Estimating Second Order Effects In Gradient Descent with the Directional Gradient Delta

Dan Goldberg

Department of Computer Science
University of Toronto
dang.goldberg@mail.utoronto.ca

Abstract

In this paper I develop an extension to first order gradient descent methods that explicitly adjusts for second order effects with relatively little overhead and works with deep neural networks. This *Directional Gradient Delta* (DGD) method approximates the directional derivative of each dimension of the gradient using typical SGD observables, and then adds this approximation to the next gradient step in proportion to the next step's cosine similarity with the direction of the approximation. With a single fairly robust hyperparameter this is shown to decrease both training and test loss faster than standard momentum-SGD in several configurations of feedforward neural networks trained on the Boston Housing dataset. This paper is a final research project for Fall 2017 University of Toronto course CSC2515H.[1]

1 Introduction

In recent years the explosion of research in deep learning has led to breakthroughs in image recognition, machine translation, reinforcement learning, meta learning, and a host of other complex machine intelligence tasks.

The power of deep neural networks is in part due to the idea of exploiting *depth*, that is, many layers of hidden units that can use the features learned by lower layers as inputs in order to learn more abstract features. However, it turns out that fully connected feedforward networks with more than a few layers become notoriously difficult to optimize. Wide and deep neural networks can often show pathological curvature where there are long narrow valleys to traverse. A first-order gradient descent algorithm will usually bounce from side to side and so will make its way through these valleys extremely slowly.

There are first-order optimization methods that attempt to improve speed to convergence by using the first and/or second moments of the distribution of gradients along a descent path. A simple and effective technique is *momentum* [Qian, 1999] which adds an exponential moving average of the gradient's first moment to the next gradient descent step. This can be thought of as a heavy ball rolling along a hilly terrain; the heavier the ball, the less that small distortions in the surface will change it's path. Another very popular technique is *Adam* [Kingma and Ba, 2014] which adapts the update step in inverse proportion to a moving average of the variance of gradient.

Second-order techniques not only take the gradient of the objective into consideration, but also the curvature of the function which can be described using the matrix of second derivatives, the *Hessian matrix*. The classic Hessian-based method is *Newton's Method*, though this and any other other approach that directly uses the Hessian matrix is usually infeasible for large neural networks since the Hessian grows in $O(n^2)$ for n parameters. Networks can easily have parameters in the orders of

[1] The code for this project is available at `https://github.com/dan-goldberg/DGDelta`

Algorithm 1 *Directional Gradient Delta*, the proposed algorithm for stochastic optimization. Here we estimate the directional derivative of the gradient for the direction of the last step and use this to make an adjustment to the next gradient descent step. The size of the adjustment is made proportional to the alignment of the current gradient with the last step, calculated as the signed Euclidean norm of the projection of the current gradient onto the span of the last gradient.

Require: θ_n: current position
Require: $\nabla f(\theta_n)$: gradient at current position
Require: θ_{n-1}: position before last step
Require: $\nabla f(\theta_{n-1})$: gradient at last position
Require: γ: desired weight of the second-order adjustment
$\Delta g \leftarrow \nabla f(\theta_n) - \nabla f(\theta_{n-1})$ (the change in the gradient after the last step)
$\tilde{\mathbf{u}} \leftarrow \theta_n - \theta_{n-1}$ (the last step)
$\Delta s \leftarrow ||\tilde{\mathbf{u}}||_2$ (the size of the last step)
$\mathbf{d} \leftarrow \frac{\Delta \mathbf{g}}{\Delta s}$ (a gross approximation of the directional derivative of the gradient at θ_{n-1})
$p \leftarrow \frac{\nabla f(\theta_n)^\top \cdot \tilde{\mathbf{u}}}{\Delta s}$ (signed norm of current gradient vector projected onto the last gradient direction)
$\mathbf{a} \leftarrow p\mathbf{d}$ (vector of second-order adjustments)
$\mathbf{z} \leftarrow \nabla f(\theta_n) + \gamma \mathbf{a}$ (add weighted second-order adjustments to gradient to get parameter update)
return $\mathbf{z}$

millions, so storing the Hessian is problematic. This, plus the need to compute and invert it makes directly using the Hessian seemingly impossible for most neural networks.

2 Preliminaries

Newton's Method Second-order methods locally approximate the objective function around a point θ as a quadratic function:

$$f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2} p^\top B p \tag{1}$$

where B is the *curvature matrix*. Newton's method directly uses the Hessian as the curvature matrix in this formula. Minimizing eq 1 within a region reasonably close to θ can be far more efficient then gradient descent, which only identifies the direction of steepest descent at θ. Moreover, Newton's Method is *scale-invariant* and is robust to regions of *pathological curvature*, where first-order methods can be completely stymied.

Directional Derivatives The gradient of the objective function, $\nabla f(\theta)$, defines the tangent plane at θ. Using it as a first-order approximation to the function, as gradient descent does, the gradient can be thought of as the anticipated change in the objective function associated with movement in input space. Of course this approximation gets worse as one travels further from θ so the step size has to be small to have a useful approximation.

The derivatives of a function are initially defined by the standard basis for the input space - so the anticipated change in the function is defined relative to a positive change in each predefined axis. It is possible, however to define a *directional derivative* which characterizes the rate of change in the function given movement in one particular direction, *not necessarily along the standard basis axes*. Where a gradient denotes a rate of change relative to multiple axes, a directional derivative can denote a rate of change relative only to a change in a single quantity - the direction of interest.

More concretely, using the notation in Figure 1 we can characterize the rate of change in the function, w, using the gradient:

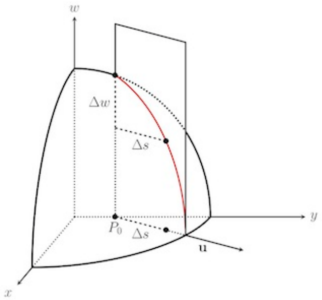


Figure 1: A directional derivative of a function of two variables. Figure copied from [Auroux, 2010] where it was cited as courtesy of John B. Lewis.

$$\nabla w(P_0) = \left(\frac{\partial w}{\partial x}, \frac{\partial w}{\partial y} \right)^\top \tag{2}$$

but if we know of a direction of interest $\mathbf{u}$, we can simply characterize the rate of change with respect to that particular direction:

$$\frac{dw}{ds} = \lim_{\Delta s \to 0} \frac{\Delta w}{\Delta s} \tag{3}$$

Even without knowing the definition of $\mathbf{u}$ in standard basis coordinates, knowing $\frac{dw}{ds}$ would be useful when moving along the span of of $\mathbf{u}$. On the other hand, it would not be at all useful for predicting the rate of change in w due to movement along $\mathbf{u}_\perp$.

