Final Report: Deep Neural Networks

Ryan Jeong[1], Will Barton[2], and Maricela Ramirez[3]

[1]University of Pennsylvania, Department of Mathematics
[2]University of Massachusetts Amherst, Department of Mathematics
[3]University of Mary Hardin-Baylor, Department of Mathematics

July 2020

Abstract

Neural networks have been of rising interest in recent years with successful examples such as GPT-3 and AlphaGo. This report is based on a review of literature surrounding Universal Approximation Theory, linear regions and connections to expressivity and learnability. We explore the generation of linear regions in shallow and deep ReLU networks and visualize the hyperplane slicing of the input space. We show numerical results for SVD classification and various Feedforward Networks on the MNIST data set.

1 Introduction

Throughout the world, there are many problems that require the assistance of mathematical algorithms and data science. These problems are defined by functions that classify data, represent systems and more. The field of deep learning - a new name for neural networks - has been essential to finding solutions to these real world issues and continues to grow as a result of its success. Neural networks are inspired by the biology of the human brain; layers of "neurons" (also referred to as units) are interconnected to make some decision. The general architecture of neural networks revolves around the layers of units "communicating" and referring information to the output layer, which is the numerical presentation of the function that is being approximated.

1.1 Mathematical Description of Neural Networks

It has been shown that neural networks are able to approximate any continuous function and thus have prevalence in the real world. Recent analysis has been done on the functions used in networks which work to classify images, approximate solutions to PDE's and ODE's, and generate languages. When trained with a proper amount of data, a neural network should be able to "learn" the function related to the task it is

training on. The architecture of a classic feedforward network with L layers (depth) of n width is as follows:

$$\begin{aligned} z^{(l)} &= W^{(l)} a^{(l-1)} + b^{(l)}, \\ a^{(l)} &= \sigma(z^{(l)}), \\ u_{NN} &= W^{(\text{L-1})} a^{(L)} + b^{(\text{L-1})}, \end{aligned} \tag{1}$$

where $a^{(0)}$ is the input to the network, u_{NN} is the output of the network and $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function.

A traditional feedforward neural network begins with a set of inputs from a given piece of data; this data (depending on the original context) will be reshaped into a column vector where each element is a neuron in the input layer. In equation 1 the weights between each layer, W and biases, b are the trainable parameters of the network. Each layer is multiplied by the weights and a bias is added to compute a linear transformation, which is then put through an activation function. Some commonly used examples of activation functions are ReLU, sigmoid, and hyperbolic tangent [8], which are seen in Figure 1.

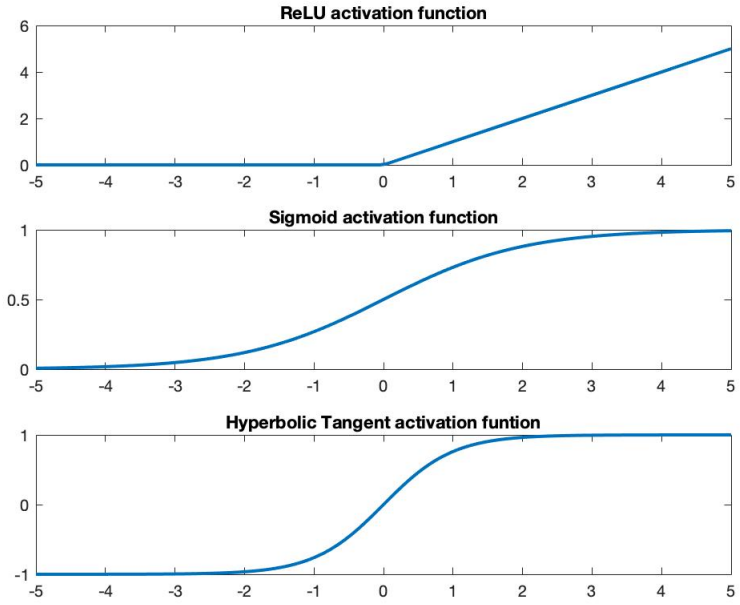


Figure 1: Commonly used activation functions

For the purposes of this paper, the ReLU activation function will be explored in more depth in later sections. It is worth noting that the functions in which neural networks attempt to model are often nonlinear, therefore, introducing nonlinearlity to the model is essential. This is done through he aforementioned activation functions such as ReLU and the sigmoid activation function. Overall, feedforward neural networks are composite functions that map from the input space, $\mathbb{R}^{in}$, to the output space, $\mathbb{R}^{out}$, and whose goal is to approximate a target function [7], for instance an underlying probability density or the solution to a differential equation.

There are many different architectures of neural networks, some architectures are more appropriate for certain tasks; within the realm of feedforward networks, the key architectures are deep and shallow networks. Although there is not a consensus on what constitutes shallow or deep, a common understanding is that a network with 1 hidden layer is considered shallow, whereas a network with a large number of hidden layers is considered deep. Generally speaking, when using a ReLU activation function, deeper networks are able to approximate functions more effectively than very shallow networks with only one or two hidden layers because the number of linear regions increases exponentially. This increase in linear regions can be thought of as an increase in the expressivity of the , or an improvement on its ability to approximate a desired unknown function. [5].

When working with neural networks, it must be trained (learn the function it is approximating), so there are two datasets - training and testing. In the case of classification problems, the training data will always have labels so that the performance of the network can be computed. The effectiveness of the network will be measured using what is referred to as a cost function. We denote the cost function by C_0, which is a function of u_{NN} and therefore is a function of all network parameters, as well. There are many different types of cost functions, but each aims to compare each node of the output layer to the desired outcome and then pool the difference between the two as a measure of how correct or incorrect the network's output was. A non-exhaustive list of cost functions includes mean squared error, cross-entropy loss, and negative log-likelihood [8]. The network's ability to "learn" and adjust the parameters to optimize the network is based on the cost function. While each network will have a unique goal, in general the training of the neural network is done to minimize the cost function so that it falls within a desired error tolerance (this tolerance will be highly context and application-dependent).

