Why Your Neural Network is Still Singular and What You Can Do About It

Jakub Dworakowski
Computer Science
Stanford University
jakub@stanford.edu

Pablo Rodriguez Bertorello
Computer Science
Stanford University
prodrig1@stanford.edu

Abstract

We investigate the effects of neural network regularization techniques. First, we reason formally through the effect of dropout and training stochasticity on gradient descent. Then, we conduct classification experiments on the ImageNet data set, as well as regression experiments in the OneNow Reinforcement Learning data set. A network layer's weight matrix is quantified via Singular Value Decomposition and Conditioning ratios. Our regression network appeared to be well conditioned. However, we find that learning for large-scale classification applications is likely to be capped by poor conditioning. We propose approaches that may prove breakthroughs in learning, providing early evidence. We introduce a gradient perturbation layer, a method to maximize generalization experimentally. Our numerical analysis showcases the opportunity to introduce network circuitry compression, relying on the principal components of a layer's weights, when conditioning peaks. Generally, we propose conditioning as an objective function constraint.

Index Terms – Neural Network, Gradient Descent, Singular Value Decomposition, Conditioning, Classification, Regression, ImageNet, Reinforcement Learning, Regularization

1. Introduction

The current stage of development of machine learning algorithms is characterized by leaps and bounds in engineering application. However, creating successful networks has become more of an art than a science as often developers have little understanding how or why they work.

For better model generalization, in deep learning, it is popular to apply empirical techniques for their regularization benefits. For example: dropout [4], batch normalization [5], and training batch sizing [17]

This is often done with little formal reasoning, with limited time invested to understand the interactions between the aforementioned techniques and other key design choices: loss function, architecture, network sizing.

The goal of this paper is to develop mathematical reasoning and intuition for the analysis of generalization techniques. Note that our goal is not to showcase instances of high generalization, but rather to distill pitfalls and exploitable concepts.

The object of study is gradient descent as typically applied to learn model parameters: dropout, vanilla update rule, momentum updates, and momentum [15]. We analyze it both in classification and regression applications.

We find important relationships between learning generalization and quantifiable characteristics of a neural layer weight matrix. As a bi-product of this work, we propose novel approaches: 1) A Gradient Perturbation layer to boost model generalization, 2) A method to compress network circuitry while in training, 3) General formulation of objective functions with a conditioning term for regularization.

Section 3 introduces image classification and reinforcement learning regression data sets. In Section 4, we provide a brief introduction to training neural networks, application of neural networks, and our approach to neural learning analysis in 4.2. Please refer to Section 5.1 for formal mathematical reasoning about key generalization techniques, and 5.2 for numerical analysis of said techniques. Experimental results for image classification are in Section 5.3, and for deep reinforcement learning in Section 5.4. Final conclusions are in Section 6

2. Related Work

To the knowledge of the authors, limited research has been conducted to understand the singular values or eigenvalues of neural network weight matrices as they undergo training. In [14] the eigenvalues of the Hessian of a loss function are analyzed. The distribution of eigenvalues is found to have two distinct sets: one concentrated around zero, and another that drives the solution to the system. This implies the network in question is indeed overparameterized, resulting in unnecessary expenditure of time and compute resources.

[6] proposes "Bounded Batch Normalization" layers,

building on the concept of “Singular Value Bounding”. These layers are designed to limit the singular values of parameter matrices, restricting them to live along a narrow band centered at one. Following this approach, state of the art results are achieved on CIFAR10 and CIFAR100 image datasets [8].

In [3] the use of Singular Value Decomposition (SVD) is proposed to factorize weight matrices. Thereafter, the network may be compressed by removing the rows and columns corresponding to the less significant singular values.

In deep learning, it is popular to employ various generalization techniques such as dropout [4] and batch normalization [5], however, little work has been done to understand their impact on the singular values of weight matrices. It is common knowledge that, when used improperly, these techniques can also prevent networks from generalizing. Learning stagnates, loss ceases to decrease multiplying the number of epochs required to train a networks.

3. Data and Software Libraries

In this study ImageNet and OneNow datasets are used. A number of software libraries are developed implemented in Python leveraging open source packages such NumPy [16] and PyTorch [11].

3.1. ImageNet

A subset of ImageNet [2] is used, that is, Tiny ImageNet, from Stanford’s Convolutional Neural Network course (cs231n). It spans 200 classes each with 500 training, 50 validation, and 50 test images. Tiny ImageNet is selected due to ease of access and ability to easily switch between a convolutional neural network (CNN) or full-connected network (FCN) for experimentation. Example images from ImageNet are shown in Figure 1.

Limited pre-processing is applied to the data set. Each image in the dataset is normalized, such that networks do not learn features specific to intrinsic qualities of an image, for example dead pixels or ambient lighting. Images are randomly cropped, re-sized and randomly horizontally flipped further enriching the dataset.

3.2. OneNow

With the goal of maximizing rewards, a reinforcement learning agent is placed in an environment. In the OneNow dataset, the agent is an investment fund, the environment the market price of a hedged portfolio, with buying/selling (aka accumulate/distribute) as the available actions.

