient is rescaled by a PSD matrix $\mathbf{P}^{-1}$, called the preconditioner.

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \alpha \mathbf{P}^{-1} \left[\mathbf{H}\boldsymbol{\theta} + \boldsymbol{\epsilon}\right]. \tag{8}$$

In lieu of trying to construct noisy quadratic analogues of particular optimizers, we analyze preconditioners of the form $\mathbf{P} = \mathbf{H}^p$ with $0 \leq p \leq 1$. Note that $\mathbf{P}$ remains fixed throughout training since the Hessian $\mathbf{H}$ is constant in the NQM. We can recover standard SGD by setting $p = 0$.

Conveniently, for our NQM, the dynamics of preconditioned SGD are equivalent to the SGD dynamics in an NQM with Hessian $\tilde{\mathbf{H}} = \mathbf{P}^{-1/2}\mathbf{H}\mathbf{P}^{-1/2}$ and gradient covariance $\tilde{\mathbf{C}} = \mathbf{P}^{-1/2}\mathbf{C}\mathbf{P}^{-1/2}$. Hence, the dynamics can be simulated using eqn. (4), exactly like the non-preconditioned case. We immediately obtain the following bound on the risk:

$$\mathbb{E}\left[\mathcal{L}(\boldsymbol{\theta}(t))\right] \leq \sum_{i=1}^{d} (1 - \alpha h_i^{(1-p)})^{2t} \mathbb{E}\left[\ell(\theta_i(0))\right] + \sum_{i=1}^{d} \frac{\alpha c_i h_i^{-p}}{2B(2 - \alpha h_i^{1-p})}. \tag{9}$$

To qualitatively understand the effect of preconditioning, first consider the first term in eqn. (8). The convergence of this term resembles that of gradient descent on a deterministic quadratic, which (with optimal $\alpha \approx 2/\tilde{h}_1$) converges exponentially at a rate of approximately $2/\tilde{\kappa}$, where $\tilde{\kappa} = \tilde{h}_1/\tilde{h}_d$ is the condition number of the transformed problem. Since $\tilde{\kappa} = \kappa^{1-p}$, this implies a factor of κ^p improvement in the rate of convergence. Hence, for near-deterministic objectives where the first term dominates, values of p closer to 1 correspond to better preconditioners, and result in much faster convergence. Unfortunately, there is no free lunch, as larger values of p will also increase the second term (steady state risk). Assuming an ill-conditioned loss surface ($\kappa \gg 1$), the steady state risk of each dimension becomes

$$\frac{1}{2B} \frac{\alpha c_i h_i^{-p}}{2 - \alpha h_i^{(1-p)}} \approx \frac{c_i}{2Bh_1} \frac{(h_i/h_1)^{-p}}{1 - (h_i/h_1)^{(1-p)}}, \tag{10}$$

which is a monotonically increasing function with respect to p. Even without this amplification effect, the steady state risk will eventually become the limiting factor in the minimization of the expected risk. One way to reduce the steady state risk, apart from using Polyak averaging [Polyak and Juditsky, 1992] or decreasing the learning rate (which will harm the rate of convergence), is to increase the batch size. This suggests that the benefits of using stronger preconditioners will be more clearly observed for larger batch sizes, which is an an effect that we empirically demonstrate in later sections.

3.4 Exponential Moving Average Reduces Steady State Risk

Following the same procedure as previous two sections, we analyze exponential moving averages (EMA) on our NQM. The update rule of EMA can be written as

$$\begin{aligned} \boldsymbol{\theta}(t+1) &= \boldsymbol{\theta}(t) - \alpha \left[\mathbf{H}\boldsymbol{\theta} + \boldsymbol{\epsilon}\right], \\ \tilde{\boldsymbol{\theta}}(t+1) &= \gamma \tilde{\boldsymbol{\theta}}(t) + (1-\gamma)\boldsymbol{\theta}(t+1). \end{aligned} \tag{11}$$

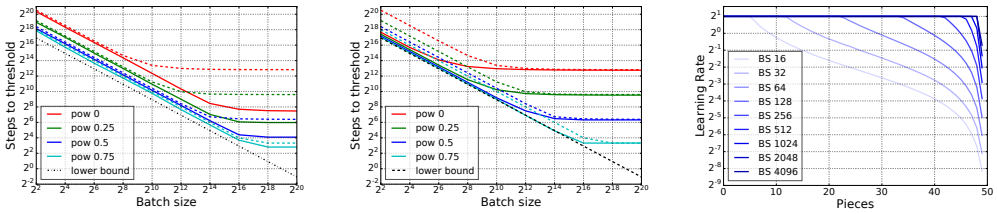
The averaged iterate $\tilde{\boldsymbol{\theta}}$ is used at test time. The computational overhead is minimal (storing an additional copy of the parameters, plus some cheap arithmetic operations). We now show that EMA outperforms plain SGD by reducing the steady state risk term.

Theorem 2. *Given a dimension index i, and $0 \leq \gamma < 1$, the expected risk at time t associated with that dimension satisfies the upper bound*

$$\begin{aligned} \mathbb{E}\left[\ell(\tilde{\theta}_i(t))\right] \leq & \left(\frac{(r_1^{t+1} - r_2^{t+1}) - \gamma(1-\alpha h_i)(r_1^t - r_2^t)}{r_1 - r_2} \right)^2 \mathbb{E}\left[\ell(\theta_i(0))\right] \\ & + \frac{\alpha c_i}{2B(2-\alpha h_i)} \frac{(1-\gamma)(1+(1-\alpha h_i)\gamma)}{(1+\gamma)(1-(1-\alpha h_i)\gamma)}, \end{aligned} \tag{12}$$

where $r_1 = 1 - \alpha h_i$ and $r_2 = \gamma$.

By properly choosing an averaging coefficient $\gamma < 1 - \alpha h_d$ such that $r_1 > r_2$, one can show that EMA reduces the steady state risk without scarificing the convergence rate. To see this, we note that the red part of eqn. (12) is strictly less than 1 given the fact $1 - \alpha h_i < 1$ while the other part is exactly the same as the steady state risk of plain SGD.



(a) Momentum and Preconditioning **(b)** Fixed LR vs. Schedules **(c)** Optimized LR Schedules

Figure 3: (a) Effects of momentum and preconditioning. Steps required to reach target loss as a function of batch size under different preconditioning power. Solid lines are momentum SGD while dashed lines are plain SGD. The black dashed line is the information theoretic lower bound. **(b) Effect of learning rate decay.** The solid lines use the optimized piecewise constant scheme, which are shown in **(c)** for power 0. The dashed curves in **(b)** are plain SGD for comparison. We observe that learning rate schedules close most of the gap between the fixed learning rate performance and the information theoretic lower bound.

3.5 Choice of H and C

We've found that the qualitative behavior of optimizers in our NQM depends on the choices of $\mathbf{H}$ and $\mathbf{C}$. Therefore, we choose matrices motivated by theoretical and empirical considerations about neural net training. First, we set the diagonal entries of $\mathbf{H}$ to be $\{\frac{1}{i}\}_{i=1}^{d}$ for some integer d, giving a condition number of d. This closely matches the estimated eigenspectrum of the Hessian of a convolutional network (see Figure 9 and Appendix E.4), and is also consistent with recent work finding heavy tailed eigenspectra of neural network Hessians [Ubaru et al., 2017, Ghorbani et al., 2019]. We choose $d = 10^4$, which approximately matches the condition number of the K-FAC Hessian approximation for ResNet8. (Qualitative behaviors were consistent for a wide range of d.)

We also set $\mathbf{C} = \mathbf{H}$ (a nontrivial assumption). This was motivated by theoretical arguments that, under the assumption that the implicit conditional distribution over the network's output is close to the conditional distribution of targets from the training distribution, the Hessian closely matches the gradient covariance in neural network training [Martens, 2014]. Empirically, this relationship appears to hold tightly for a convolutional network and moderately well for a transformer (see Appendix E.2).

3.6 Information Theoretic Lower Bound

Since our NQM assumes the infinite data (online optimization) setting, it's instructive to compare the performance of optimizers against an information theoretic lower bound. Specifically, under the assumption that $\mathbf{H} = \mathbf{C}$, the NQM is equivalent to maximum likelihood estimation of the mean vector for a multivariate Gaussian distribution with covariance $\mathbf{H}^{-1}$. Hence, the risk obtained by any optimizer can be bounded below by the risk of the maximum likelihood estimator for the Gaussian, which is $d/2N$, where d is the dimension and N is the total number of training examples visited. We indicate this bound with a dashed black line in our plots.