In order to minimize the cost function, the standard technique is to perform stochastic gradient descent using backpropagation. Backpropagation is the process in which the network can adjust its weights and biases to minimize the cost function; it begins from the output layer and works its way to the first layer weights and biases. We will find the minimum of cost function by using the negative gradient since it is known that the gradient is the direction of the maximum of a given function, so the negative gradient will take us to the minimum. These gradients will be computed with respect to the weights and biases and are very cheap to compute.

$$\frac{\partial C_0}{\partial w^{(l)}} = \frac{\partial C_0}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial w^{(l)}} \tag{2}$$

$$\frac{\partial C_0}{\partial b^{(l)}} = \frac{\partial C_0}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial b^{(l)}} \tag{3}$$

Computing the gradient and negating it provides the direction to change the weights and biases for each epoch - a full run through the training data- which can be costly to run on a full set of data, depending on the size of the data set. To combat this costly calculation, a standard technique is to employ stochastic gradient descent which is a continuous, iterated nudging of the network in the opposite direction of the gradient computed over random batches of the training data. Once training is done over the specified number of epochs, we can compute both the training and testing accuracies from the model, where accuracy refers to the total number of correctly classified examples over the total number of examples tested. It is expected that the accuracy for the

training data will be higher, but a significant difference between training and testing accuracy is an indication of overfitting - the network is overfit to the training data and no longer has capacity for variance in the test data- which can be a result of training for too many epochs.

1.2 Convolutional Neural Networks

While most of our report and experimental results have to do with standard feedforward networks (MLPs), there are other neural network architectures that are widely used and have proven to be effective at learning certain tasks. Convolutional Neural Networks in particular are highly relevant today, as they thrive at tasks involving image recognition. CNNs are the main architecture used in designing things like threat detection on security cameras, or the creation of self driving cars.

Rather than flattening out a set of data into a long, skinny input vector, CNNs attempt to preserve the structure of the image data they receive. The core of how these networks do this has to do with matrix convolution and then the application of a pooling algorithm to these convolved matrices. A convolution network has a set of filters in any given layer, which it will aim to train through backpropagation and SGD just like an MLP network. Each filter in a layer will be convolved with the input matrix, which means if there are n filters in a layer, there will be n convolved matrices as that layer's output. The convolution of each filter and input matrix will output a new matrix of values which is basically reflective of how well the filter aligned with the input matrix, or image, at various points.

A pooling algorithm is then applied to each of these matrices, often a max pooling algorithm. This pooling algorithm is a way to preserve the main features of the data while also making each of these convolved matrices smaller. As an example, if a max pooling algorithm with stride 2 is applied to a matrix, basically every 2x2 section of the matrix will have its greatest value taken and placed into a new matrix. This process of pooling is important because it reduces the number of computations the network must go through dramatically, without losing a significant amount of the information being learned.

Intuitively, one can think of the series of convolution layers as getting more and more precise and specified to the given task. The first layer may detect broad trends in an image, such as vertical or horizontal edges. However, one of the last convolution layers might be looking for things much more specific to the image data being trained on, such as a person's nose, or a dog's tail. After some number of convolution layers have been applied to the data, a CNN finishes its structure with either one or a series of fully connected layers like the ones seen in an MLP. It's after these fully connected layers that there will finally be an output from the network, which can then be compared to the intended result using a cost function. It's worth emphasizing that in this network, the weights and biases of the fully connected portion will be trained, but in the convolution layers it is the filters that are being trained. Their size (some nxn matrix) is fixed, but the values contained within each filter is a parameter that is adjusted through backpropagation with the goal of finding a minimum on the cost function.

2 Approximation Theory in Deep Learning

Cybenko: Approximation by Super-positions of a Sigmoidal Function:

Cybenko's expressivity results have served as the foundation for many further investigations in the field of neural networks and machine learning. He demonstrated that any continuous function could be approximated to an arbitrary accuracy given certain constraints of the network architecture that was learning the function, and his findings laid the groundwork for many further investigations and breakthroughs in the area.

This section will show that a one layer neural network can approximate any continuous function on the unit hypercube when a sigmoidal activation function is used. We let I_n denote the n-dimensional unit cube, $[0,1]^n$. The space of continuous functions on I_n is denoted by $C(I_n)$ and we use $\|f\|$ to denote the uniform norm of a function $f \in C(I_n)$. The results of [3] focuses on sums of the form :

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j) \tag{4}$$

The main result shows that any one-layer neural network represented by the sum of the previously mentioned form can approximate any continuous function on the unit hypercube as long as the sigmoidal function is continuous.
Definition: We say that σ is sigmoidal if

$$\sigma(t) \to \begin{cases} 1 & \text{as } t \to +\infty \\ 0 & \text{as } t \to -\infty \end{cases} \tag{5}$$

The first theorem is as follows:
Theorem 1: Let σ be any continuous sigmoidal function. Then finite sums of the form

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j)$$

are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(x)$, of the above form for which $|G(x) - f(x)| < \epsilon$ for all $x \in I_n$.
Theorem 2: Let σ be a continuous sigmoidal function. Let f be the decision function for any finite measurable partition of I_n. For any $\epsilon > 0$, there is a finite sum of the form

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j)$$

and a set $D \subset I_n$, so that $m(D) \geq 1 - \epsilon$ and $|G(x) - f(x)| < \epsilon$ for $x \in D$.

f is now a decision function for any finite measurable partition, which is precisely what a neural network is in mathematical terms(partitions data into classes). We now have a subset of I_n, D, where the measure of D can be defined by $m(D) \geq 1 - \epsilon$. An essential takeaway from this theorem is that any points within a resonable distance of

a descision region may be classified correctly. It is important to note that neural networks serve to partition data in order to classify it for a user, so the function f that has been referred to which represents the network in a mathematical way, is partitioning the data into a finite number of disjoint measurable sets. This theorem articulates the networks ability to make the measure of incorrectly classified points arbitrarily small.

3 Piecewise Linear Networks

3.1 Overview: Shift to Piecewise Linear Representations

Historically, sigmoidal activation functions have been chosen between affine transformations as the choice of nonlinearity (e.g. sigmoid, hyperbolic tangent). In the recent surge of deep learning, a notable shift has been the use of piecewise linear activation functions, specifically the rectified linear unit (ReLU), as a replacement of these sigmoidal functions. ReLUs have been used as the activation of choice in conjunction with various types of deep learning models, including both MLPs (feedforward networks) and CNNs.

Definition: A rectified linear unit (ReLU) is a real-valued piecewise linear function f defined on R, such that $f(x) = x$ if $x \geq 0$, and $f(x) = 0$ if $x < 0$:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0. \end{cases} \tag{6}$$

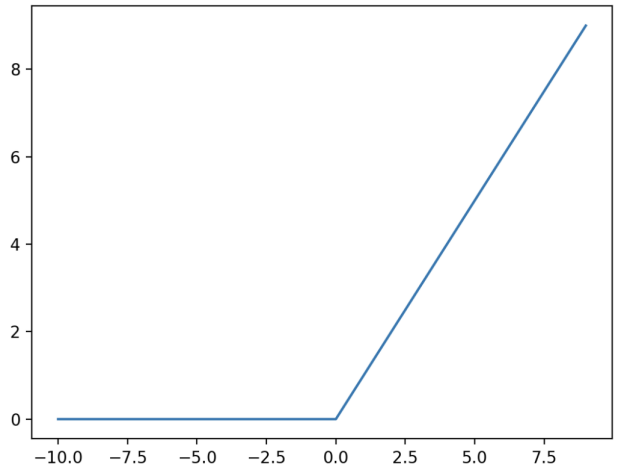


Figure 2: A ReLU activation function

There exist other types of piecewise activation functions. One is a variant of the ReLU, the "leaky ReLU", where the negative piecewise component of the ReLU (where $x < 0$) is not set to 0, but instead to some small constant multiple $cx, c > 0$.

