OPT2020: 12th Annual Workshop on Optimization for Machine Learning

GD on Neural Networks Typically Occurs at the Edge of Stability

Jeremy M. Cohen JEREMYCOHEN@CMU.EDU
Simran Kaur SKAUR@ANDREW.CMU.EDU
Yuanzhi Li YUANZHIL@ANDREW.CMU.EDU
Zico Kolter ZKOLTER@CS.CMU.EDU
Ameet Talwalkar ATALWALK@ANDREW.CMU.EDU
Carnegie Mellon University, Pittsburgh PA

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 leading eigenvalue of the training loss Hessian hovers just above the value $2/$(step size), and the training loss behaves non-monotonically, yet consistently decreases over long timescales. Our results expose a large gap between theory and practice in non-convex optimization: many analyses of gradient descent (those based on L-smoothness or monotonic descent) do not apply at the Edge of Stability.

1. Introduction

In this paper, we train neural networks using full-batch gradient descent, and find that gradient descent (GD) typically operates in a regime that is not well-explained by current optimization theory. We study the evolution during training of the *sharpness* — the leading eigenvalue of the training objective's Hessian. The sharpness play a major role in the dynamics of GD with step size η: if the sharpness is less than $2/\eta$, then the descent lemma [6] guarantees that a gradient step will decrease the training objective. On the other hand, if the sharpness is greater than $2/\eta$, then (barring other assumptions) a gradient step might increase the training objective, and, even worse, GD is (to quadratic order) an unstable algorithm: if run on the quadratic Taylor approximation, it would diverge. Consequently, analyses of GD with a fixed step size η frequently stipulate that the sharpness should be less than $2/\eta$, if not globally then at least along the optimization trajectory [5, 28].

We will show that, while there do exist step sizes η for which the sharpness will remain less than $2/\eta$ for the entire duration of training, these step sizes are so small that they are, at best, highly suboptimal in terms of convergence speed, and at worst, completely unfeasible for training. The reason why these step sizes are so small is that gradient descent on neural network training objectives seems to have an overwhelming tendency to continually increase the sharpness. Therefore, even if the initial sharpness is less than $2/\eta$, the sharpness often rises past $2/\eta$ later in training. As one would expect, this causes gradient descent to become destabilized. What is unexpected is that after becoming destabilized, gradient descent transitions into a regime we call the Edge of Stability in which it continues to successfully optimize the loss, but in a fashion that is inconsistent with much of optimization theory. At the Edge of Stability, the sharpness hovers right at (or just above) the value $2/\eta$, meaning that monotone descent is not guaranteed. Moreover, monotone descent does not occur: at the Edge of Stability, the training loss usually behaves non-monotonically over short timescales. Yet despite this, optimization is successful: over long timescales, the training loss consistently decreases. We think that understanding how gradient descent can succeed at the Edge of Stability should be viewed as an important open problem in optimization theory.

2. Sharpening and instability

We first illustrate our main points on a running example; the following section will show that they hold more generally. We consider training a network on a subset of CIFAR-10 using full-batch gradient descent. The network is a fully-connected architecture with two hidden layers of 200 units each (656,810 total parameters), and with ELU activations [8]. We train on a subset of CIFAR-10 of size 5,000. We use the square loss for classification [16], encoding the target class with 1 and the other classes with 0 (our findings also hold for cross-entropy loss, but with one minor twist). We consider training finished when the loss reaches 0.05.

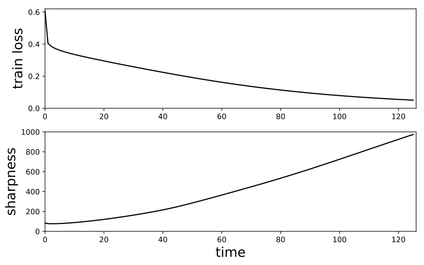

Figure 1: Evolution of train loss (top) and sharpness (bottom) during gradient flow.

To start, we "train" this network using gradient flow, numerically integrating the ODE using the Runge-Kutta RK4 algorithm [37]. Figure 1 plots the train loss (top) and sharpness (bottom) as a function of time. Crucially, observe that the sharpness continually rises over time. It seems to be a general property of gradient flow on neural network training objectives that the sharpness has an overwhelming tendency to continually increase. We call this phenomenon *progressive sharpening*. By "overwhelming tendency," we mean that gradient flow does occasionally decrease the sharpness (e.g. Figure 22), but these brief decreases always seem be followed by a return to continual increase. We do not know why progressive sharpening occurs; this is an important question for future work. In Appendix D we take a step towards reconciling progressive sharpening with NTK theory.

Gradient descent with step size η is a forward Euler discretization of the gradient flow ODE. Consequently, at sufficiently small step sizes, gradient descent roughly tracks the trajectory of gradient flow, traveling at a speed proportional to the step size. However, while gradient *flow* can traverse arbitrarily sharp regions of the loss landscape without difficulty, gradient *descent* is, to quadratic order, unstable in regions of the loss landscape where the sharpness exceeds $2/\eta$. Namely, suppose that at some point θ_0, we were to switch from running gradient descent on the real objective to running gradient descent on the quadratic Taylor approximation around θ_0. Let Δ_t be the displacement from θ_0 along the leading Hessian eigenvector and let λ be the sharpness. Then Δ_t would evolve under gradient descent as $\Delta_t = c\left[(1-\eta\lambda)^t - 1\right]$, where c is a constant (Appendix C). This sequence diverges iff $|1-\eta\lambda| > 1$, which, since $\lambda, \eta > 0$, means $\lambda > 2/\eta$. Thus, if the sharpness exceeds $2/\eta$, then gradient descent run on the quadratic Taylor approximation would oscillate with increasing magnitude along the direction of highest curvature, leading to eventual divergence.

For sufficiently small step sizes, gradient descent will stably track the gradient flow trajectory for the entire duration of training. In particular, let $\lambda_{\max}$ be the maximum sharpness along the gradient flow trajectory (and assume that $\lambda_{\max}$ is finite). If we train at a step size smaller than $2/\lambda_{\max}$, then we can roughly expect (and will verify) that: (1) gradient descent will approximately track the gradient flow trajectory from start to finish, (2) the sharpness along the gradient descent trajectory will remain less than $2/\eta$, and (3) the training loss will decrease monotonically.

What is surprising is that this network can also be successfully trained at step sizes much larger than $2/\lambda_{\max}$. At such step sizes, gradient descent roughly tracks the gradient flow trajectory *at first*; however, once the sharpness reaches $2/\eta$, gradient descent departs the gradient flow trajectory

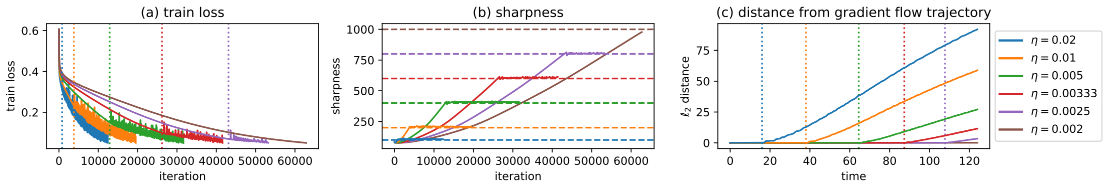

Figure 2: **Gradient descent tracks the gradient flow trajectory until the sharpness reaches $2/\eta$, and then enters the Edge of Stability.**

and enters a regime we call the Edge of Stability in which (1) the sharpness ceases to increase further and instead hovers right at (or just above) the value $2/\eta$, and (2) the train loss behaves non-monotonically. In Figure 2, we run gradient descent at a range of step sizes (see the legend on the right), plotting the train loss curves in 2(a). We draw a vertical dotted line at the iteration where the sharpness first reaches $2/\eta$; observe that the train loss decreases monotonically before this event, but behaves non-monotonically afterwards. In Figure 2(b), we plot the evolution of the sharpness, drawing a horizontal dashed line at the value $2/\eta$. Observe that once the sharpness rises to $2/\eta$, it stops increasing and remains approximately there for the duration of training. In Figure 2(c), we plot the ℓ_2 distance between the gradient flow trajectory and gradient descent trajectory. That is, for each time t, we compute the ℓ_2 distance between the gradient flow iterate at time t and the gradient descent iterate at step t/η. We draw a vertical dotted line at the time where the sharpness first crosses $2/\eta$. Observe that before this event, the distance between the gradient descent and gradient flow trajectories is essentially zero (because GD is roughly following the gradient flow trajectory), yet after this point, the difference starts to grow (because GD departs the gradient flow trajectory).

Observe from Figure 2(a) that gradient descent converges much faster at step sizes greater than $2/\lambda_{\max}$, where it eventually enters the Edge of Stability, than at step sizes less than $2/\lambda_{\max}$ (e.g. the brown curve). Step sizes less than $2/\lambda_{\max}$ are unreasonably small. We believe that the Edge of Stability should be viewed as the "rule" for gradient descent on neural networks, not the "exception."

At the Edge of Stability, gradient descent is constantly "trying" to increase the sharpness, but is being somehow restrained from doing so. If we cut the learning rate, this restraint is removed, and gradient descent is freed to follow gradient flow into a region of increased sharpness. To demonstrate this, in Figure 3, we run gradient descent until it enters the Edge of Stability, then halve the learning rate and continue training. Observe that as soon as the learning rate is cut, the sharpness starts to increase. It only stops increasing once gradient descent is back at the Edge of Stability.

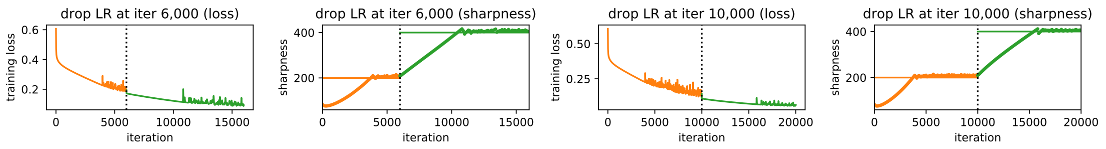

Figure 3: **After a learning rate drop, progressive sharpening resumes.** We train the network with step size 0.01 (orange), for either 6k iterations (left two panes) or 10k iterations (right two panes); then we cut the learning rate to 0.005 (green) and train for longer.

