Josiah Vincent

CS-445 Machine Learning

FINAL PROJECT: REPORT

NEURAL NETWORKS VS NAÏVE BAYES [MNIST]

ABSTRACT

Neural networks are designed to mimic the human brain where you have multiple layers of neurons and in order for those neurons to fire, they need a positive signal. The goal of a neural network is to train the network/brain when to fire and when not to fire. In doing this we can get accurate predictions about our dataset within an acceptable reason

Naïve Bayes takes a different approach. Instead of progressively "learning" the way the Neural Networks do, it takes a look at the data as a whole and computes more of a probabilistic model. Based on this it is able to predict the outcome of a positive or negative response.

INTRODUCTION

In this project my goal was to compare the results and performance of an implementation of a Neural Network and a Naïve Bayes Model over the MNIST dataset.

The MNIST dataset is a set of 28x28 pictures of handwritten digits collected from addresses on post mail. The digits vary from 0-9. There are 2 sets. The first set is a training set of 60,000 images. The second set is a test set of 10,000 images. Each image record has a label telling us what the image is.

The main idea is to apply train our model (as many times as necessary) over the train data set. Once we have trained our model, we then apply it to the test set and check the performance.

This is going to look different because the implementation of the models is completely different. The main difference at a very high level is that the Neural Network model needs to train multiple times and the Naïve Bayes is more of a one and done and the results are always going to be the same.

This paper will walk through the method of implementation for both methods and will discuss the results. We will talk about the limitations and the benefits of both methods.

Josiah Vincent

CS-445 Machine Learning

The discussion will be limited to the scope of testing on the MNIST set. Results will most likely vary on sets with more or less data in them. Diversity of the data may also play a part.

METHOD

MNIST Initialization

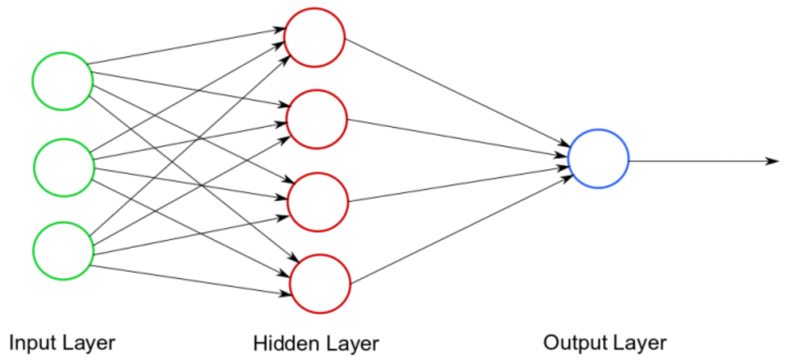
First, we need to understand the implementation of each method.

First, we should understand the dataset and how to get it set up first. We are using a csv format of the MNIST data for both the training and test sets. The pixel values can vary from 0-255. For later purposes we will just normalize all of the pixel values to be 0-1 by just dividing each pixel by 255.

Neural Networks

Let's discuss the Neural Network implementation on the MNIST sets.

In a NN we first need do decide how many hidden layers we want to have. I tried a number of different layer amounts, which are shown in the experiments, but the important part to know is that the number of layers is a fixed number constant and is predetermined at the start of the program. For our purposes we are using a single hidden layer with n hidden nodes also a constant.



The image above is a good example of a neural network. Our input will be 784 pixels and we will have as input nodes (green, left). We then have a single hidden layer with n nodes, 4 shown above (red, center). The final layer is the output layer (blue, right). There will be 10 outputs, one for each handwritten digit. For every layer except the output layer we will also have a bias node. The value of this does not matter as long as it is constant. Usually the bias is 1 or -1.

The black arrows in the image represent the weights for the model. Initially the weights will be randomized uniform between (-0.5) to (0.5). This will be updated as the network is trained, but we will discuss that in more detail later.

