An Introduction To and Applications of Neural Networks

Adam Oken

May, 2017

Abstract

Neural networks are powerful mathematical tools used for many purposes including data classification, self-driving cars, and stock market predictions. In this paper, we explore the theory and background of neural networks before progressing to different applications of feed-forward and auto-encoder neural networks. Starting with the biological neuron, we build up our understanding of how a single neuron applies to a neural network and the relationship between layers. After introducing feed-forward neural networks, we generate the error function and learn how we minimize it through gradient decent and backpropagation. Applying this, we provide examples of feed-forward neural networks in generating trend lines from data and simple classification problems. Moving to regular and sparse auto-encoders, we show how auto-encoders relate to the Singular Value Decomposition (SVD), as well as some knot theory. Finally, we will combine these examples of neural networks to discuss deep learning, as well as look at some example of training network and classifying data with these stacked layers. Deep learning is at the forefront of machine learning with applications in AI, voice recognition and other advanced fields.

1 Introduction to Neural Networks

In this section we will introduce neural networks by first discussing the biological model of a single neuron. We will then transfer that knowledge to a mathematical perspective of a single neuron, progressing further to a network of neurons. After learning what a neural network is, the architecture and applications will be briefly discussed.

1.1 Neurons

Neural networks were designed to model how a neuron interacts with surrounding neurons, as such we will start off by talking about some biology. The human body is made up of trillions of cells with a diverse range of functions. Concentrating our introduction to one of the important systems of the body, we will focus on cells in the nervous system. The nervous system consists of two main categories of cells: neurons and glial cells. Glial cells are non-neuronal cells that maintain homeostasis, form myelin, and participate in signal transduction. More importantly and the focus of this introduction, neurons are the fundamental cell of the brain. The brain consists of many neurons, each made up of dendrites, a cell body, and an axon. Figure 1 shows the structure of a typical neuron with the three domains. Dendrites are branched, tree like, projections of a neuron that propagate the electrochemical stimulation received from other neural cells and sends them to the cell body. The cell body contains two important domains: the cell nucleus and the axon hillock. The axon hillock is a specialized domain of the cell body where the axon begins and contains a large number of voltage-gated ion channels. These channels are often the sites for action potential initiation. More specifically, once the electrochemical signals are propagated to the cell body, they are summed in the axon hillock and once a triggering threshold is surpassed, an action potential propagates through the axon. Figure 2 shows a depiction of the triggering threshold and the voltage output of a action potential.

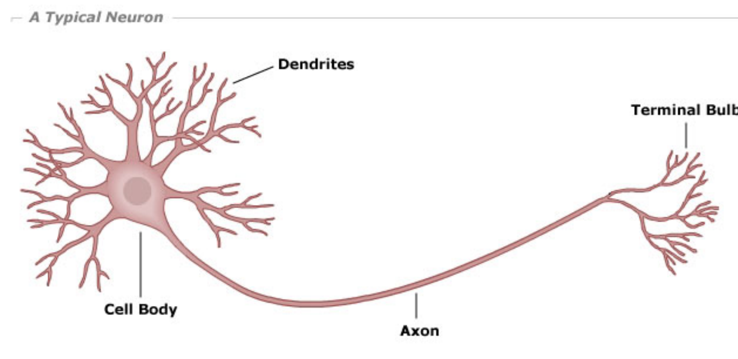


Figure 1: A typical neuron with the cell body, dendrite, axon, and terminal bulb

The final part of a neuron is the axon. The axon is the long projection of a neuron that transmits information to different neurons, muscles or glands. For example, sensory neurons transmit signals back to the cell body via an axon to trigger a sensation in the brain. Neurons are distinguishable from other cells in a few ways. They can communicate with other cells via synapses. A synapse is a structure that allows neurons to pass electrical or chemical signals to other neurons. Altogether, neurons are complicated cells that can communicate with other neurons via synapses as well as other parts of the body via electrochemical signals that are propagated from dendrites to the axons.

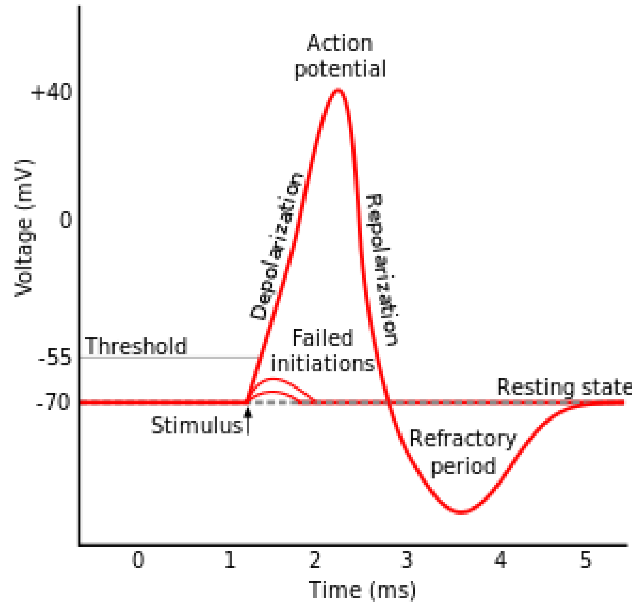


Figure 2: A graphical interpretation of action potentials and the threshold they need to attain in order to send a signal down the axon [11]

Now let's look more closely at an isolated single neuron via a mathematical perspective to understand how it can be modeled as inputs and outputs that run through a node. We will start by looking at the dendrites or what we will define as the input layer. Each dendrite propagates an electrochemical signal with different weights. Notationally, we define n dendrites in the input layer nodes as $x_1, x_2, ..., x_n$ and the corresponding n weights as $w_{11}, w_{12}, ..., w_{1n}$ where w_{ij} refers to the weight taking x_j to the node i. Figure 3 shows the structure of a single neuron with three input nodes x_1, x_2, x_3 and their corresponding weights w_1, w_2, w_3 (note that since there is only one node, the value for i was left out, but it would be w_{11}, w_{12}, w_{13}). Do note that the notation for the weights and nodes will become more complicated as more structure is added to the network. This notation will be addressed later, but for understanding how a single neuron functions we will make these simplifications. Continuing on, each dendrite and its corresponding weight are connected to a cell body or as we will define a node through an edge. Edges are connections between two nodes such that when a signal passes through an edge, it is multiplied by the corresponding weight. All the weighted signals are then summed along with a bias term b for each single node. We can think of this bias term as the resting state of the neuron. Additionally, since multiple signals come to a given node, we will assume that they arrive at the same time. Once summed, we apply an activation function to transform the input into a final output value. The activation function is usually a nonlinear transfer function which will be described more later on.

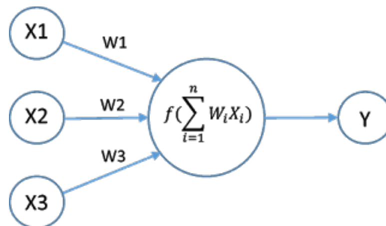


Figure 3: A mathematical depiction of a signal neuron [8]. There are three input values or nodes x_1, x_2, x_3 with corresponding weights w_{11}, w_{12}, w_{13}. They are multiplied, summed, and then an activation function is applied to them so there is a single output.

Describing a single neuron mathematically, we see that it is a function from $\mathbb{R}^n$ to $\mathbb{R}$. Using the notation we just learned above, we start with the input $\mathbf{x}$ and multiply the inputs by their corresponding weights. So,

$$\mathbf{x} \to (w_{11}, w_{12}, \ldots, w_{1n}) \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ \vdots \\ x_{1n} \end{bmatrix} = w_{11}x_1 + w_{12}x_2 + \ldots + w_{1n}x_n = \mathbf{w} \cdot \mathbf{x} + b$$

We then apply an activation function to the node, call it σ where $\sigma(\mathbf{w} \cdot \mathbf{x} + b)$ is the output of a single neuron. The transfer function, otherwise known as an activation function, corresponds to the activation state of the node [7]. As was mentioned earlier, a voltage potential must build up enough signal in the cell body to send a signal down the axon. The transfer or activation function is what is simulating this biological effect in a neural network with respect to the node and output signal. Altogether, this is how we model a single neuron. A neural network is a collection of single neurons. Thus, by understanding how a single neuron works, we can obtain a better grasp of how a neural network would function.

1.2 Network Architecture

A neural network consists of a series of layers. The first layer is called the input layer and if the input $\mathbf{x}_i \in \mathbb{R}^n$ then the layer has n nodes. In Figure 4, $\mathbf{x}_i \in \mathbb{R}^3$ so there are three nodes in the input layer. The final layer is called the output layer and if $\mathbf{y}_j \in \mathbb{R}^m$ then the layer has m nodes. In Figure 3, $\mathbf{y}_j \in \mathbb{R}^1$ so there is one node in the output layer. In between the two aforementioned layers are some number of the hidden layers each with some number of k nodes. We define the architecture of the neural network by the number of nodes in each layer. For example, Figure 4 is a depiction of a neural network with 3 nodes in the input layer, 4 nodes in the first hidden layer, 4 nodes in the second hidden layer, and 1 node in the output layer. This network would be described as a 3-4-4-1 neural network.

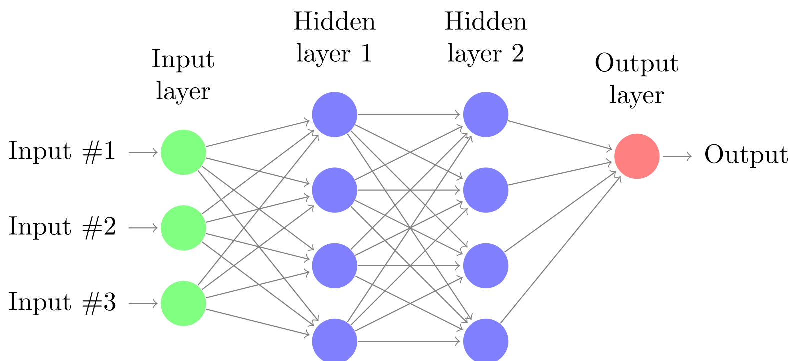


Figure 4: A 3-4-4-1 neural network.

1.3 Application and Purpose of Training Neural Networks

A neural network is a software simulation that recognizes patterns in data sets [11]. Once you train a neural net, that is give the simulation enough data to recognize the patterns, it can predict outputs in future data. We can think of training a neural network as the creation of a function between a given domain and range. Once trained, any data within the domain we provide can be mapped to the functions range. A simple example of a neural network in action is the classification of data. We are given a data set containing six characteristics of 200 wines (the input would be a 6 by 200 matrix) as well as knowing the properties of 5 different types of wine. We can train the neural network on 50 different wines and then the generated function will be able to classify the other 150 wines into the five types of wine (the output would be a 5 by 200 matrix). Neural networks can be a very powerful tool to analyze and predict data.

One important feature we must mention about training a neural network is how a network learns. There are two types of learning: supervised learning and unsupervised learning [1]. A learning algorithm of a neural network is said to be supervised learning when the output or target values are known [6]. This was the case in the aforementioned example about the classification of wine. We knew the 5 types of wine that the 200 bottles were being classified into. On the other side, unsupervised learning doesn't "know" the outputs or target values. The learning process has to find patterns within the data in order to output values. Unsupervised learning is used in many complex systems, including data processing, modeling, and classification. The goals for either type of learning are easier said

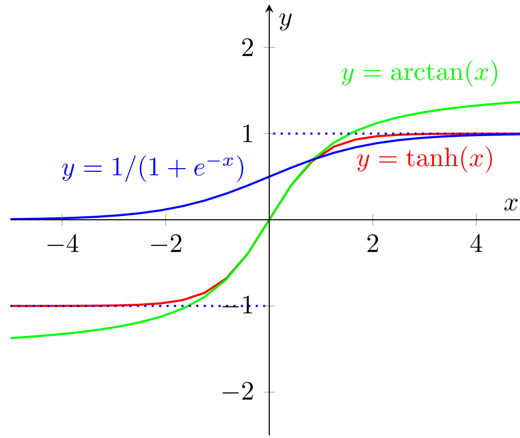


Figure 5: This graph shows the three transfer functions that are discussed above. In blue is the function $1/(1+e^{-x})$ which is bounded above at one and below at zero. In red is the hyperbolic tangent function which is bounded above at one and below at negative one. In green is the inverse tangent function that is bounded above by $\pi/2$ and below by $-\pi/2$.

then done, we find the best weights and biases for a given neural network that produce the most accurate function approximation.

