An Experimental Study of the Relationship Between Generalization and the Bias Variance Tradeoff in Neural Networks

Kathryn Leung

Advisor: Professor Boris Hanin

Submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Science in Engineering

Department of Operations Research and Financial Engineering

Princeton University

June 2021

An experimental study of the relationship between generalization and the bias variance tradeoff in neural networks

by

Kathryn Leung

Abstract

Machine learning models have proven to be very useful for several different applications and fields of research, but their inner workings are often misunderstood. The complexity of a machine learning model can be loosely defined by how large it is, i.e. how many parameters it uses to make a prediction. A certain number of parameters is necessary for the model to represent complex functions. However, classical machine learning theory posits that increasing the number of parameters past a certain point causes overfitting, a pitfall of machine learning training where the model is too closely fitted to the data it was trained on and lacks the ability to generalize to unseen data. Models that are overfitted this way are called overparameterized.

The performance of overparameterized models is often assessed by measuring the test loss, which is closely tied to the bias and variance of the model, as the test loss is the sum of the bias squared and the variance. The idea of a bias variance tradeoff as the complexity of a model grows has permeated machine learning theory, with the bias being monotonically decreasing and the variance being monotonically increasing as the complexity of the model grows. However, recent works have suggested that this classical bias variance tradeoff may not extend to all model classes, particularly overparameterized neural networks.

Neural networks are a specific class of machine learning models that have recently gained traction for their extraordinary performance, yet they still remain largely mysterious. They typically have more parameters than are necessary to fit the dataset on which they are trained. According to classical machine learning theory, this would mean that they are unable to generalize well. Yet, overparameterized neural networks have found great success for a wide range of tasks. This ability of overparameterized neural networks to generalize with minimal error has confused researchers.

In this thesis, I will conduct experiments on neural networks in various settings, and explore to what extent the classical bias variance tradeoff holds. I will reproduce experiments from other papers on the bias variance tradeoff in neural networks and probe the robustness of past work done in this area. I will also see to what extent the bias variance tradeoff changes for neural networks with different architectures and datasets, and explore theoretical results about the bias variance tradeoff to see if our empirical results are backed by theory. Lastly, I will see if my empirical results can be connected to phenomena in the generalization of overparameterized neural networks, such as the posited double descent curve. In doing so, I hope to take steps towards elucidating the phenomenon of extraordinary generalization performance by overparameterized neural networks.

Acknowledgements

I would first like to thank my advisor Professor Hanin for his invaluable mentorship on this thesis, and for guiding me through a beautiful introduction to deep learning and its many intricacies. Professor Hanin's support and encouragement throughout this process made me feel like I was writing a thesis for myself, instead of for an obligatory graduation requirement. I could not have asked for a better advisor.

I am pleased to acknowledge that the work reported on in this paper was substantially performed using the Princeton Research Computing resources at Princeton University which is a consortium of groups led by the Princeton Institute for Computational Science and Engineering (PICSciE) and Office of Information Technology's Research Computing.

I would also like to thank Connor for his endless support on this project. For getting me set up with VSCode and tigergpu, and helping me put the finishing touches on this report, but most of all, for being my loudest cheerleader. Quite frankly, much of the work in this thesis would not have been possible without him.

Thank you to Alvin, Karolen, and my other friends for empowering me through this process and for bringing lots of joy and laughter into my life in difficult times.

Most importantly, I would like to thank my family for supporting me through Princeton, for shaping me into the person I am today, and for loving me unconditionally. This thesis is dedicated to them.

Contents

1 Introduction

Neural networks have recently become the ubiquitous machine learning model to use for an expansive range of predictive modeling tasks. They are often used as black box models, and people tend to excessively increase the size of them in the hopes of increasing their predictive power. The networks are then overparameterized, meaning there are more parameters than necessary to interpolate over the observed data. Classical machine learning theory would posit that this is a naive approach, because these networks would be overfitted to the training data, as they have memorized the data instead of learning any meaningful patterns, and would not be able to generalize to unseen data. Despite this intuition, overparameterized neural networks have achieved massive success in generalization for fields such as computer vision and natural language processing.

The extraordinary generalization performance of overparameterized neural networks is a multifaceted topic of research, much of which still remains largely misunderstood. This phenomenon has been experimentally validated several times over and has been approached from many different angles in an attempt to explain it theoretically, from analyzing the optimization properties of the algorithms used in training to examining the implicit biases of the network. The wealth of research spans networks of different architectures and sizes.

An emerging theory for describing the test loss, a measure of how well a model generalizes to an unseen data set, with respect to the model complexity is called the "double descent curve". This theory directly contrasts with the standard U-shaped notion of the way test loss would behave as a function of model complexity [2] and implies that the classical bias variance tradeoff may not necessarily extend to neural networks. [3] claim that test loss cannot be extrapolated to bias and variance without deeper investigation. Their work further probes the behavior of bias, variance, and test loss in overparameterized neural networks and claims that the classical bias variance tradeoff does not apply; they find that both quantities decrease as the complexity of the network grows. They also put forward a decomposition of the variance of the network to explain this behavior. [1] further claim that the bias is monotonically decreasing, while the variance takes on a unimodal shape as the complexity of the network increases. Additionally, they connect the bias and variance curves to the double descent phenomenon.

These experimental results require further validation in order to be fully convincing. There are also newly emerging theories that suggest the double descent curve is not the entire story behind the shape of the test loss curve, which begs the question of whether the bias and variance curves can be connected to these theories as well. Elucidating this bias variance tradeoff could change its pervasive narrative in machine learning theory and have implications on how complexity and generalization of neural networks is understood.

This thesis will start by reviewing the most relevant background information for general machine learning models, neural networks and the bias variance tradeoff. Then, it will cover the current discussion surrounding the generalization of overparameterized neural networks and take a fine grained look at the role that the decomposition of test loss into bias and variance plays in this discussion. A deeper dive will then be taken as I explore these concepts experimentally. The implications of the experimental results will be unpacked in the larger generalization context, and then several avenues for research will be discussed.

1.1 Machine Learning Models

This thesis will be investigating supervised machine learning models. There are five main components to the pipeline for supervised machine learning: data acquisition, model selection, defining model fitness, specifying a learning rule (optimization) for training, and testing [4]. Data acquisition simply refers to the collection of the data that you wish to model, and splitting the data into a train and test set. Model selection is the process of choosing a model class that you think will best be able to learn the data. To define model fitness, a loss function is introduced. A loss or risk function keeps track of the average prediction error over the data points, i.e. the average distance between the prediction and the actual label. Train loss refers to the value of the loss function over the train set, and test loss refers to value of the loss function over the test set. The learning rule is how the parameters of the model get updated as the model observes the data and trains. In training, a supervised machine learning model observes the train set of data points and their associated labels, and the optimizer helps the model learn the function that governs the relationship between the data points and their labels. The last step of this pipeline, testing, is the process of applying the trained model to the test set that it did not observe in training and seeing if it generalizes well.

Supervised machine learning models all contain the same key elements:

1. $\mathcal{D}$ represents the dataset
2. $\mathbb{E}_{\mathcal{D}}$ represents the expectation taken with respect to the dataset
3. $y(x)$ represents the predicted value given x
4. $h(x)$ represents the optimal prediction given a value of x
5. $p(x)$ represents the probability density function value of the datapoint x
6. $p(x, t)$ represents the joint probability density function value of the datapoint x and its function value t, which arises from the intrinsic noise on the data

1.2 Neural Networks

Neural networks are a class of machine learning models that have recently become massively popular for solving a wide range of tasks in various fields of research, like question-answering in natural language processing and object identification in computer vision. The idea behind neural networks is to simulate the networks present in our brains that learn how to make predictions and draw conclusions based on the information that they have.

The simplest architecture for a neural network is a fully-connected feedforward neural network, which consists of connected layers of neurons (also called nodes), where the input to one neuron is a linear combination of the outputs from the neurons in the previous layer (Figure 1). These networks are also sometimes called MLPs (Multi-Layer Perceptrons) or DNNs (Deep Neural Networks). The first layer is the input layer, which takes in the data, and the last layer outputs the prediction. The layers between the first and last layer are often referred to as hidden layers.

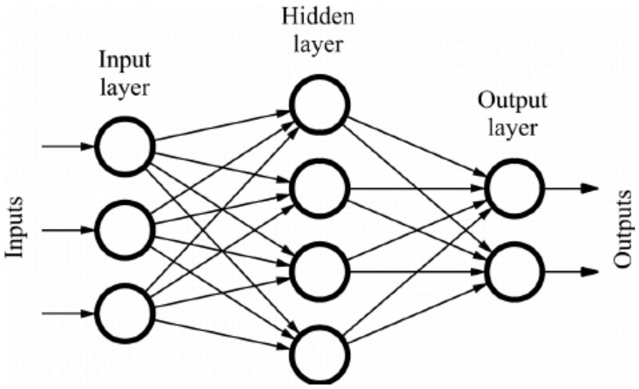


Figure 1: Feedforward neural network with a single hidden layer. Reproduced from [5].

They are essentially directed acyclic graphs, with a weight (parameter) associated with each edge. At each node, an activation function is applied on the linear combination of the output values from the previous layer weighted by their respective edge weights. A bias term is also sometimes included. These weights are learned in training using stochastic gradient descent algorithms and backpropagation.

The complexity of a single hidden layer feedforward neural network is the number of neurons in its hidden layer. As the number of neurons in the hidden layer increases, the network can learn and express more complex functions. This thesis will conduct experiments on fully connected neural networks with either one or two hidden layers.

1.2.1 Underparameterized vs. Overparameterized Regime

When a neural network is underparameterized, the number of parameters it has is less than the number of samples in the training data, so the model cannot interpolate over the data. A neural network is overparameterized when the number of parameters is greater than the number of samples in the training data. When a network is overparameterized, it should be able to achieve 100% training accuracy. Increasing the number of parameters in a neural network is equivalent to increasing its complexity. Classical machine learning theory posits that if the model complexity is too high, the network will overfit to the training data and will not generalize well, i.e. the test loss will be high (Figure 2). The value for model complexity at which this learning curve achieves a local minimum is optimal.

However, this is not the case for overparameterized neural networks. The explanation for this phenomenon is probed by [7, 2] on state of the art network architectures. Modern machine learning theory instead posits that the learning curve for overparameterized neural networks actually exhibits a double descent, and that networks with complexity beyond the interpolation threshold (the point at which the model goes from being underparameterized to overparameterized) actually have very good generalization performance. As seen in Figure 3, the test loss, begins to decrease again after reaching the interpolation threshold.

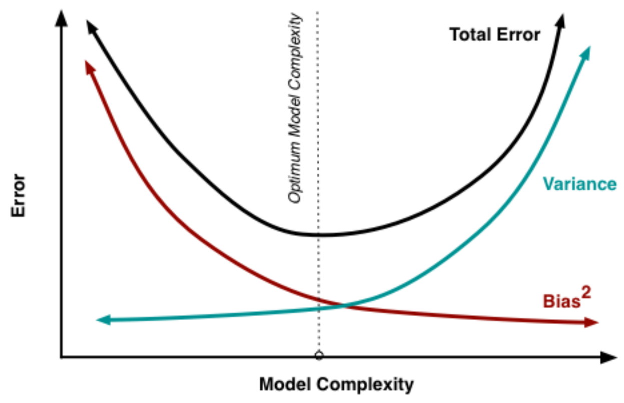


Figure 2: Classical machine learning theory learning curve in the underparameterized regime, including bias variance tradeoff. Test loss is referred to as Total Error here. Reproduced from [6].

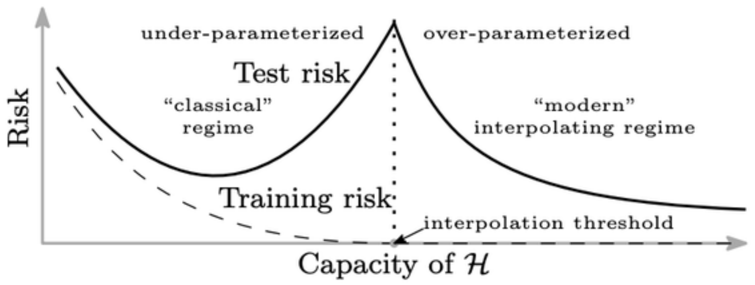


Figure 3: Modern machine learning theory double descent curve. Test loss is referred to as Test risk here. Reproduced from [2].

1.2.2 Deep Neural Networks

Neural networks with multiple layers between the input and output layers are of particular interest because their complexity not only grows as the width of the hidden layers increases, but also as the number of hidden layers increases. Overparameterized deep neural networks have seen exceptional generalization performance but are notoriously difficult to train. When investigating the generalization of overparameterized neural networks, it is important to consider all possible avenues for increasing the complexity of the neural network and how they affect the network performance differently, which is why experimenting with the depth of the neural network often goes hand in hand with experimenting with the width [3, 1].

1.2.3 Neural Network Training

There are hyperparameters and training configurations that are standard practice in neural network training. I will be reviewing the relevant ones used in this thesis to illuminate the intentional design behind my experiments.

While there are many activation functions out there, the most ubiquitous function to use on hidden layers is the ReLU activation function [8].

$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

This means that the output is $ReLU(w \cdot x)$ at each node in a hidden layer, where w is the vector of weights along the edges from each node in the previous layer to this node and x is the vector of output values from the previous layer.

Two important output activation functions I used in the experiments are the sigmoid function, which is defined for a scalar as

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}, \tag{1}$$

and the softmax function, which is defined for a vector as

$$\sigma(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}, \tag{2}$$

where K is equal to the number of classes in a classification problem and the length of $\vec{x}$. When using the sigmoid activation function, the output of the network is $S(w \cdot x)$. For the softmax activation function, there will be K nodes in the output layer, and the softmax is applied on each node such that the final output for the i-th node is $\sigma(\vec{x})_i$ where each $\vec{x}$ is the vector of output values.

These two functions map the values for each class to a value between 0 and 1, so they can be interpreted as the probability of a data point belonging to those classes. The softmax function will output a vector whose length is the number of classes in the classification problem, with each value being the probability that the data point belongs to that class and the output probabilities from all classes summing to 1.

The sigmoid function is a special case case of the softmax function, where there are only two classes. It's clear to see this when the sigmoid function is rewritten for two classes.

$$\sigma(\vec{x})_1 = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} = \frac{e^{x_1 - x_2}}{e^{x_1 - x_2} + 1} \tag{3}$$

If we define $x = x_1 - x_2$, then it is clear to see from equations 1 and 3 that the softmax function in a binary classification problem is equal to the sigmoid activation function. This means the output only needs to be one value, i.e. the probability of the data point belonging to some class, where it can be inferred that the probability of the data point belonging to the other class is just 1 minus the output probability.

An important training configuration to set is the loss function and the way that the gradient is updated in training. Some popular optimizers include Adam, RMSprop, Adadelta, but I decided to use stochastic gradient descent (SGD) for this thesis because it is the most easily interpretable and is most often analyzed in the literature. I also decided to use mean squared error (MSE) loss, again because it is more easily interpretable and is most often analyzed in the literature. MSE loss is discussed in more detail in Section 1.3.1. The formula for SGD is

$$w_{t+1} = w_t - \eta \nabla L(w_t) \tag{4}$$

where η is called the learning rate.

