

# AN EMPIRICAL ANALYSIS OF DEEP NETWORK LOSS SURFACES

Daniel Jiwoong Im<sup>1,2\*</sup>, Michael Tao<sup>3</sup>, & Kristin Branson<sup>2</sup>

<sup>1</sup>Janelia Research Campus, HHMI, <sup>2</sup>AIFounded Inc. <sup>3</sup>University of Toronto,  
{imd, bransonk}@janelia.hhmi.org  
{mtao}@dgp.toronto.edu

## ABSTRACT

The training of deep neural networks is a high-dimension optimization problem with respect to the loss function of a model. Unfortunately, these functions are of high dimension and non-convex and hence difficult to characterize. In this paper, we empirically investigate the geometry of the loss functions for state-of-the-art networks with multiple stochastic optimization methods. We do this through several experiments that are visualized on polygons to understand how and when these stochastic optimization methods find local minima.

## 1 INTRODUCTION

Deep neural networks are trained by optimizing an extremely high-dimensional loss function with respect to the weights of the network’s linear layers. The objective function minimized is some measure of the error of the network’s predictions based on these weights compared to training data. This loss function is non-convex and has many local minima. These loss functions are usually minimized using first-order gradient descent (Robbins & Monro, 1951; Polyak, 1964) algorithms such as stochastic gradient descent (SGD) (Bottou, 1991). The success of deep learning critically depends on how well we can minimize this loss function, both in terms of the quality of the local minima found and the time to find them. Understanding the geometry of this loss function and how well optimization algorithms can find good local minima is thus of vital importance.

Several works have theoretically analyzed and characterized the geometry of deep network loss functions. However, to make these analyses tractible, they have relied on simplifications of the network structures, including that the networks are linear (Saxe et al., 2014), or assuming the path and variable independence of the neural networks (Choromanska et al., 2015). Orthogonally, the performance of various gradient descent algorithms has been theoretically characterized (Nesterov, 1983). Again, these analyses make simplifying assumptions, in particular that the loss function is strictly convex, i.e. there is only a single local minimum.

In this work, we empirically investigated the geometry of the real loss functions for state-of-the-art networks and data sets. In addition, we investigated how popular optimization algorithms interact with these real loss surfaces. To do this, we plotted low-dimensional projections of the loss function in subspaces chosen to investigate properties of the local minima selected by different algorithms. We chose these subspaces to address the following questions:

- What types of changes to the optimization procedure result in different local minima?
- Do different optimization algorithms find qualitatively different types of local minima?

## 2 RELATED WORK

### 2.1 LOSS SURFACES

There have been several attempts to understand the loss surfaces of deep neural networks. Some have studied the critical points of the deep linear neural networks (Baldi, 1989; Baldi & Hornik,

---

\*Work done during an internship at Janelia Research Campus

1989; Baldi & Lu, 2012). Others further investigated the learning dynamics of the deep linear neural networks (Saxe et al., 2014). More recently, several others have attempted to study the loss surfaces of deep non-linear neural networks (Choromanska et al., 2015; Kawaguchi, 2016; Soudry & Carmon, 2016).

One approach is to analogize the states of neurons as the magnetic dipoles used in spherical spin-glass Ising models from statistical physics (Parisi, 2016; Fyodorov & Williams, 2007; Bray & Dean, 2007). Choromanska et al. (2015) attempted to understand the loss function of neural networks through studying the random Gaussian error functions of Ising models. Recent results (Kawaguchi, 2016; Soudry & Carmon, 2016) have provided cursory evidence in agreement with the theory provided by Choromanska et al. (2015) in that they found that there are no “poor” local minima in neural networks still with strong assumptions.

There is some potential disconnect between these theoretical results and what is found in practice due to several strong assumptions such as the activation of the hidden units and output being independent of the previous hidden units and input data. The work of Dauphin et al. (2014) empirically investigated properties of the critical points of neural network loss functions and demonstrated that their critical points behave similarly to the critical points of random Gaussian error functions in high dimensional space. We will expose further evidence along this trajectory.

## 2.2 OPTIMIZATION

In practice, the local minima of deep network loss functions are for the most part decent. This implies that we probably do not need to take many precautions to avoid bad local minima in practice. If all local minima are decent, then the task of finding a decent local minimum quickly is reduced to the task of finding *any* local minimum quickly. From an optimization perspective this implies that solely focusing on designing fast methods are of key importance for training deep networks.

In the literature the common method for measuring performance of optimization methods is to analyze them on nice convex quadratic functions (Polyak, 1964; Broyden, 1970; Nesterov, 1983; Martens, 2010; Erdogdu & Montanari, 2015) even though the problems are applied to non-convex problems. For non-convex problems, however, if two methods converge to different local minima their performance will be dictated on how those methods solve those two convex subproblems. It is challenging to show that one method will beat another without knowledge of the sort of convex subproblems, which is generally not known apriori. What we will explore is whether indeed are some characteristics that can found experimentally. If so, perhaps one could validate where these analytical results are valid or even improve methods for training neural networks.

### 2.2.1 LEARNING PHASES

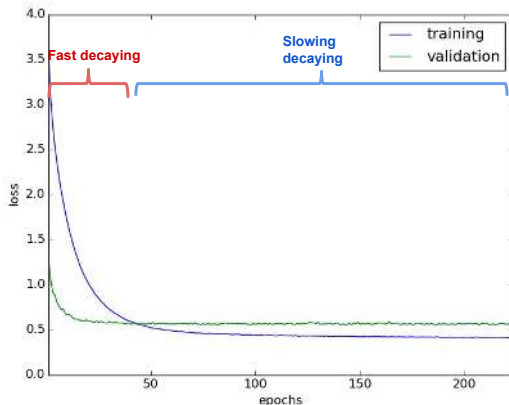


Figure 1: An example of learning curve of neural network

One of the interesting empirical observation is that we often observe is that the incremental improvement of optimization methods decreases rapidly even in non-convex problems. This behavior has been discussed as a “transient” phase followed by a “minimization” phase (Sutskever et al., 2013)

where the former finds the neighborhood of a decent local minima and the latter finds the local minima within that neighborhood. The existence of these phases implies that if certain methods are better at different phases one could create novel methods that schedule when to apply each method.

### 3 EXPERIMENTAL SETUP AND TOOLS

#### 3.1 NETWORK ARCHITECTURES AND DATA SETS

We conducted experiments on three state-of-the-art neural network architectures. Network-in-Network (NIN) (Lin et al., 2014) and the VGG(Simonyan & Zisserman, 2015) network are feed-forward convolutional networks developed for image classification, and have excellent performance on the Imagenet (Russakovsky et al., 2014) and CIFAR10 (Krizhevsky, 2009) data sets. The long short-term memory network (LSTM) (Hochreiter & Schmidhuber, 1997) is a recurrent neural network that has been successful in tasks that take variable-length sequences as input and/or produce variable-length sequences as output, such as speech recognition and image caption generation. These are large networks currently used in many machine vision and learning tasks, and the loss functions minimized by each are highly non-convex.

All results using the feed-forward convolutional networks (NIN and VGG) are on the CIFAR10 image classification data set, while the LSTM was tested on the Penn Treebank next-word prediction data set.

#### 3.2 OPTIMIZATION METHODS

We analyzed the performance of five popular gradient-descent optimization methods for these learning frameworks: Stochastic gradient descent (SGD) (Robbins & Monro, 1951), stochastic gradient descent with momentum (SGDM), RMSprop (Tieleman & Hinton, 2012), Adadelta (Zeiler et al., 2011), and ADAM (Kingma & Ba, 2014). These are all first-order gradient descent algorithms that estimate the gradients based on randomly-grouped minibatches of training examples. One of the major differences between these algorithms is how they select the weight-update step-size at each iteration, with SGD and SGDM using fixed schedules, and RMSprop, Adadelta, and ADAM using adaptive, per-parameter step-sizes. Details are provided in Section A.2.

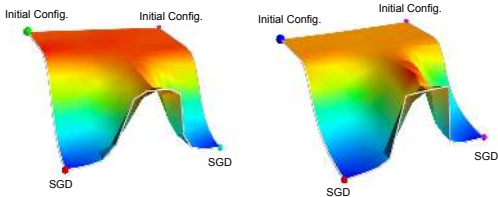
In addition to these five existing optimization methods, we compare to a new gradient descent method we developed based on the family of Runge Kutta integrators. In our experiments, we tested a second-order Runge-Kutta integrator in combination with SGD (RK2) and in combination with ADAM (ADAM&RK2). Details are provided in Section A.3).

