GRADIENT DESCENT ON NEURAL NETWORKS TYPICALLY OCCURS AT THE EDGE OF STABILITY

Jeremy Cohen **Simran Kaur** **Yuanzhi Li** **J. Zico Kolter**[1] **and** **Ameet Talwalkar**[2]
Carnegie Mellon University and: [1]Bosch AI [2] Determined AI
Correspondence to: `jeremycohen@cmu.edu`

ABSTRACT

We empirically demonstrate that full-batch gradient descent on neural network training objectives typically operates in a regime we call the Edge of Stability. In this regime, the maximum eigenvalue of the training loss Hessian hovers just above the value $2/(\text{step size})$, and the training loss behaves non-monotonically over short timescales, yet consistently decreases over long timescales. Since this behavior is inconsistent with several widespread presumptions in the field of optimization, our findings raise questions as to whether these presumptions are relevant to neural network training. We hope that our findings will inspire future efforts aimed at rigorously understanding optimization at the Edge of Stability.

1 INTRODUCTION

Neural networks are almost never trained using (full-batch) gradient descent, even though gradient descent is the conceptual basis for popular optimization algorithms such as SGD. In this paper, we train neural networks using gradient descent, and find two surprises. First, while little is known about the dynamics of neural network training in general, we find that in the special case of gradient descent, there is a simple characterization that holds across a broad range of network architectures and tasks. Second, this characterization is strongly at odds with prevailing beliefs in optimization.

In more detail, as we train neural networks using gradient descent with step size η, we measure the evolution of the *sharpness* — the maximum eigenvalue of the training loss Hessian. Empirically, the behavior of the sharpness is consistent across architectures and tasks: so long as the sharpness is less than the value $2/\eta$, it tends to continually rise (§3.1). We call this phenomenon *progressive sharpening*. The significance of the value $2/\eta$ is that gradient descent on quadratic objectives is unstable if the sharpness exceeds this threshold (§2). Indeed, in neural network training, if the sharpness ever crosses $2/\eta$, gradient descent quickly becomes destabilized — that is, the iterates start to oscillate with ever-increasing magnitude along the direction of greatest curvature. Yet once

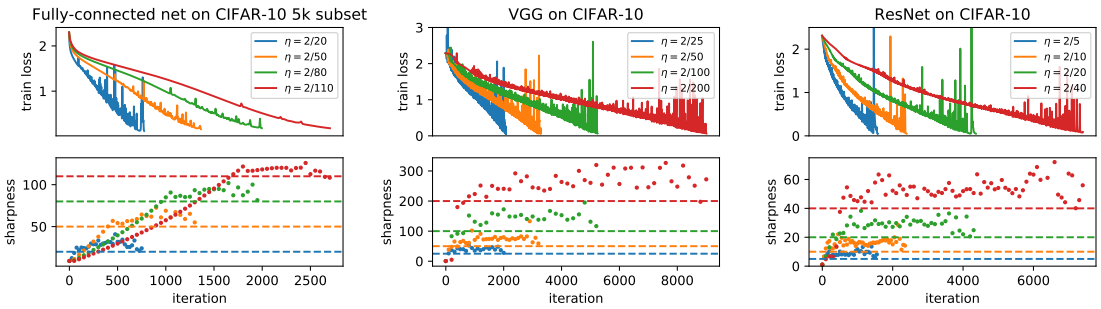

Figure 1: **Gradient descent typically occurs at the Edge of Stability.** On three separate architectures, we run gradient descent at a range of step sizes η, and plot both the train loss (top row) and the sharpness (bottom row). For each step size η, observe that the sharpness rises to $2/\eta$ (marked by the horizontal dashed line of the appropriate color) and then hovers right at, or just above, this value.

this happens, gradient descent does not diverge entirely or stall. Instead, it enters a new regime we call the *Edge of Stability* (§3.2), in which (1) the sharpness hovers right at, or just above, the value $2/\eta$; and (2) the train loss behaves non-monotonically, yet consistently decreases over long timescales. In this regime, gradient descent is constantly "trying" to increase the sharpness, but is constantly restrained from doing so. The net effect is that gradient descent continues to successfully optimize the training objective, but in such a way as to avoid further increasing the sharpness.[1]

In principle, it is possible to run gradient descent at step sizes η so small that the sharpness never rises to $2/\eta$. However, these step sizes are suboptimal from the point of view of training speed, sometimes dramatically so. In particular, for standard architectures on the standard dataset CIFAR-10, such step sizes are so small as to be completely unreasonable — at all reasonable step sizes, gradient descent eventually enters the Edge of Stability (see §4). Thus, at least for standard networks on CIFAR-10, the Edge of Stability regime should be viewed as the "rule," not the "exception."

As we describe in §5, the Edge of Stability regime is inconsistent with several pieces of conventional wisdom in optimization theory: convergence analyses based on L-smoothness or monotone descent, quadratic Taylor approximations as a model for local progress, and certain heuristics for step size selection. We hope that our empirical findings will both nudge the optimization community away from widespread presumptions that appear to be untrue in the case of neural network training, and also point the way forward by identifying precise empirical phenomena suitable for further study.

Certain aspects of the Edge of Stability have been observed in previous empirical studies of full-batch gradient descent (Xing et al., 2018; Wu et al., 2018); our paper provides a unified explanation for these observations. Furthermore, Jastrzebski et al. (2020) proposed a simplified model for the evolution of the sharpness during *stochastic* gradient descent which matches our empirical observations in the special case of full-batch SGD (i.e. gradient descent). However, outside the full-batch special case, there is no evidence that their model matches experiments with any degree of quantitative precision, although their model does successfully predict the *directional* trend that large step sizes and/or small batch sizes steer SGD into regions of low sharpness. We discuss SGD at greater length in §6. To summarize, while the sharpness does not obey simple dynamics during SGD (as it does during GD), there are indications that the "Edge of Stability" intuition might generalize somehow to SGD, just in a way that does not center around the sharpness.

2 Background: Stability of Gradient Descent on Quadratics

In this section, we review the stability properties of gradient descent on quadratic functions. Later, we will see that the stability of gradient descent on neural training objectives is partly well-modeled by the stability of gradient descent on the quadratic Taylor approximation.

On a quadratic objective function $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c$, gradient descent with step size η will diverge if[2] any eigenvalue of $\mathbf{A}$ exceeds the threshold $2/\eta$. To see why, consider first the one-dimensional quadratic $f(x) = \frac{1}{2}ax^2 + bx + c$, with $a > 0$. This function has optimum $x^* = -b/a$. Consider running gradient descent with step size η starting from x_0. The update rule is $x_{t+1} = x_t - \eta(ax_t + b)$, which means that the error $x_t - x^*$ evolves as $(x_{t+1} - x^*) = (1 - \eta a)(x_t - x^*)$. Therefore, the error at step t is $(x_t - x^*) = (1 - \eta a)^t(x_0 - x^*)$, and so the iterate at step t is $x_t = (1 - \eta a)^t(x_0 - x^*) + x^*$. If $a > 2/\eta$, then $(1 - \eta a) < -1$, so the sequence $\{x_t\}$ will oscillate around x^* with ever-increasing magnitude, and diverge.

Now consider the general d-dimensional case. Let $(a_i, \mathbf{q}_i)$ be the i-th largest eigenvalue/eigenvector of $\mathbf{A}$. As shown in Appendix A, when the gradient descent iterates $\{\mathbf{x}_t\}$ are expressed in the special coordinate system whose axes are the eigenvectors of $\mathbf{A}$, each coordinate evolves separately. In particular, the coordinate for each eigenvector $\mathbf{q}_i$, namely $\langle \mathbf{q}_i, \mathbf{x}_t \rangle$, evolves according to the dynamics of gradient descent on a one-dimensional quadratic objective with second derivative a_i. Therefore, if $a_i > 2/\eta$, then the sequence $\{\langle \mathbf{q}_i, \mathbf{x}_t \rangle\}$ will oscillate with ever-increasing magnitude; that is, the iterates $\{\mathbf{x}_t\}$ will oscillate along the direction $\mathbf{q}_i$.

[1] In the literature, the term "sharpness" has been used to refer to a variety of quantities, often connected to generalization (e.g. Keskar et al. (2016)). In this paper, "sharpness" strictly means the maximum eigenvalue of the training loss Hessian. We do not claim that this quantity has any connection to generalization.

[2] For *convex* quadratics, this is "if and only if." However, if the Hessian of a quadratic objective has a negative eigenvalue, then gradient descent with any (positive) step size will diverge along that direction.

To illustrate, Figure 2 shows a quadratic function with eigenvalues $a_1 = 20$ and $a_2 = 1$. In Figure 2(a), we run gradient descent with step size $\eta = 0.09$; since $0 < a_2 < a_1 < 2/\eta$, gradient descent converges along both $\mathbf{q}_1$ and $\mathbf{q}_2$. In Figure 2(b), we use step size $\eta = 0.11$; since $0 < a_2 < 2/\eta < a_1$, gradient descent converges along $\mathbf{q}_2$ yet diverges along $\mathbf{q}_1$, so diverges overall.

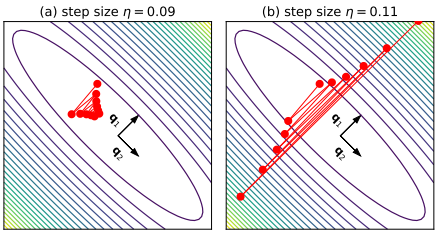

Figure 2: Gradient descent on a quadratic with eigenvalues $a_1 = 20$ and $a_2 = 1$.

Polyak momentum (Polyak, 1964) and Nesterov momentum (Nesterov, 1983; Sutskever et al., 2013) are notable variants of gradient descent which often improve the convergence speed. On quadratic functions, these two algorithms also diverge if the sharpness exceeds a certain threshold, which we call the "maximum stable sharpness," or MSS. In particular, we prove in Appendix A that gradient descent with step size η and momentum parameter β diverges if the sharpness exceeds:

$$\mathrm{MSS}_{\mathrm{Polyak}}(\eta, \beta) = \frac{1}{\eta}(2 + 2\beta), \quad \mathrm{MSS}_{\mathrm{Nesterov}}(\eta, \beta) = \frac{1}{\eta}\left(\frac{2 + 2\beta}{1 + 2\beta}\right). \tag{1}$$

The Polyak result previously appeared in Goh (2017); the Nesterov one seems to be new. Giladi et al. (2020) derived an analogous stability result for gradient descent with stale gradients. However, note that not every optimization algorithm has its stability on quadratics determined by a sharpness threshold; this is only true for algorithms that act independently along each eigenvector of $\mathbf{A}$. Moreover, the above discussion only applies to full-batch gradient descent. It is not entirely clear if or how similar concerns apply to SGD, though, as we discuss in §6, several recent papers have proposed stability analyses for SGD (Wu et al., 2018; Jastrzebski et al., 2020; Giladi et al., 2020).

Neural network training objectives are not globally quadratic. However, the second-order Taylor approximation around any point $\mathbf{x}_0$ in parameter space is a quadratic function whose "$\mathbf{A}$" matrix is the Hessian at $\mathbf{x}_0$. If any eigenvalue of this Hessian exceeds $2/\eta$, gradient descent with step size η would diverge if run on this quadratic function — the iterates would oscillate with ever-increasing magnitude along the corresponding eigenvector. Therefore, at any point $\mathbf{x}_0$ in parameter space where the sharpness exceeds $2/\eta$, gradient descent with step size η would diverge if run on the quadratic Taylor approximation to the training objective around $\mathbf{x}_0$.

3 Gradient Descent on Neural Networks

In this section, we empirically characterize the behavior of gradient descent on neural network training objectives. Section 4 will show that this characterization holds broadly.

3.1 Progressive sharpening

When training neural networks, it seems to be a general rule that **so long as the sharpness is small enough for gradient descent to be stable** ($< 2/\eta$, for vanilla gradient descent), **gradient descent has an overwhelming tendency to continually increase the sharpness**. We call this phenomenon *progressive sharpening*. By "overwhelming tendency," we mean that gradient descent can occasionally decrease the sharpness (especially at the beginning of training), but these brief decreases always seem be followed by a return to continual increase. Jastrzebski et al. (2020) previously hypothesized

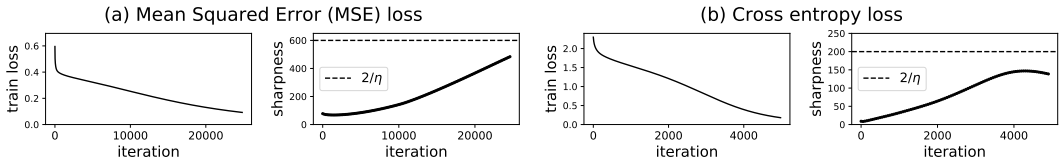

Figure 3: **So long as the sharpness is less than $2/\eta$, it tends to continually increase during gradient descent.** We use gradient descent with a very small step size to train a network to completion (99% accuracy), under both MSE loss (**left**) and cross-entropy loss (**right**).

(in their Assumption 4) that a similar phenomenon may hold for SGD, but the evidence for, and the precise scope of, this effect are currently far clearer for gradient descent than for SGD.

Progressive sharpening is illustrated in Figure 3. Here, we use (full-batch) gradient descent to train a network on a subset of 5,000 examples from CIFAR-10, and we monitor the evolution of the sharpness during training. The network is a fully-connected architecture with two hidden layers of width 200, and tanh activations. In Figure 3(a), we train using the mean squared error loss for classification (Hui & Belkin, 2020), encoding the correct class with 1 and the other classes with 0. We use the small step size of $\eta = 2/600$, and stop when the training accuracy reaches 99%. We plot both the train loss and the sharpness, with a horizontal dashed line marking the stability threshold $2/\eta$. Observe that the sharpness continually rises during training (except for a brief dip at the beginning). This is progressive sharpening. For this experiment, we intentionally chose a step size η small enough that the sharpness remained beneath $2/\eta$ for the entire duration of training.

Cross-entropy. When training with cross-entropy loss, there is an exception to the rule that the sharpness tends to continually increase: with cross-entropy loss, the sharpness typically drops at the end of training. This behavior can be seen in Figure 3(b), where we train the same network using the cross-entropy loss rather than MSE. This drop occurs because once most data points are classified correctly, gradient descent tries to drive the cross-entropy loss to zero by scaling up the margins, as detailed in Soudry et al. (2018). As we explain in Appendix B, this causes the sharpness to drop.

The effect of width. It is known that when networks parameterized in a certain way (the "NTK parameterization") are made infinitely wide, the Hessian moves a vanishingly small amount during training (Jacot et al., 2018; Lee et al., 2019; Li & Liang, 2019), which implies that no progressive sharpening occurs. In Appendix C, we experiment with networks of varying width, under both NTK and standard parameterizations. We find that progressive sharpening occurs to a lesser degree as networks become increasingly wide. Nevertheless, our experiments in §4 demonstrate that progressive sharpening occurs to a dramatic degree for standard architectures on the standard dataset CIFAR-10.

3.2 The Edge of Stability

In the preceding section, we ran gradient descent using step sizes η so small that the sharpness never reached the stability threshold $2/\eta$. In Figure 4(a), we start to train the same network at the larger step size of $\eta = 0.01$, and pause training once the sharpness rises to $2/\eta = 200$. Recall from §2 that in any region where the sharpness exceeds $2/\eta$, gradient descent with step size η would be unstable if run on the quadratic Taylor approximation to the training objective — the gradient descent iterates would oscillate with ever-increasing magnitude along the leading Hessian eigenvector. Empirically, we find that gradient descent on the real neural training objective behaves similarly — at first. Namely, let $\mathbf{q}_1$ be the leading Hessian eigenvector at the iteration where the sharpness reaches $2/\eta$. In Figure 4(b), we resume training the network, and we monitor both the train loss and the quantity $\langle \mathbf{q}_1, \mathbf{x}_t \rangle$ for the next 215 iterations. Observe that $\langle \mathbf{q}_1, \mathbf{x}_t \rangle$ oscillates with ever-increasing magnitude, similar to the divergent quadratic example in Figure 2(b). At first, these oscillations are too small to affect the objective appreciably, and so the train loss continues to monotonically decrease. But eventually, these oscillations grow big enough that the train loss spikes.

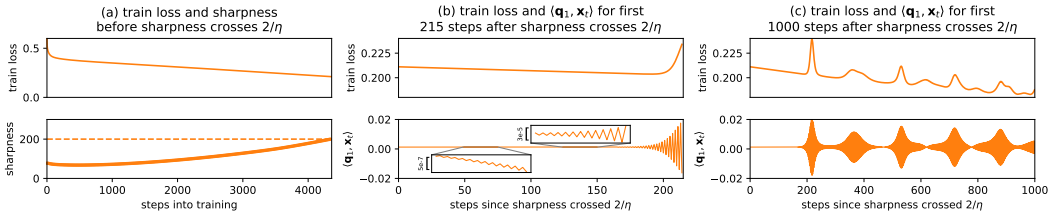

Figure 4: **Once the sharpness crosses $2/\eta$, gradient descent becomes destabilized**. We run gradient descent at $\eta = 0.01$. **(a)** The sharpness eventually reaches $2/\eta$. **(b)** Once the sharpness crosses $2/\eta$, the iterates start to oscillate along $\mathbf{q}_1$ with ever-increasing magnitude. **(c)** Somehow, GD does not diverge entirely; instead, the train loss continues to decrease, albeit non-monotonically.

Once gradient descent becomes destabilized in this manner, classical optimization theory gives no clues as to what will happen next. One might imagine that perhaps gradient descent might diverge entirely, or that gradient descent might stall while failing to make progress, or that gradient descent might jump to a flatter region and remain there. In reality, none of these outcomes occurs. In Figure 4(c), we plot both the train loss and $\langle \mathbf{q}_1, \mathbf{x}_t \rangle$ for 1000 iterations after the sharpness first crossed $2/\eta$. Observe that gradient descent somehow avoids diverging entirely. Instead, after initially spiking around iteration 215, the train loss continues to decrease, albeit non-monotonically.

This numerical example is representative. In general, after the sharpness initially crosses $2/\eta$, gradient descent enters a regime we call the *Edge of Stability*, in which (1) the sharpness hovers right at, or just above, the value $2/\eta$; and (2) the train loss behaves non-monotonically over short timescales, yet decreases consistently over long timescales. Indeed, in Figure 5, we run gradient descent at a range of step sizes using both MSE and cross-entropy loss. The left plane plots the train loss curves, with a vertical dotted line (of the appropriate color) marking the iteration where the sharpness first crosses $2/\eta$. Observe that the train loss decreases monotonically before this dotted line, but behaves non-monotonically afterwards. The middle plane plots the evolution of the sharpness, with a horizontal dashed line (of the appropriate color) at the value $2/\eta$. Observe that once the sharpness reaches $2/\eta$, it ceases to increase further, and instead hovers right at, or just above, the value $2/\eta$ for the remainder of training. (The precise meaning of "just above" varies: in Figure 5, for MSE loss, the sharpness hovers just a minuscule amount above $2/\eta$, while for cross-entropy loss, the gap between the sharpness and $2/\eta$ is small yet non-miniscule.)

At the Edge of Stability, gradient descent is constantly "trying" to increase the sharpness (progressive sharpening), but is constantly being restrained from doing so. To demonstrate this, in Figure 7, we train at step size $2/200$ until reaching the Edge of Stability, and then at iteration 6,000 (marked by the vertical black line), we drop the step size to $\eta = 2/300$. Observe that after the learning rate drop, the sharpness immediately starts to increase, and only stops increasing once gradient descent is back at the Edge of Stability. Appendix M repeats this experiment on more architectures. Intuitively, gradient descent with step size η can be viewed as a constrained optimization algorithm: there is an implicit $2/\eta$ constraint on the sharpness that is "active" at the Edge of Stability.

Observe from Figure 5 that there do exist step sizes η (in purple) small enough that the sharpness never rises to $2/\eta$. We call such a step size *stable*. However, observe that with cross-entropy loss, it takes 3700 iterations to train at the stable step size in purple, but only 1000 iterations to train at the larger step size in blue. In general, we always observe that stable step sizes are suboptimal in terms

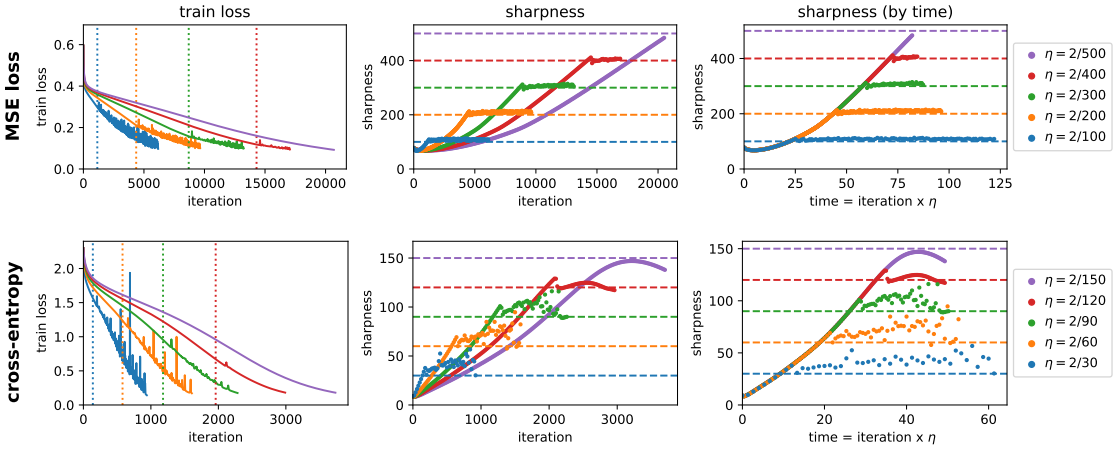

Figure 5: **After the sharpness reaches $2/\eta$, gradient descent enters the Edge of Stability**. A network is trained with gradient descent at a range of step sizes (see legend), using both MSE loss (**top row**) and cross-entropy (**bottom row**). **Left**: the train loss curves, with a vertical dotted line at the iteration where the sharpness first crosses $2/\eta$. **Center**: the sharpness, with a horizontal dashed line at the value $2/\eta$. **Right**: sharpness plotted by time (= iteration $\times$ η) rather than iteration.

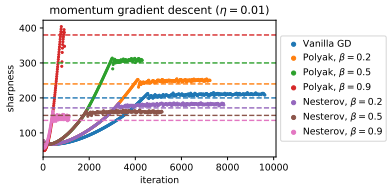

Figure 6: **Momentum.** We run GD with step size $\eta = 0.01$ and Polyak or Nesterov momentum at various β. For each algorithm, the horizontal dashed line marks the MSS from Equation 1.

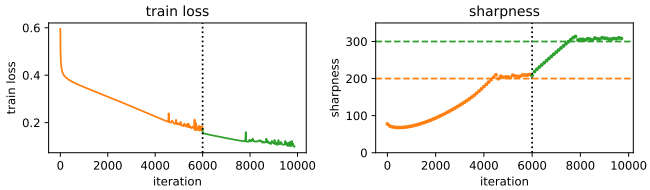

Figure 7: **After a learning rate drop, progressive sharpening resumes**. We start training at $\eta = 2/200$ (orange) and then after 6000 iterations (dotted vertical black line), we cut the step size to $\eta = 2/300$ (green). Observe that as soon as the step size is cut, the sharpness starts to rise.

of convergence speed. In fact, in §4 we will see that for standard networks on CIFAR-10, stable step sizes are so suboptimally small that they are completely unreasonable.

The "Edge of Stability" effect generalizes to gradient descent with both Polyak and Nesterov momentum. In Figure 6, we train using gradient descent with step size $\eta = 0.01$, and several different momentum settings. Observe that in each case, the sharpness rises until reaching the MSS given by Equation 1, and then plateaus there. Appendix L has more momentum experiments.

Prior work. Aspects of the Edge of Stability have been observed previously in the literature. Wu et al. (2018) noted that the sharpness at the solution reached by full-batch gradient descent was not just *less than* $2/\eta$, as was expected due to stability considerations, but was mysteriously *approximately equal* to $2/\eta$. In retrospect, we can attribute this observation to progressive sharpening. Xing et al. (2018) observed that full-batch gradient descent eventually enters a regime (the Edge of Stability) in which the training loss behaves non-monotonically, and the iterates oscillate along the direction of largest curvature; however, they did not relate this regime to the sharpness. Lewkowycz et al. (2020) found that in neural network training, if the sharpness at initialization is larger than $2/\eta$, then after becoming initially destabilized, gradient descent does not always diverge entirely (as the quadratic Taylor approximation would suggest), but rather sometimes "catapults" into a flatter region that is flat enough to stably accommodate the step size. It seems plausible that whichever properties of neural training objectives permit this so-called "catapult" behavior may also be the same properties that permit successful optimization at the Edge of Stability. Indeed, optimization at the Edge of Stability can conceivably be viewed as a never-ending series of micro-catapults. Finally, as we discuss at greater length in §6, several papers (Jastrzebski et al., 2017; 2019) have observed that large step sizes steer *stochastic* gradient descent into less sharp regions of the loss landscape, and Jastrzebski et al. (2020) attributed this effect to the stability properties of SGD.

3.3 THE GRADIENT FLOW TRAJECTORY

In the right pane of Figure 5, we plot the evolution of the sharpness during gradient descent, with "time" = iteration $\times\, \eta$, rather than iteration, on the x-axis. This allows us to directly compare the sharpness after, say, 100 iterations at $\eta = 0.01$ to the sharpness after 50 iterations at $\eta = 0.02$; both are time 1. Observe that when plotted by time, the sharpnesses for gradient descent at different step sizes coincide until the time where each reaches $2/\eta$. This is because for this network, gradient descent at $\eta = 0.01$ and gradient descent at $\eta = 0.02$ initially travel the same path (moving at a speed proportional to η) until each reaches the point on that path where the sharpness hits $2/\eta$. This path is the *gradient flow trajectory*. The gradient flow iterate at time t is defined as the limit as $\eta \to 0$ of the gradient descent iterate at iteration t/η (if this limit exists). The empirical finding of interest is that for this particular network, gradient descent does not only track the gradient flow trajectory in the limit of infinitesimally small step sizes, but for any step size that is less than $2/\text{sharpness}$.

We can numerically approximate gradient flow trajectories by using the Runge-Kutta RK4 algorithm (Press et al., 1992) to numerically integrate the gradient flow ODE. Empirically, for many <u>*but not all*</u> networks studied in this paper, we find that gradient descent at any step size η closely tracks the Runge-Kutta trajectory until reaching the point on that trajectory where the sharpness hits $2/\eta$. (This sometimes occurs even for networks with ReLU activations or max-pooling, which give rise

to training objectives that are not continuously differentiable, which means that the gradient flow trajectory is not necessarily guaranteed to exist.) For such networks, the gradient flow trajectory provides a coherent framework for reasoning about which step sizes will eventually enter the Edge of Stability. Let λ_0 be the sharpness at initialization, and let $\lambda_{\max}$ be the maximum sharpness along the gradient flow trajectory. If $\eta < 2/\lambda_{\max}$, then gradient descent will stably track the gradient flow trajectory for the entire duration of training, and will never enter the Edge of Stability. On the other hand, if $\eta \in [2/\lambda_{\max}, 2/\lambda_0]$, then gradient descent will stably track the gradient flow trajectory only until reaching the point on that trajectory where the sharpness hits $2/\eta$; shortly afterwards, gradient descent will become destabilized, depart the gradient flow trajectory, and enter the Edge of Stability.

4 FURTHER EXPERIMENTS

Section 3 focused for exposition on a single architecture and task. In this section, we show that our characterization of gradient descent holds broadly across a wide range of architectures and tasks.

Architectures. In Appendix H, we fix the task of training a 5k subset of CIFAR-10, and we systematically vary the network architecture. We consider fully-connected networks, as well as convolutional networks with both max-pooling and average pooling. For all of these architectures, we consider tanh, ReLU, and ELU activations, and for fully-connected networks we moreover consider softplus and hardtanh —- eleven networks in total. We train each network with both cross-entropy and MSE loss. In each case, we successfully reproduce Figure 5. Additionally, we observe that for architectures with continuously differentiable components (as well as some, but not all, with ReLU, hardtanh, and max-pooling), gradient descent with step size η closely hews to the Runge-Kutta trajectory until the point on that trajectory where the sharpness crosses $2/\eta$.

Since batch normalization (Ioffe & Szegedy, 2015) is known to have unusual optimization properties (Li & Arora, 2019), it is natural to wonder whether our findings still hold with batch normalization. In Appendix I, we confirm that they do, and we reconcile this point with Santurkar et al. (2018).

Tasks. In Appendix J, we verify our findings on: (1) a Transformer trained on the WikiText-2 language modeling task; (2) fully-connected tanh networks with one hidden layer, trained on a synthetic one-dimensional toy regression task; and (3) deep linear networks trained on Gaussian data. In each case, the sharpness rises until hovering right at, or just above, the value $2/\eta$. However, for the deep linear network and the Transformer, we observe that gradient descent departs substantially from the Runge-Kutta trajectory even when the sharpness is less than $2/\eta$.

Standard networks on CIFAR-10. In Appendix K, we verify our findings on three standard architectures trained on the full CIFAR-10 dataset: a ResNet with BN, a VGG with BN, and a VGG without BN. We also find that gradient descent closely tracks the Runge-Kutta trajectory so long as the sharpness remains less than $2/\eta$. Furthermore, for all three architectures, we find that progressive sharpening occurs to a dramatic degree, and, relatedly, that stable step sizes (those small enough that the sharpness remains below $2/\eta$ for the entire duration of training) are dramatically suboptimal. For example, when we train the VGG-BN to 99% accuracy using gradient flow (i.e. Runge-Kutta), we find that the sharpness rises from 6.3 at initialization to a peak sharpness of 2227.5. Consequently, any stable step size for gradient descent must be less than $2/2227.5 = 0.000897$. Since training finishes at time 15.66, gradient descent at any stable step size would require at least $15.66/0.000897 = 17,458$ iterations. Yet empirically, this network can be trained to completion at the larger, "Edge of Stability" step size of $\eta = 0.16$ in just 331 iterations. Therefore, training at a stable step size is suboptimal by a factor of at least $17458/331 = 52.7$. The situation is similar for the other two architectures we consider. In short, for standard architectures on the standard dataset CIFAR-10, stable step sizes are not just suboptimally small, they are so suboptimal as to be completely unreasonable. For these networks, gradient descent at any reasonable step size eventually enters the Edge of Stability regime.

5 DISCUSSION

We now explain why the behavior of gradient descent at the Edge of Stability contradicts several pieces of conventional wisdom in optimization.

The hidden second role of gradient descent At the Edge of Stability, gradient descent performs two functions simultaneously: it successfully optimizes the train loss at the same time that it constantly restrains itself from following gradient flow into a region of increased sharpness. To the best of our knowledge, existing mathematical analyses of gradient descent only model the first function, and never the second. Therefore, these analyses are necessarily "missing something" in spirit.