It is common to add momentum to the training procedure, which accelerates the training towards the minimum of the loss function. The equations are as follows

$$v_{t+1} = \rho v_t + \nabla L(w_t)$$
$$w_{t+1} = w_t - \eta v_{t+1}$$

The way parameters are initialized is very important as well. Nowadays, the most standard initialization to use is the He uniform initialization. The Kaiming He initialization takes into account the non-linearity of the activation function, specifically ReLU. For the Kaiming He uniform initialization, the weights in a layer are initialized as such

$$w \sim U\left(-\sqrt{\frac{6}{n^l}}, \sqrt{\frac{6}{n^l}}\right) \tag{5}$$

where n^l is the number of inputs to the layer, and $U(a, b)$ is a uniform distribution on the interval $[a, b]$. Lastly, to combat overfitting, weight decay, also known as regularization, is often implemented. Most commonly used is the l2-regularization which adds the norm squared of the parameters times some constant to the loss function. The loss function with l2-regularization is

$$L = L_{mse} + \lambda ||w||^2 \tag{6}$$

I note that these training configurations were mostly inspired from that of [1].

1.3 The Bias Variance Decomposition

In the evaluation of neural networks and other machine learning models, the test loss can be decomposed into two quantities called the bias and variance. I will now discuss the conflicting intuitions behind the way these quantities behave as the complexity of the model increases.

1.3.1 Standard Take

The mean squared error is a common loss function used to gauge the performance of a machine learning model. The goal in training is to minimize this value. The mean squared error on a data set can be broken down into three parts; bias, variance and noise from the data. Recalling the notation introduced in Section 1.1, the mathematical expressions are as follows [9]:

The expression for expected mean squared error loss is

$$\mathbb{E}[L_{mse}] = \int \{y(\mathbf{x}) - h(\mathbf{x})\}^2 p(\mathbf{x})\, d\mathbf{x} + \int \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\, d\mathbf{x}\, dt \tag{7}$$

The integrand from the first term on the RHS can be rewritten as follows, by defining it for a single data point $\mathbf{x}$ and particular dataset $\mathcal{D}$, then taking the expectation with respect to the dataset $\mathcal{D}$.

$$\mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x};\mathcal{D}) - h(\mathbf{x})\}^2\right] = \{\mathbb{E}_{\mathcal{D}}\left[y(\mathbf{x};\mathcal{D})\right] - h(\mathbf{x})\}^2 + \mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2\right] \tag{8}$$

There are no cross terms on the RHS of this equation because, as it turns out, when we take the expectation with respect to the dataset, they simplify to

$$2\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}, \tag{9}$$

which is equal to 0. The first term on the RHS of equation 8 is the (bias)^2 and the second term is the variance. With some rearranging, the expected squared loss can thus be reformulated as expected loss = (bias)^2 + variance + noise where

$$(\text{bias})^2 = \int \mathbb{E}_{\mathcal{D}} \left[\{y(\mathbf{x};\mathcal{D}) - h(\mathbf{x})\}^2 \right] p(\mathbf{x})\, d\mathbf{x} \tag{10}$$

$$\text{variance} = \int \mathbb{E}_{\mathcal{D}} \left[\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2 \right] p(\mathbf{x})\, d\mathbf{x} \tag{11}$$

$$\text{noise} = \int \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\, d\mathbf{x}\, dt \tag{12}$$

Bias quantifies how much the model assumes about what the data looks like and consequently, what the best solution for modeling the data looks like, and is mostly attributed to the model class. More precisely, bias measures on average how good the prediction is for a fixed data point. Variance captures how much the model can generalize to different data that comes from the same distribution as the training data, and measures how sensitive the learned model is to the data set it observed in training. When put into words, equation 11 is explicitly the probabilistic notion of variance for $y(\mathbf{x};\mathcal{D})$ with respect to the dataset, averaged over the data points. The intrinsic noise from the data is present for all machine learning models and impossible to reduce, in contrast to the bias and variance. [10]

It has been deeply embedded in classical machine learning theory that more training and greater complexity of models increases the bias and decreases the variance, thus introducing a tradeoff between bias and variance. Low bias, high variance is associated with overfitting, and high bias, low variance is associated with underfitting [10]. Ideally, it would be best to be able to minimize both bias and variance.

For example, models like linear regression models have high bias and low variance because they expect that the data will be linear, and when the data is linear, the model predictions will not vary by too much across datasets from the same distribution. When the data comes from a more complex function though, a linear regression model will severely underfit this data. A support vector machine would have low bias, high variance because it can perfectly fit every data point in a given training set. It only makes the assumption that the data is separable, but will not be able generalize as well to other datasets from the same distribution since the other training sets will likely contain points that corrupt the originally learned boundary.

1.3.2 Modern Take

The results from [3] suggest this tradeoff may not completely capture the relationship between bias and variance for overparameterized neural networks. They measure the bias and variance for single hidden layer neural networks trained on a variety of tasks and datasets while varying the width of the hidden layer and functionally increasing the complexity of the model. According to the canonical bias variance tradeoff, increasing the width of their networks would cause the bias to decrease, and the variance to increase. However, the empirical results from [3] indicate that both bias and variance decrease as the width increases. I will either validate or disprove their results, in addition to those from [1] that build off of this modern take on the bias variance tradeoff. I will also probe their work further to glean new insights about the bias variance tradeoff in neural networks and investigate the implications this might have for more general deep learning and generalization performance.

1.4 Methodology

I will first conduct experiments that demonstrate the double descent phenomenon and bias variance trade off in machine learning models to motivate the peculiar generalization behavior that requires further probing. I will then do a rigorous test for robustness of the results presented in [1] by reproducing a subset of the experiments. To reproduce the experiments from [1], I use the MNIST and CIFAR10 datasets which are publicly available and easily downloadable. I will also extend these experiments to more datasets for different tasks such as sentiment classification, as well as my own randomly generated datasets (polynomial regression, pure noise). I will also explore whether theoretical results can be synthesized to explain the behavior of the bias and variance in the neural networks from our experiments. I will ultimately try to unpack the implications of a modern take on the bias variance tradeoff on newly emerging theories about the generalization of neural networks.

2 Related Works

There are four relevant questions to ask when investigating the generalization phenomenon for overparameterized neural networks.

1. How can the phenomenon be described?
2. How robust is this phenomenon? Does it hold for for different kinds of network architectures, optimizers, or data sets?
3. How long does it take for the phenomenon to manifest? Can the phenomenon be achieved in reasonable training times? It is important to know whether this phenomena can be practically exploited.
4. Why does the phenomenon occur?

The following works seek answers to these questions and motivate the work done in this thesis.

2.1 The Mystery of Overparameterized Neural Networks

Classical machine learning theory leads us to expect that the test loss of a neural network will initially decrease as the model trains and complexity increases, and then the test loss increases as the network overfits to the training data. The optimal point at which to stop training would be the minimum of this curve. However it has been widely observed that the test loss curve of overparameterized neural networks takes the shape of a double descent curve. To be precise, the test loss descends again when the complexity transitions from the underparameterized to the overparameterized regime at the interpolation threshold. This phenomenon has been extensively probed by [2]. They conducted experiments to verify that this behavior is robust across data sets.

Conducting experiments is an extremely useful approach for trying to answer these questions. It is useful for investigating the conditions under which the phenomenon manifests and is a key component for empirically validating any theoretical claims that might be made about the robustness, speed, and explanation behind the phenomenon. [11] conduct experiments on models with various fully-connected architectures, optimizers, and other hyperparameters, on four different computer vision datasets to investigate the relationship between complexity and generalization. Using two metrics of complexity related to sensitivity to input perturbations, they explore the sensitivity and generalization of trained networks when varying hyperparameters. They ultimately find that sensitivity metrics are correlated with generalization.

[12] hone in on defining the generalization phenomenon precisely for a two layer ReLU neural network by investigating training speed and exactly what functions the networks can learn and generalize to. They analyze training and generalization with respect to random initialization, using the properties of a related kernel to track the dynamics. They investigate the learnability of a broad class of smooth functions for the networks trained via gradient descent, and make improvements on a tight characterization of training speed and generalization bounds. They find that linear functions are learnable by overparameterized two layer ReLU neural nets, and also test their findings experimentally on the MNIST and CIFAR10 datasets. Along that same vein, [13] study two layer overparameterized ReLU neural networks for multi-class classification via stochastic gradient descent from random initialization. They find that the networks trained with SGD exhibit good generalization if the network has the capacity to fit random labels.

A crucial part of probing the explanation for the generalization phenomenon is examining the behavior of the training loss, in addition to the test loss. For example, [7] conduct extensive experiments to show that deep neural networks can easily fit completely random labels in training, resulting in zero training error. With respect to optimizers used in training, randomly initialized first order methods like gradient descent and stochastic gradient descent have been found to achieve zero training error on non-convex and non-smooth objective functions. [14] show that as long as the number of hidden nodes in a shallow network with ReLU activation is large enough, and no two inputs are parallel, randomly initialized gradient descent converges to a globally optimal solution at a linear convergence rate for the mean squared error loss function. [8] also study the problem of training deep neural networks using gradient descent and stochastic gradient descent. They find that with the proper random weight initialization, both gradient descent and stochastic gradient descent can find the global minima of the training loss for an overparameterized deep ReLU network, given that the training data satisfies some assumptions.

Knowing the conditions under which these networks are able to achieve this extraordinary generalization performance then begs the question, how long will the networks have to train to achieve such performance? This is where a multitude of studies on convergence analyses and theories have emerged. [15, 16, 17]. [18] examine the classes of functions that two and three layer overparameterized networks can learn. They claim that the learning can be done by SGD or its variants in polynomial time using polynomially many samples.

These results motivate further research into why these phenomena occur because knowing these phenomena are robust on practical time scales makes them readily exploitable in practice. In this thesis, I will be further probing the explanation for the generalization phenomenon. If the phenomenon is to be exploited in practice, knowing why it occurs will be crucial towards our advancement of knowledge in the realm of neural networks and machine learning.

The bias variance tradeoff is intimately linked to the test loss curve, as the test loss is the sum of the bias squared and the variance. The bias variance tradeoff is not cohesive with the idea of the double descent curve, because the bias variance tradeoff only offers a reasonable explanation for the shape of the test loss in the underparameterized regime. The behavior of the bias, variance, and test loss curves in the overparameterized regime has recently become a research topic of interest because of its possible implications on the double descent curve. Analyzing the bias variance decomposition of the test loss gives us a finer grained look into where the shape of the curve is actually coming from, and might offer us some insight as to why the double descent phenomenon is ultimately occurring.

2.2 Bias Variance Decomposition

[1] measure the bias and variance of modern deep neural networks trained on commonly used computer vision datasets. As a mainline experiment, they trained a ResNet34 network on the CIFAR10 dataset, but they vary the architecture, loss function, and datasets, although all datasets are within the computer vision realm. They observe three different patterns for the risk curve — monotonically decreasing, double descent, and unimodal — and claim that the shape of the risk curve depends on whether the bias or the variance is dominant. They most commonly observe the monotonically decreasing shape for the risk curve, a result of the bias dominating the overall risk. They posit that the variance exhibits a unimodal shape as the number of parameters grows, and claim that the unimodal shape of the variance curve can be reconciled with the double descent curve when bias and variance dominate in different regimes, which they manufacture by adding label noise to the training set and thus increasing the variance.

They also explore the role of the depth of the network and test the models on out-of-distribution samples. They observe the same trends on out-of-distribution samples (using dataset CIFAR10-C). To explore the effect of model depth on bias variance, they run experiments using ResNet18 and ResNet50 in addition to their mainline

ResNet34. From these experiments, they observe that with increasing depth, the bias decreases and the variance increases. They conduct their experiments in the random-design setting, which aligns with our previous decomposition of bias and variance. Most other theoretical analyses hold for the fixed-design setting, so they present a theoretical extension from the fixed-design setting to the random-design setting, considering two-layer linear networks. They derive precise expressions for bias and variance and find that the risk of the model grows from unimodal to monotonically decreasing as the number of samples increases. This theoretical analysis supports their empirical observations of a unimodal variance curve and monotonically decreasing bias curve.

[19] propose a fine-grained bias variance decomposition and claim that the work from [1] is a special case of this decomposition. They point out that [1] consider total bias and variance, but do not try to decompose the variance like in [3]. Meanwhile, [19] describe an interpretable and symmetric decomposition of the variance into terms that isolate the randomness from sampling, initialization, and labels to allow for a more detailed analysis of the source of the double descent phenomenon. They steer away from the ambiguities of using the law of total variance to decompose the variance and instead require self-consistency under marginalization with respect to all variables. Their decomposition allows for marginalization of the variance over all sources of randomness but also accounts for the interactions between the sources of randomness in such a way that the decomposition is fully symmetric.

Following an application of their decomposition for random feature kernel regression, they propose that bias decreases monotonically with network width. They also claim that the behavior of variance curves is not monotonic and can actually diverge at the interpolation boundary without label noise, although label noise can emphasize this divergence. This kind of behavior in the bias and variance would lead to a double descent in the test loss. Ultimately, [19] claim that this divergence is caused by the interaction between sampling and initialization randomness, and while label noise can emphasize the double descent phenomenon, it is not the cause of it. This claim is slightly in conflict with the claim from [1] that label noise leads to double descent. The claims of [19] are supported by their theoretical analyses that yield divergences when they move from a bivariate decomposition to a trivariate decomposition of the variance. They see that the divergence comes from the "variance explained by the parameters and training data together beyond what they explain individually" and more specifically, the variance explained by the parameters and data inputs, and variance explained by the parameters and label noise.

[19] dispute the conclusions drawn from the multivariate decomposition approach from [3]. The way in which they decompose the total variance into variance due to sampling and variance due to optimization is misleading, they claim, because it omits the effects from the interactions between sampling and optimization. Because of the way they apply the law of total variance, conditioning on either source of randomness leads to different conclusions on the origin of double descent, which is part of the motivation for the symmetric decomposition.

2.3 Beyond Double Descent

While [19] find that their symmetric variance decomposition is consistent with the double descent phenomenon, I would like to explore whether it can be consistent with the multiple descent idea suggested by [20]. [20] claim that in the setting of linear regression, the number and location of peaks in the test loss curve can be manipulated. They also claim that this is not intrinsic to the model, but rather due to the interaction between the properties of the data and the inductive biases of the learning algorithms. Using their claim, I hypothesize that this multiple descent phenomenon could be observed in overparameterized neural networks as well.

To support their claim, [20] analyze the underparameterized and overparameterized regimes and find that in both regimes, the number of and location of the peaks can be controlled through a "feature-revealing process." In particular, they consider a dataset of arbitrarily large dimension, and find that as they increase the number of features from this data that they use for the linear regression (equivalent to increasing model complexity in the linear regression setting), an ascent will be created if the new feature follows a Gaussian mixture distribution. A new peak (ascent followed by a descent) will be created if the new feature follows a standard Gaussian distribution. In the underparameterized regime, they are only able create descents after an ascent (i.e. a local maximum of the generalization loss), but in the overparameterized regime they are able to arbitrarily create ascents and descents, thus showing that the shape of the generalization curve past the interpolation threshold is fully arbitrary and customizable to any degree. These kinds of generalization curves are rarely observed in practice, so from their analysis, they conclude that the shape of the generalization curve is due to the interactions between the properties of the data and the inductive bias of the learning algorithm.