#### 3.3 ANALYSIS METHODS

Several of our empirical analyses are based on the technique of Goodfellow et al. (Goodfellow et al., 2015). They visualize the loss function by projecting it down to one carefully chosen dimension. They plot the value of the loss function along a set of samples along this dimension. The projection space is chosen based on important weight configurations, thus they plot the value of the loss function at linear interpolations between two weight configurations. They perform two such analyses: one in which they interpolate between the initialization weights and the final learned weights, and one in which they interpolate between two sets of final weights, each learned from different initializations.

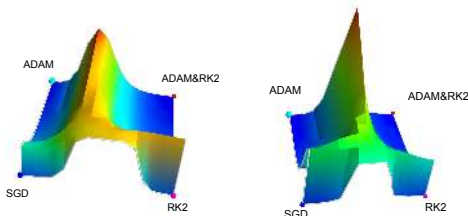
In this work, we use a similar visualization technique, but choose different low-dimensional subspaces for the projection of the loss function. These subspaces are based on the initial weights as well as the final weights learned using the different optimization algorithms and combinations of them, and are chosen to answer a variety of questions about the loss function and how the different optimization algorithms interact with this loss function. In contrast, Goodfellow et al. only looked at SGDM. In addition, we explore the use of two-dimensional projections of the loss function, allowing us to better visualize the space between local minima. We do this via barycentric and bilinear interpolation for triplets and quartets of points respectively (details in Section A.1).

We refer to the critical points found using these variants of SGD, for which the gradient is approximately 0, as local minima. Our evidence that these are local minima as opposed to saddle points is



(a) NIN

(b) VGG



(a) NIN

(b) VGG

Figure 2: Visualization of the loss surface at weights interpolated between two initial configurations and the final weight vectors learned using SGD from these initializations.

Figure 3: Visualization of the loss surface at weights interpolated between the weights learned by four different algorithms from the same initialization.

similar to that presented in Goodfellow et al. (Goodfellow et al., 2015). If we interpolate *beyond* the critical point, in this one-dimensional projection, the loss increases (Fig. 10).

### 3.4 TECHNICAL DETAILS

We used the VGG and NIN implementations from <https://github.com/szagoruyko/cifar.torch.git>.

The batch size was set to 128 and the number of epochs was set to 200. The learning rate was chosen from the discrete range between [0.2, 0.1, 0.05, 0.01] for SGD and [0.002, 0.001, 0.0005, 0.0001] for adaptive learning methods. We doubled the learning rates when we ran our augmented versions with Runge-Kutta because they required two stochastic gradient computations per epoch. We used batch-normalization and dropout to regularize our networks. All experiments were run on a 6-core Intel(R) Xeon(R) CPU @ 2.40GHz with a TITAN X.

## 4 EXPERIMENTAL RESULTS

### 4.1 DIFFERENT OPTIMIZATION METHODS FIND DIFFERENT LOCAL MINIMA

We trained the neural networks described above using each optimization method starting from the same initial weights and with the same minibatching. We computed the value of the loss function for weight vectors interpolated between the initial weights, the final weights for one algorithm, and the final weights for a second algorithm for several pairings of algorithms. The results are shown in the lower triangle of Table 1.

For every pair of optimization algorithms, we observe that the training loss between the final weights for different algorithms shows a sharp increase along the interpolated path. This suggests that each optimization algorithm found a different critical point, despite starting at the same initialization. We investigated the space between other triplets and quadruples of weight vectors (Figure 2 and 3), and even in these projections of the loss function, we still see that the local minima returned by different algorithms are separated by high loss weight parameters.

Deep networks are overparameterized. For example, if we switch all corresponding weights for a pair of nodes in our network, we will obtain effectively the same network, with both the original and permuted networks outputting the same prediction for a given input. To ensure that the weight vectors returned by the different algorithms were functionally different, we compared the outputs of the networks on each example in a validation data set:

$$\text{dist}(\theta_1, \theta_2) = \sqrt{\frac{1}{N_{test}} \sum_{i=1}^{N_{test}} \|F(x_i, \theta_1) - F(x_i, \theta_2)\|^2},$$

where  $\theta_1$  and  $\theta_2$  are the weights learned by two different optimization algorithms,  $x_i$  is the input for a validation example, and  $F(x, \theta)$  is the output of the network for weights  $\theta$  on input  $x$ .

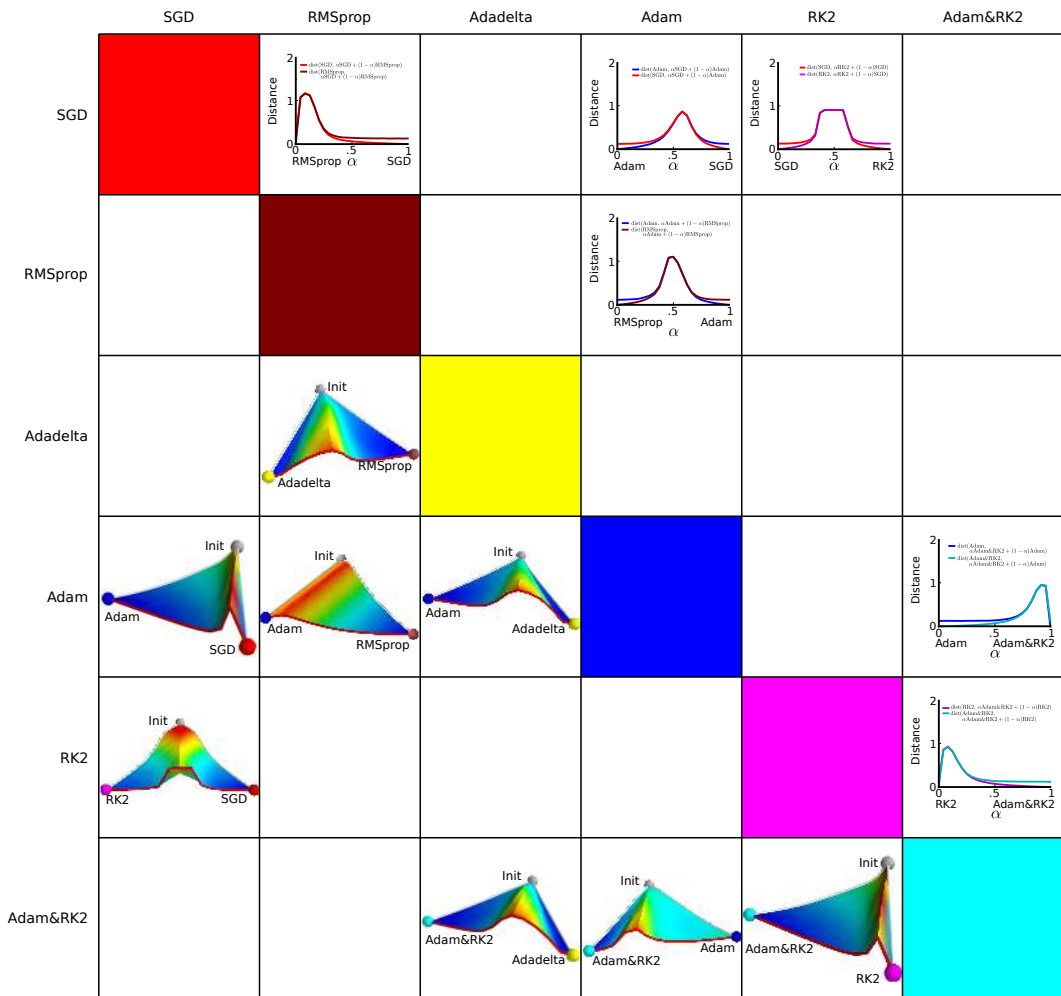


Table 1: Visualization of the loss surface near and between local minima found by different optimization methods. Each box corresponds to a pair of optimization methods. In the lower triangle, we plot the projection of the loss surface at weight vectors between the initial weight and the learned weights found by the two optimization methods. Color as well as height of the surface indicate the loss function value. In the upper triangle, we plot the functional difference between the network corresponding to the learned weights for the first algorithm and networks corresponding to weights linearly interpolated between the first and second algorithm’s learned weights. (Best viewed in zoom)