At reasonable step sizes, gradient descent cannot be analyzed using (even local) L-smoothness Many convergence analyses of gradient descent assume a bound on the sharpness — either globally or, at the very least, along the optimization trajectory. This condition, called L-smoothness (Nesterov, 1998), is intended to guarantee that each gradient step will decrease the training objective by a certain amount; the weakest guarantee is that if the local sharpness is less than $2/\eta$, then a gradient step with size η is guaranteed to decrease (rather than increase) the training objective. At a bare minimum, any convergence analysis of gradient descent based on L-smoothness will require the sharpness along the optimization trajectory to be less than $2/\eta$. Yet to the contrary, at the Edge of Stability, the sharpness hovers just above $2/\eta$. Therefore, at any step size for which gradient descent enters the Edge of Stability (which, on realistic architectures, includes any reasonable step size), gradient descent cannot be analyzed using L-smoothness. Li et al. (2020) previously argued that convergence analyses based on L-smoothness do not apply to networks with both batch normalization and weight decay; our paper empirically extends this to neural networks without either.

L-smoothness *may* be inappropriate when analyzing other optimization algorithms too It is common for optimization papers seemingly motivated by deep learning to analyze algorithms under the "non-convex but L-smooth"' setting (Reddi et al., 2016; Agarwal et al., 2017; Zaheer et al., 2018; Zhou et al., 2018; Chen et al., 2018; Li & Orabona, 2019; Ward et al., 2019; You et al., 2020; Vaswani et al., 2019; Sankararaman et al., 2019; Reddi et al., 2020; Défossez et al., 2020; Xie et al., 2020a; Liu et al., 2020; Defazio, 2020). Since our experiments focus on gradient descent, it does not *necessarily* follow that L-smoothness assumptions are unjustified when analyzing other optimization algorithms. However, gradient descent is arguably the simplest optimization algorithm, so we believe that the fact that (even local) L-smoothness fails even there should raise serious questions about the suitability of the L-smoothness assumption in neural network optimization more generally. In particular, the burden of proof should be on authors to empirically justify that their L-smoothness assumption holds throughout training.

At reasonable step sizes, gradient descent does not monotonically decrease the training loss In neural network training, SGD does not monotonically decrease the training objective, in part due to minibatch randomness. However, it is often assumed that *full-batch* gradient descent *would* monotonically decrease the training objective, were it used to train neural networks. For example, Zhang et al. (2020) proposed a "relaxed L-smoothness" condition that is less restrictive than standard L-smoothness, and proved a convergence guarantee for gradient descent under this condition which asserted that the training objective will decrease monotonically. Likewise, the analysis in Allen-Zhu et al. (2019) also implies that gradient descent will monotonically decrease the training objective. Yet, at the Edge of Stability, the training loss behaves non-monotonically over short timescales even as it consistently decreases over long timescales. Therefore, convergence analyses which assert monotone descent cannot possibly apply to gradient descent at reasonable step sizes.

The Edge of Stability is inherently non-quadratic It is tempting to try to reason about the behavior of gradient descent on neural network training objectives by analyzing, as a proxy, the behavior of gradient descent on the local quadratic Taylor approximation (LeCun et al., 1998). However, at the Edge of Stability, the behavior of gradient descent on the real neural training objective is irreconcilably different from the behavior of gradient descent on the quadratic Taylor approximation: the former makes consistent (if choppy) progress, whereas the latter would diverge (and this divergence would happen quickly, as we demonstrate in Appendix D). Thus, the behavior of gradient descent at the Edge of Stability is inherently non-quadratic.

Dogma for step size selection may be unjustified An influential piece of conventional wisdom concerning step size selection has its roots in the quadratic Taylor approximation model of gradient descent. This conventional wisdom (LeCun et al., 1998; Schaul et al., 2013) holds that if the sharpness at step t is λ_t, then the current step size η_t must be set no greater than $2/\lambda_t$ (in order to

prevent divergence); and furthermore, barring additional information about the objective function, that η_t should optimally be set to $1/\lambda_t$. Our findings complicate this conventional wisdom. To start, it is impossible to satisfy these prescriptions with a fixed step size: for any fixed (and reasonable) step size $\eta_t = \eta$, progressive sharpening eventually drives gradient descent into regions where the sharpness is just a bit greater than $2/\eta$ — which means that the step size η is purportedly impermissible. Furthermore, in Appendix E, we try running gradient descent with the purportedly optimal $\eta_t = 1/\lambda_t$ rule, and find that this algorithm is soundly outperformed by the purportedly impermissible baseline of gradient descent with a fixed $\eta_t = 1/\lambda_0$ step size, where λ_0 is the sharpness at initialization. The $\eta_t = 1/\lambda_t$ rule continually anneals the step size, and in so doing ensures that the training objective will decrease at each iteration, whereas the fixed $\eta_t = 1/\lambda_0$ step size often increases the training objective. However, this non-monotonicity turns out to be a worthwhile price to pay in return for the ability to take larger steps.

6 Stochastic gradient descent

Empirically, our precise characterization of the behavior of the sharpness only holds for (full-batch) gradient descent. In contrast, during SGD, the sharpness does not always settle at any fixed value, much less one that can be numerically predicted based on the step size alone; for example, Figure 5(a) of Jastrzebski et al. (2020) exhibits a network where the sharpness gradually drifts downwards during SGD training. Nevertheless, prior works (Jastrzebski et al., 2017; 2019; 2020) have demonstrated that the step size parameter of SGD does seem to impact the sharpness, with large step sizes (as well as small batch sizes) steering SGD into regions of the landscape with lower sharpness. The $2/\eta$ rule we have identified for full-batch gradient descent is a special case of this observation.

Even though our findings only strictly hold for gradient descent, they may have relevance to SGD as well. First, since gradient descent is a special case of SGD, any general characterization of the dynamics of SGD must reduce to the "Edge of Stability" in the full-batch special case. Second, there are indications that some of the effects we have described may have analogues for SGD. In particular, in Appendix F, we demonstrate that SGD "acclimates" to the step size and batch size in such a way that each SGD update sometimes increases and sometimes decreases the training loss in expectation, yet an SGD update with a smaller step size or larger batch size would consistently decrease the training loss in expectation.

In extending these findings to SGD, the question arises of how to model "stability" of SGD. This is a highly active area of research. Wu et al. (2018) proposed modeling stability *in expectation*, and gave a sufficient (but not necessary) criterion for the stability of SGD in expectation. Building on this framework, Jastrzebski et al. (2020) argued that if the Hessian is aligned with the second moment matrix of per-example gradients, then SGD is stable so long as a certain expression (involving the sharpness) is below a certain threshold. In the special full-batch case, their criterion reduces to the sharpness being beneath $2/\eta$ — a constraint which we have shown is "tight" throughout training. However, in the general SGD case, there is no evidence that their stability constraint is tight throughout training. Giladi et al. (2020) (whose title inspired our nomenclature) showed that the generalization gap in asynchronous SGD can be mostly ameliorated by setting the step size so as to ensure that stability properties in expectation remain identical to those of a well-tuned implementation of synchronous SGD. Finally, a number of papers have attempted to mathematically model the propensity of SGD to "escape from sharp minima" (Zhu et al., 2019; Xie et al., 2020b).

7 Conclusion

We have empirically demonstrated that the behavior of gradient decent on neural training objectives is both surprisingly consistent across architectures and tasks, and surprisingly different from that envisioned in the conventional wisdom. Our findings raise a number of questions. Why does progressive sharpening occur? At the Edge of Stability, by what mechanism does gradient descent avoid diverging entirely? Since the conventional wisdom for step size selection is wrong, how should the gradient descent step size be set during deep learning? Does the "Edge of Stability" effect generalize in some way to optimization algorithms beyond gradient descent, such as SGD? We hope to inspire future efforts aimed at addressing these questions.

8 ACKNOWLEDGEMENTS

This work was supported in part by DARPA FA875017C0141, the National Science Foundation grants IIS1705121 and IIS1838017, an Amazon Web Services Award, a JP Morgan A.I. Research Faculty Award, a Carnegie Bosch Institute Research Award, a Facebook Faculty Research Award, and a Block Center Grant. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, the National Science Foundation, or any other funding agency.

REFERENCES

Naman Agarwal, Zeyuan Allen-Zhu, Brian Bullins, Elad Hazan, and Tengyu Ma. Finding approximate local minima faster than gradient descent. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 1195–1199, 2017.

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. In *International Conference on Machine Learning*, pp. 242–252. PMLR, 2019.

David G. T. Barrett and Benoit Dherin. Implicit gradient regularization, 2020.

Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *Siam Reviews*, 60(2):223–311, 2018. URL `http://leon.bottou.org/papers/bottou-curtis-nocedal-2018`.

Xiangyi Chen, Sijia Liu, Ruoyu Sun, and Mingyi Hong. On the convergence of a class of adam-type algorithms for non-convex optimization. *arXiv preprint arXiv:1808.02941*, 2018.

Aaron Defazio. Understanding the role of momentum in non-convex optimization: Practical insights from a lyapunov analysis, 2020.

Alexandre Défossez, Léon Bottou, Francis Bach, and Nicolas Usunier. On the convergence of adam and adagrad. *arXiv preprint arXiv:2003.02395*, 2020.

Saber Elaydi. *An introduction to difference equations*. Springer Science & Business Media, 2005.

Niv Giladi, Mor Shpigel Nacson, Elad Hoffer, and Daniel Soudry. At stability's edge: How to adjust hyperparameters to preserve minima selection in asynchronous training of neural networks? In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=Bkeb7lHtvH`.

Gabriel Goh. Why momentum really works. *Distill*, 2(4):e6, 2017.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

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*, pp. 1731–1741, 2017.

Like Hui and Mikhail Belkin. Evaluation of neural architectures trained with square loss vs cross-entropy in classification tasks, 2020.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. pp. 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*, pp. 8571–8580, 2018.

Arthur Jacot, Franck Gabriel, and Clement Hongler. The asymptotic spectrum of the hessian of dnn throughout training. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=SkgscaNYPS`.

Stanislaw Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

Stanislaw Jastrzebski, Zachary Kenton, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amost Storkey. On the relation between the sharpest directions of DNN loss and the SGD step length. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=SkgEaj05t7`.

Stanislaw Jastrzebski, Maciej Szymczak, Stanislav Fort, Devansh Arpit, Jacek Tabor, Kyunghyun Cho*, and Krzysztof Geras*. The break-even point on optimization trajectories of deep neural networks. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=r1g87C4KwB`.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima, 2016.

Y LeCun, L Bottou, GB Orr, and K-R Muller. Efficient backprop. *Lecture notes in computer science*, pp. 9–50, 1998.

Jaehoon Lee, Lechao Xiao, Samuel Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. In *Advances in neural information processing systems*, pp. 8572–8583, 2019.

Aitor Lewkowycz, Yasaman Bahri, Ethan Dyer, Jascha Sohl-Dickstein, and Guy Gur-Ari. The large learning rate phase of deep learning: the catapult mechanism. *arXiv preprint arXiv:2003.02218*, 2020.

Xiaoyu Li and Francesco Orabona. On the convergence of stochastic gradient descent with adaptive stepsizes. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 983–992, 2019.

Yuanzhi Li and Yingyu Liang. Learning overparameterized neural networks via stochastic gradient descent on structured data, 2019.

Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. In *International Conference on Learning Representations*, 2019.

Zhiyuan Li, Kaifeng Lyu, and Sanjeev Arora. Reconciling modern deep learning with traditional optimization analyses: The intrinsic learning rate. *Advances in Neural Information Processing Systems*, 33, 2020.

Chaoyue Liu, Libin Zhu, and Mikhail Belkin. Toward a theory of optimization for over-parameterized systems of non-linear equations: the lessons of deep learning, 2020.

James Martens. *SECOND-ORDER OPTIMIZATION FOR NEURAL NETWORKS*. PhD thesis, 2016.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Yurii Nesterov. Introductory lectures on convex programming volume i: Basic course. 1998.

Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pp. 543–547, 1983.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 1964.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge University Press, USA, 1992. ISBN 0521431085.

Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and H. Brendan McMahan. Adaptive federated optimization, 2020.

Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. Stochastic variance reduction for nonconvex optimization. In *International conference on machine learning*, pp. 314–323, 2016.

Karthik A Sankararaman, Soham De, Zheng Xu, W Ronny Huang, and Tom Goldstein. The impact of neural network overparameterization on gradient confusion and stochastic gradient descent. *arXiv preprint arXiv:1904.06963*, 2019.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pp. 2483–2493, 2018.

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

Daniel Soudry, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. The implicit bias of gradient descent on separable data. *The Journal of Machine Learning Research*, 19 (1):2822–2878, 2018.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, 2013.

Sharan Vaswani, Aaron Mishkin, Issam Laradji, Mark Schmidt, Gauthier Gidel, and Simon Lacoste-Julien. Painless stochastic gradient: Interpolation, line-search, and convergence rates. In *Advances in Neural Information Processing Systems*, pp. 3732–3745, 2019.

Rachel Ward, Xiaoxia Wu, and Leon Bottou. Adagrad stepsizes: sharp convergence over nonconvex landscapes. In *International Conference on Machine Learning*, pp. 6677–6686, 2019.

Lei Wu, Chao Ma, and E Weinan. How sgd selects the global minima in over-parameterized learning: A dynamical stability perspective. In *Advances in Neural Information Processing Systems*, pp. 8279–8288, 2018.

Yuege Xie, Xiaoxia Wu, and Rachel Ward. Linear convergence of adaptive stochastic gradient descent. In *International Conference on Artificial Intelligence and Statistics*, pp. 1475–1485, 2020a.

Zeke Xie, Issei Sato, and Masashi Sugiyama. A diffusion theory for deep learning dynamics: Stochastic gradient descent exponentially favors flat minima, 2020b.

Chen Xing, Devansh Arpit, Christos Tsirigotis, and Yoshua Bengio. A walk with sgd. *arXiv preprint arXiv:1802.08770*, 2018.

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=Syx4wnEtvH`.

Manzil Zaheer, Sashank Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. Adaptive methods for nonconvex optimization. In *Advances in neural information processing systems*, pp. 9793–9803, 2018.

Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BJgnXpVYwS`.

Dongruo Zhou, Yiqi Tang, Ziyan Yang, Yuan Cao, and Quanquan Gu. On the convergence of adaptive gradient methods for nonconvex optimization. *arXiv preprint arXiv:1808.05671*, 2018.

Zhanxing Zhu, Jingfeng Wu, Bing Yu, Lei Wu, and Jinwen Ma. The anisotropic noise in stochastic gradient descent: Its behavior of escaping from sharp minima and regularization effects, 2019.

A STABILITY OF GRADIENT DESCENT ON QUADRATIC FUNCTIONS

This appendix describes the stability properties of gradient descent (and its momentum variants) when optimizing the quadratic objective function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c \tag{2}$$

starting from the initialization $\mathbf{x}_0$.

To review, vanilla gradient descent is defined by the iteration:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\nabla f(\mathbf{x}_t).$$

Meanwhile, gradient descent with Polyak (also called "heavy ball") momentum (Polyak, 1964; Sutskever et al., 2013; Goodfellow et al., 2016) is defined by the iteration:

$$\begin{aligned}\mathbf{v}_{t+1} &= \beta\mathbf{v}_t - \eta\nabla f(\mathbf{x}_t)\\ \mathbf{x}_{t+1} &= \mathbf{x}_t + \mathbf{v}_{t+1}\end{aligned}$$

where $\mathbf{v}_t$ is a "velocity" vector and $0 \leq \beta < 1$ is the momentum coefficient. For $\beta = 0$ the algorithm reduces to vanilla GD.

Finally, Nesterov momentum Sutskever et al. (2013); Goodfellow et al. (2016) is an adaptation of Nesterov's accelerated gradient (Nesterov, 1983) for deep learning defined by the iteration:

$$\begin{aligned}\mathbf{v}_{t+1} &= \beta\mathbf{v}_t - \eta\nabla f(\mathbf{x}_t + \beta\mathbf{v}_t)\\ \mathbf{x}_{t+1} &= \mathbf{x}_t + \mathbf{v}_{t+1}\end{aligned}$$

where $\mathbf{v}_t$ is a "velocity" vector and $0 \leq \beta < 1$ is the momentum coefficient. For $\beta = 0$ the algorithm reduces to vanilla GD.

All three of these algorithms share a special property: on quadratic functions, they act independently along each Hessian eigenvector. That is, if we express the iterates in the Hessian eigenvector basis, then in this basis the coordinates evolve independent from one another under gradient descent.

Proposition 1. *Consider running vanilla gradient descent on the quadratic objective (2) starting from $\mathbf{x}_0$. Let $(\mathbf{q}, a)$ be an eigenvector/eigenvalue pair of $\mathbf{A}$. If $a > 2/\eta$, then the sequence $\{\mathbf{q}^T\mathbf{x}_t\}$ will diverge.*

Proof. The update rule for gradient descent on this quadratic function is:

$$\begin{aligned}\mathbf{x}_{t+1} &= \mathbf{x}_t - \eta(\mathbf{A}\mathbf{x}_t + \mathbf{b})\\ &= (\mathbf{I} - \eta\mathbf{A})\mathbf{x}_t - \eta\mathbf{b}.\end{aligned}$$

Therefore, the quantity $\mathbf{q}^T\mathbf{x}_t$ evolves under gradient descent as:

$$\begin{aligned}\mathbf{q}^T\mathbf{x}_{t+1} &= \mathbf{q}^T(\mathbf{I} - \eta\mathbf{A})\mathbf{x}_t - \eta\mathbf{b}\\ &= (1 - \eta a)\mathbf{q}^T\mathbf{x}_t - \eta\,\mathbf{q}^T\mathbf{b}. \qquad (\mathbf{q}^T\mathbf{A} = a\mathbf{q})\end{aligned}$$

Define $\tilde{x}_t = \mathbf{q}^T\mathbf{x}_t + \frac{1}{a}\mathbf{q}^T\mathbf{b}$, and note that $\{\mathbf{q}^T\mathbf{x}_t\}$ diverges if and only if $\{\tilde{x}_t\}$ diverges.

The quantity $\tilde{x}_t$ evolves under gradient descent according to the simple rule:

$$\tilde{x}_{t+1} = (1 - \eta a)\tilde{x}_t$$

Since $\eta > 0$, if $a > 2/\eta$ then $(1 - \eta a) < -1$, so the sequence $\{\tilde{x}_t\}$ will diverge. □

We now prove analogous results for Nesterov and Polyak momentum.

Theorem 1. *Consider running Nesterov momentum on the quadratic objective (2) starting from any initialization. Let* $(\mathbf{q}, a)$ *be an eigenvector/eigenvalue pair of* $\mathbf{A}$*. If* $a > \frac{1}{\eta}\left(\frac{2+2\beta}{1+2\beta}\right)$*, then the sequence* $\{\mathbf{q}^T\mathbf{x}_t\}$ *will diverge.*

Proof. The update rules for Nesterov momentum on this quadratic function are:

$$\begin{aligned}\mathbf{v}_{t+1} &= \beta(\mathbf{I} - \eta\mathbf{A})\mathbf{v}_t - \eta\mathbf{b} - \eta\mathbf{A}\mathbf{x}_t \\ \mathbf{x}_{t+1} &= \mathbf{x}_t + \mathbf{v}_{t+1}.\end{aligned}$$

Define $\tilde{x}_t = \mathbf{q}^T\mathbf{x}_t + \frac{1}{a}\mathbf{q}^T\mathbf{g}$ and $\tilde{v}_t = \mathbf{q}^T\mathbf{v}_t$. Note that $\mathbf{q}^T\mathbf{x}_t$ diverges iff $\tilde{x}_t$ diverges.

The quantities $\tilde{x}_t$ and $\tilde{v}_t$ evolve under Nesterov momentum as:

$$\begin{aligned}\tilde{v}_{t+1} &= \beta(1 - \eta a)\tilde{v}_t - \eta a\tilde{x}_t \\ \tilde{x}_{t+1} &= \tilde{x}_t + \tilde{v}_{t+1}\end{aligned}$$

By noting that $\tilde{v}_t = \tilde{x}_t - \tilde{x}_{t-1}$, we can rewrite this as a recurrence in $\tilde{x}$:

$$\begin{aligned}\tilde{x}_{t+1} &= \tilde{x}_t + \beta(1 - \eta a)(\tilde{x}_t - \tilde{x}_{t-1}) - \eta\lambda\tilde{x}_t \\ &= (1 - \eta a)(1 + \beta)\tilde{x}_t - \beta(1 - \eta a)\tilde{x}_{t-1}\end{aligned}$$

This is a linear homogenous second-order difference equation. By Theorem 2.37 in Elaydi (2005), since $\eta > 0$ and $\beta < 1$, if $a > \frac{1}{\eta}\left(\frac{2+2\beta}{1+2\beta}\right)$ then this recurrence diverges.

□

The following result previously appeared in Goh (2017).

Theorem 2. *Consider running Polyak momentum on the quadratic objective (2) starting from any initialization. Let* $(\mathbf{q}, a)$ *be an eigenvector/eigenvalue pair of* $\mathbf{A}$*. If* $a > \frac{1}{\eta}(2 + 2\beta)$*, then the sequence* $\{\mathbf{q}^T\mathbf{x}_t\}$ *will diverge.*

Proof. The update rules for Polyak momentum on this quadratic function are:

$$\begin{aligned}\mathbf{v}_{t+1} &= \beta\mathbf{v}_t - \eta\mathbf{A}\mathbf{x}_t - \eta\mathbf{b} \\ \mathbf{x}_{t+1} &= \mathbf{x}_t + \mathbf{v}_{t+1}.\end{aligned}$$

Define $\tilde{x}_t = \mathbf{q}^T\mathbf{x}_t + \frac{1}{\lambda}\mathbf{q}^T\mathbf{b}$ and $\tilde{v}_t = \mathbf{q}^T\mathbf{v}_t$. Note that $\mathbf{q}^T\mathbf{x}_t$ diverges iff $\tilde{x}_t$ diverges.

The quantities $\tilde{x}_t$ and $\tilde{v}_t$ evolve under Polyak momentum as:

$$\begin{aligned}\tilde{v}_{t+1} &= \beta\tilde{v}_t - \eta a\tilde{x}_t \\ \tilde{x}_{t+1} &= \tilde{x}_t + \tilde{v}_{t+1}.\end{aligned}$$

By noting that $\tilde{v}_t = \tilde{x}_t - \tilde{x}_{t-1}$, we can rewrite this as a recurrence in $\tilde{x}$:

$$\begin{aligned}\tilde{x}_{t+1} &= \tilde{x}_t + \beta(\tilde{x}_t - \tilde{x}_{t-1}) - \eta a\tilde{x}_t \\ &= (1 + \beta - \eta a)\tilde{x}_t - \beta\tilde{x}_{t-1}.\end{aligned}$$

This is a linear homogenous second-order difference equation. By Theorem 2.37 in Elaydi (2005), since $\eta > 0$ and $\beta < 1$, if $a > \frac{1}{\eta}(2 + 2\beta)$ then this recurrence diverges. □

B CROSS-ENTROPY LOSS

In this appendix, we explain why the sharpness decreases at the end of training when the cross-entropy loss is used. Before considering the full multiclass case, let us first consider the simpler case of binary classification with the logistic loss.

Binary classification with logistic loss We consider a dataset $\{\mathbf{x}_i, y_i)\}_{i=1}^n \subset \mathbb{R}^d \times \{-1, 1\}$, where the examples are vectors in $\mathbb{R}^d$ and the labels are binary $\{-1, 1\}$. We consider a neural network $h : \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}$ which maps an input $\mathbf{x} \in \mathbb{R}^d$ and a parameter vector $\theta \in \mathbb{R}^p$ to a prediction $h(\mathbf{x}; \theta) \in \mathbb{R}$. Let $\ell : \mathbb{R} \times \{-1, 1\} \to \mathbb{R}$ be the logistic loss function:

$$\begin{aligned} \ell(z; y) &= \log(1 + \exp(-zy)) \\ &= -\log p \quad \text{where} \quad p = \frac{1}{1 + \exp(-zy)}. \end{aligned}$$

The second derivative of this loss function w.r.t z is:

$$\ell''(z; y) = p(1 - p).$$

The full training objective is:

$$f(\theta) = \frac{1}{n} \sum_{i=1}^n f_i(\theta) \qquad \text{where} \quad f_i(\theta) = \ell(f(\mathbf{x}_i; \theta); y_i).$$

The Hessian of this training objective is the average of per-example Hessians:

$$\nabla^2 f(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla^2 f_i(\theta).$$

For any arbitrary loss function ℓ, we have the so-called Gauss-Newton decomposition (Martens, 2016; Bottou et al., 2018) of the per-example Hessian $\nabla^2 f_i(\theta)$:

$$\nabla^2 f_i(\theta) = \ell''(z_i; y_i)\, \nabla_\theta h(\mathbf{x}_i; \theta)\, \nabla_\theta h(\mathbf{x}_i; \theta)^T + \ell'(z_i, y_i;)\, \nabla^2_\theta h(\mathbf{x}_i; \theta) \quad \text{where} \quad z_i = h(\mathbf{x}_i; \theta)$$

where $\nabla_\theta h(\mathbf{x}_i; \theta) \in \mathbb{R}^p$ is the gradient of the network output with respect to the weights, and ℓ' refers to the derivative of ℓ with respect to its *first* argument, the score.

Empirically, the first term in the decomposition (usually called the "Gauss-Newton matrix") tends to dominate the second, which implies the following "Gauss-Newton approximation" to the Hessian:

$$\begin{aligned} \nabla^2 f_i(\theta) &\approx \ell''(z_i; y_i)\, \nabla_\theta h(\mathbf{x}_i; \theta)\, \nabla_\theta h(\mathbf{x}_i; \theta)^T \\ \nabla^2 f(\theta) &\approx \frac{1}{n} \sum_{i=1}^n \ell''(z_i; y_i)\, \nabla_\theta h(\mathbf{x}_i; \theta)\, \nabla_\theta h(\mathbf{x}_i; \theta)^T \end{aligned} \tag{3}$$

In our experience, progressive sharpening affects $\nabla_\theta h(\mathbf{x}_i; \theta)\, \nabla_\theta h(\mathbf{x}_i; \theta)^T$. That is, $\nabla_\theta h(\mathbf{x}_i; \theta)\, \nabla_\theta h(\mathbf{x}_i; \theta)^T$ tends to grow in scale continually during training. For the square loss $\ell(z; y) = \frac{1}{2}(z - y)^2$, the second derivative $\ell''(z; y) = 1$ is the identity, so the $\nabla^2 f_i(\theta)$ grows continually as well. In contrast, for the logistic loss, many of the $\ell''(z_i; y_i)$ decrease at the very end of training. Why is this? In Figure 8, we plot both the logistic loss ℓ, and its second derivative ℓ'', as a function of the quantity yz, which is often called the "margin."

Crucially, observe that both ℓ and ℓ'' are decreasing in yz. Because the loss ℓ is decreasing in yz, once an example i is classified correctly (i.e. $y_i z_i > 0$), the training objective can be optimized further by increasing the margin $y_i z_i$. Because ℓ'' is also decreasing in yz, if the margin $y_i z_i$ increases, the term $\ell''(z_i; y_i)$ drops. Near the end of training, once most examples are classified correctly, gradient descent can easily increase the margins of all these examples by simply scaling up the final layer weight matrix. This causes the $\ell''(z_i; y_i)$ to drop. Therefore, even though progressive sharpening still applies to $\nabla_\theta h(\mathbf{x}_i; \theta)\, \nabla_\theta h(\mathbf{x}_i; \theta)^T$, the decrease in the $\ell''(z_i; y_i)$'s pulls down the leading eigenvalue of the Gauss-Newton matrix in equation 3.

This effect is illustrated in Figure 9. Here, we train a network on the binary classification task of CIFAR-10 airplane vs. automobile, using the logistic loss. In Figure 9(e), we plot the margin $y_i z_i$

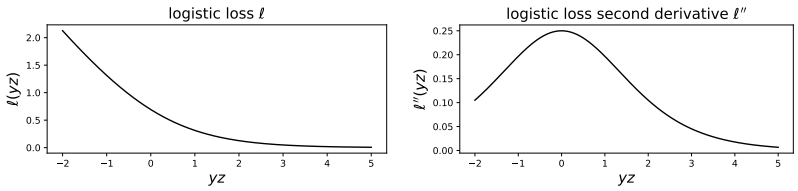

Figure 8: **Left**: logistic loss function ℓ as a function of yz. **Right**: its second derivative ℓ'' as a function of yz.

for 10 examples in the dataset. Notice that at the end of training, these margins all continually rise; this is because gradient descent "games" the objective by increasing the margins of successfully classified examples. When the margin $y_i z_i$ rises, the second derivative $\ell''(z_i; y_i)$ drops. This can be seen from Figure 9(f), where we plot $\ell''(z_i; y_i)$ for these same 10 examples. Now, all the while, the leading eigenvalue of the matrix $\frac{1}{n}\sum_{i=1}^{n} \nabla_\theta h(\mathbf{x}_i;\theta)\nabla_\theta h(\mathbf{x}_i;\theta)^T$ keeps rising, as can be seen from Figure 9(d). However, because the $\ell''s$ are dropping, the leading eigenvalue of the Gauss-Newton matrix $\frac{1}{n}\sum_{i=1}^{n} \ell''(z_i; y_i)\, \nabla_\theta h(\mathbf{x}_i;\theta)\nabla_\theta h(\mathbf{x}_i;\theta)^T$ starts to decrease at the end of training, as can be seen from the green line in Figure 9(c). Finally, since the leading eigenvalue of the Gauss-Newton matrix is an excellent approximation to the leading eigenvalue of the Hessian (i.e. the sharpness), the sharpness also drops at the end of training, as can be seen from the orange line in Figure 9(c).

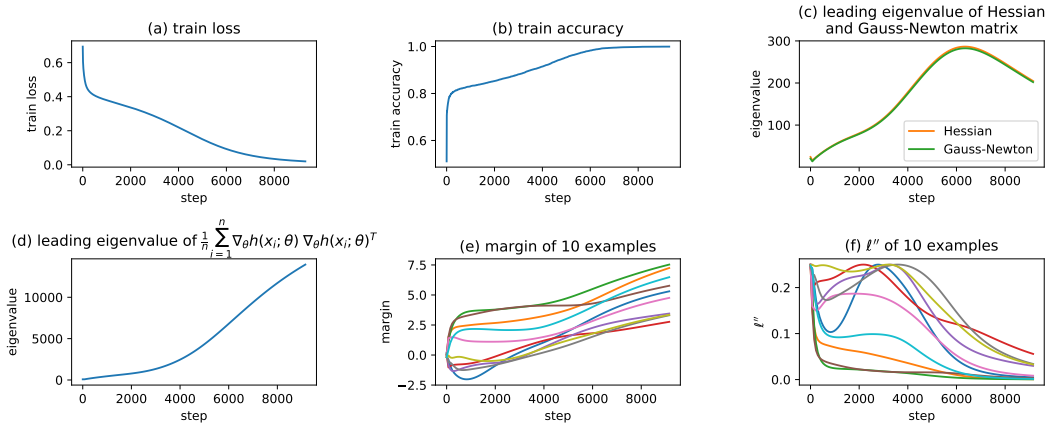

Figure 9: We train a network using the logistic loss on the binary classification problem of CIFAR-10 airplane vs. automobile. **(a)** The train loss. **(b)** The train accuracy. **(c)** The leading eigenvalue of the Hessian and of the Gauss-Newton matrix; observe that the latter is a great approximation to the former. **(d)** The leading eigenvalue of the matrix $\frac{1}{n}\sum_{i=1}^{n} \nabla_\theta h(\mathbf{x}_i;\theta)\nabla_\theta h(\mathbf{x}_i;\theta)^T$, which is the Gauss-Newton matrix without the ℓ'' terms; observe that this value constantly rises — it does not dip at the end of training. **(e)** The margin $y_i z_i$ of 10 examples; observe that all the margins rise at the end of training. **(f)** the value $\ell''(z_i; y_i)$ for 10 examples; observe that all of these curves decline at the end of training.