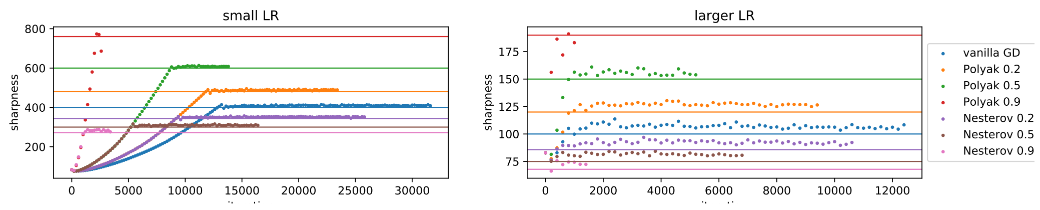

Figure 4: **Momentum gradient descent operates at the Edge of Stability.** We train the network to completion with fixed step size (left: $\eta = 0.005$; right: $\eta = 0.02$) but varying amounts β of either Polyak or Nesterov momentum (see legend). For each algorithm, the horizontal solid line of the appropriate color marks the maximum stable sharpness given by (1).

Momentum Similar to vanilla gradient descent, when run on a quadratic Taylor approximation, gradient descent with either Polyak or Nesterov momentum acts independently along each Hessian eigenvector, and will diverge along the leading eigenvector if and only if the sharpness exceeds a certain threshold (the "maximum stable sharpness", or MSS). We prove in Appendix C that for gradient descent with learning rate η and momentum parameter β, the MSS is:

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

The Polyak result previously appeared in Goh [13]; the Nesterov one seems to be new. It is interesting to note that $\text{MSS}_{\text{Polyak}}(\eta, \beta)$ is increasing in β, whereas $\text{MSS}_{\text{Nesterov}}(\eta, \beta)$ is decreasing in β. That is, adding Polyak momentum gives gradient descent a higher tolerance for sharpness, whereas adding Nesterov momentum gives gradient descent a lower tolerance for sharpness. In Figure 4, we train our network to completion with varying amounts β of both Polyak and Nesterov momentum, while keeping the learning rate η held fixed. Just like in the vanilla GD case, we see that the sharpness rises to the maximum stable sharpness and then remains approximately there.

Cross-entropy loss So far, we have focused on the squared loss. The situation is mostly the same for the commonly-used cross-entropy loss, but with one difference: for cross-entropy loss, at the end of training, we often observe that the sharpness decreases. This behavior seems to be caused by the well-known fact that at the end of training, gradient descent continually scales up the logits so as to drive the cross-entropy loss to zero [43]. When this occurs, the sharpness drops, because the second derivative of the cross-entropy goes to zero as its input increases.

3. Experiments

In Appendix A, we demonstrate that our main points hold across many different settings. Namely, we show that if the initial sharpness is less than $2/\eta$, then the sharpness will tend to continually rise during training (progressive sharpening) until gradient descent reaches the Edge of Stability, a regime where the sharpness hovers right at (or just above) $2/\eta$, and the training loss behaves non-monotonically. We demonstrate this point across different architectures (fully-connected, CNN, VGG, Transformer, deep linear network), activation functions (ReLU and tanh), loss functions (squared loss and cross-entropy), datasets (CIFAR-10, a synthetic regression task, language modeling), and on networks both with and without Batch Normalization.

4. Related work

Several previous works have measured the evolution of the sharpness (and, more generally, the Hessian spectrum) during neural network training [11, 20, 21, 23, 27, 33–35, 40]. Xing et al. [49] observed that full-batch gradient descent eventually enters a regime where the training loss behaves non-monotonically, but did not describe when/why this occurs, and did not discuss the implications of this observation for optimization theory. Prior works have argued that the stability properties of optimization algorithms could serve as a form of implicit bias in deep learning [12, 22, 30, 31, 47]

5. Conclusion

We have demonstrated that, at all but the smallest step sizes, gradient descent eventually transitions in a regime — the Edge of Stability — where the sharpness hovers right at (or just above) the value $2/\eta$, and the training loss behaves non-monotonically. From the point of view of conventional optimization theory, it is surprising that gradient descent can succeed in this regime, considering that in this regime (1) the descent lemma fails to hold, and (2) gradient descent is, to quadratic order, unstable (meaning that it would diverge if run on the quadratic Taylor approximation). We hope that this paper will inspire research aimed at understanding the Edge of Stability.

It is currently popular for authors motivated by deep learning to prove convergence results for optimization algorithms under the "non-convex but L-smooth" setting [1, 7, 9, 26, 29, 38, 39, 41, 45, 46, 48, 50, 51, 53]. For gradient descent, analyses based on L-smoothness require (at a minimum) that the sharpness should be less than $2/\eta$ along the optimization trajectory. Therefore, our results imply that for neural network training, analyses based on L-smoothness cannot explain the successful convergence of vanilla gradient descent at any reasonable step size. We think that authors may want to reconsider the suitability of the "non-convex but L-smooth" setting in light of this finding.

Even analyses of gradient descent that do not explicitly assume L-smoothness often assert that gradient descent will decrease the training objective monotonically [3, 52]. However, at the Edge of Stability, the training loss usually behaves non-monotonically over short timescales even as it consistently decreases over long timescales. Thus, any analysis that aims to prove that gradient descent will monotonically decrease the training loss is doomed to be unrealistic in practice.

While we focused in this paper on full-batch gradient descent, we believe that some of our findings may also have analogues for SGD. First, while it is obviously normal for the training loss to behave non-monotonically during SGD, we present a surprising experiment in Appendix B which suggests that SGD on neural networks rapidly "acclimates" to the learning rate and batch size in such a way that subsequent SGD steps do not even consistently decrease the training loss *in expectation*, even though SGD steps with a smaller learning rate or larger batch size *would* consistently decrease the loss in expectation. We view this experiment as evidence that SGD on neural networks may behave more similar to gradient descent at the Edge of Stability than to classical analyses of SGD as in Bottou et al. [5]. Second, much like how the learning rate in gradient descent implicitly constrains the sharpness along the optimization trajectory, the experiments in Jastrzebski et al. [22] indicate that a similar phenomenon may be at play during SGD, though SGD seems to implicitly constrain not just the sharpness but also the variance of the stochastic gradients. For SGD, we currently lack a precise quantitative understanding of this implicit bias. A fascinating open problem is whether SGD satisfies some "steady state" condition analogous to the sharpness $\approx 2/\eta$ rule that we have identified for full-batch gradient descent.

References

[1] 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*, pages 1195–1199, 2017.

[2] Guillaume Alain, Nicolas Le Roux, and Pierre-Antoine Manzagol. Negative eigenvalues of the hessian in deep neural networks, 2019.

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

[4] Francis Bach. Effortless optimization through gradient flows, 2020. URL https://francisbach.com/gradient-flows/.

[5] 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.

[6] Sébastien Bubeck. Convex optimization: Algorithms and complexity, 2014.

[7] 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.

[8] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus), 2016.

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

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

[11] Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. An investigation into neural net optimization via hessian eigenvalue density. volume 97 of *Proceedings of Machine Learning Research*, pages 2232–2241, 2019.

[12] 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.

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

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

[15] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pages 1731–1741, 2017.

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

[17] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. pages 448–456, 2015.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[25] 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*, pages 8572–8583, 2019.

[26] 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*, pages 983–992, 2019.

[27] Xinyan Li, Qilong Gu, Yingxue Zhou, Tiancong Chen, and Arindam Banerjee. Hessian based analysis of sgd for deep nets: Dynamics and generalization. In *Proceedings of the 2020 SIAM International Conference on Data Mining*, pages 190–198. SIAM, 2020.

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

[29] 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.

[30] Rotem Mulayoff and Tomer Michaeli. Unique properties of flat minima in deep networks, 2020.

[31] Kamil Nar and Shankar Sastry. Step size matters in deep learning. In *Advances in Neural Information Processing Systems*, pages 3436–3444, 2018.

[32] 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, pages 543–547, 1983.

[33] Vardan Papyan. The full spectrum of deepnet hessians at scale: Dynamics with sgd training and sample size. *arXiv preprint arXiv:1811.07062*, 2018.

[34] Vardan Papyan. Measurements of three-level hierarchical structure in the outliers in the spectrum of deepnet hessians. *arXiv preprint arXiv:1901.08244*, 2019.

[35] Vardan Papyan. Traces of class/cross-class structure pervade deep learning spectra, 2020.

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

[37] 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.

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

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

[40] Levent Sagun, Utku Evci, V Ugur Guney, Yann Dauphin, and Leon Bottou. Empirical analysis of the hessian of over-parametrized neural networks. *arXiv preprint arXiv:1706.04454*, 2017.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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*, pages 3732–3745, 2019.

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

[47] 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*, pages 8279–8288, 2018.

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

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

[50] 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.

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

[52] 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.

[53] 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.

Appendix A. Experiments

We now demonstrate our main point across different architectures, activation functions, loss functions, and tasks, and on networks both with and without batch normalization. Across all of these training setups, we will show that if the initial sharpness is less than $2/\eta$, then the sharpness will tend to continually rise during training (progressive sharpening) until gradient descent reaches the Edge of Stability, a regime where the sharpness hovers right at (or just above) $2/\eta$ and the training loss behaves non-monotonically. Training loss curves (and accuracies) are plotted in Appendix G.