3.7 Noisy Quadratic Experiments

In this section, we simulate noisy quadratic optimization using the closed-form dynamics. Our aim is to formulate hypotheses for how different optimizers would behave for neural network optimization. Our main metric is the number of steps required to achieve a target risk. For efficiency, rather than explicitly representing all the eigenvalues of $\mathbf{H}$, we quantize them into 100 bins and count the number of eigenvalues in each bin. Unless otherwise specified, we initialize $\boldsymbol{\theta}$ as $\mathcal{N}(\mathbf{0}, \mathbf{I})$ and use a target risk of 0.01. (The results don't seem to be sensitive to either the initial variance or the target risk; some results with varying target risk thresholds are shown in Appendix E.5).

3.7.1 Effect of Momentum, Preconditioning and Exponential Moving Average

We first experiment with momentum and varying preconditioner powers on our NQM. We treat both the (fixed) learning rate α and momentum decay parameter β as hyperparameters, which we tune using a fine-grained grid search.

Consistent with the empirical results of Shallue et al. [2018], each optimizer shows two distinct regimes: a small-batch (stochastic) regime with perfect linear scaling, and a large-batch (deterministic)

Data Set	Size	Model	Remarks	LR
MNIST	55,000	Simple CNN	Same as Shallue et al. [2018] except without dropout regularization.	Constant
FMNIST	55,000			
CIFAR10	45,000	ResNet8 without BN	Same as Shallue et al. [2018].	Constant
		ResNet32 with BN	Ghost batch norm is used.	Linear Decay
		VGG11 with BN	Ghost batch norm is used.	Linear Decay
LM1B	∼30M	Two-layer Transformer	Shallow model in Shallue et al. [2018]	Constant

Table 1: Data sets and models used in our experiments. See Appendix F.2 for full details.

regime insensitive to batch size. We call the phase transition between these regimes the *critical batch size*. Consistent with the analysis of Section 3.2 and the observations of Smith et al. [2018], Shallue et al. [2018], Kidambi et al. [2018], the performance of momentum-based optimizers matches that of the plain SGD methods in the small-batch regime, but momentum increases the critical batch size and gives substantial speedups in the large batch regime. Preconditioning also increases the critical batch size and gives substantial speedups in the large batch regime, but interestingly, also improves performance by a small constant factor even for very small batches. Combining momentum with preconditioning extends both of these trends.

We next experiment with EMA and varying preconditioning powers on our NQM. Following the same procedure as before, we tune both learning rate α and averaging coefficient γ using grid search. As expected, EMA reduces the number of steps required especially for plain SGD with preconditioning power 0. Another interesting observation is that EMA becomes redundant in the large batch (near-deterministic) regime since the main effect of EMA is reducing the steady-state risk, which can also be done by increasing the batch size. This implies that EMA would reduce the critical batch size and therefore achieve the same amount of acceleration with less computation.

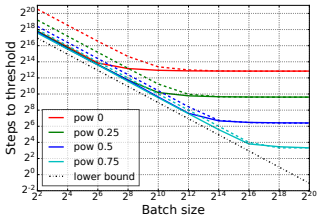


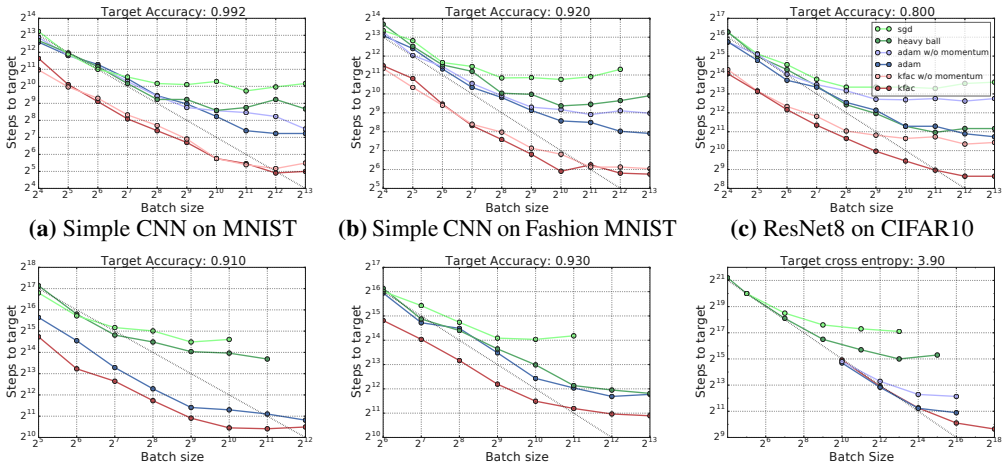
Figure 4: Effects of exponential moving average (EMA). Solid lines are SGD with EMA while dashed lines are plain SGD.

3.7.2 Optimal Learning Rate and Decay Scheme

In the NQM, we can calculate the optimal constant learning rate given a specific batch size. Figure 14 shows the optimal learning rate as a function of batch size for a target risk of 0.01. Notably, the optimal learning rate of plain (preconditioned) SGD (Figure 14a) scales linearly with batch size before it hits the critical batch size, matching the scheme used in Goyal et al. [2017]. The linear scaling also holds for the effective learning rate of momentum SGD. In the small batch regime, the optimal effective learning rate for momentum SGD matches the optimal plain SGD learning rate, suggesting that the momentum and learning rate are interchangeable in the small batch regime.

While a fixed learning rate often works well for simple problems, good performance on the ImageNet benchmark [Russakovsky et al., 2015] requires a carefully tuned schedule. Here we explicitly optimize a piecewise constant learning rate schedule for SGD (with 50 pieces), in terms of the number of steps to reach the loss threshold.[3] In Figure 3b, we show that optimized learning rate schedules help significantly in the small batch regime, consistent with the analysis in Wu et al. [2018]. We observe the same linear scaling as with fixed-learning-rate SGD, but with a better constant factor. In fact, optimized schedules nearly achieve the information theoretic optimum. However, learning rate schedules do not improve at all over fixed learning rates in the large batch regime. Figure 3c shows optimized schedules for different batch sizes; interestingly, they maintain a large learning rate throughout training followed by a roughly exponential decay, consistent with commonly used neural network training schedules. Additionally, even though the different batch sizes start with the same learning rate, their final learning rates at the end of training scale linearly with batch size (see Figure 15 in Appendix E.7).

[3]For a given schedule and number of time steps, we obtain the exact risk using dynamic programming with eqn. (3). For stability, the learning rates are constrained to be at most $2/h_1$. For a fixed number of time steps, we minimize this risk using BFGS. We determine the optimal number of time steps using binary search.



(a) Simple CNN on MNIST **(b)** Simple CNN on Fashion MNIST **(c)** ResNet8 on CIFAR10

(d) VGG11 on CIFAR10 **(e)** ResNet32 on CIFAR10 **(f)** Transformer on LM1B

Figure 5: Empirical relationship between batch size and steps to result. Key observations: 1) momentum SGD has no benefit over plain SGD at small batch sizes, but extends the perfect scaling to larger batch sizes; 2) preconditioning also extends perfect scaling to larger batch sizes, i.e. K-FAC > Adam > momentum SGD. This is most noticeable in the Transformer model; 3) preconditioning (particularly K-FAC) reduces the number of steps needed to reach the target even for small batch sizes. All of these agree with the predictions by NQM.

4 Neural Network Experiments

We investigated whether the predictions made by the NQM hold in practice by running experiments with five neural network architectures across three image classification tasks and one language modeling task (see Table 1). For each model and task, we compared a range of optimizers: SGD, momentum SGD, Adam (with and without momentum), and K-FAC (with and without momentum). For K-FAC, preconditioning is applied before momentum. See Appendix F for more details.

The primary quantity we measured is the number of steps required to reach a target accuracy (for image classification tasks) or cross entropy (for language modeling). Unless otherwise specified, we measured steps to target on the validation set. We chose the target metric values based on an initial set of experiments with practical computational budgets. For each model, task, optimizer, and batch size, we independently tuned the learning rate α, the parameters governing the learning rate schedule (where applicable), and optimizer-specific metaparameters (see Appendix F.4). We manually chose the search spaces based on our initial experiments, and we verified after each experiment that the optimal metaparameter values were far from the search space boundaries. We used quasi-random search [Bousquet et al., 2017] to tune the metaparameters with fixed budgets of non-divergent[4] trials (100 for Simple CNN, ResNet8, and Transformer, and 200 for ResNet32 and VGG11). We chose the trial that reached the target metric value using the fewest number of steps.