Multiclass classification with cross-entropy loss We consider a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n \subset \mathbb{R}^d \times \{1, \ldots, k\}$, where the examples are vectors in $\mathbb{R}^d$ and the labels are in $\{1, \ldots, k\}$. We consider a neural network $h : \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}^k$ which maps an input $\mathbf{x} \in \mathbb{R}^d$ and a parameter vector $\theta \in \mathbb{R}^p$ to a prediction $h(x;\theta) \in \mathbb{R}^k$. Let $\ell : \mathbb{R}^k \times \{1, \ldots, k\} \to \mathbb{R}$ be the cross-entropy loss function:

$$\ell(\mathbf{z}; y) = -\log \frac{\exp(z_y)}{\sum_{j=1}^{k} \exp(z_j)}$$

$$= -\log p_y \quad \text{where} \quad \mathbf{p} = \frac{\exp(\mathbf{z})}{\sum_j \exp(z_j)}.$$

The Hessian $\nabla^2 \ell(\mathbf{z}; y) \in \mathbb{R}^{k \times k}$ of this loss function w.r.t the class scores $\mathbf{z}$ is:

$$\nabla^2 \ell(\mathbf{z}; y) = \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^T.$$

Now, for any loss function $\ell : \mathbb{R}^k \times \{1, \ldots, k\} \to \mathbb{R}$, we have the Gauss-Newton decomposition:

$$\nabla^2 f_i(\theta) = \mathbf{J}_i^T \left[\nabla^2_{\mathbf{z}_i} \ell(\mathbf{z}_i; y_i)\right] \mathbf{J}_i + \sum_{j=1}^{k} \ell'(z_{ij}) \nabla^2_\theta h_j(\mathbf{x}_i; \theta)$$

where $\mathbf{z}_i = h_i(\mathbf{x}_i; \theta) \in \mathbb{R}^k$ are the logits for example i, $\mathbf{J}_i \in \mathbb{R}^{k \times p}$ is the network output-to-weights Jacobian for example i, $\nabla^2 \ell(\mathbf{z}_i; y_i) \in \mathbb{R}^{k \times k}$ is the Hessian of ℓ w.r.t its input $\mathbf{z}_i$, and $\nabla^2_\theta h_j(\mathbf{x}_i) \in \mathbb{R}^{p \times p}$ is the Hessian matrix of the j-th output of the network h on the i-th example.

Dropping the second term yields the Gauss-Newton approximation:

$$\nabla^2 f_i(\theta) \approx \mathbf{J}_i^T \left[\nabla^2_{\mathbf{z}_i} \ell(\mathbf{z}_i; y_i)\right] \mathbf{J}_i.$$

As in the binary classification case discussed above: at the end of training, for many examples i, the y_i entry of $\mathbf{p}_i$ will tend toward 1 and the other entries of $\mathbf{p}_i$ will tend to 0. Once this occurs, the matrix $\text{diag}(\mathbf{p}_i) - \mathbf{p}_i \mathbf{p}_i^T$ will broadly decrease in scale: the diagonal entries of this matrix are of the form $p(1-p)$, which goes to zero as $p \to 0$ or $p \to 1$; and the off-diagonal entries are of the form $-pq$, which also goes to zero if $p \to 0, q \to 1$ or $p \to 0, q \to 0$ or $p \to 1, q \to 0$.

This effect is illustrated in Figure 10. Here, we train a network on CIFAR-10 using the cross-entropy loss. In Figure 10(e), for ten examples i in the dataset (with output scores $\mathbf{z}_i = h(\mathbf{x}_i) \in \mathbb{R}^k$), we plot the margin $\mathbf{z}_i[y_i] - \max_{j \neq y_i} \mathbf{z}_i[j]$, which is the difference between the score of the correct class y_i and the score of the next-highest class. Observe that for all of these examples, this margin rises at the end of training. In Figure 10(f), for those same ten examples, we plot the quantity $\mathbf{p}_i[y_i](1 - \mathbf{p}_i[y_i])$.Observe that for all of these examples, this quantity continually decreases at the end of training. Now, all the while, the leading eigenvalue of the matrix $\frac{1}{n}\sum_{i=1}^n \mathbf{J}_i^T \mathbf{J}_i$ keeps rising, as can be seen from Figure 10(d). However, because $\nabla^2 \ell(\mathbf{z}_i; y_i)$ is decreasing, the leading eigenvalue of the Gauss-Newton matrix $\frac{1}{n}\sum_{i=1}^n \mathbf{J}_i^T \nabla^2 \ell(\mathbf{z}_i; y_i) \mathbf{J}_i$ starts to decrease at the end of training, as can be seen from the green line in Figure 10(c). Finally, since the leading eigenvalue of the Gauss-Newton matrix is an excellent approximation to the leading eigenvalue of the Hessian (i.e. the sharpness), the sharpness also drops at the end of training, as can be seen from the orange line in Figure 10(c).

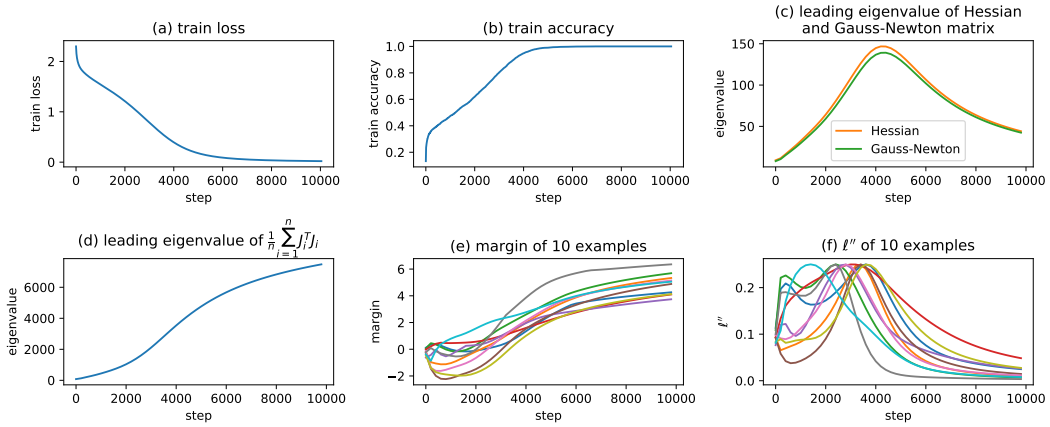

Figure 10: We train a network using the cross-entropy loss on CIFAR-10. **(a)** The train loss. **(b)** The train accuracy. **(c)** The leading eigenvalue of the Hessian and of the Gauss-Newton matrix; observe that the latter is a great approximation to the former. **(d)** The leading eigenvalue of the matrix $\frac{1}{n}\sum_{i=1}^{n} \mathbf{J}_i^T \mathbf{J}_i$, which is the Gauss-Newton matrix except the $\nabla^2 \ell$ terms; observe that this value constantly rises — it does not dip at the end of training. **(e)** The margin $\mathbf{z}_i[y_i] - \max_{j \neq y_i} \mathbf{z}_i[j]$ of 10 examples; observe that all these margins rise at the end of training. **(f)** the value $\mathbf{p}_i[y_i](1 - \mathbf{p}_i[y_i])$ for 10 examples; observe that all of these curves decline at the end of training.

C PROGRESSIVE SHARPENING IN WIDE NETWORKS

When networks parameterized in a certain way (the "NTK parameterization") are made infinitely wide and trained using gradient flow, the Hessian moves a vanishingly small amount during training, implying that no progressive sharpening occurs (Jacot et al., 2018; Lee et al., 2019; Jacot et al., 2020; Li & Liang, 2019). Therefore, a natural hypothesis is that progressive sharpening might attenuate as network width increases. In this appendix, we run experiments which support this hypothesis.

We consider fully-connected architectures with two hidden layers and tanh activations, with widths $\{32, 64, 128, 256, 512, 1024\}$. We train on a size-5,000 subset of CIFAR-10 using the cross-entropy loss. We train using gradient flow, i.e. we numerically integrate the gradient flow ODE using Runge-Kutta (details in §G.5). We consider both NTK parameterization and standard parameterization (Lee et al., 2019).

In Figure 11, for each width, we train NTK-parameterized networks from five different random initializations, and plot the evolution of the sharpness during gradient flow. Observe that the maximum sharpness along the gradient flow trajectory is larger for narrow networks, and smaller for wide networks. (As elsewhere in this paper, note that the sharpness drops at the end of training due to the cross-entropy loss.) In Figure 12, we plot summary statistics from these training runs. Namely, define $\lambda_{\max}$ as the maximum sharpness over the gradient flow trajectory, and define λ_0 as the initial sharpness. In Figure 12(a), for each width we plot the mean and standard deviation of the maximum sharpness $\lambda_{\max}$ over the five different random initializations. Observe that $\lambda_{\max}$ becomes smaller, on average, as the width is made larger. In Figure 12(b), for each width we plot the mean and standard deviation of the *maximum sharpness gain* $\lambda_{\max}/\lambda_0$ over the five different random initializations. Observe that the maximum sharpness gain $\lambda_{\max}/\lambda_0$ also becomes smaller as the width is made larger. NTK theory suggests that $\lambda_{\max}/\lambda_0$ should deterministically tend to 1 as the width $\to \infty$, and Figure 12(b) is consistent with this prediction.

In Figures 13 and 14, we conduct similar experiments, but with standard parameterization rather than NTK parameterization. Similar to NTK parameterization, we observe in Figure 13 that the sharpness rises more for narrow networks than for wide networks.

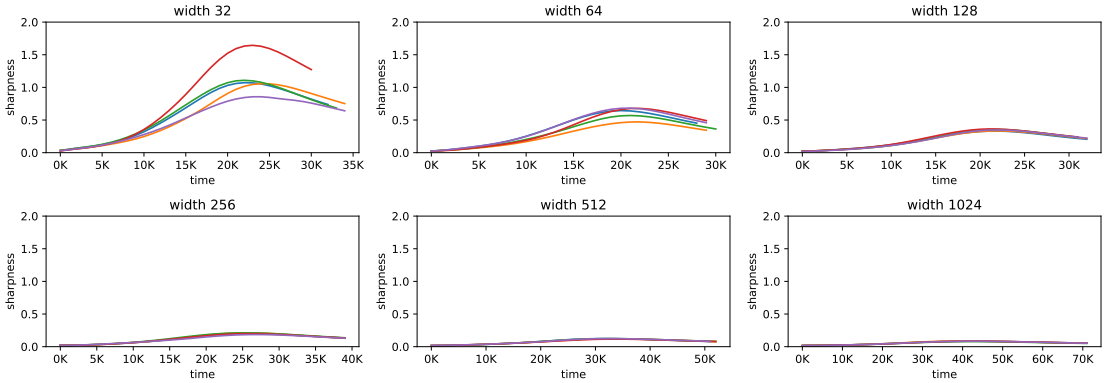

Figure 11: **NTK parameterization: evolution of the sharpness**. We use gradient flow to train NTK-parameterized networks, and we track the evolution of the sharpness during training. For each width, we train from five different random initializations (different colors). Observe that the sharpness rises more when training narrow networks than when training wide networks.

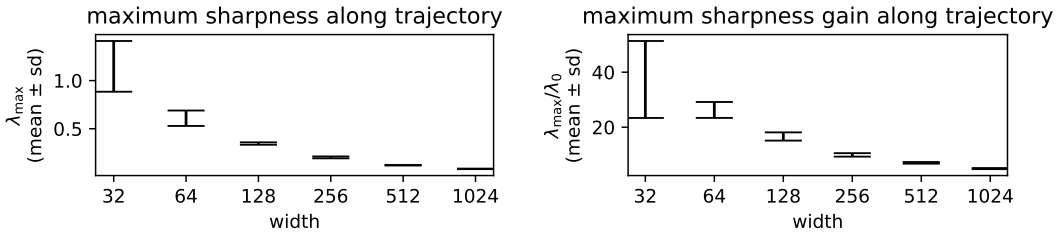

Figure 12: **NTK parameterization: summary statistics. Left**: For each network width, we plot the mean and standard deviation (over the five different random initializations) of the maximum sharpness $\lambda_{\max}$ along the gradient flow trajectory. Observe that $\lambda_{\max}$ decreases in expectation as the width increases. **Right**: For each network width, we plot the mean and standard deviation (over the five different random initializations) of the maximum sharpness gain $\lambda_{\max}/\lambda_0$ along the gradient flow trajectory. Observe that $\lambda_{\max}/\lambda_0$ decreases in expectation as the width increases. Indeed, NTK theory predicts that $\lambda_{\max}/\lambda_0$ should deterministically tend to 1 as width $\to \infty$ and this plot is consistent with that prediction.

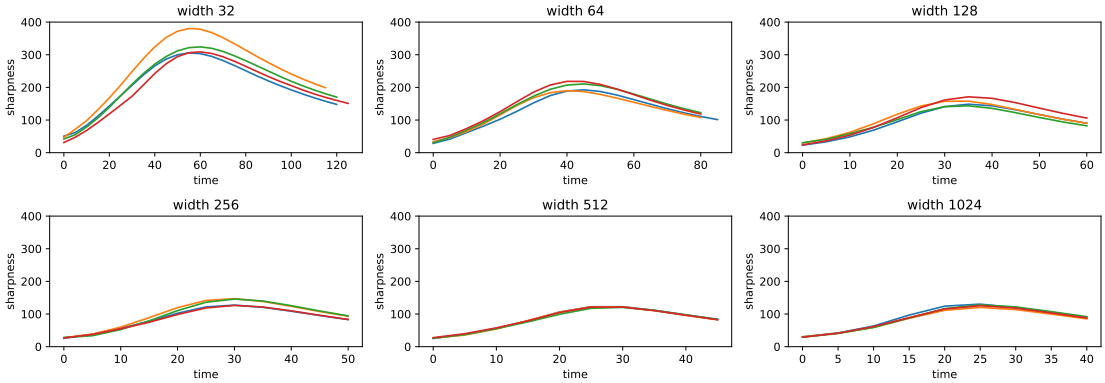

Figure 13: **Standard parameterization: evolution of the sharpness**. We use gradient flow to train standard-parameterized networks, and we track the evolution of the sharpness during training. For each width, we train from five different random initializations (different colors). Observe that the sharpness rises more when training narrow networks than when training wide networks.

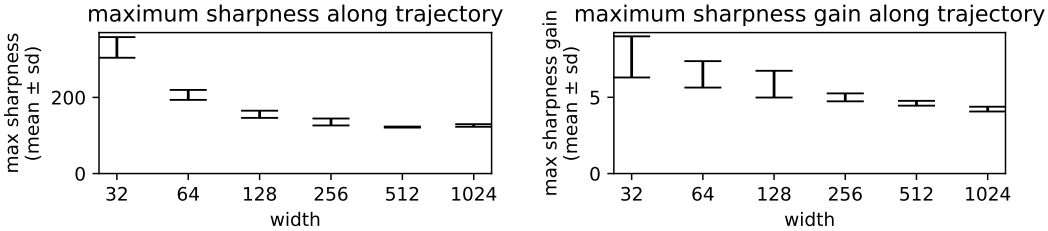

Figure 14: **Standard parameterization: summary statistics. Left**: For each network width, we plot the mean and standard deviation (over the five different random initializations) of the maximum sharpness $\lambda_{\max}$ along the gradient flow trajectory. Observe that $\lambda_{\max}$ tends to decrease in expectation as the width increases, though it is not clear whether this pattern still holds when moving from width 512 to width 1024 — more samples are needed. **Right**: For each network width, we plot the mean and standard deviation (over the five different random initializations) of the maximum sharpness gain $\lambda_{\max}/\lambda_0$ along the gradient flow trajectory. Observe that $\lambda_{\max}/\lambda_0$ decreases in expectation as the width increases. It is not clear whether or not $\lambda_{\max}/\lambda_0$ is deterministically tending to 1, but that does seem possible.

D SPEED OF DIVERGENCE ON QUADRATIC TAYLOR APPROXIMATION

If the sharpness at some iterate is strictly greater than $2/\eta$, then gradient descent with step size η is guaranteed to diverge if run on the quadratic Taylor approximation around that iterate. However, the speed of this divergence could conceivably be slow — in particular, the train loss might continue to decrease for many iterations before it starts to increase. In this appendix we empirically demonstrate, to the contrary, that at the Edge of Stability, gradient descent diverges *quickly* if, at some iterate, we start running gradient descent on the quadratic Taylor approximation around that iterate.

We consider the fully-connected tanh network from section 3, trained on a 5,000-sized subsample of CIFAR-10 using both cross-entropy loss and MSE loss. At some timestep t_0 during training, we suddenly switch from running gradient descent on the real neural training objective, to running gradient descent on the quadratic Taylor approximation around the iterate at step t_0. We do this for three timesteps before gradient descent has entered the Edge of Stability, and three afterwards. Figure 16 shows the results for cross-entropy loss, and Figure 17 shows the (similar) results for MSE loss. Before entering the Edge of Stability (**top row**), gradient descent on the quadratic Taylor approximation behaves similar to gradient descent on the real neural training objective — that is, the orange line almost overlaps the blue line. Yet after entering the Edge of Stability (**bottom row**), gradient descent on the quadratic Taylor approximation quickly diverges, whereas gradient descent on the real neural training objective makes consistent (if choppy) progress.

In short, when gradient descent is *not* at the Edge of Stability, the quadratic Taylor approximation serves as a good model for the local progress of gradient descent. But when gradient descent *is* at the Edge of Stability, the quadratic Taylor approximation is an extremely poor model for the local progress of gradient descent. It is conceivable that there exists some simple modification to the quadratic Taylor model which would fix this issue (e.g. perhaps if one ignores a certain direction, the quadratic Taylor model is accurate). Nevertheless, unless/until such a fix is discovered, it is unclear why quadratic Taylor approximations should yield any insight into the local behavior of gradient descent.

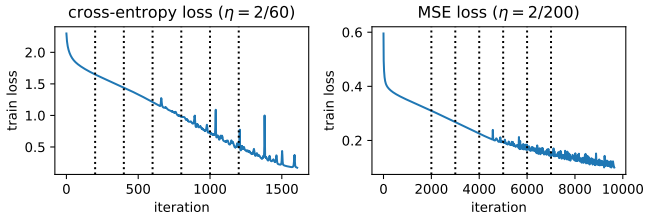

Figure 15: We train a neural network using cross-entropy loss (left) and MSE loss (right). In Figure 16 and 17, we show what happens when, at the iterations marked above by vertical dotted lines, we switch from gradient descent on the real neural training objective to gradient descent on the quadratic Taylor approximation.

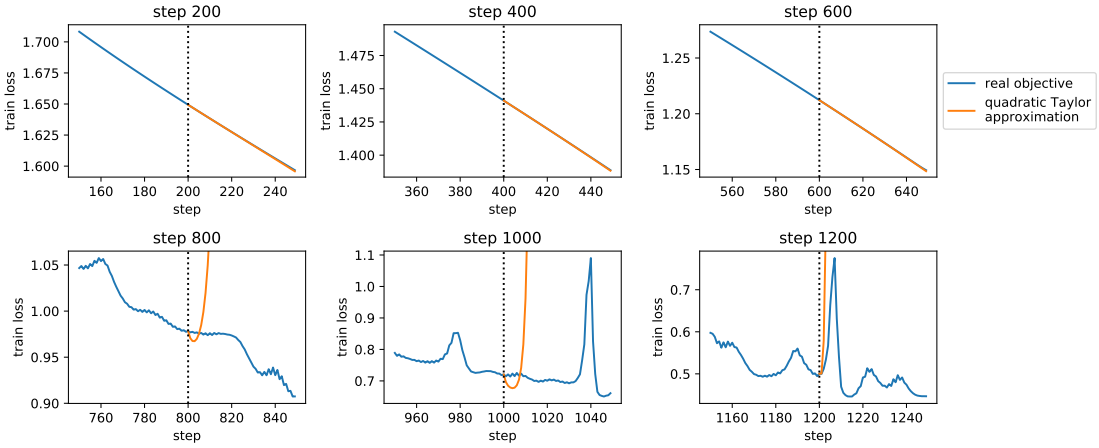

Figure 16: **Cross-entropy loss** ($\eta = 2/60$). At six different iterations during the training of the network from Figure 15 (marked by the vertical dotted black lines), we switch from running gradient descent on the real neural training objective (for which the train loss is plotted in blue) to running gradient descent on the quadratic Taylor approximation around the current iterate (for which the train loss is plotted in orange). **Top row** are timesteps $\{200, 400, 600\}$ before gradient descent has entered the Edge of Stability; observe that the orange line (Taylor approximation) closely tracks the blue line (real objective). **Bottom row** are timesteps $\{800, 1000, 1200\}$ during the Edge of Stability; observe that the orange line quickly diverges, whereas the blue line does not.

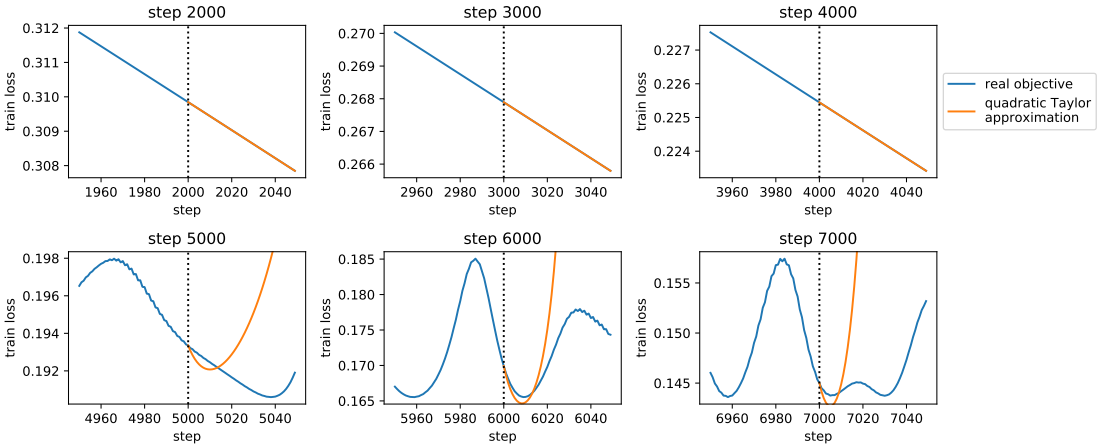

Figure 17: **MSE loss** ($\eta = 2/200$). At six different iterations during the training of the network from Figure 15 (marked by the vertical dotted black lines), we switch from running gradient descent on the real neural training objective (for which the train loss is plotted in blue) to running gradient descent on the quadratic Taylor approximation around the current iterate (for which the train loss is plotted in orange). **Top row** are timesteps (2000, 3000, 4000) before gradient descent has entered the Edge of Stability; observe that the orange line (Taylor approximation) closely tracks the blue line (real objective). **Bottom row** are timesteps (5000, 6000, 7000) during the Edge of Stability; observe that the orange line quickly diverges, whereas the blue line does not.

E "OPTIMAL" STEP SIZE SELECTION

One heuristic for setting the step size of gradient descent is to set the step size at iteration t to $\eta_t = 1/\lambda_t$, where λ_t is the sharpness at iteration t. While this heuristic is computationally impractical due to the time required to compute the sharpness at each iteration, it is often regarded as an ideal, for instance in LeCun et al. (1998) (Eq. 39) and Schaul et al. (2013) (Eq 8). The motivation for this heuristic is: if all that is known about the training objective is that the local sharpness is λ, then a step size of $1/\lambda$ maximizes the guaranteed decrease in the training objective that would result from taking a step.

In this appendix, we demonstrate (on a single numerical example) that the dynamic step size $\eta_t = 1/\lambda_t$ is considerably outperformed by the baseline approach of using the fixed step size $\eta_t = 1/\lambda_0$, where λ_0 is the sharpness at initialization. We consider a fully-connected network with two hidden layers and tanh activations, trained on a subset of 5,000 examples from CIFAR-10. In Figure 18, we train using both the dynamic $\eta = 1/\lambda_t$ step size heuristic as well as the baseline fixed step size of $\eta = 1/\lambda_0$.[3] Observe that the $\eta = 1/\lambda_0$ baseline soundly outperforms the $1/\lambda_t$ heuristic. Intuitively, because of progressive sharpening, the $\eta_t = 1/\lambda_t$ heuristic anneals the step size, and therefore ends up using step sizes that are suboptimally small. In contrast, while the $\eta_t = 1/\lambda_0$ baseline quickly becomes unstable as soon as progressive sharpening raises the sharpness to $2\lambda_0$, this instability is apparently a worthwhile "price to pay" in return for the benefit accrued by taking larger steps.

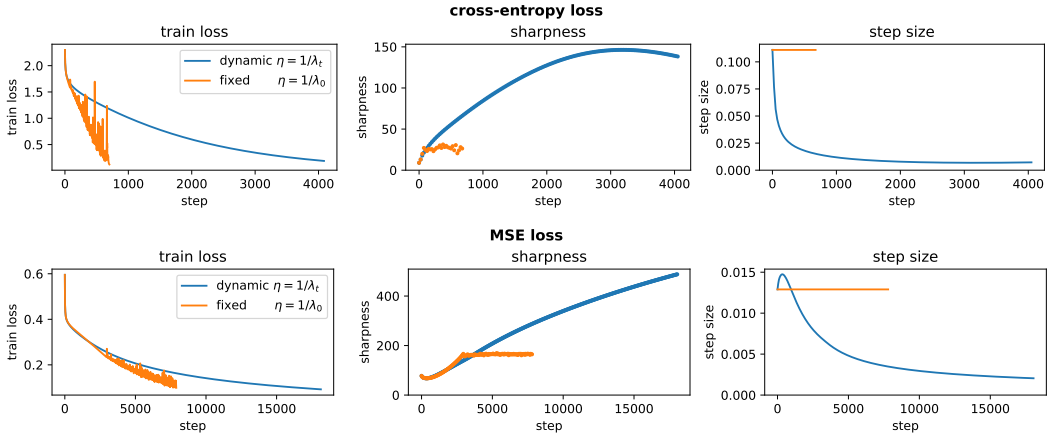

Figure 18: **The dynamic step size $\eta_t = 1/\lambda_t$ underperforms the fixed step size of $\eta_t = 1/\lambda_0$.** In blue, we run gradient descent with a dynamic step size $\eta_t = 1/\lambda_t$, where λ_t is the sharpness at iteration t (we compute the sharpness once every 50 steps and cache it). In orange, we run gradient descent with a fixed step size $\eta_t = 1/\lambda_0$. Even though the dynamic step size heuristic (blue) is supposed to be optimal (LeCun et al., 1998; Schaul et al., 2013), it trains considerably slower than the fixed step size baseline (orange). This is because it anneals the step size during training (see right column).

[3]For dynamic $\eta_t = 1/\lambda_t$ step size, we compute the sharpness once every 50 steps.

F SGD ACCLIMATES TO THE HYPERPARAMETERS

In this appendix, we conduct an experiment which suggests that some version of "Edge of Stability" may hold for SGD.

One way to interpret our main findings is that gradient descent "acclimates" to the step size in such a way that each training update sometimes increases, and sometimes decreases, the training loss, yet an update with a smaller step size would always decrease the training loss. We now demonstrate that this interpretation may generalize to SGD. In particular, we will demonstrate that SGD seems to "acclimate" to the step size and batch size in such a way that an actual update sometimes increases and sometimes decreases the loss in expectation, yet a update with a larger step size or smaller batch size would almost always *increase* the loss in expectation, and a step with a smaller step size or a larger batch size would almost always *decrease* the loss in expectation.

In Figure 19, we train the tanh network from Section 3 with MSE loss, using SGD with step size 0.01 and batch size 32. We periodically compute the training loss (over the full dataset) and plot these on the left pane of Figure 19. Observe that the training loss does not decrease monotonically, but of course this is not surprising — SGD is a random algorithm. However, what may be more surprising is that SGD is not even decreasing the training loss *in expectation*. On the right pane of Figure 19, every 500 steps during training, we use the Monte Carlo method to approximately compute the expected change in training loss that would result from an SGD step (the expectation here is over the randomness involved in selecting the minibatch). Observe that at many points during training, an SGD step would decrease the loss (as desired) in expectation, but at other points, and SGD step would increase the loss in expectation.

In Figure 20(a), while training that network, we compute the expected change in training loss that would result from taking an SGD step with the same step size used during training (i.e. 0.01), but *half the batch size* used during training (i.e. 16). We observe that an SGD step with half the batch size would consistently cause an *increase* in the training loss in expectation. In Figure 20(b) we repeat this experiment, but with *twice* the batch size used during training (i.e. 64). Notice that an SGD step with twice the batch size would consistently cause a *decrease* in the training loss in expectation. In Figure 20(c) and (d) repeat this experiment with the step size; we observe that an SGD step with a larger step size (0.02) would consistently increase the training loss in expectation, while an SGD step with a smaller step size (0.005) would consistently decrease the training loss in expectation.

In Figure 21, as a "control" experiment, we both train *and* measure the expected loss change under the following four hyperparameter settings: (step size 0.01, batch size 16), (step size 0.01, batch size 64), (step size 0.02, batch size 32), and (step size 0.005, batch size 32). In each case, we observe that, after a brief period at the beginning of training, each SGD update sometime increases and sometimes decreases the training loss in expectation.

Therefore, at least for this single network, we can conclude that no matter the hyperparameters, SGD quickly navigates to, and then lingers in, regions of the loss landscape in which an SGD update *with those hyperparameters* sometimes increases, and sometimes decreases the training loss in expectation, yet an SGD update with a smaller step size or larger batch size would consistently decrease the loss in expectation, and an SGD update with a larger step size or smaller batch size would consistently increase the loss in expectation.

