|  |
| --- |
| Franciscan University of Steubenville |
| Backpropagation in Artificial Neural Networks |
| Senior Thesis |

|  |
| --- |
| James Sarlo  5-9-2018 |

Dedication

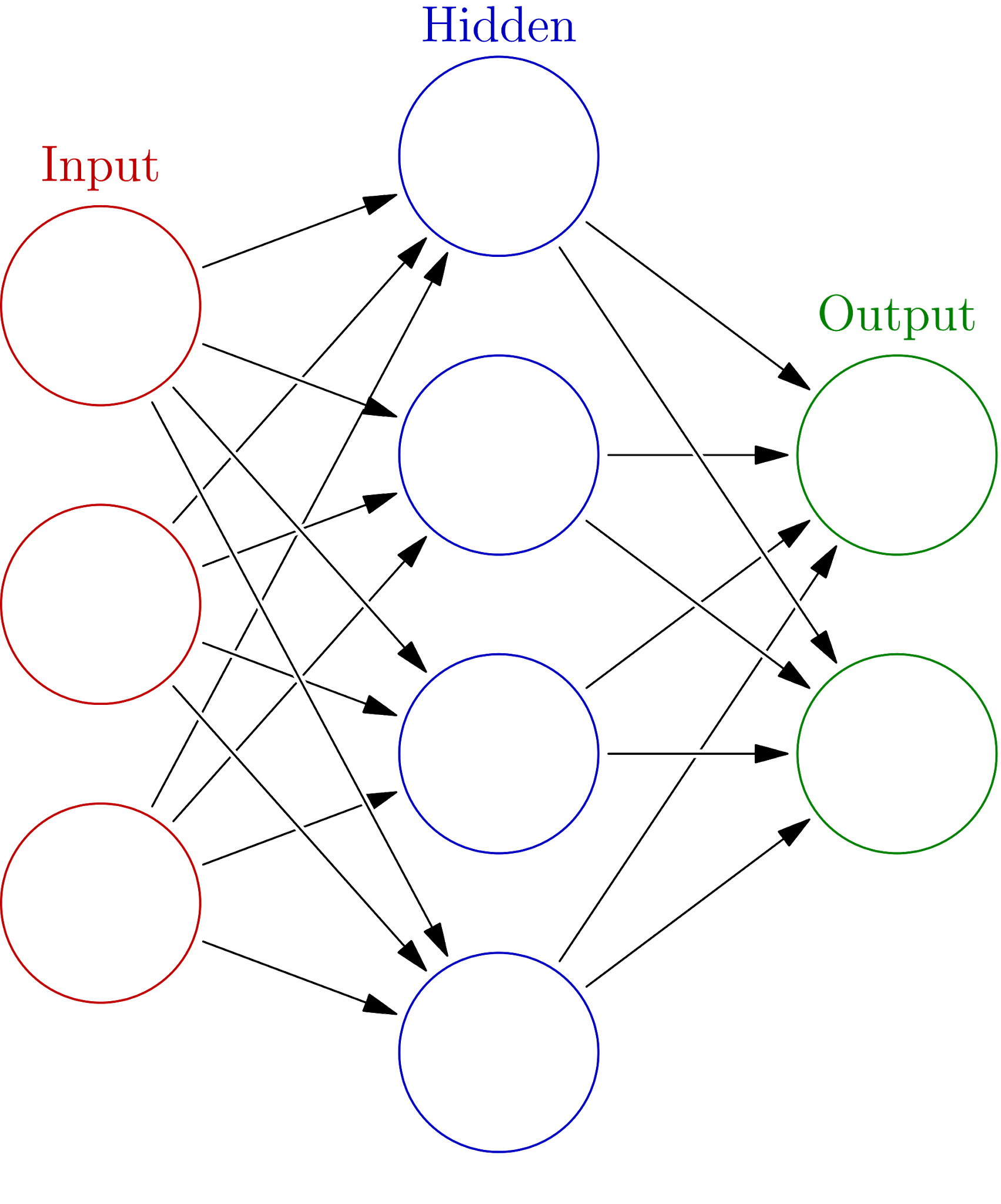
*For Jennifer (1948-2017) and Josh (1996-2017)*

