Experimental Analysis of Overparameterized Neural Networks

Brendan Kondracki (bk2793)
Shambhavi Roy (sr3767)
Deepak Dwarakanath (dd2676)

EECS E6699: Mathematics of Deep Learning

Abstract

In recent years, a great deal of theoretical research has been done analyzing the properties of overparameterized shallow neural networks. As a consequence, many interesting results have emerged demonstrating the linear behavior of neural networks when sufficiently overparameterized. Similarly, more recent results have shown that the linear behavior of neural networks can also be achieved through sufficient scaling, often referred to as "lazy training". In this paper, we take an experimental approach to analyze the effects of overparameterization on shallow neural networks using more advanced optimization techniques. In addition, we analyze the convergence and generalization of shallow networks when trained in the lazy regime. We will show that beyond overparameterization, width of a neural network is inversely correlated with training loss. Finally we will look at how network width will affect a model's ability to generalize.

1 Introduction

Despite the numerous achievements of neural networks when applied in practice, much is still unknown about the theoretical basis for these successes. Due to the high complexity of modern neural network architectures, many modern advances in neural network theory have analyzed the properties of overparameterized shallow networks, due to their mathematical elegance. We can define overparameterization as when the number of parameters exceeds the sample size, or in our case, the hidden layer m.

In [7], it was shown that an overparameterized shallow ReLU network converges to a linear kernel model, given sufficient model scaling. In addition, it was demonstrated in [1] that given a sufficient amount of overparameterization, a shallow network can achieve linear convergence to a global minimum with standard gradient descent. More recently, however, [4] showed that the scaling factor introduced through the previous results inadvertently restricts neural networks to a linear behavior. It was also shown that the weight initialization used by many modern networks can also cause this linear behavior to occur as the network width increases.

In the first half of our paper, we show the behavior of shallow ReLU networks of increasing widths when trained with various optimization methods, outside of the basic techniques utilized in [1]. Further, we discuss the concepts of Lazy Training and Neural Tangent Kernels and our experiment to corroborate with the findings in [6]. Subsequently, we present a series of experiments analyzing the convergence and generalization of neural networks when different scaling factors are introduced.

1.1 Setup

We start by analyzing a single hidden layer neural network of the form[1]:

$$f(W, a, x) = \sum_{r=1}^{m} a_r \sigma(w_r^T x) \tag{1}$$

where x $\in \mathbb{R}^d$ is the model's input, $w_r \in \mathbb{R}^d$ is the weight vector for neuron r in the network's hidden layer, σ is the ReLU activation function, and a_r is the output weight value corresponding to hidden neuron r.

Additionally, we calculate our network's loss utilizing Mean Square Error(MSE) of the form[2]:

$$L(W, a) = \sum_{i=1}^{n} \frac{1}{n} (f(W, a, x_i) - y_i)^2 \tag{2}$$

where n is the number of training examples per batch, and y_i is the true value of each data point in our set.

Initially, we consider the case of optimizing the first layer weights of our neural network using standard gradient descent with a fixed step size of the form:

$$W(t+1) = W(t) - \eta \frac{\partial L(W(t), a)}{\partial W(t)} \tag{3}$$

1.2 Experimentation

To verify the results presented in [1], we first construct a single hidden layer neural network as described above, with a fixed step/batch size. We randomly generate 1000 $d = 1000$ dimensional data points sampled from $U(-1, 1)$, and randomly generate $d = 1$ dimensional outputs from a standard Gaussian distribution, $N(0, 1)$.

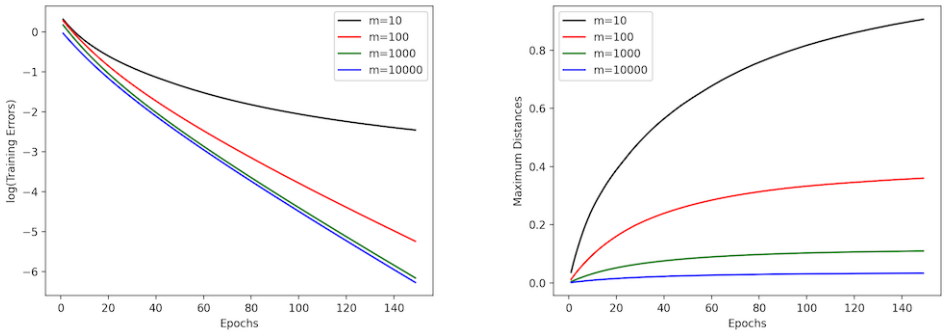


Figure 1: **Results using Standard Gradient Descent (SGD)**

In the first diagram, we measure the convergence rate of our network given various hidden layer widths. Note that there are 2 overparameterized models, $m = 1000$ and $m = 10000$. In the second diagram, we measure the maximum distance between our weight vectors and their initializations, formally:

$$max_{r \in m} ||w_r(t) - w_r(0)||_2 \tag{4}$$

In both cases, our results are comparable to that of [1], whereby the convergence of our model improves as the width of our network increases, and the degree to which our weight vectors vary from their initializations decreases as our model width increases.

[1]This equation differs slightly from that of [1], where the final linear combination of the network is divided by $\sqrt{m}$.
[2]This loss function also differs from that of [1], where the squared difference is multiplied by $\frac{1}{2}$ rather then $\frac{1}{n}$

2 Adaptive Gradient Optimization

2.1 Adagrad Overview

One of the difficulties of gradient descent is the accurate tuning of our optimization's learning rate. If the rate constant is set too high, we run the risk of overshooting a minimum of our loss function, whereas a learning rate too small can lead to slow convergence of our model. As a result, new methods of optimization have emerged in an attempt to combat some of these difficulties.

We start by looking at one such methodology, coined "Adagrad"[2]. Formally, we define the procedure of updating our model's parameters by the following equation:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \frac{\partial L(W(t), a)}{\partial w(t)} \tag{5}$$

where G_t is a diagonal matrix in which each ii entry represents the sum of the squared gradients for parameter i up to time t, and ϵ is a small positive constant. Using this new method of optimization, our training procedure can modify the learning rate for each of our parameters in real time to try and avoid some of the challenges addressed earlier with standard gradient descent.

