

CSC 447 MNIST Classification Project

By: Mark Weiss

I. What is MNIST and why do we want to Classify it?

MNIST (Mixed National Institute of Standards and Technology) is a data set created by a group of researchers based in Redmond, NY. This data set contains tens of thousands of 28x28 grayscale images that, when viewed, look like a regular old handwritten digit, 0 through 9. So why do we want to classify it? Well handwriting is something that we take for granted sometimes. A large majority of people alive today know how to read and write. So we want to create a computer model that takes these 28x28 images and learns how to read with as much accuracy as possible.

II. What does this data look like?

The 28x28 images are stored in a list of 784 numbers between 0 and 255, with 0 meaning black, and 255 meaning white. This data is then reshaped into a 2D array, with 28 columns and 28 rows, mapping this data to a 28x28 cube. The following code prints the image:

```
image = image.reshape(28, 28)
ax.imshow(image, cmap='gray', interpolation="nearest")
```

With the image printing like this:



III. How do we classify this data?

To start in the classification process, we must decide what kind of model to use. First off, we know we want to use some sort of neural network, so we have the three big options of a single layer neural network, a multi-layer perceptron (MLP), and a convolutional neural network (CNN). The CNN is commonly known as the best for image processing, as it has spatial awareness, meaning if all it knows is a dog in the upper left, then if it sees a dog in the lower right, it will still see it as a dog. The MLP however does not have this, that being said, the images only being 28x28 means that the usage of an MLP over a CNN will not have much impact as the data that is analyzed is centralized and doesn't have much space to move around. So, what model do we use? Having used the sklearn package in the past, that is preferable, and looking into their options for neural network, there is only an MLP classifier, and MLP regressor, and a Bernoulli RBM, which we do not have interest in. This leads us to a conclusion of using the MLP classifier from sklearn. While we do lose out on spatial awareness, the data is limited enough that I don't believe it will cause an issue.

IV. We have chosen a classifier; how does it work?

The chosen classifier, a multi-layer perceptron, is a type of feed-forward artificial neural network. What does that mean? It consists of three layers, input, hidden, and output. The input layer is as it sounds, the layer where the 28x28 image is input. The hidden layer is a series of layers that are changed and formed to create the output that the data is fit to. Lastly is the output layer which holds the output of the data formed by the hidden layers. Since this is a feedforward network, the data go linearly. Input => hidden => output. This simplifies the network and increases the speed of the machine. Additionally, the input and output layers utilize an activation function. This function maps the weighted inputs of one layer to the outputs of the next. There are several types of activation functions, but the best one for our purpose is a logistic function. This is because that we are predicting the probability of the image being a certain number, and logistic functions give an output between 0 and 1, essentially creating the percentage that we are looking for.

V. Hyperparameters! What are they and which ones do we use?

Hyperparameters are a number of options you have when creating a neural network. They are options such as the activation function, number of hidden layers, L2 regularization, batch size, and epochs.

The activation function was mentioned previously with the determination of using a logistic function.

The number of hidden layers is the biggest change we will look at with data later. L2 regularization, also known as Alpha, is a term representing how much we want to reduce overfitting. Batch size is the number of images that will be processed before the learning model is updated. Lastly, epochs, is the number of iterations the model will go through before completing its training. Going through every image in the training set completes a single epoch.

So which ones do we use? Well, there is a wonderful tool provided to us by the sklearn package that tells us which ones are the best. This tool is called “GridSearchCV” which takes in options for hyperparameters and runs them all to determine which parameters create the best output for our data. Here is what that code looks like:

```
from sklearn.model_selection import GridSearchCV

mlpCV = mlp.MLPClassifier()
parameter_space = {
    'verbose': [True],
    'hidden_layer_sizes': [10, 20, 30, 40, 50, 100, 200, 300],
    'activation': ['logistic'],
    'alpha': [0.0001, 0.05],
    'learning_rate': ['constant', 'adaptive'],
    'max_iter': [50, 100, 150],
    'batch_size': [100, 200, 300, 400, 500]
}

best.fit(imgs, labels)
best = GridSearchCV(mlpCV, parameter_space, n_jobs=-1, cv=5)

print('Best parameters found:\n', best.best_params_)
```

The output of this code would have returned the best parameters for us to use, however it was setup to run overnight, and over 10 hours later it still wasn't complete. So let's tune the options down a bit. The

data we will be looking at will be comparing rates based on hidden layer sizes and seeing the effects that hidden layers give. So lets trim the parameters down and run it again... hopefully faster.