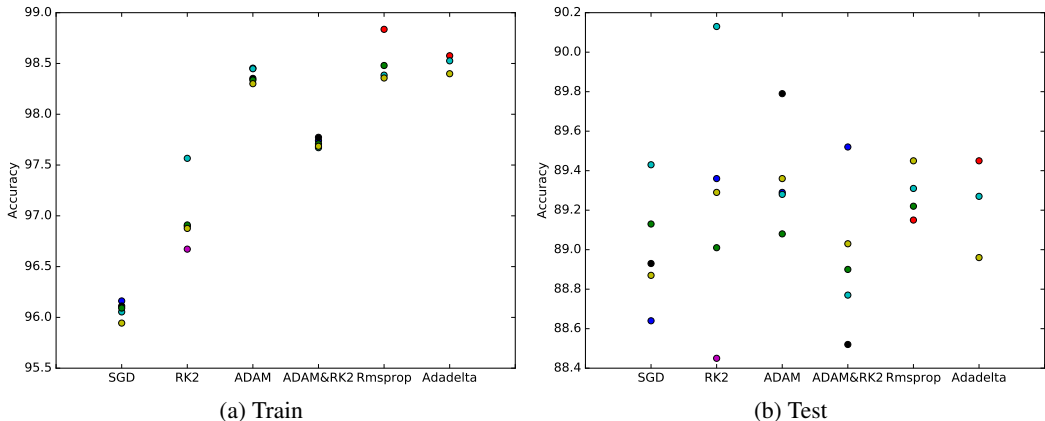
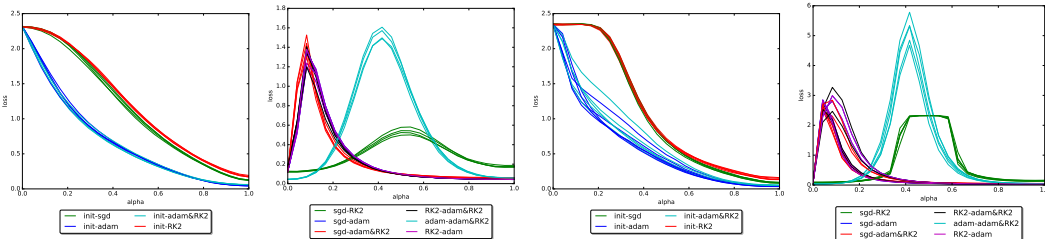


Figure 4: (a) Training accuracy and (b) test accuracy for each of the optimization methods. Colors correspond to different initializations.

We found that, for all pairs of algorithms, the average distance between the outputs of the networks (Equation 4.1) was approximately 0.16, corresponding to a label disagreement of about 8% (upper triangle of Table 1). Given the generalization error of these networks (approximately 11%, Figure 4), the maximum disagreement we could see was 22%. Thus, these networks disagreed on a large fraction of these test examples – over  $\frac{1}{3}$ rd. Thus, the local minima found by different algorithms correspond to effectively different networks, not trivial reparameterizations of the same one.



(a) NIN - Initial to Final. (b) NIN - Final to Final. (c) VGG - Initial to Final. (d) VGG - Final to Final.

Figure 5: Loss function value near local minima found by multiple restarts of each algorithm.

#### 4.2 DIFFERENT OPTIMIZATION ALGORITHMS FIND DIFFERENT TYPES OF LOCAL MINIMA

Next, we investigated whether the local minima found by the different optimization algorithms had distinguishing properties. To do this, we trained the networks with each optimization algorithm using different initial parameters. We then compared differences between runs of the same algorithm but different initializations to differences between different algorithms.

As shown in Figure 4(a), in terms of training accuracy, we do see some stereotypy for the optima found by different algorithms, with SGD finding local minima with the lowest training accuracy and ADAM, Rmsprop, and Adadelata finding local minima with the highest training accuracy. However, this could be attributed to SGD’s asymptotically slow convergence near local minima due to the gradient diminishing near extrema. Despite this limitation, Figure 4(b) shows that the generalization accuracy of these different local minima on validation data was not significantly different between algorithms. We also did not see a relationship between the weight initialization and the validation accuracy. Thus, while these algorithms fall into different local minima, they are not different in terms of their final quality.

We visualized the loss surface around each of the local minima for the multiple runs. To do this, we plotted the value of the loss function between the initial and final weights for each algorithm (Figure 5(a,c)) for each run of the algorithm from a different initialization. In addition, we plotted

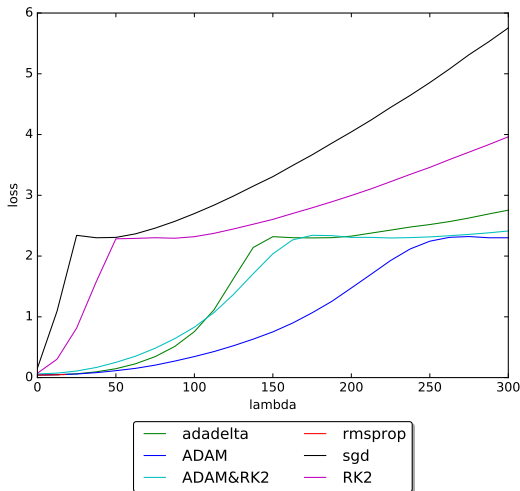


Figure 6: Observing the absolute size of basin for different local minimas found by different optimization methods.

the value of the loss function between the final weights for selected pairs of algorithms for each run (Figure 5(b,d)). We see that the surfaces look strikingly similar for different runs of the same algorithm, but characteristically different for different algorithms. Thus, we found evidence that the different algorithms land in qualitatively different types of local minima.

In particular, we see in Figure 5(a,c) that the size of the basins around the local minima found by ADAM and ADAM&RK2 are larger than those found by SGD and RK2, i.e. that the training loss is small for a wider range of  $\alpha$  values. This is a relative measure, and the magnitude of the change in the weight vector is  $\Delta\alpha\|\theta_1 - \theta_0\|$  for a change of size  $\Delta\alpha$ , where  $\theta_0$  is the initial weight vector  $\theta_1$  is the result found by a given optimization algorithm. In Figure 6, we repeat this analysis, instead showing the loss as a function of the absolute distance in parameter space:

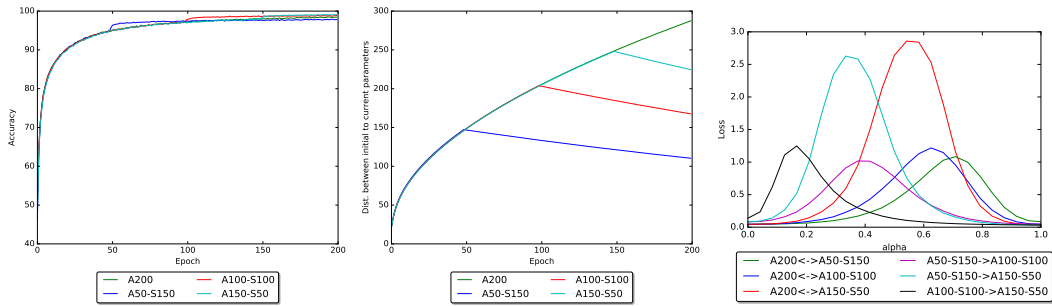
$$\theta(\lambda) = \theta_1 + \lambda \frac{\theta_0 - \theta_1}{\|\theta_0 - \theta_1\|} \quad (1)$$

We again see that the size of the basin around the local minima varies by optimization algorithm. Note that we evaluate the loss for weight vectors beyond the initial configuration, which had a loss of 2.4.

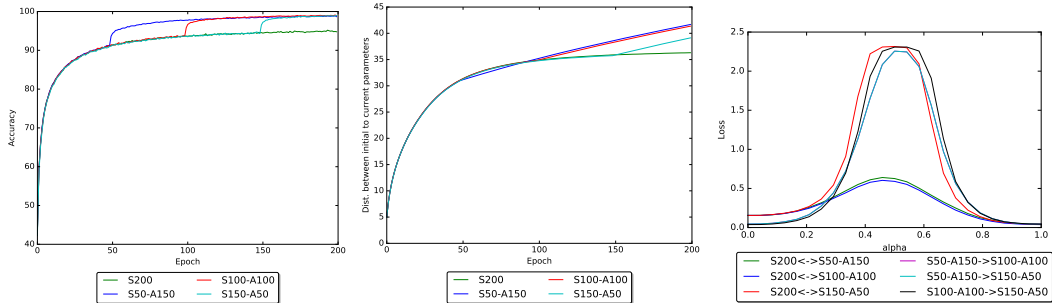
#### 4.3 ANALYZING LEARNING AFTER “TRANSIENT PERIOD”

Recall that, during optimization, it has been observed that there is a short “transient” phase when the loss decreases rapidly and a “minimization” phase in which the loss decreases slowly (Section 2.2.1 and Figure 1). In this set of experiments, we investigated the effects of switching from one type of optimization method to another at various points during training, in particular at late stages of training when it is thought that a local minimum has been chosen and is only being localized. We switched from one optimization method to another 25%, 50%, and 75% of the way through training. The results are plotted in Figure 7d. We emphasize that we are not switching methods to improve performance, but rather to investigate the shape of the loss function in regions explored during the “minimization” phase of optimization.