2 Feed-Forward Neural Networks

A feed-forward neural network creates a mapping from $\mathbb{R}^n \to \mathbb{R}^m$ that is considered supervised learning. For feed-forward neural networks, we are given the target values for the given problem. The mapping consists of an initial signal (denoted $\mathbf{x}$), prestates (denoted P_j), transfer function ($\sigma(r)$ also called a sigmoidal function because of its shape with r equal to to some prestate P_j), and states (denoted S_j). To compute the final output of the neural network, we need to calculate each of these states for each layer. Since each layer is dependent on the previous, we need to start at the input layer and work towards the output layer. Before we show how to complete the forward pass of the network, that is compute the output, it is important to know the relationship between the different states and layers. The generalized relationship between the states and two adjacent layers of a network is:

$$S_{i-1} \to P_i = W_i S_{i-1} + \mathbf{b}_i \to S_i = \sigma(P_i)$$

where $\mathbf{b}_i$ corresponds to the vector of biases for layer i, W_i is the matrix corresponding to all the weights for layer i, and $S_0 = \mathbf{x}$, the input layer. We will use the relationships that we just stated to compute the forward pass of the feed-forward neural network. Once the forward pass is completed, we can compute the error between the given target values and our output. From there, we can decrease the error by changing the weights and biases which will be discussed later.

2.1 The Transfer Function

The transfer function, otherwise known as an activation function, corresponds to the activation state of the node [7]. As mentioned earlier, the transfer function is simulating the biological effect of overcoming a voltage potential to activate an axon. This affects the outputs coming from a node. Mathematically, the transfer function $\sigma(r)$ has to be differentiable, increasing, and have horizontal asymptotes. r corresponds to the prestate in which the activation function is generating an output for. A couple of common functions for $\sigma(r)$ are:

$$\sigma(r) = \tan^{-1}(r), \qquad \sigma(r) = \frac{1}{1+e^{-r}}, \qquad \sigma(r) = \tanh(r) = \frac{e^{2r}-1}{e^{2r}+1}$$

These are the most common functions and some are used in the program Matlab. Each of the aforementioned have different asymptotes but they are generally bounded at $-\pi/2$, -1, 0, 1, and/or $\pi/2$. These three activation functions are graphed in Figure 5 below.

2.2 Computing a Forward Pass

We will now compute the forward pass of a feed-forward neural network. As mentioned earlier in the paper, the initial signals from the input layer exist in $\mathbb{R}^n$ and are denoted $x_1, x_2, ..., x_n$. We define these values as the initial

state condition S_0. To get to the first prestate, we multiply the initial state conditions by the matrix of weights, W_1, and add the corresponding biases $\mathbf{b}_1$. Thus, the first prestate $(P_1) = W_1\mathbf{x} + \mathbf{b}_1$. Then we apply the transfer function to get the next state condition, $\sigma(W_1 x + b_1) = \sigma(P_1) = S_1$. This completes the calculations for the input layer. We will now do the calculations for the second layer of the network or rather, the first hidden layer. We will compute the state conditions just as we did before, however we will use the state condition from the previous layer (input layer) for the calculation of the prestate. We will get the second prestate (P_2) by multiplying by the corresponding weights and adding the biases to the state condition from the input layer. So, $P_2 = W_2 S_1 + b_2$ and then we will apply the transfer function to compute the second state, $\sigma(W_2 S_1 + b_2) = \sigma(P_2) = S_2$. Scaling this up to a complex neural network, this relationship can be used for as many hidden layers as needed. This is accomplished by increasing the indices of each prestate, weight, bias, and state variable. This is visualized by:

$$
\begin{gathered}
x \to P_1 = W_1 x + b_1 \to S_1 = \sigma(P_1) \\
S_1 \to P_2 = W_2 S_1 + b_2 \to S_2 = \sigma(P_2) \\
S_2 \to P_3 = W_3 S_2 + b_3 \to S_3 = \sigma(P_3) \\
S_3 \to P_4 = W_4 S_3 + b_4 \to S_4 = \sigma(P_4) \\
\vdots \\
S_{n-1} \to P_n = W_n S_{n-1} + b_n \to S_n = \sigma(P_n)
\end{gathered}
$$

Once we reach the output layer, the final state condition becomes the output of the neural network. Thus, we have completed a forward pass of a neural network. Furthermore, we can obtain a overall function for the forward pass of the network. We can do this since the final state condition is the composition of many previous functions. For example, the function of a neural network with one hidden layer would be:

$$F(x_i) = W_2(\sigma(W_1 x_i + b_1)) + b_2$$

Now we will take the theory we learned above and apply it to a simple neural network to compute a forward pass. We will use the 2-2-1 neural network in Figure 6 below. The values we need are: The initial conditions $\mathbf{x} = [0.35, 0.9]$, along with the matrices of weights

$$W_{11} = \begin{bmatrix} 0.1 \\ 0.8 \end{bmatrix}, W_{12} = \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix}, W_2 = \begin{bmatrix} 0.3 \\ 0.9 \end{bmatrix} \text{ and } t = 0.5$$

For clarification, the prestate and state notation is P_{ij} and S_{ij} where P is the prestate and S is the state of the jth node of layer i respectively. Remember we need to calculate the prestate and state conditions for each node. Let us begin the computations using logsig as the transfer function [4]. $P_{01} = S_{01} = 0.35$ and $P_{02} = S_{02} = 0.9$ because we are at the input layer. Moving to the hidden layer, the first node has:

$$P_{11} = W_{11}^T [S_{01} \ S_{02}] = [0.1 \ 0.8] \begin{bmatrix} 0.35 \\ 0.9 \end{bmatrix} = 0.755 \text{ and}$$

$$S_{11} = \sigma(P_{11}) = \frac{1}{1 + e^{-P_{11}}} = 0.320$$

The second node in the hidden layer has:

$$P_{12} = W_{12}^T [S_{01} \ S_{02}] = [0.4 \ 0.6] \begin{bmatrix} 0.35 \\ 0.9 \end{bmatrix} = 0.68 \text{ and}$$

$$S_{12} = \sigma(P_{12}) = \frac{1}{1 + e^{-P_{12}}} = 0.336$$

Now we are at the output layer,

$$P_2 = W_2^T [S_{11} \ S_{12}] = [0.3 \ 0.9] \begin{bmatrix} 0.320 \\ 0.68 \end{bmatrix} = 0.708 \text{ and}$$

$$S_2 = \sigma(P_2) = \frac{1}{1 + e^{-P_{12}}} = 0.330$$

Thus, the initial output of the neural network is 0.330. This value is a bit lower than the target value of 0.5, but we can modify the weights and biases later to achieve a better result.

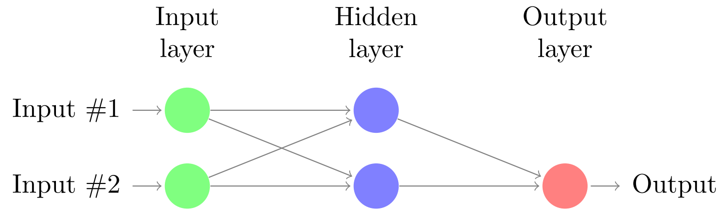


Figure 6: The 2-2-1 neural network we will use as an example throughout the feed-forward neural network section. For values: $\mathbf{x} = [0.35, 0.9]$, $W_{11} = \begin{bmatrix} 0.1 \\ 0.8 \end{bmatrix}$, $W_{12} = \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix}$, $W_2 = \begin{bmatrix} 0.3 \\ 0.9 \end{bmatrix}$, and $t = 0.5$.

2.3 Gradient Descent and Backpropagation of Error

The goal of a neural network is to construct an accurate function between a given range and domain. Our goal is to build the function so that the determined outputs equal the given target values, $F(\mathbf{x}_i) = t^i$ where F is the function, $\mathbf{x}_i$ the inputs, and $\mathbf{t}^i$ the target values. The typical method for finding such a function is to create an error function, and then find the parameters that minimize it. Since we know the targets $\mathbf{t}^i$ and the outputs $\mathbf{y}^i = F(\mathbf{x}_i)$ (as we just described earlier), our error function will be the sum of squared error between these two terms [5]:

$$E = \frac{1}{2}\sum_{i=1}^{p} ||\mathbf{t}^i - \mathbf{y}^i||^2$$

Looking at the error function further, we know that the outputs $\mathbf{y}^i$ are dependent on the weights and biases. So, the true parameters of the error function that will be modified are the matrices of weights W_i and the bias vectors $\mathbf{b}_i$. Thus, we can rewrite the error function to portray this fact:

$$E(W_i,\ \mathbf{b}_i) = \frac{1}{2}\sum_{i=1}^{p} ||\mathbf{t}^i - \mathbf{y}^i||^2$$

where the error function is dependent on the weight matrices W_i and the bias vectors $\mathbf{b}_i$ for each layer.

To decrease the error of the neural net, we will use the gradient of the error function. We take the derivative of the error function and change the weights and biases to move in the opposite direction of the gradient. Moving in the opposite direction of the gradient achieves the fastest descent. Recall that moving in the direction of the gradient is the fastest ascent, so we will do the opposite to achieve the fastest descent. This method is called gradient descent where the parameter u (remember the parameters of the function are the weights W_i and biases $\mathbf{b}_i$) is updated by:

$$u_{new} = u_{old} - \alpha \frac{dE}{du}$$

where α is the learning rate and the equation updates in the opposite direction of the gradient. To determine the term dE/du we use backpropagation. There are other techniques, but backpropagation of error is an efficient way computing the rate of change of the error in a network. For backpropagation, we first run a forward pass through the network to determine all the state conditions for each node. Then we go backwards, determining the partial derivatives through the network to get the error term Δ_m^l for each node [12]. From this, we can get a formula to update the weight W_{mn}^l which connects node n in layer $l-1$ to node m in layer l:

$$W_{mn}^l = W_{mn}^l + \epsilon \frac{dE}{dW} = W_{mn}^l + \epsilon \Delta_m^l S_n^{l-1}$$

where Δ_m^l is an error term that measures how much node m in layer l was responsible for any errors in our output and S_n^{l-1} is the state of node n in layer $l-1$. Δ_m^l is specifically defined as $\Delta_j^L = (t_j - y_j)\sigma'(P_j^L)$ for the output layer L and recursively for layers $l = 1, 2, ..., L-1$ as $\Delta_m^l = \sigma'(P_m^l)\sum_{j \in l+2} \Delta_j^{l+1} W_{jm}^{l+1}$. For clarification in the previous summation, j is indexing each node in the next layer, $l+2$, that connects to node m in layer $l+1$.

For updating the biases, we use the rule:

$$b_k^l = b_k^l + \epsilon \frac{dE}{db} = b_k^l + \epsilon \Delta_k^l$$

This is the theory behind finding the weights and biases that produce a universal approximator function. In practicality, the program Matlab can train a neural network to find a function in a few lines of code.