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ cx & \text{if } x < 0 \end{cases} \tag{7}$$

Another variant is the maxout unit, in which one partitions the pre-activations into a number of groups, and computes the maximum within each group [4]. Explicitly, we denote the vector of pre-activations by $\mathbf{z}$, and let $g(z)$ be the group assignments of all elements in the vector z. Then, we have the activation $\mathbf{a}$ as

$$\mathbf{a}_i = max_{j \in \mathbb{G}_i} \mathbf{z}_j \tag{8}$$

where $\mathbb{G}_i$ represents all values assigned to the ith group. In practice, however, the ReLU is most frequently used as the activation of choice for piecewise linear networks.

One property of ReLU networks that is particularly convenient is their relation with the affine transformations between layers inherent in a neural network. If one thinks of a feedforward neural network as a composition of functions, observing that a composition of piecewise linear functions remains piecewise linear yields that a ReLU network (a feedforward net using strictly ReLU activation functions) will yield a piecewise linear function of $\mathbb{R}^n$, where n denotes the input dimension. In other words, the ReLU network partitions the input space into a number of disjoint regions, each defined by its individual linear function. Evidently, this is much more expressive than a typical linear network, as we can "engineer" different linear functions to suit different regions of input space appropriately.

It has also been shown that there exists a bijection between the class of piecewise linear functions from $\mathbb{R}^m \to \mathbb{R}^n$ and ReLU neural networks with input dimension m and output dimension n, up to isomorphism (distinct networks that compute the exact same function - one example includes permuting two hidden units within a layer and all associated weights, which does not change the function, but does indeed modify the appropriate weight matrices) [1]. This provides an exact description of the class of functions represented by ReLU neural networks of an arbitrary architecture.

This interpretation of ReLU networks suggests the notion of the number of linear regions as a heuristic metric for the complexity of a ReLU network.

3.2 On the number of linear regions in a ReLU network

The first thorough study of ReLU networks as a division of input space into piecewise linear regions was conducted focused on two cases - shallow networks ($d = 1$) and deep networks ($d > 1$) [6].

3.2.1 Shallow networks

Formally, a shallow network is a feedforward neural network with one hidden layer, resulting in an architecture with an input layer, a hidden layer, and an output layer. A key observation to make here is the equation governing the pre-activation to any particular activation node, specifically for the ith node in the hidden layer,

$$\mathbf{z}_i = \mathbf{W}_{i,:}\mathbf{x} \tag{9}$$

Now take the condition

$$\mathbf{z}_i = \mathbf{W}_{i,:}\mathbf{x} = 0 \tag{10}$$

Observe two properties of equation 10:

1. This equation governs a hyperplane, or an $(n-1)$-dimensional subspace in n-dimensional space.
2. This equation partitions the regions in input space for which the ith activation is "on or off", where "on" denotes that we are working in the positive piecewise component of the ReLU activation at that node.

Thus, we can think about two regions in input space: one in which the ith node is activated, and one in which it is not. What the above discussion shows is that the decision boundary between the regions is precisely governed by a hyperplane in input space.

The motivation for research in this area of deep learning is what happens when we consider all hyperplanes in the hidden layer collectively, specifically, how many distinct regions this will partition the input space into. At a first glance, we may believe that this quantity is upper-bounded by 2^{n_h}, since this denotes the number of ways in which we can assign distinct patterns of activations to the hidden layer. However, an argument involving hyperplane arrangements can provide a tighter upper bound [7]:

Theorem Suppose that a shallow network N has $n_{input} = m$, $n_{hidden} = n$. Then the maximum number of activation regions, denoted $r(n, m)$, is

$$r(n, m) = \sum_{i=0}^{m} \binom{n}{i}$$

Proof Consider $r(n-1, m)$, or $n-1$ hyperplanes through $\mathbb{R}^m$. Now add the nth hyperplane, which is itself an $m-1$ dimensional space. Observe that the present $n-1$ hyperplanes will thus slice the nth hyperplane into $r(n-1, m-1)$ regions, yielding the recurrence relation

$$r(n, m) = r(n-1, m) + r(n-1, m-1)$$

One can then prove this by induction on the sum of the arguments $n+m$. Trivially, we have that $r(1, 0) = r(0, 1) = 1$, satisfying the stated equation. Now,

$$\begin{aligned}
r(n, m) &= r(n-1, m) + r(n-1, m-1) \\
&= \sum_{i=0}^{m} \binom{n-1}{i} + \sum_{i=0}^{m-1} \binom{n-1}{i} \\
&= \binom{n-1}{0} + \sum_{i=0}^{m-1} (\binom{n-1}{i} + \binom{n-1}{i+1}) \\
&= \binom{n}{0} + \sum_{i=0}^{m-1} \binom{n}{i+1} \\
&= \sum_{i=0}^{m} \binom{n}{i} \square
\end{aligned}$$

3.2.2 Deeper networks

In a network of depth greater than 1, we should observe that the behavior of the activation equations at layers past the first layer. Take the second layer as an example:

$$\mathbf{z}_2 = \mathbf{W}_2 \mathbf{a}_1 = \mathbf{W}_2 \sigma_R(\mathbf{W}_1 \mathbf{x}), \tag{11}$$

where σ_R denotes ReLU.
Consider the equation for the ith component, or

$$(\mathbf{z}_2)_i = (\mathbf{W}_2)_{i,:}\mathbf{a}_1 = (\mathbf{W}_2)_{i,:}\sigma_R(\mathbf{W}_1\mathbf{x}) \tag{12}$$

The key observation here is that σ_R can be absorbed into one of the two weight matrices ($(\mathbf{W}_2)_{i,:}$ or $\mathbf{W}_1$). Assume an activation pattern of the first layer, $A_1(n), n \in 1, \ldots, n_{hidden}$ where $A(i) = 1$ if the ith node in the first hidden layer is activated, and take S to be the set of all values such that

$$i \in S \text{ if } A(n) = 1.$$

Then

$$(\mathbf{W}_2)_{i,:}\sigma_R(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2)'_{i,:}\mathbf{W}_1\mathbf{x}_i = \mathbf{w}'x \tag{13}$$

where

$$(\mathbf{W}_{2i,:})'_j = \begin{cases} \mathbf{W}_{2i,:j} & \text{if } A(j) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

$$\mathbf{w}' = (\mathbf{W}_{2i,:})'\mathbf{W}_1 \tag{15}$$

It is readily observable that different activation patterns, represented by different assignments to the function $A(n)$, will yield different values of $\mathbf{w}'$. Thus, the hyperplane equation

$$\mathbf{w}'x = 0 \tag{16}$$

will differ for each activation pattern determined by previous layers.