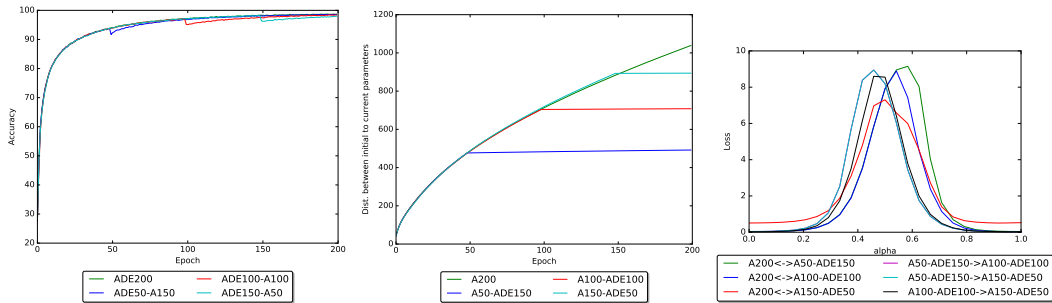
We found that, regardless of how late we switch optimization algorithms, as shown in the right-most column of Figure 7, the local minima found were all different. This directly disagrees with the notion that the local minimum has effectively been chosen before the “minimization” phase, but instead that which local minimum is found is still in flux this late in optimization. It appears that this switch from one local minimum to another happens almost immediately after the optimization method switches, with the training accuracy jumping to the characteristic accuracy for the given method within a few epochs (Figure 7, left column). Interestingly, we also see the distance between the initial and current weight vectors changes drastically after switching from one optimization



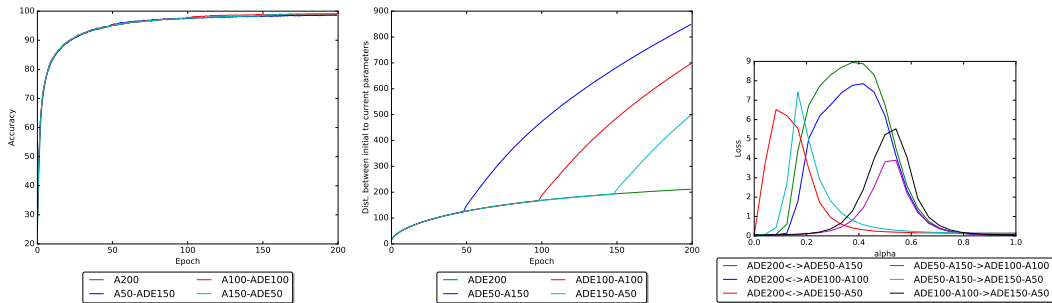
(a) The learning rate is set to 0.001 for ADAM , and then switched to SGD with learning rate 0.01.



(b) The learning rate is set to 0.1 for SGD, and then switched to ADAM with learning rate 0.0001.



(c) The learning rate is set to 0.001 for ADAM , and then switched to Adadelata (learning rate is not required).



(d) The learning rate is not required for Adadelata, and then switched to ADAM with learning rate 0.0001.

Figure 7: Switching methods from one method to another method at epoch 50 and 100 and 150. Accuracy curve (Left two columns). Distance between initial weights to weights at each epoch (Middle). The interpolation between different convergence parameters (Right). For instance, S100-A100 as trained with SGD in the first 100 epoch and switched to ADAM for the rest of the epoch. (Best viewed in zoom)



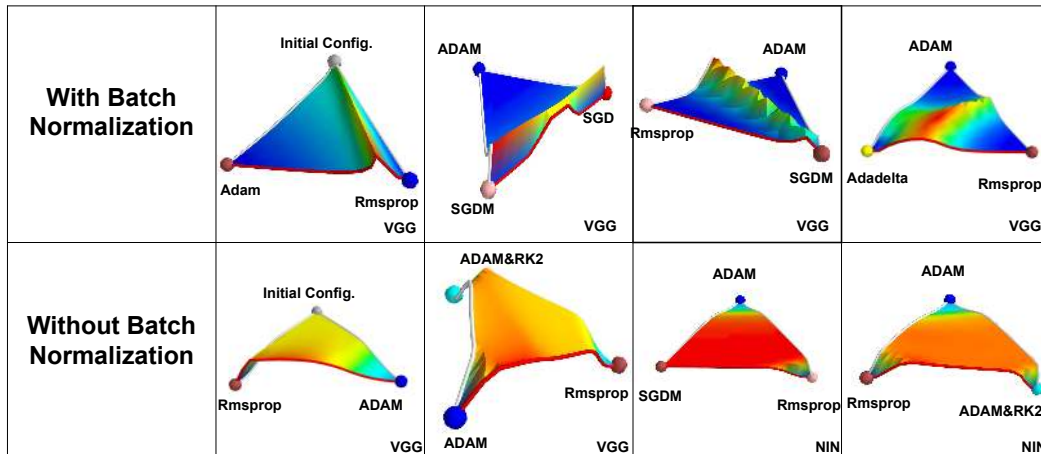


Table 2: Visualization of the Loss Surface with and without batch-normalization.

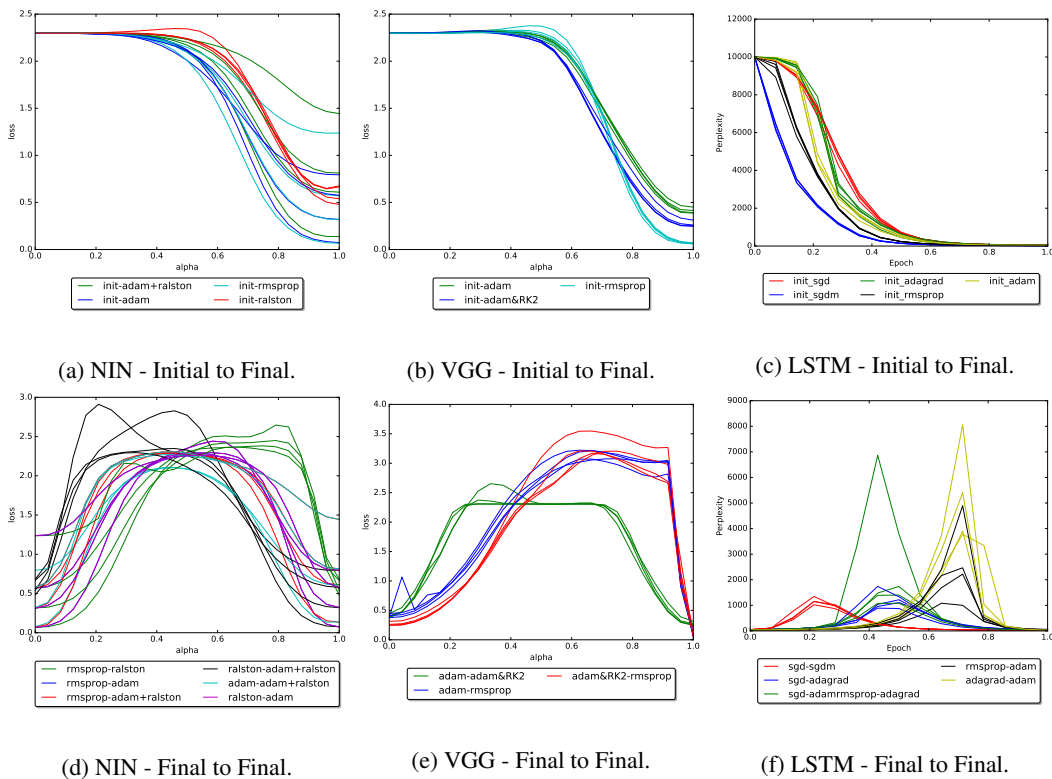


Figure 8: (Without batch normalization) Loss function with parameter interpolated between Initial to Final. (Bottom) Loss function with parameters interpolated between Final to Final. Each column of the plots are from different initializations.

method to another, and that this distance is characteristic per algorithm (Figure 7, middle column). While distance increases with training epoch for any single optimization method, it actually starts to decrease when switching from ADAM to SGD.

#### 4.4 EFFECTS OF BATCH-NORMALIZATION

To understand how batch normalization affects the types of local minima found, we performed a set of experiments comparing loss surfaces near local minima found with and without batch normal-

ization for each of the optimization methods. We visualized the surface near these local minima by interpolating between the initial weights and the final weights as well as between pairs of final weights found with different algorithms.

We observed clear qualitative differences between optimization with (Figure 5) and without (Figure 8) batch normalization. We see that, without batch normalization, the quality of local minimum found by a given algorithm is much more dependent on the initialization. In addition, the surfaces between different local minima are more complex in appearance: with batch normalization we see sharp unimodal jumps in performance but without batch normalization we obtain wide bumpy shapes that aren’t necessarily unimodal.

The neural networks are typically initialized with very small parameter values (Glorot & Bengio, 2010; He et al., 2015). Instead, we trained NIN with exotic initializations such as initial parameters drawn from  $\mathcal{N}(-10.0, 0.01)$  or  $\mathcal{N}(-1.0, 1.0)$  and observe the loss surface behaviours. The details of results are discussed in Appendix A.5.