In the first row of Figure 5, we demonstrate that our findings hold across several different architectures — a ReLU CNN, a tanh CNN, and a ReLU fully-connected network — trained using the square loss on a subset of 5k examples from CIFAR-10. In the second row, we demonstrate that our findings hold for the more commonly-used cross-entropy loss, the only difference being that

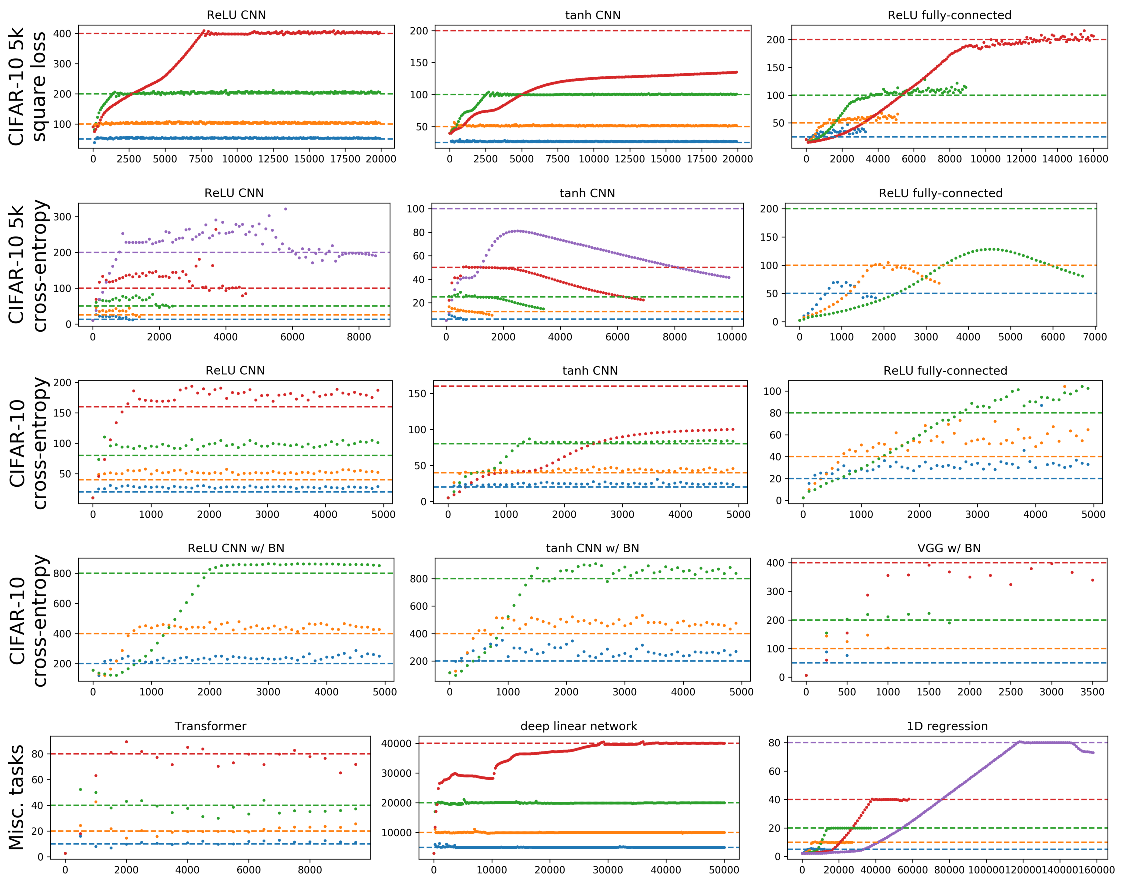

Figure 5: **Our findings hold across a diverse range of settings.** We train networks using gradient descent and measure the sharpness at regular intervals during training. Each color is a different learning rate η. The horizontal dashed line marks the value $2/\eta$. See Appendix G for plots of the train loss and train/test accuracies and additional commentary.

the sharpness starts to decrease towards the end of training. (Here we cease training once the loss reaches 0.05, but if training were allowed to run longer, the sharpness would continue to drop.) In the third row, we verify that our findings hold on the full CIFAR-10 dataset, using three small-ish architectures (where we don't train to completion). Figure **??** shows the same point for the larger, realistic VGG-11 architecture (where we do). In the experiments on full CIFAR-10, we compute the sharpness on a subsampled set of 5k datapoints.

Batch normalization [17] is known to have unusual optimization properties [28], so it is natural to wonder whether batch-normalized (BN) networks still operate at the Edge of Stability. The fourth row of Figure 5 demonstrates that the sharpness in BN networks behaves exactly the same under gradient descent as it does in non-BN networks. (As is standard practice, we use ghost batch normalization [15] to make full-batch gradient descent feasible with limited GPU memory.) Appendix E reconciles this finding with Santurkar et al. [42].

Finally, in the fifth row of Figure 5, we demonstrate that our results hold on settings beyond image classification: a Transformer trained on the WikiText-2 language modeling task; a deep linear network trained on synthetic isotropic data; and a one-hidden-layer tanh network trained on a toy one-dimensional regression problem.

Appendix B. Stochastic Gradient Descent

We have demonstrated that during full-batch gradient descent with step size η, the sharpness rises until reaching the maximum stable sharpness $2/\eta$, where it approximately remains for the duration of training. Unfortunately, prior work has shown that the evolution of the sharpness during SGD is not as easily characterized. On the one hand, just like with gradient descent, the learning rate seems to affect the sharpness along the optimization trajectory, with larger learning rates (as well as smaller batch sizes) steering SGD towards less sharp regions of the loss landscape [21, 22]. On the other hand, in behavior different from gradient descent, during SGD the sharpness sometimes does not equilibrate at any steady-state value, much less one that we can numerically predict; for example, in Figure 5(a) of Jastrzebski et al. [22], one can see that the sharpness gradually drifts downwards during SGD training. We suspect that this is partly because the sharpness does not directly affect the stability of SGD unless one makes the strong assumption that the Hessian and the covariance matrix of per-example gradients share a leading eigenvector, as in Jastrzebski et al. [22].

That said, we do suspect that there may exist some sense in which SGD, too, operates at the Edge of Stability. In particular, we now demonstrate that SGD seems to "acclimate" to the learning rate and batch size in such a way that an actual step sometimes increases and sometimes decreases the loss in expectation, yet a step with a larger learning rate or smaller batch size would almost always *increase* the loss in expectation, and a step with a smaller learning rate or a larger batch size would almost always *decrease* the loss in expectation.

In Figure 6, we train the ELU network from Section 2 using SGD with learning rate 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 6. 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 6, 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 7(a), during training of the same network, we compute the expected change in training loss that would result from taking an SGD step with the same learning rate used during training, 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 7(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 7(c) and (d) repeat this experiment with the learning rate; we observe that an SGD step with a larger learning rate would consistently increase the training loss in expectation, while an SGD step with a smaller learning rate would consistently decrease the training loss in expectation.

To be clear, there was nothing special in absolute terms about the hyperparameters we used. In Figure 8(a), we train the same network with learning rate 0.01 and batch size 16 and evaluate the expected loss change during training. Observe that some steps increase the loss in expectation, while others decrease the loss in expectation. Figure 8(b)-(d) repeats this experiment with: (b) learning rate 0.01 and batch size 64, (c) learning rate 0.02 and batch size 32, and (d) learning rate 0.005 and batch size 32.

We view this experiment as evidence that SGD on neural networks behaves more similar to gradient descent at the Edge of Stability, than to conventional analyses of SGD like the ones in Bottou et al. [5].

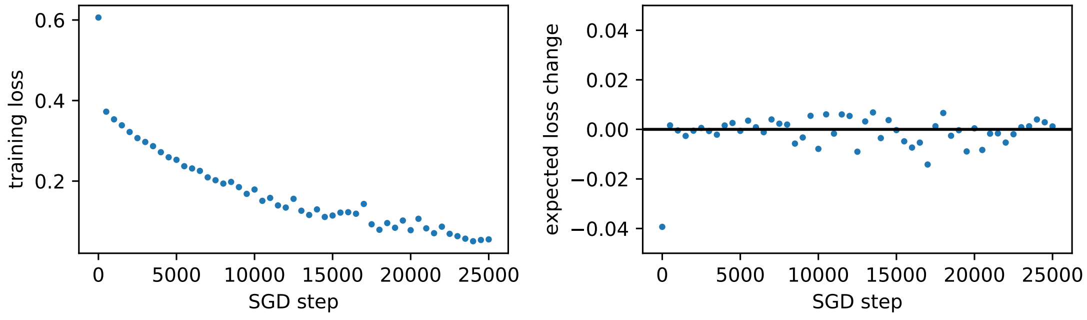

Figure 6: **SGD does not consistently decrease the training loss in expectation.** We train the ELU network from section 2 using SGD with learning rate 0.01 and batch size 32. At regular intervals during training, we compute (left) the full-batch training loss, and (right) the expected change in the full-batch training loss (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 7.

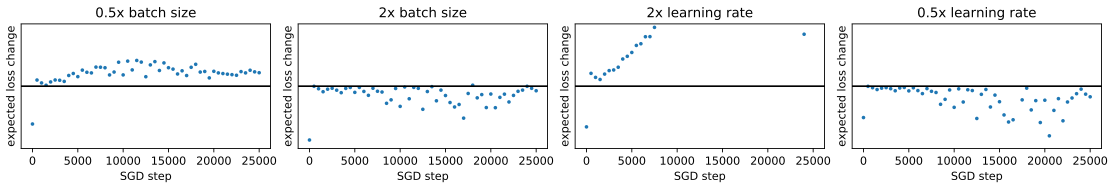

Figure 7: **An SGD step with a smaller learning rate or a larger batch size than the ones used during training *would* consistently decrease the loss in expectation.** At regular intervals during the training run depicted in Figure 6 (with $\eta = 0.01$ and batch size 32), we measure the expected change in the full-batch training loss that would result from an SGD step with a different learning rate or batch size. Observe that taking an SGD step with a smaller learning rate or a larger batch size would consistently have decreased the loss in expectation, while taking an SGD step with a larger learning rate or a smaller batch size would have consistently increased the loss in expectation.

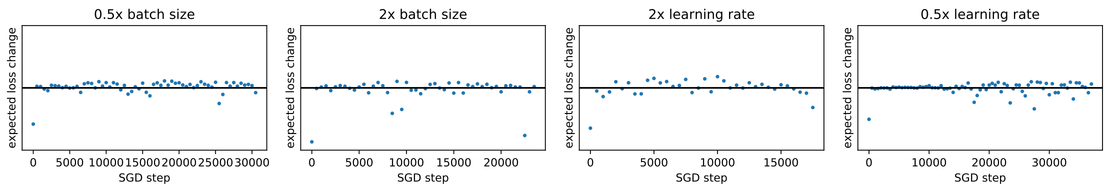

Figure 8: **There is nothing special about the hyperparameters we have used.** Figure 6 showed that if we train a network with $\eta = 0.01$ and batch size 32, the train loss does not consistently decrease in expectation, yet 7(b) showed that at every point during training that network, an SGD step with batch size 64 would have decreased the loss in expectation. A natural question is what would have happened had we trained with $\eta = 0.01$ and batch size 64. Figure 8(b) demonstrates that the train loss would not consistently decrease in expectation. The other panes are analogous.