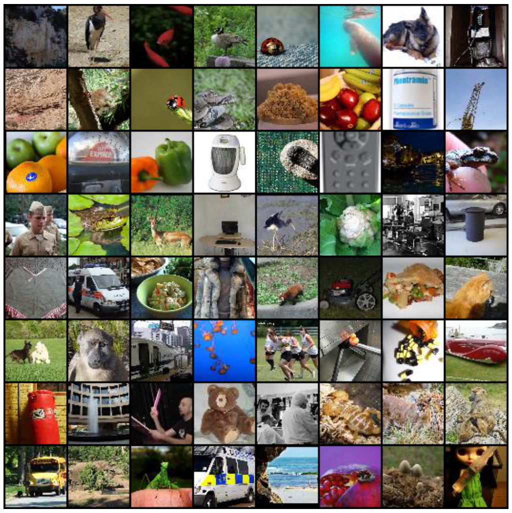


Figure 1. Example Images from ImageNet

Figure 2 illustrates a market environment’s price, normally distributed. A simulated sine environment is super imposed on top of it, to best train the agent in anticipation of ”rare events”. The expected rewards function given this approximation, shown in Figure 3, demonstrates the shortcomings of trying to optimize rewards on the basis of simply buying low and selling high.

Data pre-processing consisted of converting individual agent environments data frames into a single dataset with which a neural network can be trained. The outputs of the network are the reward function for every possible agent action.

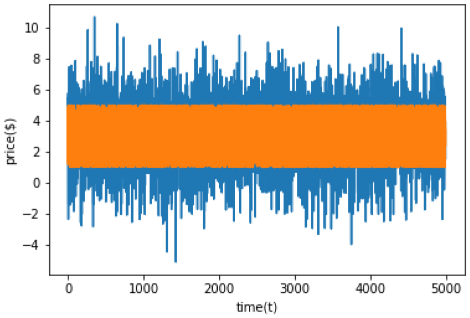


Figure 2. Market Environments: Normal and Sine Distributions

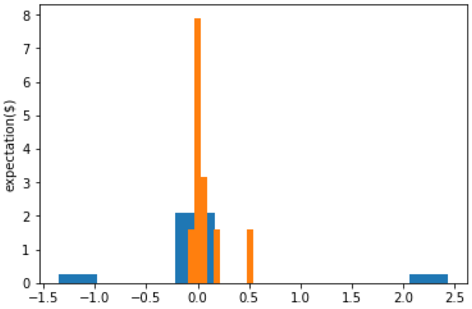


Figure 3. Reward Expectations from Accumulated Actions

3.3. Software Libraries

Image based experiments are implemented in Python, using PyTorch. Visualizations of neural networks are implemented using onnx [10], while data visualization is implemented using matplotlib. To implement these experiments the PyTorch ResNet [13] implementation is forked, which allows to easily add and remove layers, better understanding behaviour. The PyTorch Network Trainer Example [12] is also forked this bootstraps the process of training, the trainer is instrumented with custom analysis code and is extended to include newly developed networks. Several fully connected networks (FCNs) are developed from scratch using PyTorch, variations of a FCN are implemented with dropout applied at various hidden layers. Newly developed layer behaviour analysis code is implemented as well as visualization for layers. A novel "gradient perturbation" layer is also developed to help verify findings.

For reinforcement learning analysis, several libraries are implemented from scratch using Python with NumPy. This includes a Q-Learning algorithm solver for markov decision processes in two different environments. A deep learning network solver, a truncated normal weight initializer. Layer behaviour analysis, and visualizations are also implemented.

4. Technical Methods and Approach

To develop intuition on the behaviour of weight matrices, networks are trained while contemplating the training fundamentals, that is, how the matrices get updated. Building on this intuition, it is then possible to conduct a meaningful empirical analysis of singular values of weight matrices. Finally, these abstract concepts can be applied to specific problems such as image classification and reinforcement learning.

4.1. Network Training

Various techniques are used to support network training, including batch and stochastic gradient descent, and nesterov momentum.

4.1.1 Gradient Descent

Gradient descent is a basic optimization algorithm used to numerically determine a minimum of some given function. This is done by stepping along the negative gradient of the function until a minimum is reached or the algorithm reaches a termination threshold. In machine learning, back propagation is often used to determine an instantaneous numerical exact gradient. The gradient descent algorithm is described in equation 1, where $W_{current}$ is the current weight matrix that is being used to minimize function F and W_{new} is the next iteration (and lr is the learning rate). Note that gradient descent is a first order method to optimization.

$$W_{new} = W_{current} - lr\ \nabla F(W_{current}) \tag{1}$$

4.1.2 Momentum

Momentum is a second order optimization method used together with gradient descent to prevent discontinuities and produce a smoother optimization trajectory via gradient blending. This is illustrated in equation 2, where next gradient is V and $\gamma \in [0, 1]$, acting as a damper smoothing the transition.

$$V_{new} = \gamma\ V_{prev} + (1 - \gamma)\ \nabla F(W_{current}) \tag{2}$$

The gradient in equation 1 is replaced with V_{new} yielding equation 3.

$$W_{new} = W_{current} - lr\ V_{new} \tag{3}$$

4.2. Analysis of Neural Networks

Intuition into weight matrix behaviour at training time is developed via the analysis of singular values, as well as overall matrix conditioning.

