CSC411 Machine Learning
Project 2: Deep Neural Network

Ariel Kelman
Student No: 1000561368

Gideon Blinick
Student No: 999763000

22 February 2018

1 Introduction

This section provides an introduction to Project 2, including the data used, and notes on reproducing the results. Sections 2-6 discuss the implementation of a simple neural network to classify images from the MNIST database, section 7 is a theoretical anlaysis of the efficiency of backpropogation, and sections 8-10 work with deep neural networks.

1.1 Digits

The following figure shows 10 random images from the training set of each of the digits.

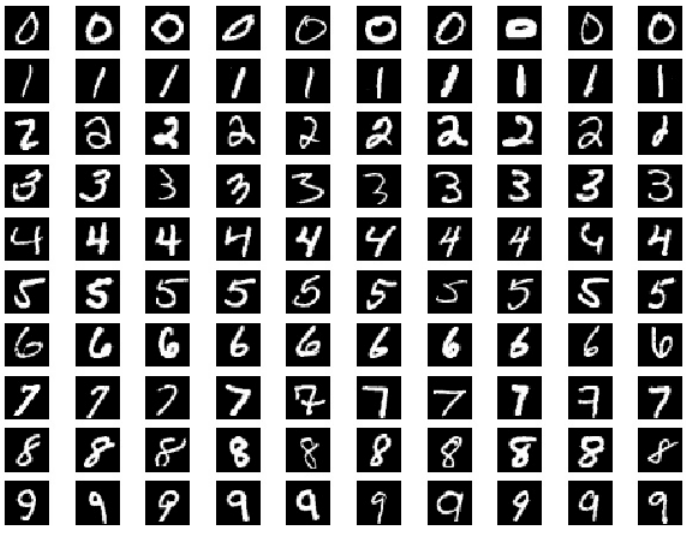

Figure 1: 10 random samples from the training set for each digit. This image was generated using `plot_samples()`.

The data was downloaded from the assignment webpage, and imported into Python using the provided code. The data was already divided into training and testing sets...

1.2 Results & Report Reproducibility

All results and plots can be generated using the python file... To reproduce the report, simply run it through latex.

2 The Network

The following function implements a neural network with no hidden layers, with the output passed through a softmax layer to estimated probabilities.

```
    def no_hidden_layers(x, W, b):
'''Compute the network'''
  #the first column of W contains the weights for output 1
    # W is (784x10), x is (784x60 000) and b is (10x60 000)
    # So L1 is (10x60000)
  L1 = np.dot(W.T, x) + tile(b, (1, np.shape(x)[1] ) )
  return softmax(L1)
```

The network is described by the weights and biases from the 784 = 28*28 inputs (representing pixel intenisites of the input image) to ten "output" nodes (with the identity as the activation function). The output from this layer is what is passed through the sofmax function $p_k = \frac{e^{o_k}}{\sum_q e^{o_q}}$.

The weights are represented as a 784 by 10 matrix W, where w_{ij} (i^{th} row, j^{th} column) represents the weight from the i^{th} input to the j^{th} output. When computing the network on a given sample, the transpose of W is multiplied by the column vector (or matrix when computing on multiple samples) representing the input. The biases are represented by a 10 by 1 vector, one entry for each output. We use the tile function on b to produce a matrix that is 10x60000 so that we can add it the dot product of W^T and x. This dot product repesents $\sum_j (w_{ji})x_j$ for each element, which together with b_i form o_i, the output to be entered into the softmax function.

Throughout the code, $y\_$ is a matrix representing the correct results for each image; each column is a vector of zeros with a 1 in the place representing the correct digit. y is a matrix representing the output probabilities from the network of the same dimensions as $y\_$; each column is a vector representing the probabilities for each digit for a particular sample.

3 Gradient

The cost function is taken to be $-\sum_k y_k ln(p_k)$ for one sample, where y_k is 1 for the correct class and 0 otherwise, and p_k is the prediction probability for class k. Writing this in vector notation (replacing the sum with a vector dot product) and summing over all the samples gives:

$$C = -\sum_s y^{(s)} ln(p^{(s)})$$

where ln is applied pointwise, and s is an index over the samples. $y^{(s)}$ is a vector of 0's with a 1 in the position representing the correct digit.

The following results will be used in the derivations throughout this section:

$$\frac{\partial p_k}{\partial o_q} = \begin{cases} -p_k p_q & \text{if } k \neq q \\ p_q(1-p_q) & \text{if } k = q \end{cases}$$

These results follow directly from the definition of the softmax function.