“In theory there is no difference between theory and practice. In practice there is” – Yogi Berra

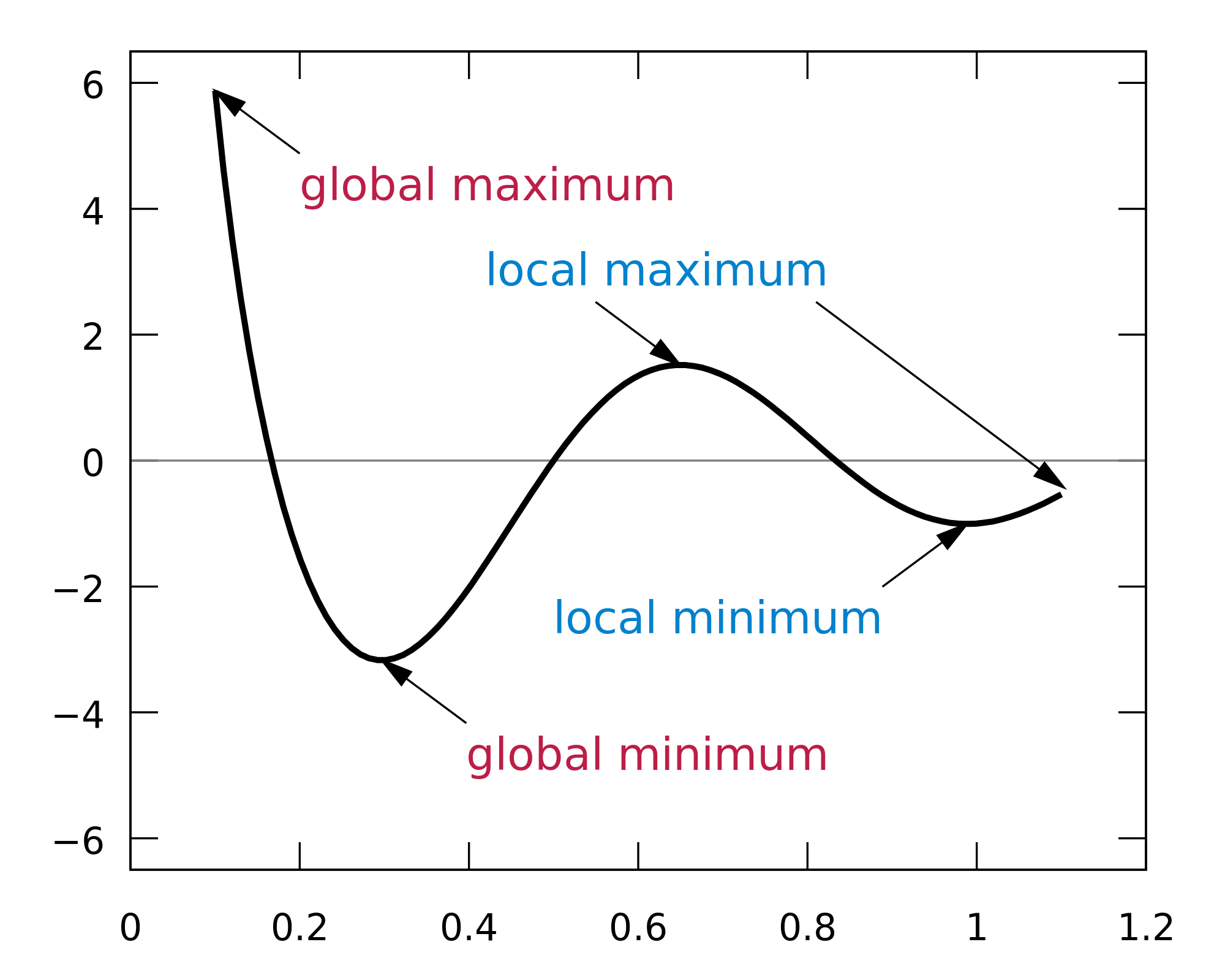
Artificial Intelligence and machine learning are two big buzzwords in modern day technology and society, and for good reason. Artificial intelligence concerns have ranged from how we can solve problems that humans have been previously relegated to, to the control problem, or whether they will take over the world. In this paper, I will discuss the background necessary to understand neural networks, networks themselves, and specifically the backpropagation algorithm.