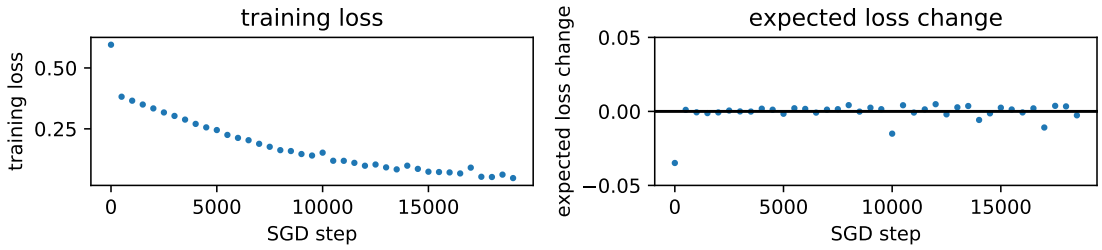

Figure 19: **SGD does not consistently decrease the training loss in expectation.** We train the tanh network from section 3 with MSE loss, using SGD with step size 0.01 and batch size 32. Periodically during training, we compute (left) the full-batch training loss, and (right) the expected change in the full-batch training loss that would result from taking an SGD step (where the expectation is over the randomness in sampling the minibatch). Strikingly, note that after the very beginning of training, the expected loss change is sometimes negative (as desired) but oftentimes positive. See Figure 20.

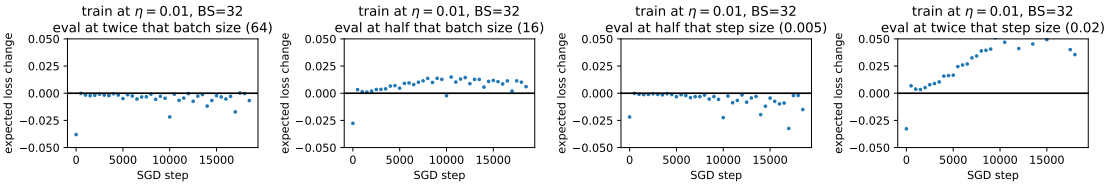

Figure 20: **An SGD step with a smaller learning rate or a larger batch size than the ones used during training *would* consistently decrease the loss in expectation.** At regular intervals during the training run depicted in Figure 19 (with $\eta = 0.01$ and batch size 32), we measure the expected change in the full-batch training loss that would result from an SGD step with a different step size or batch size. Observe that taking an SGD step with a smaller step size or a larger batch size would consistently have decreased the loss in expectation, while taking an SGD step with a larger step size or a smaller batch size would have consistently increased the loss in expectation.

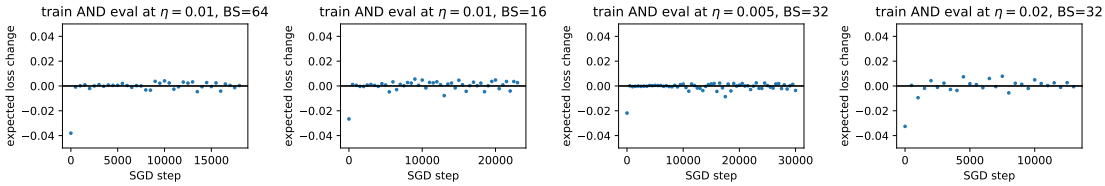

Figure 21: **Control experiment.** Above, in Figure 20(a), as we trained a network with step size 0.01 and batch size 32, we evaluated the expected change in training loss that would result from taking an SGD step with step size 0.01 and batch size 64. Here, in Figure 21(a), as a "control experiment," we train the network with step size 0.01 and batch size 64, and evaluate the expected change in training loss that would result from taking an SGD step with the same step size and batch size. We observe that an SGD step using the same step size and batch size that are used during training would sometimes increase and sometimes decrease the training loss in expectation. The other three panes are analogous.

G EXPERIMENTAL DETAILS

G.1 VARYING ARCHITECTURES ON 5K SUBSET OF CIFAR-10

Dataset. The dataset consists of the first 5,000 examples from CIFAR-10. To preprocess the dataset, we subtracted the mean from each channel, and then divided each channel by the standard deviation (where both the mean and stddev were computed over the *full* CIFAR-10 dataset, not the 5k subset).

Architectures. We experimented with two architecture families: fully-connected and convolutional. For each of these two families, we experimented with several different activation functions, and for convolutional networks we experimented with both max pooling and average pooling.

The PyTorch code for e.g. the fully-connected ReLU network is as follows:

```
nn.Sequential(
    nn.Flatten(),
    nn.Linear(3072, 200, bias=True),
    nn.ReLU(),
    nn.Linear(200, 200, bias=True)
    nn.ReLU(),
    nn.Linear(200, 10, bias=True)
)
```

Networks with other activation functions would have `nn.ReLU()` replaced by `nn.ELU()`, `nn.Tanh()`, `nn.Softplus()`, or `nn.Hardtanh()`.

The PyTorch code for e.g. the convolutional ReLU network with max-pooling is as follows:

```
nn.Sequential(
    nn.Conv2d(3, 32, bias=True, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Conv2d(32, 32, bias=True, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Flatten(),
    nn.Linear(2048, 10, bias=True)
)
```

Networks with other activation functions would have `nn.ReLU()` replaced by `nn.ELU()` or `nn.Tanh()`, and networks with average pooling would have `nn.MaxPool2d(2)` replaced by `nn.AvgPool2d(2)`.

For all of these networks, we use the default PyTorch initialization. That is, both fully-connected layers and convolutional layers have the entries of their weight matrix and bias vector sampled i.i.d from Uniform$(-\frac{1}{\sqrt{\text{fan in}}}, \frac{1}{\sqrt{\text{fan in}}})$.

Loss functions. For a k-class classification problem, if the network outputs are $\mathbf{z} \in \mathbb{R}^k$ and the correct class is $i \in [k]$, then the mean squared error (MSE) loss is defined as $\frac{1}{2}\left[(z[i]-1)^2 + \sum_{j \neq i} z[j]^2\right]$. That is, we encode the correct class with a "1" and the other classes with "0." The cross-entropy loss is defined as $-\log\left(\frac{\exp(z[i])}{\sum_{j \neq i} z[j]}\right)$.

G.2 STANDARD ARCHITECTURES ON CIFAR-10

To preprocess the CIFAR-10 dataset, we subtracted the mean from each channel, and then divided each channel by the standard deviation.

Since training with full-batch gradient descent is slow, we opted to experiment on relatively shallow networks. The VGG networks (both with and without BN) are VGG-11's, from the implementation here: `https://github.com/chengyangfu/pytorch-vgg-cifar10/`

blob/master/vgg.py, with the dropout layers removed. The ResNet is the (non-fixup) ResNet-32 implemented here: https://github.com/hongyi-zhang/Fixup.

For the two networks with batch normalization, running gradient descent with full-dataset batch normalization would not have been feasible under our GPU memory constraints. Therefore, we instead used ghost batch normalization (Hoffer et al., 2017) with 50 ghost batches of size 1,000. This means that we divided the 50,000 examples in CIFAR-10 into 50 fixed groups of size 1,000 each, and defined the overall objective function to be the average of 50 fixed batch-wise objectives. To correctly compute the overall gradient of this training objective, we can just run backprop 50 times (once on each group) and average the resulting gradients.

To compute the sharpness (the leading Hessian eigenvalue) over the full CIFAR-10 dataset would have been computationally expensive. Therefore, in an approximation, we instead computed the sharpness over just the first 5,000 examples in the dataset (or, for the BN networks, over the first 5 batches out of 50).

G.3 Batch normalization experiments

We used the CNN architecture from §H (described above in §G.1), but with a `BatchNorm2d()` layer inserted after each activation layer.

Since our GPUs did not have enough memory to run batch normalization with the full dataset of size 5,000, we used ghost batch normalization with five ghost batches of size 1,000 (see G.2 for details).

G.4 Transformer on WikiText-2

We used both the Transformer architecture and the preprocessing setup from the official PyTorch (Paszke et al., 2019) word-level language modeling tutorial: https://github.com/pytorch/examples/tree/master/word_language_model. We used the settings `ninp=200, nhead=2, nhid=200, nlayers=2, dropout=0`.

We set `bptt = 35`, which means that we divided the corpus into chunks of 35 token, and trained the network (using negative log likelihood loss) to predict each token from the preceding tokens in the same chunk.

Since computing the sharpness over the full dataset would not have been computationally practical, we computed the sharpness over a subset comprising 2500 training examples.

G.5 Runge-Kutta

We used the "RK4" fourth-order Runge-Kutta algorithm (Press et al., 1992) to numerically integrate the gradient flow ODE. The Runge-Kutta algorithm requires a step size. Rather than use a sophisticated algorithm for adaptive step size control, we decided to take advantage of the fact that we were already periodically computing the sharpness: at each step, we set the step size to α/λ, where α is a tunable parameter and λ is the most recent value for the sharpness. We set $\alpha = 1$ or $\alpha = 0.5$.

G.6 Random projections

In order to ascertain whether gradient descent at step size η initially followed the gradient flow trajectory (and, if so, for how long), we monitored the ℓ_2 distance in weight space between the gradient flow solution at time t and the gradient descent iterate at step t/η. One way to do would be as follows: (a) when running gradient flow, save the weights after every Δt units of time, for some parameter Δt; (b) when running gradient descent, save the weights at each $(\frac{\Delta t}{\eta})$-th step; (c) plot the difference between these two sequences. (Note that this approach requires Δt to be divisible by η.)

We essentially used this approach, but with one modification: regularly saving the entire network weight vector would have consumed a large amount of disk space, so we instead saved low-dimensional random projections of the network weights. To be clear, let d be the number of network weights, and let k be the number of random projections (a tunable parameter chosen such that $k \ll d$). Then we first generated a matrix $\mathbf{M} \in \mathbb{R}^{k \times d}$ by sampling each entry i.i.d from the standard normal distribution. During training, rather than periodically save the whole weight vector

(a d-dimensional vector), we premultiplied this vector by the matrix $\mathbf{M}$ to obtain a k-dimensional vector, and we periodically saved these vectors instead. Then we plotted the ℓ_2 distance between the low-dimensional vectors from gradient flow, and the low-dimensional vectors from gradient descent.

H EXPERIMENTS: VARY ARCHITECTURES

In this Appendix, we fix the task as that of fitting a 5,000-sized subset of CIFAR-10, and we verify our main findings across a broad range of architectures.

Procedure We consider fully-connected networks and convolutional networks, the latter with both max-pooling and average pooling. For all of these, we consider tanh, ReLU, and ELU activations, and for fully-connected networks we moreover consider softplus and hardtanh activations. We train each network with both cross-entropy and MSE loss. See §G.1 for full experimental details.

In each case, we first use the Runge-Kutta method to numerically integrate the gradient flow ODE (see §G.5 for details). For architectures that give rise to continuously differentiable training objectives, the gradient flow ODE is guaranteed to have a unique solution (which we call the gradient flow trajectory), and Runge-Kutta will return a numerical approximation to this solution. On the other hand, for architectures with ReLU, hardtanh, or max-pooling, the training objective is not continuously differentiable, so the gradient flow ODE does not necessarily have a unique solution, and there are no guarantees a priori on what Runge-Kutta will return (more on this below under the "findings" heading). Still, in both cases, since our implementation of Runge-Kutta automatically adjusts the step size based on the local sharpness in order to remain stable, the Runge-Kutta trajectory can be roughly viewed as "what gradient descent would do if instability was not an issue."

We then run gradient descent at a range of step sizes. These step sizes η were chosen by hand so that the quantity $2/\eta$ would be spaced uniformly between λ_0 (the sharpness at initialization) and $\lambda_{\max}$ (the maximum sharpness along the Runge-Kutta trajectory).

Results During Runge-Kutta, we observe that the sharpness tends to continually increase during training (progressive sharpening), with the exception that when cross-entropy loss is used, the sharpness decreases at the very end of training, as explained in Appendix B.

During gradient descent with step size η, we observe that once the sharpness reaches $2/\eta$, it ceases to increase much further, and instead hovers right at, or just above, the value $2/\eta$. For reasons unknown, it tends to be true that for MSE loss, the sharpness hovers just a tiny bit above the value $2/\eta$, while for cross-entropy loss the gap between the sharpness and the value $2/\eta$ is a bit larger.

For each step size, we monitor the distance between the gradient descent trajectory and the Runge-Kutta trajectory — that is, we monitor the distance between the Runge-Kutta iterate at time t and the gradient descent iterate at step t/η (see §G.6 for details). Empirically, for architectures that give rise to continuously differentiable training objectives, we observe that this distance is nearly zero before the sharpness hits $2/\eta$, and it starts to climb immediately afterwards. This means that gradient descent closely tracks the gradient flow trajectory so long as the sharpness remains less than $2/\eta$. Note that this finding was not a foregone conclusion: gradient descent is guaranteed to track the gradient flow trajectory in the limit of infinitesimal step sizes (since gradient descent is the forward Euler discretization of the gradient flow ODE), but for non-infinitesimal step sizes, there can be discretization error, as is studied in Barrett & Dherin (2020). Our empirical finding is essentially that this discretization error is very small compared to the difference between trajectories caused by instability.

On the other hand, for architectures with non-differentiable components such as ReLU or max-pooling, we sometimes observe that gradient descent tracks the Runge-Kutta trajectory so long as the sharpness remains less than $2/\eta$, but we also sometimes observe that the gradient descent trajectories differ from one another (and from Runge-Kutta) from the beginning of training. In the former case, we can infer that the gradient flow trajectory apparently does exist, and is returned by Runge-Kutta; in the latter case, we can infer that either (a) the gradient flow trajectory does not exist, or (b) that it does exist (and is returned by Runge-Kutta), but the step sizes we used for gradient descent were too large to track it.

H.1 FULLY-CONNECTED TANH NETWORK

H.1.1 SQUARE LOSS

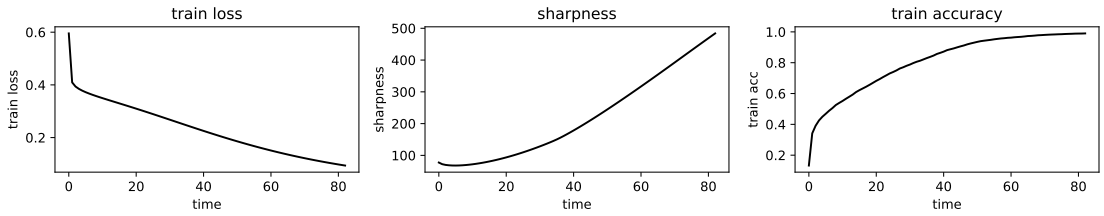

Figure 22: **Gradient flow.** We train the network to 99% accuracy with gradient flow, by using the Runge-Kutta method to discretize the gradient flow ODE (details in §G.5). We plot the train loss (left), sharpness (center), and train accuracy (right) over time. Observe that the sharpness tends to continually increase (except for a slight decrease at initialization).

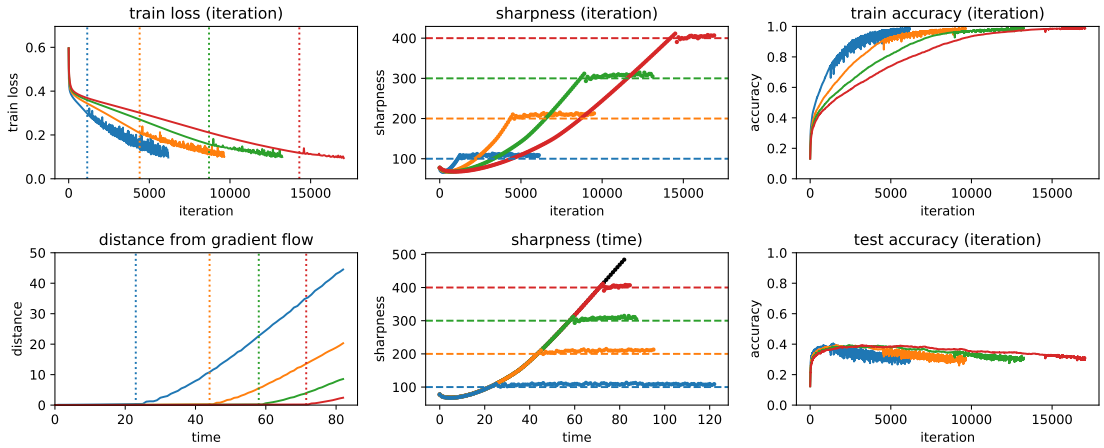

Figure 23: **Gradient descent.** We train the network to 99% accuracy using gradient descent at a range of step sizes η. **Top left:** we plot the train loss curves, with a vertical line marking the iteration where the sharpness first crosses $2/\eta$. Observe that the train loss monotonically decreases before this point, but behaves non-monotonically afterwards. **Top middle:** we plot the sharpness (measured at regular intervals during training). For each step size, the horizontal dashed line of the appropriate color marks the maximum stable sharpness $2/\eta$. Observe that the sharpness tends to increase during training until reaching the value $2/\eta$, and then hovers just a bit above that value. **Bottom left**: we track the ℓ_2 distance between (random projections of) the gradient *flow* iterate at time t and the gradient *descent* iterate at iteration t/η (details in §G.6). For each step size η, the vertical dotted line marks the time when the sharpness first crosses $2/\eta$. Observe that the distance is essentially zero until this time, and starts to grow afterwards. From this, we can conclude that gradient descent closely tracks the gradient flow trajectory (moving at a speed η) until the point on that trajectory where the sharpness reaches $2/\eta$. **Bottom middle**: to further visualize the previous point, we plot the evolution of sharpness during gradient descent, but with time ($=$ iteration $\times$ step size) on the x-axis rather than iteration. We plot the gradient *flow* sharpness in black. Observe that the gradient descent sharpness matches the gradient flow sharpness until reaching the value $2/\eta$.

H.1.2 CROSS-ENTROPY LOSS

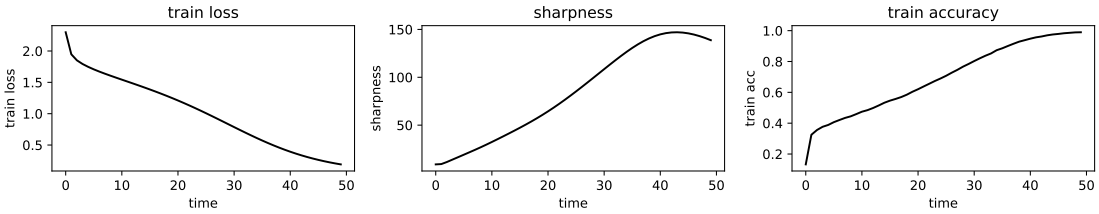

Figure 24: **Gradient flow.** Refer to the Figure 22 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

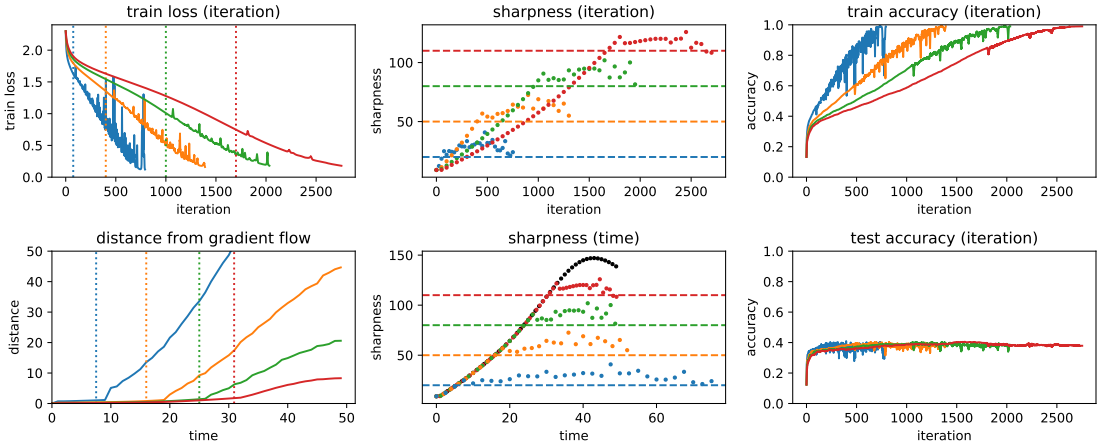

Figure 25: **Gradient descent.** Refer to the Figure 23 caption for more information.

H.2 FULLY-CONNECTED ELU NETWORK

H.2.1 SQUARE LOSS

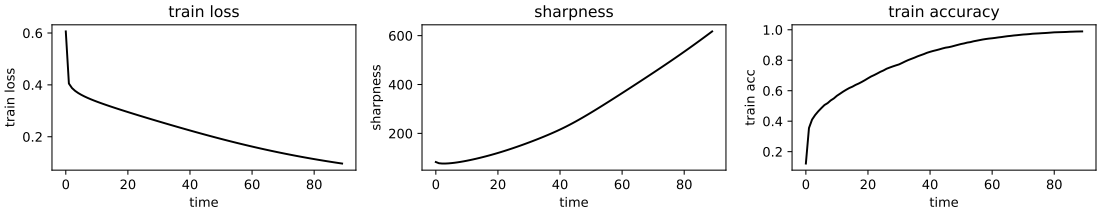

Figure 26: **Gradient flow.** Refer to the Figure 22 caption for more information.

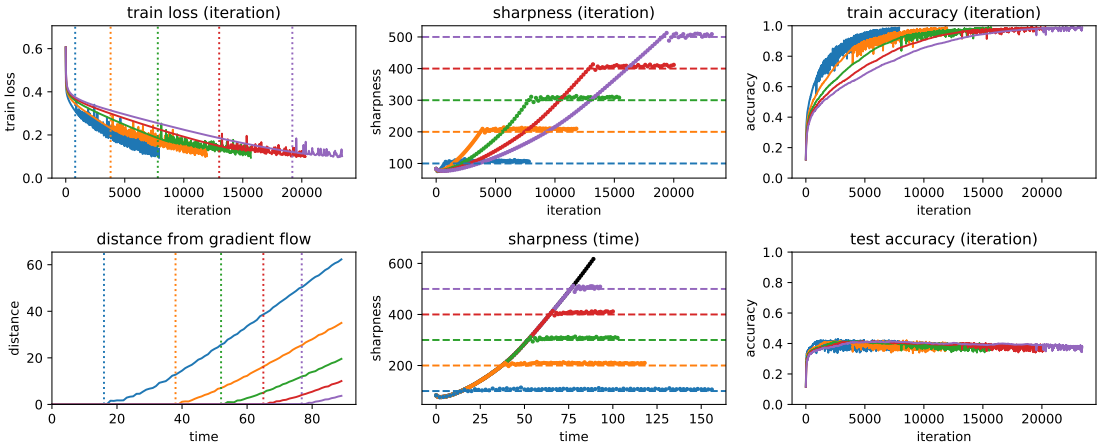

Figure 27: **Gradient descent.** Refer to the Figure 23 caption for more information.

H.2.2 CROSS-ENTROPY LOSS

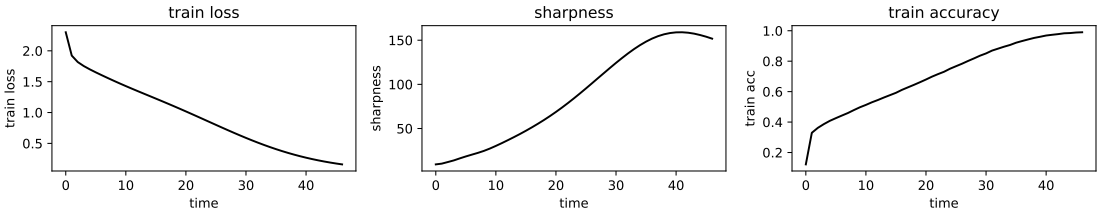

Figure 28: **Gradient flow.** Refer to the Figure 22 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

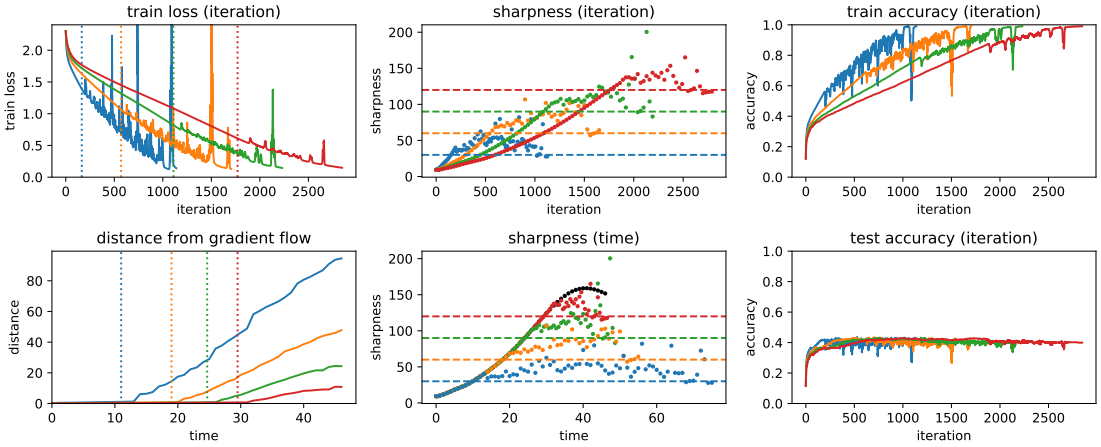

Figure 29: **Gradient descent.** Refer to the Figure 23 caption for more information.

H.3 FULLY-CONNECTED SOFTPLUS NETWORK

H.3.1 SQUARE LOSS

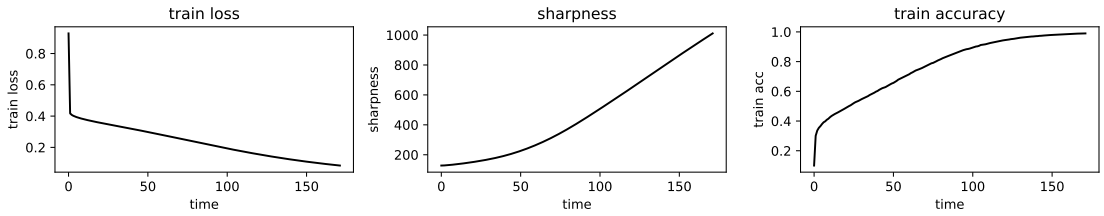

Figure 30: **Gradient flow.** Refer to the Figure 22 caption for more information.

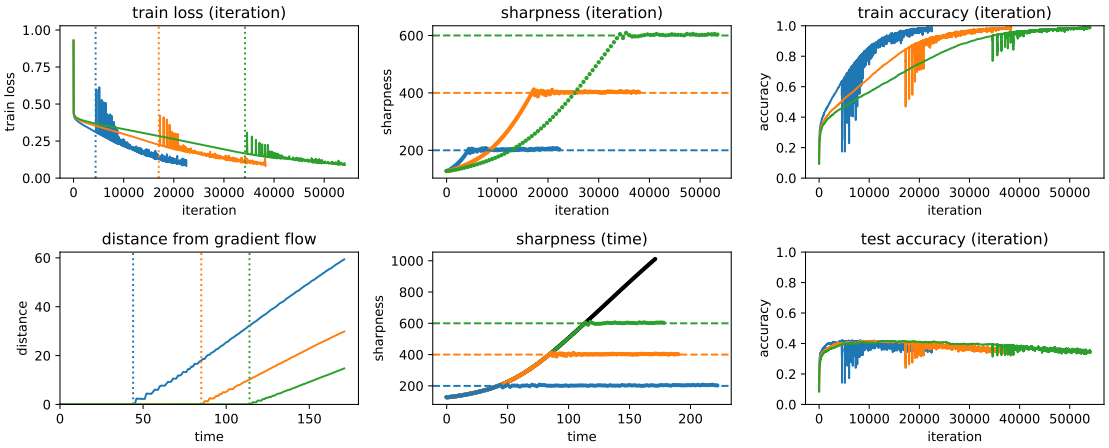

Figure 31: **Gradient descent.** Refer to the Figure 23 caption for more information.

H.3.2 CROSS-ENTROPY LOSS

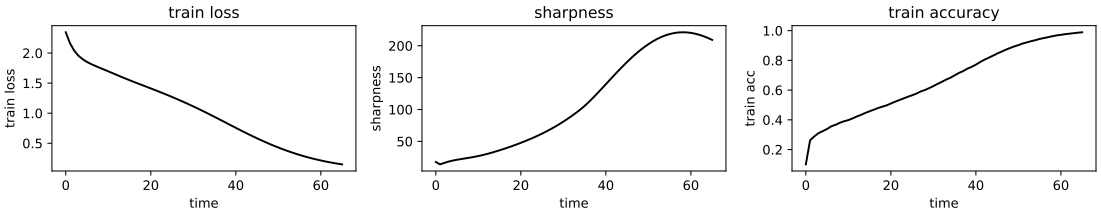

Figure 32: **Gradient flow.** Refer to the Figure 22 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

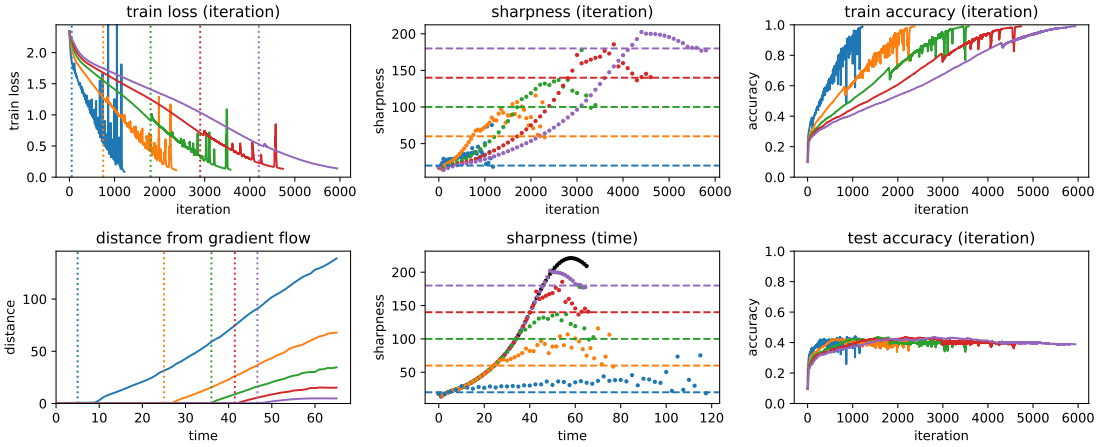

Figure 33: **Gradient descent.** Refer to the Figure 23 caption for more information.

H.4 FULLY-CONNECTED RELU NETWORK

Note that since the ReLU activation function is not continuously differentiable, the training objective is not continuously differentiable, and so a unique gradient flow trajectory is not guaranteed to exist.