4.1 Critical Batch Size Depends on the Optimizer

Figure 5 shows the relationship between batch size and steps to target for each model, task, and optimizer. In each case, as the batch size grows, there is an initial period of perfect scaling where doubling the batch size halves the steps to target, but once the batch size exceeds a problem-dependent critical batch size, there are rapidly diminishing returns, matching the results of [Goyal et al., 2017, McCandlish et al., 2018, Shallue et al., 2018]. K-FAC has the largest critical batch size in all cases, highlighting the usefulness of preconditioning. Momentum SGD extends perfect scaling to larger batch sizes than plain SGD, but for batch sizes smaller than the plain SGD critical batch size, momentum SGD requires as many steps as plain SGD to reach the target. This is consistent with both the empirical results of Shallue et al. [2018] and our NQM simulations. By contrast, Adam and K-FAC can reduce the number of steps needed to reach the target compared to plain SGD even for the smallest batch sizes, although neither optimizer does so in all cases. Finally, we see some evidence that the benefit of momentum diminishes with preconditioning (Figures 5a and 5b), as predicted by our NQM simulations, although we do not see this in all cases (e.g. Figure 5c and 5f).

[4]We discarded trials with a divergent training loss, which occurred when the learning rate was too high.

4.2 Exponential Moving Average Improves Convergence with Minimal Computation Cost

To verify the predictions of NQM on exponential moving average (EMA), we conducted some experiments on comparing EMA with plain SGD. We follow the same protocol of Figure 5 and report the results in Figure 6. As expected, the results on real neural networks closely match our predictions based on NQM analysis. In particular, SGD with EMA appears to reach the same target with fewer steps than plain SGD at small batch sizes, though the benefit of EMA diminishes with large batch sizes. Besides, we note that EMA leads to smaller critical batch sizes and achieves the same acceleration with less computation.

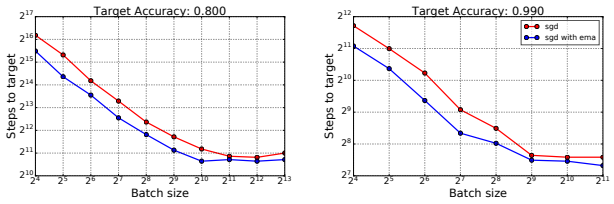


Figure 6: Steps to training accuracy versus batch size. **Left:** ResNet8 on CIFAR10; **Right**: Simple CNN on MNIST.

4.3 Optimal Learning Rate

The NQM predicts that the optimal constant learning rate for plain SGD (or effective learning rate for momentum SGD) scales linearly with batch size initially, and then levels off after a certain batch size. Figure 7 shows the empirical optimal (effective) learning rate as a function of batch size for simple CNN on MNIST and ResNet8 on CIFAR10. For small batch sizes, the optimal learning rate of plain SGD appears to match the optimal effective learning rate of momentum SGD. However, after a certain batch size, the optimal learning rate for plain SGD saturates while the optimal effective learning rate of momentum SGD keeps increasing. Interestingly, plain SGD and momentum SGD appear to deviate at the same batch size in the optimal effective learning rate and steps to target plots (Figures 5 and 7).

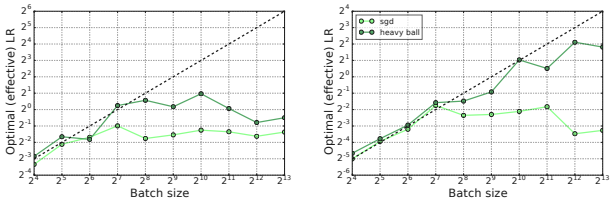


Figure 7: Optimal learning rates for plain SGD and momentum SGD. **Left**: Simple CNN on MNIST; **Right**: ResNet8 on CIFAR10

4.4 Steps to Target on the Training Set

Figure 8 shows the empirical relationship between batch size and steps to target, measured on the training set, for ResNet8 and ResNet32 on CIFAR10. For ResNet8, the curves are almost identical to those using validation accuracy (Figure 5c), but for ResNet32, the gaps between different optimizers become much smaller than in Figure 5e and the effects of momentum and preconditioning appear to become less significant. Nevertheless, the qualitative differences between optimizers are consistent with the validation set measurements.

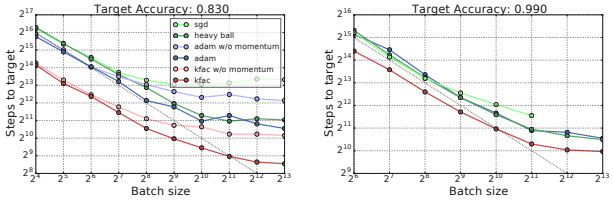


Figure 8: Steps to training accuracy versus batch size on CIFAR10. **Left:** ResNet8; **Right**: ResNet32.

5 Conclusion

In this work, we analyzed the interactions between the batch size and the optimization algorithm from two perspectives: experiments with real neural networks, and a noisy quadratic model with parameters chosen based on empirical observations about neural networks. Despite its simplicity, the noisy quadratic model agrees remarkably well with a variety of neural network training phenomena, including learning rate scaling, critical batch sizes, and the effects of momentum, preconditioning and averaging. More importantly, the noisy quadratic model allows us to run experiments in seconds, while it can take weeks, or even months, to conduct careful large-scale experiments with real neural networks. Therefore, the noisy quadratic model is a convenient and powerful way to quickly formulate testable predictions about neural network optimization.

References

Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using Kronecker-factored approximations. In *International Conference on Learning Representations*, 2017.

Francis Bach and Eric Moulines. Non-strongly-convex smooth stochastic approximation with convergence rate o (1/n). In *Advances in neural information processing systems*, pages 773–781, 2013.

Juhan Bae, Guodong Zhang, and Roger Grosse. Eigenvalue corrected noisy natural gradient. In *Workshop of Bayesian Deep Learning, Advances in neural information processing systems*, 2018.

Lukas Balles, Javier Romero, and Philipp Hennig. Coupling adaptive batch sizes with learning rates. In *Conference on Uncertainty in Artificial Intelligence (UAI) 2017*. AUAI Press, 2017.

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.

Lingjiao Chen, Hongyi Wang, Jinman Zhao, Dimitris Papailiopoulos, and Paraschos Koutris. The effect of network width on the performance of large-batch training. In *Advances in Neural Information Processing Systems*, pages 9302–9309, 2018.

A.C. Chiang. *Fundamental Methods of Mathematical Economics*. International student edition. McGraw-Hill, 1974. ISBN 9780070107809.

Aymeric Dieuleveut and Francis Bach. Nonparametric stochastic approximation with large step-sizes. *THE ANNALS*, 44(4):1363–1399, 2016.

Simon S. Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=S1eK3i09YQ.

Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a Kronecker-factored eigenbasis. In *Advances in Neural Information Processing Systems*, pages 9550–9560, 2018.

Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. An investigation into neural net optimization via hessian eigenvalue density. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2232–2241, 2019.

Gabriel Goh. Why momentum really works. *Distill*, 2(4):e6, 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.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. 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.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

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.

Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, pages 8571–8580, 2018.

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.

Stanisław Jastrzębski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in SGD. In *International Conference on Artificial Neural Networks*, 2018.

Rahul Kidambi, Praneeth Netrapalli, Prateek Jain, and Sham Kakade. On the insufficiency of existing momentum schemes for stochastic optimization. In *2018 Information Theory and Applications Workshop (ITA)*, pages 1–9. IEEE, 2018.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2014.

Jaehoon Lee, Lechao Xiao, Samuel S Schoenholz, Yasaman Bahri, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. *arXiv preprint arXiv:1902.06720*, 2019.

Todd K. Leen and Genevieve B. Orr. Optimal stochastic search and adaptive momentum. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 477–484. Morgan-Kaufmann, 1994. URL http://papers.nips.cc/paper/772-optimal-stochastic-search-and-adaptive-momentum.pdf.

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. New insights and perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193*, 2014.

James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning*, pages 2408–2417, 2015.

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Second-order optimization method for large mini-batch: Training resnet-50 on imagenet in 35 epochs. *arXiv preprint arXiv:1811.12019*, 2018.

Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1): 145–151, 1999.

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.