4.2.1 Singular Values

To determine the singular values of neural layer weight matrices, "Singular Value Decomposition" (SVD) is used as shown in equation 4. The SVD is a factorization of any $n \times m$ matrix W where U is an $m \times m$ matrix, representing the left singular vectors, V^T is an $n \times n$ matrix, representing the right singular vectors, and Σ is a diagonal $m \times n$ matrix representing the singular values.

$$W = U\ \Sigma\ V^T \tag{4}$$

4.2.2 Conditioning

In this study, emphasis is placed on the Σ matrix of the SVD and it's diagonal entries σ_i. More specifically, this analysis is centered around the condition number of weight matrices. In this context the condition number is defined by equation 5.

$$conditioning = \frac{\sigma_{max}}{\sigma_{min}} \tag{5}$$

The conditioning number indicates how sensitive a function is to changes in inputs and how they effect the outputs. Matrices with high condition numbers are said to be ill-conditioned, an indication of instability, where small changes in inputs will result in large changes in outputs. This could lead to dominant gradients pushing towards local minima amongst other things.

4.3. Applications of Neural Networks

Experiments conducted involve both networks designed to produce a regression output as well as networks producing a classification output.

4.3.1 Image Classification

In classification, the output layer of a neural network is fed into some linear classification stage, for example softmax. The intention is to ultimately output a single value corresponding to the predicted class.

In this study, various implementations of CNNs are forked from PyTorch and worked with, however, these networks proves to be cumbersome to analyze and iterate on, due to deep and complex architecture. Instead, a toy 3 layer FCN is developed with ReLu [9] between every layer; the network is illustrated in Figure 4 note that Gemm refers to a general matrix multiplication.

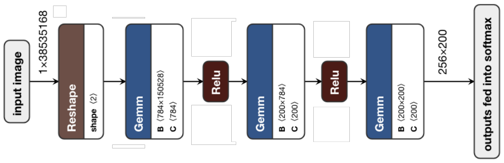


Figure 4. Three Layer Fully Connected Network Architecture

The network implementation is then changed slightly several times, introducing and removing layer types, while also tuning hyper parameters such as the weight decay and momentum. The objective function used is “Cross-Entropy Loss”,

4.3.2 Reinforcement Learning

In the case of Reinforcement Learning, the paradigm is that an agent encounters situations S, takes actions A, and experiences rewards R.

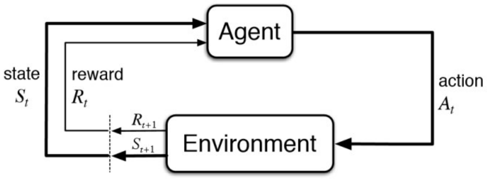


Figure 5. Neuron Operation [7]

The learned reward function is typically stored in a discrete $QTable$, where the key is a state-action tuple, which can be approximated by a neural network.

$$Q : \mathbb{S} \times \mathbb{A} \rightarrow \mathbb{R} \tag{6}$$

The agent’s ultimate goal is to learn an optimal value function V^* that can be used to estimate discounted rewards from taking a best action in a given state.

$$V^*(s) = \underset{a}{max}\ Q^*(s,a)\ \ \forall s \in \mathbb{S} \tag{7}$$

Likewise, an optimal policy function $\pi*$ can be used to choose the best action to take in a given state.

$$\pi^*(s) = \underset{a}{argmax}\ Q^*(s,a)\ \ \forall s \in \mathbb{S} \tag{8}$$

5. Results and Analysis

The mathematics of learning techniques are specifically considered to develop intuition on expected behaviour and how this behaviour manifests itself in empirical results.

5.1. Formal Reasoning

Neuron dropout and training stochasticity are considered through the lens of how they effect the gradient, given observations made a gradient perturbation layer is proposed.

5.1.1 Neuron Dropout

The discussion presented by [4] argues that dropout prevents overfitting by reducing the co-adaptation of feature detectors. Meaning dropout attempts to prevent the occurrence of feature detectors that are only useful when paired with other detectors sensitive to specific features; ultimately developing a network with increased robustness to noise on the input.

To achieve this, while training a network, weights are randomly set to zero with a probability p, while conducting the forward pass of training. Then in the backwards pass, the weights are reset to their original values, and multiplied by $1 - p$ to compensate for the probability their value in contributions to prediction. The argument is a large set of implicit sub-networks are being trained, thereby achieving a form of model averaging. Effectively, this temporarily “mutes” the contribution of individual neurons to the solution.