[21] builds off the idea of the double descent curve by rigorously analyzing the test loss in the setting of kernel regression with the Neural Tangent Kernel to more closely characterize the behavior of neural networks than the canonical analyses done on simpler linear or kernel regressions. They claim that there are additional peaks and descents beyond the double descent curve in a superabundant parameterization regime — when the number of trainable parameters is greater than the number of training samples squared — for Neural Tangent Kernel regression, and equivalently for a single layer neural network trained with gradient descent. Through precise high-dimensional asymptotic analysis, they conclude that a first peak occurs when the number of parameters is equal to the number of samples, which approximately aligns with the double descent curve, and a second peak occurs when the number of parameters is

equal to the square of the number of samples. Lastly, they support their claims with empirical evidence that a triple descent curve actually manifests for a single-layer, fully-connected, finite-sized neural network trained with gradient descent.

2.4 Into the Deep

While this thesis will not be conducting any experiments on deep neural networks beyond two hidden layers, they are still a key part of the generalization picture. Deep neural networks with an arbitrarily large number of hidden layers are of particular research interest as they are harder to train but can also attain exceptional generalization performance. The deep residual network (ResNet) [22] architecture uses a residual learning framework that makes ResNets easier to optimize and achieve significantly higher accuracy than other deep networks. [14] compare deep networks of different architectures: fully-connected feedforward, ResNet and convolutional ResNet. They analyze the conditions under which gradient descent achieves zero training loss on these overparameterized networks, and find that the convergence rate is linear for the fully-connected feedforward network and the ResNet architecture.

3 Experiments

3.1 Computational Details

Tigergpu is a Dell computer cluster comprised of 320 NVIDIA P100 GPUs across 80 Broadwell nodes, each GPU processor core has 16 GB of memory. The nodes are interconnected by an Intel Omnipath fabric. Each GPU is on a dedicated x16 PCI bus. The nodes all have 2.9TB of NVMe connected scratch as well as 256GB RAM. The CPUs are Intel Broadwell e5-2680v4 with 28 cores per node [23]. All of my experiments were run on these tigergpu nodes, but I also used Google Colab to prototype and debug my experiments.

When choosing which machine learning library to use for my experiments, I decided to opt for Keras, even though the open source code from [1] is written in Pytorch. For the straightforward DNNs I was using in my experiments, the Sequential API of Keras was the clear choice over PyTorch. Keras is more intuitive and easier to read. For example, fitting a network in Keras is a one-liner, while training a network in Pytorch requires many steps (initialize gradients, forward pass, backward pass, loss computation and weights updates). Keras also uses GPU by default, while it explicitly needs to be set up in PyTorch.

3.2 Experiments Outline

I will first present the double descent phenomenon and bias variance tradeoff with a linear regression model. I will model a randomly generated polynomial function with perturbations. To generate the data, I'll use 20 evenly distributed points in the range of [-1, 1]. To generate the labels for the data, I'll use random integers for the coefficients of the polynomial, then generate error terms following a normal distribution centered at zero and add those to the polynomial function value for each data point. I will use a random 50/50 train/test split on the data, and train the models using the SGDRegressor class from scikit-learn [24]. I will then measure the average bias, variance, expected loss for the models over many runs, while increasing the degree the data points.

I will then conduct experiments to validate the work done in [1, 3, 2]. Specifically, I will reproduce the MNIST experiment from [1].

I will then train fully connected single hidden layer feedforward neural networks with varying hidden layer width. The network will be built with ReLU/sigmoid activation functions, and trained using stochastic gradient descent and mean squared error loss. I will be training the models on the IMDB sentiment classification task, a noisy polynomial regression task, and pure noise datasets to verify the robustness of the claims from [1], as they only present results from computer vision tasks.

Then, I will conduct experiments on networks with two hidden layers for two datasets, CIFAR10 and a randomly generated polynomial regression task, while varying the widths of the hidden layers. [1] found that the variance curve from their experiments on ResNets of different depths was unimodal. We will see whether this claim is robust for a DNN. I would also like to see whether the triple descent curve from [21] manifests for the polynomial regression task by extending the number of trainable parameters to the number of samples squared.

The function I wrote to perform the bias variance decomposition for most of these experiments is heavily based on the `bias_variance_decomp` function from the Python package mlxtend [25], except I implemented the variance reducing unbiased estimator from [1]. Instead of calculating the bias, variance, and average test loss over some number of fitted models trained on randomly selected training sets, the training data is randomly split into some number of disjoint sets, n, and the bias, variance, and average test loss are computed from the models trained on these n independent sets. This process is repeated for a number of trials, N, and then the values for bias, variance, and average test loss are averaged over these trials. This means that for each model complexity, $n \times N$ copies of the model will be trained. I also rewrote the function to be more memory efficient, because as I was prototyping experiments on Google Colab, I found that there had been unresolved issues with a TensorFlow/Keras memory leak using the `predict()` function. This issue persisted on tigergpu as well. However, by rewriting my `bias_variance_decomp` function to reinitialize the model within the training loop, clear the Keras backend memory in each iteration, and use `predict_on_batch()` instead of `predict()`, I managed to resolve this issue. Lastly, I rewrote the function to account for classification problems with multiple classes when the output prediction is multidimensional.

For each of the trained $n \times N$ models, the average test loss is calculated by taking the squared norm of the difference between the prediction of the model and the correct output for each test point, and averaging this value over the number of test points. This value is averaged over all $n \times N$ models.

To calculate the unbiased variance from one trial, I take the squared norm of the difference between the average prediction taken over the n independent splits and the prediction from each n model for each test point, averaged over the number of test points, and then average again over the number of trials minus one. The final variance value is averaged over all N trials.

The bias squared value can be calculated by taking the difference between the average test loss and variance value for each model complexity.

This method for calculating these quantities comes from [1]. I implement this method in my own `bias_variance_decomp` function for all of the experiments except for the experiment in Section 3.3, in which I use the function from the library mlxtend [25].

3.3 Demonstrating the Double Descent Phenomenon and the Bias Variance Tradeoff on Linear Models via SGD

The data consists of 20 evenly spaced data points between 0 and 1 and the label for each data point is the polynomial function value perturbed by some error term. I generated a random polynomial of degree 5 with integers in the range of -5 to 5 as coefficients and normally distributed error terms for each data point. The polynomial, $-4 - 1x - x^2 + 4x^3 - 5x^4 - 5x^5$, is in Figure 4, with error terms that were generated from a normal distribution of mean 0 and variance $\frac{1}{2}$.

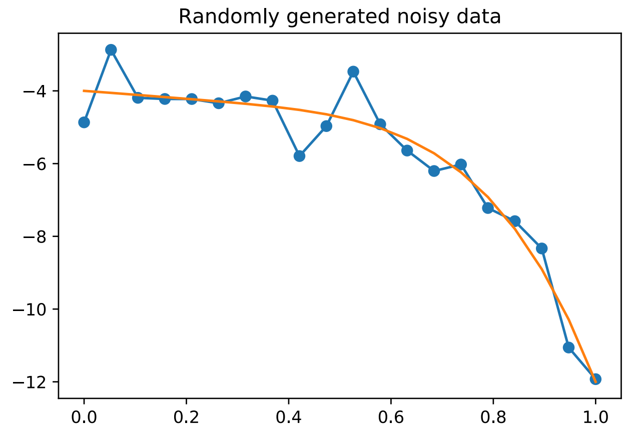


Figure 4: Randomly generated data and degree 5 polynomial for polynomial regression experiment, pictured with 20 total data points.The orange line is the true polynomial function, while the blue line connects perturbed polynomial function values, and the blue dots are the data points.

I randomly selected 10 points to make up the training set for this experiment. Then, I fit linear models using increasing degrees of these data points, from 1 to around 20. For example, for a data point x and degree p, the input to the model is a vector $(x^0, x^1, x^2, ..., x^p)$. The training of the model is done by stochastic gradient descent with MSE loss and l2 regularization with a parameter value of 0.0001 [24]. The model trains for either a maximum of 1000 epochs or until the current loss is within a small tolerance level of the minimum observed loss for 5 consecutive epochs. The increasing degree of the data points simulates increasing complexity of the data and the machine learning model. I measured the bias, variance, expected loss using the mlxtend `bias_variance_decomp` [25] from 500 training rounds for each degree fit (Figure 5).

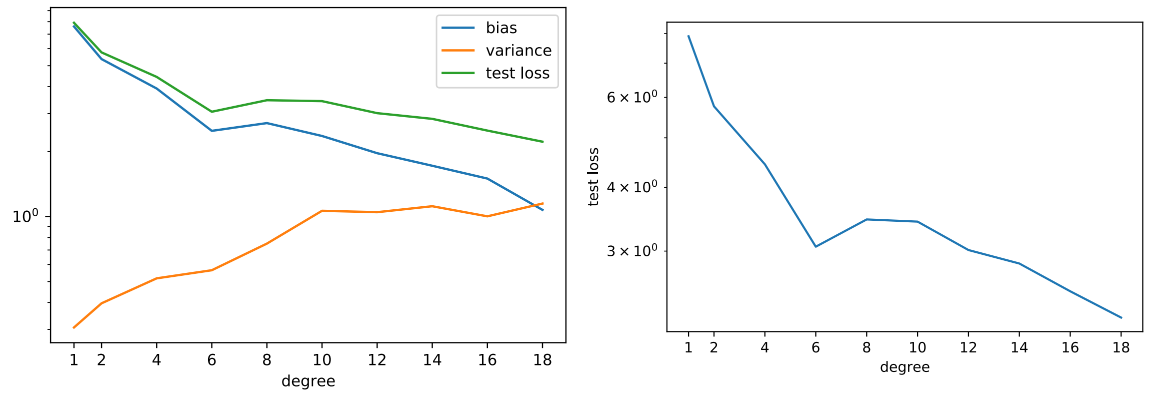


Figure 5: Results from polynomial regression experiment via SGDRegressor

It is clear to see from Figure 5 that a double descent has manifested in the test loss curve with its peak at around degree 10 which aligns with what we might expect, since there are 10 samples in the training set. The variance of the model is extremely low, which is also to be expected with a linear model. As the degree of the data increases, the variance increases while the bias decreases, which aligns with the classical bias variance trade off. However, we see that the bias and variance dominate the test loss for different degrees, which is how the double descent curve is manifested.

I ran a experiment with a different polynomial, $-3+4x-3x^2-1x^3+1x^4-4x^5$ (Figure 6), with the rest of the parameters the same, except I used 20 evenly distributed data points between -1 and 1, instead of 0 and 1.

There is also something of a bias variance tradeoff present here (Figure 7). When the degree is less than 8, the bias decreases while the variance increases. However, the behavior of the curves is less predictable beyond that. A double descent also emerges in the test loss.

We see that the peak instead occurs at around degree 5, which is the degree of the

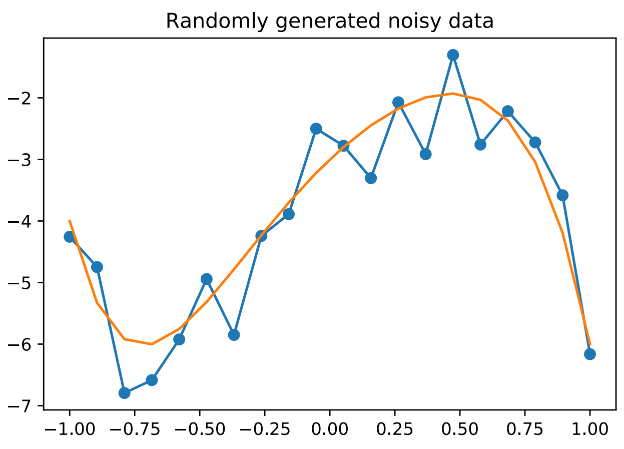


Figure 6: Randomly generated data and degree 5 polynomial for polynomial regression experiment, pictured with 20 total data points. The orange line is the true polynomial function, while the blue line connects the perturbed polynomial function values, and the blue dots are the data points.

polynomial. This raises an interesting question of whether the interpolation threshold occurs when the number of parameters is equal to the degree of the polynomial or the number of data points. There is also a slight triple descent in the test loss, but I would attribute the second peak to random perturbations.

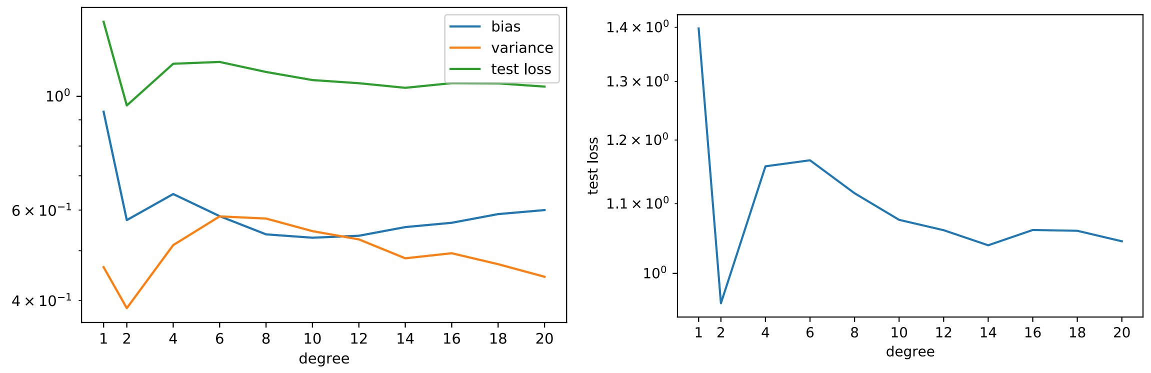


Figure 7: Results from second polynomial regression experiment via SGDRegressor

From these experiments, we see different behaviors from the test loss, bias, and variance even though the modeling task and training setup is the same, albeit with a different polynomial function. In the first experiment, the bias was roughly monotonically decreasing, while the variance was roughly monotonically increasing. In the second experiment, we see more of a unimodal shape to both the bias and variance curve. As mentioned previously, there are different ways to treat the interpolation threshold, which is exemplified by these two experiments. It is difficult to draw any substantial conclusions about the interpolation threshold and the behavior of the test loss, bias, and variance curves from these experiments due to their random nature, but it is sufficient to say that the classical bias variance trade off is not a completely robust theory.

3.4 Reproducing MNIST Experiments from [1]

Before conducting my own experiments on neural networks, I wanted to verify the robustness of the experiment framework I built in Keras against the PyTorch framework used in [1]. I started with closely examining and trying to replicate the MNIST experiments from [1] because of the simplified network architecture and the consistent unimodal variance and monotonically decreasing bias results.

The MNIST dataset consists of 60,000 training images and 10,000 testing images of handwritten digits. The 28x28 images are greyscale and are very popular for use in machine learning research [26]. The pixels of each image are flattened into a 784 dimensional input vector to be compatible with a feed forward neural network architecture.

The architecture for the MNIST experiment networks in [1] is a fully connected network with one hidden layer and ReLU activation, and a softmax activation function for the output. The weights are initialized with the Kaiming He uniform distribution. It is trained using SGD with MSE loss and momentum of 0.9. The learning rate is initially 0.1, but drops by a factor of 0.1 every 100 epochs. They also use weight decay with a parameter of 5e-4. They train the network for 200 epochs on 6 independent splits of 10,000 training samples and repeat this for 5 rounds to calculate the average bias, variance and test loss. This process is repeated and the average bias, variance, and test loss values are recorded and plotted for varying widths of the neural network.

I initially left out the momentum, learning rate scheduler and weight decay. I added in these features one by one, and my resulting plot was roughly a good match to theirs until I added the weight decay. This motivated me to build the framework to run the full MNIST experiment from [1] on tigergpu. As far as I can tell, I ran the experiment with the exact configurations that were reported. The resulting plot looked slightly different than the plot from their paper, which is reproduced in Figure 8.

