Practicum

Intro to Artificial Intelligence

Jackson Britt

12/13/2018

Introduction:

For this practicum, I decided to look at the problem of statistical classification. In this class we have covered one common method for classification, which is using Bayesian networks to predict the classification using the probabilities of the class given the features of the item to be classified. However, Bayesian networks assume that the features are independent of one another, which is not necessarily true. An alternative to this method is to use a neural network, which uses weights and a magical hidden layer to solve classification problems. Well it is not really magic, but uses a variety of activation functions and learning algorithms we will cover.

Methods:

- *Bayesian Network:*

The first method mentioned was a Bayesian network, or a Bayes Net, is a graphical model to predict how a simplified version of the world works. The a Bayes net graph is always acyclic and directed. Unsurprisingly, Bayes nets are built upon Bayes' theorem, which uses prior knowledge in order to calculate the probability of an event. If we have multiple features, which then correlate with some variables, which then feed into more variables, we can calculate the probability of of the the features or events, depending on whether the features that correlate with them happened or not. In the problem of classification, we use what is called a Naïve Bayes net. In a Naïve Bayes net, we assume that every feature has a random probability. Then we add a feature vector for each variable, which is what features correlate with a variable. Then, using the Naïve Bayes net with a set of feature vectors, we can calculate the probability of the variable with an actual set of features. However, this still assumes that the initial features are independent effects, while this could very well not be the case. Often times, the occurrence of two feature alone will not indicate an outcome, but together they would. (Soni)

- *Artificial Neural Network:*

An alternative to Bayesian networks for classification problems is an artificial neural network, which this project will cover. These are modeled after actual neural networks that exist biologically in our brains. Biological neural networks are comprised of a network of neural cells that send 'messages' through our brains. In a biological neural cell, the cell body receives 'messages' through multiple dendrites, then sends the 'message' onto the neurons it is connected to.

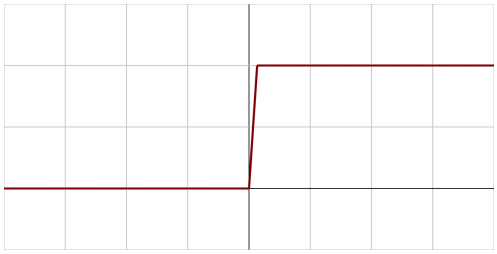
While we do not fully understand how these biological network work, we have been able to model them into an artificial network used for machine learning. These neural networks, like Bayesian networks, are acyclic directed graphs. A neural network is composed of four main parts: a layer of input features, a layer (or series of layers) of hidden

nodes receiving the input and transmitting to the output, a layer of output nodes, and the weights of the connections between the nodes in each layer. Each hidden layer contains what is called an *activation function*. We will go into more depth about these later, but a simple example for now is a step function used in logic gates. The function returns either 0 or 1, depending on the value passed into it. Supervised neural networks, unlike Bayesian networks, learn from data. But how can a machine learn? Well the network is not actually a machine, but rather a learning algorithm or series of learning algorithms.

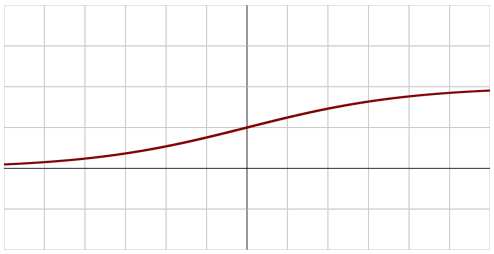
Learning in the neural network takes place within the weights of the connections between the nodes. To begin, we assign a random value to each weight. Then, inputting the data of features with known outcomes, we begin a process called *forward propagation*. First, we multiply the inputs with the initial weights. This is done in practice by taking the dot product of the input matrix with the weights matrix. Then, we run the activation function of the data in the hidden layer. If we have multiple hidden layers, we repeat this process for each layer, taking the dot product of the incoming data and the weights of the connection. Finally, we take the dot product of the data coming out of the final hidden layer with the weights going to the output, giving us are resulting prediction. (Loy)

However, it is unlikely that the first product is accurate, and our network must learn from its mistakes. This is done through a much more complicated process called *backpropagation*, in which we take the error in our data and go back correcting the weights. First, we have to take a loss function of the difference between the actual outcome and the expected outcome. Then, we use gradient descent, a process used to find the minimum of a function, in order to minimize loss for the update weights. This involves taking the derivative of the activation function and finding the amount each layer of weights contributed the error. Then for each layer are updated based on the resulting difference. We will cover the specific methods used in more depth in the application section of this project. (Shamdasani)