Josiah Vincent

CS-445 Machine Learning

First, we need to understand how to evaluate the model. Each input node is connected to all the nodes in the hidden layer and each node in the hidden layer is connected to all of nodes in the output layer. In order to evaluate each hidden node we need to take the dot product of all of the nodes in the input layer with all of the weights that are connected to the node we are evaluating in the hidden layer make sure to add the weight of the bias node. This is all shown in equation (1)

Once we have this value, we need an evaluation function. For this project we are using the sigmoid function for evaluation. This is shown in equation (3).

After we evaluate at the hidden layer the process is exactly the same from the hidden to the output layer and we use the same evaluation function.

$$h_j = \sigma(\sum_{i \epsilon input\ layer} w_{ji} x_i + w_{j0}) \text{ (1)}$$

$$o_k = \sigma(\sum_{j \epsilon hidden\ layer} w_{kj} h_j + w_{k0}) \text{ (2)}$$

$$\sigma = \frac{1}{1+e^{-x}} \text{ (3)}$$

Once we have applied the above 3 equations, we have successfully made a first forward pass. The output that activates is what the model predicts the digit is.

You of course need to check this with the label of the image to check if the model evaluated correctly. You can expect the accuracy to be very low on your first epoch. Reason for this is the weights are completely random, so it is essentially guessing without any knowledge.

From here the only thing to do is update the weights to get closer to the true result. To update the weights, we use a method called backpropagation.

We first need to know how much to update the weights. To do this we need to compute the error. We first calculate the error at the output layer and then at the hidden layer.

$$\delta_k = o_k(1 - o_k)(t_k - o_k) \text{ (4)}$$

Above o_k is the output at the output layer of a node, t_k is the target.

$$\delta_j = h_j(1 - h_j)(\sum_{k \in output\ units} w_{kj}\ \delta_k) \text{ (5)}$$

Above $h_j$is the output at the hidden layer of a node, w_{kj} weight connecting the hidden layer to the output layer.

Once we have the errors, we can now move to updating the weight values. We start by updating the weights connecting the hidden layer to the output layer.

$$w_{kj} \leftarrow w_{kj} + \Delta w_{kj} \text{ (6)}$$

$$\Delta w_{kj} = \eta \delta_k h_j + \alpha \Delta w^{t-1}{}_{kj} \text{ (7)}$$

Here the η is a learning rate. The learning rate helps with oscillation. This is a constant value which will be tested at different values. On the right side of (7) we have a momentum value $\alpha \Delta w^{t-1}{}_{kj}$ where α is a constant and t-1 refers to the previous epoch. So, $\Delta w^{t-1}{}_{kj}$ refers to the change of that particular weight previous epoch.

$$w_{ij} \leftarrow w_{ji} + \Delta w_{ji} \text{ (8)}$$

$$\Delta w_{ji} = \eta \delta_j x_i + \alpha \Delta w^{t-1}{}_{ji} \text{ (9)}$$

We do the exact same thing for the weights that connect the input layer to the hidden layer. This part does not need explanation as it is the same as previously discussed.

After you have done an update you have completed an entire epoch of training. You then $forward\ propagate\ again\ and\ check\ the\ evaluations\ and\ repeat.$

Generally, you predetermine a number of epochs or stop once you reach the desired learning rate or when the accuracy levels out.

Naïve Bayes

Luckily Naïve Bayes is not near as complicated. We need to understand this method before we can compare.

Naïve Bayes focused more on probabilities. To implement this model, we again are using the pixels of the image as our inputs. We need to make a few calculations before we can run the Naïve Bayes equation shown below.

$$N(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \text{ (10)}$$

As you can probably tell this is a Gaussian Naïve Bayes model. Here x is our input. There are a few values we do not know. μ represents the mean of that pixel value in that location of the 28x28 image over all the images. σ represents the standard deviation of that pixel value in that location of the 28x28 image over all the images. So, we need to solve for those 2 before we can apply the above.