2.2 Adagrad Experimentation

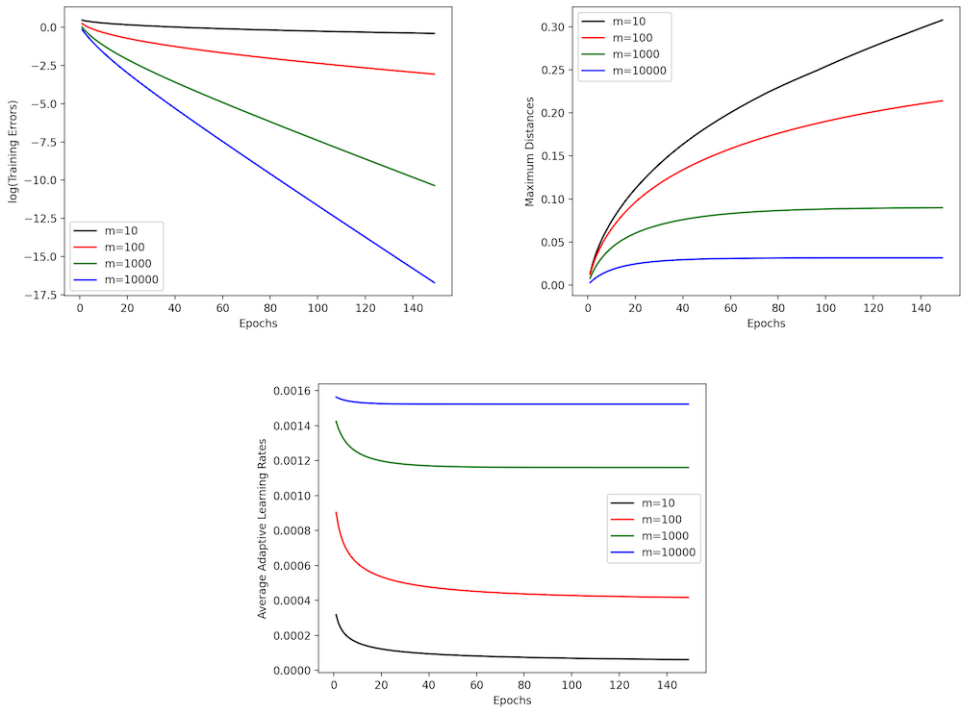


Figure 2: **Results using Adagrad**

In this experiment, we setup a neural network with the same constraints as our network in section 1, utilizing Adagrad optimization for our model's training. Analyzing the same metrics as before, we see a similar trend as with standard gradient descent. However, with this form of optimization, we see that a network with a relatively small number of hidden neurons exhibits a slower convergence rate, whereas a network with a larger number of neurons exhibits a faster convergence rate. To help explain this phenomenon, we start by

analysing the equation for the change in our model's predictions over time using fixed learning rate gradient descent:

$$\hat{y}(t+1) - \hat{y}(t) = \sum_{r=1}^{m} a_r \left(\sigma \left(\left(w_r(t) - \eta \frac{\partial L(W(t), a)}{\partial w(t)} \right)^T x_i \right) - \sigma \left(w_r(t)^T x_i \right) \right) \tag{6}$$

Substituting the equation for Adagrad optimization, we can rewrite the previous equation as:

$$\hat{y}(t+1) - \hat{y}(t) = \sum_{r=1}^{m} a_r \left(\sigma \left(\left(w_r(t) - \left(\frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \frac{\partial L(W(t), a)}{\partial w(t)} \right) \right)^T x_i \right) - \sigma \left(w_r(t)^T x_i \right) \right) \tag{7}$$

We know from our experimental results that as the width of our hidden layer increases, we see a decrease in distance between our weight vectors and their initializations. This would imply that as the width of our network increases, the magnitude of the gradients calculated for our weight vectors would decrease.

Applying this to the equation for the change in model predictions over time, we see that for a relatively small width network (i.e. number of neurons = 10, 100), the values of G_t increase rapidly due to the squaring of large magnitude gradients. Similarly, we see an inverse effect when applying the same logic to a larger width network (i.e. number of neurons = 1000, 10000). The third graph above helps to visualize this difference, where we measure the average adaptive learning rate in our network's hidden layer over time. We see that for the two largest width architectures, the effective learning rate for our parameter updates is substantially greater than its initial value of $5 \cdot 10^{-4}$. Conversely, for our m=10 width network, the effective learning rate is greatly reduced from this base value due to the large accumulation of squared gradients.

Due the scaling of our learning rates in this fashion, we see a much slower convergence rate for smaller width networks, and a much faster convergence rate for larger width networks, when comparing these results to standard gradient descent training.

2.3 RMSProp Overview

Although Adagrad gives us a more dynamic optimization approach by eliminating the need to manually tune our model's learning rate, it presents us with entirely new obstacles. As seen in our previous experiments, the incorporation of the squared sum of gradients into our optimization method can lead to a sub-optimal stopping point during our network's training due to the diminishing of our model's learning rate. Fortunately, new variations of Adagrad optimization have since been developed to prevent such situations from occurring.

One such variation, RMSProp, differs slightly from Adagrad optimization by dividing the learning rate by an exponential moving average of squared gradients. Formally, we define equation:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{8}$$

then, replacing our G_t matrix from Adagrad with the above expression, we can rewrite our update rule as:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot \frac{\partial L(W(t), a)}{\partial w(t)} \tag{9}$$

By using an exponential moving average of squared gradients in this fashion, we are able to mitigate the risk of reducing our optimizer's learning rate at too great a speed.

2.4 RMSProp Experimentation

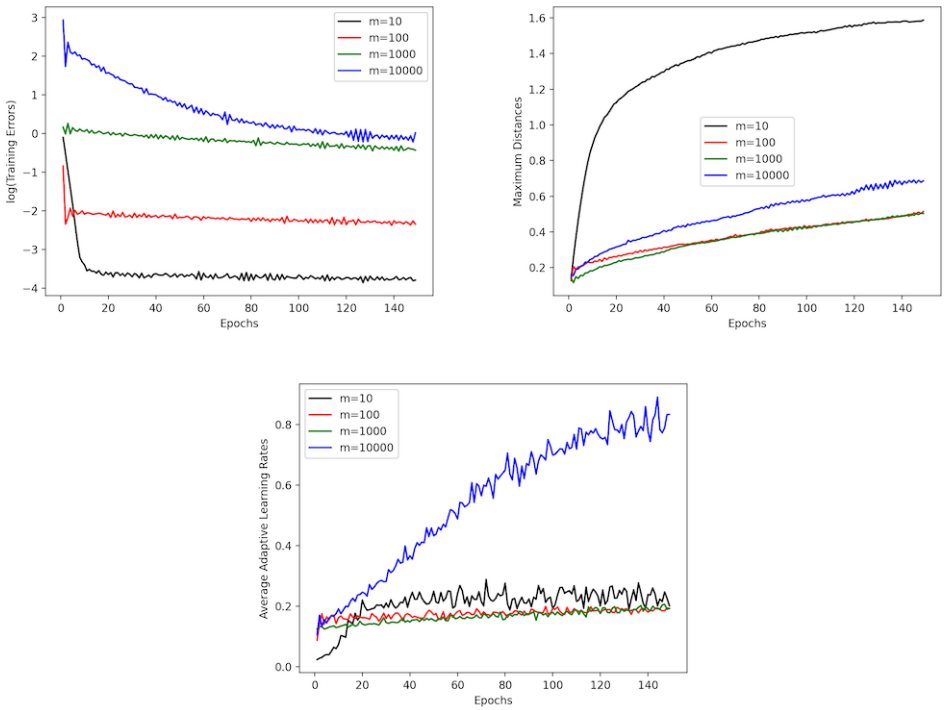


Figure 3: **Results using RMSProp**

In this experiment, we set up a series of single hidden layer neural networks using the same constraints as section 2.2, this time using RMSProp optimization with $\gamma = 0.9$. Compared to our results using Adagrad optimization, we see a few distinct differences. Most notably, we see across all network widths, the average effective learning rate is at least 300 times the base value of $5 \cdot 10^{-4}$. In addition, across all architectures, we see an increase in the effective learning rate over time (an undesirable outcome when considering the expected convergence of our model).