## 5 CONCLUSIONS

In this work, we performed a series of empirical analyses to understand the geometry of the loss functions corresponding to deep neural networks, and how different optimization methods minimize this loss to answer the two questions posed in the introduction.

*What types of changes to the optimization procedure result in different local minima?*

We found that every type of change to the optimization procedure we tested resulted in a different local minimum. Different local minima were found using the different optimization algorithms from the same initialization (Section 4.1). Even switching the optimization algorithm to another very late in optimization – during the slow “mimimization” portion of learning – resulted in a different local minimum (Section 4.3). The quality of the local minima found, in terms of training and generalization error, is similar. These different local minima were not equivalent, and made mistakes on different test examples (Section 4.1). Thus, they were not trivially different local minima, as would occur if nodes in internal layers of the network were permuted. We observed that the quality of these local minima was only consistently good when we used batch normalization for regularization. Without batch normalization, the quality of the critical points found depended on the initialization, and some solutions found were not as good as others. Our observations are in contrast to the conclusions of Goodfellow et al., i.e. that local minima are not a problem in deep learning because, in the region of the loss function explored by SGD algorithms, the loss function is well-behaved (Goodfellow et al., 2015). Instead, our observations are more consistent with the explanation that the local minima found by popular SGD optimization methods are almost all good (Choromanska et al., 2015; Kawaguchi, 2016; Soudry & Carmon, 2016).

*Do different optimization algorithms find qualitatively different types of local minima?*

Interestingly, we found that, while the local minima found by the same optimization algorithm from different initializations were different, the shape of the loss function around these local minima was strikingly similar, and was a characteristic of the optimization algorithm. In particular, we found that the size of the basin around ADAM-based optimization was larger than that around vanilla SGD (Section 4.2). A large basin is related to a large margin, as small changes in the weight vector will not affect the training error, and perhaps could have some implications for generalization error. In our experiments, however, we did not observe better generalization error for ADAM than SGD. Questions for potential future research are *why* the shapes of the loss functions around different local minima found by the same algorithm are so similar, and what the practical implications of this are.

## REFERENCES

- Pierre. Baldi. Linear learning: Landscapes and algorithms. In *In Advances in neural information processing systems.*, pp. 65–72, 1989.
- Pierre Baldi and K. Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks*, 2:53–58, 1989.
- Pierre. Baldi and Zhiqin. Lu. Complex-valued autoencoders. *Neural Networks*, 33:136–147, 2012.
- Leon Bottou. Stochastic gradient learning in neural networks. In *Proceedings of Nuero-Nimes*, 1991.
- Alan J. Bray and David S. Dean. The statistics of critical points of gaussian fields on large-dimensional spaces. In *Physics Review Letter*, 2007.
- C. G Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *Journal of Applied Mathematics*, 6:76–90, 1970.
- John C. Butcher. Coefficients for the study of runge-kutta integration processes. *Society for Industrial and Applied Mathematics*, 3:185–201, 1963.
- Anna Choromanska, Mikael Henaf, Michael Mathieu, Gerard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. *arXiv preprint arXiv:1406.2572*, 2015.
- Yann N. Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv preprint arXiv:1406.2572*, 2014.
- Murat A. Erdogdu and Andrea Montanari. Convergence rates of sub-sampled newton methods. In *Proceedings of the Neural Information Processing Systems (NIPS)*, 2015.
- Yan V. Fyodorov and Ian Williams. Replica symmetry breaking condition exposed by random matrix calculation of landscape complexity. *Journal of Statistical Physics.*, 129:1081–1161, 2007.
- Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, 2010.
- Ian J. Goodfellow, Oriol Vinyals, and Andrew M. Saxe. Qualitatively characterizing neural network optimization problems. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.
- Ernst Hairer, Nrsett Syvert P., and Gerhard Wanner. *Solving Ordinary Differential Equations I Nonstiff*. Springer, 1987.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *arXiv preprint arXiv:1502.01852*, 2015.
- S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–1780, 1997.
- Kenji Kawaguchi. Deep learning without poor local minima. *arXiv preprint arXiv:1605.07110*, 2016.
- Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.
- Alex Krizhevsky. Learning multiple layers of features from tiny images. In *MSc thesis, Univesity of Toronto*, 2009.
- Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.
- James Martens. Deep learning via hessian-free optimization. In *Proceedings of the International Conference of Machine Learning (ICML)*, 2010.

- Yurii Nesterov. A method of solving a convex programming problem with convergence rate  $o(1/\sqrt{k})$ . *Soviet Mathematics Doklady*, 27:372–376, 1983.
- Giorgio Parisi. Probabilistic line searches for stochastic optimization. *arXiv preprint arXiv:0706.0094*, 2016.
- B.T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.
- Herbert Robbins and Sutton Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400–407, 1951.
- O. Russakovsky, H. Deng, J. adn Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. In *arXiv preprint arXiv:1409.0575*, 2014.
- Andrew M Saxe, James L McClelland, and Surya. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *In International Conference on Learning Representations.*, 2014.
- Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.
- Daniel Soudry and Yair Carmon. No bad local minima: Data independent training error guarantees for multilayer neural networks. *arXiv preprint arXiv:1605.08361*, 2016.
- Ilya Sutskever, James Martens, George Dahl, and Geoffery Hinton. On the importance of momentum and initialization in deep learning. In *Proceedings of the International Conference of Machine Learning (ICML)*, 2013.
- Grzegorz Swirszcz, Wojciech Marian Czarnecki, and Razvan Pascanu. Local minima in training of deep networks. In *International conference on artificial intelligence and statistics*, 2016.
- Tijmen Tieleman and Geoffery Hinton. Rmsprop gradient optimization. In *Neural Networks for Machine Learning slide: [http://www.cs.toronto.edu/tijmen/csc321/slides/lecture\\_slides lec6.pdf](http://www.cs.toronto.edu/tijmen/csc321/slides/lecture_slides lec6.pdf)*, 2012.
- Matthew D. Zeiler, Graham W. Taylor, and Rob Fergus. Adaptive deconvolutional networks for mid and high level feature learning. In *International Conference on Computer Visio*, 2011.

## A SUPPLEMENTARY MATERIALS

### A.1 3D VISUALIZATION

Goodfellow et al. (2015) introduced the idea of visualizing 1D subspace of the loss surface between the parameters. Here, we propose to visualize loss surface in 3D space through interpolating over three and four vertices.

**Linear Interpolation** Given two parameters  $\theta_1$  and  $\theta_2$ ,

$$\theta_i = \alpha\theta_1 + (1 - \alpha)\theta_2, \quad \forall \alpha \in [0, 1]. \quad (2)$$

**Bilinear Interpolation** Given four parameters  $\theta_0$ ,  $\theta_1$ ,  $\theta_2$ , and  $\theta_3$ ,

$$\phi_i = \alpha\theta_1 + (1 - \alpha)\theta_2 \quad (3)$$

$$\varphi_i = \alpha\theta_3 + (1 - \alpha)\theta_4 \quad (4)$$

$$\theta_j = \beta\phi_i + (1 - \beta)\varphi_i \quad (5)$$

for all  $\alpha \in [0, 1]$  and  $\beta \in [0, 1]$ .

**Barycentric Interpolation** Given four parameters  $\theta_0$ ,  $\theta_1$ , and  $\theta_2$ , let  $d_1 = \theta_1 - \theta_0$  and  $d_2 = \theta_2 - \theta_0$ . Then, the formulation of the interpolation is

$$\phi_i = \alpha d_1 + \theta_0 \quad (6)$$

$$\varphi_i = \alpha d_2 + \theta_0 \quad (7)$$

$$\theta_j = \beta\phi_i + (1 - \beta)\varphi_i \quad (8)$$

for all  $\alpha \in [0, 1]$  and  $\beta \in [0, 1]$ .

### A.2 OPTIMIZATION METHODS

#### A.2.1 STOCHASTIC GRADIENT DESCENT

In many deep learning applications both the number of parameters and quantity of input data points can be quite large. This makes the full evaluation of  $U(\theta)$  be prohibitively expensive. A standard technique for alleviating computational loads is to apply a stochastic approximation to the gradient Robbins & Monro (1951). More precisely, one approximates  $U$  by a subset of  $n$  data points, denoted by  $\{\sigma_j\}_{j=1}^N$  at each timestep:

$$U^n(\theta) = \frac{1}{n} \sum_{j=1}^n \ell(\theta, \mathbf{x}_{\sigma_j}) \simeq \frac{1}{N} \sum_{i=1}^N \ell(\theta, \mathbf{x}_i) = U(\theta) \quad (9)$$

Of course this approximation also carries over to the gradient, which is of vital importance to optimization techniques:

$$\nabla U^n(\theta) = \frac{1}{n} \sum_{j=1}^n \nabla \ell(\theta, \mathbf{x}_{\sigma_j}) \simeq \nabla U(\theta) \quad (10)$$

This method is what is commonly called *Stochastic Gradient Descent* or *SGD*. So long as the data is distributed nicely the approximation error of  $U^n$  should be sufficiently small such that not only will SGD still behave like normal GD, but its wall clock time for to converge should be significantly lower as well.