Appendix C. 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

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

starting from the initialization $\mathbf{x}_0 = \mathbf{0}$. Our notation is intentionally chosen to resemble a second-order Taylor approximation.

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 [14, 36, 44] 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 Goodfellow et al. [14], Sutskever et al. [44] is an adaptation of Nesterov's accelerated gradient [32] 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 $\mathbf{x}_0 = \mathbf{0}$. Let $\mathbf{q}$ be the leading eigenvector of $\mathbf{H}$ and let λ be the sharpness. (Assume that $\lambda > 0$.) Then the sequence $\{\mathbf{q}^T\mathbf{x}_t\}$ will diverge iff $\lambda > 2/\eta$.*

Note that in the main text, we referred to $\mathbf{q}^T\mathbf{x}_t$ as Δ_t.

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

$$\mathbf{x}_{t+1} = (\mathbf{I} - \eta\mathbf{H})\mathbf{x}_t - \eta\mathbf{g}.$$

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

$$\mathbf{q}^T\mathbf{x}_{t+1} = (1 - \eta\lambda)\mathbf{q}^T\mathbf{x}_t - \eta\,\mathbf{q}^T\mathbf{g}$$

Define $\tilde{x}_t = \mathbf{q}^T\mathbf{x}_t + \frac{1}{\lambda}\mathbf{q}^T\mathbf{g}$, 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:

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

The sequence $\tilde{x}_t$ will diverge iff $|1 - \eta\lambda| > 1$. Since $\eta > 0$ and $\lambda > 0$, it is impossible for $1 - \eta\lambda > 1$, so $\tilde{x}_t$ diverges if and only if $1 - \eta\lambda < -1 \iff \lambda > 2/\eta$. ■

Remark: to recover the formula given in the main text for the evolution of $\mathbf{q}^T\mathbf{x}_t$, note that $\mathbf{q}^T\mathbf{x}_t = \tilde{x}_t - \frac{1}{\lambda}\mathbf{q}^T\mathbf{g} = (1-\eta\lambda)^t\tilde{x}_0 - \frac{1}{\lambda}\mathbf{q}^T\mathbf{g} = ((1-\eta\lambda)^t - 1)(\frac{1}{\lambda}\mathbf{q}^T\mathbf{g})$.

We now prove analogous results for Nesterov and Polyak momentum.

Theorem 2 *Consider running Nesterov momentum on the quadratic objective (2) starting from* $\mathbf{x}_0 = \mathbf{0}$*. Let* $\mathbf{q}$ *be the leading eigenvector of* $\mathbf{H}$ *and let* λ *be the sharpness. (Assume that* $\lambda > 0$*.) Then the sequence* $\{\mathbf{q}^T \mathbf{x}_t\}$ *will diverge iff* $\lambda > \frac{1}{\eta}\left(\frac{2+2\beta}{1+2\beta}\right)$*.*

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

$$\mathbf{v}_{t+1} = \beta(\mathbf{I} - \eta\mathbf{H})\mathbf{v}_t - \eta\mathbf{g} - \eta\mathbf{H}\mathbf{x}_t$$
$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}_{t+1}.$$

Define $\tilde{x}_t = \mathbf{q}^T\mathbf{x}_t + \frac{1}{\lambda}\mathbf{q}^T\mathbf{g}$ 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:

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

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

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

This is a linear homogenous second-order difference equation. By Theorem 2.37 in Elaydi [10], since $\lambda > 0$ and $\eta > 0$ and $\beta < 1$, this recurrence diverges if and only if $\lambda > \frac{1}{\eta}\left(\frac{2+2\beta}{1+2\beta}\right)$.

■

The following result previously appeared in Goh [13].

Theorem 3 *Consider running Polyak momentum on the quadratic objective (2) starting from* $\mathbf{x}_0 = 0$*. Let* $\mathbf{q}$ *be the leading eigenvector of* $\mathbf{H}$ *and let* λ *be the sharpness. (Assume that* $\lambda > 0$*.) Then the sequence* $\{\mathbf{q}^T\mathbf{x}_t\}$ *will diverge iff* $\lambda > \frac{1}{\eta}(2+2\beta)$*.*

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

$$\mathbf{v}_{t+1} = \beta\mathbf{v}_t - \eta\mathbf{H}\mathbf{x}_t - \eta\mathbf{g}$$
$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}_{t+1}.$$

Define $\tilde{x}_t = \mathbf{q}^T\mathbf{x}_t + \frac{1}{\lambda}\mathbf{q}^T\mathbf{g}$ 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 Polyak momentum as:

$$\tilde{v}_{t+1} = \beta\tilde{v}_t - \eta\lambda\tilde{x}_t$$
$$\tilde{x}_{t+1} = \tilde{x}_t + \tilde{v}_{t+1}.$$

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

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

This is a linear homogenous second-order difference equation. By Theorem 2.37 in Elaydi [10], since $\lambda > 0$ and $\eta > 0$ and $\beta < 1$, this recurrence diverges if and only if $\lambda > \frac{1}{\eta}(2 + 2\beta)$.

■

None of these three results have used the fact that $(\mathbf{q}, \lambda)$ are the *leading* eigenvector and eigenvalue; they would also apply to any eigenvector/eigenvalue pair $(\mathbf{q}_i, \lambda_i)$. Thus, along any direction of positive curvature (i.e. $\lambda_i > 0$), gradient descent (vanilla, or with Polyak/Nesterov momentum) diverges iff λ_i is strictly larger than some threshold, and converges iff λ_i is strictly smaller than that threshold. If the quadratic is convex (i.e. $\mathbf{H} \succ 0$), this means that gradient descent will converge to the solution iff the sharpness is strictly less than that threshold, and will diverge iff the sharpness is strictly greater than that threshold.

If $\mathbf{H}$ has negative eigenvalues, then gradient descent (with a positive step size) will diverge along the directions of negative curvature. Of course, neural network training Hessians do have negative eigenvalues [11, 40], yet gradient descent does not diverge during normal training. The reason may be that the directions of negative curvature are somewhat illusory: Alain et al. [2] walked along directions of negative curvature, and found that the negative curvature did not last for very long, whereas the directions of most positive curvature lasted for longer. In other words, the quadratic Taylor approximation was accurate over greater distances for the directions of positive curvature than for the directions of negative curvature.

Appendix D. Infinite Width Limits

In this appendix, we conduct an experiment which aims to reconcile progressive sharpening with the theory of infinite-width neural networks. It is well-known that, under a certain network parameterization and initialization, when training infinitely wide neural networks using gradient flow or gradient descent with sufficiently small step sizes, the parameters move almost no distance from their initial position, and therefore the Hessian does not change from initialization [18, 19, 25]. Accordingly, one might hypothesize that progressive sharpening might attenuate as networks (with NTK parameterization) become increasingly wide.

We consider fully-connected networks under the NTK parameterization [25] with two hidden layers and tanh activations. We consider networks with widths 100, 200, 400, and 800. We train these four networks on the Fashion-MNIST dataset using the cross-entropy loss, with a small learning rate (aiming to mimic gradient flow). We use the same learning rate for each network (this is sensible under the NTK parameterization). We stop each network when the loss reaches a certain threshold.

In Figure 9(b), we plot the sharpness by iteration. One can see that sharpening happens quicker for the narrow networks than for the wider networks. However, there is a confounding factor: as can be seen from Figure 9(a), the narrower networks train faster. To remove this confounding factor, in Figure 9(c), we plot sharpness as a function of train loss. Here one can clearly see that at every train loss, the narrower (resp: wider) networks have a higher (resp: lower) sharpness. This observation aligns with NTK theory.

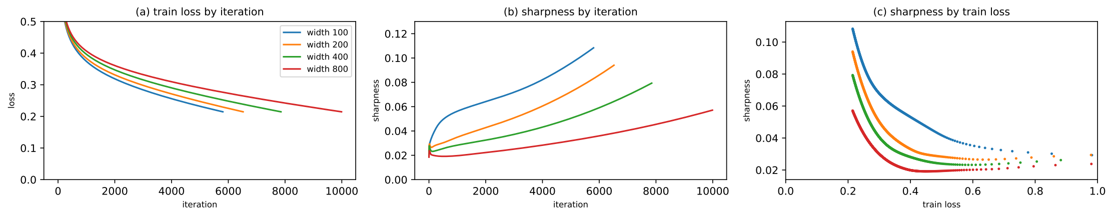

Figure 9: **Under the NTK parameterization, wider networks find a less-sharp solution for any given value of the training loss.**

Appendix E. Reconcling our results with Santurkar et al. [42]

In this appendix, we reconcile the apparent contradiction between our results and those of the well-known work Santurkar et al. [42].

Aiming to explain the effectiveness of batch normalization (BN), Santurkar et al. [42] conjectured that BN smooths the optimization landscape, in three different senses of the word “smooth”:

1. BN improves the Lipschitzness of the objective function along the optimization trajectory. To demonstrate this, at regular intervals during training, Santurkar et al. [42] measured the change in loss that would arise from taking a step in the negative gradient direction. (Note that the step size used here was different from the training step size.) Note that this step size was larger than the step size used during training — in their Figure 4(a), it was larger by 4x; in their Figure 9(a), it was larger by $3 \cdot 10^7$x.

2. BN improves the *gradient predictiveness* of the objective function along the optimization trajectory. Here, gradient predictiveness is defined as the ℓ_2 distance between the gradient at the current iterate, and the gradient after taking a very large step in the negative gradient direction. To demonstrate this, Santurkar et al. [42] measured the gradient predictiveness at regular intervals during training.