This gives rise to a noticeable downside of RMSProp optimization, whereby the value of $E[g^2]_t$ can in some cases decrease over time, as opposed to Adagrad where the value of G_t can only increase. While SGD and Adagrad performed best on the overparameterized model ($m = 1000, m = 10000$), in the same way that the effective learning rate was able to increase from its base value for a 10000 neuron Adagrad optimized network, an RMSProp optimized network of the same width was able to drastically increase the effective learning rate from its base value. In fact, as RMSProp uses an exponential moving average of squared gradients to reduce the effective learning rate, a 10000 neuron network can drastically decrease this average over time, resulting in large increases in the effective learning rate.

Relating this back to the convergence diagram of our networks, we see that these learning rate increases result in much greater oscillations when measuring the loss of our model over time. Subsequently, we see that these inflated rates shift the weight vectors of our hidden layer further from their initializations, when compared to both Adagrad and standard gradient descent.

3 Lazy Training

3.1 Overview of Concepts

So far we have seen that increasing the width of a shallow neural network has the effect of lowering training losses (in the case of SGD and Adagrad) and reducing the change in weights. It has been shown that infinitely wide neural networks behave like a linearization of the neural network around its initialization [6]. This is in line with our overview and results regarding overparametrization. However, there is an alternate viewpoint, expressed in [4], that this is because of implicit choices of scaling/weight initializations, rather than overparameterization. Introducing an explicit scaling factor can show that, practically speaking, any parametric learning model can be trained in a "lazy" manner, if its output is close to zero at initialization [4]. This scaling factor α is the fundamental parameter associated with lazy training.

The basis for lazy training is using certain assumptions about parameterized learning models. Initially, we can view a model as a black box that maps an input variable $x \in \mathbb{R}^d$ to an output variable $y \in \mathbb{R}^d$, with the following developments:

- Parametric model $f : \mathbb{R}^p \times \mathbb{R}^d \to \mathbb{R}$
- f is associated with a training algorithm that selects parameter $w \in \mathbb{R}^p$ given sequence of n data points $(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$, $i \in \{1...n\}$.
- f should be differentiable, and thus training can be gradient-based.
- A strongly-convex loss function should be used.

Setting α sufficiently high can make a parameterized model f more or less equivalent to a model that uses a Neural Tangent Kernel (NTK). Neural tangent kernels are used to simulate infinite width networks because they can be defined in terms of mathematical functions rather than parameters. We have presented a theoretical description of the Neural Tangent Kernel in the following subsection.

3.2 Neural Tangent Kernels

Recent developments in theoretical deep learning have shown that the infinite-width neural networks simplify to linear networks on application of a NTK function.

The authors in [6] present the concept of NTK using a simple fully connected neural network of depth L and a Lipschitz non-linearity σ. Consider that each layer in this network has n_i neurons. All the parameters are randomly assigned at initialization and are collected in the Θ vector as $\Theta = (w_0, \ldots, w_L)$.

Assuming that f is differentiable, we linearize the model around the initial parameters w_0 using Taylor Series approximation:

$$f_0(w, x) = f(w_0, x) + (w - w_0)\nabla_w f(w_0, x) \tag{10}$$

Here, the expansion reduces to a linear function in the weights of the network. So the Neural Tangent Kernel is defined as:

$$K(x, y) = \nabla_w f(w_0, x) \cdot \nabla_w f(w_0, y) \tag{11}$$

Equation (11) is called the Tangent Model. We can expect training $f_0(w, x)$ to approximate $f(w, x)$ when $f(w, x)$ is linear, that is $f_0(w, x) = f(w, x)$.

The NTK is notable because, using an infinite width limit, it converges to an explicit limiting function and stays constant during training. Thus it can be studied as a function, not as a collection of parameters and thus can be analyzed mathematically. The limiting NTK can be defined in terms of a positive-definite covariance matrix and is thus useful for optimization.

Recent literature shows that infinite-width networks on training with gradient descent optimization are Gaussian processes that can be computed using the NTK function. So we have experimented with this idea by comparing the prediction of the Neural Tangent Kernel theory with training an ensemble of neural networks. The NT kernel describes how the gradient descent evolves over time. An ensemble of finite width networks is used in this experiment for a quantitative comparison.

We performed this experiment using the Neural Tangents library [8]. In this experiment, we have compared the results of training an ensemble of neural networks with the prediction of the Neural Tangent Kernel concept. We have performed this experiment on a simple synthetic dataset. The training data points are drawn from the process $y = f(x) + \epsilon$. Here, x, is drawn from a Uniform distribution in $(-\pi, \pi)$ and ϵ is Gaussian noise from $\mathcal{N} \sim (0, 1)$. The function $f(x) = sin(x)$ is chosen and the testing data is $y = f(x)$ for linearly spaced x in $(-\pi, \pi)$. The training set has 5 points and testing set has 50 points.

Further, we have defined a fully-connected neural network and perform exact Bayesian inference on this network. The result of this inference is compared with the result of training an ensemble of finite-width neural networks. This is done to show that there is agreement between the NTK concept and the result of training neural networks. The result of this experiment is shown in the figure below.