Levent Sagun, Leon Bottou, and Yann LeCun. Eigenvalues of the hessian in deep learning: Singularity and beyond. *arXiv preprint arXiv:1611.07476*, 2016.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *International Conference on Machine Learning*, pages 343–351, 2013.

Nicol N Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14(7):1723–1738, 2002.

Christopher J Shallue, Jaehoon Lee, Joe Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=B1Yy1BxCZ.

Samuel L Smith, Erich Elsen, and Soham De. Momentum enables large batch training. In *Theoretical Physics for Deep Learning Workshop, International Conference on Machine Learning.*, 2019.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

Shashanka Ubaru, Jie Chen, and Yousef Saad. Fast estimation of tr(f(a)) via stochastic lanczos quadrature. *SIAM Journal on Matrix Analysis and Applications*, 38(4):1075–1099, 2017.

Twan van Laarhoven. L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350*, 2017.

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.

Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. Eigendamage: Structured pruning in the Kronecker-factored eigenbasis. In *Proceedings of the 36th International Conference on Machine Learning*, pages 6566–6575, 2019.

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-.

Lin Yang, Raman Arora, Tuo Zhao, et al. The physical systems behind optimization algorithms. In *Advances in Neural Information Processing Systems*, pages 4372–4381, 2018.

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*, pages 1998–2007, 2018.

Kun Yuan, Bicheng Ying, and Ali H. Sayed. On the influence of momentum acceleration on online learning. *Journal of Machine Learning Research*, 17(192):1–66, 2016. URL http://jmlr.org/papers/v17/16-157.html.

Guodong Zhang, James Martens, and Roger Grosse. Fast convergence of natural gradient descent for overparameterized neural networks. *arXiv preprint arXiv:1905.10961*, 2019a.

Guodong Zhang, Chaoqi Wang, Bowen Xu, and Roger Grosse. Three mechanisms of weight decay regularization. In *International Conference on Learning Representations*, 2019b. URL https://openreview.net/forum?id=B1lz-3Rct7.

A Kronecker-factored Approximate Curvature (K-FAC)

Kronecker-factored approximate curvature (K-FAC) [Martens and Grosse, 2015] uses a Kronecker-factored approximation to the curvature matrix to perform efficient approximate natural gradient updates. Considering the l-th layer in a neural network whose input activations are $\mathbf{a} \in \mathbb{R}^n$, weight matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$, and outputs $\mathbf{s} \in \mathbb{R}^m$, we have $\mathbf{s} = \mathbf{W}^\top \mathbf{a}$. Therefore, the weight gradient is $\nabla_{\mathbf{W}} \mathcal{L} = \mathbf{a}(\nabla_{\mathbf{s}} \mathcal{L})^\top$. With this formula, K-FAC decouples this layer's Fisher matrix $\mathbf{F}$ using an independence assumption:

$$\begin{aligned} \mathbf{F} &= \mathbb{E}[\mathrm{vec}\{\nabla_{\mathbf{W}} \mathcal{L}\} \mathrm{vec}\{\nabla_{\mathbf{W}} \mathcal{L}\}^\top] = \mathbb{E}[\{\nabla_{\mathbf{s}} \mathcal{L}\}\{\nabla_{\mathbf{s}} \mathcal{L}\}^\top \otimes \mathbf{a}\mathbf{a}^\top] \\ &\approx \mathbb{E}[\{\nabla_{\mathbf{s}} \mathcal{L}\}\{\nabla_{\mathbf{s}} \mathcal{L}\}^\top] \otimes \mathbb{E}[\mathbf{a}\mathbf{a}^\top] = \mathbf{S} \otimes \mathbf{A} \end{aligned} \tag{13}$$

where $\mathbf{A} = \mathbb{E}[\mathbf{a}\mathbf{a}^\top]$ and $\mathbf{S} = \mathbb{E}[\{\nabla_{\mathbf{s}} \mathcal{L}\}\{\nabla_{\mathbf{s}} \mathcal{L}\}^\top]$. Decomposing $\mathbf{F}$ into $\mathbf{A}$ and $\mathbf{S}$ not only avoids the quadratic storage cost of the exact Fisher, but also enables tractable computation of the approximate natural gradient:

$$\begin{aligned} \mathbf{F}^{-1} \mathrm{vec}\{\nabla_{\mathbf{W}} \mathcal{L}\} &= (\mathbf{S}^{-1} \otimes \mathbf{A}^{-1}) \, \mathrm{vec}\{\nabla_{\mathbf{W}} \mathcal{L}\} \\ &= \mathrm{vec}[\mathbf{A}^{-1} \nabla_{\mathbf{W}} \mathcal{L} \mathbf{S}^{-1}] \end{aligned} \tag{14}$$

As shown by eqn. (14), computing natural gradient using K-FAC only consists of matrix transformations comparable to size of $\mathbf{W}$, making it very efficient.

Later, Grosse and Martens [2016] further extended K-FAC to convolutional layers under additional assumptions of spatial homogeneity (**SH**) and spatially uncorrelated derivatives (**SUD**). Suppose the input $\mathbf{a} \in \mathbb{R}^{c_{\text{in}} \times h \times w}$ and the output $\mathbf{s} \in \mathbb{R}^{c_{\text{out}} \times h \times w}$, then the gradient of the reshaped weight $\mathbf{W} \in \mathbb{R}^{c_{\text{out}} \times c_{\text{in}} k^2}$ is $\nabla_{\mathbf{W}} \mathcal{L} = \sum \mathbf{a}_i \nabla_{\mathbf{s}_i} \mathcal{L}^\top$, and the corresponding Fisher matrix is:

$$\begin{aligned} \mathbf{F} &\approx \sum \mathbb{E}\left[\{\nabla_{\mathbf{s}_i} \mathcal{L}\}\{\nabla_{\mathbf{s}_{i'}} \mathcal{L}\}^\top\right] \otimes \mathbb{E}\left[\mathbf{a}_i \mathbf{a}_{i'}^\top\right] \\ &\approx \underbrace{\left(\frac{1}{|\mathcal{I}|} \sum \mathbb{E}\left[\{\nabla_{\mathbf{s}_i} \mathcal{L}\}\{\nabla_{\mathbf{s}_i} \mathcal{L}\}^\top\right]\right)}_{\mathbf{S}, \text{size}=(c_{\text{out}})^2} \otimes \underbrace{\left(\sum \mathbb{E}\left[\mathbf{a}_i \mathbf{a}_i^\top\right]\right)}_{\mathbf{A}, \text{size}=(c_{\text{in}} \times k^2)^2} \end{aligned} \tag{15}$$

where $\mathcal{I} = [h] \times [w]$ is the set of spatial locations, $\mathbf{a}_i \in \mathbb{R}^{c_{\text{in}} k^2}$ is the patch extracted from $\mathbf{a}$, $\nabla_{\mathbf{s}_i} \mathcal{L} \in \mathbb{R}^{c_{\text{out}}}$ is the gradient to each spatial location in $\mathbf{s}$ and $i, i' \in \mathcal{I}$.

A.1 K-FAC for Transformer

K-FAC has been implemented on the autoencoder [Martens and Grosse, 2015] and various convolutional networks [Grosse and Martens, 2016, Ba et al., 2017] before. To our knowledge, this is the first time K-FAC is implemented on the Transformer model. What is different from the previous models is the shared weight matrix between the embedding layer and the pre-softmax linear transformation [Vaswani et al., 2017]. In particular, the weight matrix is transposed at the pre-softmax layer: $\mathbf{s} = \mathbf{W}\mathbf{a}$ and $\nabla_{\mathbf{W}} \mathcal{L} = (\nabla_{\mathbf{s}} \mathcal{L})\mathbf{a}^\top$. With the same assumptions as the non-transposed case, we get

$$\mathbf{F} \approx \mathbb{E}[\mathbf{a}\mathbf{a}^\top \otimes \{\nabla_{\mathbf{s}} \mathcal{L}\}\{\nabla_{\mathbf{s}} \mathcal{L}\}^\top] = \mathbf{A} \otimes \mathbf{S} \tag{16}$$

i.e. the positions of the two Kronecker factors are swapped. If we name the two Kronecker factors "input factor" and "output factor" respectively, i.e. $\mathbf{F} \approx$ *input_factor* $\otimes$ *output_factor*, then for the weight matrix that is shared between the embedding layer and the pre-softmax layer, the *input_factor* has contributions from both the embedding inputs and the gradients of pre-softmax layer outputs; and the *output_factor* has contributions from both the pre-softmax layer inputs and the gradients of the embedding outputs. In practice, when computing a Kronecker factor, we treat contribution from multiple sources as an equivalent situation as contribution from multiple training examples from a mini-batch. Also note that because of the high dimensionality of the embedding weight matrix (with a vocabulary size of 32,768), the dense input factor would have size $[32768, 32768]$. In order to save memory, we use a diagonal matrix to estimate the *input_factor*. The *output_factor* is still estimated with a dense matrix.