H.4.1 SQUARE LOSS

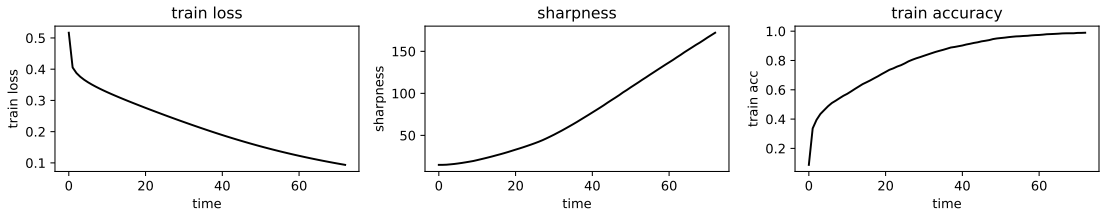

Figure 34: **Runge-Kutta.** We train the network to 99% accuracy using the Runge-Kutta algorithm. (Since a unique gradient flow trajectory is not guaranted to exist, we hesitate to call this "gradient flow"; Runge-Kutta should essentially be viewed as gradient descent with a very small step size.) Observe that the sharpness tends to continually increase.

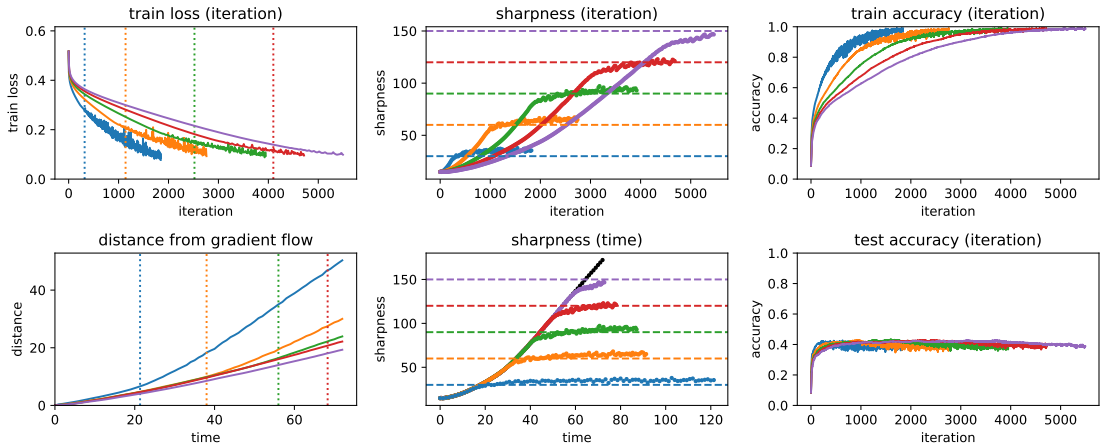

Figure 35: **Gradient descent. All panes except bottom left**: refer to the Figure 23 caption for more information. **Bottom left**: We track the ℓ_2 distance between (random projections of) the Runge-Kutta iterate at time t and the gradient descent iterate at iteration t/η. For each step size η, the vertical dotted loss marks the time when the sharpness first crosses $2/\eta$. In contrast to Figure 23, here the distance between gradient descent and gradient flow starts to noticeably grow from the beginning of training. From this we conclude that for this architecture, gradient descent does *not* track the Runge-Kutta trajectory at first.

H.4.2 CROSS-ENTROPY LOSS

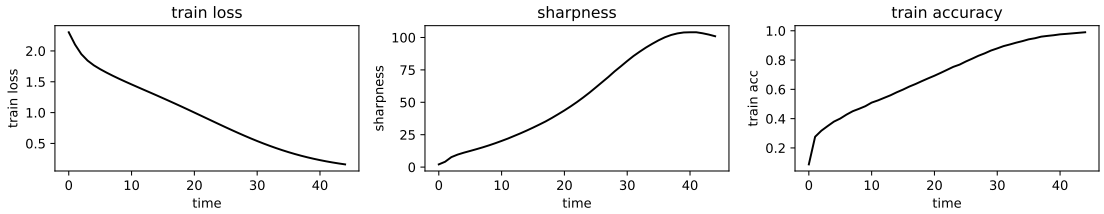

Figure 36: **Runge-Kutta.** Refer to the Figure 34 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

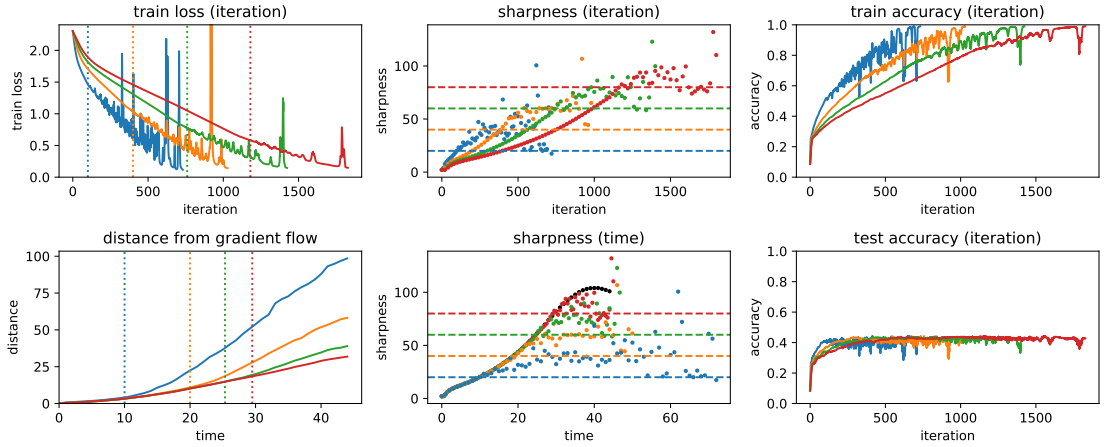

Figure 37: **Gradient descent.** Refer to the Figure 35 caption for more information.

H.5 FULLY-CONNECTED HARD TANH NETWORK

Note that since the hardtanh function is not continuously differentiable, the training objective is not continuously differentiable, and so a unique gradient flow trajectory is not guaranteed to exist.

H.5.1 SQUARE LOSS

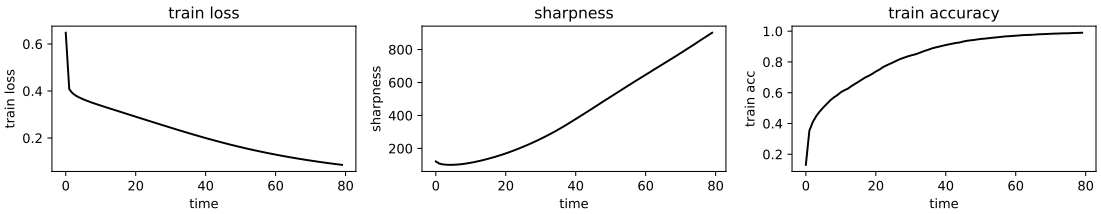

Figure 38: **Runge-Kutta.** Refer to the Figure 34 caption for more information.

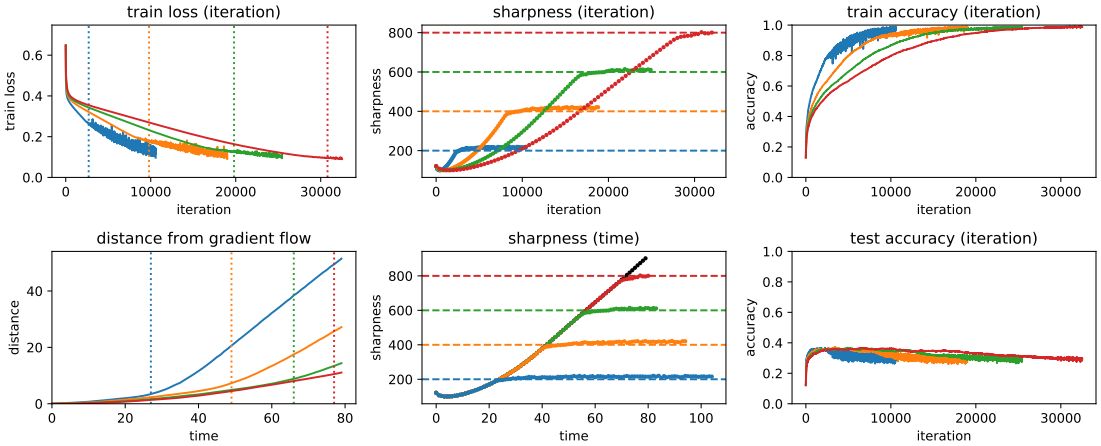

Figure 39: **Gradient descent.** Refer to the Figure 35 caption for more information.

H.5.2 CROSS-ENTROPY LOSS

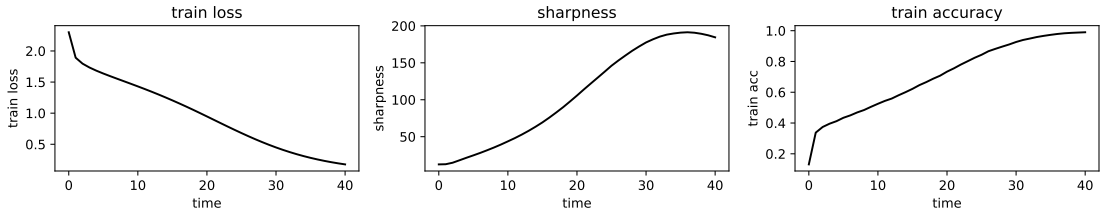

Figure 40: **Runge-Kutta.** Refer to the Figure 34 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

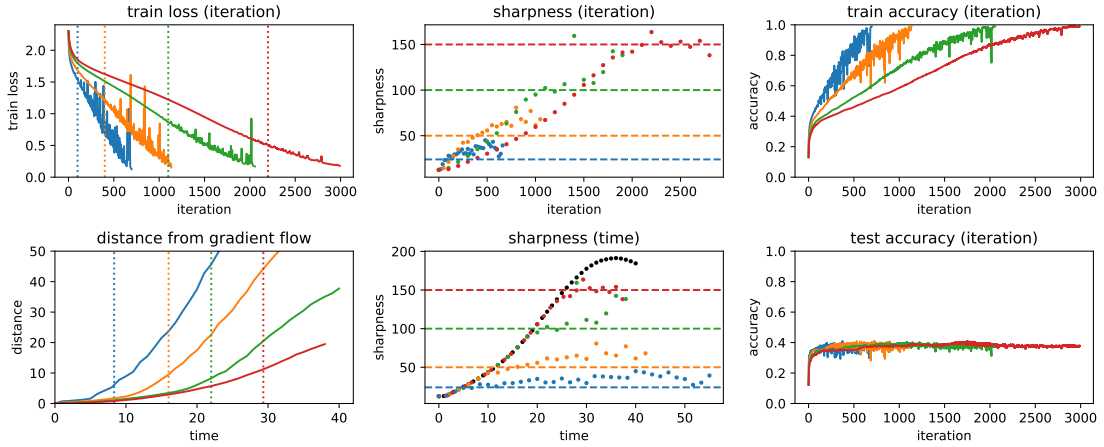

Figure 41: **Gradient descent.** Refer to the Figure 35 caption for more information.

H.6 CONVOLUTIONAL TANH NETWORK WITH MAX POOLING

Note that since max-polling is not continuously differentiable, the training objective is not continuously differentiable, and so a unique gradient flow trajectory is not guaranteed to exist.

H.6.1 SQUARE LOSS

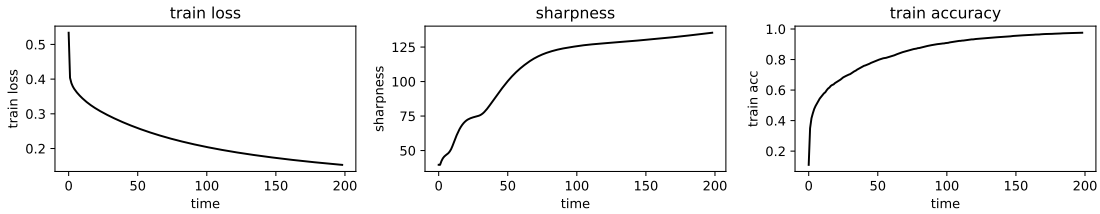

Figure 42: **Runge-Kutta.** We train the network to 99% accuracy using the Runge-Kutta algorithm. (Since a unique gradient flow trajectory is not guaranted to exist, we hesitate to call this "gradient flow.") Observe that the sharpness tends to continually increase.

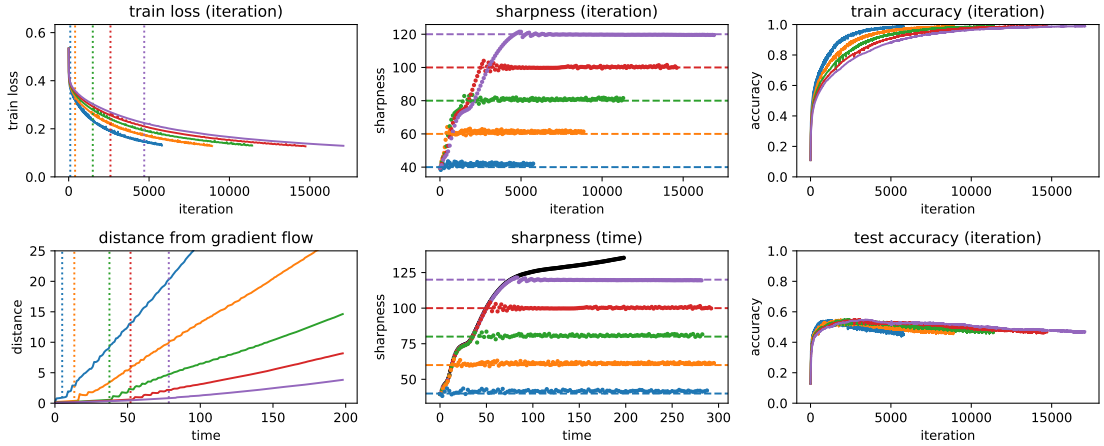

Figure 43: **All panes except bottom left**: refer to the Figure 23 caption for more information. **Bottom left**: We track the ℓ_2 distance between (random projections of) the Runge-Kutta iterate at time t and the gradient descent iterate at iteration t/η. For each step size η, the vertical dotted loss marks the time when the sharpness first crosses $2/\eta$. Observe that the distance is essentially zero until this time, and starts to grow afterwards. From this, we can conclude that even though the training objective is not differentiable, gradient descent does closely tracks the Runge-Kutta trajectory (moving at a speed η) until the point on that trajectory where the sharpness reaches $2/\eta$.

H.6.2 CROSS-ENTROPY LOSS

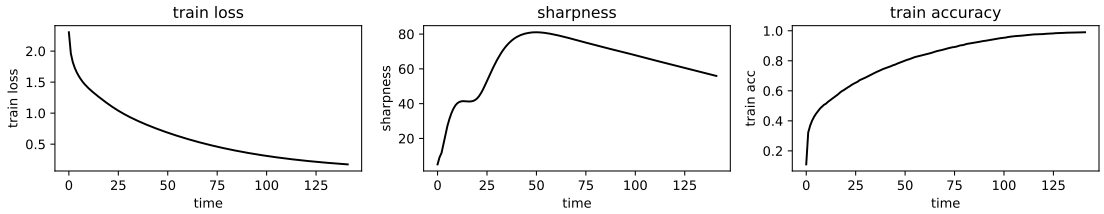

Figure 44: **Runge-Kutta.** Refer to the Figure 42 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

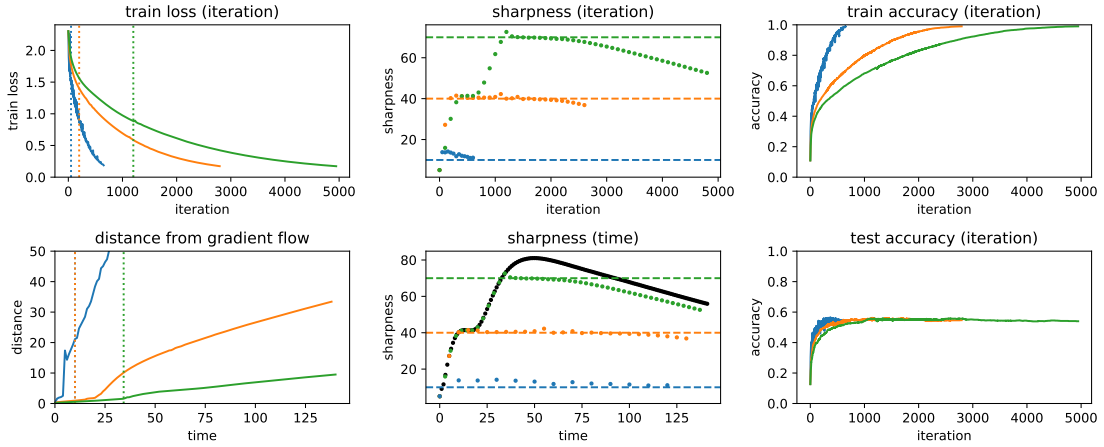

Figure 45: **Gradient descent.** Refer to the Figure 43 caption for more information.

H.7 CONVOLUTIONAL ELU NETWORK WITH MAX POOLING

Note that since max-polling is not continuously differentiable, the training objective is not continuously differentiable, and so a unique gradient flow trajectory is not guaranteed to exist.

H.7.1 SQUARE LOSS

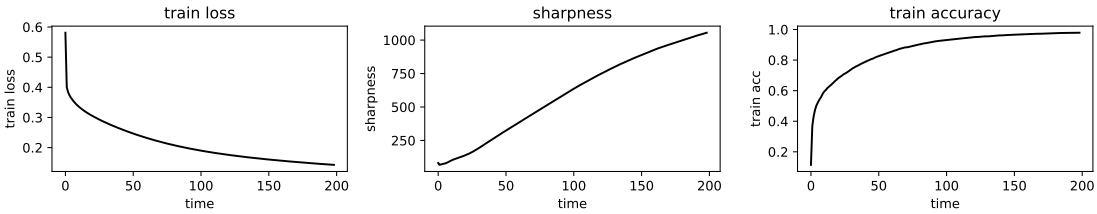

Figure 46: **Runge-Kutta.** Refer to the Figure 42 caption for more information.

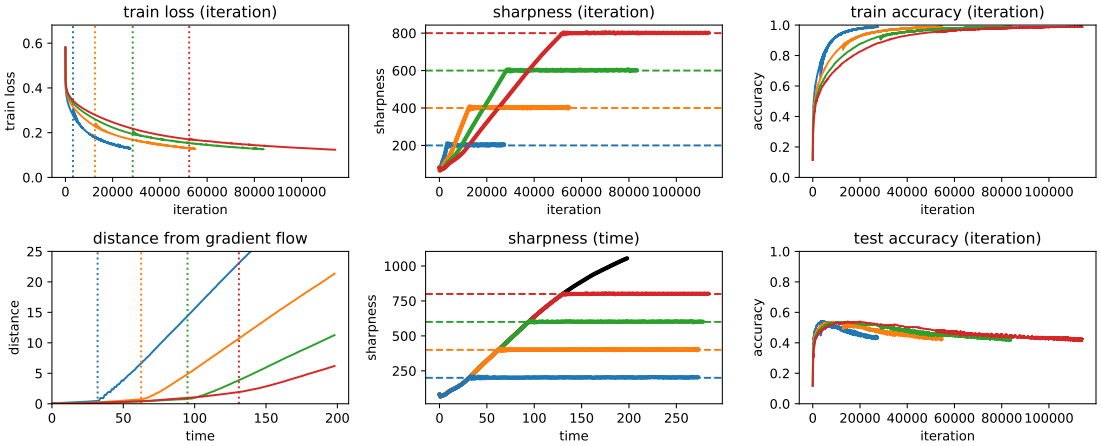

Figure 47: **Gradient descent.** Refer to the Figure 43 caption for more information.

H.7.2 CROSS-ENTROPY LOSS

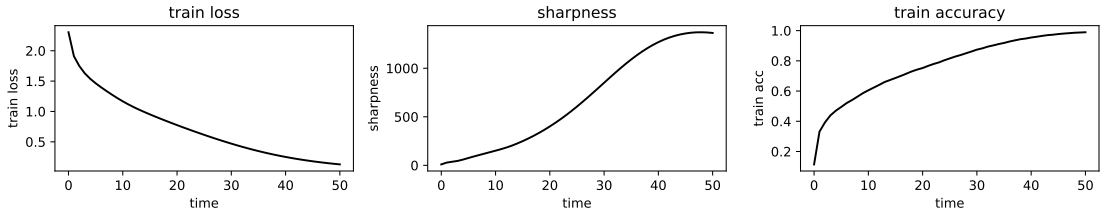

Figure 48: **Runge-Kutta.** Refer to the Figure 42 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

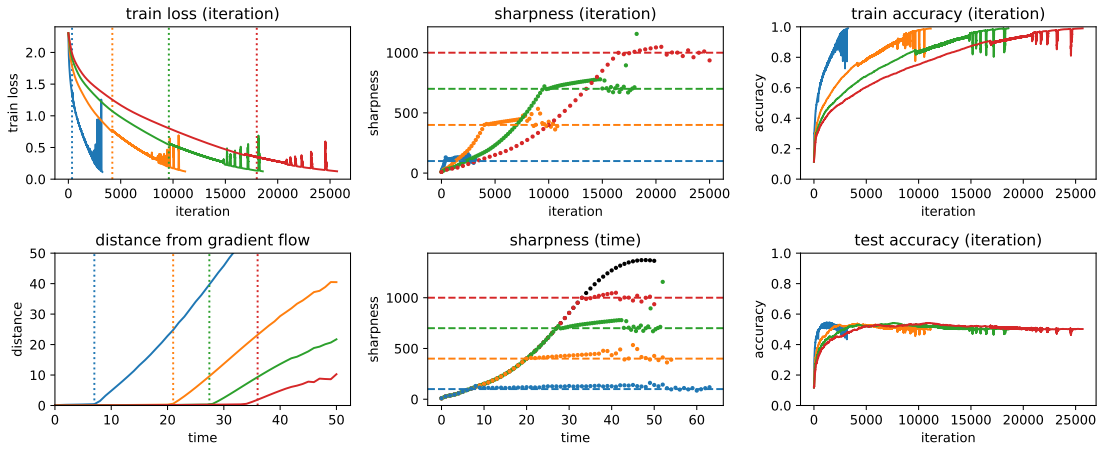

Figure 49: **Gradient descent.** Refer to the Figure 43 caption for more information.

H.8 CONVOLUTIONAL RELU NETWORK WITH MAX POOLING

Note that since ReLU and max-polling are not continuously differentiable, the training objective is not continuously differentiable, and so a unique gradient flow trajectory is not guaranteed to exist.

H.8.1 SQUARE LOSS

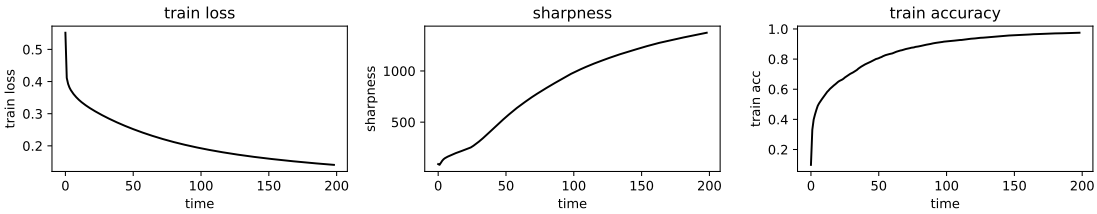

Figure 50: **Runge-Kutta.** Refer to the Figure 42 caption for more information.

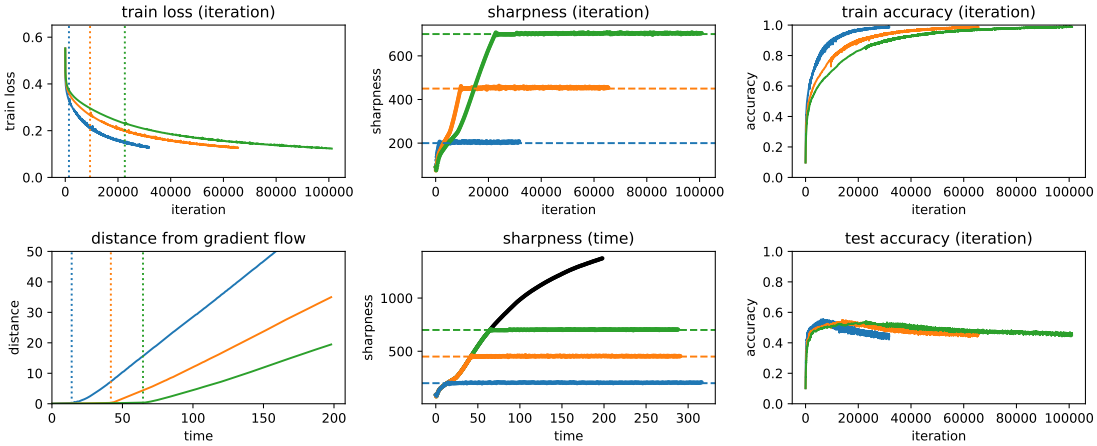

Figure 51: **Gradient descent.** Refer to the Figure 43 caption for more information.

H.8.2 CROSS-ENTROPY LOSS

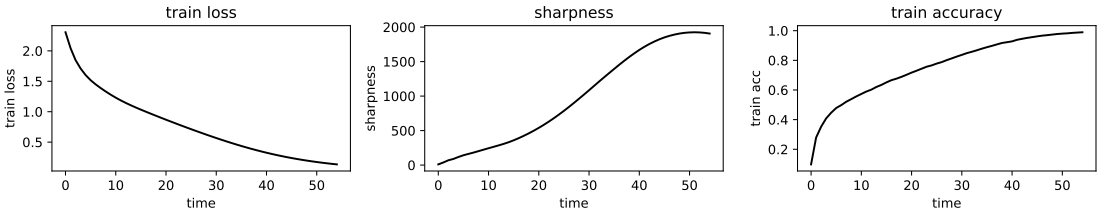

Figure 52: **Runge-Kutta.** Refer to the Figure 42 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

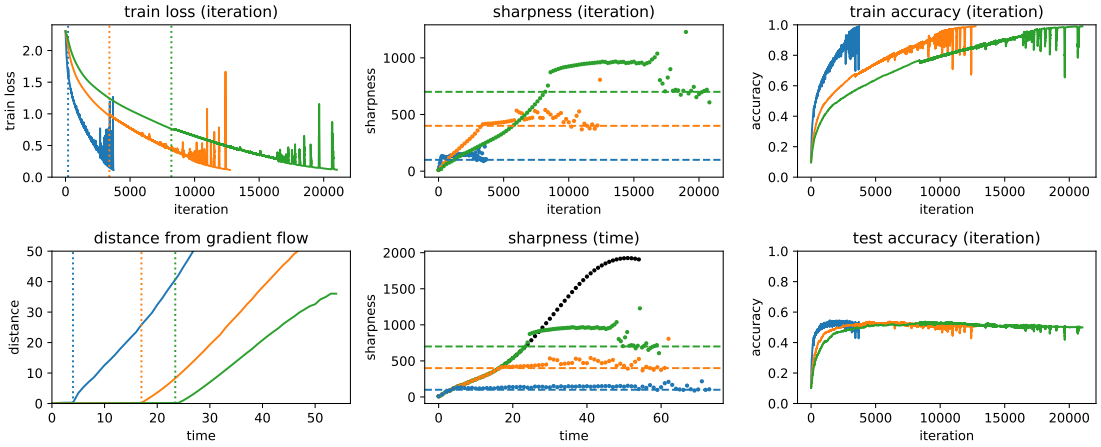

Figure 53: **Gradient descent.** Refer to the Figure 43 caption for more information.

H.9 CONVOLUTIONAL TANH NETWORK WITH AVERAGE POOLING

H.9.1 CROSS-ENTROPY LOSS

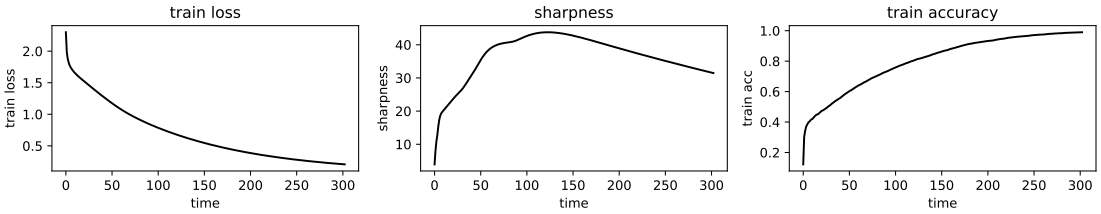

Figure 54: **Gradient flow.** Refer to the Figure 22 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

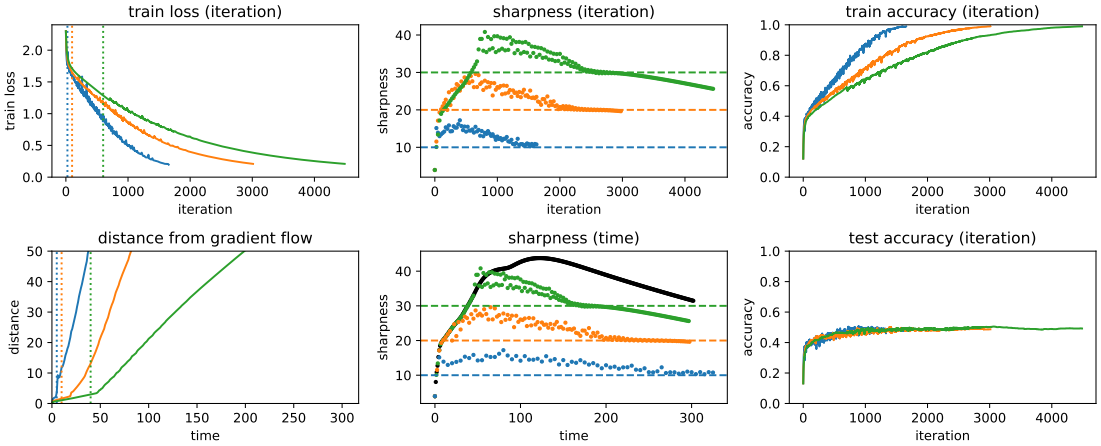

Figure 55: **Gradient descent.** Refer to the Figure 23 caption for more information.

H.10 CONVOLUTIONAL ELU NETWORK WITH AVERAGE POOLING

H.10.1 SQUARE LOSS

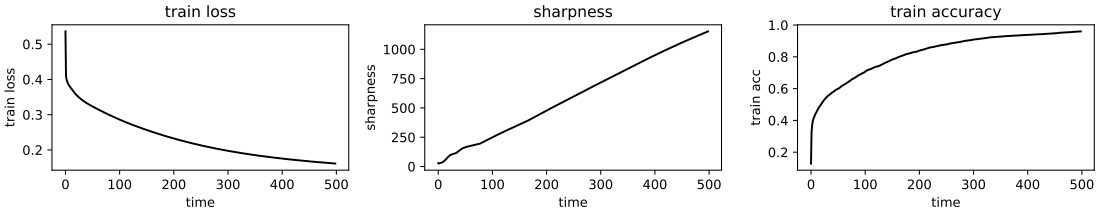

Figure 56: **Gradient flow.** Refer to the Figure 22 caption for more information.