3.1 Gradient wrt w_{ij}

First, we derive $\frac{\partial C}{\partial o_q}$ (for one sample), to be used in the derivation of $\frac{\partial C}{\partial w_{ij}}$.

$$\begin{aligned}
\frac{\partial C}{\partial o_q} &= \left[\frac{\partial C}{\partial p_q}\frac{\partial p_q}{\partial o_q} + \sum_{k\neq q}\frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial o_q}\right] \\
&= -\left[y_q\frac{1}{p_q}p_q(1-p_q) + \sum_{k\neq q} y_k\frac{-1}{p_k}p_k p_q\right] \\
&= -\left[y_q(1-p_q) - \sum_{k\neq q} y_k p_q\right] \\
&= \left[-y_q + \sum_{k} y_k p_q\right] \\
&= p_q - y_q
\end{aligned}$$

The last line follows because $\sum_k y_k = 1$. Using this result, and applying the chain rule gives:

$$\begin{aligned}
\frac{\partial C}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}}\sum_s y^{(s)} ln(p^{(s)}) \\
&= \sum_s\sum_q \frac{\partial C}{\partial o_q}\frac{\partial o_q}{\partial w_{ij}} \\
&= \sum_s\sum_q (p_q - y_q)x_i\delta_{jq} \\
&= \sum_s (p_j - y_j)x_i
\end{aligned}$$

As the outputs are linear functions of both the inputs and the weights, the derivative with respect to a weight is given by the value of the input to that weight, namely x_i. The Kronecker delta δ_{jq} results from $\frac{\partial o_q}{\partial w_{ij}}$ being nonzero only when $j = q$. The (s) superscript indicating the sample index was omitted throughout the derivation for clarity of presentation.

3.2 Vectorized Gradient Code

The following code computes the gradient with respect to the wieghts and biases (representing all input training images in a matrix X, 784 by the number of samples, in this case 60 000) gives

```
        def grad(y_, y, x):
'''Compute the gradient wrt weights and biases'''
    #y and y_ have dimension (10x60000)
    #x has dimension (784x60 000)

    diff = (y - y_) #y is output of softmax
        # diff is p_j - y_j

    grad_W = np.dot( x, diff.T )
    grad_b = np.sum( diff, 1)  #could try with np.mean()
    grad_b = np.reshape(10,1)

  return  grad_W, grad_b
```

The above code works by creating a matrix $\frac{\partial C}{\partial \mathbf{W}}$ which is of dimension 784x10 (for the 7840 weights), and where each element contains $\frac{\partial C}{\partial w_{ij}}$ for some i,j. It does this by implementing $\sum_s (p_j - y_j)x_i$ for

each matrix element, where i increases as you go down a column (from 1 to 784) and j increases as you go accross a row (from 1 to 10). Each element itself contains 60000 terms added together to aggregate the information from all samples.

4 Training

The neural network was trained on the full training set of 60000 images without momentum. A learning rate of $1 \cdot 10^{-4}$ was used for 1000 iterations, giving a training accuracy of 92.7% and a testing accuracy of 91.8%. The weights and biases were initialized using a uniform distribution to values between 0 and 1.

The following figure shows a learning curve:

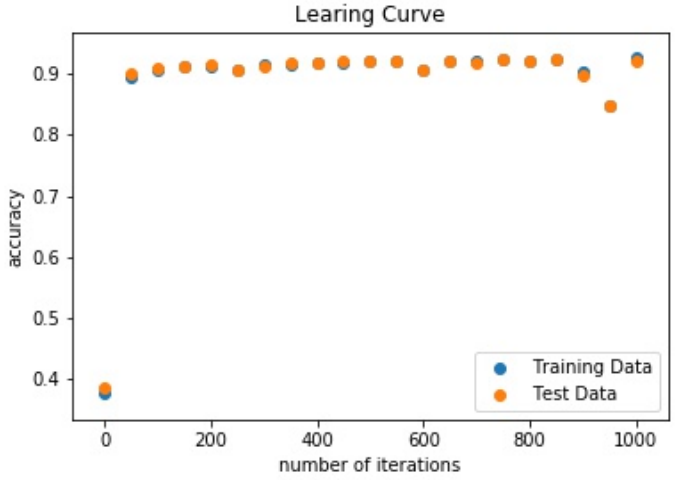

Figure 2: accuracy vs. number of iteration for learning rate of $10^{-}4$

The following figure shows the weights:

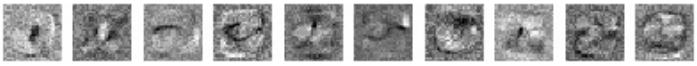

Figure 3: weight diagrams

4.1 Implementation Details:

For this section, we initialized our weights randomely using the

```
rd.rand(784, 10)
```

function. This is so that gradient descent can be run starting from a point where the neural network's guess is equivalent to chance. The learning rate we used was 10^{-4}. This was used because it produced the greatest accuracy on our validation (91.5%) when compared with 10^{-3},$5 \cdot 10^{-5}$,10^{-5}, and 10^{-6}.

5 Training with Momentum

The neural network was trained on the full training set of 60000 images with momentum, set to 0.8. A learning rate of $1 \cdot 10^{-4}$ was used for 1000 iterations, giving a training accuracy of xxx and a testing accuracy of xxx. The weights and biases were initialized using a uniform distribution to values between 0 and 1.

