CSC411 - Project #2

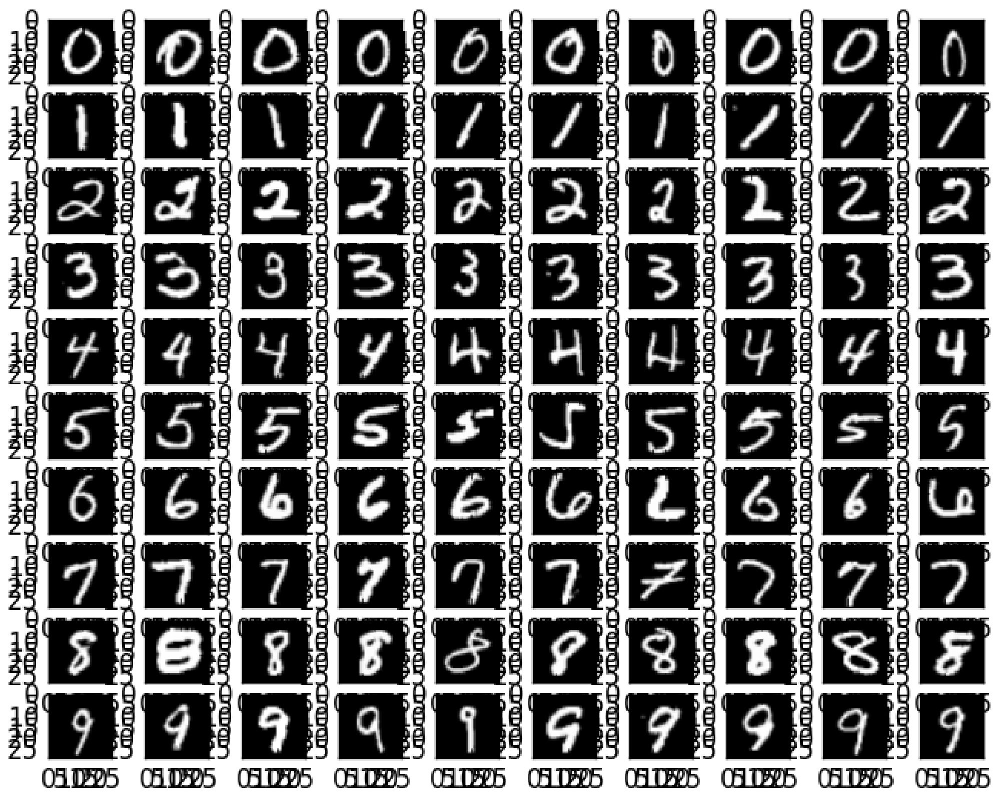
Michael Wong - [Student Number]
Yijin (Catherine) Wang - 998350476

March 7, 2017

Part 1

Question Describe the dataset of digits. In your report, include 10 images of each of the digits. You may find matplotlibs subplot useful.

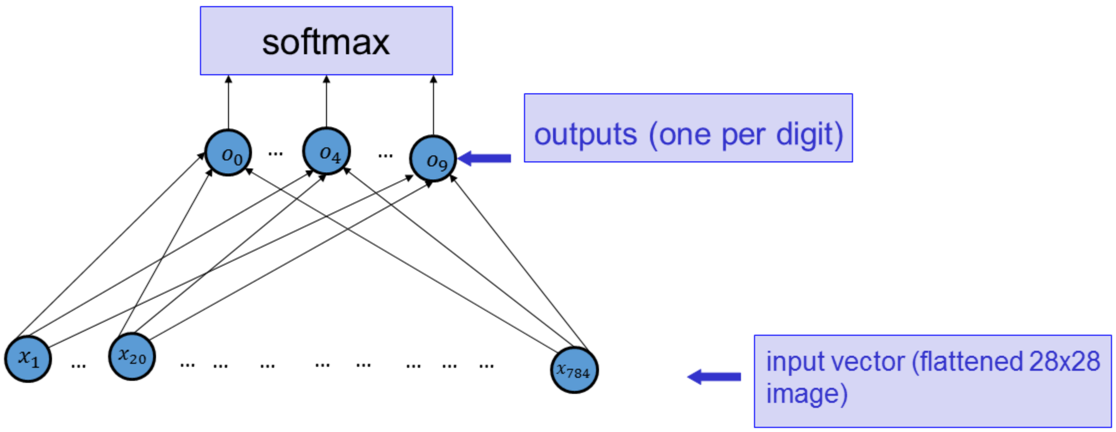
Answer Please see the images of each digits below. For the same digit, some images have thicker lines (like "8" in the second column) and some have thiner lines (like "8" in the third column). Some digit images are straight (like "1" in the first column) and some digits are rotated (like "1" in the ninth column). Most digits are clear to be identified manually but some digits are not clear (like the "6" in the seventh column).