Continuing with the simple 2-2-1 neural network example in Figure 6, we will run a backwards pass to determine the values we need to increase or decrease each weight by. In effect, this will decrease the error between output of the network and the target values. Note in this example there are no bias terms. Return to Figure 6 to see the weights and inputs for the network. For the backward pass we need to compute the Δ values and state conditions for each node. Recall that we already calculated the state conditions when computing the forward pass, thus we only need to determine the Δ values. Additionally, before we start, we need to determine the function σ'. Since we defined our transfer function as the logsig function, the derivative is simply:

$$\sigma'(P_{ij}) = \sigma(P_{ij})(1 - \sigma(P_{ij})) = S_{ij}(1 - S_{ij})$$

Moving to the computations, we can calculate the derivatives for each node. It follows:

$$\sigma'(P_{01}) = 1,\ \sigma'(P_{02}) = 1,\ \sigma'(P_{11}) = 0.320(1 - 0.320) = 0.218,$$

$$\sigma'(P_{12}) = 0.336(1 - 0.336) = 0.223,\ \sigma'(P_2) = 0.330(1 - 0.330) = 0.221$$

Next, we can calculate the Δ for each node. As we defined previously, Δ for the output layer is:

$$\Delta_1^3 = (t_1 - y_1)\sigma'(P_1^3) = (0.5 - 0.330)(0.221) = 0.03757$$

Moving inward in the network, we can calculate Δ for each node in the hidden layers:

$$\Delta_1^2 = (0.218)(0.3)(0.0375) = 0.00245,$$

$$\Delta_2^2 = (0.223)(0.9)(0.0375) = 0.00753,$$

$$\Delta_1^1 = (1)((0.1)(0.00245) + (0.4)(0.00754)) = 0.00326,$$

$$\Delta_2^1 = (1)((0.8)(0.00245) + (0.6)(0.00754)) = 0.00648$$

Now we can use the formulas above to determine the change in weights and biases. We will use a learning rate of $\alpha = 0.1$. We will start with the change in weights:

$$W_{11}^2 = W_{11}^2 + \epsilon\Delta_1^2 S_{01} = 0.1 + (0.1)(0.00245)(0.35) = 0.100086$$

$$W_{12}^2 = W_{12}^2 + \epsilon\Delta_1^2 S_{02} = 0.8 + (0.1)(0.00245)(0.9) = 0.8002$$

$$W_{21}^2 = W_{21}^2 + \epsilon\Delta_2^2 S_{01} = 0.4 + (0.1)(0.00753)(0.35) = 0.4003$$

$$W_{22}^2 = W_{22}^2 + \epsilon\Delta_2^2 S_{01} = 0.6 + (0.1)(0.00753)(0.9) = 0.6007$$

$$W_{11}^3 = W_{11}^3 + \epsilon\Delta_1^3 S_{11} = 0.3 + (0.1)(0.03757)(0.320) = 0.3012$$

$$W_{12}^3 = W_{12}^3 + \epsilon\Delta_2^3 S_{12} = 0.9 + (0.1)(0.03757)(0.336) = 0.9013$$

We have thus completed one full forward and backward pass through the neural network. The updated weights are above and these will decrease the error function. This means we updated the weights to create a more "accurate" mapping between the inputs and outputs. The updated network will output a value closer to the given target value. One would continue to run forward and backward passes with the updated weights in order to decrease error function until a desired value is achieved.

2.4 Training a Feed-Forward Neural Network

To train a feed-forward neural net, we need a data set and the targets for the data set. From there, we can pick the size and quantity of the hidden layers for establishing accuracy. In Matlab, the code to call a simple feed-forward neural net would be:

```
x = data
t = targets
hiddensize = number of nodes in the hidden layer
net = feedfoward(hiddensize)
net = train(net, x, t)
output=sim(net,x)
```

For an example of training a feed-forward neural network we can creates inputs and targets that the neural network can work with. To create data points, we can create 50 equally spaced points between -2 and 2. These will be the inputs, let's call them X. We can create a function with random error above and below it to get target data, we will call these T. For our purpose, we will use the function $T = \cos(\pi P) + 0.2\text{randn}(\text{size}(P))$ which adds a random error term to the cosine function between 0 and 0.2. The feed-forward neural network with 10 nodes in the hidden layer will produce a smooth trend line (which should loosely represent the cosine function) rather than a line that travels through each point. When the line travels through each point we have "over fit" the neural network. Over-fitting is when the network algorithm follows the data too closely rather than finding a trend line through the data. In an over-fit algorithm, we would not see a smooth trend line, but rather a very jagged line that hits too many data points specifically. Figure 7 shows the difference between the three functions: the altered data, the neural networks training, and the cosine function. We see that the neural network produced a smooth curve that emulates a combination of the cosine function and the altered function.

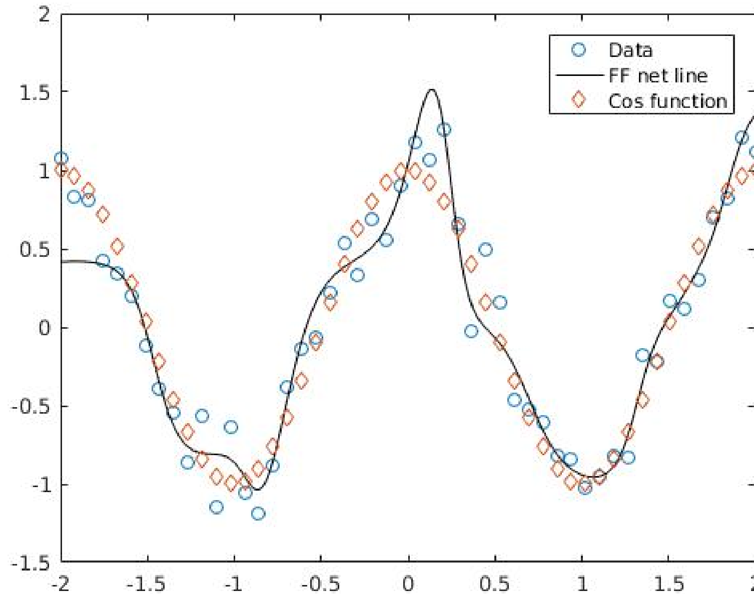


Figure 7: The plots show the difference between the three functions: the altered data (in circles), the neural networks training (smooth line), and the cosine function (diamonds).

2.5 Over Fitting a Neural Network

As mentioned in the previous section, over-fitting is when the network algorithm follows the data too closely rather than finding a trend line through the data. In an over-fit algorithm, we would not see a smooth trend line, but rather a very jagged line that hits too many data points specifically. A more general method of thinking about over-fitting is given by Matlab's documentation [9], "over-fitting occurs when the error on the training set is driven to a very small value, but when new data is presented to the network the error is large. The network has memorized the training examples, but it has not learned to generalize to new situations". This problem occurs in a number of situations. For instance, if there are too many hidden nodes in a single layer or too many hidden layers.

In Figure 8, we took the previous example of a feed-forward network and added enough nodes to the hidden layer to attain over-fitting. Our original network architecture was 1-10-1 while our new net is 1-400-1. When we added these hidden nodes, we allowed the training of the network to be very effective and in result, the network goes through almost every point in the skewed data rather than finding the trend. This is a clear result of over-fitting. When training in Matlab, the program attempts to avoid over-fitting by ending the network training early. The early stopping procedure splits the data into three different sets: training, validation, and testing. The training set is used for computing the gradient as well as updating the network weights and biases. This is were the bulk of the learning is done. The validation set monitors the error of the training set during the training. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to over-fit the data, the error on the validation set typically begins to rise. At the point when the error on the validation set increases, the training stops because that is when over-fitting starts to occur. The final data is the testing set which is data that has not been used in the training or validation checks. This set of data is put through

the network after training to determine how accurate the network training/learning was.
after the network has sbeen trained.

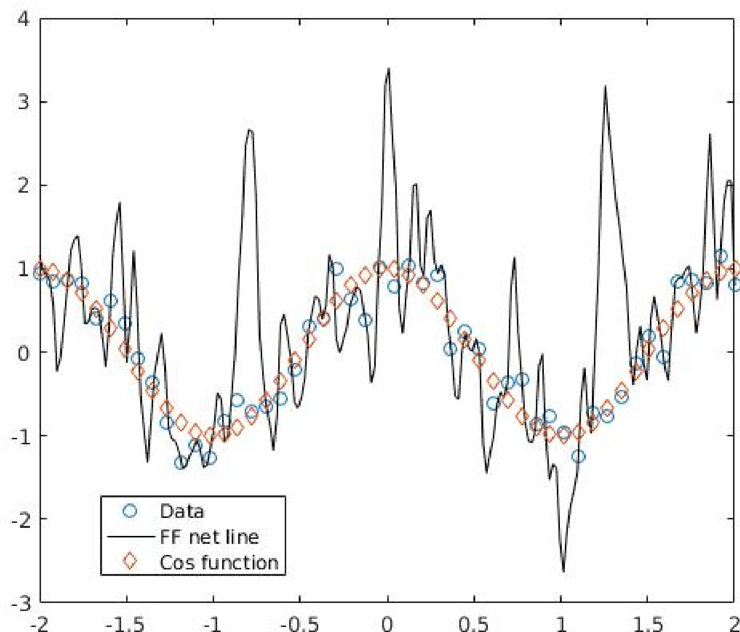


Figure 8: The plots show the difference between the three functions: the altered data (in circles), the neural networks training (fairly jagged line), and the cosine function (diamonds). The important take away of this graph is that the neural network over-fit the data, hitting nearly every skewed data point.

2.6 How to Choose the Number of Nodes and Hidden Layers

For a feed-forward neural network, choosing the number of nodes in each layer and how many hidden layers you need is a difficult and surprisingly under-documented problem. The goal of choosing the correct amount of nodes and hidden layers is to train the network so it maps new inputs accurately and doesn't over-fit the data. The creators of MatLab's help documentation state, "For a network to be able to generalize, it should have fewer parameters than there are data points in the training set. In neural networks, as in all modeling problems, we want to use the simplest network that can adequately represent the training set. Don't use a bigger network when a smaller network will work (a concept often referred to as Occam's Razor)." [2] This provides more general description for choosing the amount of nodes. Altogether, choosing the amount of hidden nodes is a very empirical choice which may require optimization through a lot of testing. One result that is helpful to know comes from the Universality Theorem which is discussed later. The important result regarding node selection is that a single hidden layer can produce the same results as a more complicated architecture. This means that a simple network architecture can work for very complicated problems.

2.7 A Fun Classification Problem

In this section we are going to look at and attempt to classify images of cats and dogs from the Internet. The data set we will look at is a set of $25,000$ images of cats and dogs that was created by kaggle (https://www.kaggle.com/c/dogs-vs-cats/data). An important aspect of this file is that all the images are different sizes (i.e. 400 by 321 pixels versus 450 by 234 pixels). Thus, the first task is to standardize the size of each image. There are two methods we took to standardize the images in Matlab. The first method is to set the image in the upper left corner and add zeros to get to a target size. The second method is to scale the image to the target size through one of Matlab's built in functions. To create the input data matrix for this problem, the second technique was chosen as it would produce less static error in the background. Additionally, we change the picture of a cat or dog to a black and white image. In Figure 9, there are two examples of non-standardized images of a cat and dog.

Figure 9: Two sample images of a cat and dog from kaggle's data set. Note that the two images are not the same pixel size, but were scaled to be equivalent for viewing purposes.

Once we have standardized the images, we can construct a simple classifier neural network in Matlab. We view the accuracy of the classification through a confusion matrix. A confusion matrix is a visual method that provides the percent occurrence that the classification matched the target values. In the matrix, the diagonal values indicate a correct classification, that is a dog for dog or cat for cat. Off the diagonal indicates an incorrect classification, that is the lower left corner indicates we classified a cat as a dog and visa versa for the upper right corner. The goal of the classification is to see the majority of the data along the diagonal. In Figure 10, we have the confusion matrix for this classification problem which indicates the results as described above. In the confusion matrix, the number 0 represents cats and 1 represents dogs.

