# CSC411: Project #2

Due on Friday, March 10, 2017

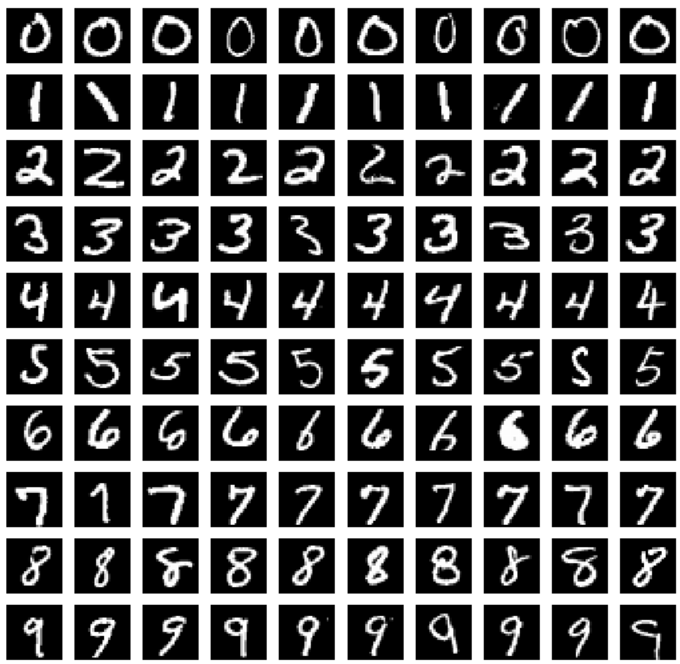
**Fiona Leung** *999766061*
**Timur Borkhodoev** *999376394*

March 10, 2017

# Part 1

*Dataset description* The dataset contained an even number of images per digit, having no more or less information the neural networks coud train off of determine one digit better apart from another. Each digit had multiple images of it where it was drawn with varying writing styles. Some digits were written with more loops and imperfections than others, had different thicknesses or were drawn slanted at different angles.Some digit images had discontinuous lines in the number shapes.
For example, in the images of the 0's some had a closed loop while others were perfect and had a small edge that jutted out near its top.

1. variety of angles
2. different styles of handwriting
3. gaps between continuous lines
4. different thickness levels



# Part 2

*Compute the network function by propagating forward and discarding intermediate results*

```
def compute_network(x, W0, b0, W1, b1):
    _,_, output = forward(x, W0, b0, W1, b1)
    return = argmax(output)
```

## Part 3

1. We will use negative log-probabilities as our cost function, and find its gradient

```
def cross_entropy(y, y_):
  return -sum(y_ * log(y))
```

2. Vectorized code for computing gradient of the cost function

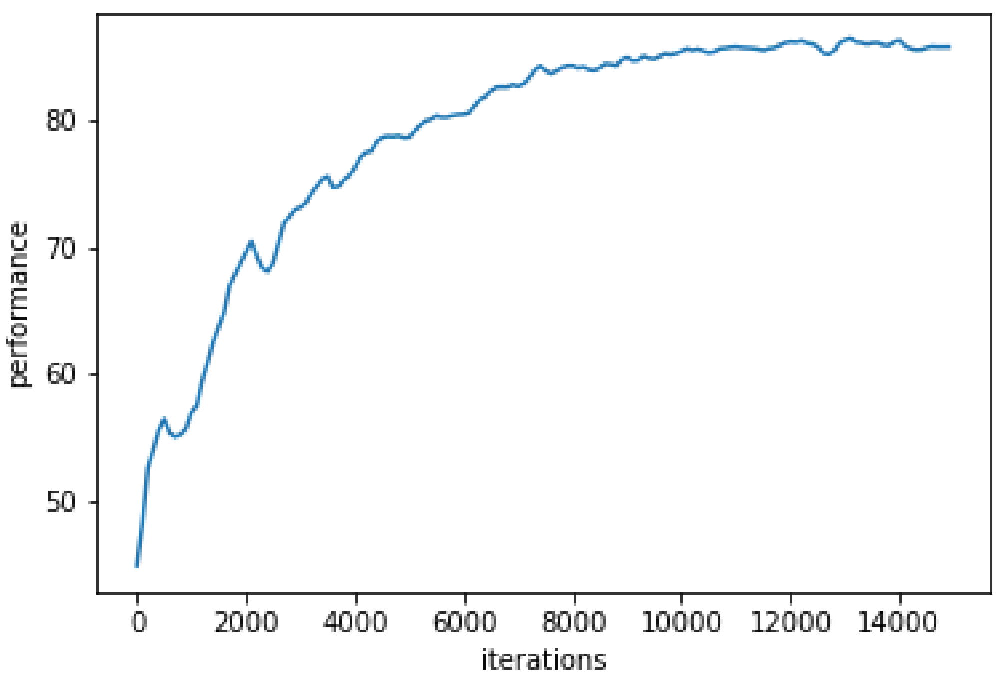
```
def deriv_multilayer(W0, b0, W1, b1, x, L0, L1, y, y_):
    dCdL1 = y - y_
    dCdW1 = dot(L0, dCdL1.T)
    dCdobydodh = dot(W1, dCdL1)
    diff = 1 - L0**2

    dCdW0 = tile(dCdobydodh, 28 * 28).T * dot(x, (diff.T))
    dCdb1 = dCdL1
    dCdb0 = dCdobydodh * diff

    return dCdW1, dCdb1, dCdW0, dCdb0
```

## Part 4

```
def train(plot=False):
    global W0, b0, W1, b1
    global plot_iters, plot_performance
    plot_iters = []
    plot_performance = []
    alpha = 1e-3
    for i in range(150):
        X, Y, examples_n = get_batch(i * 5,10)

        update = np.zeros(4)

        for j in range(examples_n):
            y = Y[j].reshape((10, 1))
            x = X[j].reshape((28 * 28, 1)) / 255.
            L0, L1, output = forward(x, W0, b0, W1, b1)
            gradients = deriv_multilayer(W0, b0, W1, b1, x, L0, L1, output, y)
            update = [update[k] + gradients[k] for k in range(len(gradients))]

        # update the weights
        W1 -= alpha * update[0]
```

```
        b1 -= alpha * update[1]
        W0 -= alpha * update[2]
        b0 -= alpha * update[3]
        if plot:
            plot_iters.append(i * examples_n)
            plot_performance.append(test_perf())
    return plot_iters,plot_performance
train(plot=True)
```

```
def get_batch(offset,example_per_class=5):
    # 5 examples per class
    classes_num = 10
    x_batch = np.zeros((example_per_class * classes_num, 28 * 28))
    y_batch = np.zeros((example_per_class * classes_num, classes_num))
    for i in range(classes_num):
        for j in range(example_per_class):
            x_batch[i * example_per_class + j] = M['train' + str(i)][j +
                                                                      offset]
            y_batch[i * example_per_class + j][i] = 1
    return x_batch, y_batch, example_per_class * classes_num
```



# Part 5

So as discussed in lecture large errors are penalized quadratically in linear regressions. So our multinomial regression doesn't suffer from it. We start with generating a noise for our data set.

```
# generate noise and N(0,σ^2)
noise = scipy.stats.norm.rvs(scale=5,size=784*50*10)
noise = noise.reshape(500,784)
X,Y,n = get_batch(offset=0,examples=50)
X += noise
```

After modifying our data set with noise - we get images that look like this