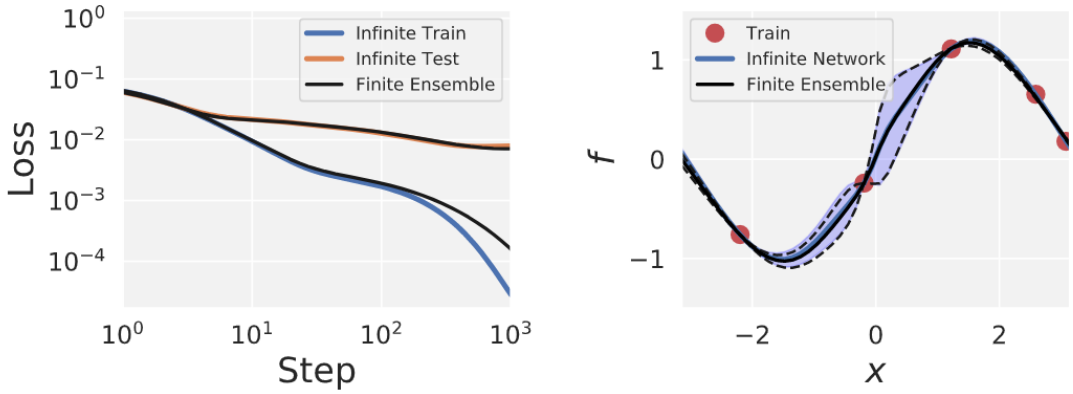
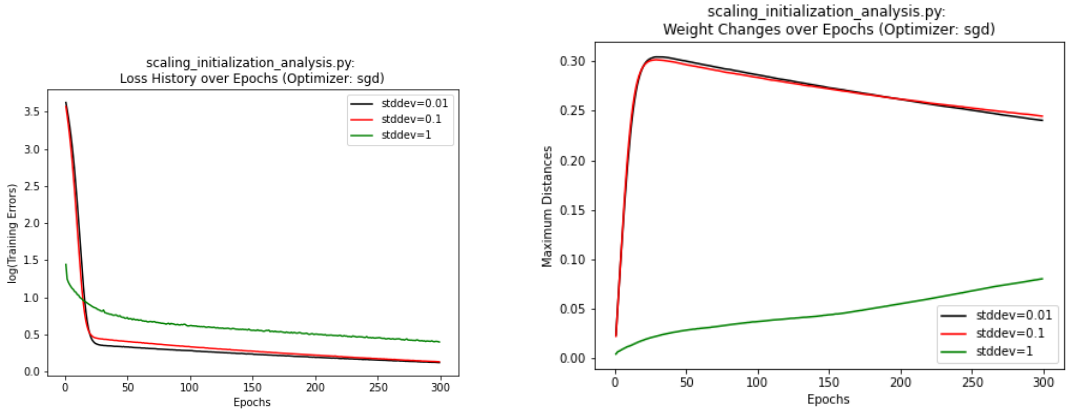


Figure 4: **Comparison of training the infinite width neural network and training an ensemble of neural networks**

The experiment is described in more detail along with the software used in the Appendix.

3.3 Scaled Model Experimentation

As mentioned in the section 3.1, a parameterized model can be trained in a lazy regime if its output is close to zero at initialization [6],[4]. In the experimental section here, we will explore the use of varying node weight initialization as a precursor to the scaling factor used in lazy training. The following figures show that initializing the layer weights such that they hover closer to 0 (by reducing the variance), can in fact get us better overall errors over training epochs.



(a) Log Loss over Epochs

(b) Weight Change over Epochs

Figure 5: **Layer Weight Initializations (m=100 nodes in hidden layer) by varying Standard Deviation (mean=0) (Uses SGD optimizer)**

The figure shows that keeping the variance/standard deviation low (around a mean of 0), does in fact improve errors over time, though initial weight changes are greater. Note that for this relatively narrow network (m=100 for hidden layer), we still had a positive log loss even after 300 epochs (see section 2 results). The inverse relationship between initial weight variance and error is in line with results from page 2 in [4], which finds that increasing variance effects lazy training. In addition, an increased variance has the effect of worsening generalization, as shown by increased overfitting with increased variance in the following figure (using the neural network model from the previous figure). This is in line with results from [5].

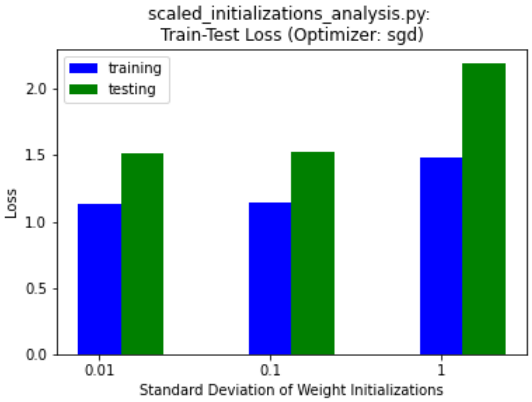
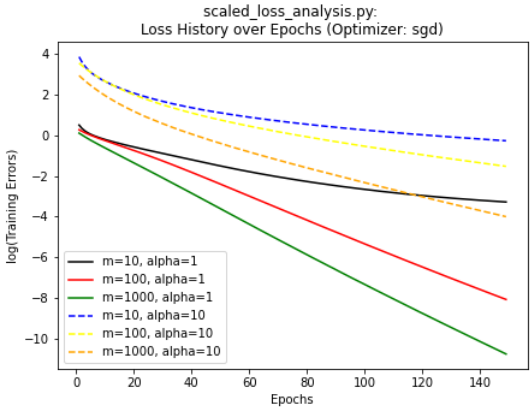
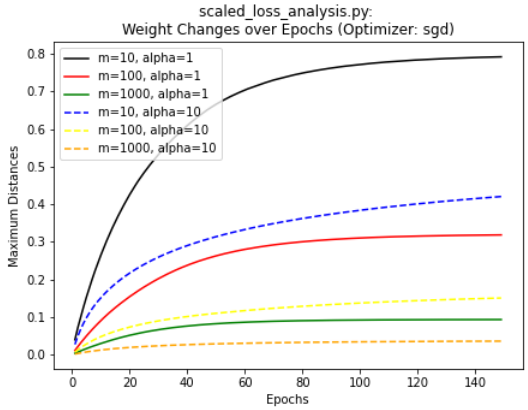


Figure 6: **Training and Test Loss - a view of generalization properties (SGD Optimizer)**

These results are a precursor to the next experiment, which introduces the lazy training scaling parameter α.



(a) Log Loss over Epochs



(b) Weight Change over Epochs

Figure 7: **Results using model scaling for Stochastic Gradient Descent**

We conducted the next experiments using a similar framework as outlined in section 1.2, with a few structural differences. First, we modified the output of our network by introducing a scaling factor α, resulting in the modified equation:

$$f(W, a, x, \alpha) = \alpha \sum_{r=1}^{m} a_r \sigma(w_r^T x) \tag{12}$$

In addition, rather then optimizing exclusively over our hidden layer weights, we optimize over our output weights as well.

Similar to the experiments conducted in [4], we utilize a modified quadratic loss function of the form:

$$L(W, a, \alpha) = \sum_{i=1}^{n} \frac{1}{n\alpha^2} (f(W, a, x_i, \alpha) - y_i)^2 \tag{13}$$

Analyzing the change in our hidden layer weights over time from the second diagram, we see that regardless of scaling, the degree to which our weights shift away from their initializations decreases as the width of our network increases. However, we also see that applying a scaling factor of 10 to our network results in a smaller change for our parameters when compared to a scaling of 1. In addition, we also see that a smaller scaling term leads to better convergence of our model on average. These results aim to verify the conclusions made in [4], whereby increasing the scaling of our model results in "lazier" training because of the small variation in parameter changes.

Interestingly, however, the shift in our model's parameters alone is not the primary indicator of feature learning in our model, but rather the overall change in the NTK. To better understand this distinction, we obtain the following numerical results for both shallow and deep ReLU networks.