10 images of digits from 0 to 9

Part 2

Question Implement a function that computes the network below. The o's here should simply be



linear combinations of the x's (that is, the activation function in the output layer is the identity). Supecifically, use $o_i = \sum_j w_{ji}x_j + b_i$. Include the listing of your implementation in your report for this Part.

Answer Please see the source code of the function below:

```
def softmax(y):
    '''Return the output of the softmax function for the matrix of output y. y
    is an NxM matrix where N is the number of outputs for a single case, and M
    is the number of cases'''
    return exp(y)/tile(sum(exp(y),0), (len(y),1))


def lin_combin(w, b, x):
    o = (dot(w.T, x) + b)
    return softmax(o)
```

Part 3

We would like to use the sum of the negative log-probabilities of all the training cases as the cost function.

Part 3(a)

Question Compute the gradient of the cost function with respect to the weight w_{ij}. Justify every step. You may refer to Slide 7 of the One-Hot Encoding lecture , but note that you need to justify every step there, and that your cost function is the sum over all the training examples.

Answer When sample size is one, we could calculate $\frac{\partial p_j}{\partial o_i}$ in two cases:

1. If $j = i$,

$$\begin{aligned} \frac{\partial p_i}{\partial o_i} &= \frac{e^{o_i}}{\sum_k e^{o_k}} - \frac{e^{o_i}}{\sum_k e^{o_k}} \cdot \frac{e^{o_i}}{\sum_k e^{o_k}} \\ &= p_i(1 - p_i) \end{aligned}$$

2. If $j \neq i$,

$$\begin{aligned} \frac{\partial p_j}{\partial o_i} &= -\frac{e^{o_j}}{\sum_k e^{o_k}} \cdot \frac{e^{o_i}}{\sum_k e^{o_k}} \\ &= -p_j p_i \end{aligned}$$

In summary,

$$\frac{\partial p_j}{\partial o_i} = \begin{cases} p_i(1 - p_i) & \text{, if } j = i \\ -p_j p_i & \text{, if } j \neq i \end{cases} \tag{1}$$

$$\frac{\partial C}{\partial p_j} = -\frac{y_j}{p_j}$$

$$\begin{aligned} \frac{\partial C}{\partial o_i} &= \sum_j \frac{\partial C}{\partial p_j} \frac{\partial p_j}{\partial o_i} \\ &= \sum_{j, j \neq i} \frac{\partial C}{\partial p_j} \frac{\partial p_j}{\partial o_i} + \frac{\partial C}{\partial p_i} \frac{\partial p_i}{\partial o_i} \\ &= \sum_{j, j \neq i} (-\frac{y_j}{p_j}) \cdot (-p_j p_i) - \frac{y_i}{p_i} \cdot p_i(1 - p_i) \\ &= \sum_{j, j \neq i} y_j p_i - y_i + y i p_i \\ &= \sum_j y_j p_i - y_i \end{aligned}$$

Because $\sum_j y_j = 1$, $\sum_j y_j p_i = p_i$. Then,

$$\frac{\partial C}{\partial o_i} = p_i - y_i$$

For the following equation, let j refers to the input size (number of x per sample) and i refers to the output size (number of o per sample), and k refers to the training set size (number of samples).

X is the input matrix with dimension $n \times m$, where n is the number x (related to j) and m is the sample size (related to k). P is the calculated result matrix after using softmax with dimension $d \times m$, where d is the output size (number of outputs per sample, related to i). Y is the expected output with the same dimension as P.

$$\frac{\partial C}{\partial o_{ik}} = p_{ik} - y_{ik}$$

$$\begin{aligned}\frac{\partial o_{ik}}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}}(w_{i1}x_{1k} + \cdots + w_{ij}x_{jk} + \cdots + w_{in}x_{nk} + b_{ik}) \\ &= x_{jk}\end{aligned}$$

$$\begin{aligned}\frac{\partial C}{\partial w_{ij}} &= \sum_k \frac{\partial C}{\partial o_{ik}} \cdot \frac{\partial o_{ik}}{\partial w_{ij}} \\ &= \sum_k x_{jk}(p_{ik} - y_{ik}) \\ &= X(P-Y)^T\end{aligned}$$

Part 3(b)

Question Write vectorized code that computes the gradient with respect to the cost function. Check that the gradient was computed correctly by approximating the gradient at several coordinates using finite differences. Include the code for computing the gradient in vectorized form in your report.