To build a rigorous understanding of what dropout implements, please refer to the example below.

$$W_{current} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 \end{bmatrix} \tag{9}$$

Now, let’s define $W_{current,P(dropout=0.5)}$, see equation 10.

$$W_{current,P(drop=0.5)} = \begin{bmatrix} 1 & 0 & 0 & 4 & 0 \\ 3 & 0 & 5 & 0 & 7 \\ 8 & 9 & 0 & 11 & 12 \end{bmatrix} \tag{10}$$

This new W is used in place of the original $W_{current}$ as an argument to F shown in equation 11.

$$W_{new} = W_{current,i} - lr \; \nabla F(W_{current,i,P(drop=0.5)}) \tag{11}$$

Equation 11, implies that when using dropout in combination with some gradient descent based optimization method, the gradient used in the update step becomes the gradient corresponding to a different matrix that $W_{current,i}$. This could potentially compromise the stability and ability to converge.

The momentum method implemented by PyTorch is a modified Nesterov Momentum method based on [15] shown in equations 12 and 13, where V is velocity.

$$V = \rho \; V + \nabla F(W_{current,i}) \tag{12}$$

$$W_{new,i} = W_{current,i} - lr \; V \tag{13}$$

This blends the gradients of previous iterations together with the currently calculated gradient.

With the addition of momentum, equations 12, 13 become 14, 15.

$$V = \rho \; V + \nabla F(W_{current,i,P(drop=0.5)}) \tag{14}$$

$$W_{new,i} = W_{current,i,P(drop=0.5)} - lr \; V \tag{15}$$

The combination of equations 14 and 15 results in a blending of gradients by weighting previous gradients together with the current gradient. This produces a solution that is arguably more stable, by eliminating typical discontinuities, as well as those introduced by randomly setting elements of weight matrices to zero.

[4] argues that this will average out, eventually resulting in a more robust solution. However, this clearly comes at the cost of machine learning scientists investing extra effort in tuning the dropout parameters and required compute resources.

5.1.2 Training Stochasticity

In gradient descent, at every update to minimize an objective function $F(x)$, all or some of the training examples may be used to update network weights. For regularization benefit, a subset of the training set may be used for each update [18].

Consider the vector of neural network weights W. Its θ components are:

$$W = [\theta_1, ..., \theta_j, ..., \theta_n] \tag{16}$$

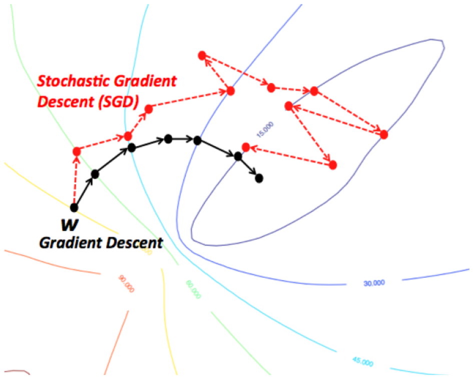


Figure 6. Stochastic vs Batch Gradient Descent[1]

Applying a truncated Taylor Series Expansion, any smooth function may be approximated at an input value c. Now for a component of the weights:

$$\theta_j(x) = \theta_j(c) + (x - c) \; F'(c) \tag{17}$$

Let's call $(c - x)$ the learning rate, lr. The above may be restated iteratively, at iteration k, as:

$$\theta_j(k) = \theta_j(k-1) - lr \; F'_{\theta_j}(k-1) \tag{18}$$

Thus, in stochastic gradient descent, for a single training example, the update rule is often stated as follows (∇ is the Jacobian of the objective function F):

$$\theta_j \leftarrow \theta_j - lr \; \nabla_{\theta_j} F(k) \tag{19}$$

Finally, batch gradient descent, with all training examples, the update rule becomes:

$$\theta_j = \theta_j(0) - lr \; \sum_{k=0}^{m} \nabla_{\theta_j} F(k) \tag{20}$$

5.1.3 Gradient Perturbation

Given the dropout analysis done in section 5.1.1, we infer that it merely causes a guided perturbation of the gradient. Thus, as a research direction, we propose a gradient perturbation layer, where in the forward pass of network training does nothing, and in the backwards pass the gradient is perturbed.

The gradient is perturbed by a normal distribution centered at zero, with a standard deviation of a multiple of the original (i.e. 3x). Thus the gradient update becomes 21.

$$\nabla F(W_{current,i}) + N(0, \; std(\nabla F(W_{current,i}) \; 3)) \tag{21}$$

Initial experimental results are shown in Section 5.3.3

5.2. Numerical Analysis

As networks are trained, various statistics are captured between epochs, in order to process at a later time. These statistics include the singular values of the weight matrices, the current accuracy, the epoch number, gradients at certain layers, and the tuning parameters.

5.3. Image Classification

Image classification is considered under a number of scenarios, however, discussion has been limited to a representative, yet interesting, subset for the sake of brevity. This representative set includes a fully connected network with both momentum and weight decay turned off, a fully connected network with momentum set to 0.9, and weight decay set to 10^{-3}, where in both fully connected networks dropout probability is varied, and finally initial results are presented for a fully connected network as well as a ResNet with the gradient perturbation layer.

