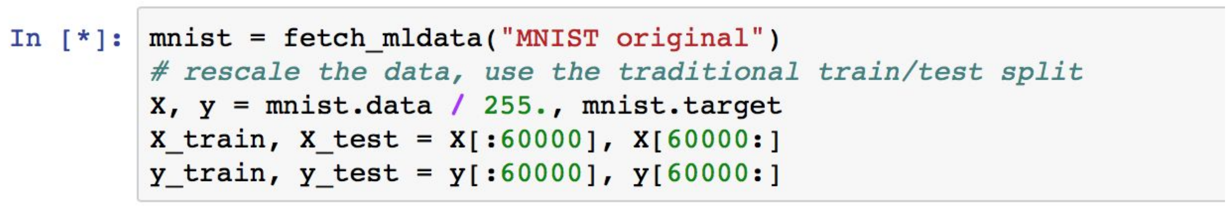
Chase Hendley & Neal Sakash

CSCI 334

March 13, 2017

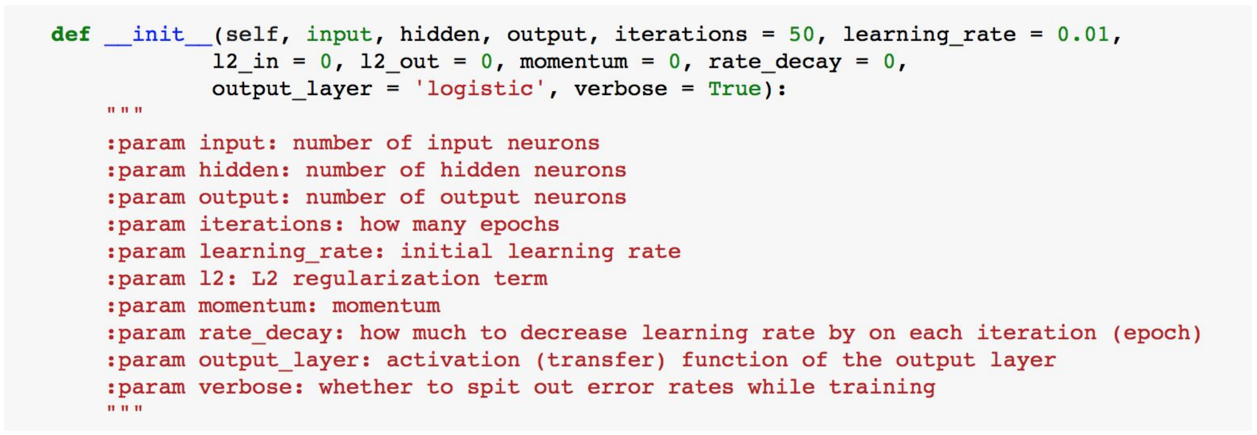
Classifying Handwritten Digits from MNIST Database

Using various architectures and parameters on the MNIST database of handwritten numbers, we are able to use a multilayer perceptron algorithm to predict which digit was the one written with various degrees of accuracy for each setup in our program. First we loaded the data using sample code provided:



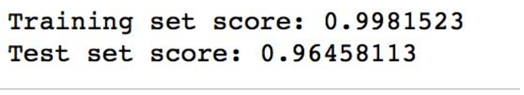
```
In [*]: mnist = fetch_mldata("MNIST original")
        # rescale the data, use the traditional train/test split
        X, y = mnist.data / 255., mnist.target
        X_train, X_test = X[:60000], X[60000:]
        y_train, y_test = y[:60000], y[60000:]
```

The first way we went about this was with this initial setup given by databoys:



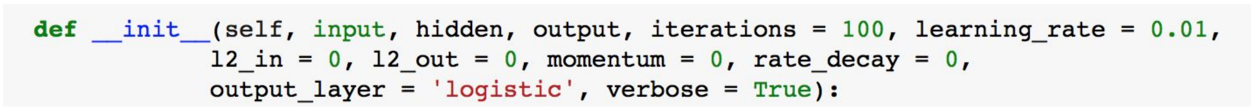
```
def __init__(self, input, hidden, output, iterations = 50, learning_rate = 0.01,
             l2_in = 0, l2_out = 0, momentum = 0, rate_decay = 0,
             output_layer = 'logistic', verbose = True):
    """
    :param input: number of input neurons
    :param hidden: number of hidden neurons
    :param output: number of output neurons
    :param iterations: how many epochs
    :param learning_rate: initial learning rate
    :param l2: L2 regularization term
    :param momentum: momentum
    :param rate_decay: how much to decrease learning rate by on each iteration (epoch)
    :param output_layer: activation (transfer) function of the output layer
    :param verbose: whether to spit out error rates while training
    """
```

What I liked about this was this initialization allowed an easy way to manipulate the parameters for continuous tweaks to the data. The following are the results for this run:

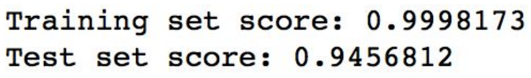


```
Training set score: 0.9981523
Test set score: 0.96458113
```