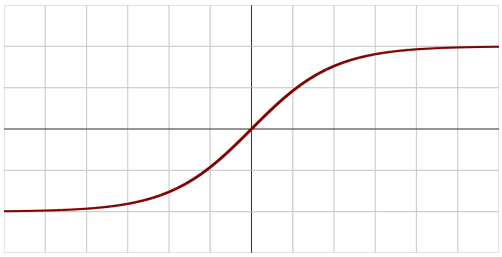
Activation functions vary between neural networks, and multiple different activation functions can be used in one network. The most simple example of an activation function is the *step function*:



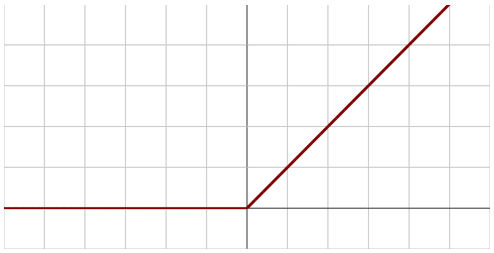
This function allows for immediate, binary results. Another very popular function, which I use in my application, is the sigmoid function and its many varieties. Here is the vanilla *sigmoid function*:



One variation of a sigmoid function is a *tanh function*:



The final example I will show is the *rectified linear unit function* (ReLU):



(V)

These activation functions are used in the hidden layer to calculate the output, and their derivatives are used in backpropagation.

For my implementation, I constructed a neural network with one hidden layer, which uses either a tanh function or regular sigmoid function. I constructed a set of sample data that classifies whether the input is a cat or a dog based on four different features:

Size, independence, loyalty, fluffiness

The first three are fairly separate between cats and dogs; dogs are bigger than cats, cats are more independent than dogs, and dogs are more loyal than cats. However, the ever important attribute of fluffiness has no direct correlation. While all cats are at least pretty fluffy (excluding some very odd breeds primarily owned by mad scientists), dogs can either be supremely fluffy or barely fluffy at all. This will give our neural network a little more difficulty in learning to distinguish the two possible classifications.

Results:

I constructed a neural network class with a single hidden layer from the ground up, using NumPy for assistance with some calculations. I also tested the neural network with two different activation functions. The first was a vanilla sigmoid function, and the second was a tanh function built into NumPy. I compared the error in the output between these two functions for each training iteration, as well as the greatest weight change during backpropagation.

Let us dive a little bit more into the backpropagation method used in the neural network. First we take the difference between what the resulting output actually is, and what the expected output is, giving us our error. Then, since we are taking the derivative of the activation function, we have to apply the chain rule to our equation. We multiple our error by 2, then multiply that by the result of the predicted output run through the derivative of the activation function. In my code, I call this chain_out. We then have to take the dot product of chain_out and the transposition of the results from the activation function of the hidden layer. This convoluted algorithm gives us the new change in weight for the weights connecting the outgoing layer to the output.

Next, we have to find the change in the weights for the weights connecting the input to the hidden layer. We first take the dot product of chain_out and the transposition of the weights connecting the hidden layer to the output. We then multiply this by the result of the activation function of the hidden layer through the derivative of the activation function. I called this chain_in in my implementation. Finally, we take the dot product of the transposition of the incoming data with chain_in. This gives us the change in weights from the input to the hidden layer. If you consider yourself to be a connoisseur of this kind of mathematica, a more in-depth description of a more complicated system can be found at: http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/.

In order to test my neural network in a classification problem, I used a simple version of the cat vs dog classification problem, using the four attributes I listed above. Here is the training data I used for my algorithm:

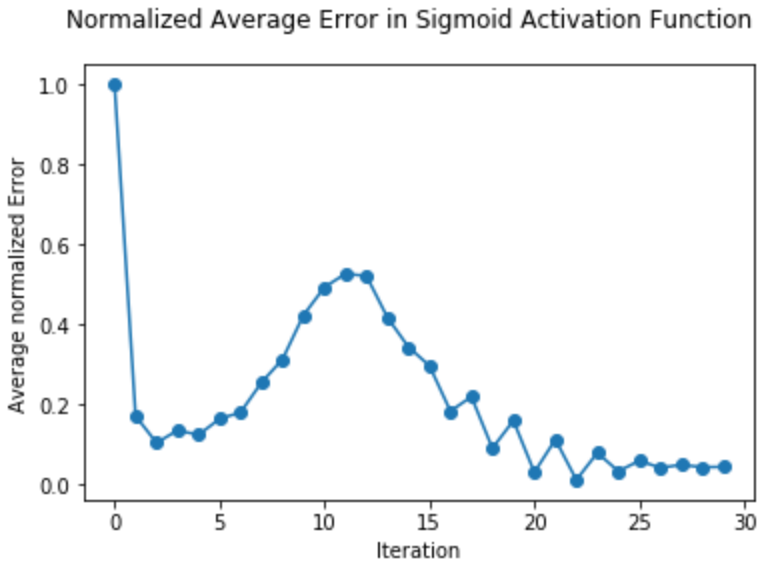
Size	Independence	Loyalty	Fluffiness	Dog or Cat
0.8	0.3	0.9	0.3	Dog
0.4	0.7	0.3	0.7	Cat
0.6	0.4	0.8	0.9	Dog
0.3	0.7	0.2	0.6	Cat

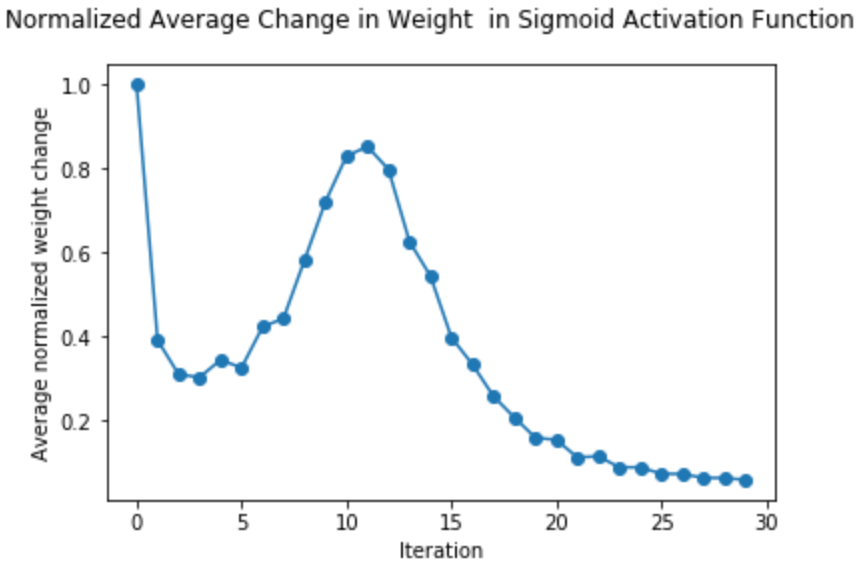
0.1	0.8	0.1	0.6	Cat
1.0	0.2	1.0	0.2	Dog
0.3	0.3	0.6	0.4	Dog
0.4	0.6	0.4	0.8	Cat
0.6	0.2	0.8	0.3	Dog
0.5	0.6	0.6	1.0	Cat

In practice, I used different numerical representations of the dog and cat depending on the activation function. For the vanilla sigmoid function, I used 1 to represent a dog, and 0 to represent a cat. For the tanh function, I used 1 to represent a dog and -1 to represent a cat.

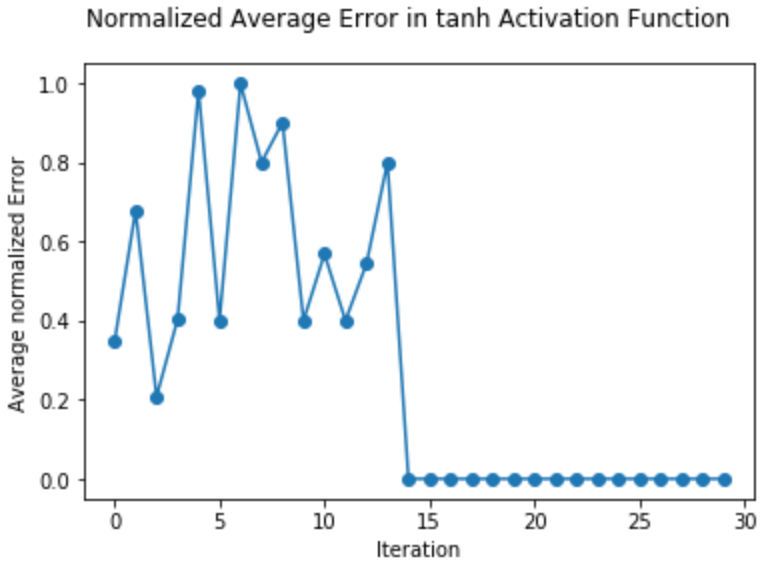
After training the data for 30 training iterations, I took both the normalized average error between the output and the expected result, as well as the normalized average change in weight of the output layer.