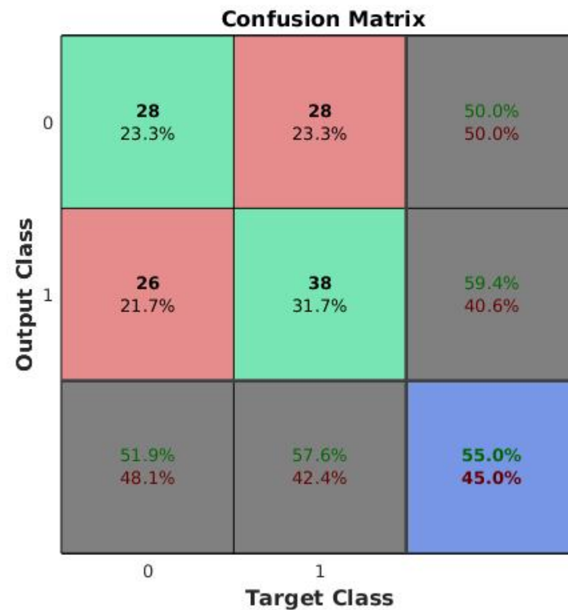


Figure 10: A confusion matrix of the cat and dog classification problem. 0 represents cats and 1 represents dogs.

Looking at this classification, the results are very bad. The classification by the feed-forward network was correct 55% of the time and wrong 45% of the time for the testing set of data. It incorrectly identified cats as dogs 21.7% of the time and dogs as cats 23.3% of the time. Thus, looking at the overall classification percentages, we see this classification isn't much better than guessing with a 50/50 probability. The neural network did not successfully create a function that can correctly classify cats and dogs.

3 Auto-Encoders

An auto-encoder network is an unsupervised learning algorithm where the input and output layers are the same size. Additionally, the three layer network sets the input values equal to the target values. This makes an auto-encoder network an approximation of the identity function [10]. This might seem pointless, but by placing constraints on the network, we can determine important structure about the data. Three constraints applied to auto-encoders are decreasing/increasing the size of the hidden layer, imposing constraints on the weights of the network, and imposing a sparsity constraint on the hidden units. In this paper, we will mention two types of auto-encoders; a regular auto-encoder (or plainly, an auto-encoder) and a sparse auto-encoder. The difference between the two is the size of the hidden layer. An auto-encoder will take the hidden layer to a smaller dimension than that of the input/output layer. For example, an identity map of an 100-50-100 network would be a regular auto-encoder. In contrast, a sparse auto-encoder will take the hidden layer to a larger dimension. For instance, a 2-25-2 network mapping the inputs to themselves would be a sparse auto-encoder. We will go into detail of these types of networks and some of their properties in the following sections.

3.1 Auto-Encoders

As mentioned above, an auto-encoder network is an approximation of the identity function that uses a smaller dimension for its hidden layer. As an example of decreasing the size of the hidden layer, suppose we have a 4-2-4 auto-encoder network such as the network in Figure 7. This would force(encode) the 4 inputs to a lower dimensional representation of 2 points. Then the decoding function must reconstruct the 4 outputs from only 2 points. From this decoding, the algorithm can determine if there is any correlated data or patterns in the data which simplify the decoding. Obviously, an example with more nodes or inputs would be much more interesting, but the smaller example demonstrates the concept. Another simple example of an auto-encoder network is a feed-forward network with one hidden layer that is a lower dimension then the equivalent input/output dimension.

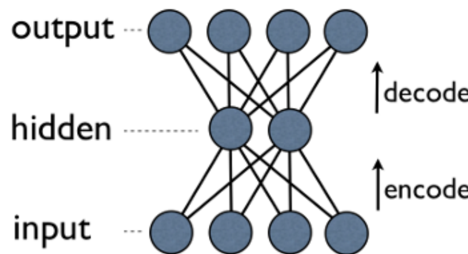


Figure 11: A 4-2-4 auto-encoder network

If we look closer at an auto-encoder, there is an additional term added to the error function that we are attempting to minimize. The new error function becomes:

$$E(W_i,\ \mathbf{b}_i) = \frac{1}{2}\sum_{i=1}^{p} ||\mathbf{t}^i - \mathbf{y}^i||^2 + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2$$

where n_l is the number of layers in the network and s_l is the number of nodes in layer l. The first term is the sum of squared error which we have seen before in feed-forward neural networks. The second term is the L_2 regularization term were λ is the coefficient for the L_2 regularizar. The L_2 regularization term that we added to the cost function prevents over-fitting by controlling (generally decreasing) the weights of the neural network. Over-fitting is when the network algorithm follows the data too closely rather than finding a trend line through the data. In an over-fit algorithm, we would not see a smooth trend line, but rather a very jagged line that hits too many data points specifically. The L_2 regularization term has more implications when we reach sparse auto-encoders later. Now we will explore some implications of auto-encoders in various applications.

3.2 Relationship between Auto-Encoders and Principal Component Analysis

3.2.1 A Brief Introduction to Principal Component Analysis (PCA)

To understand Principal Component Analysis we first need to understand the Singular Value Decomposition or SVD. We can use the SVD factorization on any matrix to determine the rank of the matrix and all four matrix subspaces [6].

Theorem 3.1 *The Singular Value Decomposition (SVD): Let A be any $m \times n$ matrix with rank r, then $A = U\Sigma V^T$ where U is an orthogonal $m \times m$ matrix, Σ is a diagonal $m \times n$ matrix, and V is an orthogonal $n \times n$ matrix.*

The SVD gives us many properties of the matrix A. The diagonal entries of the Σ matrix give us the singular values which are equal to the square roots of the eigenvalues. Also these values are increasing from right to left, so the first singular value is the greatest singular value. The U and V^T matrices correspond to the eigenvectors in increasing order as well. Furthermore, once we have the SVD factorization computing the four fundamental subspaces of a matrix A is simple. A basis for the columnspace of A is the first r columns of U or symbolically as $[u_i]_{i=1}^r$. This will be important later. A basis for the nullspace of A is $[v_i]_{i=r+1}^n$. A basis for the rowspace of A is $[v_i]_{i=1}^r$. A basis for the nullspace of A^T is $[u_i]_{i=r+1}^m$. Furthermore, we can calculate a reduced SVD.

Theorem 3.2 *The reduced SVD: Let A be any $m \times n$ matrix with rank r and $A = U\Sigma V^T$ be the SVD of A with rank r, then $A = \tilde{U}\tilde{\Sigma}\tilde{V}^T = \sum_{i=1}^r \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T$ where $\tilde{U}$ is an orthogonal $m \times r$ matrix, $\tilde{\Sigma}$ is a diagonal $r \times r$ square matrix, and $\tilde{V}$ is an orthogonal $n \times r$ matrix.*

The reduced SVD is significant because it reduces the matrices to the size of the rank. We will see the importance of the reduced SVD with determining the best basis. First we will state the theorem of the best basis.

Theorem 3.3 *The Best Basis Theorem: X is an $n \times p$ matrix of p points in $\mathbb{R}^n$ with mean $\bar{x} \in \mathbb{R}^n$. X_m is the $n \times p$ matrix of mean subtracted data. C is the covariance of X, $C = (1/p)X_m X_m^T$. Then the best orthonormal basis is given by the leading k eigenvectors of C, for any k.*

Now we can connect the best basis computation from the covariance matrix to the SVD. Consider the SVD of the $n \times p$ matrix X with rank k. We get

$$C = \frac{1}{p-1} X X^T = \frac{1}{p-1} U\Sigma V^T V \Sigma^T U^T = U(\frac{1}{p-1}\Sigma^2)U^T$$

Comparing this to the best basis theorem above, we see the best basis vectors for the column space of X are the first k columns of U and the best basis vectors for the rowspace of X are the first k columns of V. This is a very powerful result for data analysis because we can determine the best set (best representative) vectors for any given data set. The process of finding the best basis is the same as finding the principal components. Thus, we have just described Principal Component Analysis (PCA).

3.2.2 The Relationship Between PCA and Auto-Encoders

Now that we have an elementary understanding of principal component analysis, we can determine the relationship between auto-encoders and PCA. We need to first recognize that both operations reduce the data to a lower dimension. PCA uses the reduced SVD which reduces the data down to the dimension k, where k is the rank of the matrix. On the other side, an auto-encoder encodes the data into a lower dimension. If we encode the data down to a hidden layer with k nodes, where k is the rank, then we get a subspace that is approximately equal to that produced by PCA. To do this computation, we extract the matrix that encoded the data to dimension k of the auto-encoder and compare it to the best basis for the column space from the SVD. When compared, we should see the two subspaces are equal within a small error. To compare them we will produce an image of the two matrices. In Figure 13, the top two images are the best two vectors of the column space from the SVD. The bottom two images are the vectors that are used to encoder the auto-encoder. Comparing these two images we see they are very similar to each other. One method to determine how similar these images are to each other is to calculate the reconstruction error. The reconstruction error is defined as $\sum ||X^i - \text{proj}(X^i)||^2$ for the SVD and $\sum ||X^i - f(X^i)||^2$ for the auto-encoder. The two errors from the picture were determined to be $1.6695 \cdot 10^4$ for the SVD and $1.6547 \cdot 10^4$ for the auto-encoder. Since the reconstruction errors are nearly the same, we can conclude that the images are approximately equal. Thus, we have just shown that the subspace formed by taking the first k vectors of the column space from the SVD is approximately equal to the vectors formed from encoding an auto-encoder neural network.

Another metric to compute the error between the two subspaces is to calculate the angle between the two subspaces. We call the angles between the vectors within the subspaces the principle angles. Recall from linear algebra that to compute the angle between two vectors, we compute $\mathbf{u} \cdot \mathbf{v}/(|\mathbf{u}||\mathbf{v}|)$. For a quick example and visual, we will calculate the angle between a plane and a line in 3-d. Let's take F to be the line $x + 3y = 0$ and G to be the plane $x + y - 2z = 0$. We first determine the direction vector of the line and the normal vector of the plane which are $(1, 3, 0)$ and $(1, 1, -2)$ respectively. Now we will do the computation, so:

$$\frac{|(1,3,0) \cdot (1,1,-2)|}{|(1,3,0)||(1,1,-2)|} = \frac{4}{\sqrt{60}}\text{radians}$$

Figure 12 shows the 3-d graph of the two planes intersecting.

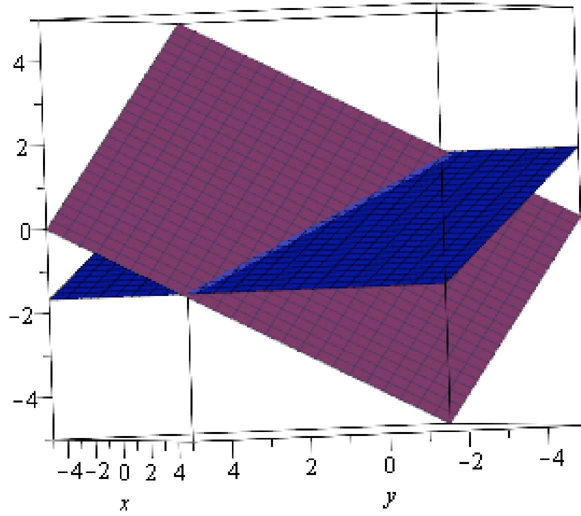


Figure 12: A 3-d plot showing the two functions $x + 3y = 0$ and $x + y - 2z = 0$. The angle between the two planes is a $\frac{4}{\sqrt{60}}$

To compute the principle angles between the vectors from two different subspaces we will take advantage of the following theorem:

Theorem 3.4 *Let F and G be subspaces of $\mathbb{R}^m$ and*

$$p = dim(F) \geq dim(G) = q \geq 1.$$

If the columns of $U_f \in \mathbb{R}^{m \times p}$ and $U_G \in \mathbb{R}^{m \times q}$ define orthonormal bases for F and G respectively, then

$$\max_{u \in F} \max_{u \in G} u^T v = \max_{y \in \mathbb{R}^p} \max_{z \in \mathbb{R}^q} y^T (Q_F^T Q_G) z$$

We are essentially commuting what we discussed above $(\mathbf{u} \cdot \mathbf{v})/(|\mathbf{u}||\mathbf{v}|)$, but we can take advantage of some helpful properties from the SVD. From the SVD, we know we are using two orthonormal vectors the expression becomes the dot product between $\mathbf{u}$ and $\mathbf{v}$. If we take the dot product of the maximum $\mathbf{u}$ and $\mathbf{v}$ we would get the maximum angle. What the above theorem is describing is the maximum $\mathbf{u}$ and $\mathbf{v}$ correspond to the maximum $\mathbf{y}$ and $\mathbf{z}$ from the SVD of a matrix C which is described next. Further more, from this theorem we get the following result,

$$\text{The SVD of } Y^T (Q_F^T Q_G) Z = Y^T (C) Z = \text{diag}(\cos(\theta_k))$$

From this theorem we can get a matrix describing the angles between the subspaces.

$$\begin{bmatrix} 9.1985 & -18.3827 \\ 13.7385 & -3.5537 \end{bmatrix}$$

The important values in the matrix are the diagonals corresponding to the angles between the first vectors in each subspace and the second vectors in each subspace. We see the first basis vector of the column space from the SVD is about 9.2 degrees off from the first basis vector generated from the encoder matrix. Similarly, the second basis vectors are only 3.6 degrees off from each other. Notice that the values are close to zero which would indicate that the two subspaces are the nearly the identical. Because the two vectors that we are comparing come from two very different analytical methods we should expect some error, especially when encoding 109 vectors into two. Thus, we see that the two methods, PCA and auto-encoders, have a very similar subspace that connects the two mathematical subjects.