Next we updated the the parameters to have 100 iterations with the following results:

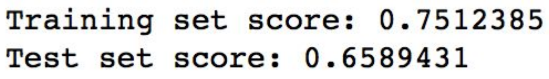


```python
def __init__(self, input, hidden, output, iterations = 100, learning_rate = 0.01,
             l2_in = 0, l2_out = 0, momentum = 0, rate_decay = 0,
             output_layer = 'logistic', verbose = True):
```



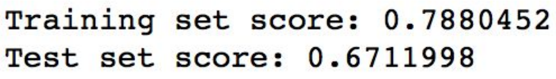
```
Training set score: 0.9998173
Test set score: 0.9456812
```

Our third try involved going back to 50 iterations, but with a learning rate of 0.2 which led to, which was a lot worse than previous attempts:



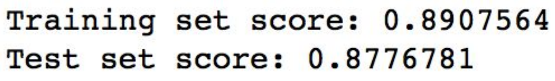
```
Training set score: 0.7512385
Test set score: 0.6589431
```

Our fourth setup involved keeping the learning rate at 0.2 with 100 iterations, with these results:



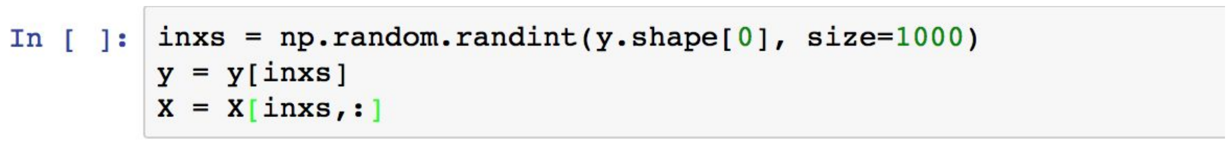
```
Training set score: 0.7880452
Test set score: 0.6711998
```

Finally, our fifth setup reduced the learning rate back down to 0.1 and having only 25 iterations. We wanted to check and see if less is more with this one so reducing the number of times to iterate through the network seemed like it might produce that chance. These are the results from that:



```
Training set score: 0.8907564
Test set score: 0.8776781
```

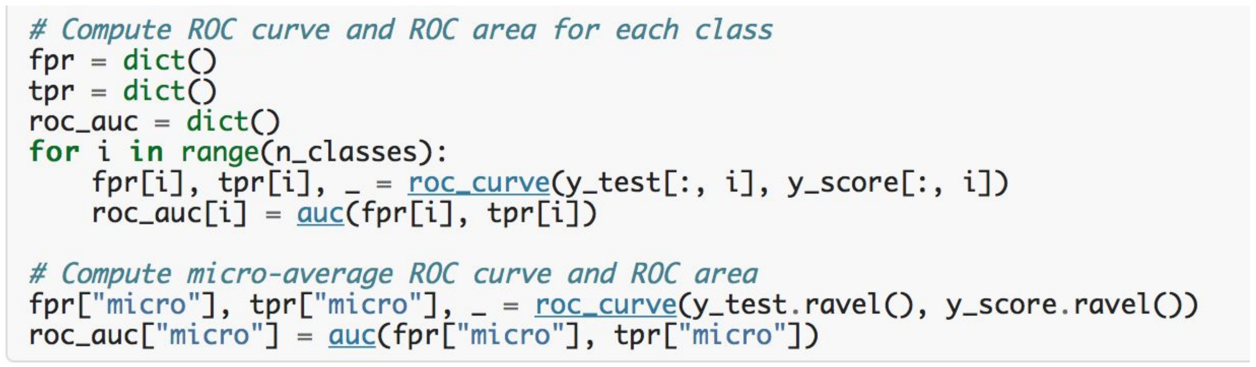
Because our starter code taken from databoys was their final setup, they had actually come up with the best solution given their algorithm and initialization. When discussing with other classmates, I found that they had slightly different algorithms that allowed them better results. However, one classmate shared that they had a result of 1 on the training set score and a 0.99+ test set score. While I believe this may be possible, I would be worried about overfitting the data in that case and would be more confident on our results to correctly predict future handwritten digits. Problems continued with getting a ROC curve to work with a error. After using a suggestion to cut the number of training to better test, with Dr. Anderson’s code of



```
In [ ]: inxs = np.random.randint(y.shape[0], size=1000)
        y = y[inxs]
        X = X[inxs,:]
```

and by also using documentation from

http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html



```
# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
```

We were still unable to produce a proper result.

Sites Used:

http://scikit-learn.org/stable/auto_examples/neural_networks/plot_mnist_filters.html#sphx-glr-aut

o-examples-neural-networks-plot-mnist-filters-py

https://databoys.github.io/ImprovingNN/

http://yann.lecun.com/exdb/mnist/index.html

https://github.com/moeabdol/mlp-mnist/blob/master/MLP.py