There are too many fascinating examples of applications of neural networks to list. A widely recognized example is image processing, which has unlimited variations and subtleties. A simple example common with beginners is number recognition: give a computer a handwritten digit and it will output a number from 0 to 9 telling you what it interpreted it as. Image recognition is also used in a wide variety of fields from identifying gravitational lensing to face recognition. Another example of the power of neural networks is Spotify’s music recommendations service; neural networks can be added to a myriad of ways that Spotify recommends music to users (Dieleman, 2014). Other applications include identifying cancer cells, image search engines, and analyzing financial trends.

1. Neural Networks: An Overview

A neural network is a type of computer learning method that is modeled after our understanding of the human brain (Jones 2005). The network is based on a collection of connected units or nodes known as artificial neurons that take input to produce an output. The neurons transmit signal from one neuron to the next layer (Ripley, 2009). At its simplest, a neural network has three layers: an input layer, a hidden layer, and an output layer. Each neuron takes information that is ‘fed’ to it, changes its activation state depending on a predefined activation function, and then produce output depending on the input and activation (Ripley, 2009). The input layer has no precursor while the output layer has no successor. The hidden layer(s) have both predecessors and successors. Most neural networks have multiple layers; networks with more than one hidden layer are known as deep neural networks . Each neuron has a weight that increases or decreases the influence of a specific neuron (Mazur, 2015). Adjusting the weights of the neural network is the main adjustment the backpropagation algorithm makes to train the network. The most common activation function in neural networks is the sigmoid function, which essentially takes an input and gives the equivalent number for the input between 0 and 1. However, a shift to be mentioned later in recent years has caused the rectifier function to be used more commonly (Ramachandran, Zoph, & Le, 2017). A rectifier is a class of activation function that is defined as the positive part of its argument: where x is the neuron’s input. One way to implement a rectifier is to use an approximation of the analytic function: . The derivative of this function is the logistic function, (Ramachandran, Zoph, & Le, 2017). There are several variants of rectifiers that can be used in different situations; for example, a Noisy ReLU has been more efficient at solving computer vision related problems than the softplus function mentioned above.

1. Training Neural Networks: Considerations and Issues

Training neural networks is essentially how neural networks learn. First, the programmer must decide what the goal of the neural network is. Is it a binary network that when you show it a picture, it tells you if it is a dog or not? Or is it classification based, where you show it a picture and it tells you what it is based on a list of preselected possibilities? In the training phase, the correct answer for the information fed to the network is already known; this technique is called supervised learning (Ripley, 2009). We feed the network values that we already know the answer to, check how the network did, and adjust the weights based on how far off the network was. The delta rule is a measure of how far off each node was from the correct value; the error terms are used to adjust the network. Typically, a training data set is used for this purpose before letting the neural network see a real data set (MathWorks, n.d.). Although there are multiple methods for training neural networks, the backpropagation algorithm is the most common. However, all the training methods involve the adjustment of the weights of the network. The backpropagation algorithm is a method in neural networks used to adjust the weights of the network (Jones, 2003). Before talking about anything else, the loss function must be introduced. Having applications in many different fields, the loss function maps the price for inaccuracy. The loss function is what is used to compare desired output to actual output. Two assumptions are made about the loss function: the first is that the average error rate can be written as a summation where x is the individual training examples and is the error function, and n is the number of individual training examples. This allows us to generalize the error function for a single training example over a whole data set. The second assumption is that the error function can be expressed as a function of outputs from the neural network. The standard choice for the error function is the Euclidean distance of . An important note is that the exponent and ½ term will cancel out when the function is differentiated. Combining the two we get . The partial derivative with respect to the outputs becomes . Keep in mind that y represents the actual outputs while y’ is the expected outputs.