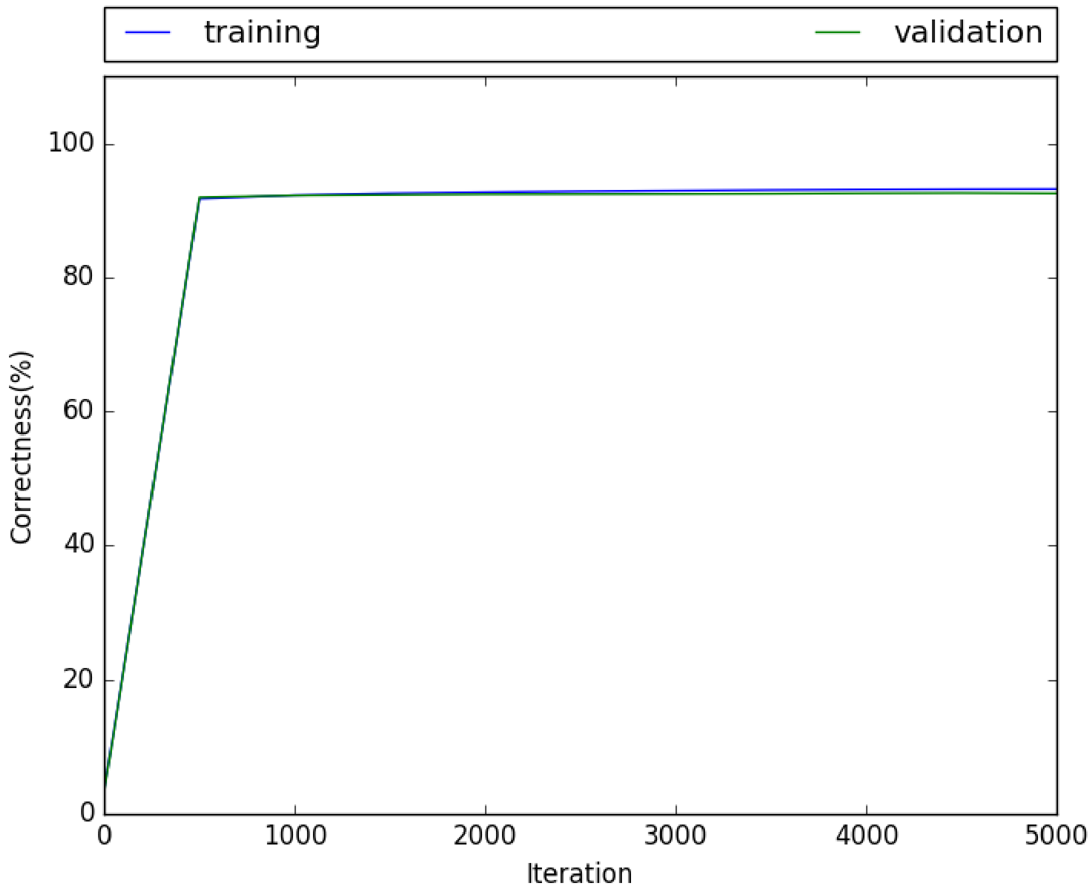
```
def part3b(M):
    """Test gradient for part 3b."""

    x1 = M["train8"][0]
    x2 = M["train8"][1]
    x = vstack((x1, x2)).T
    y = array([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]).T
    w = zeros((x.shape[0], 10))
    b = zeros((10, 2))
    test_gradient(x, y, b, w)


def test_gradient(x, y, b, w):
    """
    Compare the result of my gradient function NLL_gradient and the gradient get
    from finite differences method. Test three times on different w_ij.
    """
```

```
    j_list = [5, 8, 8]
    i_list = [300, 601, 600]

    h = 10e-5

    for k in range(0, 3):
        i = i_list[k]
        j = j_list[k]
        g_finite = finite_diff_gradient(x, y, w, b, i, j, h)
        g_mine = NLL_gradient(x, y, w, b)[i, j]
        diff = abs(g_mine - g_finite)
        print("Test "+str(k)+":")
        print("gradient_finite_differences = "+str(g_finite))
        print("gradient_my_function = "+str(g_mine))
        print("difference = "+str(diff))
        print("---------------------------------------")


def finite_diff_gradient(x, y, w, b, i, j, h):
    """
    Use finite difference to calculate the gradient on w_ij with h.
    x: input
    y: expected output
    w: weight
    b: bias
    """

    c = NLL(x, y, w, b)
    new_w = w
    new_w[i, j] += h
    new_c = NLL(x, y, new_w, b)
    return (new_c - c)/h


def NLL(x, y, w, b):
    p = lin_combin(w, b, x)
    return -sum(y*log(p))


def NLL_gradient(x, y, w, b):
    p = lin_combin(w, b, x)
    return dot(x, (p - y).T)
```

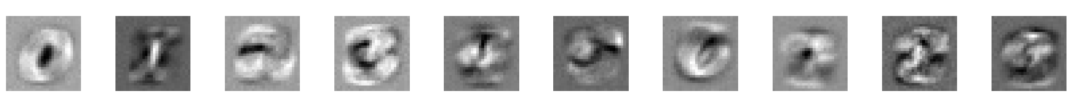
Part 4

Question Train the neural network you constructed. Plot the learning curves. Display the weights going into each of the output units. Describe the details of your optimization procedure.