Before finding the unknowns, I split the data. One side for the digit that I want and the other for the image that I don't want. This allows me to find a mean and standard deviation for both the correct images that I am looking for and the images that I am not looking for.

To find the mean(s) you can think of each image just being one long row of pixel values.

Image1 = [1,2,3,4]

Image2 = [5,6,7,8]

With the example above to find the mean we want to add the column values together vertically and then divide by the number of images. So, for our example after addition we would have [6,8,10,12]. We have 2 images, so we divide by a scalar of 2. So, our final means are [3,4,5,6]. This is the value we are using for μ.

$$\sigma = \sqrt{\frac{\sum_i (x_i - \mu)^2}{count\ of\ handwritten\ digit}} \quad (11)$$

Next we need to find the standard deviation. Std dev is denoted by σ shown above. We essentially are taking the sum of each pixel minus its mean squared and dividing that by the number of images that have that same handwritten digit. We are also separating this for the number that do and the number that don't. Each will have their own standard deviation values.

At this point we have everything we need to just plug into the Naïve Bayes model (10). At this point our x is going to be an image from our test set not the train set.

For the application of the model I had to loop for each different number and test to see if it was the number in the label. This is why we separated into means and std deviations for is the model and is not the model. This gives us something to compare. We plug x in and our values for the mean and std deviation and get some value back for both "*is and not*". For example, if we plugged in our means and std deviation for the digit 1 we would compare the final values we got back if the "is" value is larger we predict the digit is a 1. If the "not" value is larger we predict it is not a 1.

To check the accuracy each time we get a prediction we check the image label to see if we predicted correctly. We also check to see if we predicted a false positive. You can see below the general idea in equation (12).

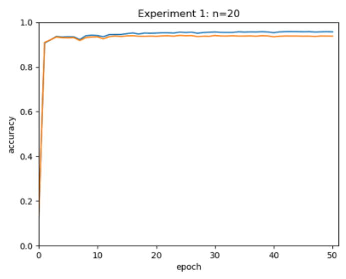
$$accuracy = \frac{predicted\ correct}{predicted\ wrong + predicted\ correct}$$

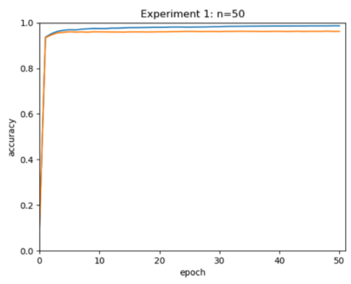
This is essentially it for understanding the implementation of both methods. In the next section we will take a look at some examples and how each model performed over the set.

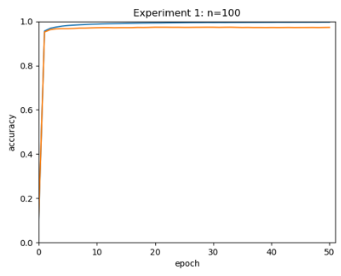
EXAMPLES/RESULTS

Neural Network Results

We will first have a look at the performance of the Neural Network. Immediately you may notice with a Neural Network one of the difficulties is there are lots of knobs to turn to tune the model. You may not always be after the highest accuracy, but instead need to account for time as well. So, we needed to try a lot of different combinations to see what performed well with the NN.