Gradient descent is the method that is used to find the minimum of the function; in this situation, we are trying to find the local minimum of the error function (Sanderson, 2017). It is not guaranteed to find the global minimum of the function. A special type of gradient descent is used to find the minimum of the function, batch gradient descent. What is the difference between batch gradient descent and regular gradient descent? Stochastic gradient descent takes random fractions of the training data, calculates the average error, then adjusts the weights accordingly. The first step is to randomly scramble the dataset, which prevents the network from “memorizing” the training examples, so to speak which we will talk about more soon (MathWorks, n.d.). The second is to find an average error rate and adjust the weights using that error rate (Mazur, 2015). This is much faster than regular gradient descent which adjusts the weights for each training example in each layer individually. It is computationally very expensive for any decent-sized data set. Batch gradient descent is more efficient (Botton & Bousquet, 2008). The weights are updated for each training batch until a satisfactory error threshold is reached.

One of the main considerations when it comes to training neural networks is the problem of overfitting: on a training set a network may have a very low error rate but on a new data set the error rate is noticeably higher (MathWorks, n.d.). One way to improve performance of a network is to make multiple copies of the same network, adjusting the initial weights and biases of the networks. We then train them on the same data sets. We then train them on the validation set and compare performances. Another way to avoid overfitting is early stopping (Ripley, 2009). Generally, when (Schwartz, 2016) the error on the validation set rises above a user-defined threshold, the training is stopped, though no satisfactory rule for all cases has been discovered.. Other methods involve dividing the training data: the user can feed inputs randomly or cycle through training, validation and test sets. These methods allow us to generalize out the use of a neural network to allow for the best possible performance (MathWorks, n.d.).

Another consideration of neural network training is the role of the hidden layer. Early neural networks did not use hidden layers to process information. The output of the network was simply a function of the input. This forces the level of the input to be limited. However, with a hidden layer, a multiple neurons’ function can be composed with another to generate more complexity (Jones, 2003). Thus, more hidden layers allow for more complex data processing. The next natural question is how many hidden layers are needed? And how many neurons per layer? The answer is dependent on the unknown function that our network needs to approximate. One method is to set a weight of the network to 0 or near zero during training, which negates the influence of the neuron before finding an accurate combination of weights. This method is known as pruning the network (Ripley, 2009). The next method is to choose the number of hidden units: both layers and neurons. The principle is similar to choosing the number of regressors in a linear regression model (Ripley, 2009). Another proposed method is incremental network construction, which seeks to grow the network. Hidden neurons are added one at a time, either in their own layers or in existing hidden layers. There are different algorithms for accomplishing this, including the pyramid method, the SNC method (Moody & Utans 1995), cascade correlation (Fahlman & Lebiere, 1990). In general, however, a trial and error method is typically the best approach, as each problem facing neural networks varies greatly. (Utans & Moody, 1995)

Two more related problems are the disappearing (or vanishing) gradient and the exploding gradient (Grosse, 2017). The disappearing gradient is when the gradient of the error function is too small to warrant a change in the value of the weight. The opposite problem is the exploding gradient: when we repeatedly multiply gradients within layers of the network that are too large the gradient will explode (Grosse, 2017). Modern computing power has advanced exponentially from the point where these problems were first discovered which reduces the likelihood but they have not disappeared (Grosse, 2017). These problems are solved in many of the same ways: in 2018, the most common neural network activation function is a rectifier, which reduces the probability of the problems occurring.

.

1. The Derivation of the Full Algorithm

Earlier we looked at the derivation of the loss function; now we will look at the full backpropagation algorithm in detail and its mathematical derivation. The intuition of the algorithm is that each of the weights of the network has an affect on the total error of the network. First, we need to find the total net input for the network. For a network with x hidden layers are y input, hidden and output neurons and a bias neuron, the net input for a neuron is (Mazur, 2015). This is then plugged into our activation function; in this case we use the sigmoid function. . The output is then passed to the next hidden layer or output layer. This process is then repeated for each of the hidden layer neurons. The next step is to calculate the error rate for each of the output neurons. We use a summation of the Euclidian distance formula mentioned earlier, . This is performed for each of the output neurons then added together to get total error (Mazur, 2015).