3 Hessian Free Optimizaion

James Martens along with Ilya Sutskever introduced the deep learning community to Hessian-Free optimization, which considers second-order effects in iterative optimization without directly using the Hessian matrix [Martens, 2010, Martens and Sutskever, 2012]. This method leverages the fact that the Hessian can be used indirectly in the linear Conjugate Gradient algorithm via a Hessian-vector product that can be approximated using finite-differences:

$$H(\theta)d = \lim_{\epsilon \to 0} \frac{\nabla f(\theta + \epsilon d) - \nabla f(\theta)}{\epsilon} \tag{4}$$

Note that this Hessian-vector product is the directional derivative of the gradient for the direction of vector d. Unfortunately this finite difference formula cannot be used directly to approximate $H(\theta)d$ in deep learning due to numerical issues that come with applying finite difference to highly nonlinear functions like neural networks.

Instead, Martens proposed using forward-propagation to arrive at the Hessian-vector product approximation. This method is similar to back-propagation in that it uses the chain rule to sequentially calculate values for nodes in a computational graph. Unlike back-prop, forward-prop in this context propagates intermediate finite difference-approximated directional derivative values from inputs to outputs through the network, and comes out with numerically stable approximation of $H(\theta)d$. By finding the Hessian-vector product one can then use the conjugate gradient algorithm to minimize the local quadratic approximation to the curve at θ within some dampened region, making extremely efficient progress.

4 Directional Gradient Delta Optimization

Here I introduce a new optimization method that can be used in deep learning, adjusts for the curvature of the objective function, and produces little overhead for both computation and memory. This method, which for lack of a better term I'm calling the *Directional Gradient Delta* or DGD for short, is closely related to the Hessian-Free optimization approach developed by Martens, but I believe at its core it is still a first-order optimization technique. After all, it is an extension that runs on top of typical SGD-based optimizers like momentum-SGD, using observations from the standard course of a gradient descent path.

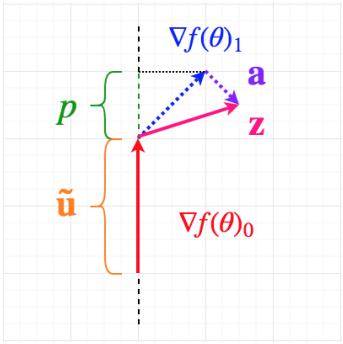


Figure 2: An example of one DGD adjustment for the second step in a gradient descent path. The vectors are to scale for learning rate $\alpha = 1.0$ and adjustment weight $\gamma = 1.0$.

At a high level this method first approximates the directional derivative of the gradient *nearby* the current location. Next it estimates how useful that approximation might be to the current position and trajectory, and finally makes an adjustment to the next descent step in proportion to that estimated relevancy. The experiments done for this paper were built in Tensorflow [Abadi et al., 2015], and all the machinery just mentioned is integrated with out-of-the-box optimizers like tf.GradientDescentOptimizer, tf.MomentumOptimizer, and

tf.AdamOptimizer; all that was required was to insert this functionality between opt.compute_gradients() and opt.apply_gradients(). The specific procedure is outlined explicitly in Algorithm 1.

The Directional Delta The approximation of the directional derivative of the gradient calculated in steps 1-4 of Algorithm 1 is a gross approximation at best. It can be considered a finite difference approximation, though instead of pushing ϵ towards zero we are simply taking the difference of the gradients after a normal gradient descent step, and dividing by the Euclidean norm of that step. Since this is such a monstrous approximation, I've called it the *directional delta* instead of directional derivative (this is the directional delta of the gradient, hence the name Directional Gradient Delta method).

Such an approximation will inevitably be terrible in most cases, and in theory will get worse with increasing step size. The saving grace for this method is that it's impact decreases as the gradients of subsequent steps become more orthogonal; why this happens will be explained below.

Projecting onto $\tilde{\mathbf{u}}$ Armed with some idea of how the curve is changing in the direction of the last step, $\tilde{\mathbf{u}}$, we then project the gradient calculated at the current position onto the span of that direction, as on line 5 of Algorithm 1. The signed Euclidean norm of this projection is the cosine similarity between $\nabla f(\theta_n)$ and $\tilde{\mathbf{u}}$ multiplied by the signed magnitude of the current gradient, or equivalently, the dot product of the current gradient and the unit vector in the direction of $\tilde{\mathbf{u}}$:

$$p = \frac{\nabla f(\theta_n)^\top \cdot \tilde{\mathbf{u}}}{||\tilde{\mathbf{u}}||_2} = \frac{\nabla f(\theta_n)^\top \cdot \tilde{\mathbf{u}}}{\Delta s} = \nabla f(\theta_n)^\top \cdot \mathbf{u} \tag{5}$$

By 'signed' magnitude I simply mean relative to the direction of $\tilde{\mathbf{u}}$ (i.e. positive if the projection points in the same direction and negative if it points in the opposite direction).

The signed magnitude of this projection can be thought of as a measure of how far the next step will take us in that direction, and therefore a measure of how relevant the directional gradient delta is to our next movement. If the gradient at our current position (and hence our trajectory) is orthogonal to the direction of our last step then the directional delta that we calculated for that direction will be useless. Conversely, if our current trajectory takes us in exactly the same direction as our last step did then the directional delta will be informative - how informative it is will depend on the distance traveled from the centre of the approximation along the third-order character of the objective function in the current neighbourhood. In that case, if we assume the third derivatives are constant then in fact the directional delta will be a perfect approximation of the directional derivative (though an assumption unlikely to be true for neural networks).

Applying Directional Gradient Delta Adjustments In step 6 of Algorithm 1 the directional gradient delta is multiplied by the anticipated movement in the direction of interest to get the anticipated change in the gradient after that movement:

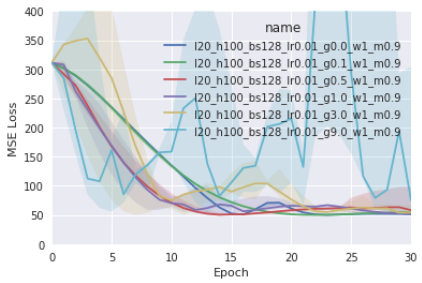
$$\mathbf{a} = p\mathbf{d} = \Delta\hat{s}\frac{\Delta g}{\Delta s} = \Delta\hat{g} \tag{6}$$

In step 7 we finally update the original descent step by adding the anticipated change in the gradient, weighted by a hyperparameter γ. Assuming a non-zero projection magnitude, in the case of a directional slope becoming more negative the adjustment will make the step in that direction larger. Conversely, the directional step would shrink if the directional delta shows the slope becoming less negative.