5.3.1 Dropout in Classification: without momentum nor weight decay

Here we evaluate a FCN, which is trained with various levels of dropout, with no momentum or weight decay. The results of these experiments are summarized in Table 1. Observe that the baseline network performs best, followed by each dropout network in order of increasing drop probability.

Given the deductions from Section 5.1.1 we speculate that these result reflect an ongoing attempting to traverse along the gradient of the weight matrix, mostly unrelated to the original weight matrix–thus arriving at local minima.

Network Dropout Probability	Max Accuracy (%)
Baseline – No Dropout	7.55
0.25	6.83
0.5	6.54
0.75	5.22

Table 1. Network accuracy ranking

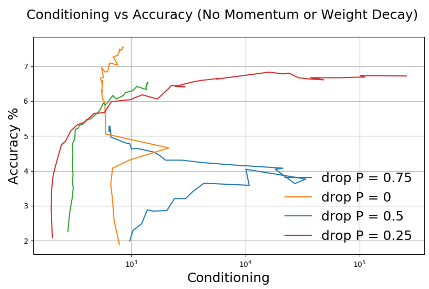


Figure 7. No Momentum or Weight Decay Fully Connected Network Conditioning vs. Accuracy

Generally it appears that, when the conditioning of the weight matrix becomes too large, it becomes difficult for the network to increase accuracy. It is as if first the condition number has to drop. This is made especially clear when observing Figure 7, more specifically noting the performance of $P_{drop} = 0.25$ and $P_{drop} = 0.75$, in the case of $P_{drop} = 0.25$ one can observe that the accuracy begins to climb to 6.5%.

This observation is further illustrated in Figure 8, which shows the change in accuracy plotted against the change in conditioning; observe that generally accuracy does not tend to change when changes in conditioning are large.

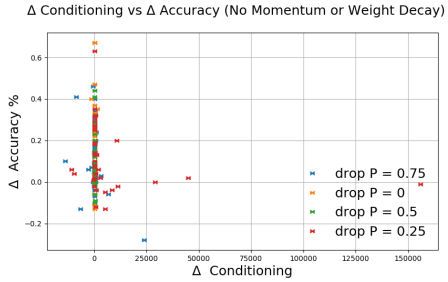


Figure 8. No Momentum or Weight Decay Fully Connected Network Difference in Conditioning vs. Difference in Accuracy

The conditioning then begins to explode and the accuracy then makes very little progress in either the positive or negative direction; this appears to be a local minima.

The line shown for $P_{drop} = 0.75$ illustrates a similar story, the conditioning explodes and accuracy begins to stagnate. In order to make any meaningful progress on accuracy, the conditioning must first decline. Note that the accuracy still seems to be low and that the conditioning axis is a log plot scale.

It is interesting to observe that generally the burst in conditioning is not a result of the maximum singular value increasing. Instead, it is due to the minimum singular value decreasing in orders of magnitude! This implies that increasing dropout leads the optimizer to believe that the network is over-sized, at least in the case of the observed layer. This behaviour is exactly as expected given the discussion in [4] as co-adaption between neurons is eliminated.

Thus, it may be possible to use dropout in hidden layers to yield a compression in the network's layers, via Principal Component Analysis, with negligible loss in accuracy, as suggested in [3].

5.3.2 Dropout in Classification: with momentum and weight decay

The maximum accuracy observed is still achieved by the baseline model followed by the dropout models in order of ascending dropout probability. This is possibly the result of requiring greater attention to tuning of hyper parameters,

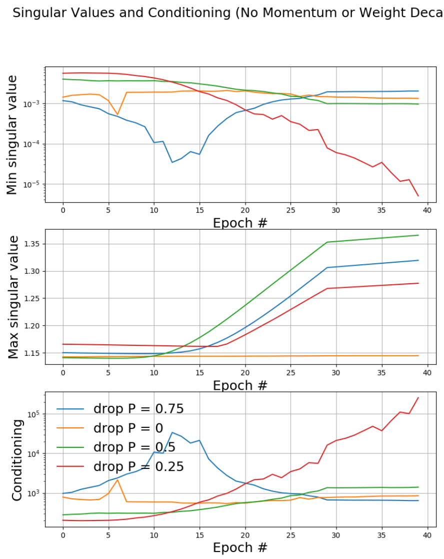


Figure 9. No Momentum or Weight Decay Fully Connected Network Singular Values vs. Epochs

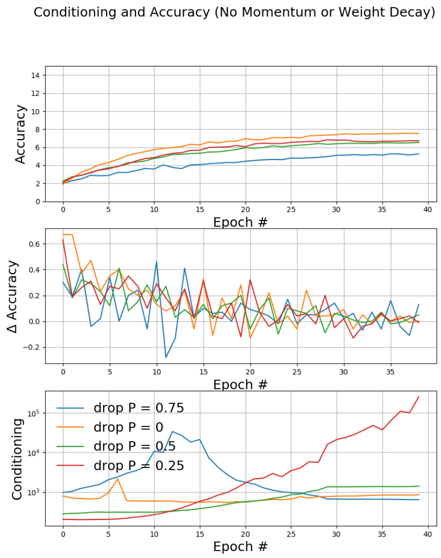


Figure 10. No Momentum or Weight Decay Fully Connected Network Singular Values vs. Accuracy

that is the learning rate and weight decay. The results are summarized in Table 2.