Next is the backwards walkthrough. As stated earlier, we must consider how much each weight affects the total error, which is done by taking the partial derivative with respect to that weight (Mazur, 2015). is the total error with respect to weight . Applying the chain rule gives us . Now we can expand each part of the equation. First is the total error with respect to the output neurons. For a network with two neurons, becomes that simplifies to ) (Mazur, 2015). Next is the output of the neuron with respect to the net output. Recall that the output of a neuron is given by calculating the sigmoid function. The derivative is equal to . Finally, the net output is affected by each weight (Mazur, 2015). which becomes . This process is repeated for each of the output weights and layers. Now on to the weight updates. For where eta is the learning rate of the network and is the new weight for weight n. however, we don’t update the weights until after we walk through each neuron in the hidden layer (Mazur, 2015). For a hidden layer neuron that affects n output layers, we add up the total error from each hidden neuron . Starting with we get . while is equal to the weight term between the hidden and output layer. This is repeated for each of the hidden neurons in layer order (Mazur, 2015). If a network has multiple hidden layers, this process would be repeated for each of the connections between hidden layers as well. Next we calculate the weight update for the hidden layer neurons. This becomes . Next we calculate the effect the weight of the previous layer’s neuron has on the neuron . Finally we calculate the new weight and update it: . As one could see, the error rate will not change very much during one iteration of the algorithm, but after many thousands of training examples and walkthroughs, the error rate will plummet (Mazur, 2015).

1. Backpropagation: Criticisms, Applications and the Future

Neural network applications are simply limitless. What does this mean for the future of artificial intelligence? It means that more efficient algorithms for training neural networks need to be created (Rojas, 1996). One such algorithm, fastprop, is seeing usage. Fastprop is based around the instantaneous relationship between weights and outputs *as the network is being propagated*. Although still in experimental stages, tests have been promising (Rojas, 1996). Of course, it is possible to use other algorithms for training neural networks; backpropagation is generally agreed, however, to be the best class of algorithms to train neural networks.

However, backpropagation is not without its faults. One glaringly obvious problem is that backpropagation requires a large amount of training data. This training data also must be labeled: this combination presents a tedious problem for interns and grad students alike. Annotating these images is incredibly time consuming. This problem can be solved by using what’s known as unsupervised learning (Ripley, 2009). Other criticisms of the backpropagation algorithm include sheer amount of computational power needed to perform these some of the problems. It may not be the most efficient use of processing power depending on the situation.

Interesting problems in the future of neural networks are limitless. Google tore out the guts of its most important features: image search to voice recognition, recreating them from the ground up with machine learning (Schwartz, 2016). Computer vision is an incredibly important field that will have implications from self-flying planes to Snapchat filters to allowing the blind to see. The possibilities are endless but remember that neural networks are just a tool, not the end-all be-all of artificial intelligence. Similarly, backpropagation is not the only way to train neural networks. One should not plan on throwing algorithms at a problem until it works but consider each problem carefully. Neural networks are just a tool in the toolbox of programmers.

1. The Algorithm in C++

The backpropagation algorithm can be implemented in a number of ways in a variety of programming languages. For a first artificial intelligence program, I used C++ with a simple program to demonstrate a network. It is more of a proof of concept than a working network. The program using stochastic gradient descent rather than batch gradient descent to demonstrate the use of the algorithm. Unfortunately, I was not able to get the network to accept training data beyond a trivial matrix of numbers. I would like to be able to use a network for a more interesting task such as image recognition. This network is adapted from a tutorial (Miller, 2013).

The structure of the network, however, is the important thing. The program is comprised of two classes: Net and Neuron. The Net class defines the structure of the network. It has the functions feedForward, backprop, and getResults. These functions perform the functions feeding data forward, backpropagation, and getting results. By using a for loop, the program loops from through the training data and inserts them into the input layer. Then, the network propagates through by referencing each neuron in the previous layer. It also takes into account the bias neuron. This function is recursive. The next function is the backprop function. First, we sum up the target values minus the expected values for each output layer neuron. The residual mean squared error is calculated, then the overall net error is calculated. Finally, we loop through each layer, starting with the output then the hidden layers to calculate their gradients. The last function is getResults, which simply loops through the output layer and prints the output value.