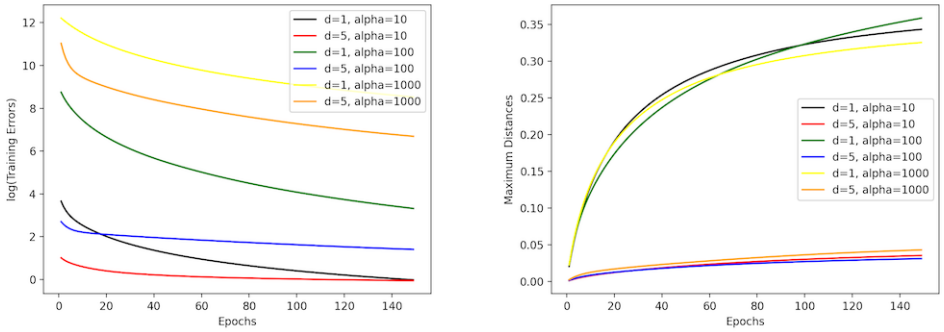


Figure 8: **Left: Loss, Right: Weight Changes - for Deep vs Shallow ReLU Networks**

In contrast to the previous experiment conducted, we now look at the behavior of our networks with a fixed width of m=10, and increasing scaling/depth. To accommodate for the additional hidden layers, we introduce a modified measurement of parameter changes over time by averaging the maximum distances over all hidden layers, L. Formally:

$$\sum_{l=1}^{L} \frac{max_{r \in m} ||w_r^{[l]}(t) - w_r^{[l]}(0)||_2}{L} \tag{14}$$

The first thing to notice from these results is the stark contrast in weight changes between the shallow and deep networks. We see that regardless of model scaling, the networks of depth d=5 exhibit much smaller parameter changes compared to the networks of depth d=1. However, when analyzing the log errors of these two architectures, we see that deeper network architectures consistently perform better on our synthetic data. Intuitively, these results can be attributed to an "amplifying" effect present in deep architectures, whereby small changes in our model's parameters can still lead to large changes in the NTK.

4 Generalization and Learnability

We have seen that overparameterized shallow neural networks can approach low training error and loss, through empirical risk minimization. However, the ultimate goal is generalization of the model, i.e. can the model predict on test data accurately? If not, we have the problem of overfitting.

Generalization is an important precursor to the concept of learnability, which is the study of whether a hypothesis class of functions can properly separate and classify, or "shatter" data points. This is the concept of PAC-learnability, i.e. Probably approximately correct learning. PAC-Learnability has 2 parameters associated with it, error ϵ, and probability of error δ. PAC-Learnability can determine if a learning algorithm can return a hypothesis over its data that has accuracy $1 - \epsilon$, with probability $1 - \delta$. Practically speaking, in order to ensure these desired outcomes, there needs to be a sufficiently large sample size passed into the algorithm. Determining this number involves another computational learning concept, VC (Vapnik-Chervonenkis Theory) [9]. This theory helps determine the consistency of the learning process, as well as generalization, in particular the rate of convergence of a learning process. The mathematics of VC Theory are quite complex, and perhaps beyond the scope of this paper, but nevertheless, the theory has practical implications regarding generalization issues.

The VC dimension is an important parameter defined by the theory. VC dimension measures the richness of a set of functions that can be learned by a binary classification theory (for greater-than-binary cases, there is Rademacher complexity). The specific definition is the cardinality (set size) of the largest set of data points that can be classified, or "shattered" by the binary algorithm. One simple example is a class of linear classifiers, which can be infinite in size (i.e. all possible lines bisecting a space). The VC dimension of the linear class is 3, because all possible linear classifiers can shatter at most 3 points in

space. If, for example, you had 4 points of either output class 0 or 1, the set of 4 points in Cartesian space, $\{((1,0),0),((0,1),1),((-1,0),0),((0,-1),1)\}$, i.e. all at 90 degrees from each other, cannot be shattered by any linear classifier. More complex examples exist, but that will be beyond the scope of this paper. Practically speaking, the VC dimension can help determine the minimum number of data points needed for a space of binary functions to be PAC-learnable:

$$N = \Theta\left(\frac{D + ln\frac{1}{\delta}}{\epsilon}\right) \tag{15}$$

2-layer neural networks like we have studied here have an associated VC-dimension, which varies depending on the number of edges connecting nodes, as well as the activation functions. In this case, the neural network is approximating a class of binary classifiers. A paper by Bartlett 2019 has hypothesized an upper bound $O(W * L * log(W))$ and a lower bound $\Omega(W * L * log(W/L))$, which they posit is better than previously derived bounds. W is the number of weights and L is the number of layers [10].

In the examples we will look at here, we have input data with dimension = 20, 2 layers in the network (1 hidden, 1 output), and varying layer widths (nodes), m. Such a network has $W = 20*(m+1)+(m+1)*1$ weight parameters, so its VC dimension is lower bounded by $\Omega((20*(m+1)+(m+1)*1)*2*log((20*(m+1)+(m+1)*1)/2))$. For a network of width = 100, therefore, its VC dimension lower bound would be 29552. For a network of width 10, the VC dimension lower bound is 2195, and for width 1000, it is 389314.

We have hypothesized a VC dimension for the 2-layer neural network given various hidden layer widths. In addition to equation 15, the VC dimension can also be used to determine the probability that the model's generalization error stays within a certain upper bound. Generalization error is difference between training error and test error. The exact formula for this is:

$$Pr\left(\text{test error} - \text{training error} \leq \sqrt{\frac{1}{N}\left(D\left(log(\frac{2N}{D}) + 1\right) - log(\frac{\eta}{4})\right)}\right) = 1 - \eta \tag{16}$$

For this equation to be valid, $N >> D$, otherwise the calculated values will be invalid. Officially, using these parameters, for each VC dimension based on width, you would need a large number of samples. For width=10, VC_dim=2195, for width=100, VC_dim=29552, and for width=1000, VC_dim=389314, and the number of samples would be in the tens of thousands at a minimum. However, we have run a small experiment that sees low generalization error despite comparatively small sample sizes. We generated a dataset with binary outputs using `Scikit Learn`, and performed cross-validation on it, using a train-test split of 0.95 to 0.05. This is normally small for a validation fraction, but in this case, we used it to prove that the VC dimension doesn't seem to be informative. From equation 15, let's assign $\epsilon = 0.1$ and $\delta = 0.1$. Therefore, the number of samples required for width 10 is approximately 19574, and for width 100 it is 292204, and for width 1000, it is 3888857. Θ is roughly speaking a middle bound, though.