Network Dropout Probability	Max Accuracy (%)
Baseline – No Dropout	13.71
0.25	11.90
0.5	10.21
0.75	7.56

Table 2. Network accuracy ranking

A similar trend manifests when using both momentum and weight decay as illustrated in Section 5.3.1. Generally, with the introduction of momentum and weight decay, the network seems to be better able to "escape" regions appearing to be minima. This is observable in Figure 11, with sharp peeks in conditioning, where accuracy does not improve followed by a drop in conditioning and subsequently in an improvement in accuracy. This is likely a result of the effective blending the gradients of previous dropout iterations via momentum as discussed in Section 5.1.1.

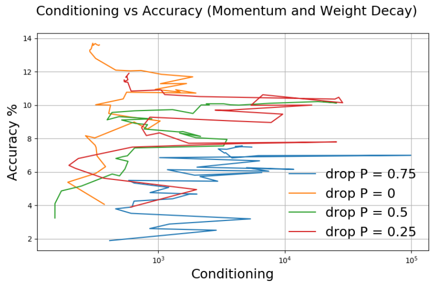


Figure 11. Momentum and Weight present Decay Fully Connected Network Conditioning vs. Accuracy

Nonetheless it seems that the maximum singular values are not becoming excessively large, see Figure 12. The drastic increase in conditioning is a result of the minimum singular values tending towards zero. This is similarly observed in the experiments conducted in Section 5.3.1.

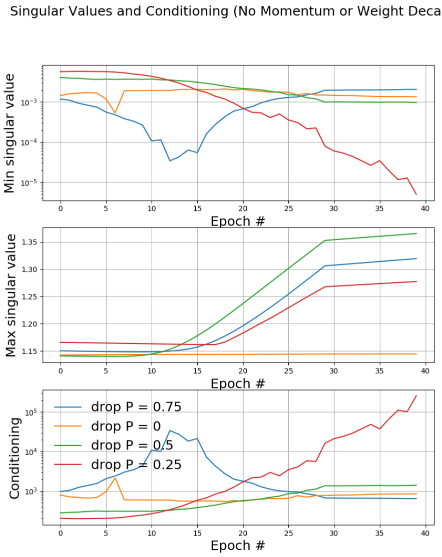


Figure 12. Momentum and Weight Decay Present Fully Connected Network Singular Values vs. Epochs

5.3.3 Gradient Perturbation

Preliminary experimental results for networks using the novel gradient perturbation layer are compared to networks without it in Table 5.3.3.

Observe that ResNet and FCN with the addition of the gradient perturbation layer are both able to marginally outperform stock networks with a $\%0.12$ performance boost. It is likely that to fully benefit from the gradient perturbation layer requires additional tuning.

Network	Layer Change	Best Accuracy
Fully Connected	No change	13.71
	Dropout	11.90
	Gradient Perturbation	13.84
ResNet18	No change	27.11
	Gradient Perturbation	27.24

Table 3. Network accuracy ranking

The gradient perturbation layer offers some desirable properties, observe in Figure 13, whre ResNet18 is trained with and without the layer, as the number of epochs increases the conditioning remains stable, and the minimum singular value tends to increase rather than tend towards zero.

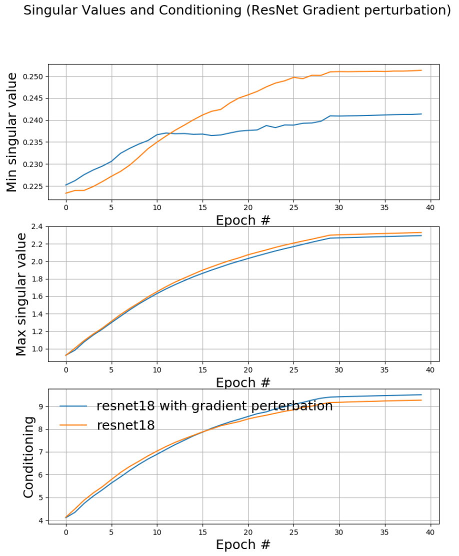


Figure 13. ResNet Gradient Perturbation Singular Values and Epochs

5.4. Deep QLearning

First, a reinforcement learning agent is allowed to roam, and provided the raw sensory information from the environment. Under model-free constraints, the agent at no time had any knowledge of the environment's transition function. Two environments were analyzed, with different input distribution: Sine, and Normal. As a result, the agent learned the optimal $QTable$ in each environment. For the general method, see Section 4.3.2.

Second, to avoid the curse of dimensionality, brought about by the number of possible state-action tuples for which a function must be learned, a fully-connected neural network is trained. The network approximates an agent's $QTable$, and is thus able to predict the optimal value function (and it's equivalent optimal policy), under any circumstance. The network has the added benefit of seamlessly providing interpolation between discrete state-action tuples. It is trained with with residual L2 norm as the objective function.