3. BN improves the *effective smoothness* along the optimization trajectory, defined as the Lipschitz constant of the gradient in the direction of the negative gradient. Given an objective function f, an iterate θ, and a step size α, the effective smoothness is defined as
$$\sup_{\gamma \in [0, \alpha]} \frac{\|\nabla f(\theta) - \nabla f(\theta - \gamma \nabla f(\theta))\|_2}{\|\gamma \nabla f(\theta)\|_2}.$$
To demonstrate this, Santurkar et al. [42] measured the effective smoothness at regular intervals during training.

Note that Santurkar et al. [42] did *not* conjecture that BN improves the L-smoothness (a.k.a decrease the sharpness) along the optimization trajectory — only that BN improves the *effective* smoothness, which is the L-smoothness in the negative gradient direction.

Our main paper demonstrates that BN does not improve the L-smoothness along the optimization trajectory, at least for full-batch gradient descent. During gradient descent with step size η, for both BN and non-BN networks, if the sharpness is less than $2/\eta$, then the sharpness eventually rises to approximately $2/\eta$, where it remains for the duration of training. Thus, in the long run, for both BN and non-BN networks, the smoothness settles at approximately the same value — and moreover, this is precisely the value that voids the descent lemma (and arguments based on L-smoothness).

Moreover, we now demonstrate that BN does not improve the effective smoothness either (at least during full-batch gradient descent). In Figure 10, we train a ReLU CNN both with (top row) and without (bottom row) batch normalization, at the same grid of learning rates, using the cross-entropy loss on a subset of 5k examples from CIFAR-10. (For the BN network we use ghost batch normalization with two ghost batches of size 2,500, since a batch with all 5k examples does not fit into GPU memory.) We measure both the sharpness/smoothness and the effective smoothness at regular intervals during training. (We numerically evaluate the sup in the definition of effective smoothness using a grid of 10 evenly spaced points.) Observe that for each learning rate, the effective smoothness behaves similar to the smoothness: it rises to $2/\eta$ and then remains approximately there, until the end of training, when both values drop because we are using the cross-entropy loss. Thus, BN does not improve the effective smoothness along the optimization trajectory. Despite this, one can see that the BN network trains substantially faster than the non-BN network.

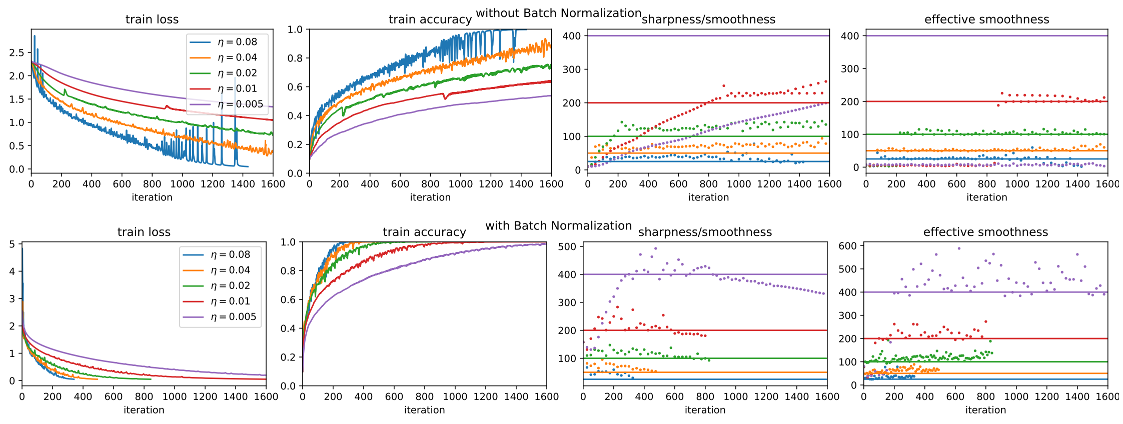

Figure 10: **Batch normalization does not improve the smoothness or the effective smoothness along the optimization trajectory.**

Note that this finding is actually consistent with Figure 4(c) and Figure 11(d) in Santurkar et al. [42], which are meant to show, respectively, that both BN and their ℓ_p normalization schemes improve effective smoothness on a VGG network. These two plots show that for both BN and non-BN networks trained using SGD with step size $\eta = 0.1$, the effective smoothness hovers around the value $20 = 2/\eta$, which is exactly what we observe. Their plots do show that the effective smoothness behaves more *regularly* for the batch-normalized network than for the non-BN network. But we strongly disagree with their interpretation of these two figures as demonstrating that BN improves the effective smoothness during training.

The third and final piece of evidence in Santurkar et al. [42] for the conjecture that BN improves the effective smoothness is their Figure 9(c), which shows that BN improves the effective smoothness on a deep linear network. For this figure, the step size α used in their computation of effective smoothness was larger than the training step size η by a factor of $3 \cdot 10^7$. (For the VGG network, α was larger than η by a factor of 4.) The effective smoothness at this massive distance has no bearing on training.

Appendix F. Experimental details

F.1. Tasks

We used the following tasks:

- **CIFAR-10** [24]: we standardize each channel by subtracting the mean and dividing by the standard deviation.

- **CIFAR-10 5k**: this is a subset of CIFAR-10 of size 5,000. In particular, we retain every 10-th train example.

- **Wikitext-2**: this is a language modeling dataset. We use `bptt = 35`, i.e. we predict text in contiguous blocks of 35 words. We use only the first 4970 (it is divisible by 35) training examples in the dataset.

- **Deep linear network task**: The task is to map the matrix $\mathbf{X} \in \mathbb{R}^{n \times d} = \begin{bmatrix} \mathbf{x}_1, & \dots, & \mathbf{x}_n \end{bmatrix}^T$ to the matrix $\mathbf{Y} \in \mathbb{R}^{n \times d} = \begin{bmatrix} \mathbf{y}_1, & \dots, & \mathbf{y}_n \end{bmatrix}^T$ using a deep linear network $f : \mathbb{R}^d \to \mathbb{R}^d$ that consists of a stack of L matrices each of size $d \times d$. Error is measured using the squared loss: $\frac{1}{n} \sum_{i=1}^n \|f(\mathbf{x}_i) - \mathbf{y}_i\|_2^2$.

  We use $n = d = 50$. We generate $\mathbf{X}$ as a random whitened matrix (i.e. $\frac{1}{n}\mathbf{X}^T\mathbf{X} = \mathbf{I}$) and generate $\mathbf{Y} = \mathbf{X}\mathbf{A}^T$, where $\mathbf{A} \in \mathbb{R}^{d \times d}$ is a random matrix of standard Gaussians.

  To generate $\mathbf{X}$ as a random whitened matrix, we sample a 50×50 matrix of standard Gaussians, then we let $\mathbf{X}$ be $\sqrt{50}$ times the Q factor in the QR factorization of that matrix.

- **Toy regression task**: This is a one-dimensional regression task using the square loss. We sample 20 datapoints uniformly spaced between -1 and 1, and we label these datapoints noiselessly using the Chebyshev polynomial of degree 5. The dataset is plotted in Figure 11.

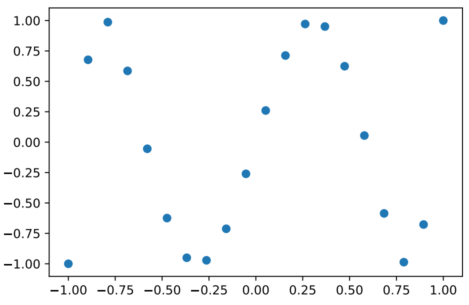

Figure 11: The dataset for the toy one-dimensional regression task.

F.2. Architectures

- **VGG-11 (no BN)**: we took the VGG-11 sized for CIFAR-10 available at `https://github.com/chengyangfu/pytorch-vgg-cifar10`, and we removed the dropout layers. In Figure **??**, we "warm-started" optimization by running GD for 400 iterations at learning rate 0.02. The reason we do this is that loss landscape is very flat at initialization, and if we did not do this, the network trained at, say, $\eta = 0.00125$ would basically not budge for the first several thousand iterations of training.

- **VGG-11 (BN)**: same as above (though we do not need to warm start training). Since GPUs do not have enough memory to do a full-batch step, we use ghost batch normalization [15] with 50 ghost batches of size 1,000.

- **ReLU CNN**: The PyTorch code for this network is:

  ```
  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)
  )
  ```

  We use the default PyTorch initialization.

- **ReLU CNN (BN)** Same as the ReLU CNN, but we insert a `nn.BatchNorm2d` layer after each ReLU. We use 2 ghost batches of size 2,500 each.

- **Tanh CNN and Tanh CNN (BN)**: same as the ReLU CNN, except we use the tanh activation instead of ReLU.

- **ReLU fully-connected network** The PyTorch code for this network is:

  ```
  nn.Sequential(
      nn.Linear(3072, 100, bias=True),
      nn.ReLU(),
      nn.Linear(100, 100, bias=True)
      nn.ReLU(),
      nn.Linear(100, 10, bias=True)
  )
  ```

  We use the default PyTorch initialization.

- **Tanh fully-connected network**: Same as the ReLU fully-connected network, but with tanh activations.

- **ELU fully-connected network**: Same as the ReLU fully-connected network, but with ELU activations, and with hidden layer size 200 rather than 100. (There is no nefarious reason for the difference in hidden layer size; we just ran the ReLU and tanh experiments first, and then decided to make the network more overparameterized.)
- **Deep linear network**: this is a deep linear network with 20 layers, each a 50×50 matrix. We initialize each matrix using Xavier initialization: each entry is drawn from $\mathcal{N}(0, \frac{1}{50})$.
- **Toy 1d tanh network**: The PyTorch code for this network is:

  ```
  nn.Sequential(
      nn.Linear(1, 100, bias=True),
      nn.Tanh(),
      nn.Linear(100, 1, bias=False),
   )
  ```

  We initialize using Xavier initialization.
- **Transformer**: we use the Transformer from the Pytorch language modeling tutorial: https://github.com/pytorch/examples/tree/master/word_language_model with ninp=200, nhead=2, nhid=200, nlayers=2, dropout=0.