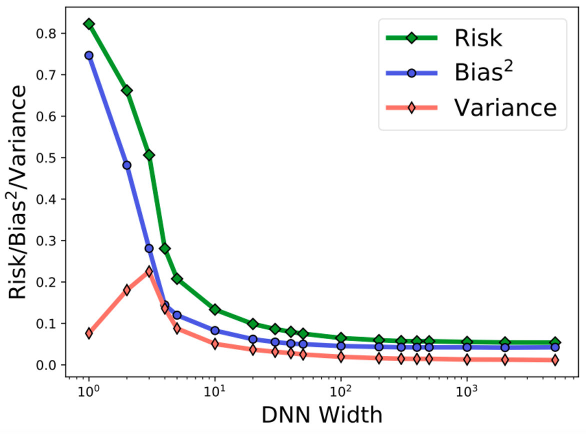
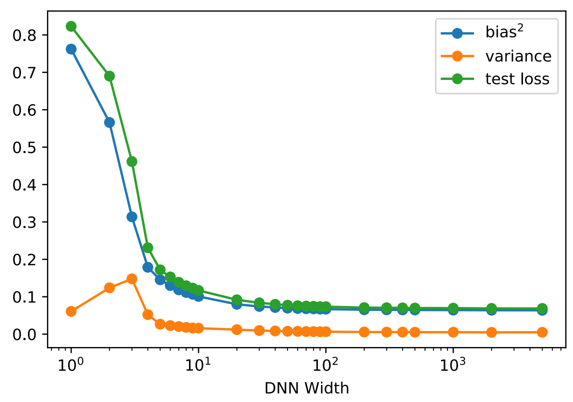


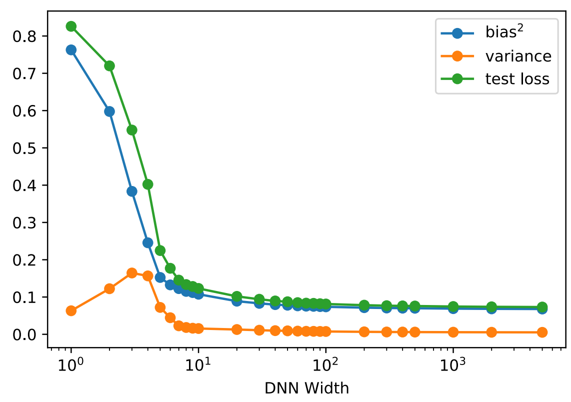
Figure 8: MNIST plot from [1]

I ran both my own framework and the framework for the MNIST experiments from [1] with different configurations to investigate this anomaly. I kept all other factors consistent with how they are described above, except I ran the two frameworks with and without weight decay. The results of this experiment are shown in Figure 9.

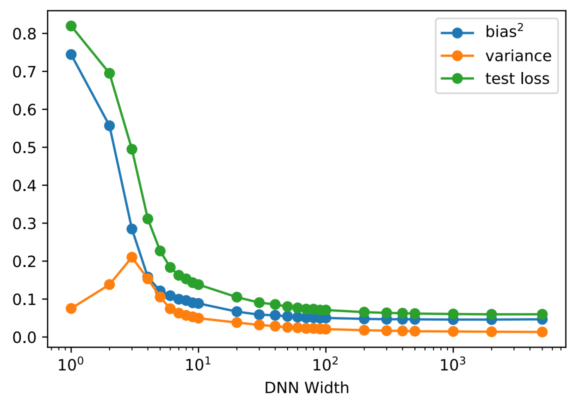
The results were consistent with what I was expecting. The amplitude of the variance is not as large as is presented in the paper when the frameworks were run with weight decay. For this reason, I decided not to run any more experiments with weight decay.



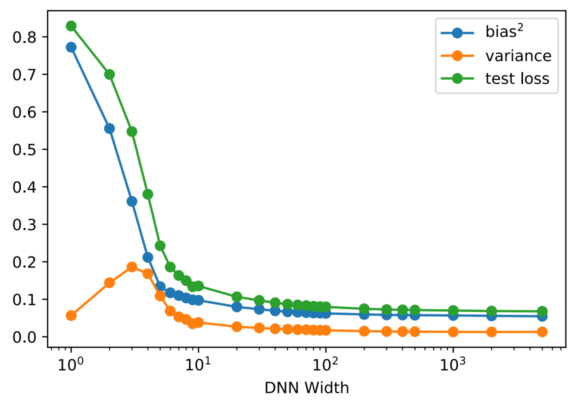
(a) [1] with weight decay



(b) Keras framework with weight decay



(c) [1] without weight decay



(d) Keras framework without weight decay

Figure 9: Comparison between resulting plots from [1] code and my experiment framework using Keras

[1] reported that their experiments were run with weight decay, but the results of my experiments would indicate that this might not have been the case. I also ran their code with the hyperparameters as they reported and the plot from their paper does not match the resulting plot from this run. It seems that they may have misreported this detail.

The results from my experiments and those from [1] are qualitatively the same. My results are shown in more detail in Figure 10. The test loss and bias are monotonically decreasing, while the variance is unimodal.

However, in repeatedly running these experiments, there were some microscopic scale issues that persisted. The test loss doesn't seem to fall as quickly as it does in

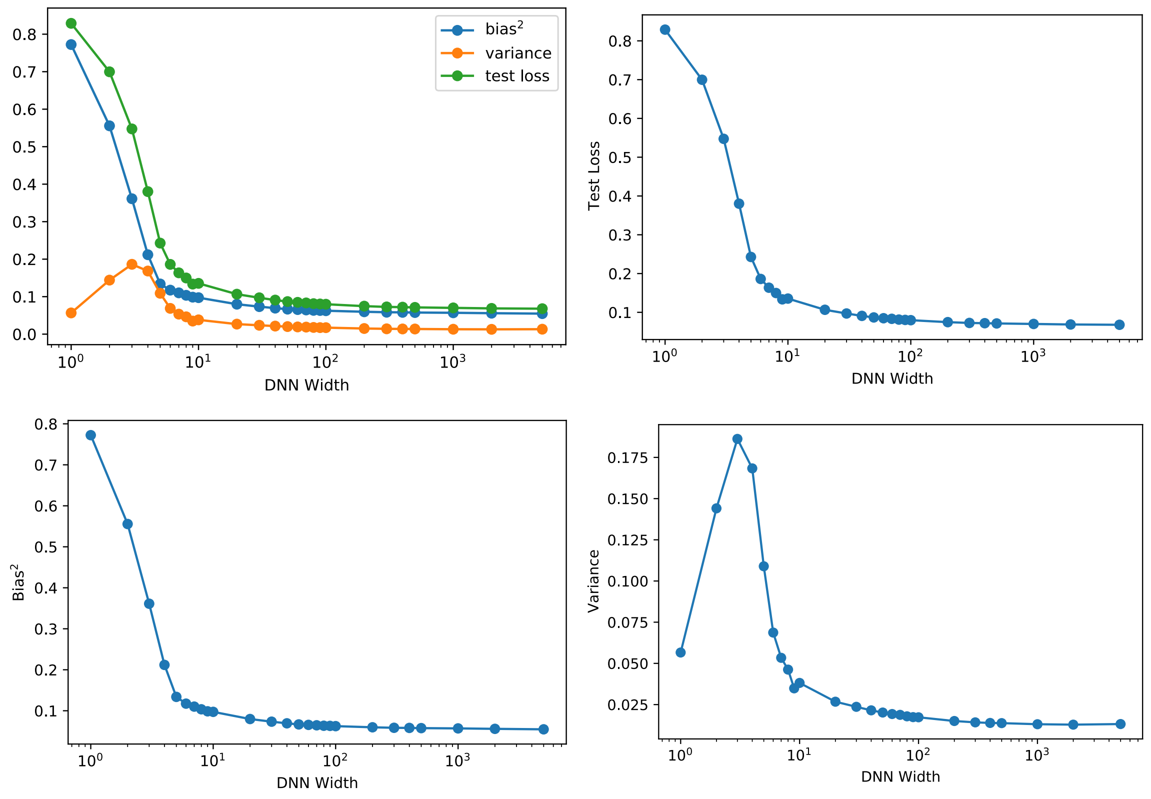


Figure 10: Test loss, bias, variance from my MNIST experiment

the experiments from [1]. For example, the values for bias, variance, and test loss are consistently lower from the [1] experiments at width 4 than they are from my experiments. This is the only consistently significant difference between the plots from my experiments and from the [1] experiments, and I am not sure what the cause of this might be.

3.5 NLP Model

After verifying my experiment framework against that of [1], I decided to run experiments on a NLP dataset, since all of the experiments from [1] had been run on computer vision datasets. I wanted to verify that the claims from [1] are robust across different types of data.

The IMDB dataset comes with 25,000 training data points and 25,000 testing data points of movie reviews from users that are labeled as either positive or negative. The data comes preprocessed as a sequence of indices that correspond to words that have been ranked by their frequency from the entire dataset [27]. The distribution of the ranked words is shown in Figure 11.

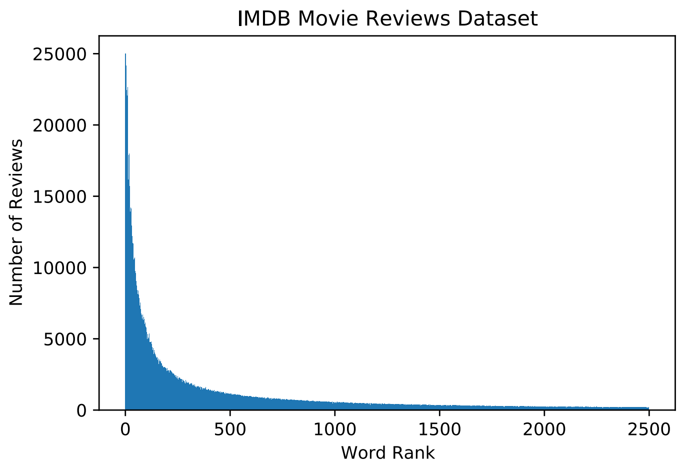


Figure 11: Distribution of ranked words from IMDB dataset

I built a fully connected network with one hidden layer using ReLU activation and sigmoid activation for the output layer. I initialized the weights of the layers using the Kaiming He uniform initializer. The network is trained by MSE loss using SGD optimizer with a learning rate of 0.1 that decreases by a factor of 0.1 every 100 epochs and momentum of 0.9. I trained the network for 200 epochs with a batch size of 128. I trained the network on 2 independent splits of size 12500 from the training data, and repeated this for 5 trials to calculate the average bias, variance and test loss. This process is repeated for different widths of network's hidden layer. I used approximately the same widths for the hidden layers as [1] on their MNIST experiment.

I initially tried two different preprocessing steps on the data. Upon visual inspection of Figure 11, I decided to use only the top 1500 most frequent words. I first vectorized the reviews using the Bag of Words approach. Then, I tried capping the reviews at

500 words with the vocabulary consisting of the 1500 most used words in the dataset, and adding a Keras Embedding layer with a dimension of 32.

The weights of the Keras Embedding layer get trained as part of the fit process. While the Embedding layer does not inherently add any information to the model, it is a part of the learning process and contributes a new representation of the sentences in training that may be helpful in making predictions.

I decided against the network with the Embedding layer in because the train and test performance was not as good, and this approach was not as intuitive as the Bag of Words method. It would also be difficult to isolate the effects of this Embedding layer on the experiment results.

To run this experiment, I used my bias variance decomposition function to measure the bias, variance, and average test loss while varying the width of the hidden layer of this network from 1 up to 10000 and training on the data as explained above. The results of this experiment are shown in Figure 12 and 13.

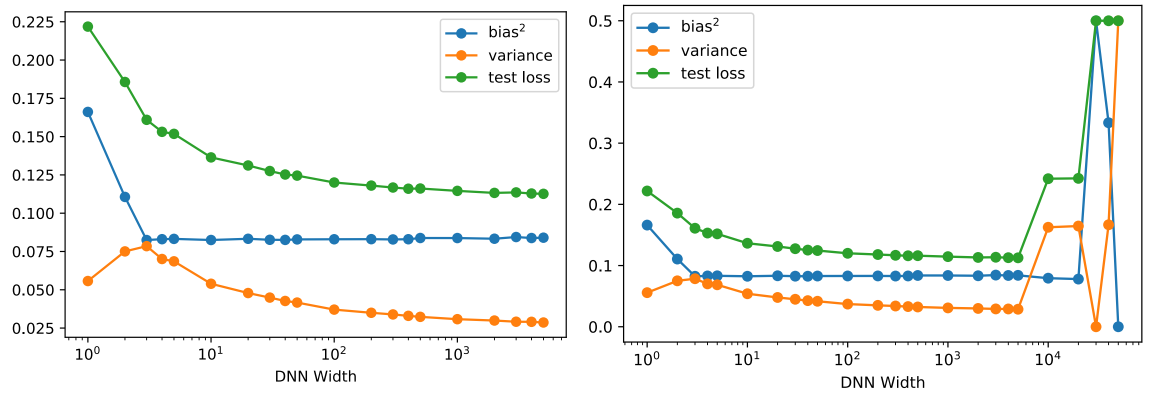


Figure 12: Results from IMDB experiment, plotted without and with the largest widths

These plots beg the question of why the bias, variance, and test loss are not monotonically decreasing for the largest widths. With the experimental design being consistent to the MNIST experiments, which I verified against the results from [1], the behavior

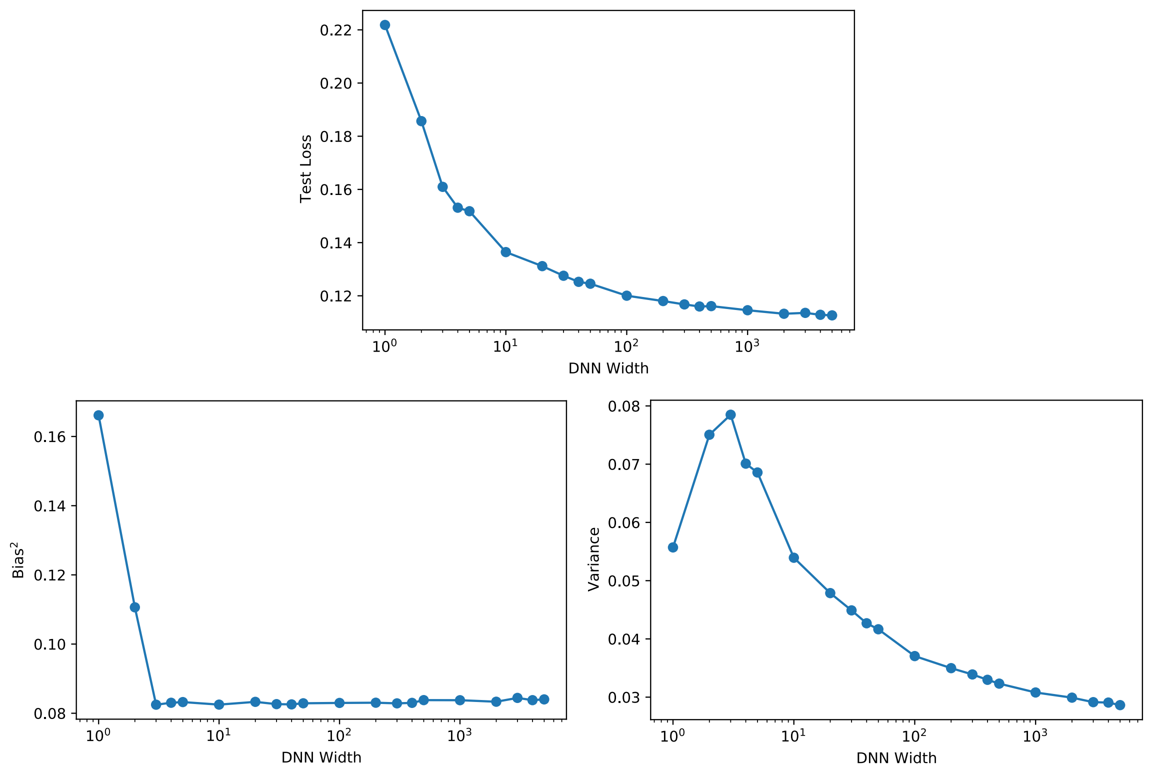


Figure 13: Test loss, Bias, variance from IMDB experiment, without the largest widths

from the bias, variance and loss at the largest widths was unexpected. It seems that the model became too complex for this problem, often plateauing at a train and test accuracy of 50%, meaning the model was randomly guessing the labels.