Two studies on the network's output layer are published: relating the layer's singular values to its conditioning, and relating the network's error on unseen data to the layer's conditioning. The impact of training stochasticity is evaluated, using a network without any stochasticity as the baseline. The mathematical foundation of this method is outlined in Section 5.1.2.

5.4.1 Stochasticity in Regression: singular values vs. conditioning

Figures 14 and 15 show that, for regression, neural networks can be well conditioned. For the QLearning agent, operating in two different environments, the output layer's conditioning number grows in concave fashion. In both environments, early in the training epochs, the minimum singular value quickly stabilizes with a small number. Meanwhile, the maximum singular value grows during training, more significantly in one of the environments, eventually leveling. We observe that training stochasticity can have a large percent impact over the smallest singular value, or the largest singular value, or both.

Given the deductions in Section 5.1.2, these results reflect the gyrations of the gradient under training stochasticity. As the result of parameter search in the test data set, ultimately a small enough learning rate assures us of convergence to the optimal network weights.

5.4.2 Stochasticity in Regression: test error vs. conditioning

In Figures 16 and 17, a negligible impact of stochasticity on conditioning (and vice versa) may be seen. In relatively few epochs, the neural network is able to learn weights that drastically reduce testing error. Thereafter, the reduction in epoch error (gain) reduces practically to zero in both agent environments.

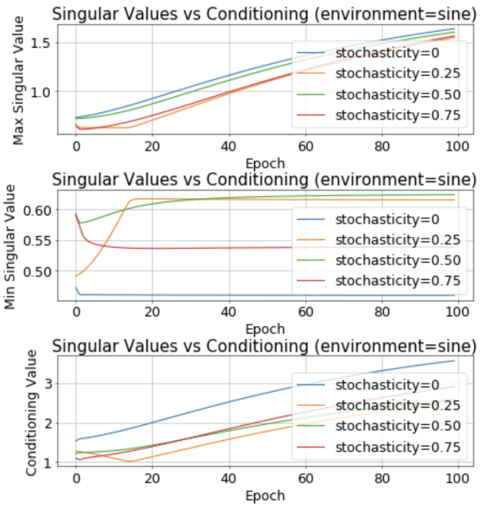


Figure 14. Maximum and Minimum Singular Values, versus Weight Matrix Conditioning (agent in Sine environment

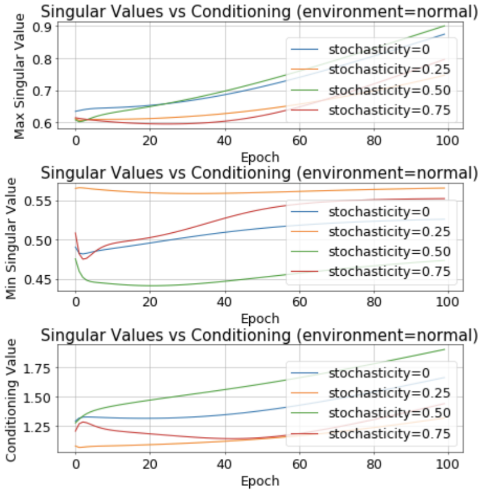


Figure 15. Maximum and Minimum Singular Values, versus Weight Matrix Conditioning (agent in Normal environment

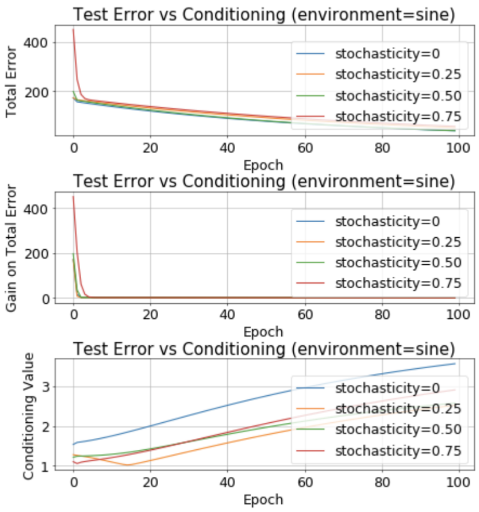


Figure 16. Test Error, versus Weight Matrix Conditioning (agent in Sine environment

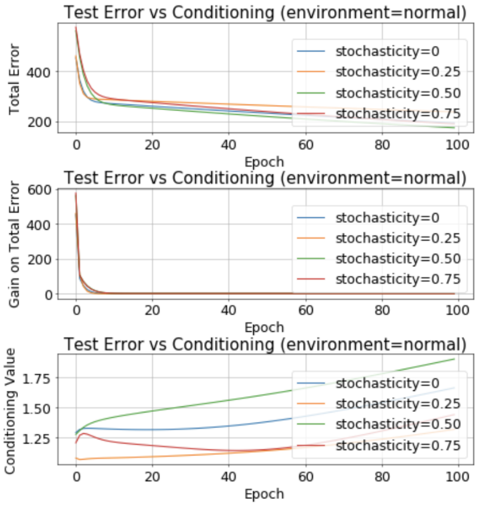


Figure 17. Test Error, versus Weight Matrix Conditioning (agent in Normal environment

6. Conclusion and Future Work

Analyzing the conditioning of weight matrices of a network as it is trained offers a great deal of insight into the behaviour of neurons as different techniques are applied for generalization. In order to make meaningful progress on training it is important that conditioning does not become overly large, otherwise, it is possible to encounter difficult to escape local minima. It is important to realize that exploding conditioning, however, may not be the result of maximum singular values becoming excessively large. Instead, it may be a result of minimum singular values approaching zero. And thus, weight matrices becoming rank deficient.