B Dynamics of momentum SGD on noisy quadratic model

Similar to plain SGD, by treating θ_i as a random variable, we can explicitly write down the dynamics of its expectation and variance. But due to the use of momentum, we need to take into account m_i and its correlation with θ_i. Because each dimension evolves independently, we drop the the dimension subscripts. We first calculate the expectation of the parameter and velocity:

$$\begin{aligned}\mathbb{E}\left[\theta(t+1)\right] &= (1-\alpha h)\mathbb{E}\left[\theta(t)\right] - \alpha\beta\mathbb{E}\left[m(t)\right] \\ \mathbb{E}\left[m(t+1)\right] &= \beta\mathbb{E}\left[m(t)\right] + h\mathbb{E}\left[\theta(t)\right]\end{aligned} \tag{17}$$

We then calculate the variance:

$$\begin{aligned}\mathbb{V}\left[\theta(t+1)\right] &= (1-\alpha h)^2\mathbb{V}\left[\theta(t)\right] + (\alpha\beta)^2\mathbb{E}\left[m(t)\right] - 2(1-\alpha h)\alpha\beta\mathrm{Cov}(t) + \frac{\alpha^2 c}{B} \\ \mathbb{V}\left[m(t+1)\right] &= \beta^2\mathbb{V}\left[m(t)\right] + h^2\mathbb{V}\left[\theta(t)\right] + 2\beta h\mathrm{Cov}(t) + \frac{c}{B}\end{aligned} \tag{18}$$

where $\mathrm{Cov}(t) = \mathrm{Cov}(\theta(t), m(t))$ evolves as

$$\mathrm{Cov}(t+1) = (1-\alpha h)h\mathbb{V}\left[\theta(t)\right] - \alpha\beta^2\mathbb{V}\left[m(t)\right] + (1-2\alpha h)\beta\mathrm{Cov}(t) - \frac{\alpha c}{B} \tag{19}$$

Because the expected risk is totally decided by $\mathbb{E}\left[\theta\right]^2 + \mathbb{V}\left[\theta\right]$, we define $A(\cdot) = \mathbb{E}\left[\cdot\right]^2 + \mathbb{V}\left[\cdot\right]$ and $C(t) = \mathbb{E}[\theta(t)]\mathbb{E}[m(t)] + \mathrm{Cov}(\theta(t), m(t))$. We can then simplify the dynamics as follows

$$\begin{aligned}A(\theta(t+1)) &= (1-\alpha h)^2 A(\theta(t)) + (\alpha\beta)^2 A(m(t)) - 2(1-\alpha h)\alpha\beta C(t) + \frac{\alpha^2 c}{B} \\ A(m(t+1)) &= \beta^2 A(m(t)) + h^2 A(\theta(t)) + 2\beta h C(t) + \frac{c}{B} \\ C(t+1) &= (1-\alpha h)hA(\theta(t)) - \alpha\beta^2 A(m(t)) + (1-2\alpha h)\beta C(t) - \frac{\alpha c}{B}\end{aligned} \tag{20}$$

or equivalently

$$\underbrace{\begin{bmatrix} A(\theta(t+1)) \\ \alpha^2 A(m(t+1)) \\ -\alpha C(t+1)\end{bmatrix}}_{\mathbf{v}(t+1)} = \underbrace{\begin{bmatrix} (1-\alpha h)^2 & \beta^2 & 2(1-\alpha h)\beta \\ (\alpha h)^2 & \beta^2 & -2\beta\alpha h \\ -(1-\alpha h)\alpha h & \beta^2 & (1-2\alpha h)\beta\end{bmatrix}}_{\text{transition matrix } \mathbf{T}} \underbrace{\begin{bmatrix} A(\theta(t)) \\ \alpha^2 A(m(t)) \\ -\alpha C(t)\end{bmatrix}}_{\mathbf{v}(t)} + \underbrace{\begin{bmatrix} \frac{\alpha^2 c}{B} \\ \frac{\alpha^2 c}{B} \\ \frac{\alpha^2 c}{B}\end{bmatrix}}_{\mathbf{n}} \tag{21}$$

The convergence rate is determined by the transition matrix $\mathbf{T}$ which has the characteristic polynomial

$$|\mathbf{T} - \lambda\mathbf{I}| = -(\lambda-\beta)(\lambda^2 - (\beta^2 - 2\alpha h\beta + (1-\alpha h)^2)\lambda + \beta^2) \tag{22}$$

With the momentum value $\beta = (1-\sqrt{\alpha h})^2$, all eigenvalues of the transition matrix are equal to each other with the value β, giving the fastest convergence.

C Proof of Theorem 1

For a linear dynamical system like eqn. (21), we can get $\mathbf{v}(t)$ in the following form:

$$\mathbf{v}(t) = \mathbf{T}^t\mathbf{v}(0) + \sum_{p=1}^{t+1}\mathbf{T}^{p-1}\mathbf{n} \leq \mathbf{T}^t\mathbf{v}(0) + \sum_{p=1}^{\infty}\mathbf{T}^{p-1}\mathbf{n} \tag{23}$$

We first analyze the stochastic term $\sum_{p=1}^{\infty}\mathbf{T}^{p-1}\mathbf{n}$. For notational convenience, we define

$$\sum_{p=1}^{\infty}\mathbf{T}^{p-1}\mathbf{n} \triangleq \sum_{p=0}^{\infty}\left[x_p, y_p, z_p\right]^\top \tag{24}$$

In eqn. (24), we append zero vector $[x_0, y_0, z_0]^\top$ for convenience. To compute the infinite sum, we first focus on a single term. We have the following update:

$$\begin{aligned}\sqrt{x_{p+1}} &= (1-\alpha h)\sqrt{x_p} + \beta\sqrt{y_p} \\ \sqrt{y_{p+1}} &= -\alpha h\sqrt{x_p} + \beta\sqrt{y_p}\end{aligned} \tag{25}$$

Since we only care x_p which totally decide the loss, so we get rid of y_p by merging two updates, which yields a second-order difference equation:

$$\sqrt{x_{p+1}} = (1 - \alpha h + \beta)\sqrt{x_p} - \beta\sqrt{x_{p-1}} \tag{26}$$

with initial conditions $\sqrt{x_0} = 0$ and $\sqrt{x_1} = \sqrt{\frac{\alpha^2 c}{B}}$. To solve the second-order difference equation, we leverage the Z-transform to get the analytical form. Based on basic manipulation of the Z-transform, we have the Z-domain function

$$X(Z) = \frac{\sqrt{\frac{\alpha^2 c}{B}} Z}{Z^2 - (1 - \alpha h + \beta)Z + \beta} = \frac{\sqrt{\frac{\alpha^2 c}{B}}}{r_1 - r_2}\left(\frac{1}{1 - Z^{-1} r_1} - \frac{1}{1 - Z^{-1} r_2}\right) \tag{27}$$

where r_1 and r_2 are two roots of equation $z^2 - (1 - \alpha h + \beta)z + \beta$. Then, we use the inverse Z-transform to get $\sqrt{x_p}$:

$$\sqrt{x_p} = \sqrt{\frac{\alpha^2 c}{B}} \frac{r_1^p - r_2^p}{r_1 - r_2} \tag{28}$$

and therefore

$$x_p = \frac{\alpha^2 c}{B} \frac{r_1^{2p} + r_2^{2p} - 2(r_1 r_2)^p}{(r_1 - r_2)^2} \tag{29}$$

Now, we are ready to compute the infinite sum $\sum_{p=0}^{\infty} x_p$:

$$\begin{aligned} \sum_{p=0}^{\infty} x_p &= \frac{\frac{\alpha^2 c}{B}}{(r_1 - r_2)^2}\left(\frac{1}{1 - r_1^2} + \frac{1}{1 - r_2^2} - \frac{2}{1 - r_1 r_2}\right) \\ &= \frac{\alpha^2 c}{B} \frac{1 + r_1 r_2}{(1 - r_1^2)(1 - r_2^2)(1 - r_1 r_2)} \end{aligned} \tag{30}$$