We ran a small experiment to check how "probably approximate" a binary class shallow neural network behaves. We set $m = 1000$, implying a VC dimension lower bound as per Bartlett et al. [10] of almost $400,000$. Basically we ran a cross-validation type operation over 1000 epochs, splitting a binary-labeled dataset of $n = 10000$ into $95:5$ training-validation splits, then found the difference between the training and test error, which we could view as a generalization error as per equation 16. Results are as follows:

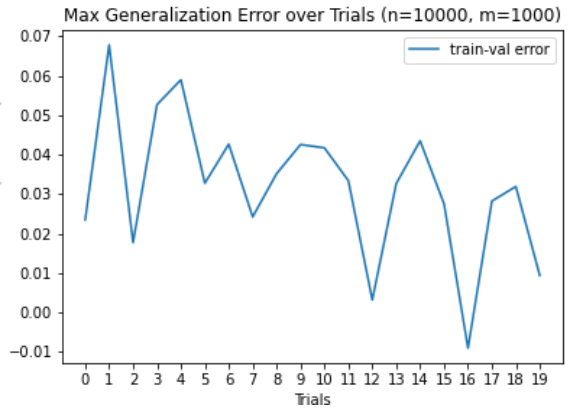


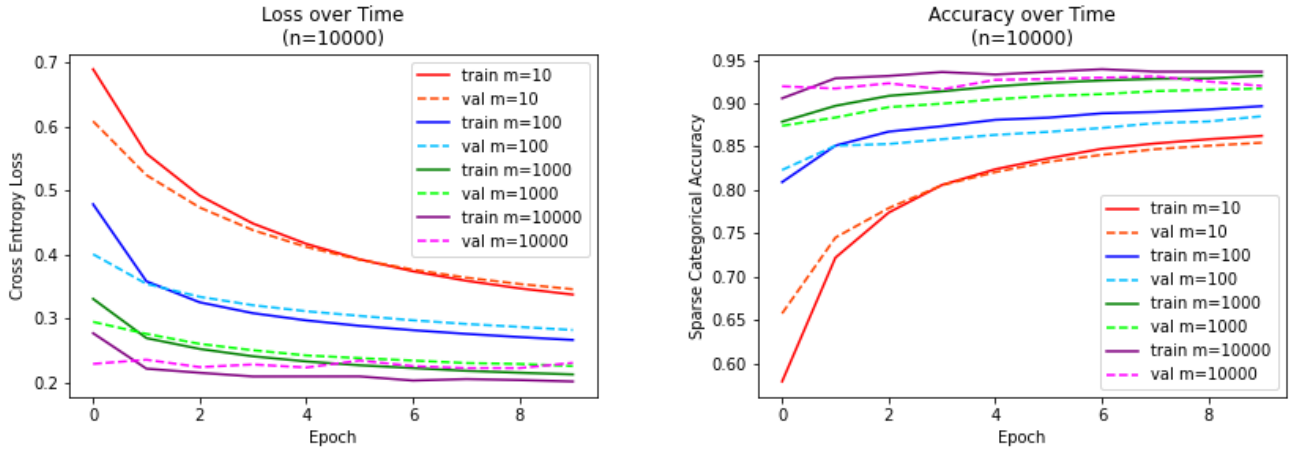
Figure 9: **Maximum difference between training and test error over 20 trials**

The difference between training and test error (equivalently accuracy) was never more than 0.07. None of the trials had high error values, so this is in line with the "Probable" aspect of PAC-learnability. We would expect VC dimensions would predict major overfitting issues. This simple experiment shows that theoretical VC dimensions of a neural network binary classifier cannot explain the behavior of neural networks, which is consistent with much of the existing literature [11].

Much of the existing literature on this subject has found that neural networks generalize well despite violating concepts regarding complexity and learnability. The paper by Neyshabur 2018 discusses several issues that factor into the generalization of neural networks. [11].

- The concept of *capacity*, which is effectively the minimum data size required to generalize is covered. We have seen a version of this in equations 15 and 16 with respect to VC dimensions. They found that as expected, high perceived capacity doesn't predict poor generalization.
- They discuss a concept called *sharpness*, which is a measure of robustness to perturbations in the parameter space (i.e. weights). It is basically a kind of norming of training error. For example, a network with whose weights are almost orthogonal (i.e. dot product is low) and don't interact with each other is considered sharp. This is analogous to a weight or set of weights not having disproportionate effect on outputs (common regularization methods like Ridge and Lasso basically increase sharpness during optimization). It also is not found explain generalization well.
- Norms are used to explain generalization. For example, the Froebenius norm (square root of the sum of the absolute squares of a matrix's entries) of the weights matrix, tends to decrease as network width increases, probably since weight parameters end up being fractions of 1, particularly when overparameterized .
- They were able to find a lower bound for Rademacher complexity with overparameterized networks, which would lower the capacity needed for generalization

In order to further understand generalization and get results similar to the existing literature, we ran experiments training neural networks of varying widths on binary classified data generated using `SciKitLearn`. This was also used in the previous experiment, and had $n = 10000$ with $d = 20$ (majority of d parameters don't predict output and act as noise).

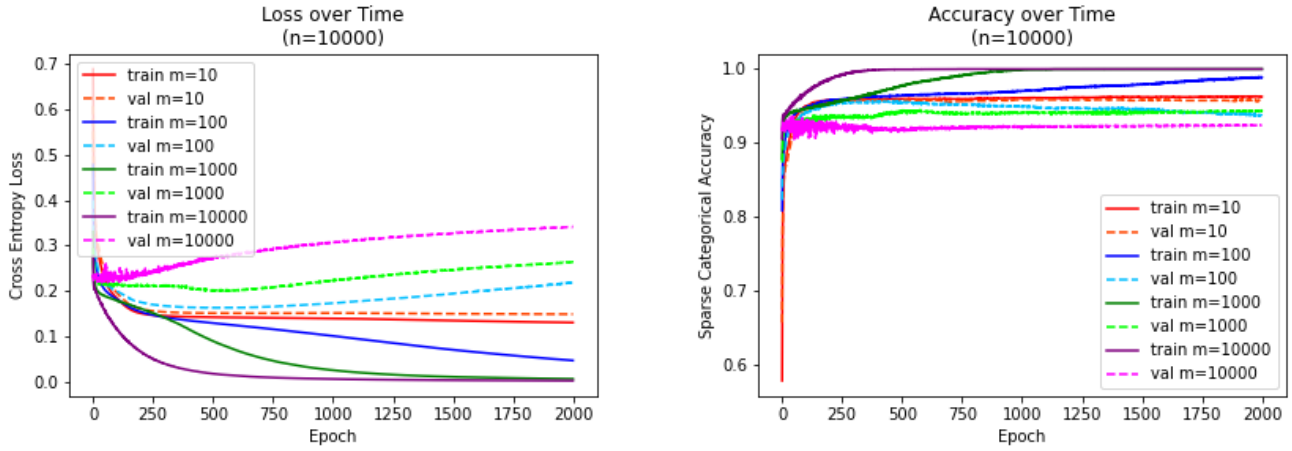


(a) Loss over 10 epochs

(b) Accuracy of Binary Labeling

Figure 10: **Results from running train-validation splits on 2-layer neural networks of varying widths (First 10 epochs)**

Looking at this figure, the $m = 10000$ network, which is overparameterized ($m \geq n$, plus bias terms), sees the best results initially, for both training and test error. Training error via width is consistent with our earlier results. However, as we move past epoch 5, the test loss for the widest network starts converging with the losses for the other networks. Similarly accuracy (opposite of error) starts to converge as well. This somewhat contradicts the results of [11]. In fact, if we look at the results going much further into the epochs, the results get worse.