Hence, any activation node in the second layer will induce a unique hyperplane division for every activation pattern determined by the first layer's nodes. This causes the bending behavior of the second layer's "hyperplanes": at the first layer's hyperplanes. This idea evidently extends to activation nodes in all future layers, which will bend whenever they intersect a previous layer's (bent, if $l > 1$) hyperplanes. This is shown in Figure 3, [7].

Proceeding from the study of linear regions on a 1 layer network,is the number of regions a deep ReLU neural network can partition the input space into. In a constructive argument, we are provided the following lower bound on the maximum number of regions in a feedforward ReLU network with depth L [6]:

Theorem The *maximal* number of linear regions of the functions computed by a NN with n_0 input units and L hidden layers, with $n_i \geq n_0$ rectifiers at the ith layer, is lower bounded by

$$(\Pi_{i=1}^{L-1}\lfloor\frac{n_i}{n_0}\rfloor^{n_0})(\Sigma_{j=0}^{n_0}\binom{n_L}{j}) \tag{17}$$

Something notable about this theorem is the exponential dependence of this lower bound on the maximum possible number of regions on L, resulting from the product term. This is significantly better than what is possible for a shallow network, which has a combinatorial (and hence polynomial) upper bound on the number of possible regions. In particular, if the maximal width of a feedforward neural network is bounded, it naturally follows that this exponential lower bound can also be expressed as a function of the number of neurons in the network as well, a fact we will return to in Section 6.

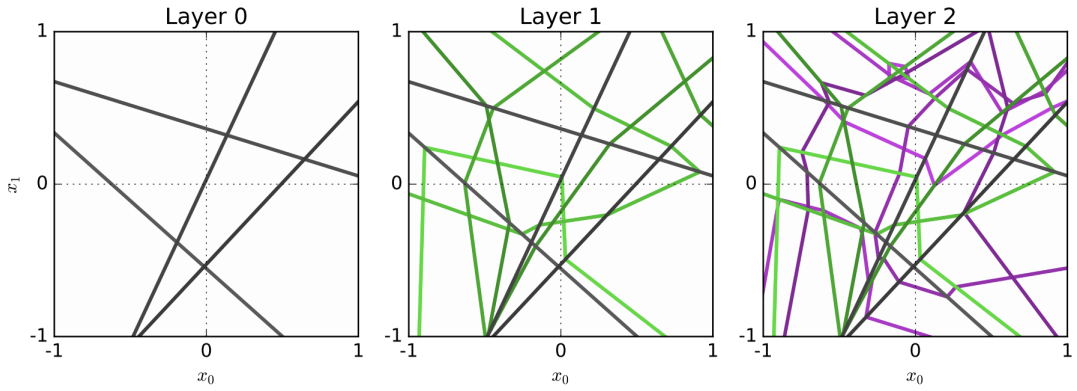


Figure 3: The partitioning of the input space by activation nodes (here assumed to be $\mathbb{R}_2$). The leftmost image includes only the hyperplanes determined by the first hidden layer, while the middle and right diagrams add on the second and third hidden layers, respectively. Observe that here, the new partitioning lines bend at previous layers' hyperplanes.

3.3 Visualizing the hyperplane slicing

One possible application, perhaps as a tool for further research, is to have some kind of visualization for slicing the input space with (bent) hyperplanes to understand the learning trajectory of piecewise linear networks (Figure 4).

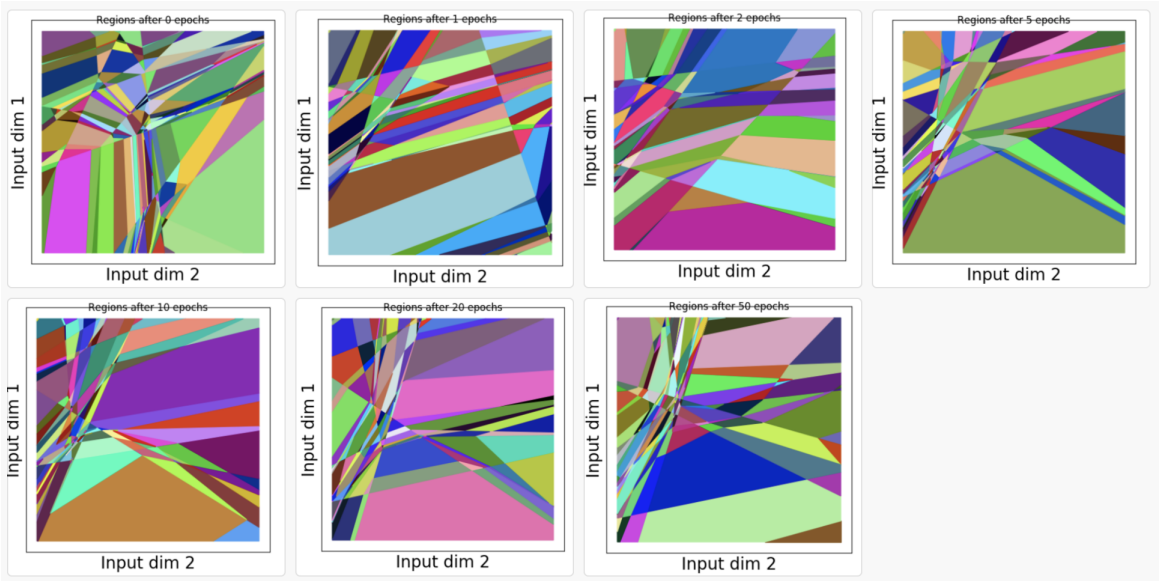


Figure 4: A visualization of the linear regions through a random 2D-subspace of the input space over training of the network, $\mathbb{R}^n$. Trained on MNIST; the network had five hidden layers, each with 8 hidden units.

One can make out a mild convergence of the linear region partitioning on the upper-left side of the image over epochs, but rather broad segments of the input space remain

over time.