Our trimmed down parameters:

```
from sklearn.model_selection import GridSearchCV

mlpCV = mlp.MLPClassifier()
parameter_space = {
    'verbose': [True],
    'hidden_layer_sizes': [10],
    'activation': ['logistic'],
    'alpha': [0.0001, 0.05],
    'learning_rate': ['constant', 'adaptive'],
    'max_iter': [50, 100, 150],
    'batch_size': [100, 200, 300]
}

best = GridSearchCV(mlpCV, parameter_space, n_jobs=-1, cv=5)
best.fit(imgs, labels)

print('Best parameters found:\n', best.best_params_)
```

This only took an hour and a half to run, so let's see what the grid search says are our best options!





```
Best parameters found:
{'activation': 'logistic', 'alpha': 0.0001, 'batch_size': 200, 'hidden_layer_sizes': 10, 'learning_rate': 'adaptive', 'max_i
ter': 150, 'verbose': True}
```

What does this mean? Well we only gave the grid search the option of logistic, since we decided that was our best option, the alpha of 0.0001 means we reduce our weights just a tiny bit each time an update is performed to the model. The batch size being 200 means that updating the models after every 200 images have been processed gives the highest probability that the model will be updated correctly. The hidden layers we will look at modifying later, the learning rate being adaptive means that the learning rate is kept constant at 0.001 as long as the training loss continues to decrease. If the training loss does not decrease over two epochs, the learning rate is divided by 5. Max iterations have been set to 200 just in case. All runs of this model has the fit converge before 150 iterations are hit but setting the max iterations to 200 allows the model just a little more time to converge if it needs.

And lastly the verbose option just tells the model to print out the loss every iteration, for visual confirmation that the model is indeed running.

VI. Setting up the MLP with the chosen hyperparameters.

Now we get into creating the actual perceptron and starting to train our model to fit the MNIST dataset. How do we do this? Well first, we need to download the data set, and unzip it into it's base files as seen:

 t10k-images-idx3-ubyte	4/28/2022 6:52 PM	File	7,657 KB
 t10k-labels-idx1-ubyte	4/28/2022 6:52 PM	File	10 KB
 train-images-idx3-ubyte	4/20/2022 12:08 PM	File	45,938 KB
 train-labels-idx1-ubyte	4/28/2022 6:52 PM	File	59 KB

These files are in simple byte format, without any delimiters, meaning each image is stored in 784 bytes, and the 785th byte is the start of the second image. But there are 4 files? Well, there are two files per usage, there is the training set, and the testing set. The training set is labeled with “train” and has one file for the images, and one file for the labels which is much smaller as it only holds a single number for each image. This is the same as the testing set, labeled “t10k” for “test 10 thousand” as there are 10,000 test data points. The code to turn these into python 2D arrays is fairly simple, and provided to us already. It is seen below:

```
def load_mnist(path, kind='train'):
    #Load MNIST data from `path`

    labels_path = os.path.join(path, '%s-labels-idx1-ubyte' % kind)
    images_path = os.path.join(path, '%s-images-idx3-ubyte' % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II', lbpath.read(8))
        labels = np.fromfile(lbpath, dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII", imgpath.read(16))
        images = np.fromfile(imgpath, dtype=np.uint8).reshape(len(labels), 784)

    return images, labels
imgs, labels = load_mnist("./")
```

So now we have two variables: `imgs` and `labels`. `imgs` holds a 2D array of all 6000 training images in a format that looks something like this:

Img #	Pixel value				
0	0	1	2	...	783
1	0	1	2	...	783
2	0	1	2	...	783
:					
5999	0	1	2	...	783

Where `labels` is a 1D array of all 6000 answers to what number the image looks like.

With this data we can now create our MLP classifier with the tuned hyperparameters and fit the data.