Because r_1 and r_2 are two roots with $r_1 r_2 = \beta$, $r_1 + r_2 = 1 - \alpha h + \beta$, we have

$$\sum_{p=0}^{\infty} x_p = \frac{\alpha c (1 + \beta)}{B h (2\beta + 2 - \alpha h)(1 - \beta)} \tag{31}$$

Now, we analyze the deterministic term. Similar to the analysis of stochastic term, we have the same second-order difference equation

$$\sqrt{x'_{p+1}} = (1 - \alpha h + \beta)\sqrt{x'_p} - \beta\sqrt{x'_{p-1}} \tag{32}$$

except the initial conditions become $\sqrt{x'_0} = \sqrt{x'_1} = \sqrt{A(\theta(0))}$. According to Z-transform, we have

$$x'_t = \left(\frac{r_1^{t+1} - r_2^{t+1} - \beta(r_1^t - r_2^t)}{r_1 - r_2}\right)^2 A(\theta(0)) \tag{33}$$

Along with eqn. (31), we have

$$A(\theta(t)) \le \left(\frac{r_1^{t+1} - r_2^{t+1} - \beta(r_1^t - r_2^t)}{r_1 - r_2}\right)^2 A(\theta(0)) + \frac{\alpha c (1 + \beta)}{B h (2\beta + 2 - \alpha h)(1 - \beta)} \tag{34}$$

D Proof of Theorem 2

Similar to plain SGD, by treating θ_i as a random variable, we can explicitly write down the dynamics of its expectation and variance. But due to the use of moving averaging, we need to take into account $\tilde{\theta}_i$ and its correlation with θ_i. Because each dimension evolves independently, we drop the the dimension subscripts. We first calculate the expectation of the parameter and the average:

$$\begin{aligned} \mathbb{E}\left[\theta(t+1)\right] &= (1 - \alpha h)\mathbb{E}\left[\theta(t)\right] \\ \mathbb{E}[\tilde{\theta}(t+1)] &= \gamma \mathbb{E}[\tilde{\theta}(t)] + (1 - \gamma)(1 - \alpha h)\mathbb{E}\left[\theta(t)\right] \end{aligned} \tag{35}$$

We then calculate the variance:

$$\begin{aligned}\mathbb{V}\left[\theta(t+1)\right] &= (1-\alpha h)^2\mathbb{V}\left[\theta(t)\right] + \frac{\alpha^2 c}{B} \\ \mathbb{V}[\tilde{\theta}(t+1)] &= \gamma^2\mathbb{V}[\tilde{\theta}(t)] + (1-\gamma)^2(1-\alpha h)^2\mathbb{V}\left[\theta(t)\right] \\ &\quad + 2\gamma(1-\gamma)(1-\alpha h)\mathrm{Cov}(t) + \frac{(1-\gamma)^2\alpha^2 c}{B}\end{aligned} \tag{36}$$

where $\mathrm{Cov}(t) = \mathrm{Cov}(\theta(t), \tilde{\theta}(t))$ evolves as

$$\mathrm{Cov}(t+1) = (1-\gamma)(1-\alpha h)^2\mathbb{V}\left[\theta(t)\right] + (1-\alpha\gamma)\mathrm{Cov}(t) + \frac{(1-\gamma)\alpha^2 c}{B} \tag{37}$$

Because the expected risk is totally decided by $\mathbb{E}[\tilde{\theta}]^2 + \mathbb{V}[\tilde{\theta}]$, we define $A(\cdot) = \mathbb{E}\left[\cdot\right]^2 + \mathbb{V}\left[\cdot\right]$ and $C(t) = \mathbb{E}[\theta(t)]\mathbb{E}[\tilde{\theta}(t)] + \mathrm{Cov}(\theta(t), \tilde{\theta}(t))$. We can then simplify the dynamics as follows

$$\underbrace{\begin{bmatrix} A(\theta(t+1)) \\ \frac{A(\tilde{\theta}(t+1))}{(1-\gamma)^2} \\ \frac{C(t+1)}{(1-\gamma)} \end{bmatrix}}_{\mathbf{v}(t+1)} = \underbrace{\begin{bmatrix} (1-\alpha h)^2 & 0 & 0 \\ (1-\alpha h)^2 & \gamma^2 & 2\gamma(1-\gamma)(1-\alpha h) \\ (1-\alpha h)^2 & 0 & \gamma(1-\alpha h) \end{bmatrix}}_{\text{transition matrix } \mathbf{T}} \underbrace{\begin{bmatrix} A(\theta(t)) \\ \frac{A(\tilde{\theta}(t))}{(1-\gamma)^2} \\ \frac{C(t)}{(1-\gamma)} \end{bmatrix}}_{\mathbf{v}(t)} + \underbrace{\begin{bmatrix} \frac{\alpha^2 c}{B} \\ \frac{\alpha^2 c}{B} \\ \frac{\alpha^2 c}{B} \end{bmatrix}}_{\mathbf{n}} \tag{38}$$

For such a linear dynamical system, we can easily get the $\mathbf{v}(t)$ in the following form:

$$\mathbf{v}(t) = \mathbf{T}^t\mathbf{v}(0) + \sum_{p=1}^{t+1}\mathbf{T}^{p-1}\mathbf{n} \leq \mathbf{T}^t\mathbf{v}(0) + \sum_{p=1}^{\infty}\mathbf{T}^{p-1}\mathbf{n} \tag{39}$$

Now, to get the closed-form of $\mathbf{v}(t)$, we first analyze the second term which involves the infinite sum. For notational convenience, we introduce the following notations:

$$\sum_{p=1}^{\infty}\mathbf{T}^{p-1}\mathbf{n} \triangleq \sum_{p=0}^{\infty}\left[x_p, y_p, z_p\right]^\top \tag{40}$$

In eqn. (40), we append zero vector $[x_0, y_0, z_0]^\top$ for convenience. To compute the infinite sum, we first focus on a single term. We have the following update:

$$\begin{aligned}\sqrt{x_{p+1}} &= (1-\alpha h)\sqrt{x_p} \\ \sqrt{y_{p+1}} &= (1-\alpha h)\sqrt{x_p} + \gamma\sqrt{y_p}\end{aligned} \tag{41}$$

Since we only care y_p which totally decide the loss, so we get rid of x_p by merging two updates, which yields a second-order difference equation:

$$\sqrt{y_{p+1}} = (1-\alpha h+\gamma)\sqrt{y_p} - (1-\alpha h)\gamma\sqrt{y_{p-1}} \tag{42}$$

with initial conditions $\sqrt{y_0} = 0$ and $\sqrt{y_1} = \sqrt{\frac{\alpha^2 c}{B}}$. To solve the second-order difference equation, we leverage the Z-transform to get the analytical form. Based on basic manipulation of the Z-transform, we have the Z-domain function

$$Y(Z) = \frac{\sqrt{\frac{\alpha^2 c}{B}}Z}{Z^2 - (1-\alpha h+\gamma)Z + \gamma} = \frac{\sqrt{\frac{\alpha^2 c}{B}}}{r_1 - r_2}\left(\frac{1}{1-Z^{-1}r_1} - \frac{1}{1-Z^{-1}r_2}\right) \tag{43}$$

where r_1 and r_2 are two roots of equation $z^2 - (1-\alpha h+\gamma)z + (1-\alpha h)\gamma$. Then, we use the inverse Z-transform to get $\sqrt{y_p}$:

$$\sqrt{y_p} = \sqrt{\frac{\alpha^2 c}{B}}\frac{r_1^p - r_2^p}{r_1 - r_2} \tag{44}$$

and therefore

$$y_p = \frac{\alpha^2 c}{B}\frac{r_1^{2p} + r_2^{2p} - 2(r_1 r_2)^p}{(r_1 - r_2)^2} \tag{45}$$

Now, we are ready to compute the infinite sum $\sum_{p=0}^{\infty} y_p$:

$$\begin{aligned}\sum_{p=0}^{\infty} y_p &= \frac{\frac{\alpha^2 c}{B}}{(r_1 - r_2)^2}\left(\frac{1}{1-r_1^2} + \frac{1}{1-r_2^2} - \frac{2}{1-r_1 r_2}\right) \\ &= \frac{\alpha^2 c}{B}\frac{1+r_1 r_2}{(1-r_1^2)(1-r_2^2)(1-r_1 r_2)}\end{aligned} \tag{46}$$