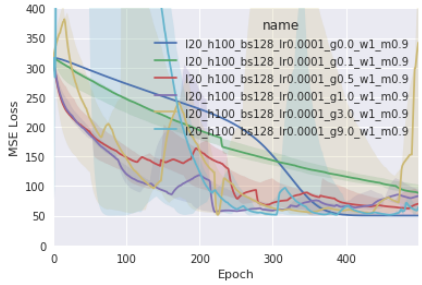
5 Experiments

This method was implemented in Tensorflow sitting on top of out-of-the-box optimizer classes. The performance was evaluated on a set of neural network architectures that varied in depth and width, trained on the Boston Housing toy dataset via scikit-learn [Pedregosa et al., 2011].

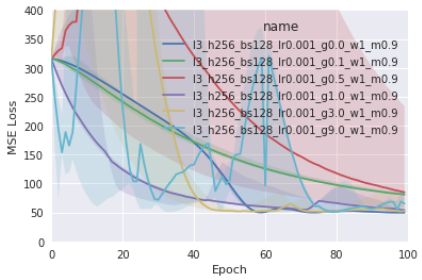
The base optimizers used were MomentumOptimizer, StochasticGradientOptimizer, and AdamOptimizer. Neural networks with hidden modules of shape (hidden layers x hidden units per layer) 20x100, 3x256, 1x1024, and 2x64 were trained with each optimizer and both the learning rate α and



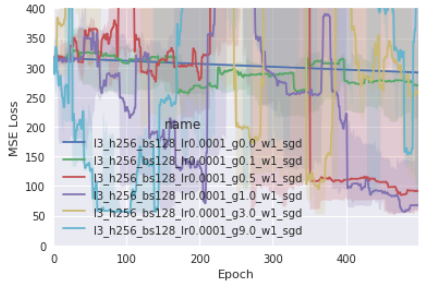
(a) 20×100 neural network - 0.01 lr - Momentum



(b) 20×100 neural network - 0.0001 lr - Momentum



(c) 3×256 neural network - 0.001 lr - Momentum



(d) 3×256 - 0.0001 lr - SGD

Figure 3: Some examples of *success* cases for DGD.

the DGD adjustment weight γ varied over a grid search. These results were plotted plotted separately for each architecture-optimizer-learning rate combination. All the plots can be found in the Appendix.

Some important static hyperparameters were the following: parameters were initialized by sampling from a random normal distribution $\sim \mathcal{N}(0, \frac{1}{200})$; the momentum optimizer used a momentum of 0.9; the Adam optimizer used $\beta_1 = 0.9$ and $\beta_2 = 0.99$; all experiments used a batch size of 128.

6 Results

There are 48 plots in total each with 6 experiments (1 control with $\gamma = 0.0$ and 5 different active γ values). I will highlight a few interesting examples here, and leave the rest for you to investigate in the Appendix in figures 6-53.

The control experiment for each plot is the dark blue line, denoted by the name containing 'g0.0' in the legend (meaning a γ setting of 0.0). The other lines in each respective plot only differ in their setting of γ. Each plot has a unique combination of architecture-optimizer-learning rate so that the relative performance of different settings of γ can be compared with respect to that configuration.

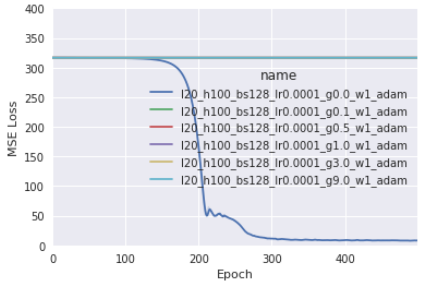
Success Modes In Figure 3 we see some cases where DGD essentially converged to a local minimum significantly faster than the base optimizer alone.

Failure Modes In Figure 4 we see some cases where DGD either didn't learn (4a), diverged completely (4c), or converged to a local minimum that was significantly worse than the optimum found by the base optimizer alone (4b and 4c).

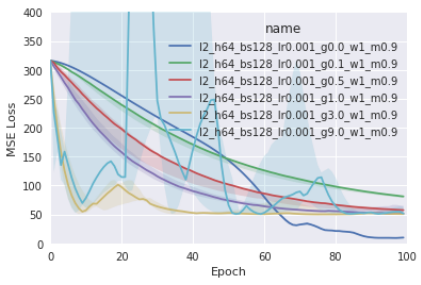
7 Discussion

7.1 Analysis of Results

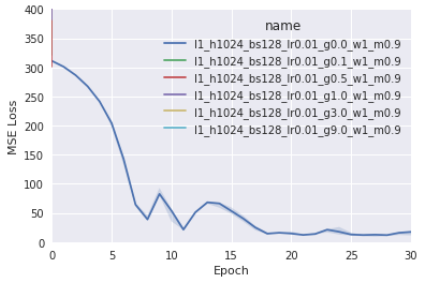
There are many dimensions of these experiments and even more hyperparameters that were not changed during the course of this research, so the conclusions drawn from the results observed cannot be made with a high degree of confidence. Due to time constraints I was not able to give a



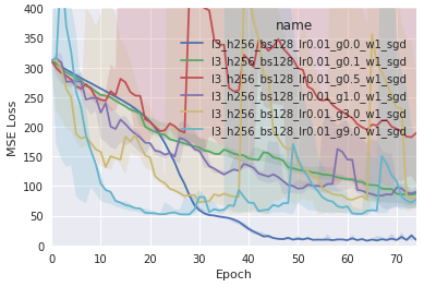
(a) 20×100 neural network - 0.0001 lr - Adam



(b) 2×64 neural network - 0.001 lr - Momentum



(c) 1×1024 neural network - 0.01 lr - Momentum



(d) 3×256 - 0.01 lr - SGD

Figure 4: Some examples of *failure* cases for DGD.

methodical treatment over the many variables that may have a huge impact on the performance of any optimization method, and more serious research must be left to a future investigation.

That being said, the experiments presented in this paper do give a window into how DGD performs in some fairly common configurations, and at the very least the results do show potential for DGD to outperform the base optimizers used in a non-trivial breadth of optimization problems. Here I will present a brief discussion on DGD and how the major variables of these experiments affected the performance.