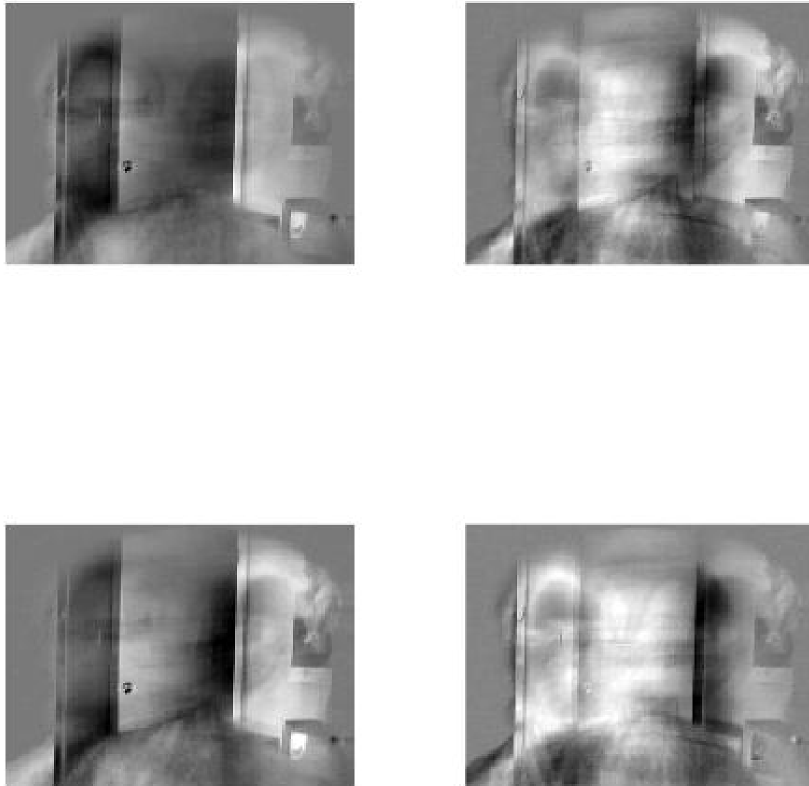
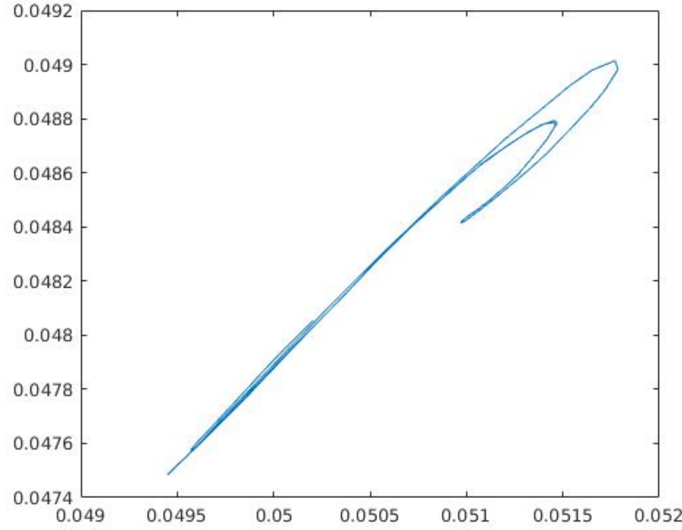


Figure 13: The top two images are constructed from the the best two vectors of the columns space from the SVD (PCA). Specifically, each picture is either the first or second column of the U matrix from the reduced SVD. Each vector is then reconstructed into an image of 160 by 190 pixels. The bottom two images are constructed from the encoding of the auto-encoder. The encoder matrix was the same dimension as the reduced SVD, a matrix of two vectors.

As another example of an auto-encoder, we will encode the 109 images from the previous example where we demonstrated the relationship between PCA and auto-encoders into a lower dimension, view the weights of the encoding function, and then view the reconstructed images. The first part of Figure 14 corresponds to the encoding matrix down to 2-d. This graph represents the path of the weights for the auto-encoder. The second image shows 25 reconstructed images from the auto-encoder. Notice we lost a fair bit of data from the auto-encoder since we projected it down into two dimensions.



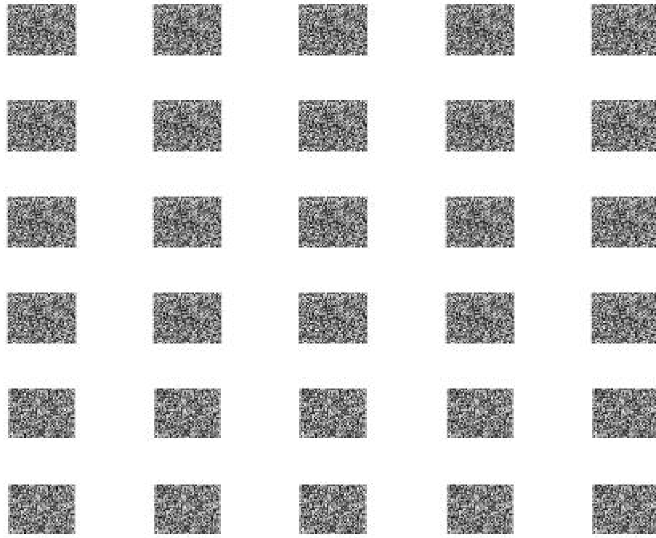


Figure 14: The images produced from the encoding function from a sparse auto-encoder using the best two basis's from the aforementioned auto-encoder to $\mathbb{R}^{25}$. We represent the 25 vectors as the above 25 images.

3.3 Sparse Auto-encoders

Now we will describe a sparse auto-encoder. We first set a sparsity parameter ρ of our choosing (usually small), where ρ is the activation of the neuron. Next define

$$\rho_j = \frac{1}{m}\sum_{i=1}^{m}[a_j^{(2)(x^{(i)}}]$$

to be the average activation of hidden unit j. We would like to enforce the constraint $\rho_j = \rho$ as the sparsity constraint. To do this, we add an extra penalty to the cost function. This is the main important distinction with a sparse auto-encoder. We will add two regularization terms to the sum of squared errors. We get:

$$E(W_i,\ \mathbf{b}_i) = \frac{1}{2}\sum_{i=1}^{p}||\mathbf{t}^i - \mathbf{y}^i||^2 + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2 + \beta\sum_{j=1}^{s_2}\rho\log\frac{\rho}{\rho_j} + (1-\rho)\log\frac{1-\rho}{1-\rho_j}$$

The first term is the sum of squared error which we have seen before. The second term is the L_2 regularization term were λ is the coefficient for the L_2 regularizer. This was the regularization term we added for the regular

auto-encoders. The third term is the sparsity regularization term that corresponds to the sparsity constraint ρ. The value ρ corresponds to the average activation of each hidden neuron. By setting the ρ to some number, we are constraining the activation of each hidden node to be close to ρ. The third term we added to the cost function enforces that constraint. The following sections present examples of sparse auto-encoding. These include the MNIST data set and untying a torus knot.

3.4 Unknotting the knot

As an application of auto-encoders, we will untie a knot. As a quick aside, we will define a knot and an unknot in the context of knot theory. A knot is defined as a closed, non-self-intersecting curve that is embedded in three dimensions and cannot be untangled to produce a simple loop (i.e., the unknot) [14]. Thus, when we are talking about unsolved knots, we are referring to unknots. An example of an unknot is a crinkled up rubber band. The rubber band is an unknot because it can always be made into a non-crossing loop by pulling it apart. The general strategy for untying the knot is to use a sparse auto-encoder to encode the knot into a higher dimensional space. Once the knot is in a higher dimension, let's say 12 dimensional space (it only needs to be in 4 dimensional space to unknot itself though), the knot will untie itself. Once the knot is untied, we use a regular auto-encoder to encode the higher dimensional representation into a 2 or 3 dimensional space. Upon graphing the lower dimensional representation, we will see a loop or line that never crosses itself. This signifies that the knot has been untied and is now the unknot.

The knot that we will focus on is the $(4, 3)$ torus which is a function of three parametric equations:

$$x = \cos(3t)(3 + \cos(4t)),\ y = \sin(3t)(3 + \cos(4t)),\ \text{and } z = \sin(4t) \text{ with } 0 \leq t \leq 2\pi$$

The knot in $3 - d$ is shown below in the top image of Figure 11. Next, we used a sparse auto-encoder to encode the knot into a 12 dimensional space. In 12 dimensional space, the knot will become the unknot. We then encode the unknot down to a 2 and 3 dimensional representation using an auto-encoder to view it. The bottom graph within Figure 11 is the unknot corresponding to the $(4, 3)$ torus from above.

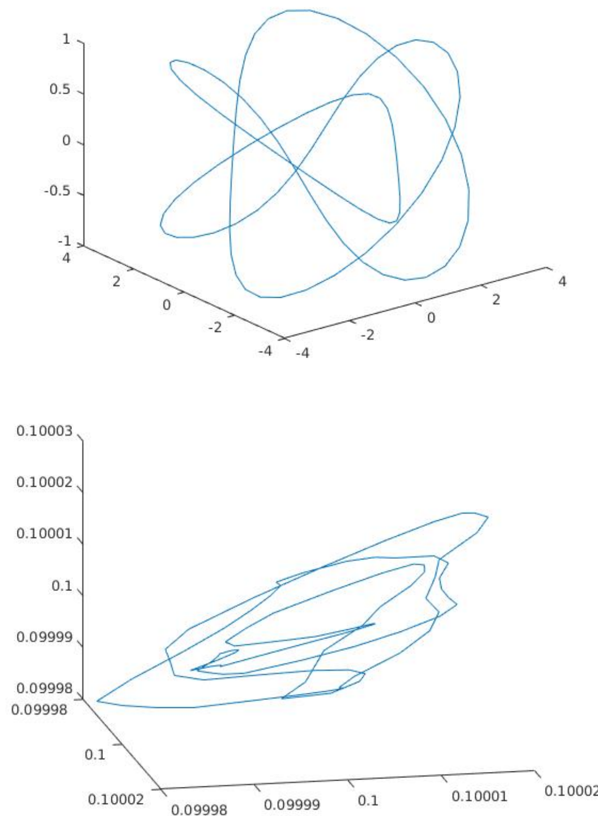


Figure 15: The $(4, 3)$ torus knot formed by the above parametric equations plotted in $3 - d$. Under it is the unknot in $3 - d$.