It is easy to see that $r_1 = 1 - \alpha h$ and $r_2 = \gamma$, we then plug them back into eqn. (46) and get

$$\sum_{p=0}^{\infty} y_p = \frac{\alpha c(1 + (1-\alpha h)\gamma)}{Bh(2-\alpha h)(1-\gamma^2)(1-(1-\alpha h)\gamma)} \tag{47}$$

For the other term $\mathbf{T}^t \mathbf{v}(0)$, we can reuse the same second-order difference equation (42) except with initial conditions $\sqrt{y_0} = \sqrt{y_1} = \frac{1}{1-\gamma}\sqrt{A(\theta(0))}$. According to Z-transform, we have

$$y_t = \frac{1}{(1-\gamma)^2}\left(\frac{(r_1^{t+1} - r_2^{t+1}) - \gamma(1-\alpha h)(r_1^t - r_2^t)}{r_1 - r_2}\right)^2 A(\theta(0)) \tag{48}$$

Therefore, we have the following upper bound:

$$A(\tilde{\theta}(t)) \leq \left(\frac{r_1^{t+1} - r_2^{t+1} - \gamma(1-\alpha h)(r_1^t - r_2^t)}{r_1 - r_2}\right)^2 A(\theta(0)) + \frac{\alpha c(1-\gamma)(1+(1-\alpha h)\gamma)}{Bh(2-\alpha h)(1+\gamma)(1-(1-\alpha h)\gamma)} \tag{49}$$

E More results on the NQM

E.1 Eigenspectra of Neural Networks

The main objective of this section is to examine the loss surface of modern neural networks in different stages of training in order to justify the assumptions made in NQM. Nevertheless, it is hard to visualize such a high dimensional space. Following recent work [Sagun et al., 2016, Ghorbani et al., 2019], we instead focus on analyzing the eigenspectrum of the Hessian/Fisher matrices. The Hessian/Fisher of the training loss (with respect to the parameters) is crucial in determining many behaviors of neural networks. The eigenvalues of the Hessian/Fisher characterize the local curvature of the loss surface which determines many training behaviors, including first-order methods optimization rates (at least for convex problems.)

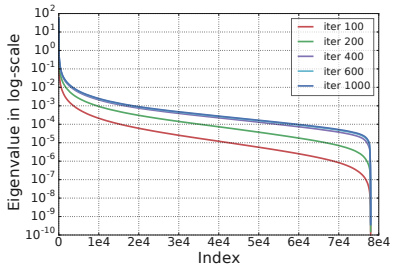


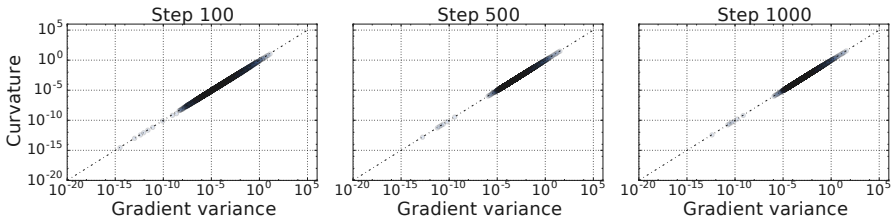
Figure 9: Eigenspectra of the K-FAC approximate Fisher matrix of ResNet8 at different training iterations. The model is trained on CIFAR-10 with batch size 3000.

It has been noted that the *true* Fisher matrix is equivalent to the generalized Gauss-Newton Hessian matrix [Martens, 2014], so we take it as a proxy of the Hessian. To construct the eigenspectrum of the true Fisher matrix, we first leverage the Kronecker-factored approximation of the Fisher to get an estimation of the eigenspectrum, which may shed light upon the true eigenspectrum. Specifically, we train the network with K-FAC and then perform eigen-decomposition on saved Kronecker factors of the Fisher to calculate the eigenvalues.

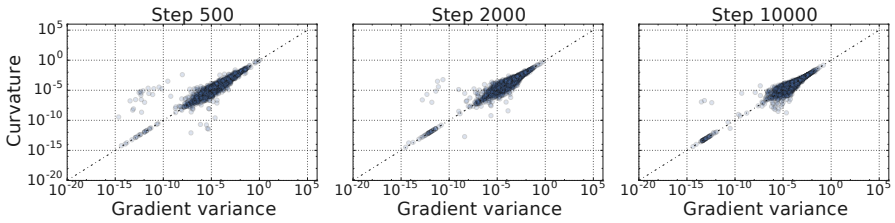
The eigenspectra are plotted in Figure 9. One interesting observation is that there are only a few large eigenvalues and a few small eigenvalues in the approximate Fisher matrices; the bulk of eigenvalues are in the middle of the spectrum. We also note that after 200 iterations of training the eigenspectrum remains mostly unchanged.

E.2 Gradient Covariance in the Kronecker-Factored Eigenbasis

To verify the assumption in Section 3.5 that $\mathbf{H}$ and $\mathbf{C}$ are codiagonalizable, we test it on practical neural networks by comparing the gradient variance to the curvature. This assumption is motivated by theoretical considerations that suggest $\mathbf{H} \approx \mathbf{C}$ for neural network training [Martens, 2014]. Ideally,



(a) ResNet8



(b) Transformer

Figure 10: Scatter plots of second moment v.s. variance of gradients. The gradients are projected onto the Kronecker-factored eigenbasis, which approximates the eigenbasis of the true Fisher. Each point compares the gradient variance and the second moment of the gradient in the direction of an eigenvector of the K-FAC approximated Fisher.

we would like to compare the gradient variance and the curvature of the Fisher in the directions of the eigenvectors of the true Fisher. However, it is typically infeasible to get all these eigenvectors, especially for low curvature directions. To resolve this we instead use the Kronecker-factored eigenbasis [George et al., 2018, Bae et al., 2018, Wang et al., 2019], which is obtained from the K-FAC approximation. For this experiment, we are not relying on this basis being an accurate approximation to the eigendecomposition of the true Fisher; rather, we use the eigenbasis only as a way to obtain a diverse set of directions with both high and low curvature. For a given eigenvector $\mathbf{v}$, we project the gradients $\mathbf{g}$ of each training example onto $\mathbf{v}$ and compute the gradient variance $\mathrm{Cov}(\mathbf{v}^\top \mathbf{g})$, as well as the curvature $\mathbf{v}^\top \mathbf{F} \mathbf{v}$. (The latter quantity can be obtained using matrix-vector products [Schraudolph, 2002].) As shown in Figure 10, the gradient variances closely match the curvature (especially for the ResNet8 model on CIFAR10), validating our assumption that $\mathbf{H} \approx \mathbf{C}$.

E.3 Plots for the Evolution of the First Term in Eqn. (6)

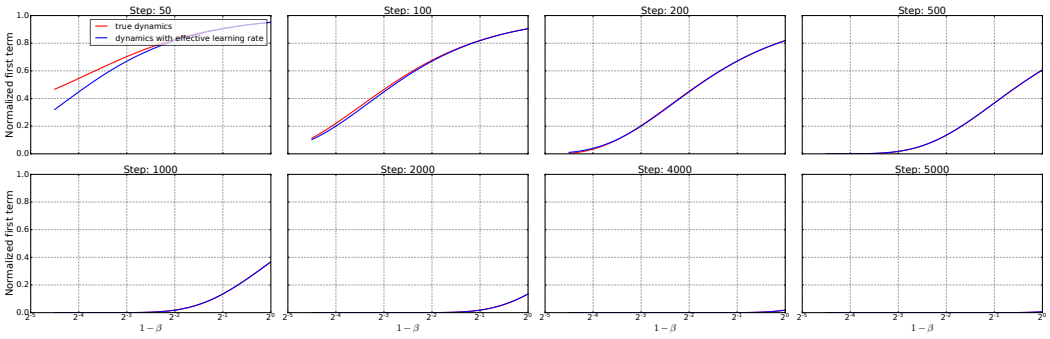


Figure 11: Comparison in convergence between momentum SGD and SGD with adjusted learning rate. This plot shows values for the first term in eqn. (6) as a function of $(1 - \beta)$, which is the scaling between the "effective learning rate" and the true learning rate for momentum SGD. The red curves show the first term when using momentum, while the blue curves show the first term when using plain SGD with the learning rate set to the effective learning rate of momentum.