Thanks to `sklearn` for making this code very short and simple as seen here:

```
import time

t0 = time.time()
perc = mlp.MLPClassifier(activation='logistic', alpha= 0.0001, hidden_layer_sizes=20,
                        learning_rate= 'adaptive', max_iter= 200, verbose=True)

perc.fit(imgs, labels)
t1 = time.time()
print("Training time: " + str(abs(t0 - t1)))
```

So, what is happening here? Well with the import of `time`, and setting `t0` and `t1`, we can see how long it takes for this classifier to fit itself to the image data, as seen on that last line, the time it takes is a variable we will consider when looking at the overall data. As for the machine itself, we declare `perc`, short for perceptron as the MLP classifier. We setup this MLP with the hyperparameters we decided on earlier, only changing the hidden layer sizes, and the max iterations as we want the machines to converge, and we want the hidden layer sizes to be our independent variable in our data. Next, we do the very simple function call of “fit” telling the MLP to fit its model to the data we got earlier. This is simple on the surface, but there is a mountain of information hidden behind that function call. In layman’s terms, the fit function takes one image, and puts it into the input layer, and takes the label, and says “This is the expected output of that image”. It will do this for as many images as the batch

size declares, in our case 200. After that first batch, it will manipulate the hidden layer to “learn” and change the data given from the input layer to create the type of data expected from the output layer. Each time a batch is finished running, it runs the same images through itself to see how well it learned, this creates what is known as loss. Loss is the bread and butter of learning for our machine. The higher the loss, the worse the machine models our data, the lower the loss, the better. So, we want to aim for as low a loss as possible. All in all, we have an MLP with tuned hyper parameters that is fit to our data So now we can start predicting the 10,000 test images!

VII. Setting up predictions and changing the hidden layer values.

To begin our gathering of data, we need to first get the testing data set into some python lists, this is done by using the same “load_mnist()” function from earlier, just setting “kind='t10k'” so that we are grabbing the testing data set. Then taking our fit model and scoring it on the testing dataset. This gives it a percent correct out of all 10,000 images, displaying our accuracy. This is done with just two lines of code since the load_mnist function was created earlier:

```
test_imgs, test_labels = load_mnist("./", "t10k")
print(perc.score(test_imgs, test_labels))
```

Since we want to change our hidden layer values to see how that effects the machine, we will attach this to the same snippet as the declaration and fit of the model takes place. Making the whole block of code look like this:

```
import time

t0 = time.time()
perc = mlp.MLPClassifier(activation='logistic', alpha= 0.0001, hidden_layer_sizes=20,
                        learning_rate= 'adaptive', max_iter= 200, verbose=True)

perc.fit(imgs, labels)
t1 = time.time()
print("Training time: " + str(abs(t0 - t1)))

test_imgs, test_labels = load_mnist("./", "t10k")
print(perc.score(test_imgs, test_labels))
```

In the snippet of code, the “20” is highlighted in yellow, this is because that is the independent variable we will change to see how it effects out data. Speaking of data lets get into it.

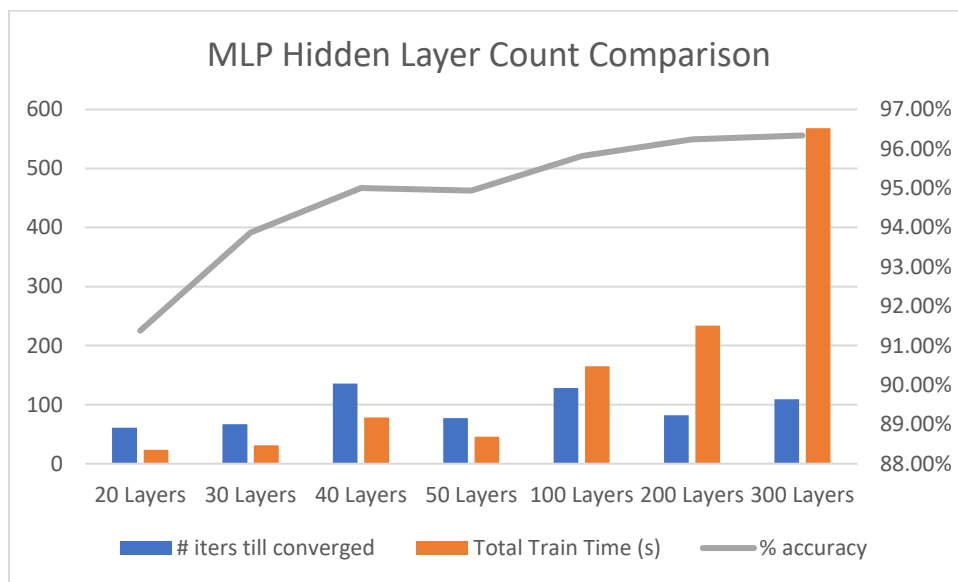
In creatin multiple MLP classifiers with different hidden layer sizes, we have created seven different versions of the same classifier, only with variations on the number of hidden layers, for that we decided to go with 20, 30, 40, 50, 100, 200, and 300 hidden layers. More layers meaning more time taken, but higher accuracy, and fewer layers meaning shorter fit times, but lower accuracy. They question is: What is the acceptable range for accuracy? Do we want to prioritize accuracy, or speed?