3.5 The MNIST Dataset

The MNIST (Mixed National Institute of Standards and Technology) data set is a collection of handwritten digits. There are $60,000$ training images and $10,000$ testing images. The set of images comes from a combination of two NIST databases: Special Database 1 (SD1) which consists of digits written by high school students and Special Database 3 (SD3) which consists of digits written by employees of the United States Census Bureau. Each SD database contributes $30,000$ images which are 20 x 20 pixels to the set. This data set is commonly used for training and testing machine learning (i.e., neural networks). The goal of the training is to classify each of the hand written digits to its corresponding label or true number. Thus far, the best training obtained a 0.21 percent error rate by using an ensemble of 5 convolutional neural networks (a type of feed-forward neural network). Below in Figure 16 are four sample images from the MNIST data set.

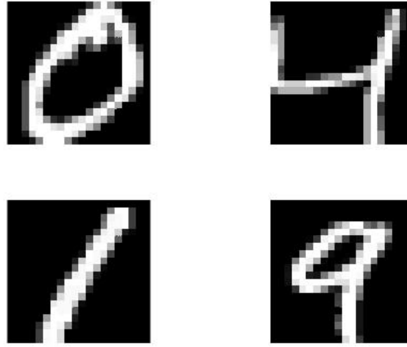


Figure 16: The four images correspond to four 20x20 matrices that are within the MNIST data set. The corresponding labels to the images are $0, 4, 1, 9$ in order from top left to bottom right.

Now we will look at how a sparse auto-encoder can encode the images to a higher dimension. In Figure 17, we encoded the data set up in dimension. Resulting in each image becoming "specialized". More specifically, each neuron corresponding to an image specializes by responding to some feature that is only present in a small subset of the training examples. This is because we can think of each node corresponding to a single image, if there are more nodes than images, each node can specialize on a special feature of the images. Thus, in Figure 17 we see each image has the general shape of a number, but not exactly the same shape.

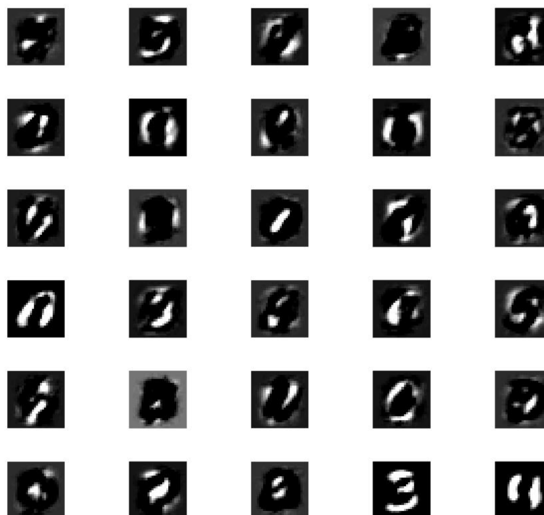


Figure 17: These images correspond to the 1000 images in the hidden layer of the sparse auto-encoder.

4 Deep Nets and Deep learning

In this section we will return to the cats and dogs example as well as the MNIST data sets presented in the previous sections. In each section we constructed a simple classifier network for cats and dogs or a sparse auto-encoder for the MNIST data set. Now we will construct a deep network to classify the data sets and compare the differences. A deep network is a series of stacked feed-forward, softmax, and auto-encoder layers which together perform deep learning. The ability to stack multiple networks creates a much better training for the deep net as compared to an individual network. Another difference between deep nets and the other networks discussed in this paper is the huge size of a deep net. Deep nets have become a much more powerful tool recently due to the vast improvements in processing and data storage technologies. Researchers can use massive amounts of data to train the deep net. For example, Google can train a voice recognition deep net with hundreds of thousands of hours of people talking. This creates a much more powerful recognition tool.

4.1 Softmax Algorithm

The softmax algorithm is a probability function that distributes a k dimensional vector $\mathbf{x}$ of arbitrary real values into a k dimensional vector $\sigma(\mathbf{x})$ of real values in the range $(0,\ 1)$ that add up to 1 [13]. The function is:

$$\sigma(x)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

The softmax algorithm is often used as the final layer of neural network for a a classification problem. Since the function returns a real value within a known distribution. Furthermore, if we set the maximum value for each distribution in the data set, with each piece of data a column vector, to 1 and the rest to 0, then we can use that maximum value in the vector as our classification output. For example, in the cats and dogs data set, each image corresponded to a column vector. The maximum value in the column vector was set to one and the other set to zero. If the one value was in the top row, the image corresponded to a cat and if in the bottom, it corresponded to a dog. Altogether, a softmax layer is often the final layer in a deep net because it is a accurate method to classify data.

4.2 Cats and Dogs Classifier with a Deep Net

As described above, a deep net is a series of stacked networks. In this example, we move from a regular feed-forward classifier which performed terribly and institute a deep net. The deep net here will consist of two auto-encoder networks followed by a softmax layer. The first auto-encoder has 10 hidden nodes and the second has 3 hidden nodes. Figure 18 shows the confusion matrix for the training of the deep net. We see that 66.4% of the time it classified cats and dogs correctly while 33.6% of the time, it classified incorrectly. Although this is not nearly a perfect classification, the network is recognizing the images of cats and dogs more. Also, the deep net is working much better than the simple feed-forward, which was essentially guessing the classification.

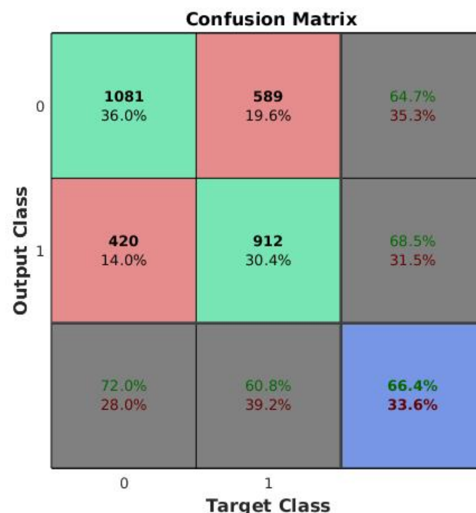


Figure 18: The confusion matrix from the deep net classifier for the cats and dogs dataset.

4.3 MNIST Data Set Classifier with a Deep Net

Using a deep net, we can classify the images within the MNIST data set to their corresponding numeric values between 0 and 9. For this deep net, we used two auto-encoders followed by a soft max layer. The auto-encoders had hidden layers of 100 and 50, so the network reduced the 60,000 images to 100, then to 50 and classified them using a softmax layer corresponding to the 10 numbers. Figure 19 shows the confusion matrix for the testing of the neural network with about 72% being classified correctly and 28% being classified incorrectly. Additionally, some of the interesting results from the confusion matrix seemed to cause slight confusion. The output from the deep net confused the numbers 0, 3, and 8 with the number 5. We see this because the output of the deep net mis-classified zero as five 133 times, mis-classified three as five 177 times and eight as five 103 times. This is likely because the number 0, 3, and 8 look very similar to 5 when they are poorly drawn. One other big error in classification was the mis-classification of 6 as 9, a likely error because the numbers are rotations of each other.

Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	338 6.8%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	3 0.1%	0 0.0%	0 0.0%	0 0.0%	99.1% 0.9%
2	0 0.0%	381 7.6%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	99.5% 0.5%
3	2 0.0%	110 2.2%	368 7.4%	5 0.1%	2 0.0%	5 0.1%	18 0.4%	2 0.0%	0 0.0%	1 0.0%	71.7% 28.3%
4	0 0.0%	19 0.4%	7 0.1%	256 5.1%	5 0.1%	12 0.2%	0 0.0%	12 0.2%	12 0.2%	8 0.2%	77.3% 22.7%
5	3 0.1%	0 0.0%	7 0.1%	0 0.0%	385 7.7%	1 0.0%	12 0.2%	6 0.1%	1 0.0%	105 2.1%	74.0% 26.0%
6	133 2.7%	0 0.0%	24 0.5%	177 3.5%	36 0.7%	419 8.4%	22 0.4%	18 0.4%	103 2.1%	48 1.0%	42.8% 57.2%
7	6 0.1%	0 0.0%	28 0.6%	0 0.0%	7 0.1%	6 0.1%	411 8.2%	0 0.0%	0 0.0%	0 0.0%	89.7% 10.3%
8	1 0.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	400 8.0%	1 0.0%	3 0.1%	98.3% 1.7%
9	5 0.1%	19 0.4%	58 1.2%	69 1.4%	36 0.7%	14 0.3%	16 0.3%	15 0.3%	374 7.5%	20 0.4%	59.7% 40.3%
10	1 0.0%	1 0.0%	0 0.0%	2 0.0%	26 0.5%	1 0.0%	0 0.0%	109 2.2%	3 0.1%	298 6.0%	67.6% 32.4%
	69.1% 30.9%	71.9% 28.1%	74.6% 25.4%	50.3% 49.7%	77.2% 22.8%	91.5% 8.5%	85.3% 14.7%	71.0% 29.0%	75.7% 24.3%	61.7% 38.3%	**72.6%** **27.4%**

Figure 19: The confusion matrix from the deep net classifier for the MNIST dataset.

A recent article was published regarding deep learning that has significant implications for machine learning. The article posited that the networks were memorizing the data rather than learning the general trends. This was shown by training two deep nets, the first on the MNIST data set and the second on the GTSRB data set. For reference, the GTSRB data set is a set of over 39,000 images which consist of 43 different kinds of traffic signs. Once each deep net was trained, the testing images were run with correct classification rates of 98.7-99.2% for the MNIST and 94.8-98.08% for the GTSRB. The testing set of images were then multiplied by negative one to create their corresponding negative image. The negative images should be correctly identified by the deep net since the patterns for the images were kept exactly the same. However, when they tested the negative images on the deep net, the network completely failed to correctly classify the negatives. The classification rates did not get above 16% and had a low of 3.79%. This suggests that the deep net was memorizing the images because it could not classify an image with the same pattern, but with inverted colors. Here, we have replicated their results to the best of our abilities. In Figure 20 we see four negative images from the MNIST data set and going back to Figure 16, we see four positive

images from the data set. Upon classification, using the same deep net as described in the aforementioned MNIST classification section, we get a classification rate of 1.3% for the negative images. Our positive classification rate was 72.6% which shows that the deep net could not classify the negative images.

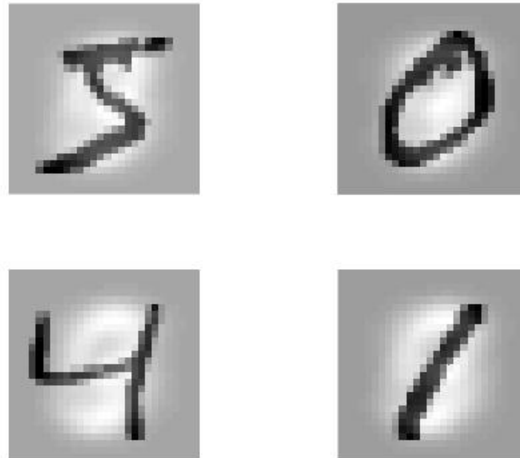


Figure 20: Four negative images from the MNIST dataset. They are simply the inverse colors. Note the images are mean subtracted which may have caused the unusual "whiteness" in the images.

Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	0 0.0%	49 1.0%	9 0.2%	5 0.1%	2 0.0%	10 0.2%	8 0.2%	10 0.2%	4 0.1%	7 0.1%	0.0% 100%
2	311 6.2%	1 0.0%	200 4.0%	47 0.9%	192 3.8%	62 1.2%	286 5.7%	221 4.4%	110 2.2%	259 5.2%	0.1% 99.9%
3	89 1.8%	48 1.0%	20 0.4%	195 3.9%	170 3.4%	135 2.7%	15 0.3%	171 3.4%	149 3.0%	158 3.2%	1.7% 98.3%
4	7 0.1%	0 0.0%	26 0.5%	1 0.0%	43 0.9%	12 0.2%	25 0.5%	4 0.1%	6 0.1%	1 0.0%	0.8% 99.2%
5	57 1.1%	19 0.4%	16 0.3%	32 0.6%	0 0.0%	54 1.1%	2 0.0%	1 0.0%	20 0.4%	0 0.0%	0.0% 100%
6	9 0.2%	187 3.7%	76 1.5%	49 1.0%	63 1.3%	37 0.7%	42 0.8%	131 2.6%	14 0.3%	28 0.6%	5.8% 94.2%
7	12 0.2%	1 0.0%	1 0.0%	148 3.0%	1 0.0%	99 2.0%	0 0.0%	14 0.3%	40 0.8%	24 0.5%	0.0% 100%
8	2 0.0%	224 4.5%	127 2.5%	31 0.6%	25 0.5%	41 0.8%	101 2.0%	5 0.1%	150 3.0%	6 0.1%	0.7% 99.3%
9	1 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	5 0.1%	1 0.0%	6 0.1%	0 0.0%	0 0.0%	0.0% 100%
10	1 0.0%	1 0.0%	18 0.4%	1 0.0%	2 0.0%	3 0.1%	2 0.0%	0 0.0%	1 0.0%	0 0.0%	0.0% 100%
	0.0% 100%	0.2% 99.8%	4.1% 95.9%	0.2% 99.8%	0.0% 100%	8.1% 91.9%	0.0% 100%	0.9% 99.1%	0.0% 100%	0.0% 100%	1.3% 98.7%

Figure 21: The resulting confusion matrix when negative images are run through a deep net that was trained on positive images. This is a terrible training, possibly showing memorization. The positive training results for the deep net were 76% correct and 24% incorrect classification.

This result has significant implications for machine learning, if indeed the deep nets are memorizing the images rather than learning the trends. If deep nets are not learning the trends, but rather memorizing patterns, deep net based algorithms and AI will be greatly impacted. Furthermore, future research and applications would be effected because the current theory regarding deep nets could be misunderstood.

5 Conclusion and Final Result of Neural Networks

In this paper, we have provided examples for different types of neural networks that have tremendous applications in mathematics. One of the most important characteristics of a neural network has been suggested throughout the paper in theory and in different examples. The result is as follows: A feed-forward neural network is a universal function approximator. We can find a neural network that will approximate any function with an arbitrarily small error. Additionally, the neural network needs only a single hidden layer. This result is an implication of the Universal Approximation Theorem.

Theorem 5.1 *The Universal Approximation Theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function* [3].

Knowing that we can approximate any continuous function with a simple feed-forward neural network with only one hidden layer is a tremendous tool in mathematics. Once we train the network on some data, we can use it to approximate whatever the training data was based on. Thus, the applications to neural networks are endless. Additionally, with the vast improvements of deep learning and deep nets, these networks are being used for tremendously complicated programs. For example, deep networks are currently being used as the main algorithm in the field of artificial intelligence as well as voice recognition. In conclusion, we have presented the reader with an introduction to the theory and useful applications of feed-forward neural networks, auto-encoder networks, and deep nets. Now we will leave the reader with more complex and real life applications of these powerful tools to extend the reach of this paper. Google's artificial intelligence systems are based on deep nets which are used for classifying objects, translating languages using webcams, computer drawing recognition, as well as other applications. Please visit Google's artificial intelligence website at:

`https://aiexperiments.withgoogle.com/`

References

[1] Jochen Fröhlich. Supervised and unsupervised learning, Jan 2017.

[2] Hagan, Martin T., Demuth, Howard B., Beale, Mark Hudson, De Jesús, Orlando. *Neural Network Design*. eBook, 2014.

[3] Hassoun, Mohamad H. *Fundamentals of Artificial Neural Networks*. MIT Press, 1995.

[4] Douglas R. Hundley. Backproperror, Jan 2017.

[5] Douglas R. Hundley. Ffneural, Jan 2017.

[6] Douglas R. Hundley. Linear algebra fundamentals, Jan 2017.

[7] Horst-Michael Gross Klaus Debes, Alexander Koenig. Transfer functions in artificial neural networks - a simulation-based tutorial. *Brains, Minds, & Media*, Jul 2005. http://www.brains-minds-media.org/archive/151/supplement/bmm-debes-suppl-050704.pdf.

[8] Nicholas Lincoln. Identifying subatomic particles with neural networks, Feb 2017.

[9] Matlab. Math works, Apr 2017.

[10] Andrew Ng. sparseautoencoder, Jan 2017.

[11] Wikipedia. Action potential, Jan 2017.

[12] Wikipedia. Backpropagation, Jan 2017.

[13] Wikipedia. Softmax function, Jan 2017.

[14] Wolfram Math World. Knot, Apr 2017.

6 Matlab code/Appendix

In this section we present the Matlab code for each of the examples. Additionally, the first section in the appendix is essentially a "how to" for writing an auto-encoder in Matlab. A description of each parameter within the auto-encoder class is outlined and the operations to optimize the function are briefly described.

6.1 Auto-Encoders in Matlab

In this section we will provide a description of all the parameters that go into Matlab's auto-encoder function and how to build an auto-encoder with specified values. Let's start by reviewing the different variables inside the auto-encoder function in the order they appear. To initiate the training of the auto-encoder, we must first call the function: trainAutoencoder. The first input in the function is the data matrix, which is then followed by the hidden layer size. For auto-encoding we choose a dimension for the hidden layer smaller than what the data is represented in, and for sparse auto-encoding we choose a dimension larger. To set the hidden layer size you can either input a positive integer or specify a variable with a positive integer value. Our function now looks like: trainAutoencoder(X, hiddenSize) for some data X and hidden layer size. Moving on, we will format the rest of the arguments in the function as the argument's name followed by its value in pairs and single quotes: 'name1', 'value1', ... 'name2', 'value2', ... 'nameN', 'valueN'. Now we will go through the possible arguments. We can choose our transfer function for the encoder and decoder functions. The name we provide is either 'EncoderTransferFunction' or 'DecoderTransferFunction'. For the corresponding encoder value, we choose from the linear function 'satlin' or the logistic function 'logsig'. For the corresponding decoder value, we choose from the linear function 'purelin', the positive saturating linear function 'satlin', or the logistic function 'logsig'.

As an example, if we want to train an auto-encoder with linear transfer functions we would use the code in Figure 22.

```
autoenc= trainAutoencoder(Xmean,hiddensize,...
    'EncoderTransferFunction','satlin',...
    'DecoderTransferFunction','purelin',...
```

Figure 22: This is an example of an auto-encoder function with linear encoding and decoding transfer functions.

The default setting, if not specified for both functions, is 'logsig'. Continuing on, we can specify the maximum number of training epochs by the value pair: 'MaxEpochs', positive integer value. The default maximum is 1000. Next, we can specify the coefficient for the L_2 weight regularizer which controls the relative importance between the sum of squared error and L_2 regularization term. The L_2 regularization term helps prevent over fitting by controlling the magnitude of the weights. This should typically be quite small, the default setting in Matlab is 0.001. Generally, start by decreasing the coefficient for the L_2 weight regularizer to optimize the training. To set this term we state the value pair: 'L2WeightRegularization', .02. Continuing on, we will now define the sparsity proportion or as mentioned above, the sparsity parameter ρ_i. The value pair to set the sparsity parameter, a positive scalar in the range of 0 to 1, is: 'SparsityProportion', 0.1. Generally, a low sparsity parameter increases specialization of each neuron and increases sparsity. Next, we will set the sparsity regularization coefficient which controls the impact of the sparsity regularizer term in the cost function. The value for the coefficient is a positive scalar value with the default set to 1. To change the variable, we would state the value pair: 'SparsityRegularization', 2. The value pair arguments above are the important variables to optimizing a training for both types of auto-encoders. In Figure 23, we can see an example of the code one would use to train an auto-encoder neural network. Additionally, we indicated some suggestions and possible trends for the values of each argument. These are great starting points, but only with testing can we optimize the auto-encoder network.

The following value pair arguments are additional commands to alter the training but are almost always left as default. This is because they are either vitally important to the training or very unimportant. Thus, we will only state the commands and their default settings (default means they are assumed without indicating them) so we know they exist. The first is to specify the costs/error/loss function which for auto-encoders is stated as: 'LossFunction', 'msesparse'. The second is showing the training window where the default is true. If we don't want to see the training, we could state the value pair: 'ShowProgressWindow', false. The third is specifying the training algorithm The default is scaled conjugate gradient descent stated as: 'TrainingAlgorithm', 'trainscg'. The fourth is a true or false statement to indicate the use of GPU for training. The default is false and is called by: 'UseGPU', false.

One we have trained the auto-encoder using the arguments from above, we can use the data in various methods. Two important tools we can use are the encode and decode functions. The encoding function will return the matrix that encoded the input data into the hidden layer. To call this function we write: $Z = \text{encode}(\text{autoenc}, X)$ where

```
autoenc= trainAutoencoder(Xmean,hiddensize,...
    'EncoderTransferFunction','satlin',...
    'DecoderTransferFunction','purelin',...
    'MaxEpochs',1200,...
    'L2WeightRegularization',0.01,...
    'SparsityRegularization',4,...
    'SparsityProportion',0.10);
```

Figure 23: This is an example of an auto-encoder function with linear transfer function, a maximum of 1200 epochs, and specified regularization terms.

Z will be the encoding matrix, autoenc is the trained auto-encoder, and X is the input data. On the other side, the decode function will return the matrix that decoded the hidden layer into the output layer. Similarly, to call the decode function we write: $Y = \text{decode}(\text{autoenc}, M)$ where Y will be the decoding matrix, autoenc is the trained auto-encoder, and M is the data encoded by autoenc. An example of the application of these two function can be seen with the sparse auto-encoder example above with 30 images. After optimizing the training of the network, we used the matrix from the encoding function to return a matrix that contained the 30 images. Another function that we can use is the predict function. Predict returns the predictions Y for the input data X using the auto-encoder autoenc. It is called by writing: $y = \text{predict}(\text{autoenc}, X)$ where the variables are as above. The next few functions are helpful to know, but are used less often. They all use the trained auto-encoder autoenc. The first will be the function view which allows us to view the auto-encoder. To call it write: view(autoenc). Next, we can view the weights by calling the unction plotweights(autoenc). We can also convert an auto-encoder object into a network object by using: network(autoenc). Finally, we can generate a function to run the auto-encoder autoenc on input data. We call: generatefunction(autoenc). With these tools we can use auto-encoders for many different problems.