Answer Please see the figures below.



Learning Curve



w0 w1 w2 w3 w4 w5 w6 w7 w8 w9

Weights corresponding to the 10 digits at 500 iterations

We used gradient descent to train our network. Using the cost function and its gradient we got from previous parts, we ran gradient descent on both the weights and bias. We set the maximum number of iterations to 10000, EPS to 10^{-10}, and we tested several times to get the alpha 0.00001.

Part 5

Question The advantage of using multinomial logistic regression (i.e., the network you constructed here) over using linear regression as in Project 1 is that the cost function isnt overly large when the actual outputs are very far away from the target outputs. That causes the network to not adjust the weights too much because of a single training case that causes the outputs to be very large. You should come up with a dataset where the performance on the test set is better when you use multinomial logistic regression. Do this by generating training and test sets similarly to how we did it in lecture . Show that the performance using multinomial logistic regression (on the test set) is substantially better. Explain what you did and include code to clarify your point. Clearly explain why your experiment illustrates your point.

Part 6

Question Backpropagation can be seen as a way to speed up the computation of the gradient. For a network with N layers each of which contains K neurons, determine how much faster is (fully-vectorized) Backpropagation compared to computing the gradient with respect to each weight individually. Assume that all the layers are fully-connected. Show your work. Make any reasonable assumptions (e.g., about how fast matrix multiplication can be peformed), and state what assumptions you are making.

Answer Notation:

- h_{iq} - q^{th} nequron in layer i
- $w^{i,p,q}$ - weight from p^{th} neuron in layer $i-1$ to q^{th} neuron in layer i
- Assume for each neuron h_{iq}, $h_{iq} = \sum_p w^{i,p,q} h_{i-1,p} + b_{i,q}$
- Assume the model is the same model as earlier parts, so o_is $\rightarrow$ softmax $\rightarrow$ p_is $\rightarrow$ NLL
- Assume the model is for one sample in the training set

1.
$$\frac{\partial C}{\partial w_{n,p,q}} = \frac{\partial C}{\partial o_q} \cdot \frac{\partial o_q}{\partial w^{n,p,q}} = (p_q \cdot y_q) h_{n-1,p}$$
$$\nabla_n C = (H_{n-1})^T (P - Y)$$
where H, P, and Y are $k \times 1$ vectors.

2.
$$\frac{\partial C}{\partial h_{n-1,q}} = \sum_i \frac{\partial C}{\partial o_i} \cdot \frac{\partial o_i}{\partial h_{n-1,q}} = \sum_i \frac{\partial C}{\partial o_i} \cdot w^{n,q,i}$$
Then,
$$\frac{\partial C}{\partial h_{n-1}} = W^n \cdot \frac{\partial C}{\partial O}$$
where W^n is a $k \times k$ matrix and $\frac{\partial C}{\partial O}$ is a $k \times 1$ vector.

3.
$$\frac{\partial C}{\partial w^{n-1,p,q}} = \frac{\partial C}{\partial h_{n-1,q}} \cdot \frac{\partial h_{n-1,q}}{\partial w^{n-1,p,q}} = \frac{\partial C}{\partial h_{n-1,q}} \cdot h_{n-2,p}$$
$$\nabla_{n-1} C = (\frac{\partial C}{\partial H_{n-1}})^T \cdot H_{n-2}$$
where $\frac{\partial C}{\partial H_{n-1}}$ and H_{n-2} are $k \times 1$ vectors.

4.
$$\frac{\partial C}{\partial h_{n-2}, q} = \sum_i \frac{\partial C}{\partial h_{n-1,i}} \cdot \frac{\partial h_{n-1,i}}{\partial h_{n-2,q}} = \sum_i \frac{\partial C}{\partial h_{n-1,i}} \cdot w^{n-1,q,i}$$
$$\nabla_{n-2} C = (\frac{\partial C}{\partial H_{n-1}})^T \cdot H_{n-3}$$

In conclusion, for layer i:

$$\frac{\partial C}{\partial H_i} = W^{i+1} \cdot \frac{\partial C}{\partial H_{i+1}}$$

which have k^2 multiplication.

$$\nabla_i C = \frac{\partial C}{\partial H_i} \cdot H_{i-1}$$

which have k multiplication. So for $n-1$ layers, we have $k^2(n-1)$ multiplications, and n gradient calculation takes kn multiplications. The runtime for backpropagation is polynomial. If we don't use backpropagation and any storage method, for layer i, we need to recalculate layer the derivative at layer $i+1$, the runtime is exponential. Therefore, backpropagation speed up the computation of the gradient.