4 Experiments

4.1 Feedforward Networks on MNIST Data

In real world applications, deep learning has been successful on a multitude of problems and it is useful to understand how the architecture of the algorithm being used will affect the expressively of the algorithm. As mentioned in Section 2.1, a one hidden layer feed forward network has been proven to be able to approximate any continuous function to some arbitrarily small error; this being said, it does not means that the one layer network will perform the best for all problems, as it has also been shown that depth can be beneficial for some funcitons. We used the MNIST data set of handwritten digits which is composed of 60,000 training images and 10,000 testing examples for simplicity and uniformity (it is said to be the "Hello World" of machine learning); the goal of a network trained on MNIST is to classify any handwritten digit. The following table shows a collection of different architectures of feed forward networks and their corresponding accuracies, some performing better than others.

Different Neural Network Architectures and their Accuracies					
Hidden Layers	Width	Trainable Parameters	EPOCHS	Training Accuracy	Testing Accuracy
1	20	15,910	3	95.5	94.9
1	64	50,890	3	97.9	96.8
1	64	50,890	8	99	97.3
1	128	101,770	8	99.3	97.5
1	128	101,770	12	99.7	97.8
4	100	109,810	8	98.9	97.6
8	32	32,842	8	97.2	96.2
8	64	80,010	8	98.1	97.3
8	128	217,354	8	98.2	97.1
10	25	257,354	8	96.4	95.4
10	50	62,710	10	97.3	96.7

Table 1: This table includes varying architectures, the number of trainable parameters and their accuracies during training and testing of the MNIST data set of handwritten digits. The learning rate was kept at a constant rate of 0.001 and a batch size of 10 with ReLU as the activation function. Each network is trained and tested with the entire MNIST training and testing sets.

Table 1 provides examples of the universal approximation theorem, along with empirical data which reveals overfitting in some instances. The universal approximation theorem states that a network with only one hidden layer can approximate any continuous function on some specific domain to some acceptable error, ϵ. As seen in the table, the highest accuracy achieved is on the neural network with 1 hidden layer that has 128 neurons and was ran through the entire data set 12 times (epochs); this is a

great example of universality. Do note that the difference between training accuracy and test accuracy for this particular architecture is just under 2%, whereas for many of the other architectures, the difference is closer to 1. It is expected that the training accuracy will be slightly larger than the test accuracy because the network is trained on the training data, if the difference in accuracy is significantly large this is a sign of overfitting. Although this particular network composition results in the highest accuracy it is also subject to the most apparent overfitting in this table of results. The difference in results of this specific architecture with a varying number of epochs is not significant enough to attribute the overfitting to training for too long rather than the structure itself. Despite having the lowest accuracy out of these experiments, the first architecture has the lowest amount of overfitting. The networks with 10 hidden layers did not perform well with the given number of epochs in comparison to the one-layer networks; the takeaway from this comparison is that for this particular task, a shallow network may be a better option.

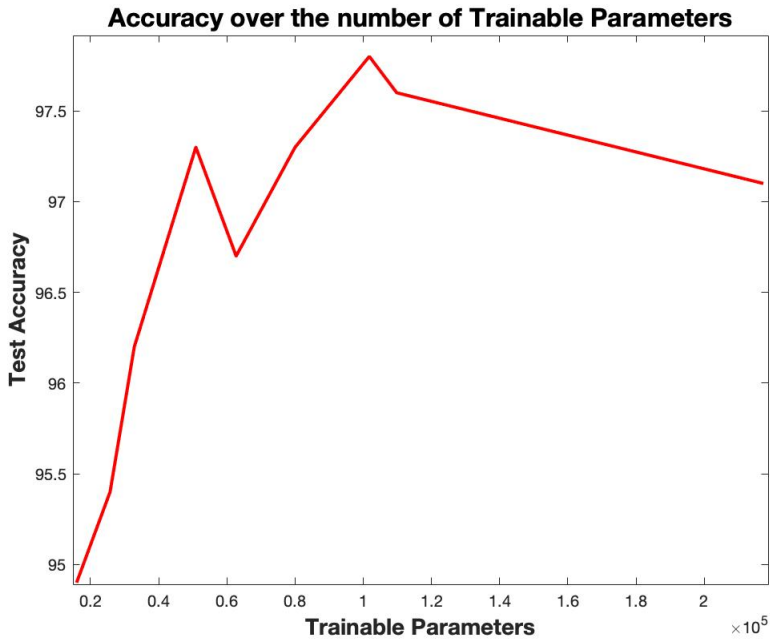


Figure 5: This plot shows accuracies from various architectures with the corresponding neural networks trainable parameters. This plot is representative of the lack of a correlation between the number of trainable parameters and test accuracy.

4.2 SVD Classifier

The SVD algorithm has many applications in data science and machine learning, for our purposes, we trained an SVD classifier on the MNIST data.

Definition: Singular Value Decomposition (SVD) Let A be an $m \times n$, and let r be the rank of A. Then there exists a matrix factorization called SVD of A with the form $A = U\Sigma V^T$, where

1. U is an $m \times m$ column-orthogonal matrix
2. Σ is a $m \times n$ diagonal matrix where the diagonal entries are called the singular values of A
3. V is an $n \times n$ column-orthogonal matrix

The MNIST data set is comprised of 60,000 training images and 10,000 test images of handwritten digits. In order to use SVD as a means to classify the data, we need 10 matrices, A_i, $i \in 0, 1, ..., 9$, where each column of a matrix A_i will be comprised of an image of a handwritten digit i. We then compute the SVD for each matrix A_i. The U_i matrix is comprised of the left singular vectors of the corresponding A_i matrix, which contains important information about each digit and is the only part of the SVD of A_i needed to complete this classification. A hyper-parameter for this algorithm is how many columns of U_i will be used when testing the data which will be explored later in this section. Once the SVD is computed for each A_i the "training" process of the algorithm is complete. The actual classification process is done through computing the residual between any given test image, b, and its projection onto the column space of each U_i matrix. The minimum computed residual will be used to choose the classification of the test image, for each U_i matrix, $\arg\min_{i \in 0,...,9} ||b - (U_i)(U_i^T)b||_2$.

Some experiments include plotting the accuracy of the SVD classifier based on the training to test ratio on a dataset of 10,000 images from MNIST, computing the accuracy of the SVD classifier based on the manipulation of data in three different ways: the entire MNIST data, 5,000 training images and 5,000 testing images (from their respective data sets, and 10,000 training images and 8,000 test images. All experiments we ran consisted of randomizing the columns of both the test and train data before the algorithm begins by randomly permuting the columns so they are not in the same order each time.