The Neuron class defines the mathematical operations needed to run the network. It consists of seven functions: setOutputVal, getOutputVal, feedForward, calcOutputGradients, calcHiddenGradients, updateInputWeights, transferFunction, transferFunctionDerivative, sumDOW, and updateInputWeights. setOutputVal and getOutputVal are both trivial setters and getters. calcOutputGradients takes the target value minus the output value for the output neuron and calculates the gradient by multiplying the delta value and the derivative of the transfer function for the output value. calcHiddenGradients does the same for the hidden layers. sumDOW is the total contribution of errors for each of the neurons. updateInputWeights calculates the new delta weight by looping through the input layer, multiplying the learning rate term by the output value of the previous layer, the gradient, adding momentum and the old delta weight. transferFunction in this network is the hyperbolic tangent function, tanh(x). transferFunctionDerivative is the function used to calculate the gradient; the derivative of which is .

This is just one example of the many ways that a neural network can be implemented in code. A direction that I would like to go is to be able to run more interesting training data through the network; once I can accomplish that I’d like to move on to more and more complex problems, making adjustments as necessary. I would also like to try implementations in different languages such as C# and Python, as they are both more mathematically inclined than C++. Another interesting direction is to use the TensorFlow library, a prebuilt machine learning framework that implements neural networks.

# Bibliography

Botton, L., & Bousquet, O. (2008). The Tradeoffs of Large Scale Learning. *Advances in Neural Information Processing Systems* (pp. 161-168). Vancouver: NIPS Foundation.

Dieleman, S. (2014, August 5). *Recommending music on Spotify with deep learning*. From Sander Dieleman: http://benanne.github.io/2014/08/05/spotify-cnns.html

Fahlman, & Liebere. (190). The cascade correlation learning architecture. *NIPS2*, 524-532.

Grosse, R. (2017, March 7th). *L 15 Exploding and Vanishing Gradients.* From University of Toronto Computer Science: http://www.cs.toronto.edu/~rgrosse/courses/csc321\_2017/readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf

Jones, M. T. (2003). *AI Application Programming.* Hingham: Charles River Media.

Liao, Y. (n.d.). *Neural Networks in Hardware: a Survey.* From Louisiana State University Division of Computer Science and Engineering: http://bit.csc.lsu.edu/~jianhua/shiv2.pdf

MathWorks. (n.d.). *Improve Neural Network Generalization and Avoid Overfitting*. From MathWorks: https://www.mathworks.com/help/nnet/ug/improve-neural-network-generalization-and-avoid-overfitting.html

Mazur, M. (2015, March 17). *A Step by Step Backpropagation Example*. From Matt Mazur: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Miller, M. (2013, March 9th). *Neural Net in C++ Tutorial on Vimeo*. From YouTube: https://www.youtube.com/watch?v=KkwX7FkLfug

Ramachandran, P., Zoph, B., & Le, Q. V. (2017, October 27th). *1710.05941 Searching for Activation Functions.* From Cornell University Library: https://arxiv.org/abs/1710.05941

Ripley, B. D. (2009). *Pattern Recognition and Neural Networks.* Cambridge: Cambridge University Press.

Rojas, R. (1996). *Neural Networks: A Systematic Introduction.* Springer-Verlag: Berlin.

Sanderson, G. (2017, October 16). *Gradient Descent, how neural networks learn | Chapter 2, deep learning*. From YouTube: https://www.youtube.com/watch?v=IHZwWFHWa-w

Schwartz, B. (2016, October 18). *How Google uses machine learning in its search algorithms*. From Search Engine Land: https://searchengineland.com/google-uses-machine-learning-search-algorithms-261158

Utans, & Moody, J. (1995). *Architecture selection strategies for neural networks: application to corporate bond rating predictionn.* Chichester: Wiley.