Appendix G. Complete Experiment Plots

In this section, we present complete plots for the experiments described in Section **??**. The purpose of these experiments is to demonstrate the following points:

1. When sharpness is less than $2/\eta$, gradient descent has an overwhelming tendency to increase the sharpness (*progressive sharpening*).

2. So long as the sharpness is less than $2/\eta$, gradient descent at different learning rates roughly follows the same path, just at different speeds. This is because gradient descent is roughly tracking the gradient flow trajectory (see the animation in Bach [4] for a visualization of what we mean). The way we demonstrate this is by plotting the train loss and the sharpness by time; when plotted by time, the sharpness and training loss of GD at different learning rates almost exactly coincide.

3. If the sharpness rises to $2/\eta$ (see the caveat below), gradient decent subsequently enters the Edge of Stability, a regime where (1) the training loss typically behaves non-monotonically, and (2) the sharpness hovers right at, or just above, the value $2/\eta$, rather than increasing further.

4. For sufficiently small learning rates η, gradient descent can terminate before the sharpness reaches $2/\eta$. We will demonstrate that these small learning rates are highly suboptimal in terms of convergence speed — therefore, we say that they are not “reasonable.”

   On some problems, it is feasible to actually train to completion at these small learning rates. In this case, our experiments demonstrate that larger learning rates (which enter the Edge of Stability) train faster. On other problems, sharpening seems to occur so rapidly that training to completion at these small learning rates is not a practical experiment for us to run; however, our gradient flow thought experiment from section 2 implies that these small learning rates should exist.

There is one slight caveat. On a few optimization problems (e.g. Figure 24), we observe that the gradient descent enters the Edge of Stability regime when the sharpness is a little bit smaller than $2/\eta$. This is because we are only measuring the sharpness right at the iterates themselves. However, if the sharpness *in between* the iterates (which we do not measure) exceeds $2/\eta$, then it is possible for a gradient step to increase the training loss, and for gradient descent to become destabilized.

Note that a common pattern in our plots (visible in Figure **??**) is for the train loss to spike right when the sharpness first crosses $2/\eta$, and then *appear* to decrease monotonically for a while before later turning non-monotonic. Here is what is going on: after the sharpness first crosses $2/\eta$, only the leading Hessian eigenvalue is greater than $2/\eta$, and the others are less than $2/\eta$. During this period, the training loss *is* actually behaving non-monotonically, but the non-monotonicity is very slight: often the training loss follows a sawtooth pattern where every other step slightly increases the loss. While this is happening, the second-largest eigenvalue is steadily increasing. (Sharpening also occurs with non-leading eigenvalues.) Once the second-largest eigenvalue crosses $2/\eta$, *then* the major non-monotonicity ensues.

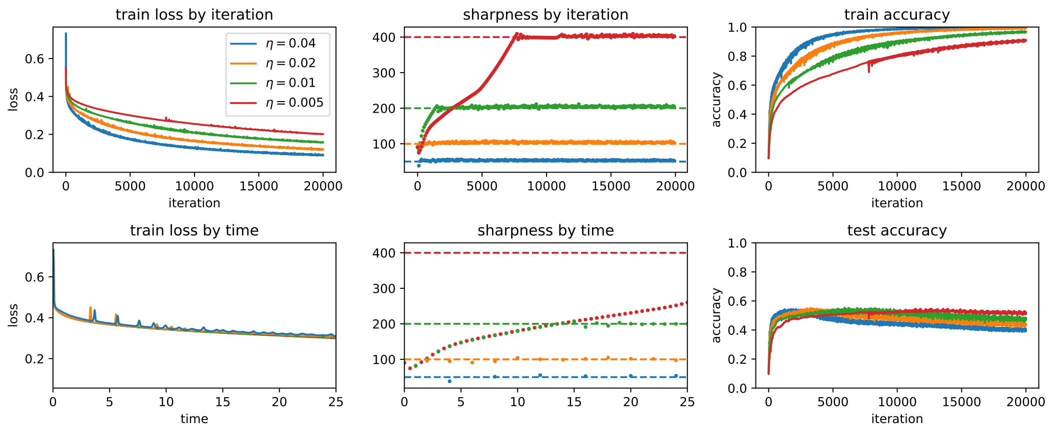

Figure 12: **ReLU CNN trained with square loss on 5k subset of CIFAR-10.** (1) For the blue learning rate, the initial sharpness was greater than $2/\eta$, so gradient descent catapulted at the beginning of training (which is why the blue train loss doesn't coincide with the others when plotted by time). (2) When we plot the sharpness by time, the red and green dots almost (but not exactly) overlap before the green one enters the Edge of Stability, the red and green learning rates are roughly following the same path.

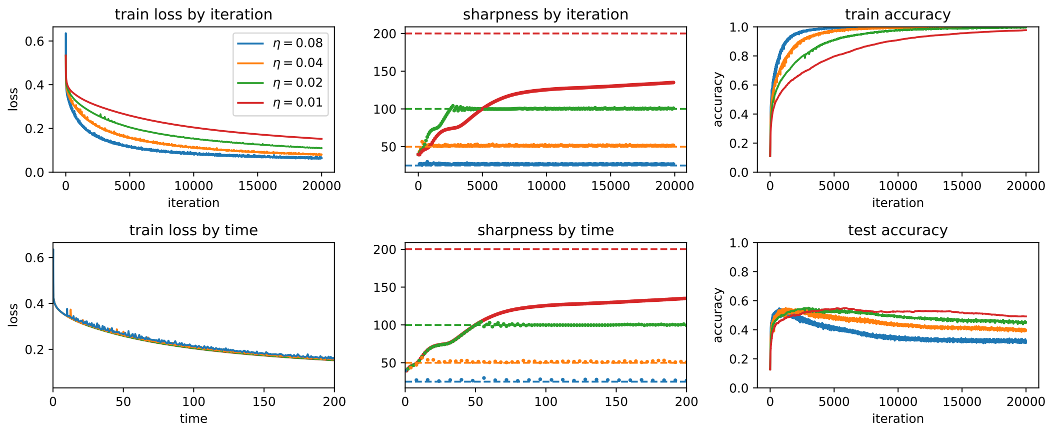

Figure 13: **Tanh CNN trained with square loss on 5k subset of CIFAR-10.** The blue learning rate is large enough that the initial sharpness was greater than $2/\eta$.

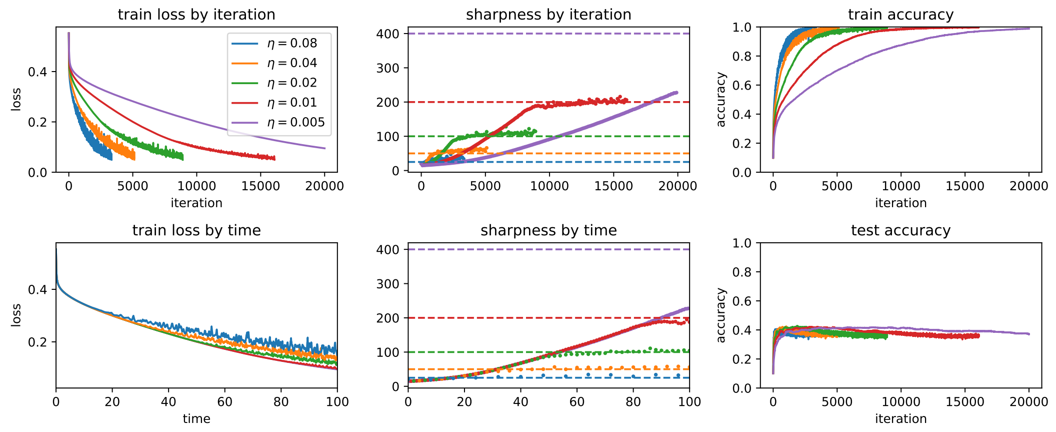

Figure 14: **ReLU fully-connected network trained with square loss on 5k subset of CIFAR-10.**. Here we use the cross-entropy loss, so the sharpness drops near the end of training.