The results from Figure 1 were expected becasue as you decrease the amount of images being tested there is less room for error so the accuracy will increase. Another reason the accuracy may have increased is because as the percent of training images increases, the singular value decomposition is able to extract more information from each A matrix, therefore establishing a better trained model than with less training images.

Similar to the idea of overfitting data during training of a neural network, too high of a rank for the column space of U will result in the training algorithm to not be generalized for a wide range of data. Using 30 columns of our U_i matrices results in the best accuracy (0.01% better than the rank 20) when using the entire MNIST data set; after rank 30, the accuracy begins to decrease at a significant rate. When using 10,000 images to train the classifier and 8,000 to test it, Rank 40 performs the best, but not significantly better than the Rank 30 U matrix. This loss of accuracy may be due to the fact that as the column space of U increases, the projection onto any given test image does not leave room for much variation in the images. Overall, the results when using the entire MNIST data set and 10,000-8,000 train-test ration performed very similarly, no notable differences.

The test when using 5,000 training images and 5,000 test images resulted in a very different learning curve in comparison to the previous tests. The rank 20 U matrix performed the best, with a relatively significant difference to the surrounding ranks. It is also an interesting observation that the rank 100 U matrix performed better than

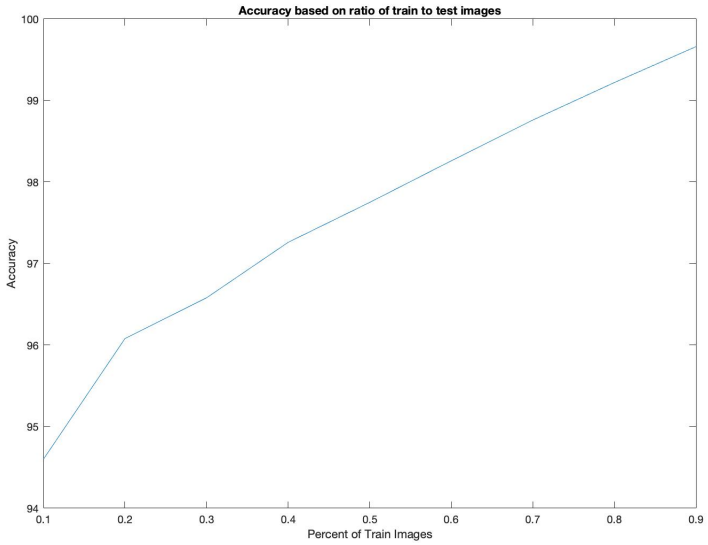
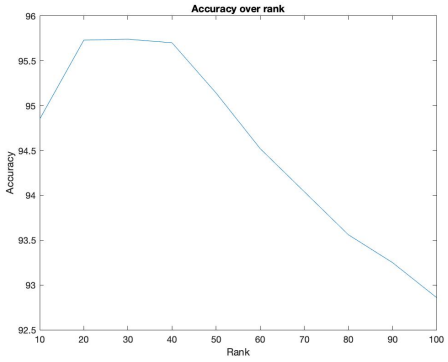
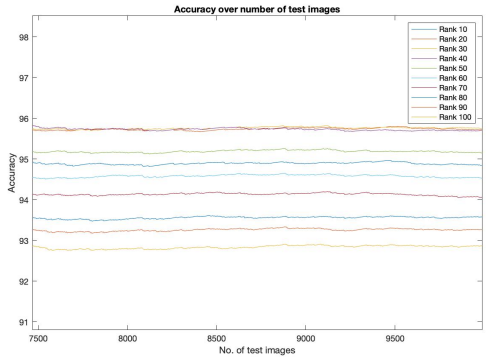


Figure 6: Using the 10,000 image test set from the MNIST data, we plotted a learning curve for this SVD classifier to show accuracy based on the ratio of training images to test images. Example: When 20% of the data is being used for training, 80% for testing, accuracy is approximately 96%.



(a) This plot shows the relationship between accuracy and what rank the matrix, U, is on the entire MNIST data set.

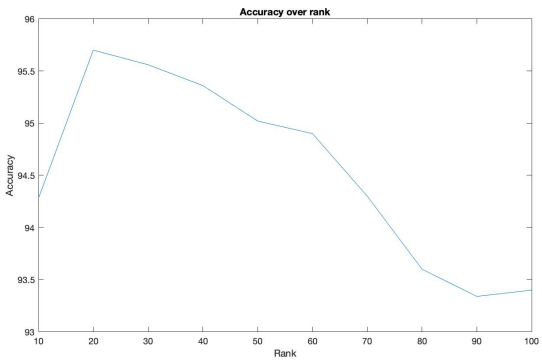


(b) Individual learning curves.

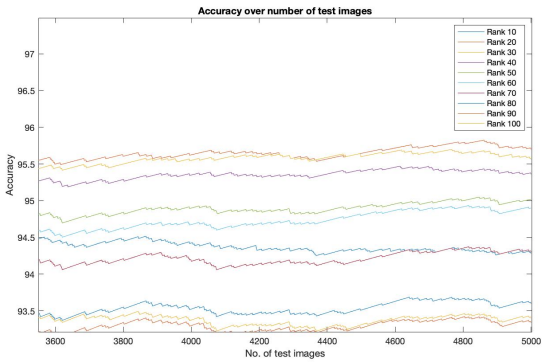
Figure 7

the rank 90, an new twist to the monotonic decrease that was seen the other results. The amount of training data to test data did not seem to affect the overall accuracy of each experiment- each of the tests highest accuracy is $\approx$ 95.6-95.8%.

While the SVD classifier did perform "well" for a trivial task such as MNIST, it only outperformed two feed-forward architectures. Of course, in real life, depending on the nature of the problem different algorithms may be more appropriate, but for this problem both performed relatively well.

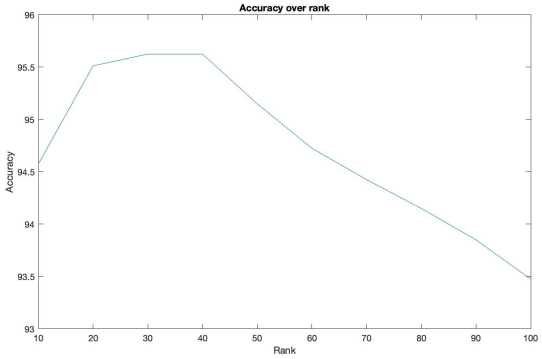


(a) This plot shows the relationship between accuracy and what rank the matrix, U, is on 5000 unique training and test images.