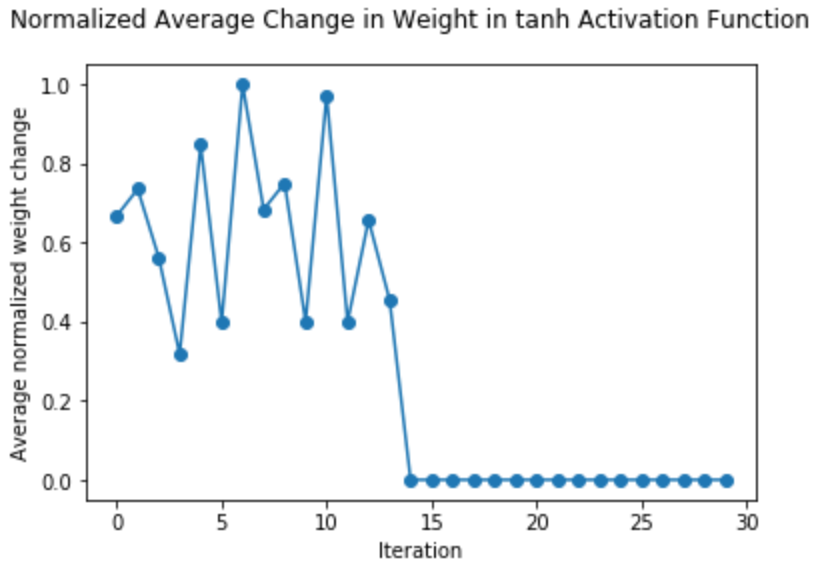
Here is a result for the sigmoid function:





Here is a result for the tanh function:





As we can see, the tanh function seems much more rectionary than the sigmoid function. However, the tanh function also arrives at a minimal loss much quicker that the sigmoid function. When I tested each function to predict if I had fed it a dog or a cat, each provided a correct answer.

Conclusions:

The most difficult aspect of building a neural network was certainly learning and writing the backwards propagation function. However, despite the difficulty in building a neural network from scratch, it was much easier for me than building a Bayesian network. However, I have never built a Naïve Bayes net, which may be simpler for me to understand that a regular Bayes net. I have previously studied neural networks, and I imagined that they it would have been much more difficult to build one, even the very simple one that I have built.

If I were to uses this neural network for bigger, more comprehensive classification problems, I would certainly use the tanh activation function. In almost every trial tested, the tanh function achieved minimal error and weight changes much faster than the sigmoid function. I have also read about the ReLU activation function, and would be excited to see how that may compare to these sigmoid-based activation functions.

In the future, I would like to build more complicated neural networks, that use more than one hidden layer. When we incorporate multiple hidden layers, we can add more than one activation functions to our neural network. I have also read about more complex neural networks that do not have all nodes connecting to every other node in the next layer.(Ng)

I have read about python imports that can make building any neural network much simpler. For example, PyTorch can construct a multilayered neural network, with a variety

of different activation functions, in a single line of code. This would make the processes of building a simple neural network much easier, although I doubt I would understand neural networks as well as I do now if I had taken that approach. (Jain)

References:

1. Jain, Y. (2018, June 27). Create a Neural Network in PyTorch - And Make Your Life Simpler. Retrieved from https://medium.com/coinmonks/create-a-neural-network-in-pytorch-and-make-your-life-simpler-ec5367895199
2. Loy, J. (2018, May 14). How to build your own Neural Network from scratch in Python. Retrieved from https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6
3. Ng, A. *et al.* Multi-Layer Neural Network. Retrieved from http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/
4. Shamdasani, S. Build a Neural Network in Python. Retrieved from https://enlight.nyc/projects/neural-network/
5. V, A. S. (2017, March 30). Understanding Activation Functions in Neural Networks. Retrieved from https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0
6. Soni, D. (2018, June 08). Introduction to Bayesian Networks – Towards Data Science. Retrieved from https://towardsdatascience.com/introduction-to-bayesian-networks-81031eeed94e