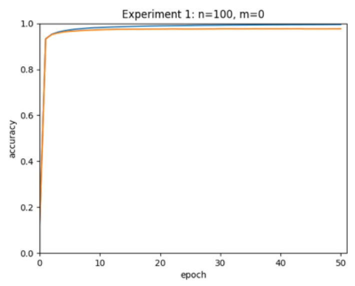


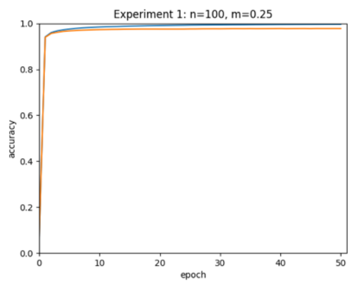


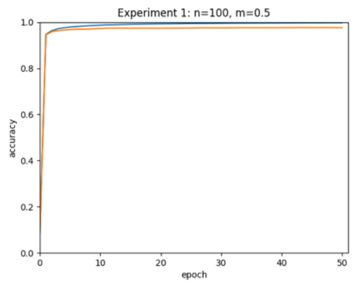


The first knob we tried to turn was the number of hidden neurons in our hidden layer denoted by n in the charts above. You can see for all of the plots we start out at a very low accuracy about 20% or lower. Each experiment was over 50 cycles or epochs. Also note the blue line is for running over the train set and the orange is over the test set. (You can see the higher the value of n the higher the accuracy and the faster it converges. It looks like we were high 90s% within 10 epochs. Accuracy is not always everything. The higher the value for n the longer it takes. When n=20 It took a couple of hours. When n=100 the model took 5-6 hours to run.

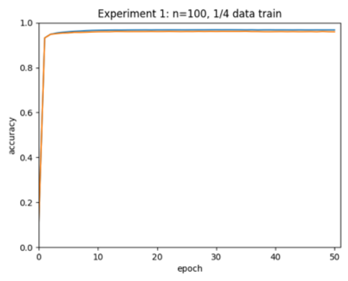
The next knob I played with was the momentum value. This is denoted by m and I fixed the value of n=100 because it had the best accuracy.

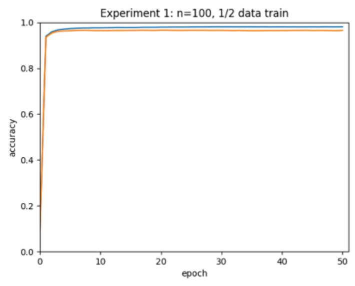






The colors are the same here. You can see from the plots that by increasing the learning rate it converges slightly faster. This doesn't appear to hurt the accuracy at all so I would recommend trying even higher values for m.





Finally, I wanted to see how the model would perform using less data to train over. You can see by the plots that the amount of data plays a huge factor. This was expected.

Naive Bayes Results

Naïve Bayes is much different. Because it does not progressively learn like the Neural Network it is not useful for us to show it on a graph.

In running my method, I had an accuracy for each digit. All I did was take the average accuracy of all of them.

([0.9929, 0], [0.9903, 1], [0.9913, 2], [0.995, 3], [0.9949, 4], [0.9933, 5], [0.9948, 6], [0.9874, 7], 0.9959, 8, 0.9891, 9) = .9925 (average)

You can see the results are really good. The best part was it only took under a minute to run!

The only other way I could think to test this is by limiting the amount of train data like I did for the Neural Network.

At ¼ data here is the accuracies:

([0.9969, 0], [0.9903, 1], [0.9952, 2], [0.999, 3], [0.9919, 4], [0.9989, 5], [0.9948, 6], [0.9893, 7], [1.0, 8], [0.9921, 9] = 0.9948 (average)

This was a little shocking because for digit 8 it got 100%! Overall, I was very impressed with this method and I was expecting the values to be lower since it is a probabilistic model.

Comparison

I did research on both methods and I have played with them outside of what was shown here. I think that the Neural network is going to win over a wider range of things. Here we had a set of images that were all really close in size and I can see how the Naïve Bayes did so well, but in a set where your inputs were not all so close together I think that the NN would have done a better job at learning.

I think where the Naïve Bayes wins out is its speed. I think if you were not as concerned with accuracy and you needed something close that ran much faster. I would choose the Naïve Bayes model for maybe something not quite as critical. If I had less data based on these results, I think I would choose the Naïve Bayes as well. I think the NN performed much poorer with less data. Definitely if I did not have much time, I would run the Naïve Bayes as it ran in just under a minute.

RESOURCES

All the code for this project can be found here. Everything is compiled using python.