Usually one uses the stochastic gradient rather than the true gradient, but the inherent noisiness must be kept in mind. In what follows we will always mean the stochastic gradient.

#### A.2.2 MOMENTUM

In order to alleviate both noise in the input data as well as noise from stochasticity used in computing quantities one often maintains history of previous evaluations. In order to only require one extra variable one usually stores variables of the form

$$\mathbb{E}[F]_t = \alpha F_t + \beta \mathbb{E}[F]_{t-1}. \quad (11)$$

where  $F_t$  is some value changing over time and  $\mathbb{E}[F]_t$  is the averaged quantity.

An easy scheme to apply this method to is to compute a rolling weighted average of gradients such as

$$\mathbb{E}[\mathbf{g}]_t = (1 - \alpha)\mathbf{g}_t + \alpha\mathbb{E}[\mathbf{g}]_{t-1}$$

but there will be other uses in the future.

### A.2.3 PERTINENT METHODS

With the aforementioned tools there are a variety of methods that can be constructed. We choose to view these algorithms as implementations of Explicit Euler on a variety of different vector fields to remove the ambiguity between  $\eta$  and  $\mathbf{g}_t$ . We therefore can define a method by the vector field  $\mathcal{X}_t$  that explicit Euler is applied to with a single  $\eta$  that is never changed.

**SGD with Momentum (SGDM)** By simply applying momentum to  $\mathbf{g}_t$  one obtains this stabilized stochastic version of gradient descent:

$$\mathcal{X}_t = -\mathbb{E}[\mathbf{g}]_t. \quad (12)$$

This is the most fundamental method that is used in practice and the basis for everything that follows.

**Adagrad** Adagrad rescales  $\mathcal{X}_t$  by summing up the squares of all previous gradients in a coefficient-wise fashion:

$$\mathcal{X}_t = -\frac{\mathbf{g}_t}{\sqrt{\sum_{i=1}^t \mathbf{g}_i^2 + \epsilon}}. \quad (13)$$

Here  $\epsilon$  is simply set to some small positive value to prevent division-by-zero. In the future we will neglect this term in denominators because it is always necessary.

The concept is to accentuate variations in  $\mathbf{g}_t$ , but because the denominator is monotonically nondecreasing over time this method is doomed to retard its own progress over time. The denominator can also be seen as a form of momentum where  $\alpha$  and  $\beta$  are both set to 1.

**Rmsprop** A simple generalization of ADAGrad is to simply allow for  $\alpha$  and  $\beta$  to be changed from 1. In particular one usually chooses a  $\beta$  less than 1, and presumably  $\alpha = 1 - \beta$ . Thus one arrives at a method where the effects of the distance history are diminished:

$$\mathcal{X}_t = -\frac{\mathbf{g}_t}{\sqrt{\mathbb{E}[\mathbf{g}^2]_t}}. \quad (14)$$

**Adadelta** Adadelta adds another term to RMSprop in order to guarantee that the magnitude of  $\mathcal{X}$  is balanced with  $\mathbf{g}_t$  Zeiler et al. (2011). More precisely it maintains

$$\frac{\mathcal{X}_t}{\sqrt{\mathbb{E}[\mathcal{X}_t^2]}} = -\frac{\mathbf{g}_t}{\sqrt{\mathbb{E}[\mathbf{g}_t^2]}} \quad (15)$$

which results in the following vector field:

$$\mathcal{X}_t = -\frac{\sqrt{\mathbb{E}[\mathcal{X}_t^2]}}{\sqrt{\mathbb{E}[\mathbf{g}_t^2]}} \mathbf{g}_t. \quad (16)$$

and  $\eta$  is set to 1.

**ADAM** By applying momentum to both  $\mathbf{g}_t$  and  $\mathbf{g}_t^2$  one arrives at what is called ADAM. This is often considered a combination of SGDM + RMSprop,

$$\mathcal{X}_t = c_t \frac{\mathbb{E}[\mathbf{g}]_t}{\sqrt{\mathbb{E}[\mathbf{g}^2]_t}}. \quad (17)$$

$c_t = \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$  is the *initialization bias correction term* with  $\beta_1, \beta_2 \in [0, 1)$  being the  $\beta$  parameters used in momentum for  $\mathbf{g}$  and  $\mathbf{g}^2$  respectively. Initialization bias is caused by the history of the momentum variable being initially set to zero.

### A.3 RUNGE KUTTA

Runge-Kutta methods Butcher (1963) are a broad class of numerical integrators categorized by their truncation error. Because the ordinary differential equations Runge-Kutta methods solve generalize gradient descent, our augmentation is quite straightforward. Although our method applies to all explicit Runge-Kutta methods we will only describe second order methods for simplicity.

The general form of second-order explicit Runge-Kutta on a time-independent vector field is

$$\theta_{t+1} = \theta_t + (a_1 \mathbf{k}_1 + a_2 \mathbf{k}_2)h \quad (18)$$

$$\mathbf{k}_1 = \mathcal{X}(\theta_t) \quad (19)$$

$$\mathbf{k}_2 = \mathcal{X}(\theta_t + q_1 h \mathbf{k}_1) \quad (20)$$

where  $a_1, a_2$ , and  $q_1$  are parameters that define a given Runge-Kutta method. Table 3 refers to the parameters used for the different Runge-Kutta variants we use in our experiments.

Table 3: The coefficients of various second order Runge-Kutta methods Hairer et al. (1987)

Method Name	$a_1$	$a_2$	$q_1$
Midpoint	0	1	$\frac{1}{2}$
Heun	$\frac{1}{2}$	$\frac{1}{2}$	1
Ralston	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{4}$

#### A.3.1 AUGMENTING OPTIMIZATION WITH RUNGE KUTTA

For a given timestep, explicit integrators can be seen as a morphism over vector fields  $\mathcal{X} \rightarrow \bar{\mathcal{X}}^h$ . For a gradient  $\mathbf{g}_t = \nabla_{\theta} U$  we can solve a modified RK2 gradient  $\bar{\mathbf{g}}_t$  in the following fashion:

$$\theta_{t+1} = \theta_t + \bar{\mathbf{g}}_t h = \text{Advect}_{\mathbf{g}}^{rk2}(\theta, h) \quad (21)$$

rearranged with respect to  $\bar{\mathbf{g}}_t$

$$\bar{\mathbf{g}}_t = \frac{\text{Advect}_{\mathbf{g}}^{rk2}(\theta, h) - \theta_t}{h} \quad (22)$$

$$= \frac{\theta_t + (a_1 \mathbf{k}_1 + a_2 \mathbf{k}_2)h - \theta_t}{h} \quad (23)$$

$$= (a_1 \mathbf{k}_1 + a_2 \mathbf{k}_2). \quad (24)$$

If we simply substitute the gradient  $\mathbf{g}_t$  with  $\bar{\mathbf{g}}_t$  one obtains an RK2-augmented optimization technique.

### A.4 EXPERIMENTS WITH RUNGE-KUTTA INTEGRATOR

The results in Figure 9 illustrates that, with the exception of the Midpoint method, stochastic Runge-Kutta methods outperform SGD. ‘‘SGD x2’’ is the stochastic gradient descent with twice of the learning rate of ‘‘SGD’’. From the figure, we observe that the Runge-Kutta methods perform even better with half the number of gradient computed by SGD. The reason is because SGD has the accumulated truncated error of  $O(h)$  while second-order Runge-Kutta methods have the accumulated truncated error of  $O(h^2)$ .

Unfortunately, ADAM outperforms ADAM+RK2 methods. We speculate that this is because the way how ADAM’s renormalization of input gradients in conjunction with momentum eliminates the value added by using our RK-based descent directions.

### A.5 EFFECTS OF BATCH-NORMALIZATION AND EXTREME INITIALIZATIONS

The neural networks are typically initialized with very small parameter values (Glorot & Bengio, 2010; He et al., 2015). Instead, we trained NIN with exotic initializations such as initial parameters drawn from  $\mathcal{N}(-10.0, 0.01)$  or  $\mathcal{N}(-1.0, 1.0)$  and observe the loss surface behaviours. The results are shown in Figure 11. We can see that NIN without BN does not train at all with any of these initializations. Swirszcz et al. (2016) mentioned that bad performance of neural networks trained with these initializations are due to finding a bad local minima. However, we see that loss surface region around these initializations are plateau <sup>1</sup> rather than a bad local minima as shown in Figure 11b. On

<sup>1</sup>We used same initializations as (Swirszcz et al., 2016) but we trained different neural networks with SGD on a different dataset. We used NIN and CIFAR10 and Swirszcz et al. (2016) used smaller neural network and MNIST.