6.2 Feed-Forward Example

```
%Create some artificial data
P=linspace(-2,2,50); %creates 50 equally space points between -2 and 2
T=cos(pi*P)+0.2*randn(size(P)); %the function we are trying to emulate

% We'll build a 1-10-1 network and train it:
net=feedforwardnet(10); %10 nodes in hidden layer
net=configure(net,P,T); % Initialize weights and biases

% Training:
net=train(net,P,T);     %Training command
y2=sim(net,P);          % Network output after training
```

6.3 Showing The Relationship Between The SVD and Auto-Encoders

```
%% PCA using the SVD
load author
X11 = double(Y1);
Y1mean=mean(X11,2);
X1mean=X11-repmat(Y1mean,1,109);
[U S V]=svd(X1mean,'econ');

%The best basis vectors for the column space of X are the first k
%columns of U. Similarly, the best basis vectors for the row space of
%X are the first k columns of V .

figure(1)
for j=1:2
    subplot(1,2,j)
    imagesc(reshape(U(:,j),120,160)); colormap(gray); axis equal; axis off
end
```

```
%% Autoencoders
load author
hiddensize=(2);
X = double(Y1);
Ymean=mean(X,2);
Xmean=X-repmat(Ymean,1,109);
Xmean=Xmean';
autoenc= trainAutoencoder(Xmean,hiddensize,...
        'EncoderTransferFunction','satlin',...
        'DecoderTransferFunction','purelin',...
        'L2WeightRegularization',0.01,...
        'SparsityRegularization',4,...
        'SparsityProportion',0.10);
%%
z=encode(autoenc,Xmean);
z=z';
% v=decode(autoenc,z);
% figure(1)
%The two pictures in this figure are created from the best basis's of the
%columns space of the data from the U matrix from the SVD
p=[z(:,2) z(:,1)];

%%
for j=1:2
    subplot(1,2,j)
    imagesc(reshape(U(:,j),120,160)); colormap(gray); axis equal; axis off
end

figure(2)
%The two pictures in this figure are created from the encoding of the
%autoencoder with 2 nodes in the hidden layer.
for j=1:2
    subplot(1,2,j)
    imagesc(reshape(p(:,j),120,160)); colormap(gray); axis equal; axis off
end
%notice they are very similiar but come from two very different places.

%%
theta=subspace(U(:,1:2),p(:,1:2))
norm(X1mean-U(:,1:2)*U(:,1:2)'*X1mean)
norm(Xmean-predict(autoenc,Xmean))
U(:,1:2)'*p(:,1:2)
```

7 Sparse Auto-Encoder

```
load author
X = double(Y1);
Ymean=mean(X,2);
Xmean=X-repmat(Ymean,1,109);
Xmean=Xmean/30000;
autoenc1= trainAutoencoder(Xmean, 2, ...
 'L2WeightRegularization',0.001,...
 'SparsityRegularization',1,...
 'SparsityProportion',0.05,...
 'ScaleData',false);

z=encode(autoenc1,Xmean);
plot(z(1,:),z(2,:))
```

```
%%
z2=predict(autoenc1,Xmean);
for j=1:30
    subplot(6,5,j)
    imagesc(reshape(z2(:,j),120,160)); colormap(gray); axis equal; axis off
end
```

7.1 Unknotting the Torus Knot Problem

```
t=linspace(0,2.*pi);
x=cos(3.*t).*(3+cos(4.*t));
y=sin(3.*t).*(3+cos(4.*t));
z=sin(4.*t);
X=[x; y; z];
figure(1)
plot3(x,y,z)
Ymean=mean(X,2);
Xmean=X-repmat(Ymean,1,100);

%%
autoenc1= trainAutoencoder(Xmean,12,...
        'EncoderTransferFunction','logsig',...
        'DecoderTransferFunction','purelin',...
        'MaxEpochs',1000,...
        'L2WeightRegularization',0.01,...
        'SparsityRegularization',4,...
        'SparsityProportion',0.1);
%%
Y=encode(autoenc1,Xmean);
autoenc2= trainAutoencoder(Y,2,...
        'EncoderTransferFunction','satlin',...
        'DecoderTransferFunction','purelin',...
        'MaxEpochs',1000,...
        'L2WeightRegularization',0.01,...
        'SparsityRegularization',4,...
        'SparsityProportion',.1);
%%
Y1=encode(autoenc2,Y);
autoenc3= trainAutoencoder(Y1,12,...
        'EncoderTransferFunction','satlin',...
        'DecoderTransferFunction','purelin',...
        'MaxEpochs',1000,...
        'L2WeightRegularization',0.01,...
        'SparsityRegularization',4,...
        'SparsityProportion',.1);
%%
Y2=encode(autoenc3,Y1);
autoenc4= trainAutoencoder(Y2,3,...
        'EncoderTransferFunction','satlin',...
        'DecoderTransferFunction','purelin',...
        'MaxEpochs',1000,...
        'L2WeightRegularization',0.01,...
        'SparsityRegularization',4,...
        'SparsityProportion',.1);

%%
Y3=encode(autoenc4,Y2);
m=Y3(1,:);
n=Y3(2,:);
```

```
o=Y3(3,:);
figure(2)
plot3(m,n,o)
```

7.2 MNIST Data set code for Matlab

```
%%
%The function readMNIST organizes the files into there 20x20 matrices with
%the corresponding label
[imgs,label]=readMNIST('train-images.idx3-ubyte','train-labels.idx1-ubyte',60000,0);
images = loadMNISTImages('train-images.idx3-ubyte');
labels = loadMNISTLabels('train-labels.idx1-ubyte');
Ymean=mean(images,2);
Xmean=images-repmat(Ymean,1,60000);
%%
image=reshape(images, 28,28,60000);
%%
imagess=images(:,1:55000);
imagess=imagess/2;
labels=labels'
labelss=labels(:,1:55000);
timagess=images(:,55001:60000);
timagess=timagess/5;
tlabelss=labels(:,55001:60000);
%%
figure(1)
for j=1:4
    subplot(2,2,j)
    imagesc(reshape(images(:,j),28,28)); colormap(gray); axis equal; axis off
end
%%
image=images(:,1:5000);
xmeantest=Xmean(:,1:4000);
%%
autoenc= trainAutoencoder(images,1000,...
        'EncoderTransferFunction','satlin',...
        'DecoderTransferFunction','purelin',...
        'MaxEpochs',200,...
        'L2WeightRegularization',0.001,...
        'SparsityRegularization',4,...
        'SparsityProportion',.15);
%%
z=encode(autoenc, xmeantest');
z=z';
for j=1:30
    subplot(6,5,j)
    imagesc(reshape(z(:,j),28,28)); colormap(gray); axis equal; axis off
end

%%
mnist=predict(autoenc,imagetest);
for j=1:10
    subplot(2,5,j)
    imagesc(reshape(mnist(:,j),28,28)); colormap(gray); axis equal; axis off
end


%%
hiddenSize1 = 100;
autoenc1 = trainAutoencoder(imagess,hiddenSize1, ...
```

```
    'MaxEpochs',400, ...
    'L2WeightRegularization',0.00005, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.15, ...
    'ScaleData', false);
feat1 = encode(autoenc1,imagess);
hiddenSize2 = 50;

autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...
    'MaxEpochs',100, ...
    'L2WeightRegularization',0.0002, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.1, ...
    'ScaleData', false);
feat2 = encode(autoenc2,feat1);
hiddenSize2 = 50;
t1=eye(10);
ttrain=t1(:,labelss+1);

softnet = trainSoftmaxLayer(feat2,ttrain,'MaxEpochs',400);
deepnet = stack(autoenc1,autoenc2, softnet);
%%
y = deepnet(Ttimagess);
t=eye(10);
tttrain=t1(:,Ttlabelss+1);
plotconfusion(tttrain,y);
%%

y = deepnet(timagess);
%%
plotconfusion(tlabelss,y);
```

7.3 Negatives for MNIST Dataset

```
%%
%The function readMNIST organizes the files into there 20x20 matrices with
%the corresponding label
[imgs,label]=readMNIST('train-images.idx3-ubyte','train-labels.idx1-ubyte',60000,0);
images = loadMNISTImages('train-images.idx3-ubyte');
labels = loadMNISTLabels('train-labels.idx1-ubyte');
Ymean=mean(images,2);
Xmean=images-repmat(Ymean,1,60000);
Xmeanneg=-1*Xmean;
%%
image=reshape(images, 28,28,60000);
%%
imagess=Xmean(:,1:25000);
imagess=imagess/2;
labels=labels';
labelss=labels(:,1:25000);
timagess=Xmean(:,55001:60000);
timagess=timagess/2;
tlabelss=labels(:,55001:60000);

nimagess=Xmeanneg(:,1:25000);
nimagess=nimagess/2;
nlabelss=labels(:,1:25000);
ntimagess=Xmeanneg(:,55001:60000);
ntimagess=ntimagess/2;
ntlabelss=labels(:,55001:60000);
```

```
Timagess=[imagess nimagess];
Ttimagess=[timagess ntimagess];
Tlabelss= [labelss nlabelss];
Ttlabelss=[tlabelss ntlabelss];
%%
figure(1)
for j=1:4
    subplot(2,2,j)
    imagesc(reshape(Xmeanneg(:,j),28,28)); colormap(gray); axis equal; axis off
end
%%
image=images(:,1:5000);
xmeantest=Xmean(:,1:4000);
%%
autoenc= trainAutoencoder(images,1000,...
        'EncoderTransferFunction','satlin',...
        'DecoderTransferFunction','purelin',...
        'MaxEpochs',200,...
        'L2WeightRegularization',0.001,...
        'SparsityRegularization',4,...
        'SparsityProportion',.15);
%%
z=encode(autoenc, xmeantest');
z=z';
for j=1:30
    subplot(6,5,j)
    imagesc(reshape(z(:,j),28,28)); colormap(gray); axis equal; axis off
end

%%
for j=1:10
    subplot(2,5,j)
    imagesc(reshape(Xmean(:,j),28,28)); colormap(gray); axis equal; axis off
end


%%
hiddenSize1 = 100;
autoenc1 = trainAutoencoder(Timagess,hiddenSize1, ...
    'MaxEpochs',400, ...
    'L2WeightRegularization',0.0001, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.15, ...
    'ScaleData', false);
feat1 = encode(autoenc1,Timagess);
hiddenSize2 = 50;

autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...
    'MaxEpochs',100, ...
    'L2WeightRegularization',0.002, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.1, ...
    'ScaleData', false);
feat2 = encode(autoenc2,feat1);
hiddenSize2 = 50;

t1=eye(10);
ttrain=t1(:,Tlabelss+1);
```

```
softnet = trainSoftmaxLayer(feat2,ttrain,'MaxEpochs',400);
deepnet = stack(autoenc1,autoenc2, softnet);

y = deepnet(Ttimagess);
%%
t=eye(10);
tttrain=t1(:,Ttlabelss+1);
plotconfusion(tttrain,y);
```

7.4 Cats and Dogs

```
% Read files file1.txt through file20.txt, mat1.mat through mat20.mat
% and image1.jpg through image20.jpg.  Files are in the current directory.
targetSize = [200 200];
folder = cd;
j=[]
for k = 0:500
  jpgFilename  = sprintf('dog.%d.jpg', k);
  imageData    = imread(jpgFilename);
  cat1_resized = (imresize(imageData, targetSize));
  c1=reshape(double(cat1_resized(:,:,1)),200*200,1) ;
  j=[j c1];
end
%%
for k = 0:500
  jpgFilename  = sprintf('cat.%d.jpg', k);
  imageData    = imread(jpgFilename);
  cat1_resized = (imresize(imageData, targetSize));
  c1=reshape(double(cat1_resized(:,:,1)),40000,1) ;
  j=[j c1];
end
%%
net=patternnet(10);
T=[ones(1,501) zeros(1,501)];
[net,tr] = train(net,j,T);
testX = j(:,tr.testInd);
testT = T(:,tr.testInd);
testY = net(testX);
testIndices = vec2ind(testY)
%%
figure, plotconfusion(testT,testY)
%% First autoencoder
hiddenSize = 15;
autoenc1 = trainAutoencoder(j,hiddenSize,...
 'L2WeightRegularization',0.001,...
 'SparsityRegularization',4,...
 'SparsityProportion',0.05,...
 'DecoderTransferFunction','purelin');
% Extract the "features" in the hidden layer
features1 = encode(autoenc1,j);
view(autoenc1);
%% Second Autoencoder
hiddenSize2 = 4;
autoenc2 = trainAutoencoder(features1,hiddenSize2,...
 'L2WeightRegularization',0.001,...
 'SparsityRegularization',4,...
 'SparsityProportion',0.05,...
 'DecoderTransferFunction','purelin',...
 'ScaleData',false);
% Extract the features on the hidden layer
```

```
features2 = encode(autoenc2,features1);
view(autoenc2);
%% The Softmax Layer
%Train a softmax layer:
softnet = trainSoftmaxLayer(features2,T,'LossFunction','crossentropy');
view(softnet);
n=sim(softnet,features2) %kinda cool to see the outputs from the three back to back networks
%% The deep net
% Stack the encoders together to get the deep network:
deepnet = stack(autoenc1,autoenc2,softnet);
[deepnet,tr]=train(deepnet,j,T);
photos=deepnet(j);
%% This is the confusion matrix for the deep net
plotconfusion(T,photos)
```