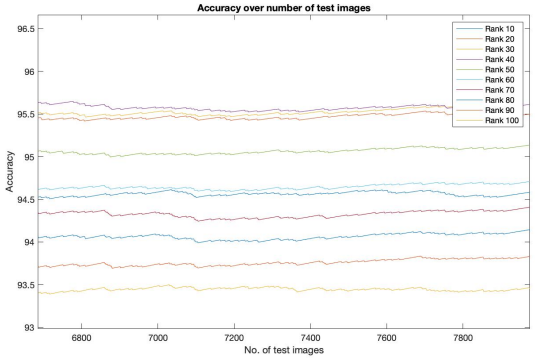


(b) Individual learning curves.

Figure 8



(a) This plot shows the relationship between accuracy and what rank the matrix, U, is on 10,000 training images and 8,000 test images.



(b) Individual learning curves.

Figure 9

5 Expressivity vs. Learnability

5.1 Motivation of Open Problem

There has been significant work done in using ideas from analysis and approximation theory to understand the class of functions that neural networks are capable of expressing, and the benefits of depth in extending this class of functions. Notably, shallow networks (and thus neural networks in general) are universal approximators of any continuous function, and deep neural networks (depth $d > 1$) can express certain classes of functions with "exponentially better guarantees" (such as an exponentially large number of linear regions for deep ReLU networks, which isn't possible for their shallow counterparts).

Two issues are particularly prevalent:

1. There are no known results that show what the weights must be set to approximate an arbitrary function. Hence, we can only speak of the existence of such networks, and not how to obtain them.
2. In practice, neural networks are trained with gradient-based optimization algorithms; the weights are not deliberately set. Hence, over optimization, it is possible that these algorithms fail to learn a large class of functions that are in theory expressible, or that the class of learnable functions, denoted F_{learn}, by a neural network of an arbitrary architecture is a (very humble) proper subset of what is expressible by this network, denoted $F_{express}$.

Here, we focus on issue (2) with respect to ReLU networks. Particularly, the literature has consistently shown a class of functions that can be learned by a deep ReLU network, but not by a shallow one with the same resources (number of neurons/parameters).

A primitive result suggests a divide in expressivity and learnability in deep nets [2]; they trained models for various tasks on a number of deep networks, then performed "imitation learning," in which a dataset was generated from the trained model and used to train a randomly initialized shallow network with comparable resources. It was also found that the shallow networks performed just as well, suggesting that deep nets failed to learn the class of expressive functions that they are theoretically capable of realizing.

This was further investigated in [5], who mathematically proved that at initialization, the number of linear regions that a random m-dimensional subspace intersects is $O(n^m)$, or polynomial in the input size, and empirically showed that this does not change over optimization.

Figure 10 shows this phenomenon for 2D subspaces.

5.2 Initial Experiments

The motivation behind these experiments is whether or not ReLU networks can be trained to represent functions in the "exponential regime", or having a number of regions exponential in the number of neurons

One such function that lies in the exponential regime is the "sawtooth function". This function, defined on $[0, 1]$, partitions $[0, 1]$ into 2^n pieces (for some value n), then oscillates between them the maximal number of times; as seen in Figure 11.
To develop the sawtooth function mathematically, consider the following $f(x)$, which we denote the mirror map [9]:

$$f(x) = \begin{cases} 2x & \text{when } 0 \leq x \leq \frac{1}{2} \\ 2(1-x) & \text{when } \frac{1}{2} \leq x \leq 1 \end{cases}$$

Observe that this function can be represented exactly as a two-layer feedforward ReLU neural network:

$$f(x) = \sigma_R(2\sigma_R(x) - 4\sigma_R(x - \tfrac{1}{2}))$$

Composing the mirror map with itself n times will yield a sawtooth function with 2^n affine pieces.

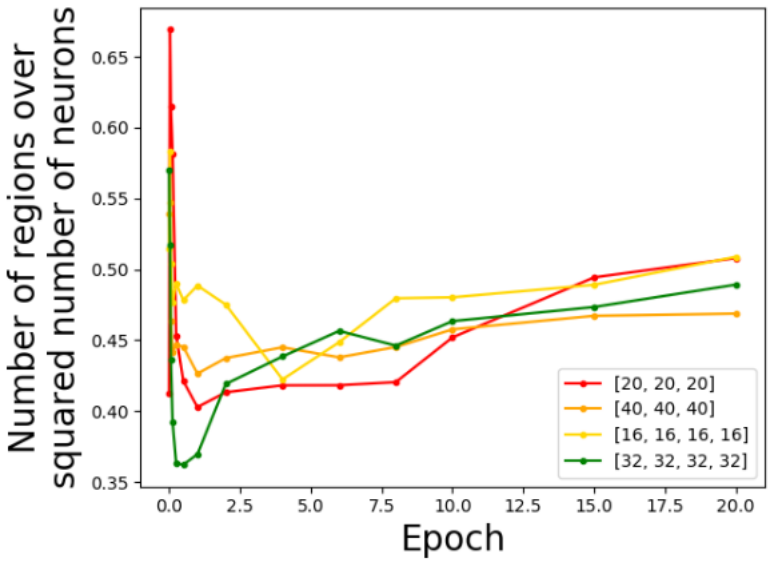


Figure 10: Number of linear regions intersected by a random 2D subspace in ReLU nets over squared number of neurons, for a variety of architectures. Observe that all tested architectures converge to roughly $\frac{n^2}{2}$, far below what is theoretically possible.

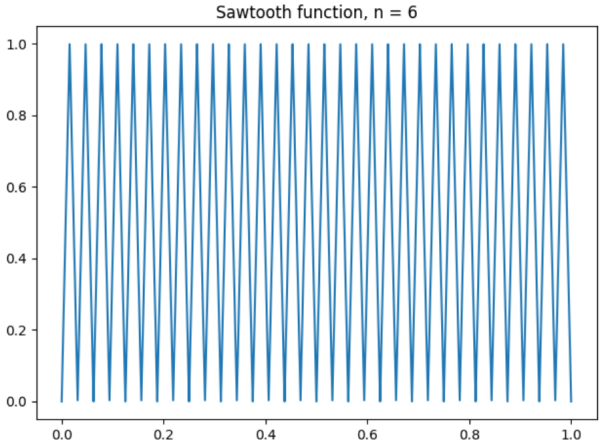


Figure 11: Sawtooth functions, $n = 6$

5.2.1 Multiplicative Weight Perturbations

Considering the sawtooth network as a representative model for the class of functions in the exponential regime, one natural set of experiments to try is to observe how robust the sawtooth function is to zero-mean multiplicative weight perturbation, or a perturbation to a weight w of the form $w(1+\epsilon)$. Loosely speaking, in practice, various forms of error exist (e.g. rounding error, batch of data); a model that is not robust to stochastic perturbation suggests that it will not be learnable in practical optimization schemes.