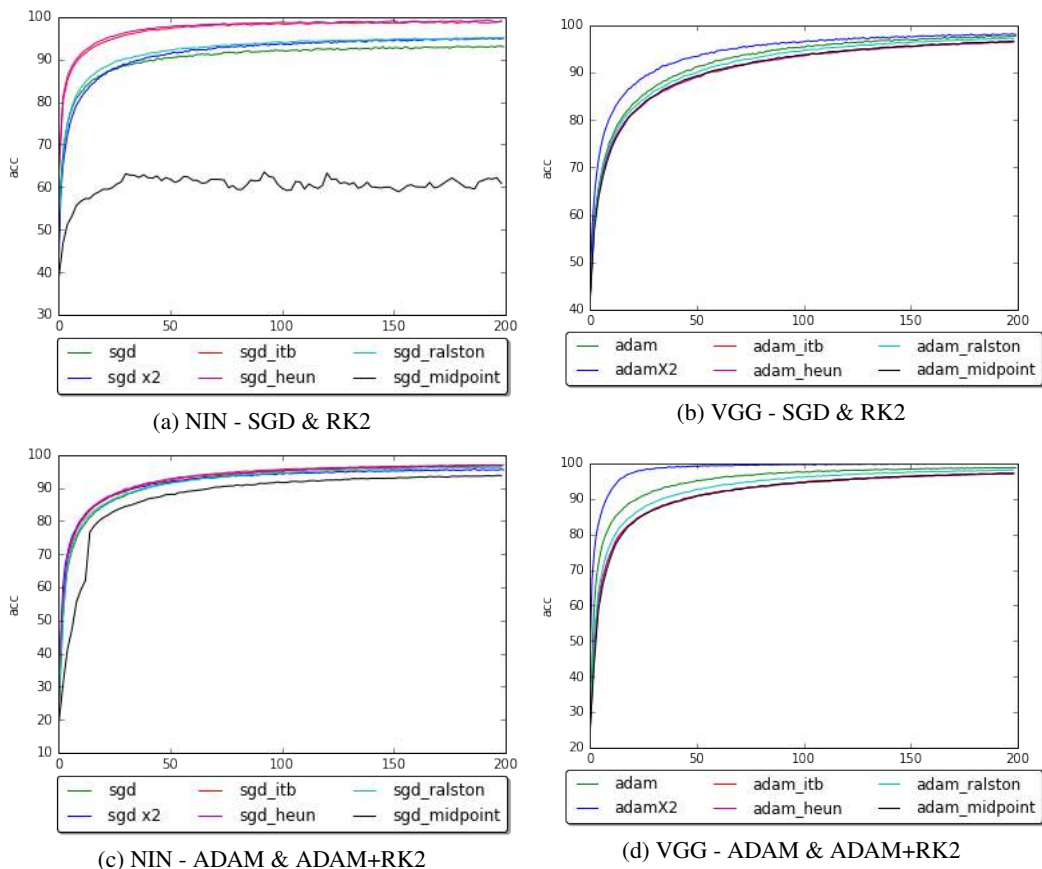


Figure 9: Training accuracy curve

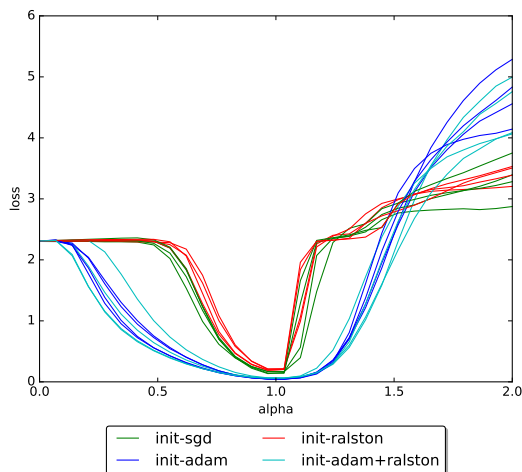


Figure 10: Interpolation between initial points to final points upto  $\alpha = 2$  in Equation 2.

the other hand, NIN with BN does train slowly over time but finds a local minima. This implies that BN redeems the ill-posed loss surface (plateau region). Nevertheless, the local minima it found was not good as when the parameters were initialized with small values. However, it is not totally clear whether this is due to difficulty of training or due to falling in a bad local minima.



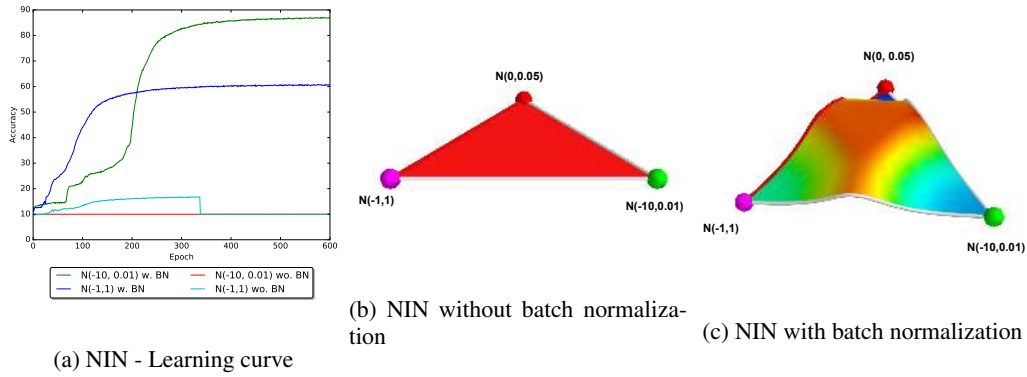


Figure 11: NIN trained from different initializations.

### A.6 SWITCHING OPTIMIZATION METHODS

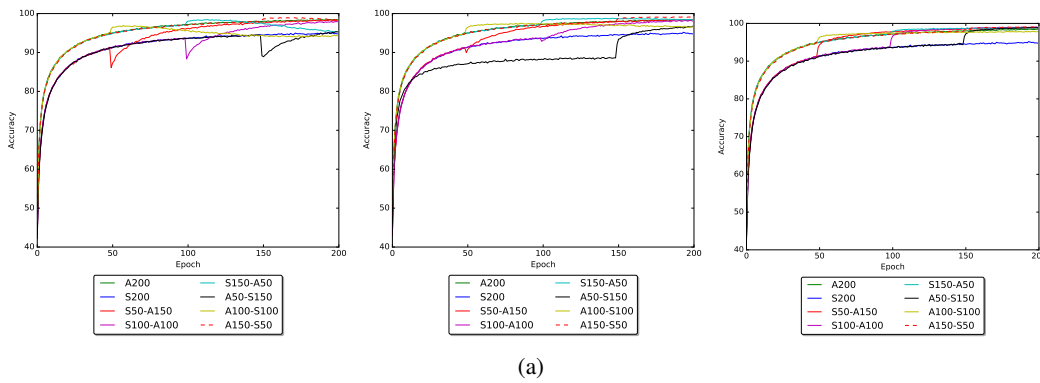
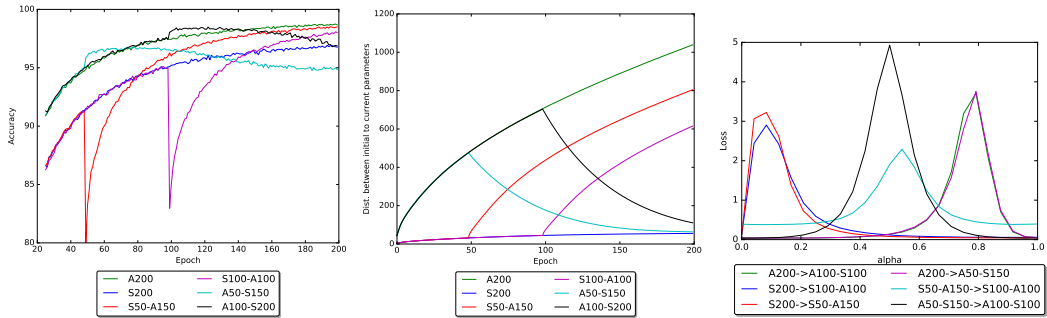
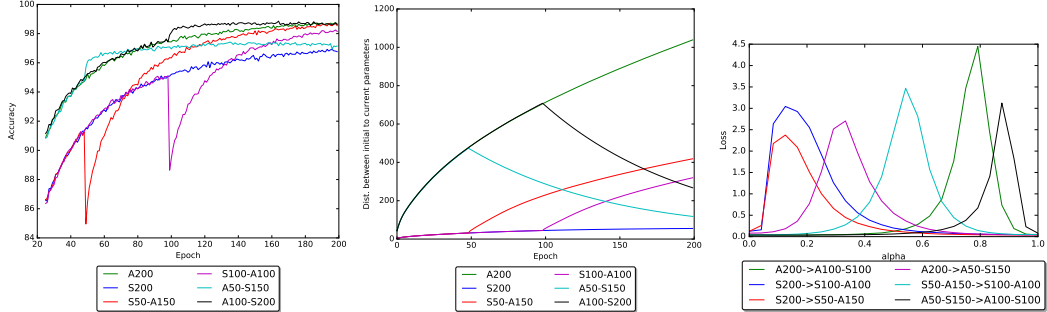