(a) Loss over 10 epochs

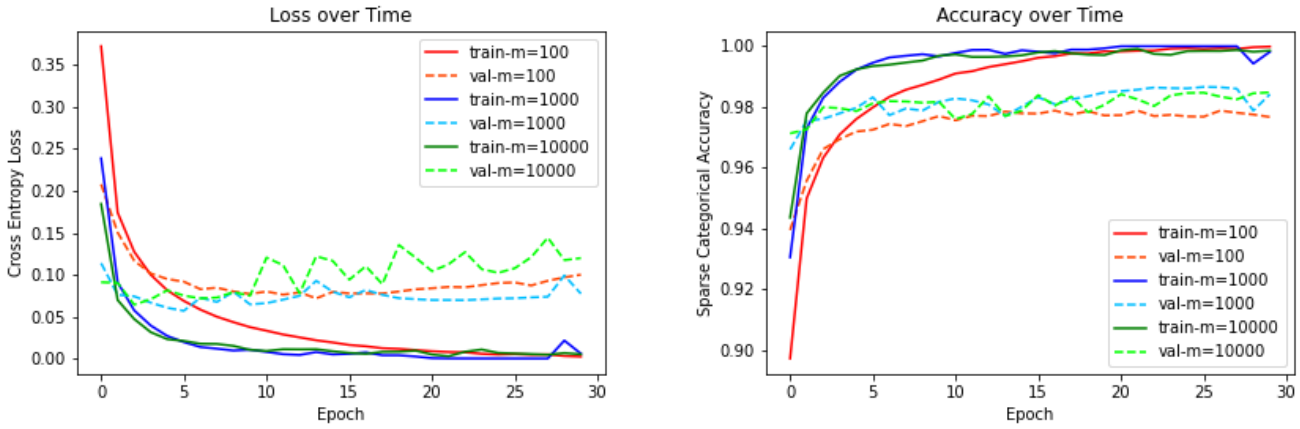
(b) Accuracy of Binary Labeling

Figure 11: **Results from running train-validation splits on 2-layer neural networks of varying widths (2000 epochs)**

At this point, the accuracy for the $m = 10000$ network is lowest, while the loss is increasing. However, looking closely at the graph, loss is increasing at an almost logarithmic rate, so it may converge to a constant at some point. Accuracy, though low relative to the other networks, remains flat.

We also looked beyond binary classifiers, as complexity and capacity can be accounted for with other measures like Rademacher complexity, which we didn't calculate. We used the famous MNIST dataset, which classifies images as being numbers in range 0-9. This set works well with various neural network architectures.

We found results consistent with our previous results on binary-classed data.



(a) Loss

(b) Accuracy

Figure 12: **Results on MNIST dataset**

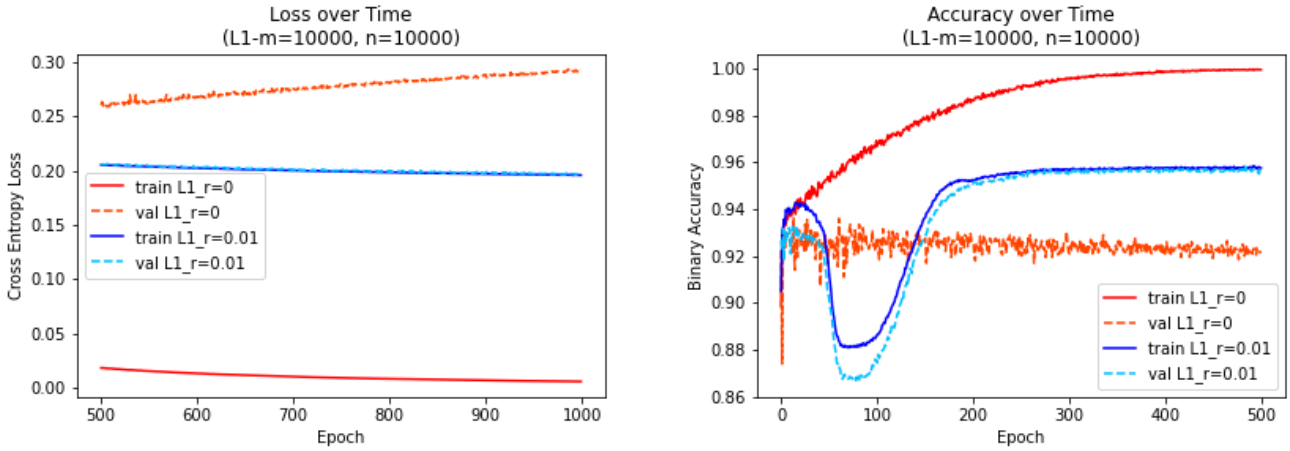
Again, we see that the widest network performs better in the initial epochs, but generalization worsens over time. We could have ran this for more epochs, but we found that the loss and error increase, but at a somewhat flat rate (almost logarithmic in its shape). All our results seem to contradict the results of [11]. However, other literature has found that without some form of explicit regularization, generalization is not realized [12].

Various forms of regularization exist. Common methods include Ridge (adding a loss penalty over the sum of the square of the weights), and Lasso (adding a loss penalty over the sum of the absolute values of the weights). Ridge penalizes weights overall, while Lasso tends to penalize outliers. Early stopping is another way of regularizing. It causes the algorithm to exit training once training and test error start to diverge. In our results, had we used early stopping, it is likely that the algorithm would have exited within the first few epochs. The paper by Weinan et al. [12] already confirmed the issues we have seen in our experiments. They propose a form of L1 regularization, as follows:

$$minimize_{\Theta} R_n(\Theta) + \lambda \sqrt{\frac{ln(d)}{n}} \|\Theta\|$$

$$\|\|\Theta\|\| = \sum_{k=1}^{m} |a_k| \|w_k\|_1 \tag{17}$$

d is data dimension, n is data size, and a_k is an output weight on the weights of the hidden layer nodes. However, [12] relied on an unusual training method, in that they used a very small training set, and a much larger test set, likely to really emphasize overparameterization. They used $\lambda = 0.01$, which is curious because the overall regularization would be quite small when adding the $\sqrt{(\frac{ln(d)}{n})}$. As an alternative, we looked at common L1 and L2 regularization method, varying λ to see whether it improved generalization, specifically for $m = 10000$, which is slightly overparameterized (we have found that width reduces training error somewhat proportionally regardless of whether you are above or below the overparametrization threshhold).