The following experiments, detailed in Figure 5, perturb different subsets of the

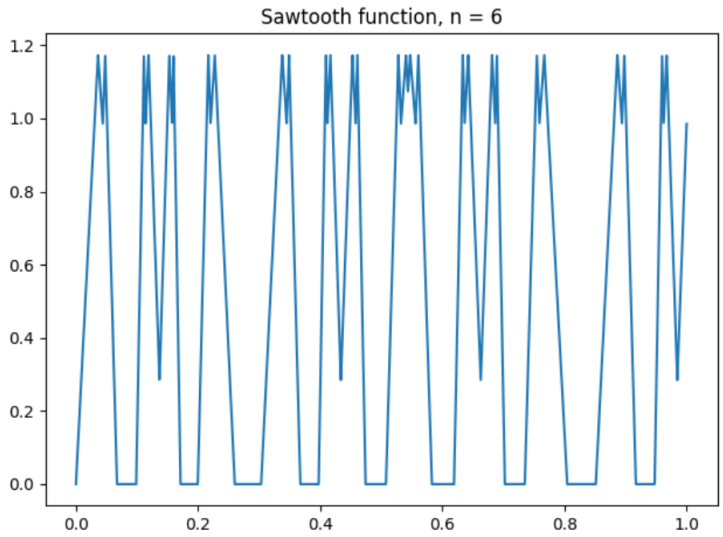


Figure 12: Sawtooth function with multiplicative Gaussian perturbation

parameters with zero-mean Gaussian noise; the variance of this distribution is varied, and 50 trials were performed for each particular value of the variance.

As might be expected intuitively, greater variance yields a smaller number of affine regions, with weights (specifically, early-layer weights, where errors can be propagated) being more sensitive than biases.

6 Conclusion

Throughout our research, we demonstrated through literature review and experiments that neural networks can be used to classify the MNIST data set. In our experiments results between a SVD classification algorithm and feedforward networks, the feedforward networks performed with higher success. We also demonstrated the overall significance of slicing the input space with hyperplanes to the expressive abilities of the network.

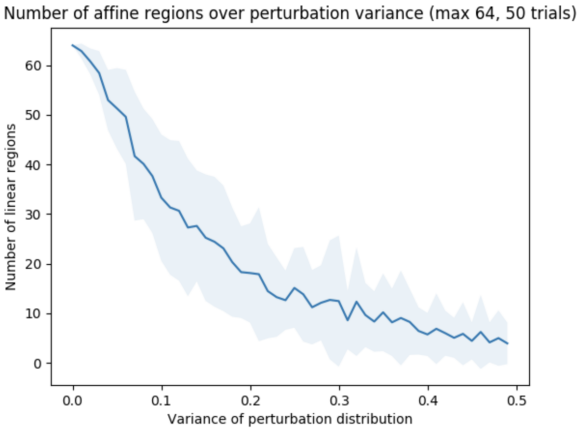
References

[1] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee, *Understanding deep neural networks with rectified linear units*, arXiv preprint arXiv:1611.01491, (2016).

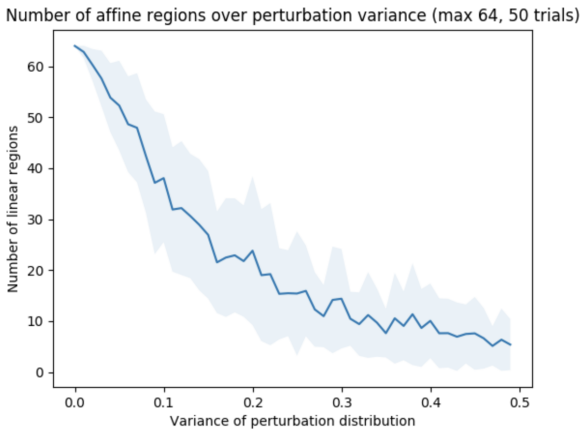
[2] J. Ba and R. Caruana, *Do deep nets really need to be deep?*, in Advances in neural information processing systems, 2014, pp. 2654–2662.

[3] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Mathematics of control, signals and systems, 2 (1989), pp. 303–314.

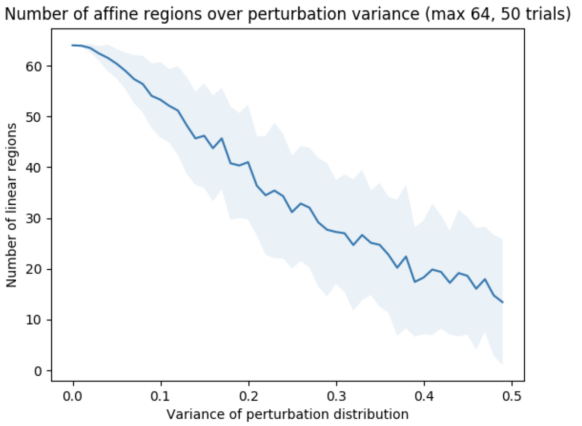
[4] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, *Maxout networks*, in International conference on machine learning, 2013, pp. 1319–1327.



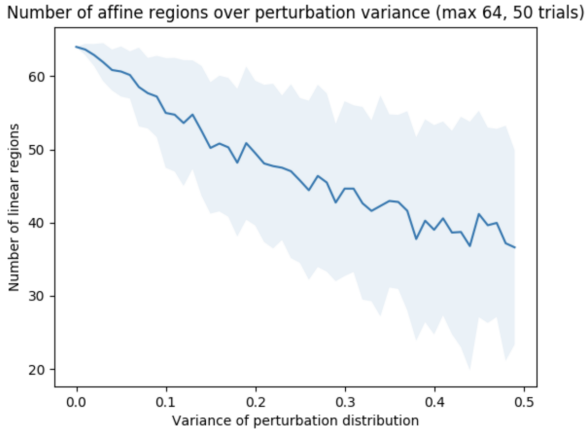
(a) All parameters perturbed



(b) All weights perturbed



(c) All biases perturbed



(d) Only first layer parameters perturbed

Figure 13: Perturbations of parameters and number of affine regions.

[5] B. Hanin and D. Rolnick, *Complexity of linear regions in deep networks*, arXiv preprint arXiv:1901.09021, (2019).

[6] G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio, *On the number of linear regions of deep neural networks*, in Advances in neural information processing systems, 2014, pp. 2924–2932.

[7] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein, *On the expressive power of deep neural networks*, in international conference on machine learning, 2017, pp. 2847–2854.

[8] G. Strang, *Learning from data*, in Linear Algebra and Learning from Data.

[9] M. Telgarsky, *Representation benefits of deep feedforward networks*, 2015.