The training loss and train accuracy would either plateau at 0.5 in training, or train to zero loss on each trial for the larger widths. We can see that for the last few widths, the test loss is consistently 0.5 which means the test loss was 0.5 for every trial. The behavior of the bias/variance is interesting, but I do not think any meaningful patterns can really be gleaned from it. We can thus conclude the training becomes unstable at the largest widths; it would be interesting to investigate what properties of the optimizer are making the training unstable.

These results also beg the question of whether the optimizer was stable for the MNIST experiments from [1]. I ran their experiment with a few added widths, and there were no resulting irregularities, so I did not include those plots in my discussion of the MNIST experiments.

It is clear that without the largest widths, the curves behave as we would expect them to. The loss is monotonically decreasing and the bias is also monotonically decreasing within a small error margin, while the variance takes on a unimodal shape. Therefore, we can conclude that the general patterns reported in [1] for the MNIST experiment are consistent for data coming from the realm of natural language processing.

However, the specific behavior of the curves was slightly different from what we observed in the MNIST experiment. We can see that for the first few DNN widths, the classical bias variance tradeoff holds, as the bias decreases and the variance increases. Beyond that, the variance dominates in driving the test loss down as the bias is roughly constant. I hypothesize this might be because of the Bag of Words vector representation of the data. The sparse representation of the data introduces an implicit bias that can not even be overcome by increasing the complexity of the model beyond a certain point.

3.6 CIFAR10 Noise Experiment

To simulate an experiment on a pure noise data set, I will be corrupting an increasing percentage of training labels from the CIFAR10 dataset until they are ultimately completely random. I will be measuring the corresponding bias, variance, average loss of the models at different complexities, as was done in the label noise experiments from [1], but I will be running this experiment on a DNN instead of a ResNet architecture and driving up the noise corruption to 100% instead of just 20%. A DNN is certainly less well equipped to properly model the CIFAR10 data than a ResNet, but this is a worthwhile experiment for comparing DNN performance on the CIFAR10 and MNIST datasets.

The CIFAR10 dataset is 60,000 32x32 color images with 10 labels: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. There are 50,000 training images and 10,000 test images [28].

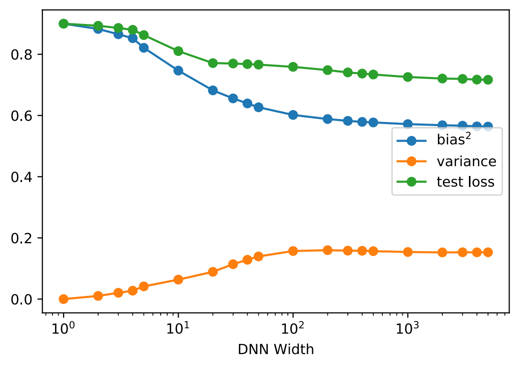
3.6.1 One Hidden Layer

I started with a DNN with one hidden layer. The experiment framework is very similar to the framework for the MNIST experiment. The pixel values for each image are first flattened into a 3072-dimensional input vector, and the activation function on the hidden layer is ReLU, with a softmax output activation function. I used the Kaiming He uniform initialization on the weights, with no bias terms. I trained the network using SGD with momentum of 0.9, and a learning rate scheduler that initialized the learning rate as 0.1 and decreased by a factor of 0.1 every 100 epochs. I increased the number of training epochs from what I used on MNIST to mirror the growing complexity of the data, training the network for 500 epochs with a batch size of 128.

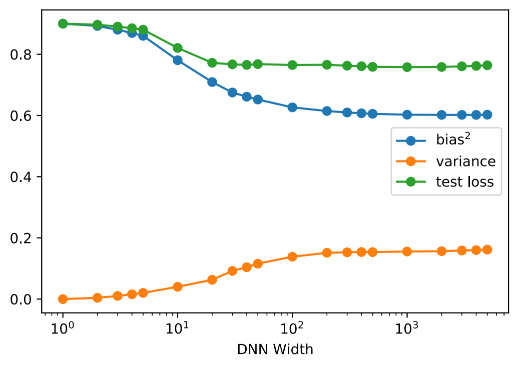
To corrupt the labels, I randomly selected the appropriate percentage of labels from the training data and changed them to a uniformly selected label from the ten options.

I randomly sampled 5 independent splits of 10000 data points from the training set to calculate the average expected test loss, bias, and variance and averaged the values from this process for 5 trials. I recorded and plotted these values for increasing widths of the hidden layer.

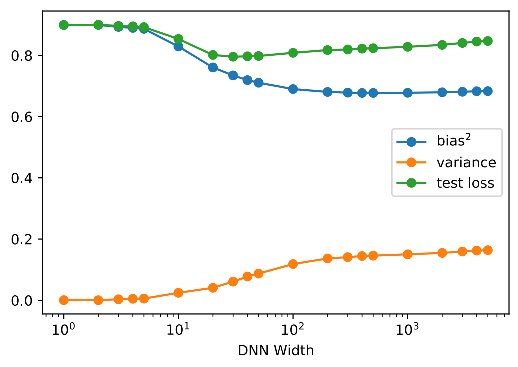
I conducted the bias variance experiment as described above on the CIFAR10 dataset, and repeated it 5 different times at varying levels of corruption, i.e. 0%, 25%, 50%, 75%, 100% corruption of the training labels. The results of this experiment are shown in Figure 14. The test accuracy is low, as is to be expected from a DNN.



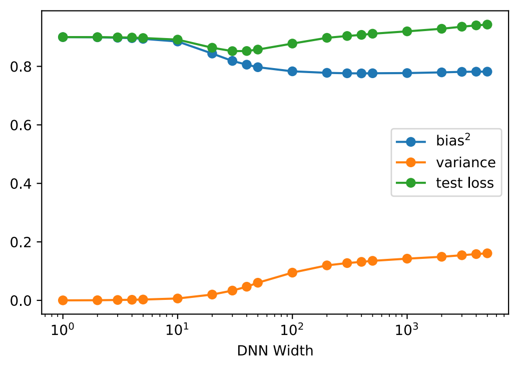
(a) 0% noise corruption



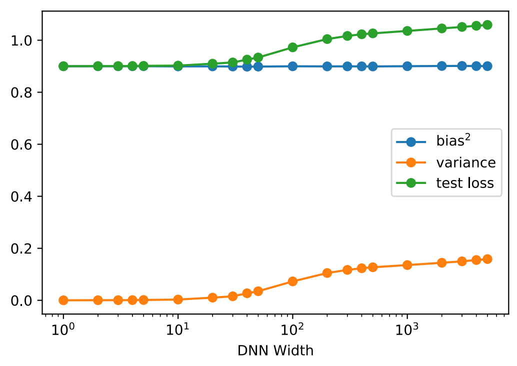
(b) 25% noise corruption



(c) 50% noise corruption



(d) 75% noise corruption



(e) 100% noise corruption

Figure 14: Results from CIFAR10 noise experiment via one hidden layer neural network at 0%, 25%, 50%, 75%, 100% label corruption

As expected, the test loss increases instead of decreases at the larger widths as the level of noise corruption increases.

3.6.2 Two Hidden Layers

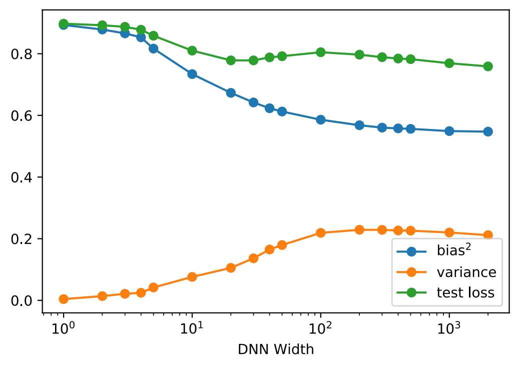
After running these CIFAR10 experiments on a single hidden layer DNN, I decided to try a DNN with two hidden layers to see if it could achieve better test performance.

The training configuration and architecture is exactly the same as from the experiment with one hidden layer, except I added another hidden layer of the same width as the first layer with ReLU activation to the architecture of the network for this experiment. The results of this experiment are shown in Figure 15. They look surprisingly similar to the results of the experiment with one hidden layer.

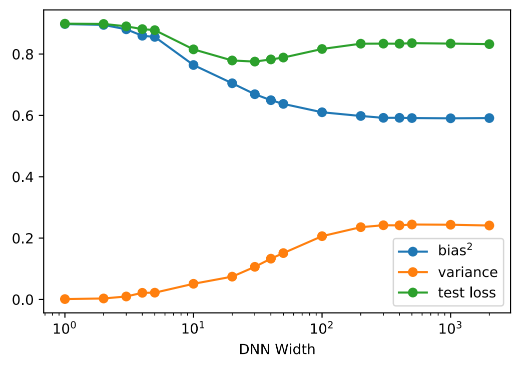
It is clear that a DNN is not a very strong model for the CIFAR10 dataset. While this is still a worthwhile experiment for the sake of exploration, it would also be useful to try a more robust model architecture for this noise experiment, similarly to how [1] used a ResNet.

For the one hidden layer experiment, we can see that at almost all levels of noise corruption, the bias is monotonically decreasing, while the variance is monotonically increasing. However, at 100% noise corruption, the bias is constant. We also see that although the loss is initially monotonically decreasing, as the level of noise corruption increases, the loss begins to increase after initially decreasing.

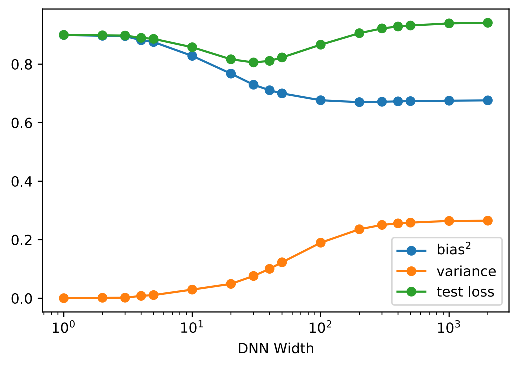
For the two hidden layer experiment, it is interesting that something of a double descent is present in the two layer experiment with 0% noise. We can see this is because the bias and variance dominate the loss at different widths. This double descent aligns with the idea from [1] of a unimodal variance and decreasing bias leading to double descent. This double descent neutralizes as the level of noise corruption increases, though, and the variance levels out instead of taking on a unimodal shape. We can see that the second descent also flattens as the level of noise corruption increases and at 100% noise corruption, the loss actually increases after initially being constant.



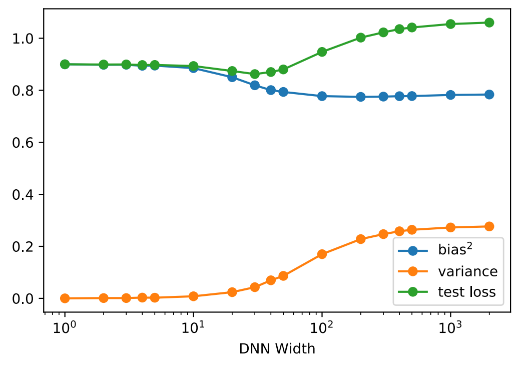
(a) 0% noise corruption



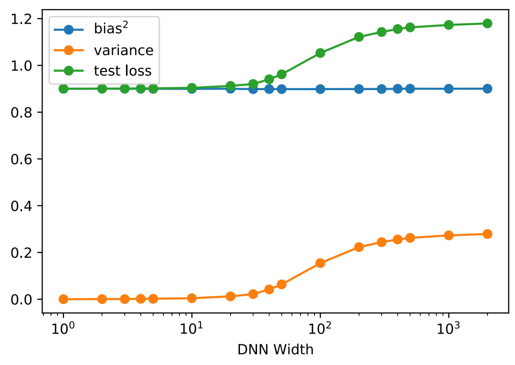
(b) 25% noise corruption



(c) 50% noise corruption



(d) 75% noise corruption



(e) 100% noise corruption

Figure 15: Results from CIFAR10 noise experiment via two hidden layer neural network at 0%, 25%, 50%, 75%, 100% label corruption

[2] claim that the point of interpolation is when the number of parameters is equal to the number of data points times the number of classes, which is nearly true for my two layer experiment. The number of parameters at width 100 is approximately 300,000 and at 50 and 200, the number of parameters are 100,000 and 700,000 respectively. While the peak of the test loss curve occurs at width 100, it would make the most

sense if it actually occurred at width 50, when the number of parameters is $100,000 = 10,000 \times 10$.

Future studies investigate why a double descent manifests for the two hidden layer model but not for the one hidden layer model. I would also research why the minimum and peak of the test loss curve occur at these certain widths.

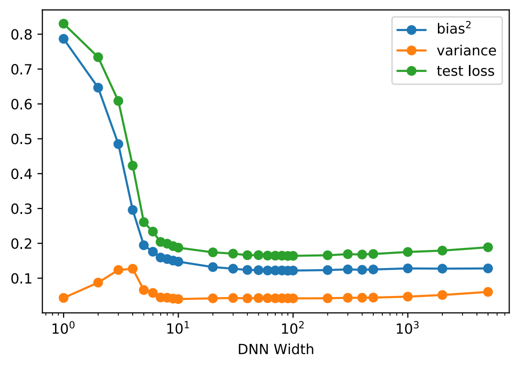
3.7 MNIST Noise Experiment

I also ran a label noise experiment on the MNIST dataset, which was not explicitly done in [1]. They claim that a double descent manifests in the loss curve when label noise is added unless the variance is not large enough to overpower the bias, which is the case for MNIST, but they did not include an experiment to verify this in their paper.

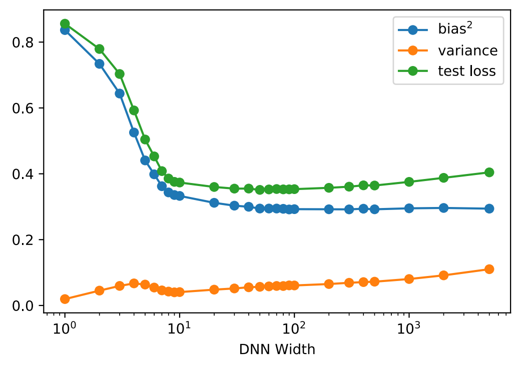
The experiment framework for this experiment is the same as it was for the MNIST experiment without label noise. The architecture for the network is a fully connected network with one hidden layer and ReLU activation. It is trained using SGD for MSE loss with momentum of 0.9, and a learning rate that is initially 0.1, but drops by a factor of 0.1 every 100 epochs. I train the network for 200 epochs on 6 independent splits of 10,000 training samples and repeat this for 5 trials. To corrupt the labels, I randomly selected the appropriate percentage of labels from the training data and changed them to a uniformly selected label from the ten options. I repeated this procedure for increasing widths of the hidden layer.