Base Optimizer The three base optimizers used presented three very distinct modes of DGD. Out of the three, the Momentum base optimizer showed the best performance with DGD, the algorithm was able to increase the speed to near-convergence by almost double in many cases. This appears to be a significant improvement over what is currently an extremely popular optimizer.

Momentum is theoretically well suited to help DGD perform well because the momentum term added to the gradient encourages movement in more parallel directions from step to step, relative to simple SGD. This means that any wild perturbations in the gradient from step to step are somewhat muted for standard SGD, and the same applies to DGD. Any directional gradient delta from one step to another is noisy at best, and momentum may prevent too wild a change towards an orthogonal direction that may be caused by a large DGD adjustment. From the results seen using SGD it appears that these adjustments cause quite wild swings in loss, so momentum appears to act as a regularizer on DGD. The convergence of Momentum-DGD seems not only relatively the most stable, but also the most robust to different settings of γ, an added plus.

As just mentioned, SGD experiments show very unstable learning, though even with that instability the DGD optimization is able to discover areas of parameter space that give a very low value of the cost function. That adding momentum is able to curtail this volatility is a clue that the DGD adjustments are taking the trajectory far from where SGD would step.

Using Adam, on the other hand, was a complete failure in each and every experiment run. The Adam-DGD experiments showed essentially no learning whatsoever, whereas Adam alone almost always would converge to a good optimum.

Network Shape Network shape typically has an enormous impact on learning efficacy in general, as well as which optimizer performs best. DGD performed best on the two deeper networks (hidden

layer depth 20 and 3), and did not perform well on the shallow networks (depth 2 and 1). The width of the networks don't appear to have much of an impact, though truthfully the experiments designed did not make for a rigorous evaluation of width or depth on with other variables controlled.

What is clear is that for cases in which the base optimizer could break out of local optima around a MSE of 50 and go down to below 10, the DGD extension was unable to break out of that local optima. Deeper networks are known to cause optimization getting stuck in local optima and underfitting, so the affect isn't as apparent with the 20-layer network as it is with the shallower networks.

Learning Rate The learning rate parameter for the most part does not seem to have any major effects on how the DGD optimizer relates to the base optimizer alone. Empirically, the learning rate affects how γ relates to performance in that higher γ values shows more volatility and a higher incidence of divergence as the learning rate decreases.

The learning rate also appears significant for determining whether the base optimizer and the DGD optimizer converge to the same minimum, or if the base optimizer can find a lower optimum to settle in. As noted above, the DGD extension was unable to break into the sub-10 MSE optima even when the base optimizers were able to. It would very interesting to experiment with annealing the learning rate to see if the DGD method could match the optima found by the other base optimizers when run at low enough learning rates to converge to the lower optimum; this is left for a further investigation due to time constraints.

Gamma The γ parameter seems fairly robust to changes in the other dimensions of these experiments. Values of 0.5 and 1.0 seemed to consistently perform well and were not especially volatile, though moreso 1.0. When set to 0.1 the experiments usually show not much difference from the base optimizer, or worse performance. The high values of 3.0 and 9.0 were very interesting in that they would either show incredibly unstable training or diverge completely, or they would find relatively near-optimal minima very quickly. Even when finding these local optima the experiments run with high γ values would sometimes jump out of these optima within a few steps. This extra volatility is certainly not unexpected, though still it is pleasantly surprising to see these high settings perform so well in some configurations. More experimentation should be done to determine a relationship with the learning rate or other dimensions of a training task, though if I had to choose one I would recommend starting with $\gamma = 1.0$.

7.2 Pathological Curvature

One explanation for the volatility of DGD as well as it's ability to progress so much faster for deeper networks may be that it, in accounting for second-order effects, can break out of narrow valleys quite easily.

In figure 5 I present a calculation of what would happen if the direction of the gradient at the current position is opposite to the direction of the gradient from the previous position, as is shown in typical narrow valley examples in the literature. Here we see that that the DGD adjustment pushes the step direction to be almost parallel with the previous step and so the optimizer jumps out of the valley entirely.

This is a significant effect that not only is notable for valleys, but really any local minimum. If the gradient reverses course this diagram shows that DGD pushes it to continue, much like momentum. Unlike momentum, however, there can be some orthogonal effects as well. For the example in Figure 5, $\nabla f(\theta)$ has increased in the $\tilde{\mathbf{u}}_{\perp}$ direction, so the directional gradient delta includes a push in that direction as well.

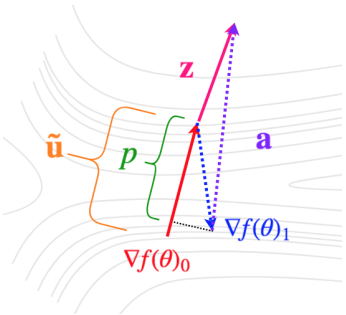


Figure 5: A possible effect of Directional Gradient Delta in a narrow valley. The vectors are approximately to scale for $\alpha = 1.0$ and $\gamma = 1.0$. Please note that the contours are added only for effect, and that these gradients are constructed instead of being the result of evaluating a function.

8 Limitations & Further Directions

The DGD method seems to be limited in that it has not been shown to work with Adam, and one could infer that it might be equally ineffective with other adaptive gradient descent optimizers. The idea of adaptive learning rate is surely a good one, as seen from the efficacy of Adam alone in the experiments for this paper. This is a serious limitation if not resolved. The idea of adaptive learning rate is a good one, and controlling the size of γ dynamically may lead to some improvements. Since the directional gradient delta is only relevant to a one dimensional span in parameter space it seems that one could not use any past directional gradient deltas to inform a future one, though it may be possible to change the gamma based on the last step size; as noted early on, the DGD approximation to the directional derivative of the gradient would ostensibly get better as the distance between the two positions shrinks.

The other major issue observed was the inability for any DGD extension to converge to any sub-10 MSE minima, whereas the base optimizers were able to given an appropriate combination of network architecture and learning rate. It is possible that an annealed learning rate or annealed γ would allow the DGD method to reach the same minima, and this seems like a great next direction for future research.

One important question is how hurtful the volatility observed would be to using DGD in practice. The results displayed show the median since the high fluctuations in loss diverge relatively frequently. Since the goal of optimization is to find the lowest point it would seem that in most cases it wouldn't matter how high the optimizer goes as long as it discovers a low point along the way. On the other hand, one might object that it is harder to know when to stop training when the optimizer is so volatile, and in fact it hampers the ability to use early stopping, an important form of implicit regularization and big source of time savings.