(a) Loss

(b) Accuracy

Figure 13: **L1 Regularization (r=λ, r=0 is NO regularization penalty)**

Note that $\lambda \sum_1^W |w|$ for $L1$ and $\lambda \sum_1^W w^2$ for $L2$. For figure 13a, we ignored the 1st 500 epochs because loss is artificially high because of the $L1$ penalty, but by design it will converge to a fractional loss as with no regularization. We note that $\lambda = 0.01$, training and test/val error converge very well, and accuracy for both training and test/val converge. This implies that we have achieved good generalization. Loss and accuracy diverge for the unregularized network as in previous results. We have practically demonstrated that a (slightly) overparameterized network can generalize well when using a moderate amount of regularization. This is consistent with the results from Weinan et al., who used their $L1$ variant and reduced overtraining significantly [12].

Mathematically speaking, overparameterization, and network width are interesting. Lets assume that you are choosing from a data that follows a distribution (Gaussian or other), and choose samples i.i.d. to optmize over. From this, it appears that stochastic gradient descent seems to pick representative samples to optimize over. Also increasing width of a network distributes the data over more nodes and thus weight parameters, so an individual weight has less impact. Earlier we had mentioned that the Froebenius norm decreases with increasing width. Thus the weights move less, and loss decreases faster, which we have shown in our results. Also, if you are drawing you training and test data from some distribution, the trained network seem to generalize well. Nevertheless, we saw limitations that had to be corrected using regularization. However, generalization error for unregularized models was still less than what computational learning theory (VC, Rademacher) would predict.

5 Conclusion

We have seen that overparameterized neural networks can allow for better convergence when using stochastic gradient descent, as well as Adagrad optimization. Other variants, such as RMSProp, however, can present difficulties in model training when increasing our network width. In addition, we have also seen that there is a direct correlation between the width of a shallow neural network, and the distance the network's weights can move from their initializations. Namely, we have seen that overparameterized networks implicitly cause its weights to undergo smaller variations from their initial values.

In the latter half of our paper, we have also shown that this phenomenon (lazy training) can be replicated through sufficient model scaling/weight initializations. In general, we have also shown that models trained in the lazy regime have worse performance than those where the weights are given more freedom to move from their initial values. Finally, we demonstrated experimentally how VC theory cannot explain certain generalization properties of neural networks. In the coming years, it will be interesting to see further

work done in the area of overparameterized neural networks, as well as the phenomenon of lazy training.

References

[1] Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. *Gradient Descent Provably Optimizes Over-parameterized Neural Networks.* In *International Conference on Learning Representations*, 2019.

[2] J. Duchi, E. Hazan, and Y. Singer. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.* In *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[3] Sanjeev Arora, Simon S. Du, Wei Hu, Zhiuan Li and Ruosong Wang. *Fine-Grained Analysis of Optimization and Generalization for Overparameterized Two-Layer Neural Networks.* In *International Conference on Learning Representations*, 2019.

[4] Lenaic Chizat, Edouard Oyallon, and Francis Bach. *On Lazy Training in Differentiable Programming,* 2020

[5] Lenaic Chizat and Francis Bach. *A Note on Lazy Training in Supervised Differentiable Programming*, 2018

[6] Arthur Jacot, Franck Gabriel, and Clément Hongler. *Neural Tangent Kernel: Convergence and Generalization in Neural Networks.* In *Advances in Neural Information Processing Systems*, 2018.

[7] Youngmin Cho and Lawrence K Saul. *Kernel methods for Deep Learning.* In *Advances in Neural Information Processing Systems*, 2009

[8] Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A. Alemi, Jascha Sohl-Dickstein and Samuel S. Schoenholz. *Neural Tangents: Fast and Easy Infinite Neural Networks in Python.* In *International Conference on Learning Representations*, 2020

[9] Shalev-Shwartz, S., Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms. Cambridge: Cambridge University Press.* doi:10.1017/CBO9781107298019

[10] Peter L. Bartlett, Nick Harvey, Christopher Liaw and Abbas Mehrabian (2019). *Nearly-tight VC-dimension and Pseudodimension Bounds for Piecewise Linear Neural Networks.* In *Journal of Machine Learning Research*, 20(63), 1-17.

[11] Behnam Neyshabur, Zhiyuan Li, Srinadh Bhojanapalli, Yann LeCun and Nathan Srebro (2018). *Towards Understanding the Role of Over-Parametrization in Generalization of Neural Networks.* CoRR, abs/1805.12076.

[12] Weinan E, Chao Ma and Lei Wu (2019). *A Comparative Analysis of the Optimization and Generalization Property of Two-layer Neural Network and Random Feature Models Under Gradient Descent Dynamics.* CoRR, abs/1904.04326.

[13] `https://rajatvd.github.io/NTK/`

[14] `https://colab.research.google.com/github/google/neural-tangents/blob/master/notebooks/neural_tangents_cookbook.ipynb`

[15] `https://github.com/google/jax#random-numbers-are-different`

Appendix

We have performed our experiments on Adaptive Gradient Optimization, Generalization and Learnability using basic TensorFlow functions.

We have referred to the code in [14] to understand and perform experiments on infinite-width networks and Neural Tangent Kernels using the Neural Tangent library described in [8]. We have seen that the predictions infinite-width neural networks trained with gradient descent are Gaussian and can be computed using the Neural Tangent Kernel [6]. Our experiment consists of finding results using the Neural Tangent Kernel and compare it with results obtained by training an ensemble of neural networks.

We have used the JAX framework [15] to perform this experiment. JAX is a combination of Autograd and XLA which are used for fast differentiation of Python functions and Optimizing Compiler for Machine Learning on GPU/TPU respectively. We have performed this experiment on Google Colaboratory.

We begin this experiment by creating a simple synthetic dataset. The training data points are drawn from the process $y = f(x) + \epsilon$. Here, x, is drawn from a Uniform distribution in $(-\pi, \pi)$ and ϵ is Gaussian noise from $\mathcal{N} \sim (0, 1)$. The function $f(x) = sin(x)$ is chosen and the testing data is $y = f(x)$ for linearly spaced x in $(-\pi, \pi)$. The training set has 5 points and testing set has 50 points. The random values of x and ϵ are generated using the JAX function `random.split`.

We define a fully-connected neural network. This is done using the JAX library `stax`. The layers in `neural_tangents.stax` are a triplet of the functions `(init_fn, apply_fn, kernel_fn)` where the first two functions are the same as their `stax` equivalent but the third function, `kernel_fn`, computes infinite-width kernels corresponding to the layer. These fully-connected layers are created using the `Dense` in `stax`. The function `jit` is used to execute in a single call to the GPU. Here we have observed the several random draws from the parameters of the network.

Next, we perform Bayesian inference on this infinite-width neural network defined above using the `neural_tangents.predict.gradient_descent_mse_ensemble`. We observe the result of Bayesian inference with gradient descent for infinite time.

Further, we perform training on the network using the JAX optimizer. We define loss and gradient of the loss using the `grad` function. We train the network for 10000 steps and update the optimizer at each step. We store the train loss and test loss at each step and plotted the loss over training and the function after training compared with inference.

We compare the infinite width inference with training an ensemble of neural networks using the JAX function `vmap`. It is observed that the exact inference of the infinite-width neural network and the ensemble is very similar. This result is shown in Section 3.2.