I conducted the bias variance experiment as described above on the MNIST dataset, and repeated it 5 different times at varying levels of corruption, i.e. 0%, 25%, 50%, 75%, 100% corruption of the training labels. The results of this experiment are shown in Figure 16.

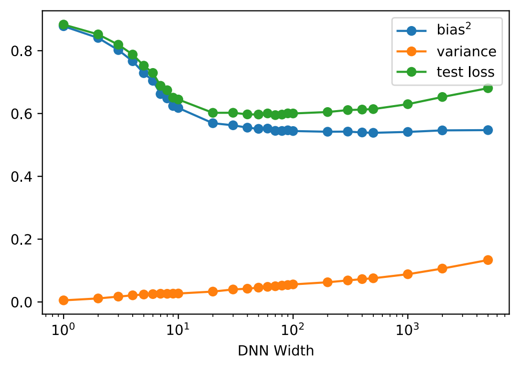
As expected from [1], the variance is not large enough to overwhelm the test loss and



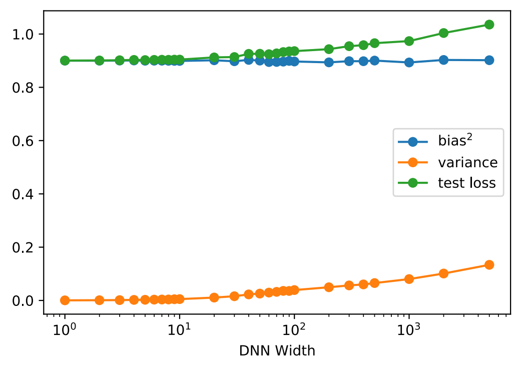
(a) 25% noise corruption



(b) 50% noise corruption



(c) 75% noise corruption



(d) 100% noise corruption

Figure 16: Results from MNIST experiment with label corruption. Figure with 0% noise corruption is not included for sake of repetition.

manifest a double descent. Instead, we see that as the percentage of noise corruption increases, the test loss changes from being monotonically decreasing to increasing after initially decreasing. At 100% corruption, the test loss is monotonically increasing, with the shape of the curve being dominated by the variance as the width of the DNN increases. This is consistent with what was seen in the CIFAR10 noise experiments.

This aligns with what we would expect. On a completely noisy training set, the model's guess will initially be random. As complexity increases and the model is able to better fit the noisy training data, the variance will increase because the model will have modeled a distribution based off the noisy training data, but it will not match the testing data distribution at all. Therefore the model will become more sensitive to the data set it observed in training. The bias will be constant because at all complexities, the average prediction for a fixed data point x will be a random shot in the dark.

3.8 Polynomial Regression via Neural Network Experiment

I also tested a neural network with two hidden layers with increasing widths — both layers of the same width — on the task of polynomial regression with label noise. I am using the polynomial regression to see how well the neural network approximates a simple nonlinear function at varying complexities and how this might differ from the other data types we have examined so far.

With a hidden width of 1, this would be nearly equivalent to running a linear regression with SGD, as we did in Section 3.3. We have seen linear regression with increasing complexity of the data, but now we will examine neural network with increasing model complexity on a regression task. This experiment is not very realistic but is surely an interesting comparison to the SGDRegressor experiment. In this experiment, I can further probe the claims made in [1] about the shapes of the bias, variance, test loss curves. I decided to use a two layer ReLU network for this experiment to test the robustness of the claim from [12] that linear functions are learnable by two layer ReLU networks.

I am running the experiment on a polynomial (degree 5) regression with Gaussian coefficients with a neural network with two hidden layers. The data set was built by generating 90 and 120 evenly spaced x values between -1 and 1, and randomly generating an error term for each x from a normal distribution with mean 0 and standard deviation of $\frac{1}{8}$. The data is shown in Figure 17.

I used one third of the x values as the test set, then randomly selected two evenly sized independent train sets from the remaining subset of the data. The input to

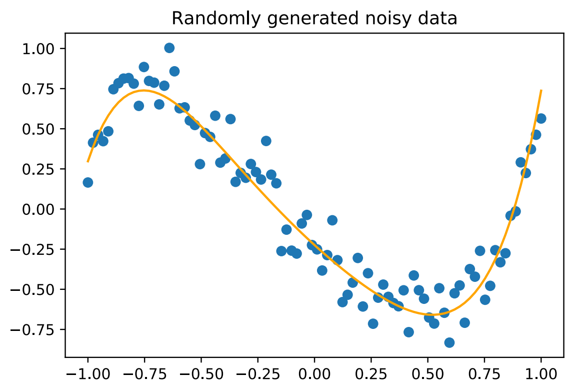


Figure 17: Randomly generated data and degree 5 polynomial for polynomial regression experiment, pictured with 90 total data points. The orange line is the true polynomial function, and the blue dots are the data points

the network for each data point, x, is the vector of the powers of x from 0 to 5. I repeated this for 30 trials to calculate the average bias, variance, and test loss. The activation functions on the two hidden layers are both ReLU, and a linear activation function is used on the output layer. I trained the networks using the Adam optimizer with MSE loss for 1000 epochs and a batch size of 1, because I found that SGD was unstable for this task. I repeated this procedure for increasing width of the two hidden layers.

I will be testing the robustness of the polynomial regression experiments to see if any discernible pattern emerges, like where the peaks and descents occur by adjusting the degree of the polynomial and number of datapoints. We will see whether a triple descent emerges.

I ran this experiment with two different training set sizes: 30 and 40. The results are shown in Figures 18 and 19, respectively.

The variance curves for this experiment are bimodal for both data set sizes instead of the unimodal shape that was observed in previous experiments. The different points at which the test loss and variance peak when trained on 30 and 40 data points implies that the test loss curve is somewhat customizable in this case [20]. As the

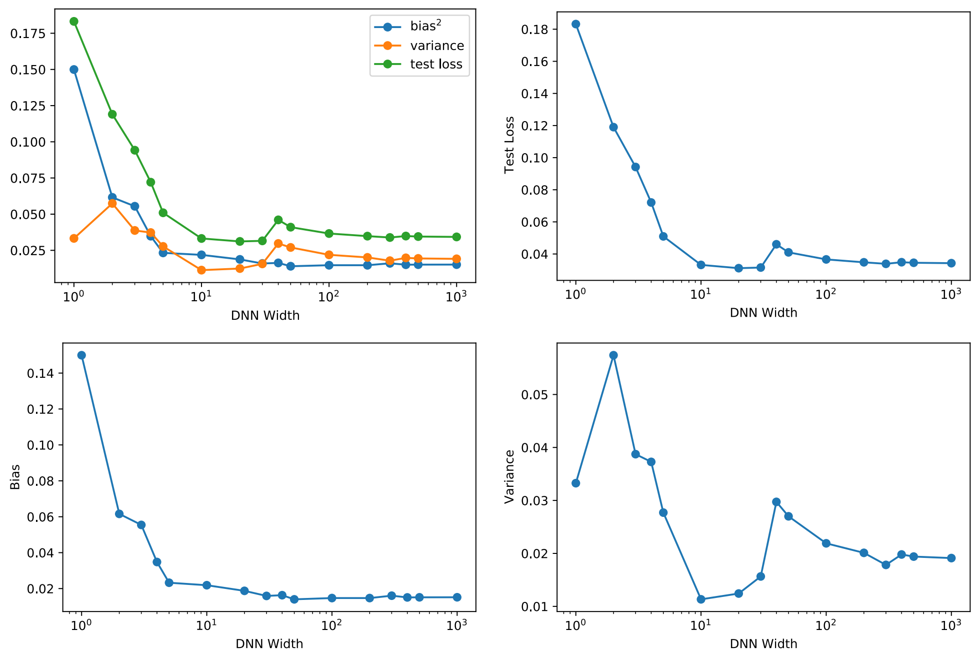


Figure 18: Results from polynomial regression experiment via two hidden layer neural network with 30 training data points.

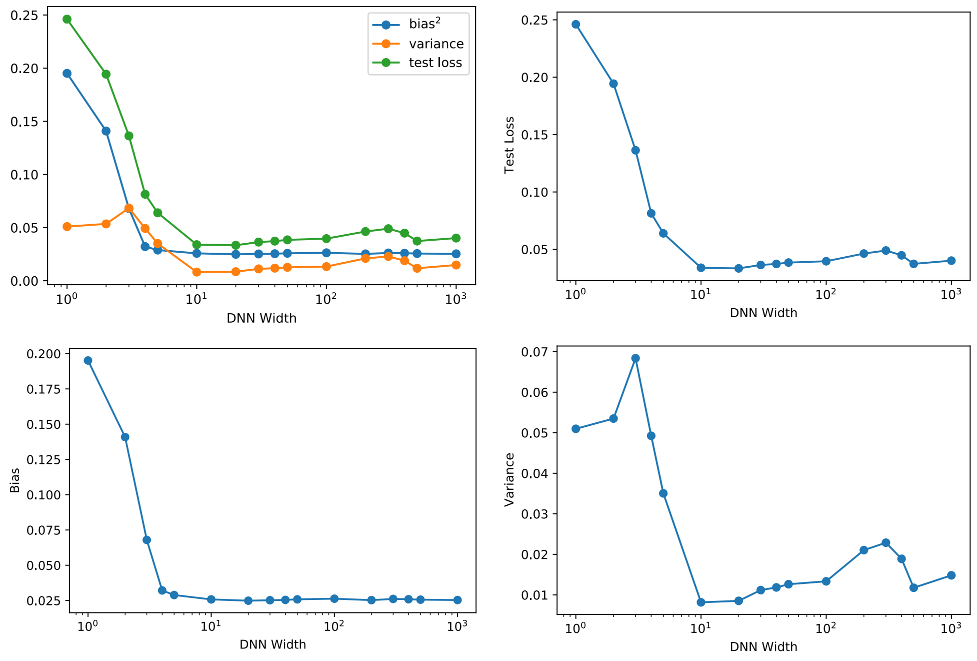


Figure 19: Results from polynomial regression experiment via two hidden layer neural network with 40 training data points.

number of training data points increases, the two peaks of the variance and the one peak of the test loss both shift to the right. While the peaks of the curves do not align with exactly with what we would expect to be the interpolation threshold, we would expect the location of the peaks to shift to the right as we increase the number of training samples.

The two peaks of the variance curve in the experiment with 30 training points occur at DNN Width 2 and 40, which correspond to 16 and 1,840 trainable parameters. The two peaks of the variance curve in the experiment with 40 training points occur at DNN Width 3 and 300, which correspond to 27 and 91,800 trainable parameters. In the experiment with 30 training points, the width at which the peaks occur are somewhat close to the linear scaling and quadratic scaling interpolation points from [21].

3.9 Drawing Conclusions from Experimental Results

In Section 3.3, we showed that even for a model as simple as a linear regression using SGD, it is difficult to draw precise conclusions in regards to double descent, the bias variance tradeoff, and the interpolation threshold.

Moving into the realm of neural networks, [1] claim that the test loss displays one of three patterns: monotonically decreasing, double descent, and unimodal. They also claim that double descent in the test loss can be explained by unimodal variance and monotonically decreasing bias. With more label noise, the variance will increase and the double descent test loss is observed.

In testing the robustness of these claims, I have found that the results of the experiments mostly aligned with what I would expect given these claims. From my MNIST experiment in Section 3.4, I was able to verify that the results and claims put forward by [1] for their MNIST experiment were robust to parameter changes such as weight decay and learning rate schedulers.

In the NLP experiment conducted by [2], they used a dense feature vector representation using word embeddings instead of a sparse feature vector representation like I chose to do. They observe a double descent in their test loss curve. However, from my NLP experiment in Section 3.5, I observed monotonically decreasing test loss and bias, and unimodal variance. This implies that either a double descent or monotonically decreasing test loss could manifest for natural language processing tasks. We conclude that the claims put forward by [1] about the shapes of the bias, variance, and test loss curves, are robust for natural language processing tasks from the results of my experiment.

I observe a monotonically increasing test loss and variance curve for a completely random and noisy dataset in the CIFAR10 and MNIST noise experiments in Sections 3.6 and 3.7. The addition of label noise did not manifest a double descent in my case, but rather generated new shapes of the test loss curve: monotonically increasing, and not unimodal, but rather uni-minimum.

[21] claim that double descent can occur without label noise, but that the divergence is often exacerbated by it. They also claim that the divergence is caused by variance terms due to the interactions between sampling and initialization. [1] observe from their experiments that the test loss is usually monotonically decreasing, but increases with label noise, thus producing a double descent. They claim that label noise leads to double descent, so the claims in these two papers are not exactly cohesive. Notice that, in my CIFAR10 noise experiment, a double descent has manifested even without the presence of label noise. In face, adding label noise actually eliminated the double descent, it rather flattened the bias curve and did not increase the variance peak. This is a nuance that is not captured in [1]. Without having measured the decomposed variance from [19], it is difficult to draw empirical conclusions about the cause of the

double descent, but it is clear that adding label noise does not always lead to double descent.

The variance curves from the polynomial regression experiments in Section 3.8 resemble the triple descent generalization phenomena observed in [21], but it is difficult to draw conclusions about these peaks without more consistent experimental results.

It is clear to see from these experiments that the classical take on a bias variance tradeoff leaves much to be desired. Instead of being monotonically decreasing and monotonically increasing, as the classical theory might suggest, the bias and variance actually take on a multitude of behaviors in a variety of experimental settings.

3.10 Possible Sources of Inconsistencies or Errors

The conclusions drawn from these experiments should be taken with a grain of salt, as there are several possible sources of inconsistencies or errors that could subject them to unreliable results or incorrect conclusions. In translating the work of [1] from PyTorch to Keras, a few possible inconsistencies arose while I was designing my framework that might explain any differences between my results and the results from [1].

The initialization of the networks can be controlled by the kernel initializer argument for keras, and the weights argument for scikit-learn SGDRegressor. The default kernel initialization is Glorot uniform, which is consistent with how the smallest network is initialized in [2]. I ended up initializing weights using the Kaiming He uniform distribution. As far as I can tell, this is how weights are initialized in PyTorch as well, but if that is not the case then this could potentially be a source of inconsistency. I will also be initializing the weights of the SGDRegressor to be normally distributed with a mean of 0 with small variance.

In adding weight decay to my networks in Keras, I added the kernel regularizer argument to the hidden layer. I am not sure whether it should also be added to the output layer. If my understanding is correct, the kernel regularizer should be applied to all layers, as the regularization is applied to all weights, but in my research, it seemed like common practice to only add it to the hidden layer. Either way, the MNIST plots more closely resembled the plots from the [1] paper without weight decay altogether. I also believe a kernel regularizer argument of 2.5×10^{-4} for Keras is equal to a weight decay parameter of 5×10^{-4} for PyTorch.

The mean reduction of MSE loss for PyTorch is used by [1]. As far as I can tell, the Keras MSE class defaults to average as well, but again this may be a source of inconsistency if this is not the case.

Most of the experiments were largely robust to small degrees of randomness. For example, changing the random seed of the random sampling algorithm did not significantly change the shape or amplitude of the curves. However, this was not true for the polynomial regression experiment. The results of this experiment were observably less consistent than the other experiments.