Another interesting extension would be to consider a directional gradient delta between the current position and positions from more than one step in the past. As happens with SGD in narrow valleys it is very possible for a 2-step distance to be less than a 1-step distance, and so in these cases it may be beneficial to include the 2-step (or more-step) DGD adjustments. In fact I did build this functionality into the code, though was unable to fully experiment with it due to the time constraint.

As already touched upon, the experiments run were extremely limited in the dimensions varied. In deep learning there are a plethora of knobs one can tune and buttons one can push, and it is possible that there are many factors that might affect the performance of this method, both positively and negatively. Some that come to mind as the most important are: the variable initialization scheme, the momentum hyperparameter, the Adam hyperparameters, a learning rate schedule, the minibatch size, a regularization term on the objective function, batch normalization, dropout, gradient noise, residual connections, and other popular architectures like CNNs and RNNs.

I would also like to note that only one very small dataset was used in these experiments, and only regression was performed. Using other datasets on other tasks with other kinds of loss functions may show very different results. I would have liked to pursue a more thorough investigation but was constrained in my time available since this was a school course project.

9 Conclusion

In this paper I introduced the *Directional Gradient Delta* (DGD) method for gradient descent optimization. This method uses a one-step memory of the last gradient and position to estimate the curvature of the objective function surface in the direction of the last step. This curvature estimate is then used to adjust the original gradient descent step in proportion to the alignment between the last step direction and the current gradient direction. This method can sit on top of existing popular optimizers like SGD and Momentum-SGD, though was found not to work with Adam. From experiments training deep neural networks on the Boston Housing toy dataset it was shown that in some cases DGD can improve speeds to near-convergence over SGD or Momentum-SGD by as much as double, albeit with extreme volatility.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Denis Auroux. Multivariable calculus, Fall 2010. URL `https://ocw.mit.edu`.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`.

James Martens. Deep learning via hessian-free optimization. In Johannes Fürnkranz and Thorsten Joachims, editors, *ICML*, pages 735–742. Omnipress, 2010. URL `http://dblp.uni-trier.de/db/conf/icml/icml2010.html#Martens10`.

James Martens and Ilya Sutskever. Training deep and recurrent networks with hessian-free optimization. In *Neural networks: Tricks of the trade*, pages 479–535. Springer, 2012.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1): 145–151, 1999. URL `http://dblp.uni-trier.de/db/journals/nn/nn12.html#Qian99`.

Appendix

All plots displayed show the test set error by epoch, during training. Each plot has a legend describing the major experiment parameters, though the lines displayed only vary in their γ parameter, as denoted by the number to the right of the 'g' in the legend name. For example, the name 'l20_h100_bs128_lr0.1_g0.0_w1_m0.9' means layers = 20, hidden units per layer = 100, batch size = 128, learning rate = 0.1, gamma = 0.0, window = 1 (you can ignore this), and momentum optimizer with momentum = 0.9. The legend names were made like this to allow for easy slicing of data into different groupings, and due to time constraints I unfortunately could not clean up the names for the plots displayed here.

The blue lines in each plot have gamma = 0.0, and these are the control experiments where there is no DGD adjustment added to the descent step. The lines show the median value, with the shaded region showing one standard deviation from the mean.

(please see next page)

