PROJECT 4
MACHINE LEARNING ON THE MNIST DATA

OVERVIEW
In this lab you will investigate two machine learning techniques for handwritten digit classification. In doing so, you are encouraged to try out other toolkits besides Numpy and OpenCV.

TASKS
Starter code. Download the starter code from http://www.swarthmore.edu/NatSci/mzucker1/e27_s2017/project4.zip. It contains the MNIST handwritten digit data with 60,000 training examples and 10,000 test examples (described in detail at http://yann.lecun.com/exdb/mnist/), as well as a program mnist_demo.py that demonstrates how to read this data into numpy arrays, and how best to preprocess it for use with learning algorithms (most classification software will want a single matrix with one training example per row, in floating point format, in a reasonable range like $[-1, 1]$).

There is also another demo knn_demo.py that has a nice vectorized implementation of $k$-nearest neighbor queries using numpy. Please read over and run both starter programs to understand what they are doing.

Choose two different classification algorithms to recognize digits. In class, we have discussed the following approaches:

- Single-layer neural network. The network would have 784 inputs (the number of pixels per digit image) directly connected to 10 outputs using the softmax activation function. In Machine Learning lingo, this would also be called a “one-vs.-all logistic regression problem”.

- Multi-layer neural network. In between the 784 inputs and the 10 outputs, there will be at least one hidden layer, using a non-linear activation function such as tanh or ReLU. If you’re going for a single hidden layer, I suggest somewhere in the neighborhood of 50-300 nodes in the hidden layer. If you want to get more fancy, you could do more hidden layers or even a convolutional network.

- $k$-Nearest Neighbors Classifier. This one is easy (no training step!), but will be somewhat slow to test because the training data is so large ($60,000 \times 784$).

- PCA + $k$-Nearest Neighbors. A bit harder to implement, but should be much quicker to evaluate on the test set if you reduce the training data to a reasonable dimensionality (my guess is you can retain quite a bit of accuracy even with 50 or so dimensions).
There are a couple of other classification techniques that we haven’t looked at yet in class, but I would be happy to see you try out:

- Boosted decision trees or boosted decision stumps.
- Support Vector Machines using a radial basis function kernel.
- Other... but please message me first on Piazza to propose your own method.

Note that you don’t need to implement any of these approaches yourself! There’s plenty of pre-existing classification software already available.

OpenCV implements Artificial Neural Network classifiers in Python, but the documentation/tutorials can be a bit spotty and Python implementations have changed a lot between OpenCV versions (especially moving from 2.4 to 3.0), so try not to waste too much time following the documentation for the wrong OpenCV version. There is very stable and easy-to-use support for PCA in OpenCV via cv2.PCACompute.

I also suggest you investigate other libraries/toolkits. I am very partial to scikit-learn (http://scikit-learn.org/), which implements many common Machine Learning algorithms. If you want to try out convolutional neural networks or deep nets, you can look at Keras (https://keras.io/), TensorFlow (https://www.tensorflow.org/), or Theano (http://deeplearning.net/software/theano/). The latter two are both fairly low-level APIs for producing neural networks and similar learners; Keras is an easier-to-use wrapper on top of either TensorFlow or Theano.

Because it’s a bit more mature and stable, scikit-learn should be easy to install via package management tools like MacPorts, apt-get or Anaconda. You will probably need to install the other toolkits through virtualenv/pip. Use Piazza if you want help installing any software.

Before you commit to a particular toolkit/package/implementation, I’d suggest finding and running a simple “hello world” example that convinces you the software has been installed correctly and works as advertised.

**Train your two classifiers.** Before you begin training your classifiers, form a hypothesis about which of the two approaches will achieve a lower error rate on the test set, and why.

Then, train each classification method twice on the 60,000-element training set, and validate it on the 10,000-element test set. For each method, vary one of the hyperparameters (e.g. number of nodes in the hidden layer, number of principal components for PCA, $k$ for $k$NN classification, etc. – see https://www.quora.com/What-are-hyperparameters-in-machine-learning). Once again, form a hypothesis about whether the change will increase or decrease accuracy on the test set, and why.

You may use existing code to train your classifiers. Many toolkits feature MNIST code as part of their tutorial/example suite. I have no problems with you using this code to produce your classifier, but **you must cite any demo/tutorial code you use, and you must run the code on your own computer**, modifying a relevant hyperparameter as described above.
For each of the four conditions, report the following information:

- Training time – this includes backprop, gradient descent, PCA, or any other learning operations. You can use `datetime.now()` to get the current time, and difference two `datetime` objects to get elapsed time in seconds.

- Training error – the percentage error on the training set. For instance, if your learner misclassifies 2,400 of the 60,000 training examples, the training error is 4%.

- Test time – how long does it take to classify all 10,000 examples of the test set? Again, use the difference between two `datetime` objects to get elapsed time.

- Test set error – percentage error on test set.

Summarize this data in a table in your final report.

**WHAT TO TURN IN**

Along with source code for programs implementing both classifiers, please submit a 2-4 page PDF report which addresses the following points/questions:

- Describe your approach to understanding the software you used to train your classifiers. Did you use online tutorials, documentation, or Stack Overflow? What resources were most useful?

- Provide a table summarizing the timing and accuracy data requested above.

- Were your hypotheses about the classifiers’ performance confirmed or disproved? Did you see any significant differences in classification accuracy between the two approaches? By varying the hyperparameters? Discuss any significant differences you saw. *Note: it may be useful to look over the MNIST documentation to see how successful other past approaches have been...*

- Why is it important to validate your final classifier on a separate dataset from the one it was trained on? Did you see substantial differences between training set and test set accuracy?

Submit a zipfile containing your code and report to Moodle by 11:55PM on Sunday, April 30.