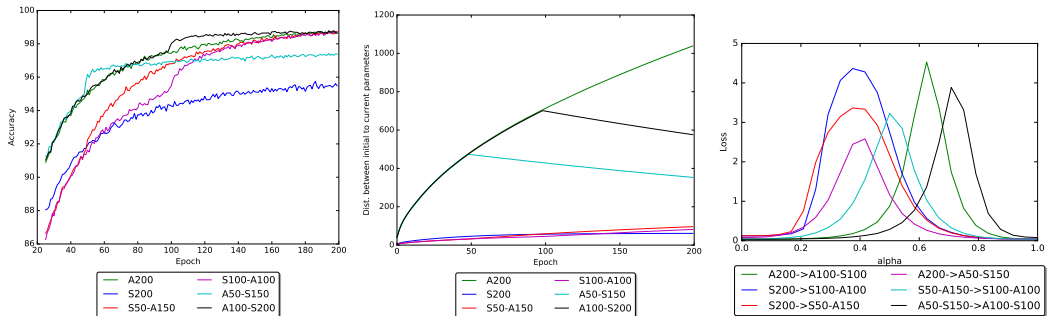
Figure 12: NIN - Learning curve when switching methods from SGD to ADAM and visa versa at epoch 50 and 100. Learning rate switched from SGD (ADAM) to ADAM (SGD) at (left) 0.001 (0.1) to 0.1 (0.001), (middle) 0.001 (0.1) to 0.05, (0.001), and (right) 0.001 (0.1) to 0.01 (0.001).



(a) The learning rates is set to 0.001 and 0.05 for ADAM and SGD in the beginning, and then switched it to 0.05, 0.001 for SGD and ADAM.

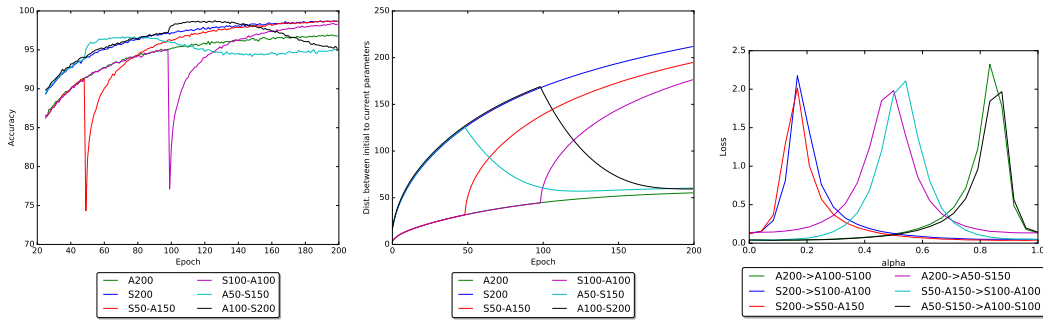


(b) The learning rates is set to 0.001 and 0.05 for ADAM and SGD in the beginning, and then switched it to 0.05, 0.0005 for SGD and ADAM .

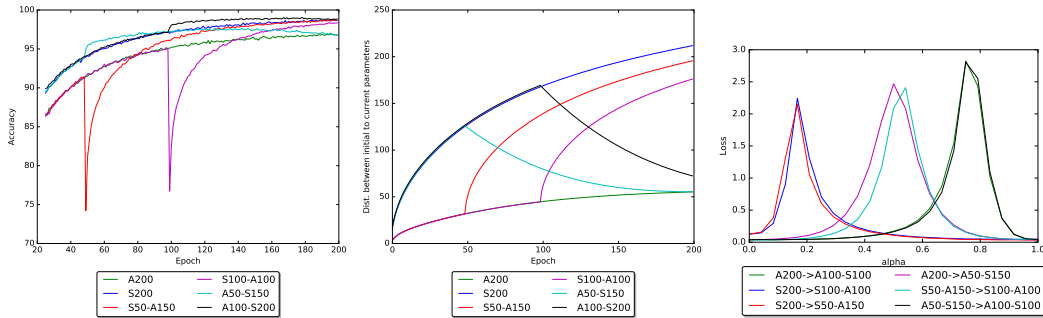


(c) The learning rates is set to 0.001 and 0.05 for ADAM and SGD in the beginning, and then switched it to 0.01, 0.0001 for SGD and ADAM.

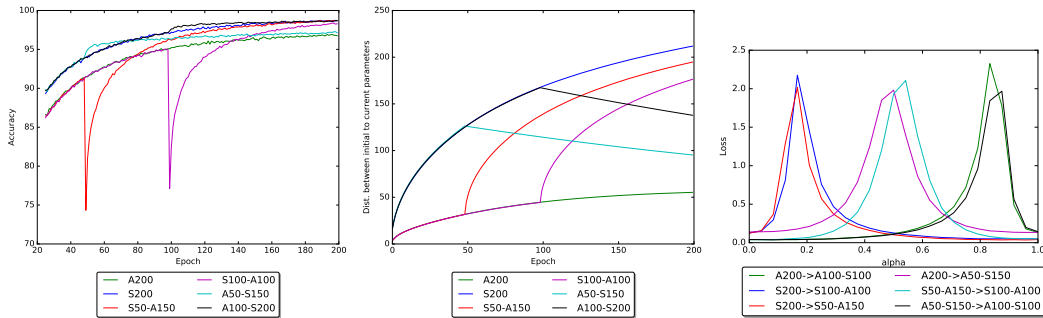
Figure 13: VGG - Switching methods from SGD to ADAM and ADAM to SGD at epoch 50 and 100. Zoomed in version (Left). Distance between initial weights to weights at each epoch (Middle). The interpolation between different convergence parameters (Right). Each figure shows the results of switching methods at different learning rate. We label the switch of methods in terms of ratio. For instance, S100-A100 as trained with SGD in the first 100 epoch and switched to ADAM for the rest of the epoch.



(a) Learning rate is not required for Adadelta. Learning rate is set to 0.05 for SGD in the beginning, and then switched it to 0.1.

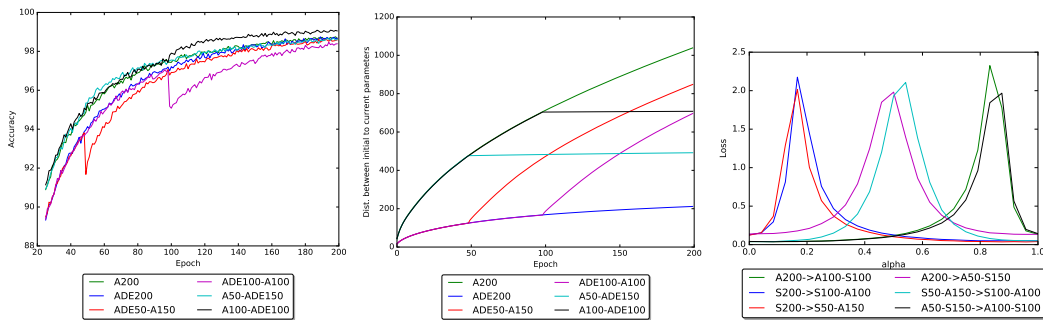


(b) Learning rate is set to 0.05 for SGD.



(c) Learning rate is set to 0.05 for SGD in the beginning, and then switched it to 0.01.

Figure 14: VGG - Switching methods from SGD to Adadelta and Adadelta to SGD at epoch 50 and 100. Zoomed in version (Left). Distance between initial weights to weights at each epoch (Middle). The interpolation between different convergence parameters (Right). Each figure shows the results of switching methods at different learning rate. We label the switch of methods in terms of ratio. For instance, S50-A50 as trained with SGD in the first 100 epoch and switched to Adadelta for the rest of the epoch.



(a) Learning rate is not required for Adadelata. Learning rate is set to 0.05 for SGD in the beginning, and then switched it to 0.1.

Figure 15: VGG - Switching methods from ADAM to Adadelata and Adadelata to ADAM at epoch 50 and 100. Zoomed in version (Left). Distance between initial weights to weights at each epoch (Middle). The interpolation between different convergence parameters (Right). Each figure shows the results of switching methods at different learning rate. We label the switch of methods in terms of ratio. For instance, S50-A50 as trained with SGD in the first 100 epoch and switched to Adadelata for the rest of the epoch.