4 Closing Notes and Future Work

In this thesis, I have synthesized pieces of the current discussion on the generalization phenomenon of overparameterized neural networks. I investigated the role that the bias variance decomposition has in this phenomenon through experimentation. I probed and verified parts of other works that investigated this phenomenon as well. Ultimately, I found that the claims from [1] regarding their MNIST experiment were mostly robust. I also verified that the trends in the bias, variance, and test loss curves could be observed for natural language processing tasks, as well as computer vision tasks. I highlighted that different trends could be observed when injecting label noise into the experiments. I've shown that the classical bias variance tradeoff needs to be thoroughly reworked to be cohesive with the generalization trends of overparameterized neural networks.

At each iteration of experiments, I found that there was a nearly infinite list of microscopic tweaks and adjustments to be made that would better my work. This thesis represents the convergence of those lists on a local maximum. There are still infinitely many more alterations that could be made and other things to do that would expand and improve on this work, so that it would converge on a global maximum.

The primary addition I would have really liked to add to this work is measuring the decomposed variance on the experiments using the decomposition from [19]. Decomposing the variance this way would allow for deeper analysis into the role that different sources of randomness play in the generalization of the network. While [21] go about this analysis theoretically, I think it would be interesting to verify their claims empirically on all of my experiments.

I would also conduct the CIFAR10 experiments using a ResNet architecture instead of a DNN, as was done in [1]. Using a more complex network architecture on a data set that is as complex as CIFAR10 would be a more realistic experimental setting.

It seems as though the focus of experiments in the literature is often on computer vision data sets. I would conduct more experiments on NLP data sets, like the 20-NewsGroups data set used in [2] to verify the phenomena for different types of data. I would also try a different representation of the text than a sparse Bag of Words, and maybe try using a set of word embeddings, as was used in [2]. It would also be interesting to investigate why SGD becomes unstable for this experiment at the larger network widths.

To expand on my polynomial regression experiments, I would try to design a more consistent experiment framework, so that more reliable conclusions could be drawn. I am not sure exactly why the resulting curves of this experiment would be very different looking upon each run of the experiment, but some things to try would maybe be changing the number of trials to average the bias and variance over, increasing the number of training epochs, and tuning the optimizer. To further probe this experiment, I would also compare the results when using other numbers of data points and when changing the degree of the polynomial.

It would also be interesting to probe the idea of multiple descent that arose in [21] and [20]. I would have liked to design and include an experiment that would explicitly manifest these multiple descents.

Lastly, it would also be interesting to conduct similar experiments to those in this thesis while varying the depths of the networks. Given the previous discussion on the key role that deep neural networks play in the generalization puzzle, it would be important to compare the results of these experiments on shallow and deep neural networks. Both [1] and [3] conducted experiments where they varied the depth of the networks. [1] did this by using different ResNet architectures of varying depths, but it would certainly be interesting to see these experiments on a DNN.

5 References

[1] Z. Yang, Y. Yu, C. You, J. Steinhardt, and Y. Ma, "Rethinking bias-variance trade-off for generalization of neural networks," 2020.

[2] M. Belkin, D. Hsu, S. Ma, and S. Mandal, "Reconciling modern machine learning practice and the bias-variance trade-off," 2019.

[3] B. Neal, S. Mittal, A. Baratin, V. Tantia, M. Scicluna, S. Lacoste-Julien, and I. Mitliagkas, "A modern take on the bias-variance tradeoff in neural networks," 2018.

[4] B. Hanin, "ORF 350: Analysis of Big Data, Lecture 1," 2021.

[5] R. Quiza and J. Davim, *Computational Methods and Optimization*, pp. 177–208. 01 2011.

[6] S. Fortmann-Roe, "Understanding the Bias-Variance Tradeoff." http://scott.fortmann-roe.com/docs/BiasVariance.html. Online; accessed 3 April 2021.

[7] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," 2017.

[8] D. Zou, Y. Cao, D. Zhou, and Q. Gu, "Stochastic gradient descent optimizes over-parameterized deep relu networks," 2018.

[9] C. Bishop, *Pattern Recognition and Machine Learning*, ch. 3, pp. 147–152. Springer Science+Business Media, 2006.

[10] K. Weinberger, "Lecture 12: Bias variance tradeoff." https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote12.html, 2018. Online; accessed 26 October 2020.

[11] R. Novak, Y. Bahri, D. A. Abolafia, J. Pennington, and J. Sohl-Dickstein, "Sensitivity and generalization in neural networks: an empirical study," 2018.

[12] S. Arora, S. S. Du, W. Hu, Z. Li, and R. Wang, "Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks," 2019.

[13] Y. Li and Y. Liang, "Learning overparameterized neural networks via stochastic gradient descent on structured data," 2019.

[14] S. S. Du, X. Zhai, B. Poczos, and A. Singh, "Gradient descent provably optimizes over-parameterized neural networks," 2019.

[15] Z. Allen-Zhu, Y. Li, and Z. Song, "A convergence theory for deep learning via over-parameterization," 2019.

[16] Y. Li and Y. Yuan, "Convergence analysis of two-layer neural networks with relu activation," 2017.

[17] S. Arora, N. Cohen, N. Golowich, and W. Hu, "A convergence analysis of gradient descent for deep linear neural networks," 2019.

[18] Z. Allen-Zhu, Y. Li, and Y. Liang, "Learning and generalization in overparameterized neural networks, going beyond two layers," 2020.

[19] B. Adlam and J. Pennington, "Understanding double descent requires a fine-grained bias-variance decomposition," 2020.

[20] L. Chen, Y. Min, M. Belkin, and A. Karbasi, "Multiple descent: Design your own generalization curve," 2020.

[21] B. Adlam and J. Pennington, "The neural tangent kernel in high dimensions: Triple descent and a multi-scale theory of generalization," 2020.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[23] "Princeton research computing, tiger." https://researchcomputing.princeton.edu/systems/tiger.

[24] 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*, vol. 12, pp. 2825–2830, 2011.

[25] S. Raschka, "Bias-variance decomposition." https://rasbt.github.io/mlxtend/user_guide/evaluate/bias_variance_decomp/. Online; accessed 2 April 2021.

[26] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[27] "IMDB movie review sentiment classification dataset." https://keras.io/api/datasets/imdb/. Online; accessed 3 April 2021.

[28] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research),"

A Code Appendix

There is overlapping code between experiments, but each experiment was submitted to tigergpu with its own Python script and they are all slightly different. I've included each of them in their entirety here, but I will eventually modularize and consolidate the code base, then upload it to GitHub, as I think it could be useful to others who are trying to conduct similar research and experiments.

A.1 SGDRegressor Polynomial Regression Experiment Code

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import SGDRegressor
from sklearn.model_selection import train_test_split
from mlxtend.evaluate import bias_variance_decomp

% matplotlib inline

def powers_x(x, p):
    xs = []
    for i in range(1, p+1):
      xs.append(x**i)
    return np.array(xs).T

x = np.linspace(-1,1,20)

a = np.random.randint(-5,5)
b = np.random.randint(-5,5)
h = np.random.randint(-5,5)
k = np.random.randint(-5,5)
c = np.random.randint(-5,5)
d = np.random.randint(-5,5)
degree_5_function = lambda x: c + a*x + b*x**2 + h*x**3 + k*x**4 + d*x**5

epsilon = np.random.normal(0,1/2, len(x))
y = degree_5_function(x) + epsilon
```

```
plt.plot(x, y, marker='o')
plt.plot(x, degree_5_function(x))
plt.title("Randomly generated noisy data")

bias = []
variance = []
loss = []
random_state=np.random.randint(1000)

complexity = [1,2,4,6,8,10,12,14,16,18,20]

for i in complexity:
    X_train, X_test, y_train, y_test =
                train_test_split(powers_x(x, i), y,
                        test_size=0.5,shuffle=True, random_state=110)

    model = SGDRegressor(max_iter=10000, learning_rate='constant',
                            eta0=1e-3, alpha = 0.0001)

    avg_expected_loss, avg_bias, avg_var =
                bias_variance_decomp(model, X_train, y_train,
                        X_test, y_test, loss = 'mse', num_rounds=500)

    bias.append(avg_bias)
    variance.append(avg_var)
    loss.append(avg_expected_loss)

plt.plot(complexity, bias, label = "bias")
plt.plot(complexity, variance, label = "variance")
plt.plot(complexity, loss, label = "test loss")
plt.legend()
plt.yscale('log')
plt.xlabel('degree')
plt.xticks(complexity)
plt.show()

plt.plot(complexity, loss)
plt.yscale('log')
plt.xlabel('degree')
plt.ylabel('test loss')
plt.xticks(complexity)
plt.show()
```

A.2 Reproducing MNIST Experiment from [1] Code

```
import keras
import keras.backend as K
import numpy as np
import tensorflow as tf
import math

from keras.datasets import mnist
from keras.utils import to_categorical
from sklearn.metrics import accuracy_score
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras.callbacks import LearningRateScheduler
from keras.regularizers import l2


def _draw_bootstrap_sample(rng, X, y, num_splits):
    sample_indices = np.arange(X.shape[0])
    bootstrap_indices = rng.choice(sample_indices,
                                   size=10000,
                                   replace=False)
    picked = bootstrap_indices

    X_list = [X[bootstrap_indices]]
    y_list = [y[bootstrap_indices]]

    for i in range(num_splits-1):
        bootstrap_indices = rng.choice([i for i in sample_indices
                                        if i not in picked],
                                        size = 10000, replace=False)
        picked = np.concatenate((picked, bootstrap_indices))
        X_list.append(X[bootstrap_indices])
        y_list.append(y[bootstrap_indices])

    return X_list, y_list

def step_decay(epoch):
        initial_lrate = 0.1
        drop = 0.1
        epochs_drop = 100.0
        lrate = initial_lrate * math.pow(drop, math.floor(epoch/epochs_drop))
        return lrate

def bias_variance_decomp(w, X_train, y_train, X_test, y_test,
                         num_rounds=5, num_splits=6, random_seed=None,
                         **fit_params):

    rng = np.random.RandomState(random_seed)

    dtype = np.float

    var = []
    losses = []
    biases = []

    for i in range(num_rounds):
        all_pred = np.zeros((num_splits, y_test.shape[0], 10), dtype=dtype)

        X, y = _draw_bootstrap_sample(rng, X_train, y_train, num_splits)

        for j in range(num_splits):

            model = Sequential()
            model.add(Flatten(input_shape=(28, 28))),
            model.add(Dense(w,activation='relu',kernel_initializer='he_uniform',
                            use_bias=False))
            model.add(Dense(10,activation = 'softmax',
                            kernel_initializer='he_uniform',
                            use_bias=False))

            opt = keras.optimizers.SGD(learning_rate=0.0, momentum=0.9)
            model.compile(loss='mse', optimizer=opt, metrics=['accuracy'])

            lrate = LearningRateScheduler(step_decay)
            callbacks_list = [lrate]

            model.fit(X[j], y[j], callbacks=callbacks_list,**fit_params)

            pred = model.predict(X_test)
            all_pred[j] = pred

            scores = model.evaluate(X_test, y_test, verbose=0)
            print("Accuracy: %.2f%%" % (scores[1]*100))
            K.clear_session()
        avg_expected_loss = np.mean([np.sum((all_pred[x,:,:] - y_test)**2)/
                                    y_test.shape[0] for x in range(num_splits)])

        main_predictions = np.mean(all_pred, axis=0)

        avg_var = np.sum((main_predictions - all_pred)**2) /
                    ((all_pred.shape[0]-1)*all_pred.shape[1])
        avg_bias = avg_expected_loss - avg_var

        var.append(avg_var)
        biases.append(avg_bias)
        losses.append(avg_expected_loss)

    model.summary()

    return np.mean(losses), np.mean(biases), np.mean(var)


#https://machinelearningmastery.com/
#how-to-develop-a-convolutional-neural-network-from-scratch-
#for-mnist-handwritten-digit-classification/

# load train and test dataset
def load_dataset():
        # load dataset
        (trainX, trainY), (testX, testY) = mnist.load_data()
        # reshape dataset to have a single channel
        # one hot encode target values
        trainY = to_categorical(trainY)
        testY = to_categorical(testY)
        return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
        # convert from integers to floats
        train_norm = train.astype('float32')
        test_norm = test.astype('float32')
        # normalize to range 0-1
        train_norm = train_norm / 255.0
        test_norm = test_norm / 255.0
        # return normalized images
        return train_norm, test_norm
```

```
X_train, y_train, X_test, y_test = load_dataset()
# prepare pixel data
X_train, X_test = prep_pixels(X_train, X_test)

widths = [1, 2, 3, 4, 5, 6, 7, 8, 9,10,20,30,40,50,60,70,80,90,
            100,200,300,400,500,1000,2000,5000]

loss = []
bias = []
variance = []

for w in widths:

    print("width: ", w)

    avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
        w, X_train, y_train, X_test, y_test,
        num_rounds=5,
        num_splits=6,
        random_seed=3282021,
        epochs=200,
        verbose=1,
        batch_size = 128)

    bias.append(avg_bias)
    variance.append(avg_var)
    loss.append(avg_expected_loss)
    print(bias)
    print(variance)
    print(loss)

print(widths)
print(bias)
print(variance)
print(loss)
```

A.3 IMDB Experiment Code

```
import keras
import keras.backend as K
import numpy as np
import math
```

```
from sklearn.metrics import accuracy_score
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras.callbacks import LearningRateScheduler
from keras.regularizers import l2

def _draw_bootstrap_sample(rng, X, y):
    sample_indices = np.arange(X.shape[0])
    bootstrap_indices = rng.choice(sample_indices,
                                   size=12500,
                                   replace=False)
    bootstrap_indices_2 = [i for i in sample_indices
                            if i not in bootstrap_indices]
    return [X[bootstrap_indices], X[bootstrap_indices_2]],
            [y[bootstrap_indices], y[bootstrap_indices_2]]


def step_decay(epoch):
        initial_lrate = 0.1
        drop = 0.1
        epochs_drop = 100.0
        lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
        return lrate

def bias_variance_decomp(w, X_train, y_train, X_test, y_test,
                         num_rounds=3, random_seed=None,
                         **fit_params):
    rng = np.random.RandomState(random_seed)

    dtype = np.float

    var = []
    losses = []
    biases = []

    for i in range(num_rounds):
        all_pred = np.zeros((2, y_test.shape[0]), dtype=dtype)

        X, y = _draw_bootstrap_sample(rng, X_train, y_train)

        for j in range(2):
            model = Sequential()
            model.add(Dense(w, activation='relu',kernel_initializer='he_uniform',
                            use_bias=False))
            model.add(Dense(1, activation='sigmoid',
                            kernel_initializer='he_uniform',use_bias=False))

            opt = keras.optimizers.SGD(learning_rate=0.0, momentum=0.9)
            model.compile(loss="mean_squared_error", optimizer=opt,
                            metrics=['accuracy'])

            lrate = LearningRateScheduler(step_decay)
            callbacks_list = [lrate]

            model.fit(X[j], y[j], callbacks=callbacks_list, **fit_params)

            pred = model.predict_on_batch(X_test).reshape(1, -1)
            all_pred[j] = pred

            scores = model.evaluate(X_test, y_test, verbose=1)
            print("Accuracy: %.2f%%" % (scores[1]*100))
            K.clear_session()

        avg_expected_loss = np.apply_along_axis(
            lambda x:
            ((x - y_test)**2).mean(),
            axis=1,
            arr=all_pred).mean()

        main_predictions = np.mean(all_pred, axis=0)

        avg_var = np.sum((main_predictions - all_pred)**2) /
                        ((all_pred.shape[0]-1)*all_pred.shape[1])
        avg_bias = avg_expected_loss-avg_var

        var.append(avg_var)
        biases.append(avg_bias)
        losses.append(avg_expected_loss)

    return np.mean(losses), np.mean(biases), np.mean(var)

dim = 1501
#https://builtin.com/data-science/how-build-neural-network-keras
(training_data, training_targets), (testing_data, testing_targets) =
                                imdb.load_data(num_words=dim, index_from=1)
```