20 Layers x 100 Nodes - Momentum

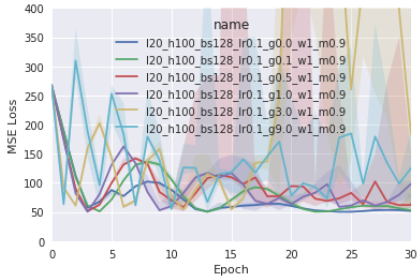


Figure 6: $\alpha = 0.1$

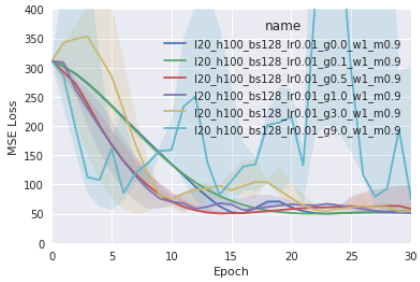


Figure 7: $\alpha = 0.01$

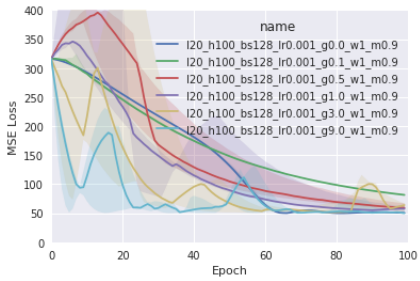


Figure 8: $\alpha = 0.001$

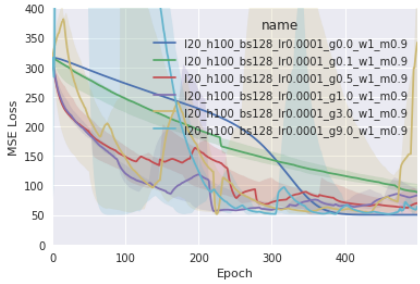


Figure 9: $\alpha = 0.0001$

20 Layers x 100 Nodes - SGD

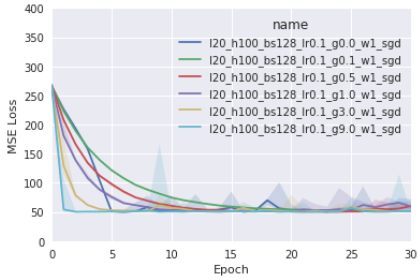


Figure 10: $\alpha = 0.1$

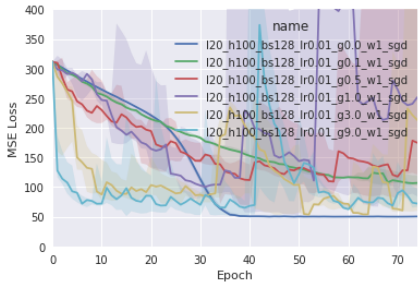


Figure 11: $\alpha = 0.01$

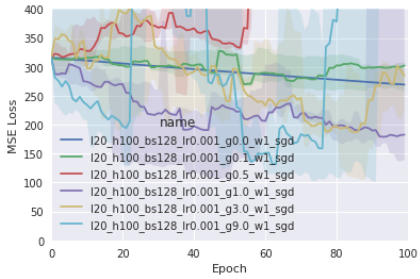


Figure 12: $\alpha = 0.001$

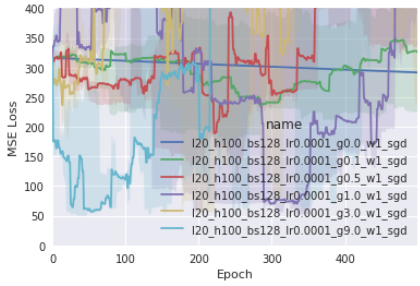


Figure 13: $\alpha = 0.0001$

20 Layers x 100 Nodes - Adam

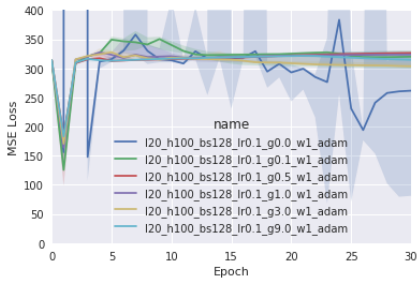


Figure 14: $\alpha = 0.1$

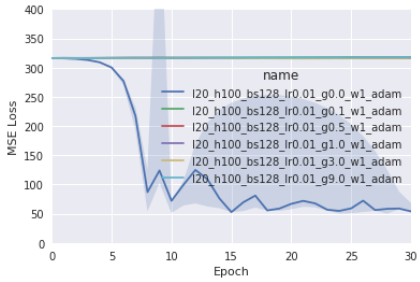


Figure 15: $\alpha = 0.01$

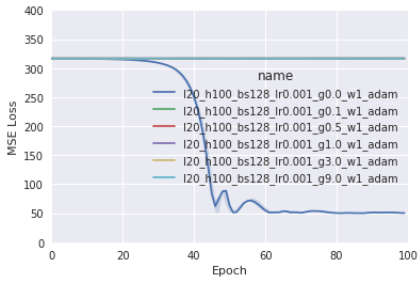


Figure 16: $\alpha = 0.001$

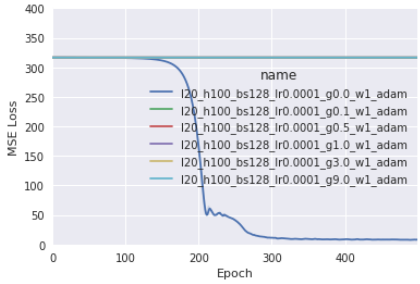


Figure 17: $\alpha = 0.0001$

3 Layers x 256 Nodes - Momentum

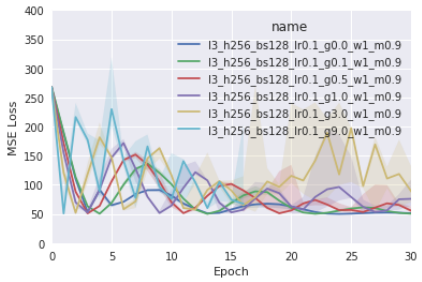


Figure 18: $\alpha = 0.1$

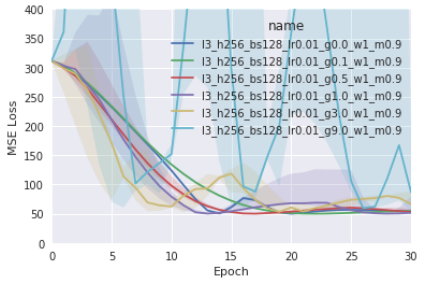


Figure 19: $\alpha = 0.01$

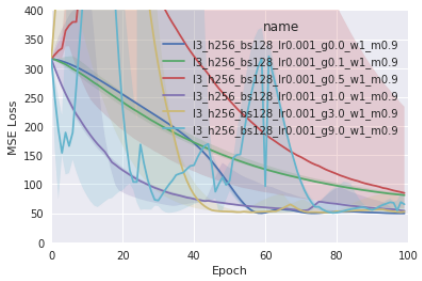


Figure 20: $\alpha = 0.001$

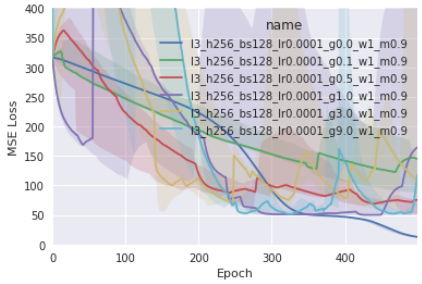


Figure 21: $\alpha = 0.0001$

