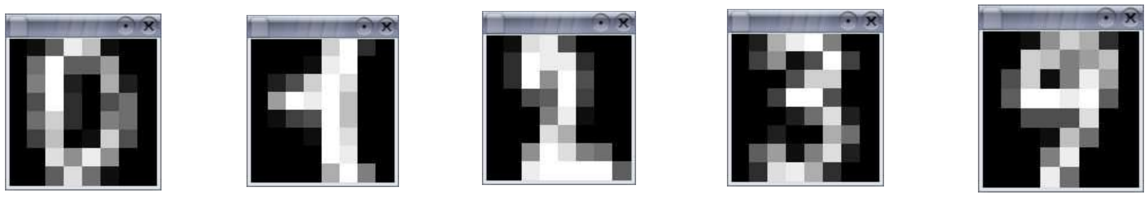
CS 30 Lab 12 — Handwritten Digit Recognition with Neural Networks

You are encouraged to work with a partner on this lab. We'll be experimenting with training a neural network to recognize images of handwritten digits. Each image will be an 8 × 8 grid of grayscale pixels, which we'll represent as a list of 64 real numbers ranging from 0.0 (black) to 1.0 (white). This format is suitable for processing by neural networks. Some example images are shown below:



To get started, you may want to review the notes for Lab 11, where we set up simple networks to learn AND, XOR, etc. Training networks on digit images will be very similar.

1. Copy the files for this lab to your own directory using the following Linux commands:
   ```
   cp -r /common/cs/cs30/labs/lab12 .
   cd lab12
   ls
   ```
   The image patterns are in the file **handwritten-inputs.dat.** Each line of the file contains 64 values, representing a single image. The target patterns corresponding to the correct image classifications are in the file **handwritten-targets.dat**. Each target pattern encodes a digit classification from 0-9. To represent category *N*, the *N*th value in the target is 1, and the rest are 0's. For example, category *three* is represented as the pattern 0 0 0 1 0 0 0 0 0 0, with the third position set to 1 (counting from zero). There are 100 images in all.

2. The file **digits.py** is the neural network program we developed in class. Open it in IDLE and look over the code. This program sets up a network with 64 input units, 5 hidden units, and 10 output units (one for each possible digit category). You can view and manipulate images in the dataset using ordinary Python commands. For example:

   `n.inputs[77]` returns input pattern #77 in the dataset as a list of 64 real numbers
   `n.targets[77]` returns the target classification pattern for image #77 (in this case, the digit 9)
   `n.showPattern(n.inputs[77])` displays image #77 visually in a pop-up window

   Try out these commands at the Python prompt. Look at a few other images in the dataset and their corresponding target classifications.

3. When you're ready to train your network on the 100 input patterns, just type `n.train()` at the Python prompt and watch as the error of the network gradually falls towards zero. How many epochs does it take for the network to learn to classify all 100 patterns correctly? Next, reset the network to random weights by calling the method `n.initialize()` and train the network again. Do this a few more times and record the average number of epochs taken on each trial.

4. To check the performance of the trained network on individual patterns, we can use the `prop` and `classify` methods (see **digits.py**). For example, to see the actual output pattern produced by the network on image #77 (a nine), type the command `n.prop(n.inputs[77])`. To see the corresponding classification, type the command `n.classify(n.inputs[77])`. The classification returned should match the category specified by `n.targets[77]`. Check the network's behavior for a few other images as well.

5. To get a better sense of what happens to the hidden units during training, we can view the weights associated with each hidden unit graphically. The `showWeights` method creates a graphical display of all of the weights going into a particular unit. Perform the following steps:

   1. Reset the network to random weights by calling `n.initialize()`

   2. Type **`for i in range(5): n.showWeights('hidden', i)`** at the Python prompt (you may need to hit Return twice). This creates a graphical display of all of the connections from the 64 input units into each

hidden unit. Connection strengths are indicated by grayscale values, with darker shades corresponding to negative (inhibitory) connections and lighter shades to positive (excitatory) connections. If you randomized your weights, you should see what amounts to just noise for all five hidden units, since all connections have been reinitialized to small random values close to 0.

3. Now train your network again and watch what happens to the connection strengths during learning. You should see interesting patterns appear as the connection strengths are tuned to classify the images in the dataset. Do the hidden units develop sensitivities to particular parts of the "visual field", or do they all end up with the same general pattern of connection strengths?

6. Once your network is trained, it's easy to save the weights in a file and reload them later. This way you can avoid having to retrain your network each time you start up Python. For example, to save the current set of weights in a file called **digit-recognizer.wts**, just type this:

   ```
   n.saveWeightsToFile('digit-recognizer.wts')
   ```

   To load them back in, do this:

   ```
   n.loadWeightsFromFile('digit-recognizer.wts')
   ```

   (Note: In order to reload weights from a file, you must use a network with the same architecture as the network whose weights were saved, *i.e.* both networks must have the same number of layers and units in each layer.) Test this out by saving your trained network's weights, calling `n.initialize()`, and then loading them back in.

7. To make it easier to evaluate your network's performance, write a method called `evaluate()` that goes through all of the input patterns in the dataset and classifies each one, reporting the ones that the network got wrong along with the network's actual output, and keeping track of the total number of incorrect answers. Your method should not print out anything for patterns that were classified correctly. The output of `evaluate` should look something like this (for an incompletely trained network):

   ```
   >>> n.evaluate()
   network classified image #5 (a 2) as 3
   output: [0.06 0.10 0.05 0.95 0.12 0.14 0.02 0.08 0.07 0.04]
   network classified image #14 (a 3) as 8
   output: [0.01 0.15 0.08 0.02 0.02 0.04 0.12 0.09 0.92 0.06]
   network classified image #36 (a 1) as ???
   output: [0.23 0.78 0.02 0.56 0.12 0.63 0.02 0.25 0.15 0.08]
   network classified image #77 (a 9) as 3
   output: [0.02 0.05 0.09 0.85 0.03 0.03 0.02 0.06 0.09 0.05]
    ...
   network got 21 wrong
   ```

8. Repeat the above experiments a few more times using different numbers of hidden units in the network (warning: the more hidden units you have, the more connections there are to train, so training will take longer). Does the network take fewer epochs to learn with more hidden units in general? Does its performance improve, get worse, or stay about the same?

9. The method `splitData` in **digits.py** partitions the current dataset into a *training* portion and a *testing* portion. This will allow us to train the network on one set of patterns, and then test its performance on a different set of patterns. `splitData` takes a number in the range 0-100 as input, representing a percentage value *p*, and stores *p* percent of the dataset patterns in the list `self.inputs`, and the remaining patterns in `self.testInputs`. It stores the corresponding target patterns in `self.targets` and `self.testTargets`. The `train` method only trains on `self.inputs`, not `self.testInputs`. Change your `evaluate` method to use `self.testInputs` and `self.testTargets` instead of `self.inputs` and `self.targets`, so that the networks' performance will be evaluated on the novel (untrained) patterns instead of the patterns used for training.

   One advantage of neural networks is that they often perform well even on data they weren't trained on. This ability to *generalize* their behavior to novel input patterns is a very important property of this class of models. Using `splitData`, experiment with different relative sizes of training and testing sets with your network.