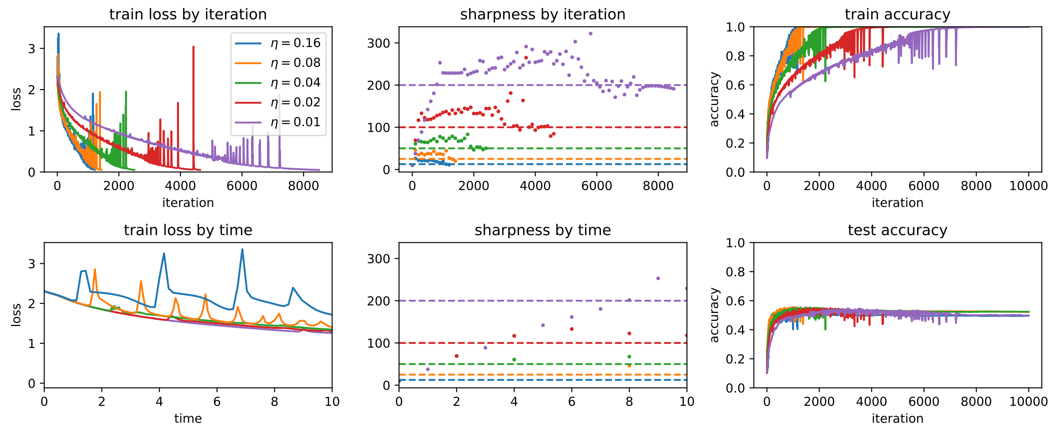

Figure 15: **ReLU CNN trained with cross-entropy on 5k subset of CIFAR-10.**

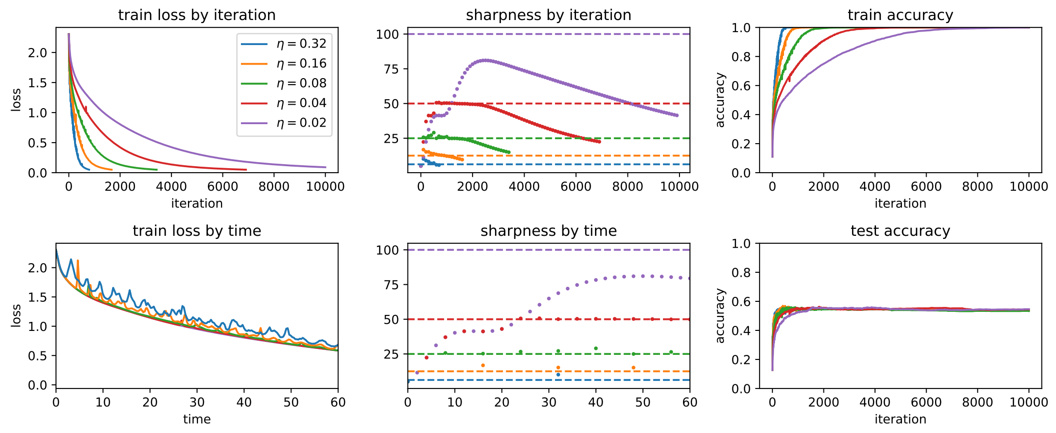

Figure 16: **Tanh CNN trained with cross-entropy on 5k subset of CIFAR-10.**

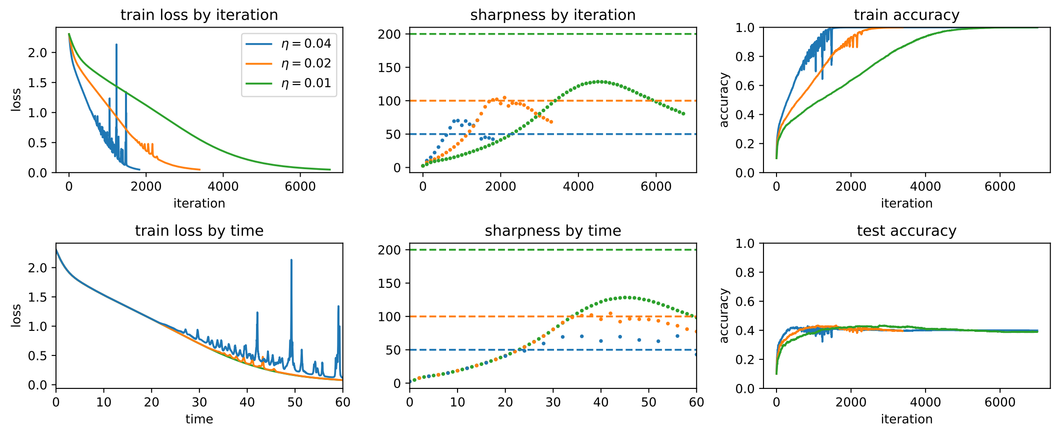

Figure 17: **ReLU fully-connected network trained with cross-entropy on 5k subset of CIFAR-10.**

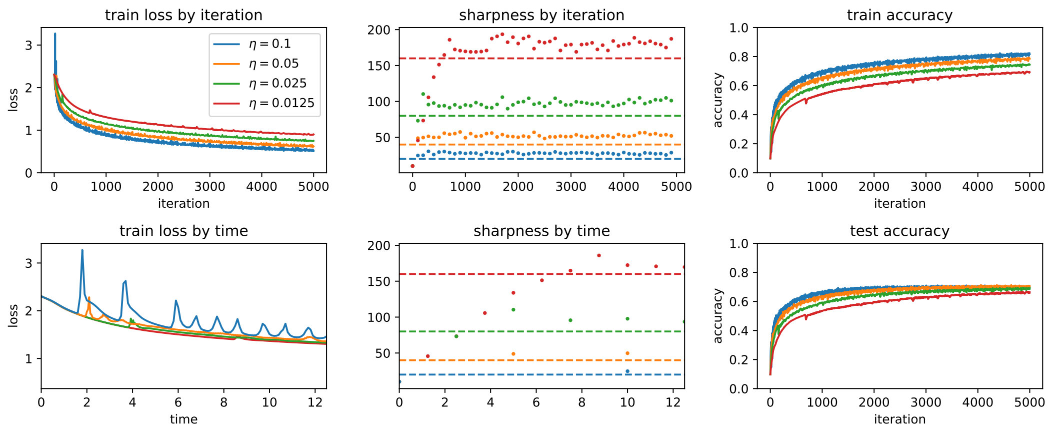

Figure 18: **ReLU CNN trained with cross-entropy on full CIFAR-10.**

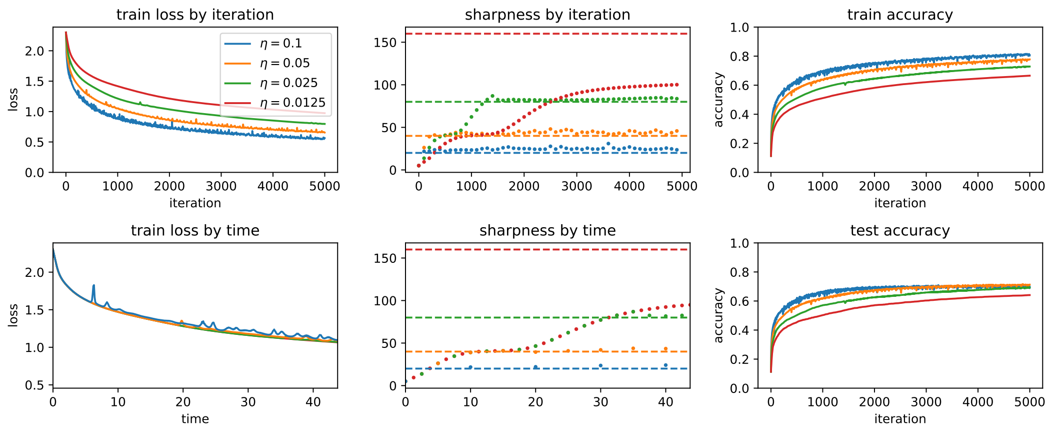

Figure 19: **Tanh CNN trained with cross-entropy on full CIFAR-10.**

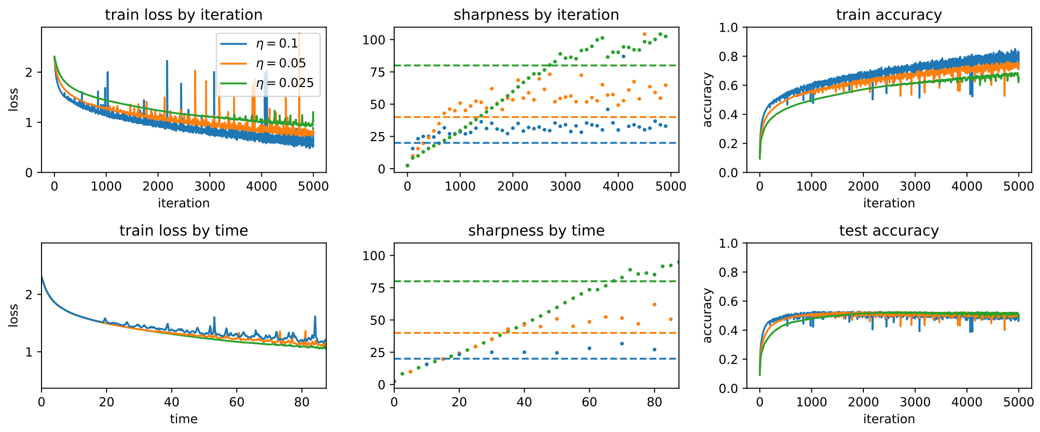

Figure 20: **ReLU fully-connected network trained with cross-entropy on full CIFAR-10.**

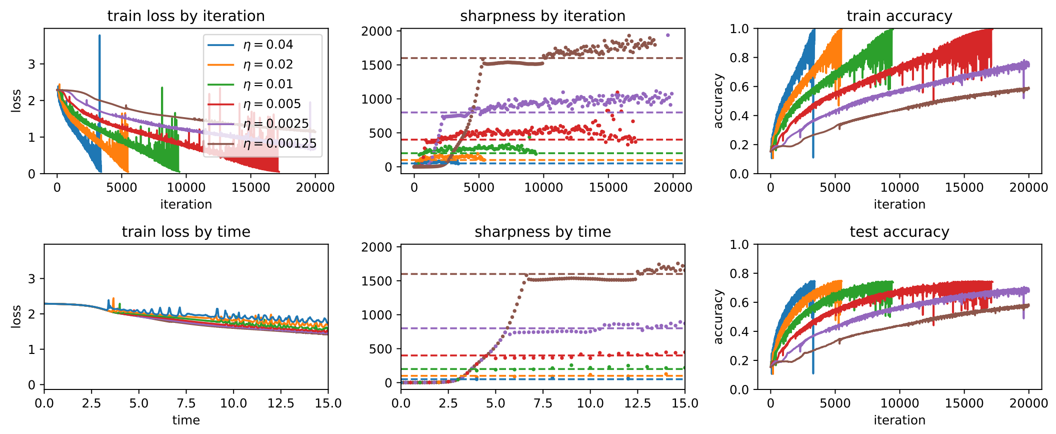

Figure 21: **VGG-11 (no BN) trained with cross-entropy on full CIFAR-10.**

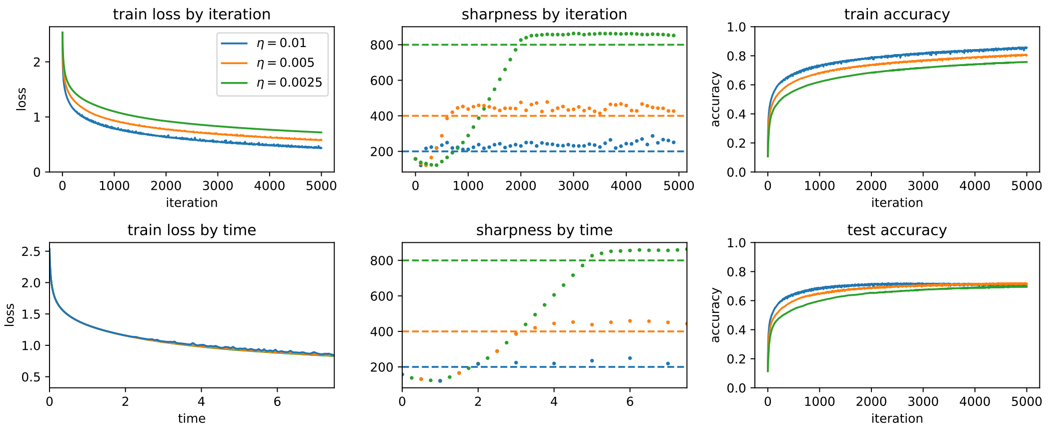

Figure 22: **ReLU CNN (with BN) trained with cross-entropy on full CIFAR-10.** Note that at the beginning of training, the sharpness decreases for a short while. This is why we phrase progressive sharpening as an "overwhelming tendency" for the sharpness to increase during gradient flow, not a hard-and-fast rule.