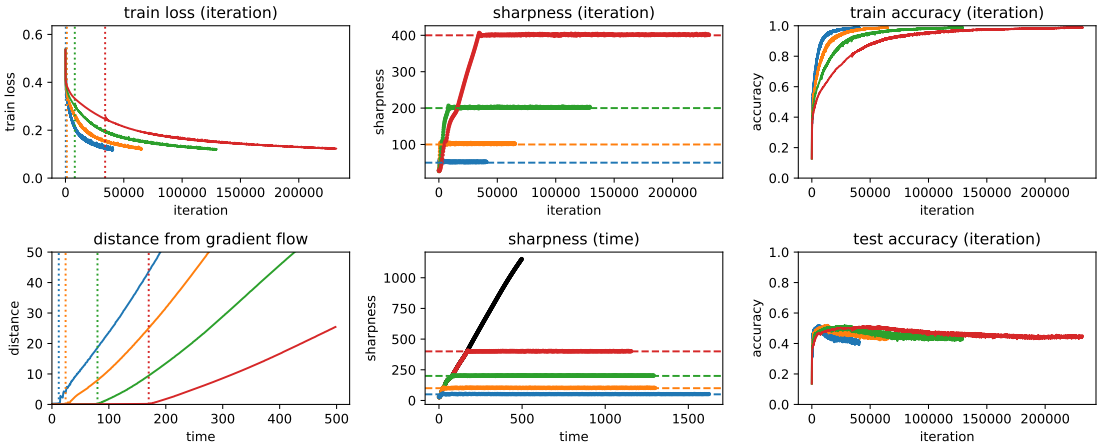

Figure 57: **Gradient descent.** Refer to the Figure 23 caption for more information.

H.10.2 CROSS-ENTROPY LOSS

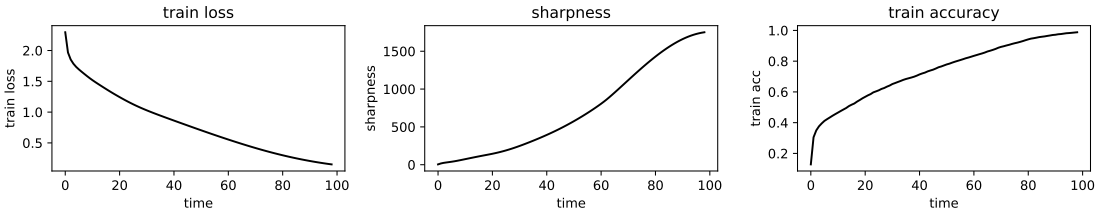

Figure 58: **Gradient flow.** Refer to the Figure 22 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

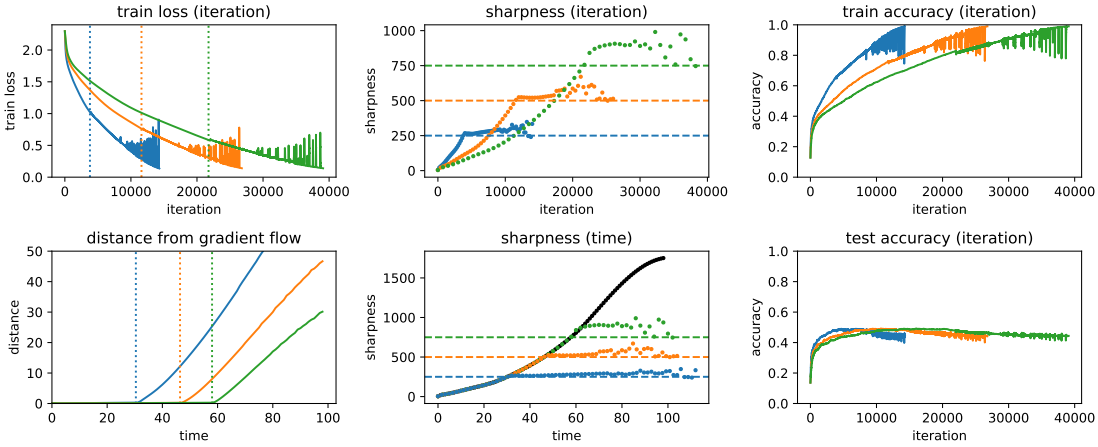

Figure 59: **Gradient descent.** Refer to the Figure 23 caption for more information.

H.11 CONVOLUTIONAL RELU NETWORK WITH AVERAGE POOLING

Note that since ReLU is not continuously differentiable, the training objective is not continuously differentiable, and so a unique gradient flow trajectory is not guaranteed to exist.

H.11.1 SQUARE LOSS

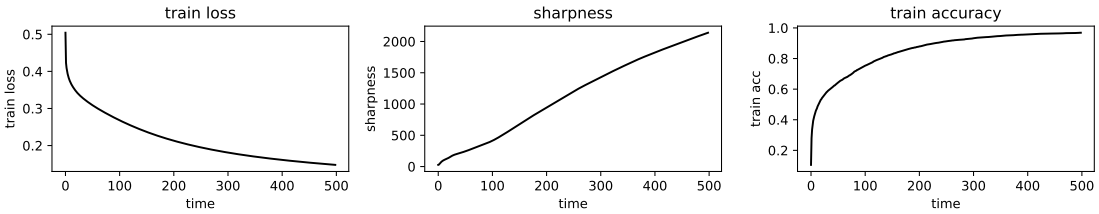

Figure 60: **Runge-Kutta.** Refer to the Figure 42 caption for more information.

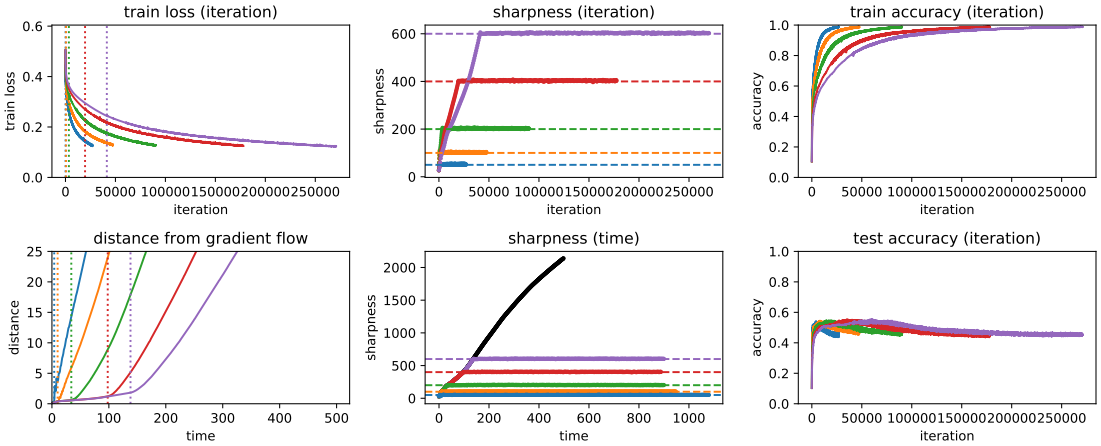

Figure 61: **Gradient descent.** Refer to the Figure 43 caption for more information.

H.11.2 CROSS-ENTROPY LOSS

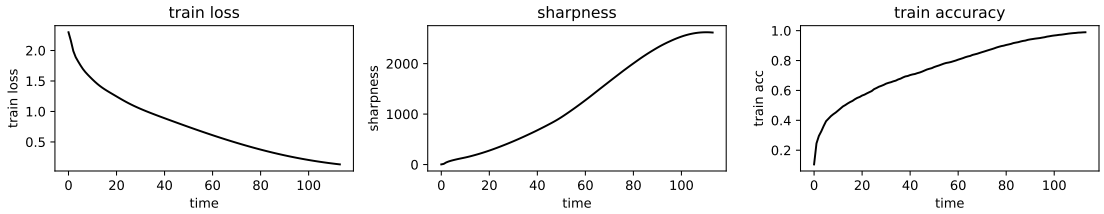

Figure 62: **Runge-Kutta.** Refer to the Figure 42 caption for more information. Additionally, in this figure the sharpness drops at the end of training due to the cross-entropy loss.

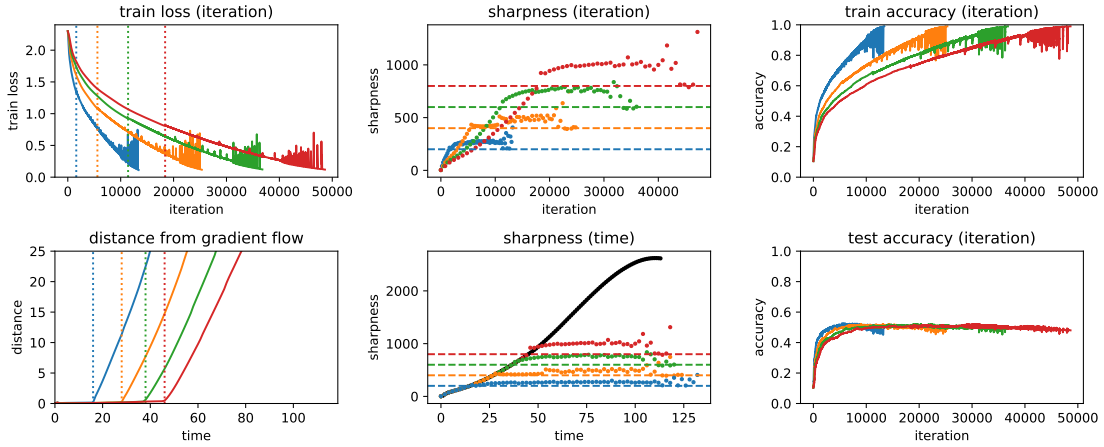

Figure 63: **Gradient descent.** Refer to the Figure 43 caption for more information.

I BATCH NORMALIZATION EXPERIMENTS

In this appendix, we demonstrate that our findings hold for networks that are trained with batch normalization (BN) (Ioffe & Szegedy, 2015). We experiment on a size-5,000 subset of CIFAR-10, and we consider convolutional networks with three different activation functions: ELU (Figure 64-65, tanh (Figure 66-67), and ReLU (Figure 68-69). See §G.3 for experimental details.

Empirically, our findings hold for batch-normalized networks. The one catch is that when training batch-normalized networks at very small step sizes, it is apparently inadequate to measure the sharpness directly at the iterates themselves, as we do elsewhere in the paper. Namely, observe that in Figure 65(e), when we run gradient descent at the red step size, the sharpness (measured directly at the iterates) plateaus a bit *beneath* the value $2/\eta$. At first, this might sound puzzling: after all, if the sharpness is less than $2/\eta$ then gradient descent should be stable. The explanation is that the sharpness *in between* successive iterates does in fact cross $2/\eta$. In Figure 65(b), we track the maximum sharpness on the path "in between" successive iterates. (To estimate the maximum sharpness between a pair of successive iterates, we compute the sharpness at a grid of eight points spaced evenly between them, and then take the maximum of these values.) Observe that this quantity does rise to $2/\eta$ and hover there. We do not know why measuring the sharpness between iterates is necessary for batch-normalized networks, whereas for non-BN networks it suffices to measure the sharpness only at the iterates themselves.

In §I.1, we reconcile these findings with Santurkar et al. (2018).

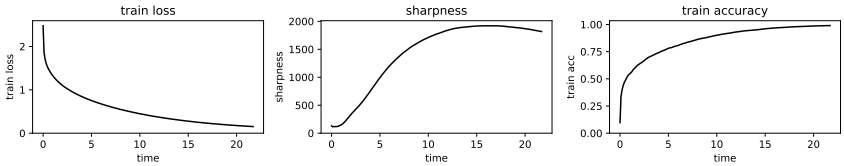

Figure 64: We train a **ELU CNN (+ BN)** using **gradient flow**.

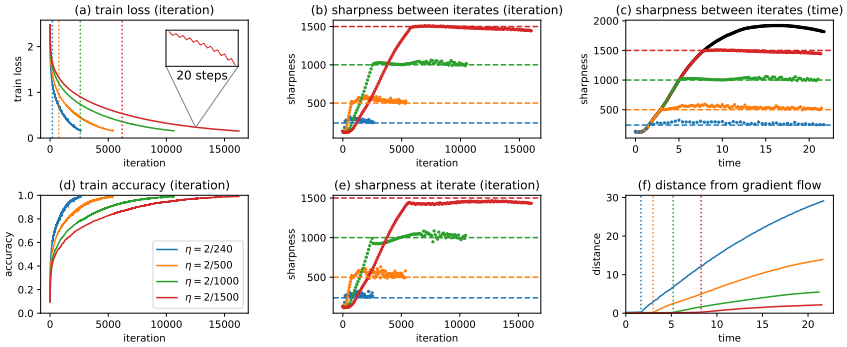

Figure 65: We train an **ELU CNN (+ BN)** using **gradient decent** at a range of step sizes. **(a)** we plot the train loss, with a vertical dotted line marking the iteration where the sharpness on the path first crosses $2/\eta$. The inset shows that the red curve is indeed behaving non-monotonically. **(b)** we track the sharpness "between iterates." This means that instead of computing the sharpness right at the iterates themselves (as we do elsewhere in the paper, and in pane (e) here), we compute the maximum sharpness on the line between between successive iterates. Observe that this quantity rises to $2/\eta$ (marked by the horizontal dashed line) and then hovers right at, or just above that value. **(c)** we plot the same quantity by "time" (= iteration $\times$ step size) rather than iteration. **(e)** we plot the sharpness at the iterates themselves. Note that for the red step size, this quantity plateaus at a value that is beneath $2/\eta$. **(f)** distance from the gradient flow trajectory.

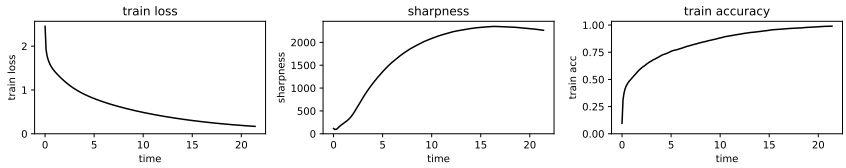

Figure 66: We train a **tanh CNN (+ BN)** using **gradient flow**.

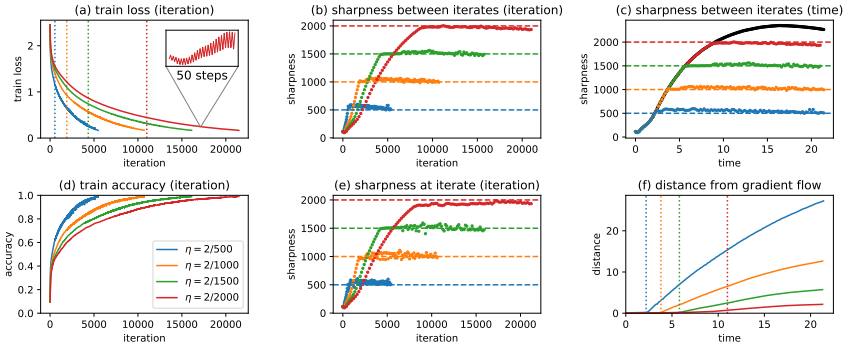

Figure 67: We train a **tanh CNN (+ BN)** using **gradient decent**.

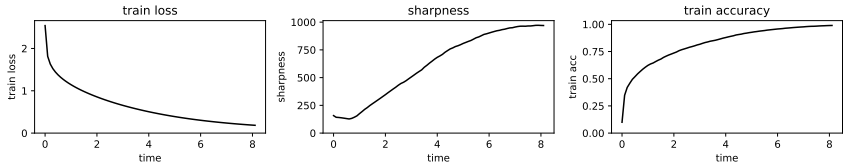

Figure 68: We train a **ReLU CNN (+ BN)** using **gradient flow**.

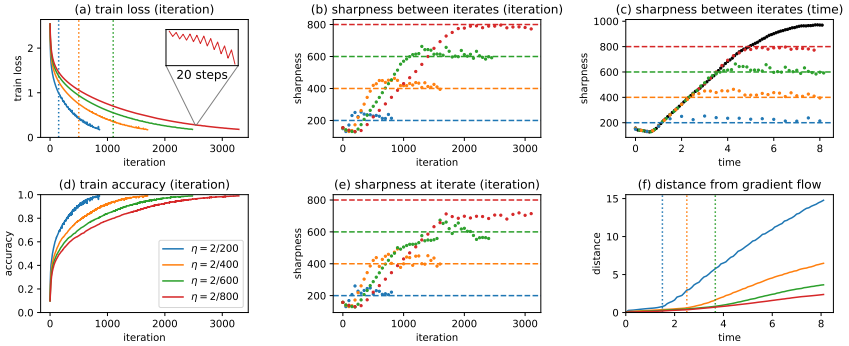

Figure 69: We train a **ReLU CNN (+ BN)** using **gradient descent**.

I.1 Relation to Santurkar et al. (2018)

We have demonstrated that the sharpness hovers right at (or just above) the value $2/\eta$ during gradient descent training of both BN and non-BN networks (at reasonable step sizes). Therefore, at least in the case of full-batch gradient descent, it cannot be said that batch normalization decreases the sharpness (i.e. improves the local L-smoothness) along the optimization trajectory.

Santurkar et al. (2018) argued that batch normalization improves the *effective smoothness* along the optimization trajectory, where effective smoothness is defined as the Lipschitz constant of the gradient in the update direction (i.e. the negative gradient direction, for full-batch GD). That is, given an objective function f, an iterate θ, and a distance α, the effective smoothness of f at parameter θ and distance α is defined in Santurkar et al. (2018) as

$$\sup_{\gamma \in [0,\alpha]} \frac{\|\nabla f(\theta) - \nabla f(\theta - \gamma \nabla f(\theta))\|_2}{\|\gamma \nabla f(\theta)\|_2}$$

where the $\sup$ can be numerically approximated by evaluating the given ratio at several values γ spaced uniformly between 0 and α.

In Figure 70, we train two ReLU CNNs — one with BN, one without — at the same set of step sizes, and we monitor both the sharpness (i.e. the L-smoothness) and the effective smoothness. When computing the effective smoothness, we use a distance $\alpha = \eta$. Observe that for both the BN and the non-BN network, the effective smoothness initially hovers around zero (indicating that the gradient does not point in the direction of highest curvature), but once gradient descent enters the Edge of Stability, the effective smoothness jumps to the value $2/\eta$ and then remains there. Thus, at least for full-batch gradient descent on this particular architecture, batch normalization does *not* improve the effective smoothness along the optimization trajectory. (Despite this, note that for each step size, the BN network trains faster than the non-BN network, confirming that BN does accelerate training.)

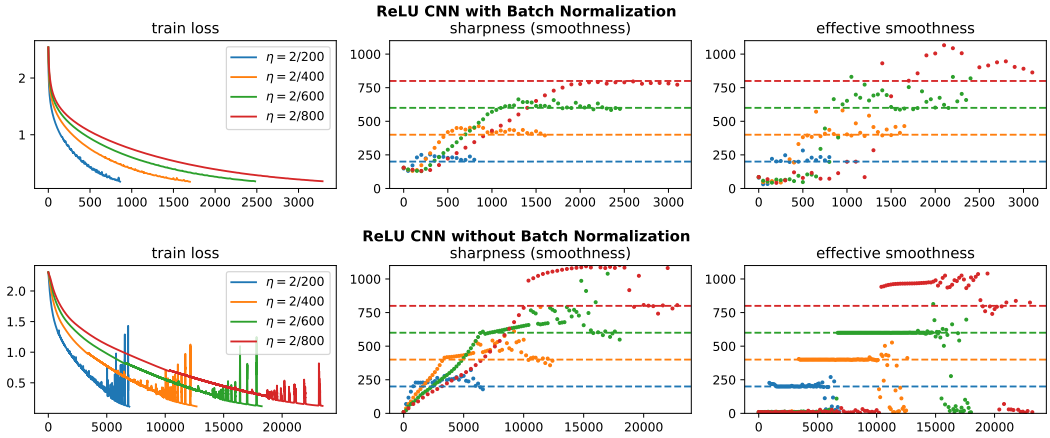

Figure 70: On a 5,000-size subset of CIFAR-10, we train a ReLU CNN both with BN (**top row**) and without BN (**bottom row**) at the same grid of step sizes. We plot the sharpness/smoothness (**center column**) as well as the effective smoothness (Santurkar et al., 2018) (**right column**). Observe that for both networks, the effective smoothness hovers around zero initially, and jumps up to $2/\eta$ once gradient descent enters the Edge of Stability.

Note that this finding is actually consistent with Figure 4(c) in Santurkar et al. (2018), which is meant to show that BN improves effective smoothness when training a VGG network using SGD. Their Figure 4(c) shows that during SGD with step size $\eta = 0.1$, the effective smoothness hovers around the value 20 for both the BN and the non-BN network. Since $20 = 2/(0.1)$, this is fully consistent with our findings (though they use SGD rather than full-batch GD). Figure 4(c) does show that the effective smoothness behaves more *regularly* for the BN network than for the non-BN network. But we disagree with their interpretation of this figure as demonstrating that BN improves the effective smoothness during training.

The other piece of evidence in Santurkar et al. (2018) in support of the argument that batch normalization improves the effective smoothness during training is their Figure 9(c). This figure shows that a deep linear network (DLN) trained without BN has a much larger (i.e. worse) effective smoothness during training than a DLN trained with BN. However, for this figure, the distance α used in the computation of effective smoothness was larger than the training step size η by a factor of 30. The effective smoothness at distances larger than the step size is not relevant for training. We have

verified that when effective smoothness is computed at a distance equal to the training step size (i.e. $\alpha = \eta$), the effective smoothness for the DLN with BN and for the DLN without BN both hover right at $2/\eta$.

Specifically, we train a DLN both with (Figure 71) and without BN (Figure 72), and we measure the effective smoothness at a distance $\alpha = 30\eta$, as is done in Figure 9(c) of Santurkar et al. (2018). We use the same experimental setup and the same step size of $\eta =$ 1e-6 as they do, and we repeat the experiment across four random seeds. Observe that when training the BN network, the effective smoothness hovers right at $2/\eta$ (marked by the horizontal black line), whereas when training the non-BN network, the effective smoothness is much larger. This is consistent with Figure 9(c) in Santurkar et al. (2018).

On the other hand, in Figure 73 and 74, we measure the effective smoothness at the actual step size $\alpha = \eta$. When effective smoothness is computed in this way, we observe that for both the network with BN and the network without BN, the effective smoothness hovers right at $2/\eta$.

Therefore, we conclude that there is no evidence that the use of batch normalization improves either the smoothness or the effective smoothness along the optimization trajectory.

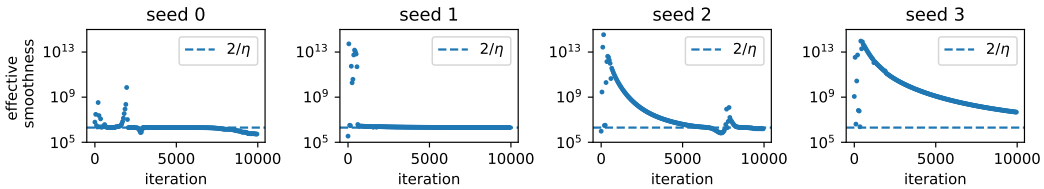

Figure 71: When training a deep linear network **without BN**, we measure effective smoothness at a distance $\alpha = 30\eta$ that is **30x larger than the step size**.

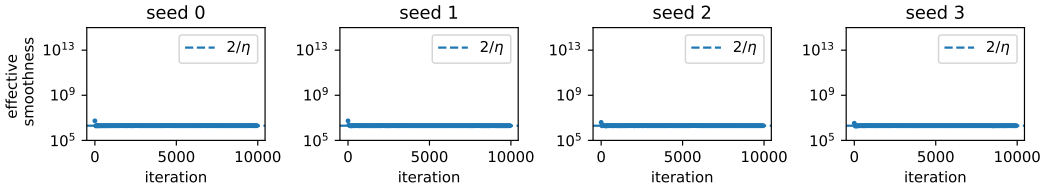

Figure 72: When training a deep linear network **with BN**, we measure effective smoothness at a distance $\alpha = 30\eta$ that is **30x larger than the step size**.

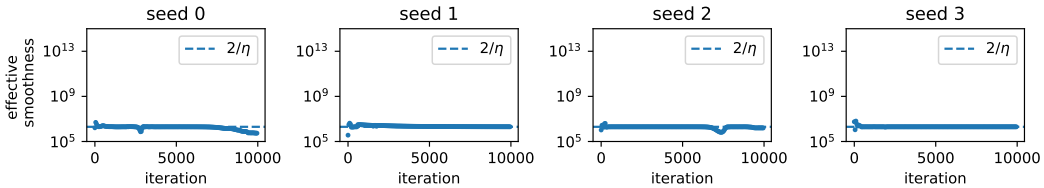

Figure 73: When training a deep linear network **without BN**, we measure effective smoothness at a distance $\alpha = \eta$ that is **equal to the step size**.

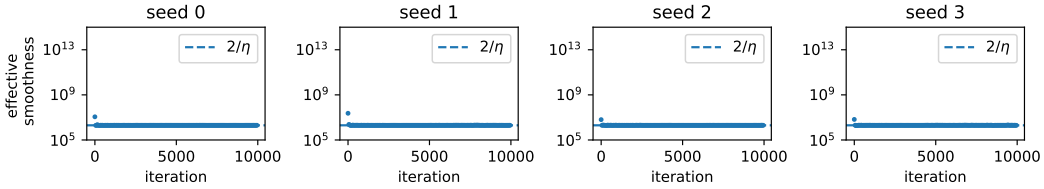

Figure 74: When training a deep linear network **with BN**, we measure effective smoothness at a distance $\alpha = \eta$ that is **equal to the step size**.

J ADDITIONAL TASKS

So far, we have verified our findings on image classification and language modeling. In this Appendix, we verify our findings on three additional tasks: training a Transformer on the WikiText-2 language modeling dataset, training a one-hidden-layer network on a one-dimensional toy regression task (J.2), and training a deep linear network (J.3).

J.1 TRANSFORMER ON WIKITEXT-2

In Figure 75, we use gradient descent to train a Transformer on the WikText-2 word-level language modeling dataset (Merity et al., 2016). See §G.4 for full experimental details. Training this network to completion would be impractical, so we instead train at each step size for 20,000 iterations. Consistent with our general findings, for each step size η, we observe that the sharpness rises to $2/\eta$ and then hovers right at, or just above, that value. However, for this Transformer, we do *not* observe that gradient descent closely tracks the gradient flow trajectory at the beginning of training.

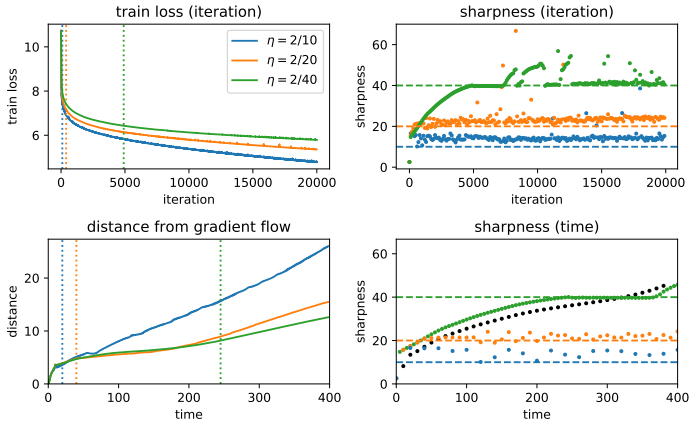

Figure 75: **Training a Transformer using gradient descent**. We train a Transformer for 20,000 iterations on the WikiText-2 language modeling dataset. (For this problem, training to completion would not be computationally practical.). **Top left:** we plot the train loss curves, with a vertical dotted line marking the iteration where the sharpness first crosses $2/\eta$. The train loss decreases monotonically before this line, but behaves non-monotonically afterwards. **Top right:** we plot the evolution of the sharpness, with a horizontal dashed line marking the value $2/\eta$. Observe that for each step size, the sharpness rises to $2/\eta$ and then hovers right at, or just above, that value. **Bottom left**: for the initial phase of training, we plot the distance between the gradient descent trajectory and the gradient flow trajectory, with a vertical dotted line marking the gradient descent iteration where the sharpness first crosses $2/\eta$. Observe that the distance begins to rise from the start of training, indicating that for this architecture, gradient descent does *not* track the gradient flow trajectory initially. **Bottom right**: for the initial phase of training, we plot the evolution of the sharpness by "time" = iteration $\times$ η rather than iteration. The black dots are the sharpness during gradient flow.

J.2 ONE-DIMENSIONAL TOY REGRESSION TASK

Task Our toy regression problem is to approximate a Chebyshev polynomial using a neural network. To generate a toy dataset, we take 20 points spaced uniformly on the interval $[-1, 1]$, and we label them noiselessly using the Chebyshev polynomial of some degree k. Note that the Chebyshev polynomial of degree k is a polynomial with k zeros with range $[-1, 1]$ on the domain $[-1, 1]$. Figure 76 shows the Chebyshev datasets for degree 3, 4, and 5.

Network For the network, we use a tanh architecture with one hidden layer of $h = 100$ units, initialized using Xavier initialization. We train using the MSE loss until the loss reaches 0.05.

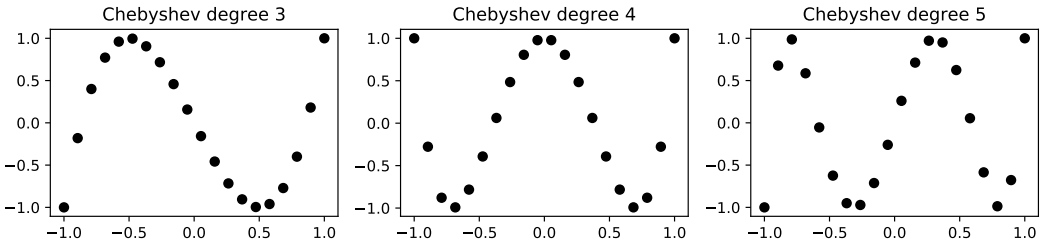

Figure 76: The datasets for the toy one-dimensional regression task.

Results In Figure 77, we fit the Chebyshev degree 3, 4, and 5 datasets using gradient flow. Empirically, the higher the degree, the more the sharpness rises during the course of training: on the degree 3 polynomial, the sharpness rises by a factor of 1.2; on the degree 4 polynomial, by a factor of 3.2; and on the degree 5 polynomial, by a factor of 63.5.

In Figure 78 and Figure 79, we fit the degree 4 and 5 datasets using gradient descent at a range of step sizes. We observe mostly the same Edge of Stability behavior as elsewhere in the paper. The only difference is that for the degree 5 dataset, after the sharpness hits $2/\eta$, the training loss *first* undergoes a temporary period of non-monotonicity in which no progress is made, and *then* it decreases monotonically until training is finished (in contrast to our other experiments where we observe that training loss behaves non-monotonically at the same time as it is consistently decreasing).