On one hand, high conditioning may impair learning. On the other, it could potentially signal the opportunity to compress the network's circuitry. Before continuing to learn, the number of neurons in a layer could be reduced, by henceforth relying on the principal singular values.

In fact, we believe that adding conditioning to a network's objective function, explicitly optimizing for better conditioning, could prove a breakthrough to learning performance. And particularly for large classification applications, hierarchical learning could prove a key to achieving lower-conditioned models.

Future work should involve formal reasoning coupled with numerical analysis, on the effect of all the generalization techniques that practitioners rely on. For these techniques, as well as for neuron dropout and learning stochasticity explored here, it may be advantageous to deffer their application to later epochs in the learning process.

This work should also be extended to better understand convolutional layers, and other more complex types of neural networks.

Finally, further investigation is necessary into the tuning and implementation of the proposed gradient perturbation layers. For example, we suspect that applying larger perturbations every n epochs could result in better escaping local minima.

The project's repository is at https://github.com/pablo-tech/DeepNetwork-Conditioning-vs-Performance

7. Contributions

Both authors contributed to the overall direction, formal analysis, and conclusions.

Jakub: Brought forth the mathematics of gradient descent and momentum in the context of dropout. Singular value analysis of image classifier as well as interpretation of results. Gradient perturbation layer. Several implementations of FCNs with and without dropout and gradient perturbation layer. Perturbed implementations of ResNet. Proposals for exploitation of singular values. Experiments were also conducted to look at varying weight decay and momentum, however, they did not meaningfuly contribute to the overall paper result.

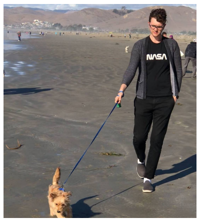
Pablo: Delved mathematical reasoning on stochasticity of Gradient Descent. Reinforcement learning and deep learning solvers relying only on Numpy. Application of these solvers for regression of the OneNow dataset. Neural networks for multiple agent actions, in multiple environments, and under various degrees of stochasticity. Originated conditioning as a regularization term in objective functions.

8. Acknowledgements

The authors are grateful to Winnie Lin for her skillful guidance, as well as the whole course staff of cs205L.

Special thanks to Professor Ron Fedkiw for letting us in on ultimate solutions to linear problems.

Authors



Jakub Dworakowski is an Algorithms Engineer focused on the performance of real time computer vision algorithms on embedded systems. Jakub has worked in a wide variety of fields ranging from Machine Learning to Controls Engineering.

Pablo Rodriguez Bertorello leads Next Generation data engineering at Cadreon, a marketing technology platform company. Previously he was CTO of Airfox, which completed a successful Initial Coin Offering. He is the co-inventor of cloud platform company acquired by Oracle. And the original designer of the data bus for Intel's Itanium processor. Pablo has been issued over a dozen patents.

References

[1] BogoToBogo. Batch gradient descent vs stochastic gradient descent. *BogoToBogo*, 2017. 5

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009. 2

[3] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev. Compressing deep convolutional networks using vector quantization. *CoRR*, abs/1412.6115, 2014. 2, 6

[4] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012. 1, 2, 4, 5, 6

[5] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. 1, 2

[6] K. Jia. Improving training of deep neural networks via singular value bounding. *CoRR*, abs/1611.06013, 2016. 1

[7] J. Jordan. Neural networks: representation. *JeremyJordan*, 2017. 4

[8] A. Krizhevsky. Learning multiple layers of features from tiny images. 2009. 2

[9] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines vinod nair. volume 27, pages 807–814, 06 2010. 4

[10] ONNX. Open neural network exchange format. https://onnx.ai/. 3

[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017. 2

[12] Pytorch. pytorch/examples. https://github.com/pytorch/examples/tree/master/imagenet, Feb 2019. 3

[13] Pytorch. pytorch/vision. https://github.com/pytorch/vision, Mar 2019. 3

[14] L. Sagun, L. Bottou, and Y. LeCun. Singularity of the hessian in deep learning. *CoRR*, abs/1611.07476, 2016. 1

[15] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org, 2013. 1, 5

[16] S. v. d. Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, Mar. 2011. 2

[17] D. Wilson and T. R. Martnez. The general inefficiency of batch training for gradient descent learning. *ScienceDirect*, 2001. 1

[18] D. R. Wilson and T. R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Netw.*, 16(10):1429–1451, Dec. 2003. 5