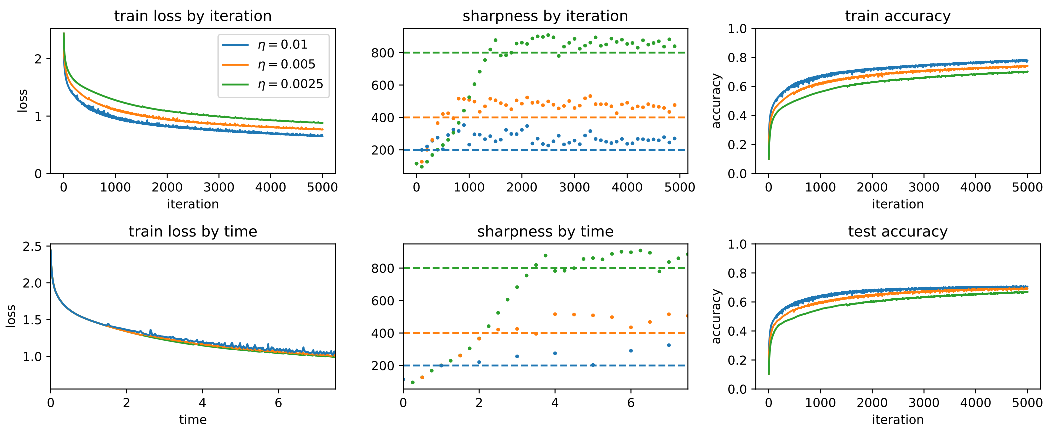

Figure 23: **Tanh CNN (with BN) trained with cross-entropy on full CIFAR-10.**

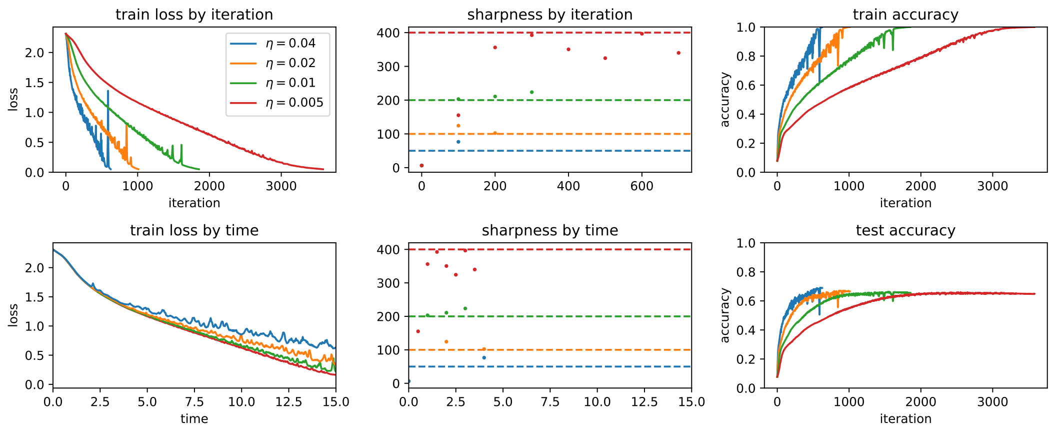

Figure 24: **VGG-11 (with BN) trained with cross-entropy on full CIFAR-10.** The red learning rate enters the Edge of Stability when the sharpness directly on the gradient descent trajectory is slightly less $2/\eta$; we have verified that during this period, the sharpness in *between* iterates exceeds $2/\eta$.

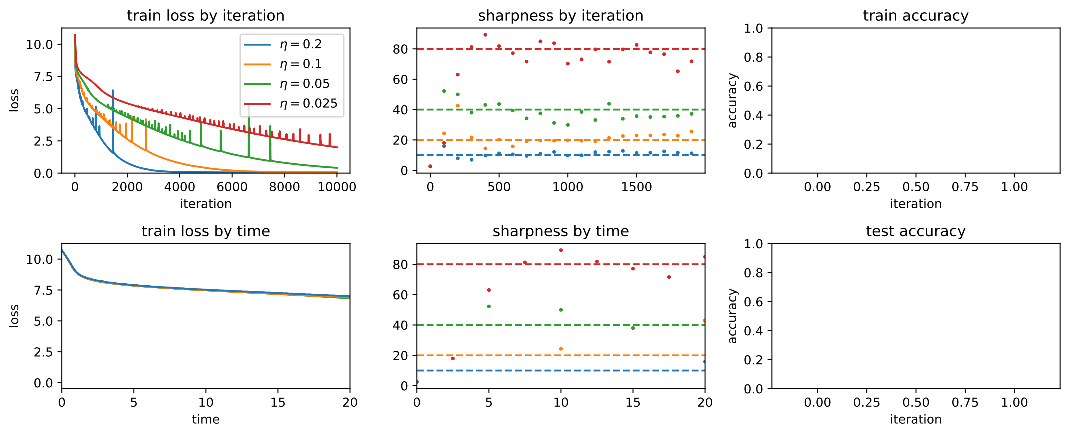

Figure 25: **Transformer trained on a subset of the WikiText-2 language modeling dataset.**

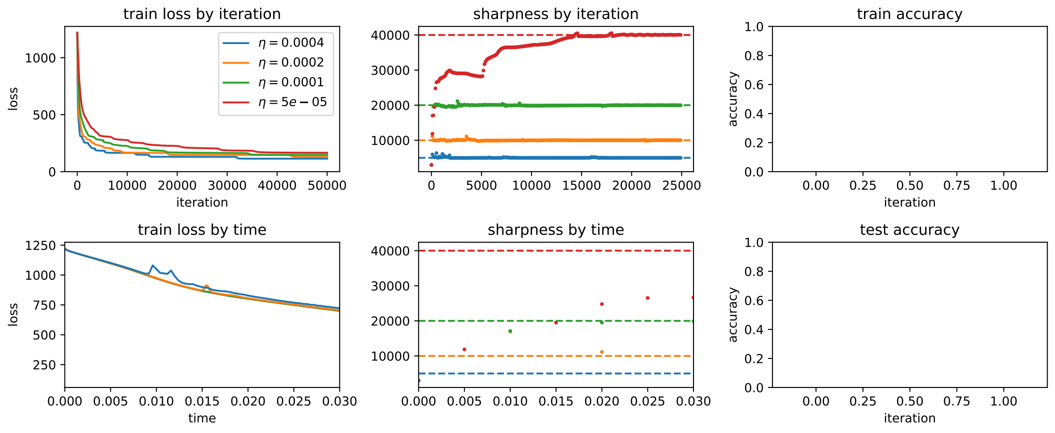

Figure 26: **Deep linear network trained on isotropic Gaussian data.**

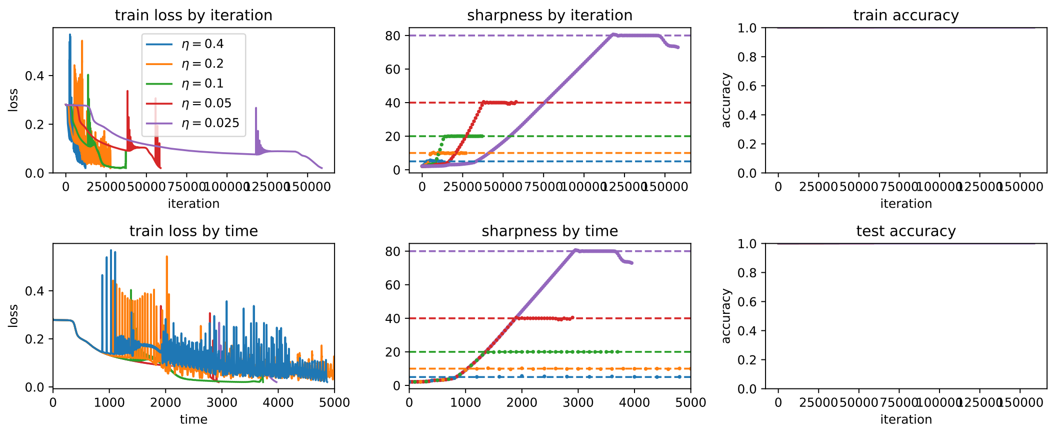

Figure 27: **Tanh fully-connected network with one hidden layer trained on toy 1-dimensional regression task.**

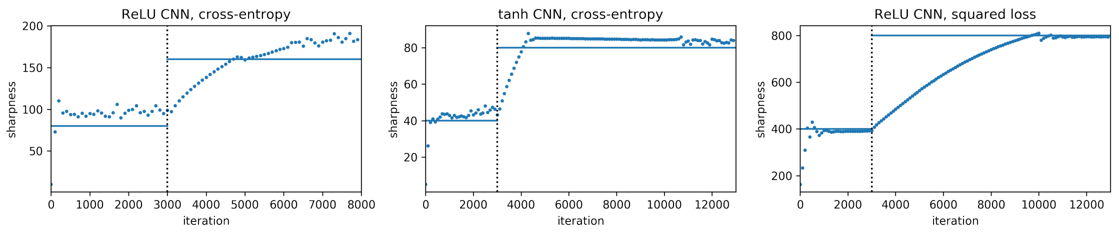

Figure 28: **Additional learning rate drop experiments.** If we cut the learning rate while the network is at the Edge of Stability, then progressive sharpening resumes until the network is once again at the Edge of Stability. (These networks are trained on the full CIFAR-10 dataset.)