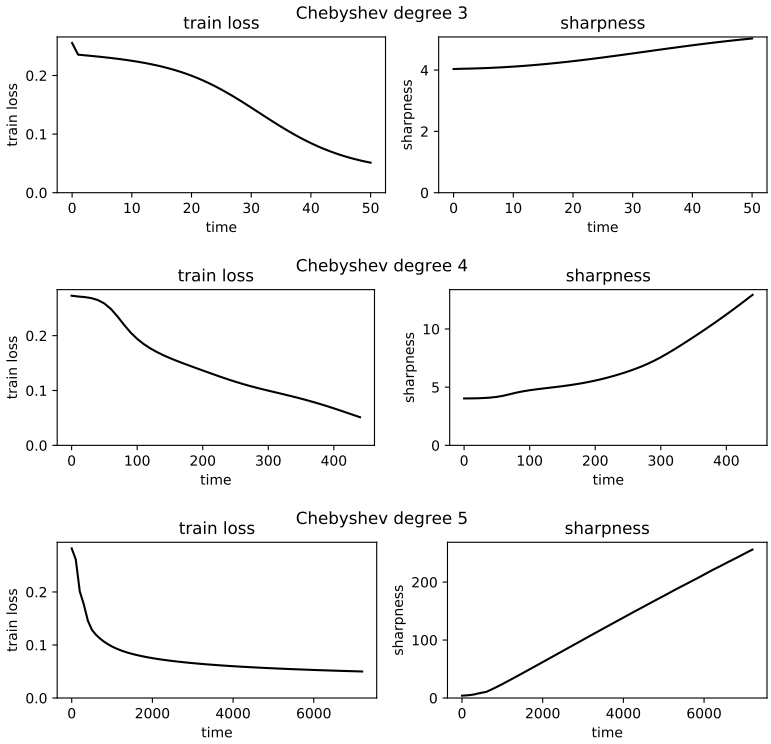

Figure 77: **Fitting Chebyshev polynomials using gradient flow**. We use gradient flow to fit a one-hidden-layer tanh network to the Chebyshev polynomials of degrees 3, 4, and 5. We observe that the sharpness rises more when fitting a higher-degree polynomial, likely because the dataset is more complex.

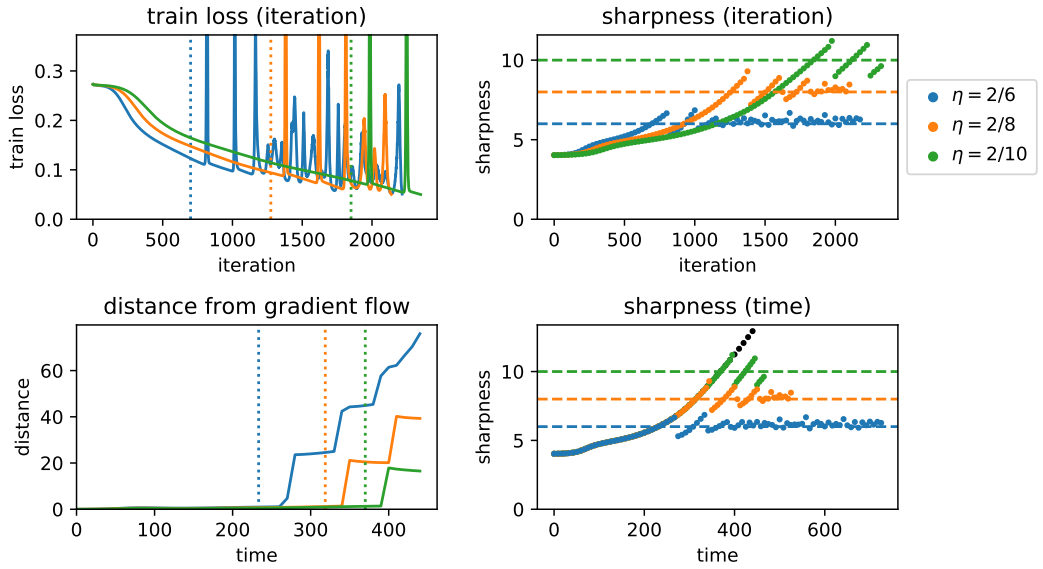

Figure 78: **Chebyshev degree 4 (gradient descent)**. We fit the Chebyshev polynomial of degree 4 using gradient descent at a range of step sizes (see legend).

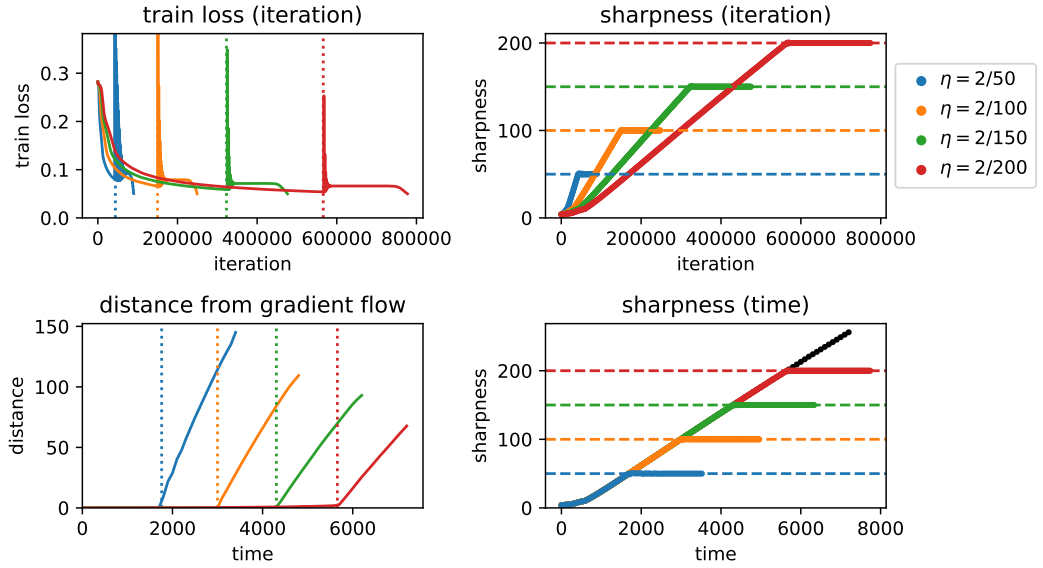

Figure 79: **Chebyshev degree 5 (gradient descent)**. We fit the Chebyshev polynomial of degree 5 using gradient descent at a range of step sizes (see legend).

J.3 Deep linear network

Task The task is to map n inputs $\mathbf{x}_1, \ldots, \mathbf{x}_n \subseteq \mathbb{R}^d$ to n targets $\mathbf{y}_1, \ldots, \mathbf{y}_n \subseteq \mathbb{R}^d$ using a function $f : \mathbb{R}^d \to \mathbb{R}^d$. Error is measured using the square loss, i.e. the objective is $\frac{1}{n}\sum_{i=1}^n \|f(\mathbf{x}_i) - \mathbf{y}_i\|_2^2$. Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be the vertical stack of the inputs, and let $\mathbf{Y} \in \mathbb{R}^{n \times d}$ be the vertical stack of the targets. We first generate $\mathbf{X}$ as a random whitened matrix (i.e. $\frac{1}{n}\mathbf{X}^T\mathbf{X} = \mathbf{I}$). To generate $\mathbf{X}$ as a random whitened matrix, we sample a $n \times d$ matrix of standard Gaussians, and then set $\mathbf{X}$ to be $\sqrt{n}$

times the Q factor in the QR factorization of that matrix. We then generate $\mathbf{Y}$ via $\mathbf{Y} = \mathbf{X}\mathbf{A}^T$, where $\mathbf{A} \in \mathbb{R}^{d \times d}$ is a random matrix whose entries are sampled i.i.d from the standard normal distribution.

We use $n = 50$ datapoints with a dimension of $d = 50$.

Network The function $f : \mathbb{R}^d \to \mathbb{R}^d$ is implemented as a L-layer deep linear network: $f(\mathbf{x}) = \mathbf{W}_L \dots \mathbf{W}_2 \mathbf{W}_1 \mathbf{x}$, with $\mathbf{W}_\ell \in \mathbb{R}^{d \times d}$. We initialize all layers of the deep linear network using Xavier initialization: all entries of each $\mathbf{W}_\ell$ are drawn i.i.d from $\mathcal{N}(0, \frac{1}{d})$.

We use a network with $L = 20$ layers.

Results In Figure 80, we train the network using gradient flow. (Since it is unclear whether the network can be trained to zero loss, and how long this would take, we arbitrarily chose to stop training at time 100.) In Figure 81, we train the network using gradient descent at a range of step sizes. We observe mostly the same Edge of Stability behavior as elsewhere in the paper. The only difference is that in Figure 81, the train loss does not really behave non-monotonically — for each step size η, there is a brief blip at some point, but otherwise, the train loss decreases monotonically.

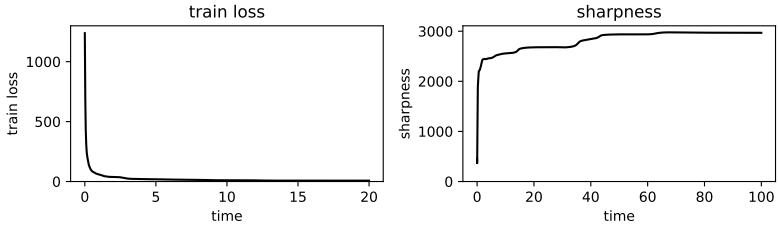

Figure 80: **Training a deep linear network using gradient flow**. We use gradient flow to train a deep linear network. Since it is unclear whether this network can be trained to zero loss (or how long that would take), we arbitrarily chose to stop training at time 100.

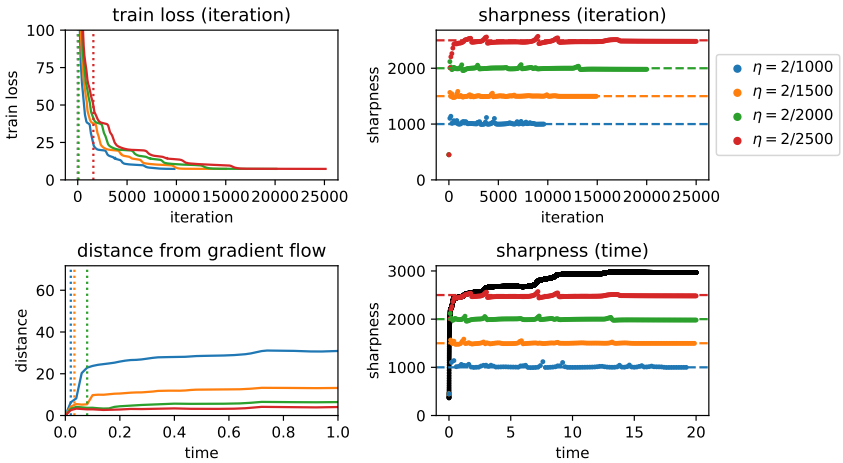

Figure 81: **Training a deep linear network using gradient descent**. We train a deep linear network using gradient descent at a range of step sizes (see legend). (Note that in the top left pane, the blue, orange and green dotted lines are directly on top of one another.)

K EXPERIMENTS: STANDARD ARCHITECTURES ON CIFAR-10

In this appendix, we demonstrate that our findings hold for three standard architectures on the standard dataset CIFAR-10. The three architectures are: a VGG with batch normalization (Figures 82-83), a VGG without batch normalization (Figures 84-85), and a ResNet with batch normalization (Figures 86 -87). See §G.2 for full experimental details.

For each of these three architectures, we confirm that: (1) so long as the sharpness is less than $2/\eta$, the sharpness tends to increase (progressive sharpening); and (2) if the sharpness reaches $2/\eta$, gradient descent enters a regime (the Edge of Stability) in which (a) the training loss behaves non-monotonically, yet consistently decreases over long timescales, and (b) the sharpness hovers right at, or just above, the value $2/\eta$. Moreover, we observe that even though these architectures use the ReLU activation function (which is not continuously differentiable), gradient descent closely tracks the Runge-Kutta trajectory until reaching the point on that trajectory where the sharpness hits $2/\eta$.

Furthermore, for these three standard architectures, we find that (1) progressive sharpening occurs to a dramatic degree, and (2) stable step sizes are so small as to be completely unreasonable.

Progressive sharpening occurs to a dramatic degree To assess the degree of progressive sharpening, we train these networks using Runge-Kutta / gradient flow, which can be viewed as gradient descent with an infinitesimally small step size. (In practice, the Runge-Kutta algorithm does have a step size parameter, and throughout training, we periodically adjust this step size in order to ensure that the algorithm remains stable.) Intuitively, training with gradient flow tell us how far the sharpness would rise "if it didn't have to worry about" instability caused by nonzero step sizes. In Figure 82, we train the VGG with BN to completion (99% training accuracy) using Runge-Kutta / gradient flow, and find that the sharpness rises from its initial value of 6.38 to a peak value of 2140.72. For the other two architectures, progressive sharpening occurs to such a degree that it is not computationally feasible for to train using Runge-Kutta / gradient flow all the way to completion. (The reason is that in regions where the sharpness is high, the Runge-Kutta step size must be made small, so Runge-Kutta requires very many iterations.) Therefore, we instead train these two networks only partially. In Figure 84, for the VGG without BN, we find that the sharpness rises from its initial value of 0.64 to the value 2461.79 at 37.1% accuracy, when we stop training. In Figure 86, for the ResNet, we find that the sharpness rises from its initial value of 1.07 to the value 750.64 at 42% accuracy, when we stop training. Thus, even though we observed in Appendix C that progressive sharpening attenuates as the width of fully-connected networks is made larger, it appears that either: (1) this does not happen for modern families of architectures such as ResNet and VGG, or (2) this does happen for modern families of architectures, but practical network widths lie on the narrow end of the spectrum.

Stable step sizes are so small as to be unreasonable Recall from 3.3 that if $\lambda_{\max}$ is the maximum sharpness along the gradient flow trajectory, then any stable step size must be less than $2/\lambda_{\max}$. Therefore, for these three architectures, because progressive sharpening occurs to a dramatic degree (i.e. $\lambda_{\max}$ is extraordinarily large), any stable step size must be extraordinarily small, which means that training will require many iterations. Yet, at the same time, we find that these three networks can be successfully trained in far fewer iterations by using larger step step sizes, which means that training at a stable step size is extremely suboptimal.

VGG with BN. For this network, gradient flow terminates at time 15.66, and the maximum sharpness along the gradient flow trajectory is 2227.5. Therefore, the largest stable step size is $2/2140.72 = 0.000897$, and training to completion at this step size would take $15.66/0.000897 = 17458$ iterations. Meanwhile, we empirically observe that the network can also be trained to completion at the much larger step size of $\eta = 0.16$ in just 331 iterations. Therefore, using a stable step size is suboptimal by a factor of at least $17458/331 = 52.7$.

For the other two architectures, since we are unable to train *to completion* using gradient flow, we are unable to obtain a tight lower bound for the number of iterations required to run gradient descent to completion at a stable step size. Therefore, by extension, we are unable to compute a tight lower bound for the suboptimality factor of stable step sizes. As a substitute, we will instead compute both: (1) a tight lower bound on the suboptimality of training *partially* at a stable step size, and (2) a very loose lower bound on the suboptimality of training to completion at a stable step size.

VGG without BN. For this network, gradient flow reaches 37% accuracy at time 8, and the maximum sharpness up through this point on the gradient flow trajectory is 2461.8. Therefore, the largest stable step size is $2/2461.8 = 0.00081$, and training to 37% accuracy at this step size would require $8/0.00081 = 9,876$ iterations. Meanwhile, we empirically observe that the network can also be trained to 37% accuracy at the larger step size of $\eta = 0.16$ in just 355 iterations. Therefore, when training this network to 37% accuracy, stable step sizes are suboptimal by a factor of at least $9876/355 = 27.8$. This is a tight lower bound on the suboptimality of training to 37% accuracy at a stable step size.

To obtain a loose lower bound on the suboptimality of training the VGG without BN *to completion* at a stable step size, we note that (a) since the maximum sharpness up through time 8 is 2461.8, the maximum sharpness along the *entire* gradient flow trajectory must be at least 2461.8; and (b) since by time 8 gradient flow has only attained 37% training accuracy, the time to reach 99% accuracy (i.e. completion) must be at least 8. (Note that both of these lower bounds are extremely loose.) Therefore, training this network to completion at a stable step size would require at least $8/(2/2461.8) = 9,876$ iterations (which is the same number of iterations as training to 37% accuracy at a stable step size). Meanwhile, we find that the network can be trained to completion at the larger step size of $\eta = 0.16$ in just 1782 iterations. Therefore, training to completion at a stable step size is suboptimal by a factor of at least $9,876/1782 = 5.54$.

ResNet. For this network, gradient flow reaches 43% accuracy at time 70, and the maximum sharpness up through this point on the gradient flow trajectory is 760.6. Therefore, the largest stable step size is $2/760.6 = 0.0026$, and training to 43% accuracy at this step size would require $70/0.0026 = 26,923$ iterations. Meanwhile, we empirically observe that the network can also be trained to 43% accuracy at the larger step size of $\eta = 2.0$ in just 99 iterations. Therefore, when training this network to 37% accuracy, stable step sizes are suboptimal by a factor of at least $26,923/99 = 271.9$. This is a tight lower bound on the suboptimality of training to 43% accuracy at a stable step size. For the loose lower bound on the suboptimality of training to completion at a stable step size, note that (just as for the VGG-without-BN above), training to completion at a stable step size must require at least $26,923$ iterations. Meanwhile, we find that the network can be trained to completion at the larger step size of $\eta = 2.0$ in just 807 iterations. Therefore, training to completion at a stable step size is suboptimal by a factor of at least $26,923/807 = 33.3$.

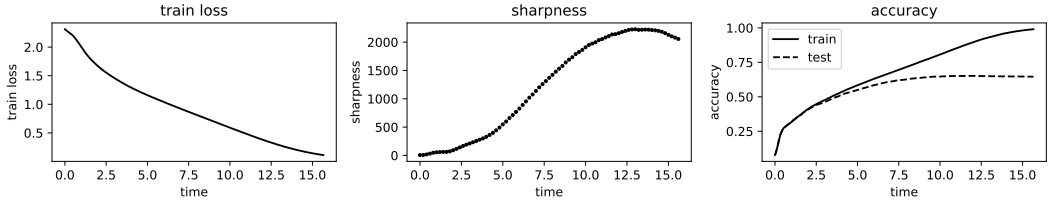

Figure 82: We train a **VGG with BN** to completion using **gradient flow** (Runge-Kutta). Observe that the sharpness rises dramatically from 6.38 at initialization to a peak of 2140.72.

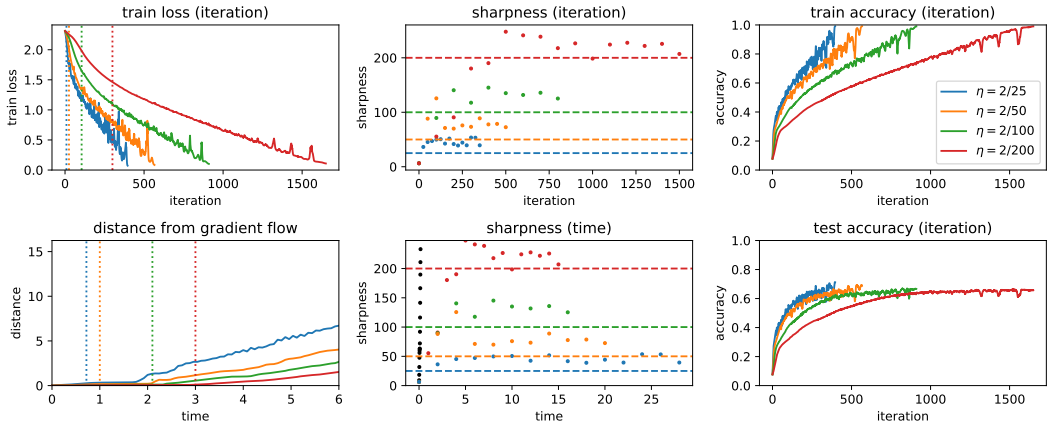

Figure 83: We train a **VGG with BN** to completion using **gradient descent** at different step sizes (see legend in the top right pane). **Top left**: we plot the train loss, with a vertical dotted line marking the iteration where the sharpness first crosses $2/\eta$. **Top center**: we plot the evolution of the sharpness, with a horizontal dashed line (of the appropriate color) marking the value $2/\eta$. **Top right**: we plot the train accuracy. **Bottom left**: for the initial phase of training, we monitor the distance between the gradient descent iterate at iteration t/η, and the gradient flow solution at time t, with a vertical dotted line (of the appropriate color) marking the time when the sharpness crosses $2/\eta$. Observe that the distance between gradient descent and gradient flow is almost zero before this instant, but starts to rise shortly afterwards. **Bottom center**: we plot the sharpness by time (= iteration $\times\eta$) rather than iteration. The black dots are the sharpness of the gradient flow trajectory, which shoots up immediately. **Bottom right**: we plot the test accuracy.

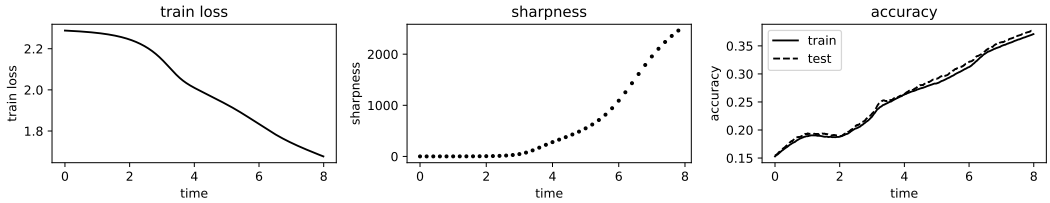

Figure 84: We train a **VGG without BN** to 37% accuracy using **gradient flow** (Runge-Kutta). Observe that the sharpness rises dramatically from 0.64 at initialization to 2461.78 at 37% accuracy. We train this network only partway because training this network to completion would be too computationally expensive: Runge-Kutta runs very slowly when the sharpness is high (because it is forced to take small steps) and for this network the sharpness is extremely high when the train accuracy is only 37% (which means that there is a long way to go).

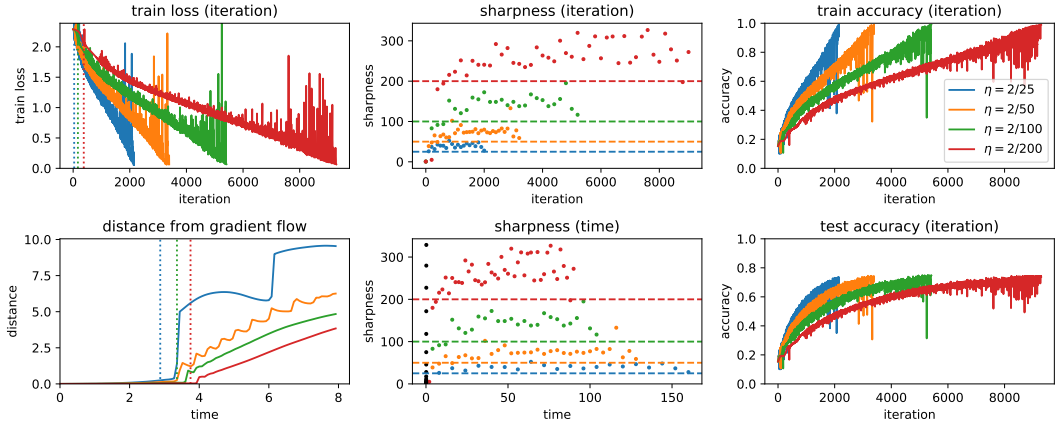

Figure 85: We train a **VGG without BN** to completion using **gradient descent** at a range of step sizes (see legend in the top right pane). Refer to the Figure 83 caption for more information.

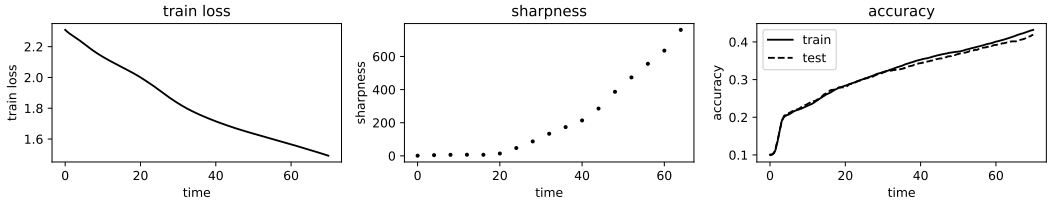

Figure 86: We train a **ResNet** to 42% accuracy using **gradient flow** (Runge-Kutta). Observe that the sharpness rises dramatically from 1.07 at initialization to 760.63 at 42% accuracy. We train this network only partway because training this network to completion would be too computationally expensive: Runge-Kutta runs very slowly when the sharpness is high (because it is forced to take small steps) and for this network the sharpness is extremely high when the train accuracy is only 42% (which means that there is a long way to go).

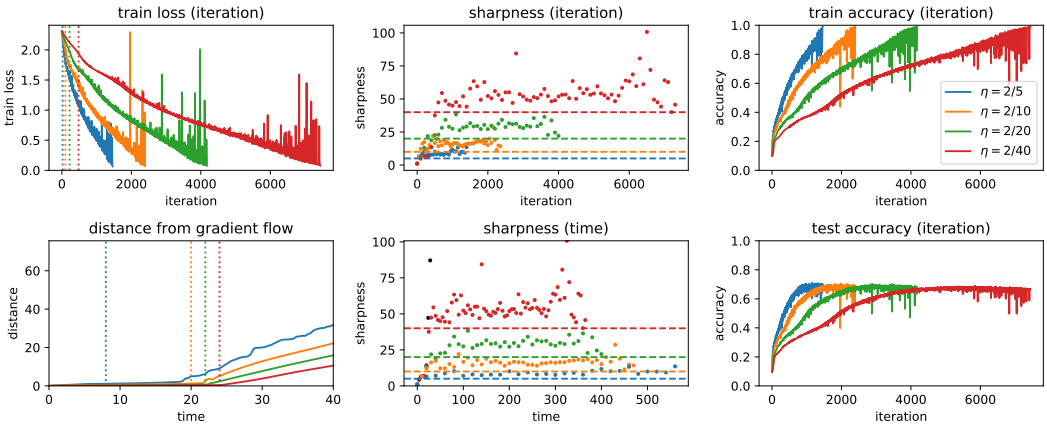

Figure 87: We train a **ResNet** to completion using **gradient descent** at a range of step sizes (see legend in the top right pane). Refer to the Figure 83 caption for more information.

L EXPERIMENTS: MOMENTUM

This appendix contains systematic experiments for gradient descent with Polyak momentum and Nesterov momentum. Our aim is to demonstrate that the sharpness rises until reaching the maximum stable sharpness (MSS) given by equation 1, and then either plateaus just above that value, or oscillates around that value.

We experiment on a 5k-sized subset of CIFAR-10, using four architectures: a tanh fully-connected network (section L.1), a ReLU fully-connected network (section L.2), a tanh convolutional network (section L.3), and a ReLU convolutional network (section L.4). For each of these four architectures, we experiment with both the square loss (for classification) and cross-entropy loss. For each architecture and loss function, we experiment with both Polyak momentum at $\beta = 0.9$, and gradient descent with Nesterov momentum at $\beta = 0.9$. We run gradient descent at a range of several step sizes which were chosen by hand so that the MSS's are approximately spaced evenly. Note that for Polyak momentum with step size η and momentum parameter $\beta = 0.9$, the MSS is $\frac{2+2\beta}{\eta} = \frac{3.8}{\eta}$. For Nesterov momentum with step size η and momentum parameter $\beta = 0.9$, the MSS is $\frac{2+2\beta}{\eta(1+2\beta)} \approx \frac{1.35714}{\eta}$. We run gradient descent until reaching 99% accuracy.

We find that the sharpness rises until reaching the maximum stable sharpness (MSS) given by equation 1, and then either plateaus just above that value, or oscillates around that value. Sometimes these oscillations are rapid (e.g. Figure 89), sometimes they are a bit slower (e.g. Figure 96), and sometimes they are slow (e.g. Figure 102).

L.1 FULLY-CONNECTED TANH NETWORK

In the leftmost plot, the vertical dotted line marks the iteration where the sharpness first crosses the MSS. In the middle plot, the horizontal dashed line marks the MSS.

L.1.1 SQUARE LOSS

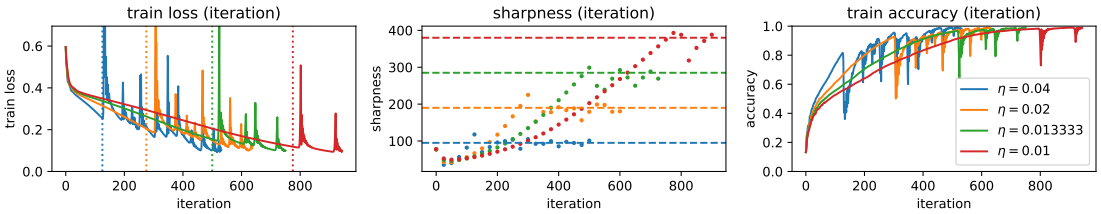

Figure 88: Gradient descent with **Polyak** momentum, $\beta = 0.9$.

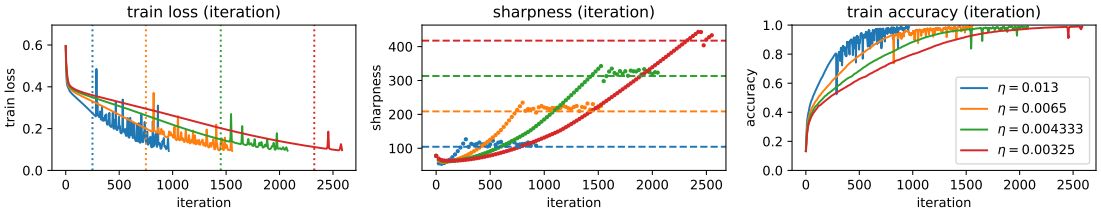

Figure 89: Gradient descent with **Nesterov** momentum, $\beta = 0.9$.

L.1.2 CROSS-ENTROPY LOSS

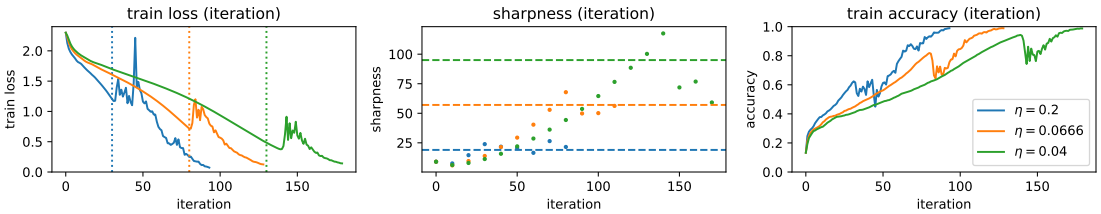

Figure 90: Gradient descent with **Polyak** momentum, $\beta = 0.9$.