So, after running the MLP classifier 5 times on each number of hidden layers, and taking the avg of each of those times, we got some interesting results as you can see here:

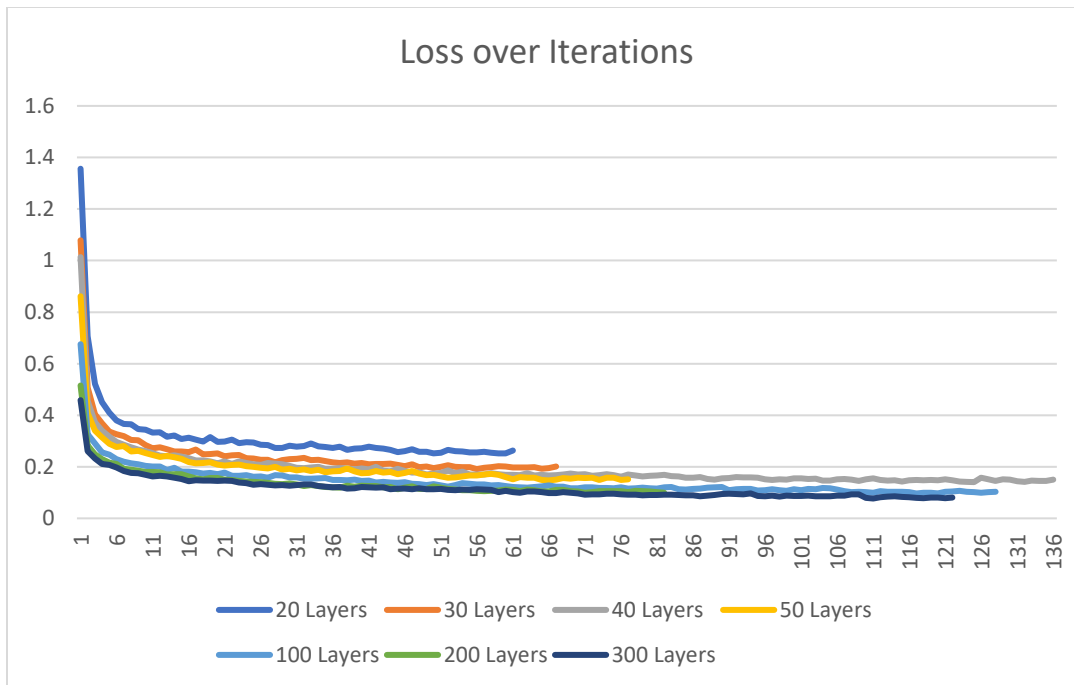
	20 Layers	30 Layers	40 Layers	50 Layers	100 Layers	200 Layers	300 Layers
# iters till converged	61	67	136	77	128	82	109
Total Train Time (s)	23.71	31.14	78.08	45.79	164.78	233.98	568.03
% accuracy	91.38%	93.87%	95.00%	94.94%	95.82%	96.24%	96.34%

What does this data mean? Well we can see a dramatic increase in the number of iterations till converged on 40, 100, and 300 hidden layers. It is interesting to see that certain layer counts, despite being higher, take shorter to converge than others. Take 40 and 50 for example. In 40 we have a 95% accuracy, but 136 iterations till it converged, making the training take longer at about 78 seconds. Whereas 50 layers runs 33 seconds faster, converges after 77 iterations, and has a 94.94% accuracy. Within 10,000 testing points, that’s only 6 images that 50 layers incorrectly guessed where 40 layers correctly guessed. Another point of interest is the fact that, with more hidden layers, accuracy

increases. However, at what point does this become diminishing returns? We see the jump between 20 to 50 layers is a dramatic 3.6%, but the jump from 50 to 300 layers is only 1.4%. Something else to note along the same vein is that the more hidden layers there are, the more time it takes to fit the model, which is not a linear curve, as the jump between 100 to 200 is about 70 seconds, but the jump from 200 to 300 is about 235 seconds. Which is a very slippery slope to fall down. To make this data a bit more visually pleasing, here is a graph that compares the number of hidden layers with each other to see how each one compares side by side:



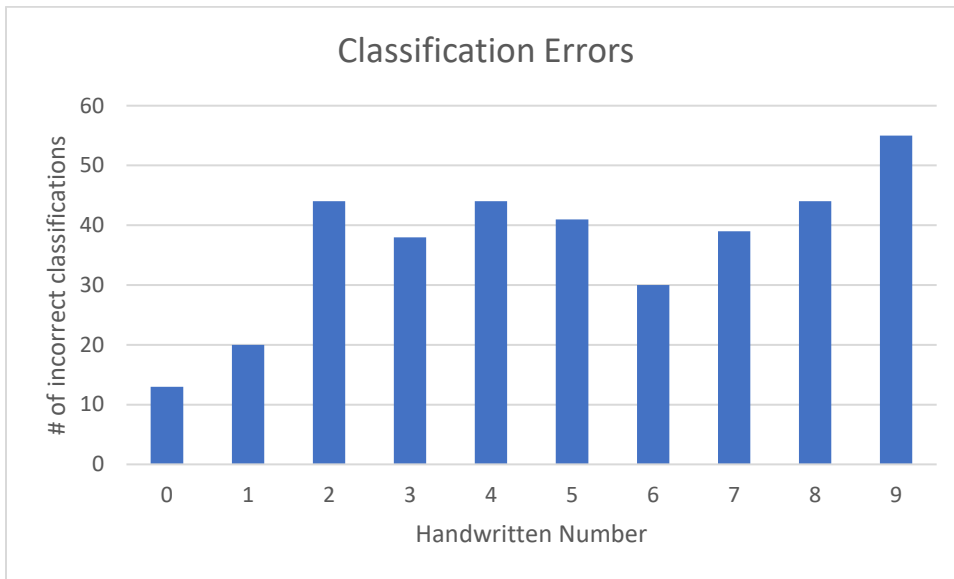
Another way we can determine which number of hidden layers is the most optimal for us to use is to look at the loss after each iteration of the fit. The following graph shows exactly that:



So what is this graph saying exactly? Well, each line is a different number of layers, with the horizontal axis being the number of iterations, and the vertical axis being the loss value given to the program. Remember as stated earlier, the lower the loss, the better our machine is learning. So what we are looking for on this graph is a layer count that quickly drops to a low loss state, but converges, or ends after a small number of iterations. Seeing how the 300 hidden layer model takes almost 10 minutes to run, it would be nice to avoid that one if possible. And seeing how handwriting is something we are really good at recognizing in all shapes and forms, the machine should have as high a percentage accuracy as possible. This points us to the 200 hidden layer option. It only takes about 4 minutes on average to fit, but it has a 96.24% accuracy meaning there are only 376 numbers out of the 10,000 samples that it incorrectly identified.

So, our final chose model is an MLP classifier with 200 hidden layers, an alpha of 0.001, a batch size of 200, a logistic activation function, and adaptive learning rate. Which takes approximately 4 minutes to train on 6000 images, providing an accuracy of 96.24% when compared against 10,000 test points.

VIII. Struggles, and failed points



Above, there is a simple graph. This graph shows the expected letter, and the number of errors it had during the classification. This is going off the MLP we decided on earlier with 200 hidden layers. As seen on the graph, the highest number of failures are with digits such as 9, 2, 4, and 5. Below is the first 10 digits that failed from the testing set:



Now, some of these might seem obvious, that they are certain letters, but as humans we have adapted to reading bad handwriting as accurately as possible. So some of these might have strange angles, or curvatures that the model was not able to train on, resulting in an inaccurate prediction on a number that should be easy to classify.

IX. Conclusion

In all, our multilayer perceptron classification model did it's best. It may not have scored 100% accuracy on the testing set, but our hyper parameter optimizations created a model that had a solid 96.34% accuracy on its highest performing training. And 96.34 is an A according to normal grading standards, so this neural network is capable of identifying handwritten numbers to an acceptable standard. In retrospection, a convoluted neural network might have helped identify some of the common errors the MLP had, however since the accuracy was so high on the MLP the difference was arguable. But in looking for a model with the highest accuracy regardless of complexity, the CNN would likely have had a higher accuracy, and would be a better choice for a task like identifying handwritten numbers.