```python
def vectorize(sequences, dimension = dim):
    results = np.zeros((len(sequences), dimension-1))
    for i, sequence in enumerate(sequences):
        sequence = np.array(sequence)-1
        results[i, sequence] = 1
    return results

X_test = vectorize(testing_data)
X_train = vectorize(training_data)

X_test = vectorize(testing_data)
X_train = vectorize(training_data)

y_train = np.array(training_targets).astype("float32")
y_test = np.array(testing_targets).astype("float32")

widths = [1, 2, 3, 4, 5,10,20,30,40,50,100,200,300,400,500,
            1000,2000,3000,4000,5000,10000,20000,30000,40000,50000]

loss = []
bias = []
variance = []

for w in widths:
    print(w)
    avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
        w, X_train, y_train, X_test, y_test,
        num_rounds=3,
        random_seed=3282021,
        epochs=200,
        verbose=1,
        batch_size = 128,
        )
    bias.append(avg_bias)
    variance.append(avg_var)
    loss.append(avg_expected_loss)
    print(bias)
    print(variance)
    print(loss)

print(widths)
print(bias)
print(variance)
print(loss)
```

A.4 CIFAR10 Noise Experiment Code

```
import keras
import keras.backend as K
import numpy as np
import tensorflow as tf
import math
import sys

from keras.datasets import cifar10
from keras.utils import to_categorical
from sklearn.metrics import accuracy_score
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras.callbacks import LearningRateScheduler
from keras.regularizers import l2

def corrupt_labels(y_train, p):
    rng = np.random.RandomState(33198)
    sample_indices = np.arange(y_train.shape[0])
    bootstrap_indices = rng.choice(sample_indices,
                                   size=int(p*y_train.shape[0]),
                                   replace=False)
    for i in bootstrap_indices:
        j = np.random.randint(10)
        y_train[i] = np.zeros(y_train.shape[1])
        y_train[i][j] = 1

    return y_train

def _draw_bootstrap_sample(rng, X, y, num_splits):
    sample_indices = np.arange(X.shape[0])
    bootstrap_indices = rng.choice(sample_indices,
                                   size=10000,
                                   replace=False)
    picked = bootstrap_indices
    X_list = [X[bootstrap_indices]]
    y_list = [y[bootstrap_indices]]

    for i in range(num_splits-1):
        bootstrap_indices = rng.choice([i for i in sample_indices
                                        if i not in picked], size = 10000,
                                        replace=False)
        picked = np.concatenate((picked, bootstrap_indices))
        X_list.append(X[bootstrap_indices])
        y_list.append(y[bootstrap_indices])

    return X_list, y_list

def step_decay(epoch):
        initial_lrate = 0.1
        drop = 0.1
        epochs_drop = 100.0
        lrate = initial_lrate * math.pow(drop, math.floor(epoch/epochs_drop))
        return lrate

def bias_variance_decomp(w, X_train, y_train, X_test, y_test,
                         num_rounds=5, num_splits=6, random_seed=None,
                         **fit_params):

    rng = np.random.RandomState(random_seed)

    dtype = np.float

    var = []
    losses = []
    biases = []

    for i in range(num_rounds):
        all_pred = np.zeros((num_splits, y_test.shape[0], 10), dtype=dtype)

        X, y = _draw_bootstrap_sample(rng, X_train, y_train, num_splits)

        for j in range(num_splits):

            model = Sequential()
            model.add(Flatten(input_shape=(32, 32,3))),
            model.add(Dense(w, activation='relu',kernel_initializer='he_uniform',
                            use_bias=False))
            model.add(Dense(10, activation = 'softmax',
                            kernel_initializer='he_uniform',use_bias=False))
            opt = keras.optimizers.SGD(learning_rate=0.0, momentum=0.9)
            model.compile(loss='mse', optimizer=opt, metrics=['accuracy'])

            lrate = LearningRateScheduler(step_decay)
            callbacks_list = [lrate]

            model.fit(X[j], y[j], callbacks=callbacks_list,**fit_params)

            pred = model.predict(X_test)
            all_pred[j] = pred

            scores = model.evaluate(X_test, y_test, verbose=0)
            print("Accuracy: %.2f%%" % (scores[1]*100))
            K.clear_session()

        avg_expected_loss = np.mean([np.sum((all_pred[x,:,:] - y_test)**2)/
                            y_test.shape[0] for x in range(num_splits)])

        main_predictions = np.mean(all_pred, axis=0)

        avg_var = np.sum((main_predictions - all_pred)**2) /
                        ((all_pred.shape[0]-1)*all_pred.shape[1])
        avg_bias = avg_expected_loss - avg_var

        var.append(avg_var)
        biases.append(avg_bias)
        losses.append(avg_expected_loss)

    model.summary()

    return np.mean(losses), np.mean(biases), np.mean(var)

# the following two functions come from https://machinelearningmastery.com/
# how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/
# load train and test dataset
def load_dataset():
        # load dataset
        (trainX, trainY), (testX, testY) = cifar10.load_data()
        # reshape dataset to have a single channel
        # one hot encode target values
        trainY = to_categorical(trainY)
        testY = to_categorical(testY)
        return trainX, trainY, testX, testY
```

```python
# scale pixels
def prep_pixels(train, test):
        # convert from integers to floats
        train_norm = train.astype('float32')
        test_norm = test.astype('float32')
        # normalize to range 0-1
        train_norm = train_norm / 255.0
        test_norm = test_norm / 255.0
        # return normalized images
        return train_norm, test_norm


X_train, y_train, X_test, y_test = load_dataset()
        # prepare pixel data
X_train, X_test = prep_pixels(X_train, X_test)

percentages = [0,0.25,0.50,0.75,1.0]

losses = []
biases = []
variances = []

for p in percentages:

    y_train = corrupt_labels(y_train, p)

    widths = [1, 2, 3, 4, 5, 10,20,30,40,50,100,200,300,400,500,
                                    1000,2000,3000,4000,5000,10000]

    loss = []
    bias = []
    variance = []

    for w in widths:

        print("width: ", w)

        avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
            w, X_train, y_train, X_test, y_test,
            num_rounds=5,
            num_splits=5,
            random_seed=3282021,
            epochs=500,
            verbose=1,
            batch_size = 128)
        bias.append(avg_bias)
        variance.append(avg_var)
        loss.append(avg_expected_loss)

    biases.append(bias)
    variances.append(variance)
    losses.append(loss)

print(widths)
print(biases)
print(variances)
print(losses)
```

A.5 MNIST Noise Experiment Code

```
import keras
import keras.backend as K
import numpy as np
import tensorflow as tf
import math

from keras.datasets import mnist
from keras.utils import to_categorical
from sklearn.metrics import accuracy_score
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras.callbacks import LearningRateScheduler
from keras.regularizers import l2


def _draw_bootstrap_sample(rng, X, y, num_splits):
    sample_indices = np.arange(X.shape[0])
    bootstrap_indices = rng.choice(sample_indices,
                                   size=10000,
                                   replace=False)
    picked = bootstrap_indices

    X_list = [X[bootstrap_indices]]
    y_list = [y[bootstrap_indices]]

    for i in range(num_splits-1):
        bootstrap_indices = rng.choice([i for i in sample_indices
                                        if i not in picked], size = 10000,
                                        replace=False)
        picked = np.concatenate((picked, bootstrap_indices))
        X_list.append(X[bootstrap_indices])
        y_list.append(y[bootstrap_indices])

    return X_list, y_list

def step_decay(epoch):
        initial_lrate = 0.1
        drop = 0.1
        epochs_drop = 100.0
        lrate = initial_lrate * math.pow(drop, math.floor(epoch/epochs_drop))
        return lrate

def bias_variance_decomp(w, X_train, y_train, X_test, y_test,
                         num_rounds=5, num_splits=6, random_seed=None,
                         **fit_params):

    rng = np.random.RandomState(random_seed)

    dtype = np.float

    var = []
    losses = []
    biases = []

    for i in range(num_rounds):
        all_pred = np.zeros((num_splits, y_test.shape[0], 10), dtype=dtype)

        X, y = _draw_bootstrap_sample(rng, X_train, y_train, num_splits)

        for j in range(num_splits):

            model = Sequential()
            model.add(Flatten(input_shape=(28, 28))),
            model.add(Dense(w,activation='relu',kernel_initializer='he_uniform',
                            use_bias=False))
            model.add(Dense(10,activation = 'softmax',
                            kernel_initializer='he_uniform',use_bias=False))
```

```
                opt = keras.optimizers.SGD(learning_rate=0.0, momentum=0.9)
                model.compile(loss='mse', optimizer=opt, metrics=['accuracy'])

                lrate = LearningRateScheduler(step_decay)
                callbacks_list = [lrate]

                model.fit(X[j], y[j], callbacks=callbacks_list,**fit_params)

                pred = model.predict(X_test)
                all_pred[j] = pred

                scores = model.evaluate(X_test, y_test, verbose=0)
                print("Accuracy: %.2f%%" % (scores[1]*100))
                K.clear_session()

            avg_expected_loss = np.mean([np.sum((all_pred[x,:,:] - y_test)**2)/
                                        y_test.shape[0] for x in range(num_splits)])

            main_predictions = np.mean(all_pred, axis=0)

            avg_var = np.sum((main_predictions - all_pred)**2) /
                                ((all_pred.shape[0]-1)*all_pred.shape[1])
            avg_bias = avg_expected_loss - avg_var

            var.append(avg_var)
            biases.append(avg_bias)
            losses.append(avg_expected_loss)

        model.summary()

        return np.mean(losses), np.mean(biases), np.mean(var)

    #https://machinelearningmastery.com/
    #how-to-develop-a-convolutional-neural-network-from-scratch-
    #for-mnist-handwritten-digit-classification/

    # load train and test dataset
    def load_dataset():
            # load dataset
            (trainX, trainY), (testX, testY) = mnist.load_data()
            # reshape dataset to have a single channel
            # one hot encode target values
            trainY = to_categorical(trainY)
            testY = to_categorical(testY)
            return trainX, trainY, testX, testY
```

```
# scale pixels
def prep_pixels(train, test):
        # convert from integers to floats
        train_norm = train.astype('float32')
        test_norm = test.astype('float32')
        # normalize to range 0-1
        train_norm = train_norm / 255.0
        test_norm = test_norm / 255.0
        # return normalized images
        return train_norm, test_norm


X_train, y_train, X_test, y_test = load_dataset()
        # prepare pixel data
X_train, X_test = prep_pixels(X_train, X_test)

widths = [1, 2, 3, 4, 5, 6, 7, 8, 9,10,20,30,40,50,60,70,80,90,
                                100,200,300,400,500,1000,2000,5000]

loss = []
bias = []
variance = []

for w in widths:

    print("width: ", w)

    avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
        w, X_train, y_train, X_test, y_test,
        num_rounds=5,
        num_splits=6,
        random_seed=3282021,
        epochs=200,
        verbose=1,
        batch_size = 128)

    bias.append(avg_bias)
    variance.append(avg_var)
    loss.append(avg_expected_loss)
    print(bias)
    print(variance)
    print(loss)

print(widths)
print(bias)
print(variance)
print(loss)
```

A.6 Polynomial Regression via Neural Network Experiment Code

```
import keras
import keras.backend as K
import numpy as np
import tensorflow as tf
import math
import matplotlib.pyplot as plt

from keras.datasets import mnist
from keras.utils import to_categorical
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras.callbacks import LearningRateScheduler
from keras.regularizers import l2

def _draw_bootstrap_sample(rng, X, y):
    sample_indices = np.arange(X.shape[0])
    bootstrap_indices = rng.choice(sample_indices,
                                   size=int(X.shape[0] // 2),
                                   replace=False)
    bootstrap_indices_2 = [i for i in sample_indices if i not in bootstrap_indices
    return [X[bootstrap_indices], X[bootstrap_indices_2]], [y[bootstrap_indices],


def bias_variance_decomp(w,X_train, y_train, X_test, y_test,
                         num_rounds=3, random_seed=None,
                         **fit_params):
    rng = np.random.RandomState(random_seed)

    dtype = np.float
```

```
var = []
losses = []
biases = []

for i in range(num_rounds):
    all_pred = np.zeros((2, y_test.shape[0]), dtype=dtype)

    X, y = _draw_bootstrap_sample(rng, X_train, y_train)
    num_feats = 5

    for j in range(2):

        model = Sequential()
        model.add(Dense(w,kernel_initializer='he_uniform',
                        activation='relu',use_bias=False))
        model.add(Dense(w,kernel_initializer='he_uniform',
                        activation='relu',use_bias=False))
        model.add(Dense(1,kernel_initializer='he_uniform',use_bias=False))

        opt = keras.optimizers.SGD(learning_rate=0.01)
        model.compile(loss='mse', optimizer='adam')

        model.fit(X[j], y[j],**fit_params)

        pred = model.predict(X_test).reshape((len(y_test),))
        all_pred[j] = pred
        K.clear_session()

    avg_expected_loss = np.apply_along_axis(
        lambda x:
        ((x - y_test)**2).mean(),
        axis=1,
        arr=all_pred).mean()

    main_predictions = np.mean(all_pred, axis=0)

    avg_var = np.sum((main_predictions - all_pred)**2) /
                    ((all_pred.shape[0]-1)*all_pred.shape[1])
    avg_bias = avg_expected_loss - avg_var

    var.append(avg_var)
    biases.append(avg_bias)
    losses.append(avg_expected_loss)

return np.mean(losses), np.mean(biases), np.mean(var)
```

```
def powers_x(x, p):
    xs = []
    for i in range(0, p+1):
      xs.append(x**i)
    return np.array(xs).T

a,b,h,k,c,d = [-1.2850919410602808, 0.5691033984789772, 0.09893866923026028,
                    0.17269485304517362, -0.22446008702035375, 1.4059103650630964]
degree_5_function = lambda x: c + a*x + b*x**2 + h*x**3 + k*x**4 + d*x**5

num_datapoints = [90,120,150]

biases=[]
variances=[]
losses=[]

for j in num_datapoints:

    x = np.linspace(-1,1,j)

    epsilon = np.random.normal(0,1/8, len(x))
    y = degree_5_function(x) + epsilon

    bias = []
    variance = []
    loss = []

    X_train, X_test, y_train, y_test = train_test_split(powers_x(x,5), y,
                                                        test_size=1/3,
                                                        shuffle=True,
                                                        random_state = 3282021)
    widths = [1,2,3,4,5,10,20,30,40,50,100,200,300,400,500,
                                    1000,2000,3000,4000,5000]

    for i in widths:
        print(i)
        avg_expected_loss,
            avg_bias,
                avg_var = bias_variance_decomp(i, X_train, y_train, X_test,
                                                y_test,num_rounds=30,epochs=1000,
                                                batch_size=1,random_seed=3282021,
                                                verbose=0)
        bias.append(avg_bias)
        variance.append(avg_var)
```

```
            loss.append(avg_expected_loss)

        print(widths)
        print(bias)
        print(variance)
        print(loss)

        biases.append(bias)
        variances.append(variance)
        losses.append(loss)

print(widths)
print(biases)
print(variances)
print(losses)
```