In Section 3.2, we claim that the convergence of momentum SGD for a single dimension is very close to that of plain SGD with an adjusted learning rate (note that we already verified that the steady state risk of momentum SGD matches plain SGD using effective learning rate in Figure 2). Here we verify this argument by comparing them in the NQM. The total risk consists of two terms (eqn. (6)): the

first term determines convergence, while the second term (steady state risk) stays constant throughout training. Given that the second stays unchanged, we only plot the first term of eqn. (6) in Figure 11. Note that the values are normalized in the figures. We observe that the convergence dynamics of the two update rules closely match each other. For this experiment we set $\alpha h = 0.0005$, but the results are not sensitive to this value.

E.4 Verification of Eigenspectrum

In Section 3.7, we assume the diagonal entries of $\mathbf{H}$ are $\{\frac{1}{i}\}_{i=1}^{d}$. To justify this choice, we compare the K-FAC eigenspectra of ResNet8 to this distribution in Figure 12. The distribution of eigenvalues we chose for $\mathbf{H}$ in the NQM very closely matches the eigenspectra of the real neural network, validating the assumption that the diagonal entries of $\mathbf{H}$ are $\{\frac{1}{i}\}_{i=1}^{d}$ in Section 3.5.

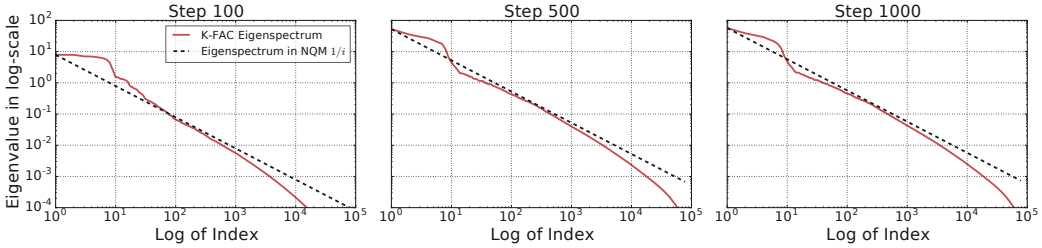


Figure 12: Comparison between K-FAC Fisher eigenspectra and the $\frac{1}{i}$ distribution used in the NQM.

E.5 Effect of Loss Threshold

Recall that a main objective of this work is to characterize the effects of increasing the batch size on training time, as measured in the number of steps necessary to reach a goal target error/loss. Here we experiment with different loss thresholds to study the relationship between batch size and number of training steps. To obtain the minimal training steps for a given batch size, we do grid search over constant learning rates. Figure 13 shows that increasing the batch size initially decreases the required number of training steps proportionally, but eventually there are diminishing returns, which matches the empirical findings [Golmant et al., 2018, Shallue et al., 2018]. The shape of the curves is characteristically the same for different loss thresholds, though the critical batch size seems to increase for more difficult thresholds.

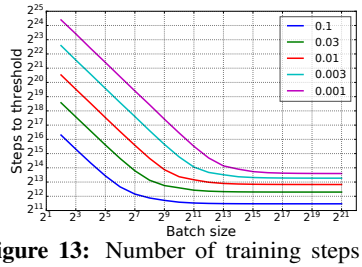
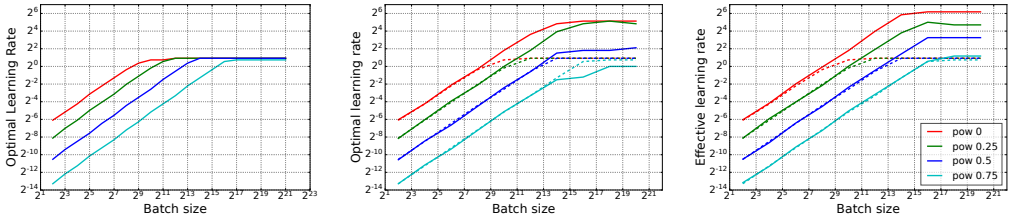


Figure 13: Number of training steps required to reach a target loss as a function of batch size for different loss threshold values.

E.6 Results of Optimal Learning Rate on NQM



(a) Without Momentum **(b)** Fixed Momentum 0.9 **(c)** Tuned Momentum

Figure 14: Optimal learning rate v.s. batch size for different preconditioning powers. (a) When momentum is not used, the learning rate increases with batch size until it is limited by the maximum stable learning rate. Larger preconditioning powers reduce the optimal learning rate for the same batch size, thus extending the batch size where the learning rate levels off. **(b, c)** Fixed (0.9) and tuned momentum values. In (b) and (c), we plot the *effective learning rate* for momentum SGD, defined as $\frac{\alpha}{1-\beta}$. The dashed lines are the same plots from (a) for easier comparison.

E.7 Final Learning Rate of Different Batch Sizes for PWC Learning Rate Scheme

In Section 3.7.2, we study the piecewise constant learning rate scheme. The optimal scheme starts with a high learning rate which drops later in training (Figure 3c). Recall that for fixed learning rates, we observed that the optimal learning rate scaled linearly with the batch size for small batch sizes, but it is unclear whether there is a similar phenomenon for learning rate decay. In Figure 15, we plot the final learning rate as a function of batch size and show that it also scales linearly with batch size.

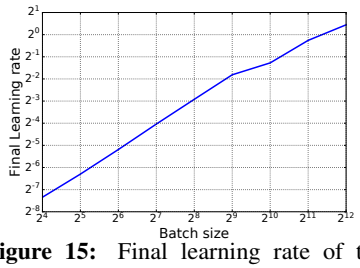


Figure 15: Final learning rate of the piecewise-constant learning rate scheme v.s. batch size.

F More Details for Experiments

F.1 Data Sets

The data sets in Table 1 (MNIST, Fashion MNIST, CIFAR10, ImageNet and LM1B) are identical to those of Shallue et al. [2018] (described in their Appendix A.1). For CIFAR10 we used data augmentation (including horizontal flip and random crop), but they did not.

F.2 Model Details

This section provides details of models in Table 1. The models are very similar (and some identical) to those used in Shallue et al. [2018] (described in their Appendix B). Any modifications from them are highlighted in this section.

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. Unlike in Shallue et al. [2018], we did not use any dropout regularization (while they used dropout with probability 0.4 in the fully connected layer). We used 32 and 64 filters in the convolutional layers and 1,024 units in the fully connected layer. This corresponds to the "base" configuration in Shallue et al. [2018].

ResNet8 [He et al., 2016] consists of 7 convolutional layers with residual connections followed by 1 fully connected hidden layer. We used the identical architecture as Shallue et al. [2018]. In particular, we did not use batch normalization. The only difference is that we used data augmentation in our experiments.

ResNet32 [He et al., 2016] consists of 31 convolutional layers with residual connections followed by 1 fully connected hidden layer (see Section 4.2 of He et al. [2016]). 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. We set the label smoothing parameter to 0.1 in all experiments. Instead of using weight decay, we applied channel-wise weight normalization by constraining the Frobenius norm of each convolutional channel to be exactly 1, which controls the effective learning rate [Zhang et al., 2019b, van Laarhoven, 2017].

VGG11 [Simonyan and Zisserman, 2015] consists of 8 convolutional layers followed by 1 fully connected hidden layers. as in ResNet32, we used Ghost batch normalization, label smoothing, and channel-wise weight normalization.

Transformer Vaswani et al. [2017] is a self-attention model. We chose the Transformer model identical to the "base" model described in Vaswani et al. [2017], except with only two hidden layers instead of six. This is identical to the "Transformer Shallow" model in Shallue et al. [2018].

F.3 Learning Rate Schedules

This section describes two learning rate schedules mentioned in Table 1: constant schedule and linear decay schedule. Constant schedule simply keeps a fixed learning rate throughout training:

$$\alpha(t) = \alpha_0,$$

where t is the training step index. Linear decay schedule is

$$\alpha(t) = \alpha_0 - (1 - \gamma)\frac{t}{T},$$

where α_0 is the initial learning rate, γ is the rate of decay, and T is the number of steps taken to reach the final learning rate. Shallue et al. [2018] experimented with various learning rate schedules and found that linear decay matched performance of the other schedules with fewer hyperparameters to tune. Therefore, we also chose the linear decay schedule, for which we tuned α_0, γ and T.

F.4 Optimizer-Specific Hyperparamters

For momentum SGD, we tuned the momentum β. For Adam, we tuned β_1, β_2, and ϵ (see Kingma and Ba [2014]). For K-FAC, we tuned damping and the trust region constraint (also known as the KL clipping term) for Transformer, keeping momentum $= 0.9$ and the moving average parameter for damping $= 0.99$; for all other models, we tuned all four parameters (see Martens and Grosse [2015]).