3 Layers x 256 Nodes - SGD

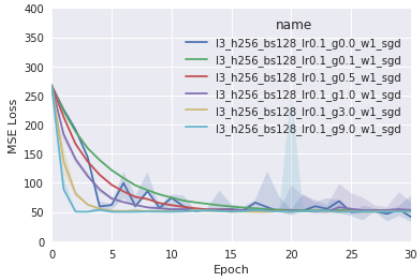


Figure 22: $\alpha = 0.1$

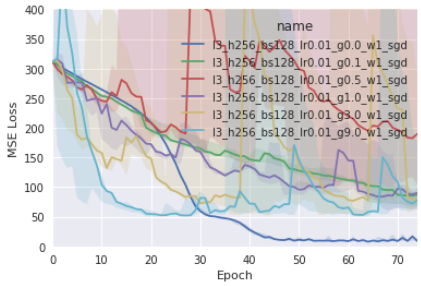


Figure 23: $\alpha = 0.01$

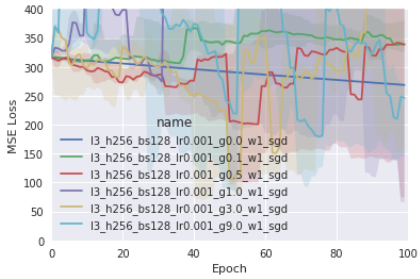


Figure 24: $\alpha = 0.001$

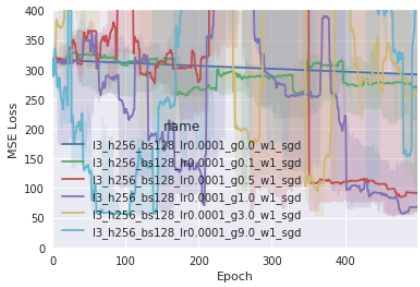


Figure 25: $\alpha = 0.0001$

3 Layers x 256 Nodes - Adam

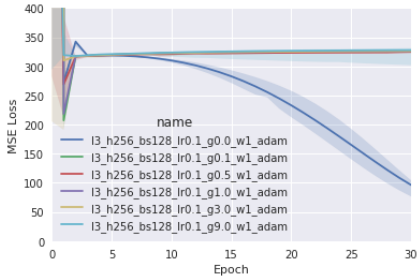


Figure 26: $\alpha = 0.1$

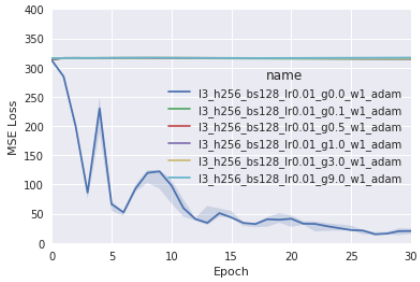


Figure 27: $\alpha = 0.01$

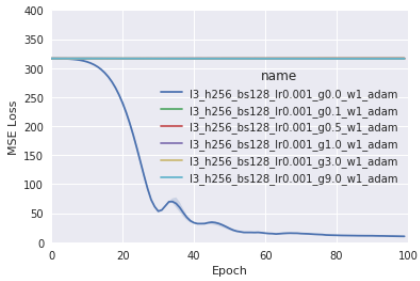


Figure 28: $\alpha = 0.001$

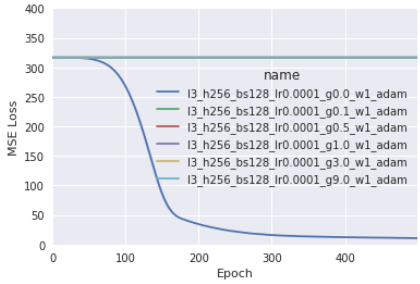


Figure 29: $\alpha = 0.0001$

2 Layers x 64 Nodes - Momentum

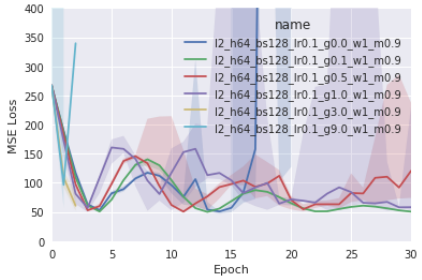


Figure 30: $\alpha = 0.1$

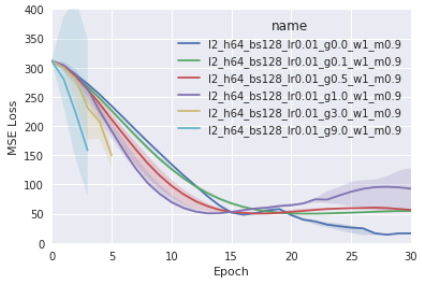


Figure 31: $\alpha = 0.01$

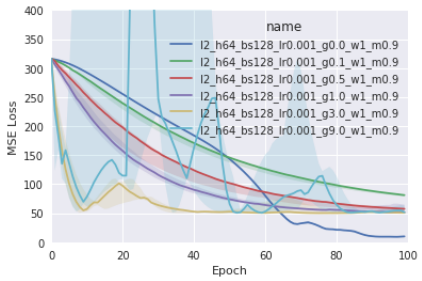


Figure 32: $\alpha = 0.001$

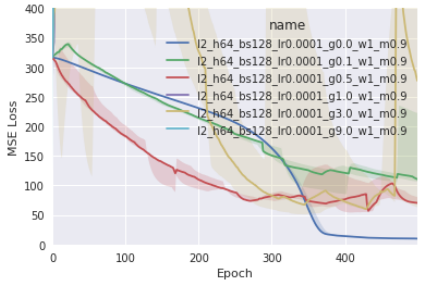


Figure 33: $\alpha = 0.0001$

2 Layers x 64 Nodes - SGD

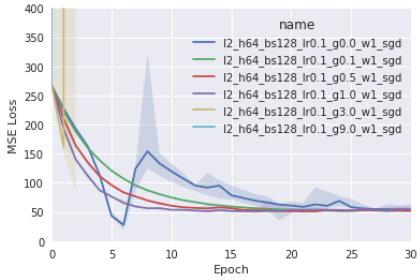


Figure 34: $\alpha = 0.1$

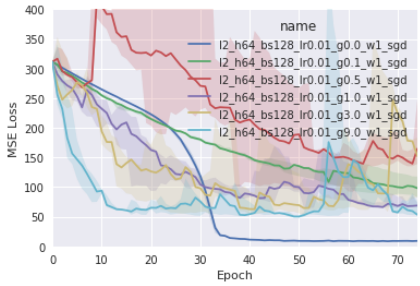


Figure 35: $\alpha = 0.01$

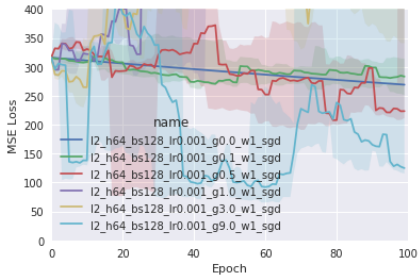


Figure 36: $\alpha = 0.001$

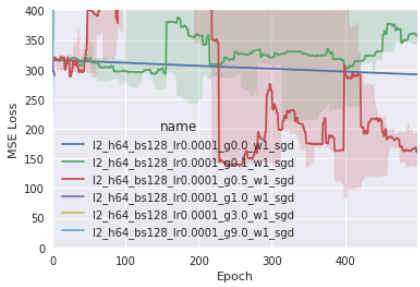


Figure 37: $\alpha = 0.0001$