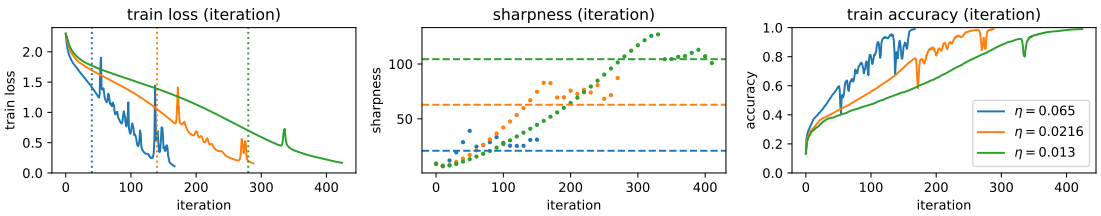

Figure 91: Gradient descent with **Nesterov** momentum, $\beta = 0.9$.

L.2 FULLY-CONNECTED RELU NETWORK

In the leftmost plot, the vertical dotted line marks the iteration where the sharpness first crosses the MSS. In the middle plot, the horizontal dashed line marks the MSS.

L.2.1 SQUARE LOSS

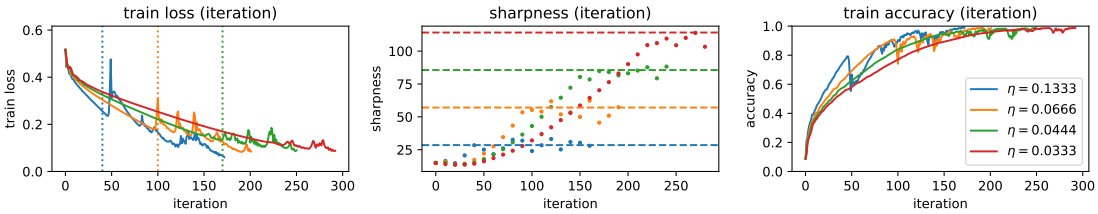

Figure 92: Gradient descent with **Polyak** momentum, $\beta = 0.9$.

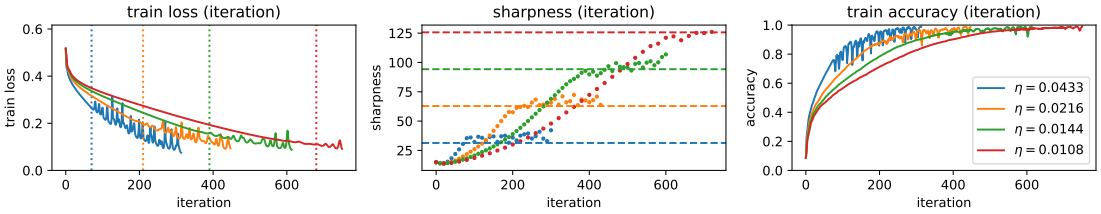

Figure 93: Gradient descent with **Nesterov** momentum, $\beta = 0.9$.

L.2.2 CROSS-ENTROPY LOSS

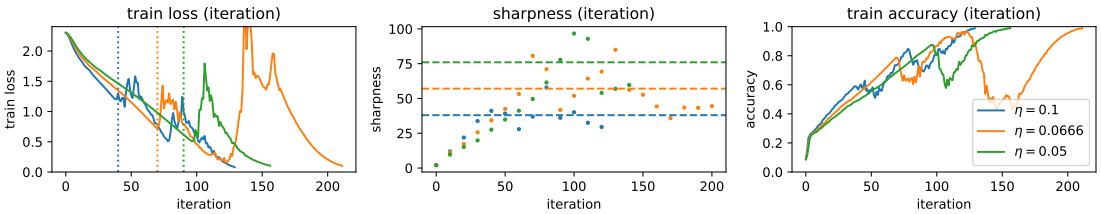

Figure 94: Gradient descent with **Polyak** momentum, $\beta = 0.9$.

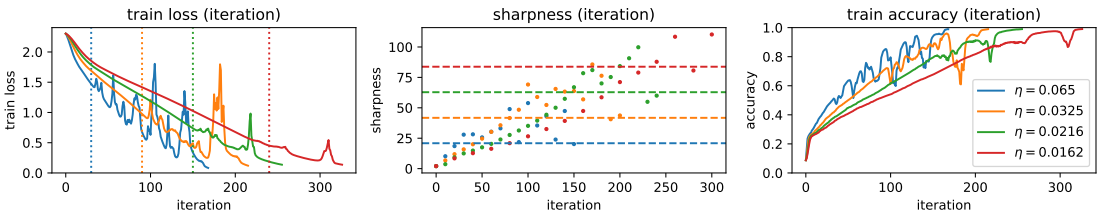

Figure 95: Gradient descent with **Nesterov** momentum, $\beta = 0.9$.

L.3 CONVOLUTIONAL TANH NETWORK

In the leftmost plot, the vertical dotted line marks the iteration where the sharpness first crosses the MSS. In the middle plot, the horizontal dashed line marks the MSS.

L.3.1 SQUARE LOSS

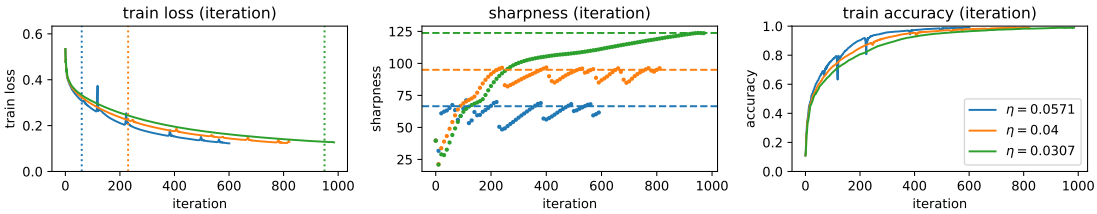

Figure 96: Gradient descent with **Polyak** momentum, $\beta = 0.9$.

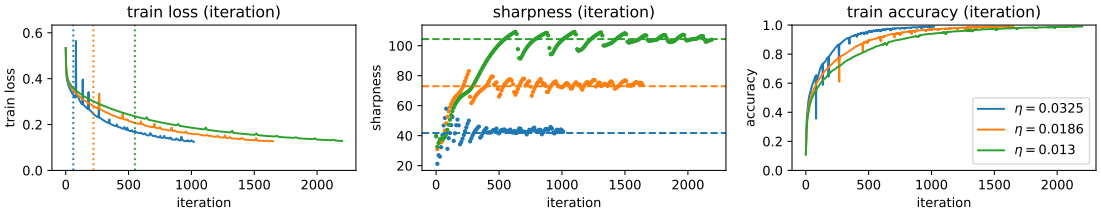

Figure 97: Gradient descent with **Nesterov** momentum, $\beta = 0.9$.

L.3.2 CROSS-ENTROPY LOSS

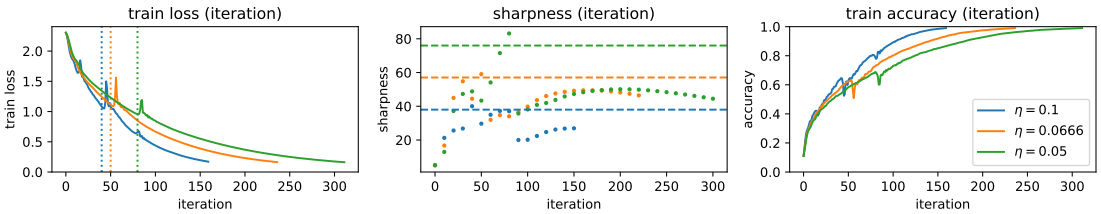

Figure 98: Gradient descent with **Polyak** momentum, $\beta = 0.9$.

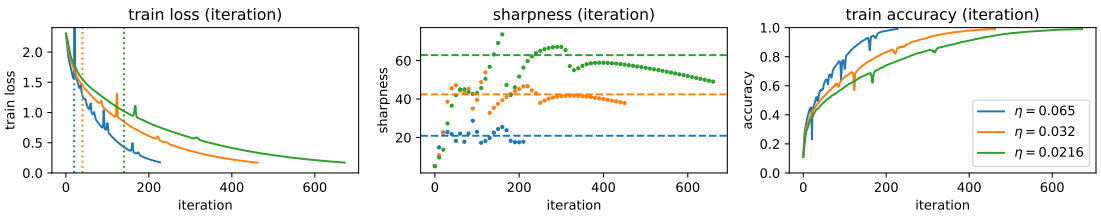

Figure 99: Gradient descent with **Nesterov** momentum, $\beta = 0.9$.

L.4 CONVOLUTIONAL RELU NETWORK

In the leftmost plot, the vertical dotted line marks the iteration where the sharpness first crosses the MSS. In the middle plot, the horizontal dashed line marks the MSS.

L.4.1 SQUARE LOSS

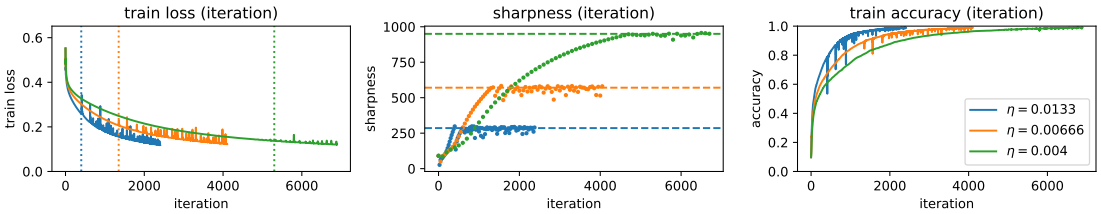

Figure 100: Gradient descent with **Polyak** momentum, $\beta = 0.9$.

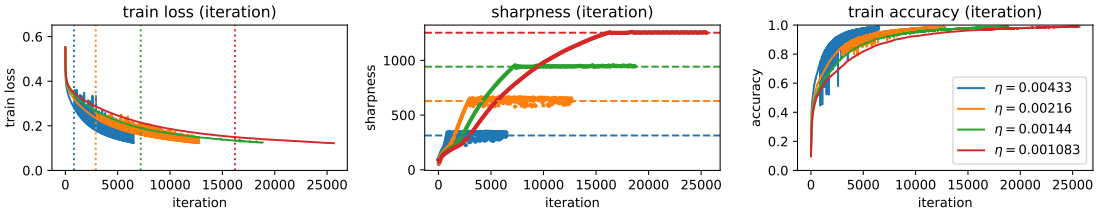

Figure 101: Gradient descent with **Nesterov** momentum, $\beta = 0.9$.

L.4.2 CROSS-ENTROPY LOSS

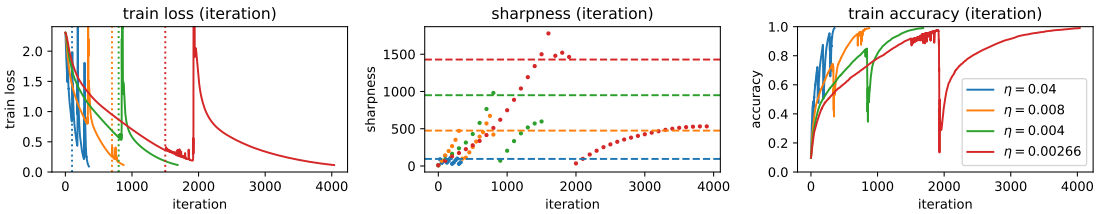

Figure 102: Gradient descent with **Polyak** momentum, $\beta = 0.9$.

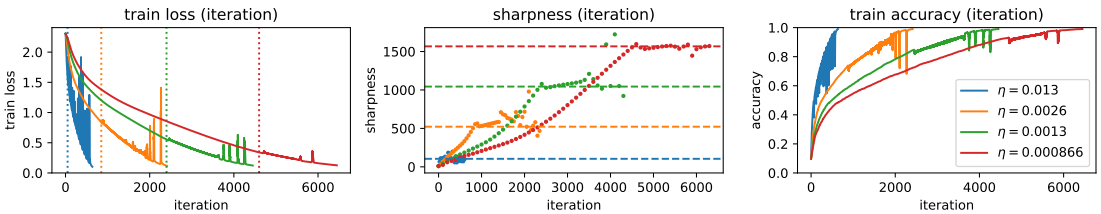

Figure 103: Gradient descent with **Nesterov** momentum, $\beta = 0.9$.

M EXPERIMENTS: LEARNING RATE DROP

In this appendix, we run gradient descent until reaching the Edge of Stability, and then we cut the step size. We will see that the sharpness starts increasing as soon as the step size is cut, and only stops increasing once gradient descent is back at the Edge of Stability (or training is finished). As a consequence, one can interpret the Edge of Stability as a regime in which gradient descent is constantly "trying" to increase the sharpness beyond $2/\eta$, but is constantly being blocked from doing so. Our experiments focus on image classification on a 5k-sized subset of CIFAR-10. We study two architectures (a fully-connected tanh network and a convolutional ReLU network) and two loss functions (squared loss and cross-entropy loss).

M.1 FULLY-CONNECTED TANH NETWORK: SQUARE LOSS

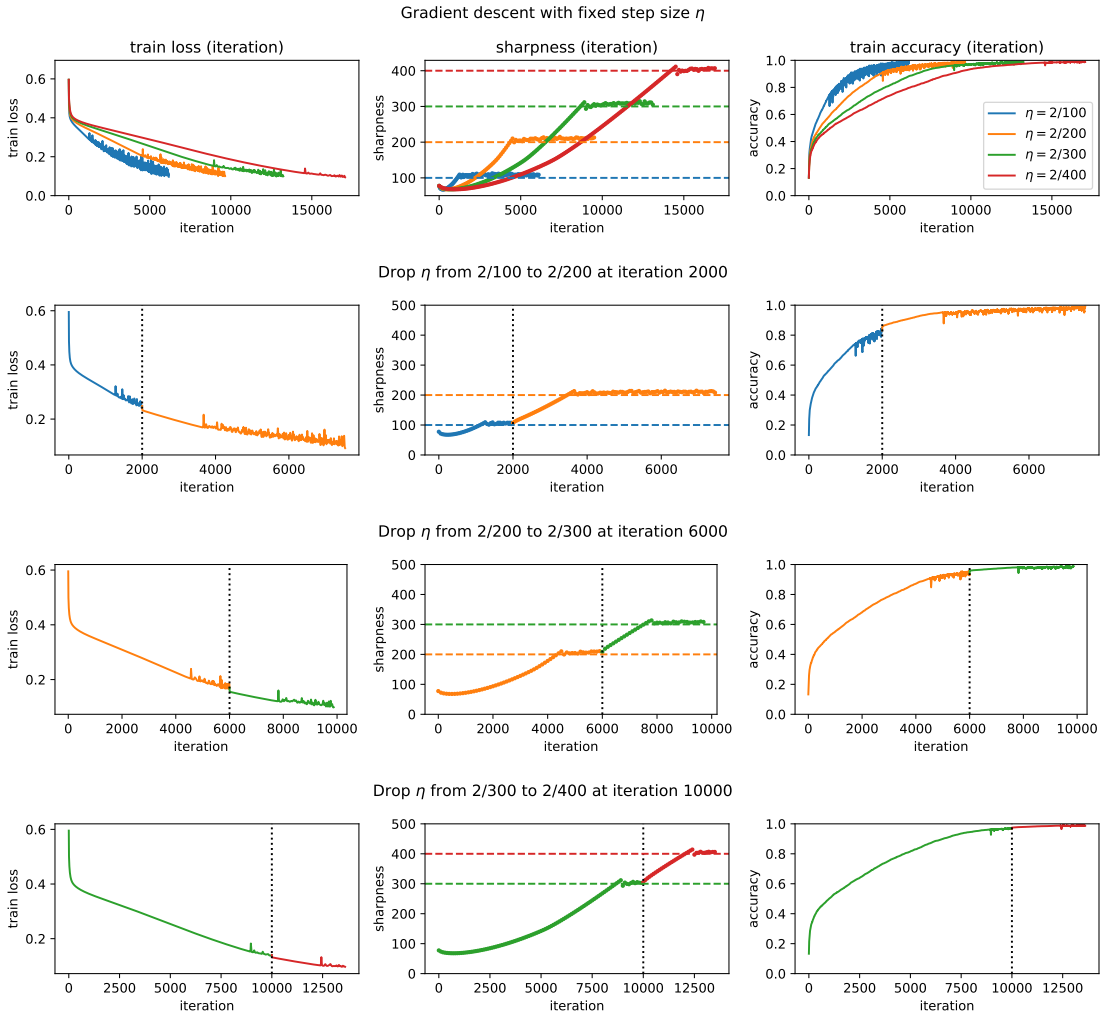

Figure 104: **Top row**: we train a network using gradient descent at a range of step sizes η (see the legend in the right pane). **Bottom three rows**: Once gradient descent has reached the Edge of Stability, we cut the step size. The black vertical dotted line marks the iteration where the step size is cut. This iteration was chosen by hand, but was not cherry-picked. We use different colors (consistent with the legend) to plot the train loss, sharpness, and train accuracy before and after the learning rate drop. In the middle sharpness plot, the two horizontal lines mark the maximum stable sharpness $2/\eta$ for both the old and new step size. **Takeaway**: once we decrease the step size, the sharpness immediately starts to increase until it reaches the maximum stable sharpness $2/\eta$ for the new step size η (or training finishes).

M.2 FULLY-CONNECTED TANH NETWORK: CROSS-ENTROPY LOSS

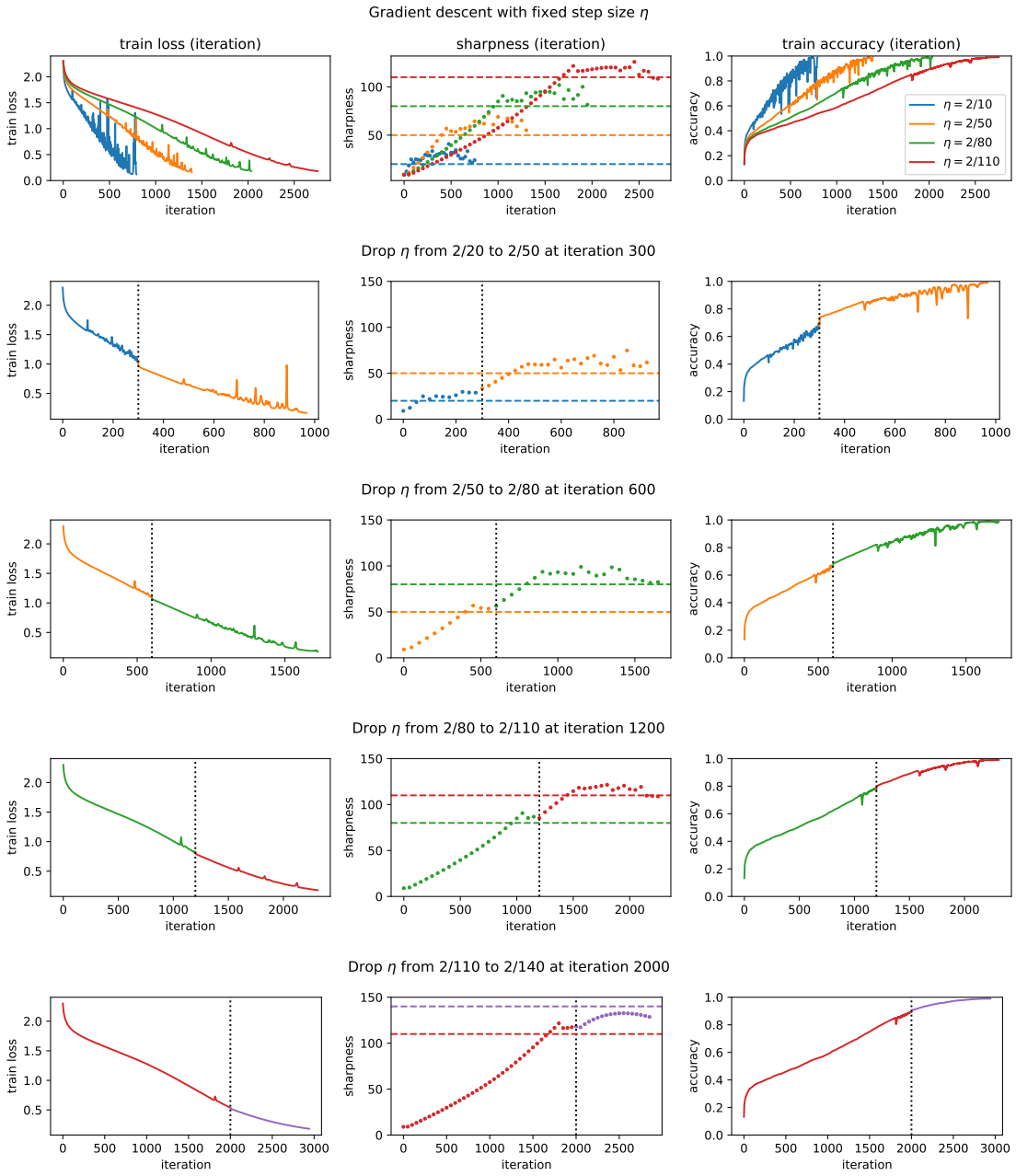

Figure 105: **Top row**: we train a network using gradient descent at a range of step sizes η (see the legend in the right pane). **Bottom four rows**: Once gradient descent has reached the Edge of Stability, we cut the step size. The black vertical dotted line marks the iteration where the step size is cut. This iteration was chosen by hand, but was not cherry-picked. We use different colors (consistent with the legend) to plot the train loss, sharpness, and train accuracy before and after the learning rate drop. In the middle sharpness plot, the two horizontal lines mark the maximum stable sharpness $2/\eta$ for both the old and new step size. **Takeaway**: once we decrease the step size, the sharpness immediately starts to increase until it reaches the maximum stable sharpness $2/\eta$ for the new step size η (or training finishes).

M.3 CONVOLUTIONAL RELU NETWORK: SQUARE LOSS

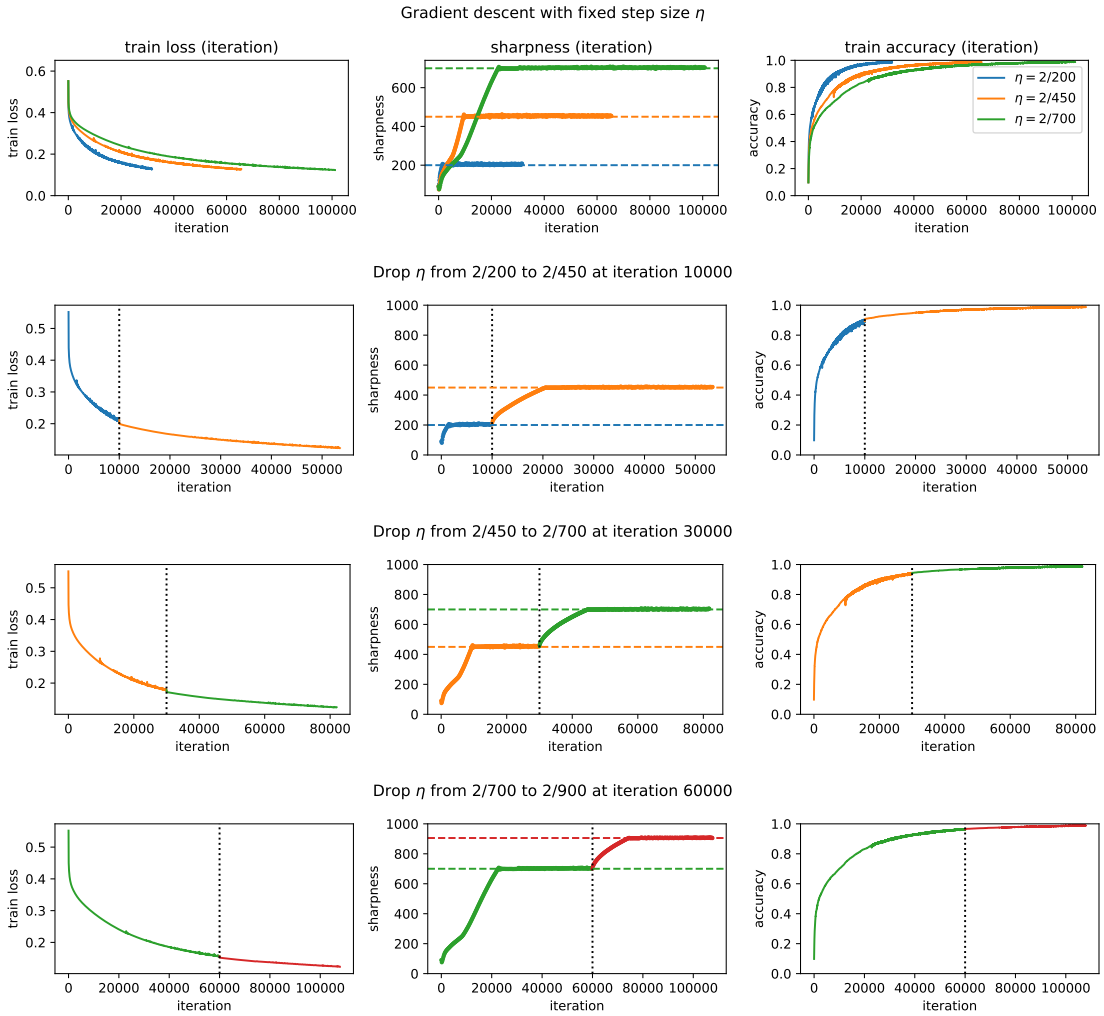

Figure 106: **Top row**: we train a network using gradient descent at a range of step sizes η (see the legend in the right pane). **Bottom three rows**: Once gradient descent has reached the Edge of Stability, we cut the step size. The black vertical dotted line marks the iteration where the step size is cut. This iteration was chosen by hand, but was not cherry-picked. We use different colors (consistent with the legend) to plot the train loss, sharpness, and train accuracy before and after the learning rate drop. In the middle sharpness plot, the two horizontal lines mark the maximum stable sharpness $2/\eta$ for both the old and new step size. **Takeaway**: once we decrease the step size, the sharpness immediately starts to increase until it reaches the maximum stable sharpness $2/\eta$ for the new step size η (or training finishes).

M.4 CONVOLUTIONAL RELU NETWORK: CROSS-ENTROPY LOSS

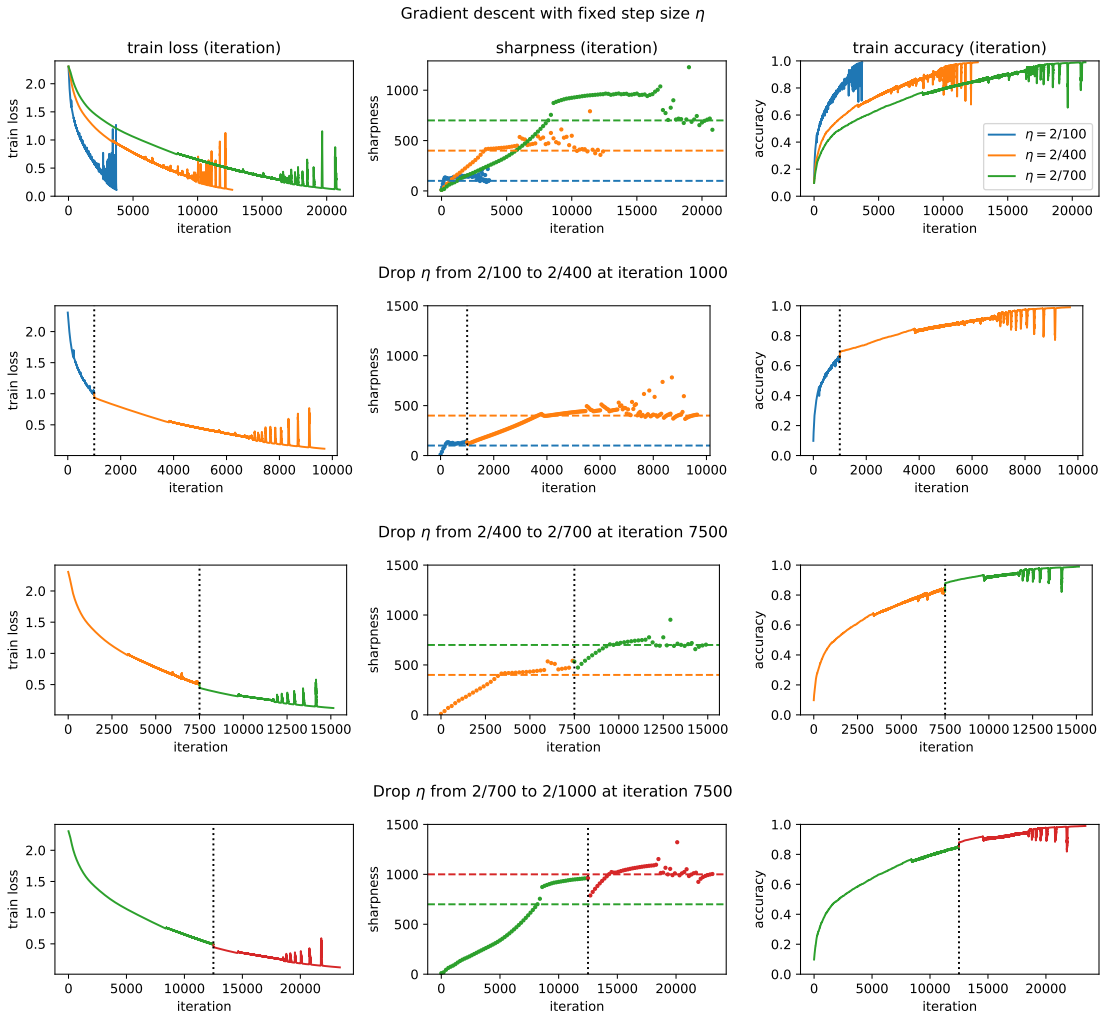

Figure 107: **Top row**: we train a network using gradient descent at a range of step sizes η (see the legend in the right pane). **Bottom three rows**: Once gradient descent has reached the Edge of Stability, we cut the step size. The black vertical dotted line marks the iteration where the step size is cut. This iteration was chosen by hand, but was not cherry-picked. We use different colors (consistent with the legend) to plot the train loss, sharpness, and train accuracy before and after the learning rate drop. In the middle sharpness plot, the two horizontal lines mark the maximum stable sharpness $2/\eta$ for both the old and new step size. **Takeaway**: once we decrease the step size, the sharpness immediately starts to increase until it reaches the maximum stable sharpness $2/\eta$ for the new step size η (or training finishes).