2 Layers x 64 Nodes - Adam

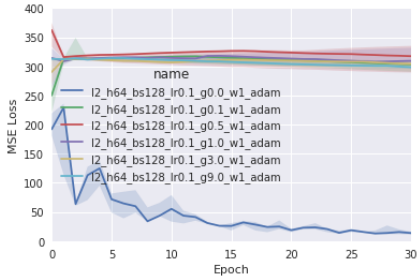


Figure 38: $\alpha = 0.1$

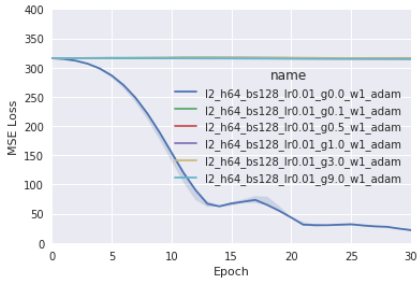


Figure 39: $\alpha = 0.01$

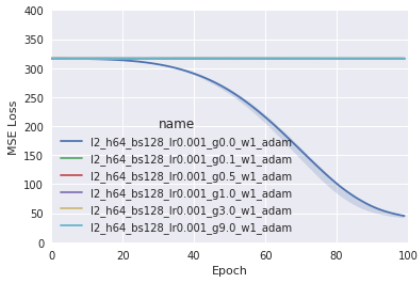


Figure 40: $\alpha = 0.001$

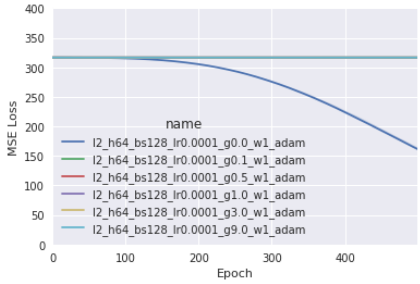


Figure 41: $\alpha = 0.0001$

1 Layers x 1024 Nodes - Momentum

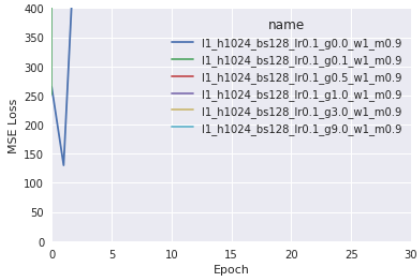


Figure 42: $\alpha = 0.1$

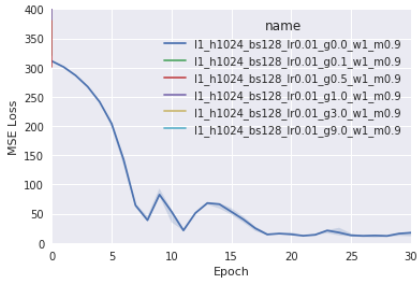


Figure 43: $\alpha = 0.01$

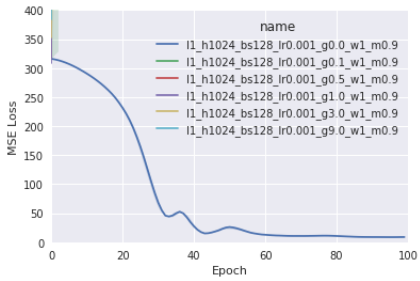


Figure 44: $\alpha = 0.001$

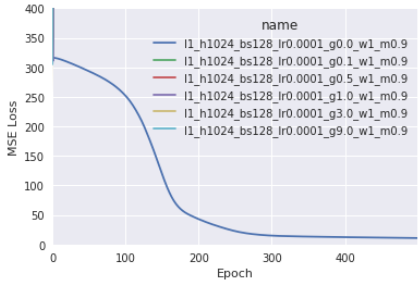


Figure 45: $\alpha = 0.0001$

1 Layers x 1024 Nodes - SGD

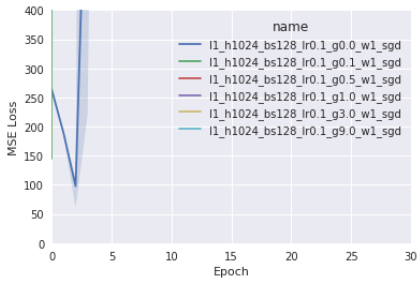


Figure 46: $\alpha = 0.1$

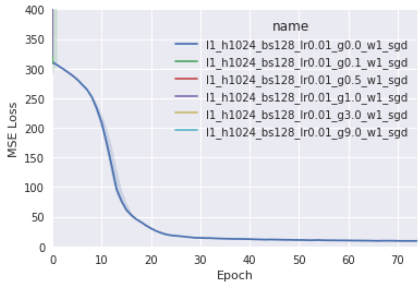


Figure 47: $\alpha = 0.01$

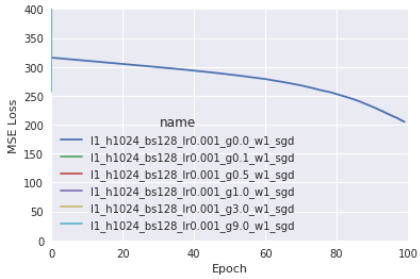


Figure 48: $\alpha = 0.001$

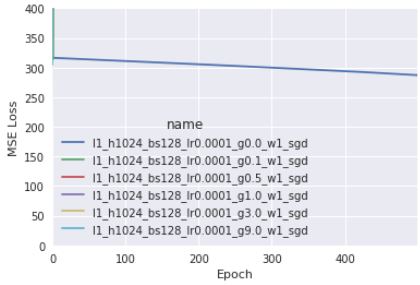


Figure 49: $\alpha = 0.0001$

1 Layers x 1024 Nodes - Adam

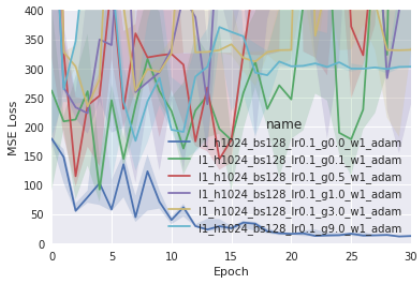


Figure 50: $\alpha = 0.1$

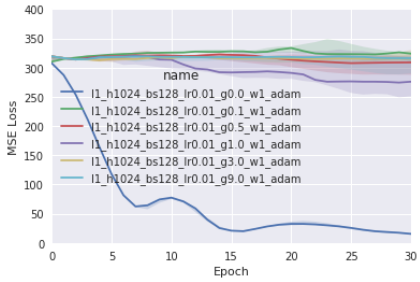


Figure 51: $\alpha = 0.01$

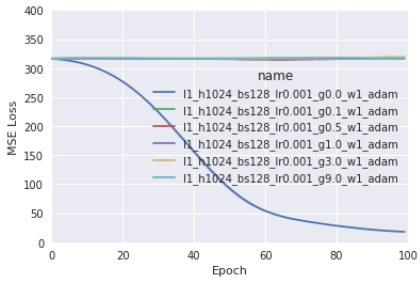


Figure 52: $\alpha = 0.001$

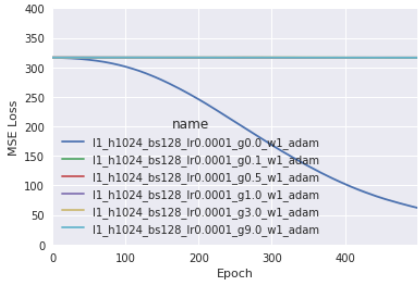


Figure 53: $\alpha = 0.0001$