The following figure shows the learning curves:

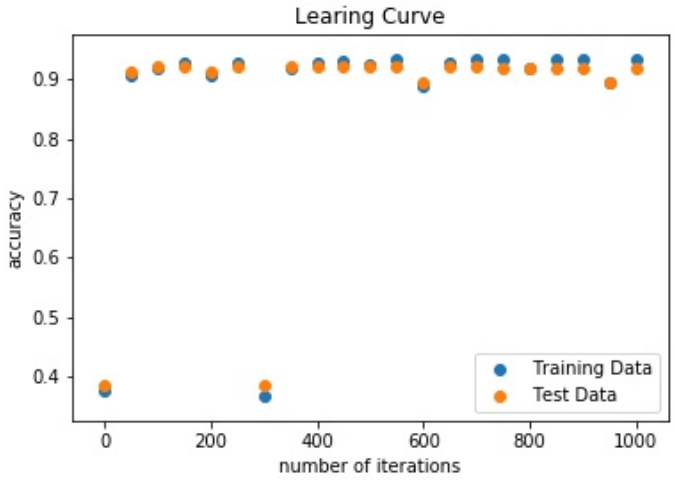

Figure 4: learning curves for neural network with momentum of 0.8

Our new learning curve for training performs better than the learning curve without momentum, but the learning curve for testing performs slightly worse (taking the accuracy after 1000 iterations). Without momentum, a success rate of 92.7% and 91.8% were achieved on the training and test sets, respectively, after 1000 iterations. With momentum, a success rate of 93.2% and 91.6% were achieved on the training and test sets, respectively, after 1000 iterations.

The backpropogation algorithm from part 4 was altered for this part to accomodate momentum. A momentum term (set by default to 0) was added, along with the corresponding update equations for W and b that make use of the momentum term. Thus, our new backprogation function was implemented as:

```
def backprop(x_train, y_train, x_val, y_val, W, b, rate, max_iter, mom=0, filename
=''):
# Part 4
iter_acc = []
train_acc = []
test_acc = []

nu_W = np.zeros( np.shape(W) )
```

```
    nu_b = np.zeros( np.shape(b) )

    iter = 0
    while iter <= max_iter:
        y = no_hidden_layers(x_train, W, b)
        #prevW = W.copy()          #don't need

        grad_W, grad_b = grad(y_train, y, x_train)
        nu_W = mom*nu_W + rate*grad_W
        nu_b = mom*nu_b + rate*grad_b
        W -= nu_W
        b -= nu_b

        if iter%50 == 0:
            iter_acc += [iter]

            y = no_hidden_layers(x_train, W, b)
            res = check_results(y_train, y)
            #print( 'Train Results: ' + str(res.count(1)) + '/' + str(len(res)) )
            train_acc += [ res.count(1)/len(res) ]

            y = no_hidden_layers(x_val, W, b)
            res = check_results(y_val, y)
            #print( 'Test Results: ' + str(res.count(1)) + '/' + str(len(res)) )
            test_acc += [ res.count(1)/len(res) ]
        iter += 1

    #Part 4.1.1: Plot the learning curves.
    if filename:
        plt.scatter(iter_acc, train_acc, label='Training Data')
        plt.scatter(iter_acc, test_acc, label='Test Data')
        plt.title('Learing Curve')
        plt.xlabel('number of iterations')
        plt.ylabel('accuracy')
        plt.legend()
        plt.savefig('resources/' + filename)
        #plt.show()
        plt.close()

    return W, b
```

The following function was implemented in order to determine the optimal value for our momentum variable:

```
    def optimize_momentum(learning_rate, x_train, y_train, x_val, y_val, W, b,
    max_iter, momentum_rates):
    # Part 5
    val_acc = []
    k = 0
    for momentum in momentum_rates:
        k += 1
        rd.seed(0)
        W = rd.rand(784, 10)
        b = rd.rand(10, 1)
        W, b = backprop(x_train, y_train, x_val, y_val, W, b, learning_rate, max_iter,
     momentum, filename='part5_optimize_momentum'+str(k)+'.jpg' )
        y = no_hidden_layers(x_val, W, b) #the network guesses for the validation set
        res = check_results(y_val, y)
        val_acc += [ res.count(1)/len(res) ]
    return val_acc
```

6 Analysis of Momentum

For the analysis in this section, the weights w_{ab} and w_{cd} were used (recall that W is a 784 by 10 matrix). After training the network as described in the previous section, these weights had the values $w_{ab} = xxx$ and $w_{cd} = yyy$.

6.1 Contour Plot

The following plot shows the variation of the cost function (negative log loss) around the values for w_{ab} and w_{cd} chosen above.

6.2 Trajetory without Momentum

6.3 Trajetory with Momentum

6.4 Discussion

7 Efficiency of Vectorized Backpropogation

Consider a fully connected neural network with N layers each containing K neurons. By convention, the input layer will not be counted toward N layers, but the final output layer will be counted (thus there are N sets of weights and biases that must be learned). Any function (such as softmax) applied to the final layer is ignored; none of these conventions will affect the asymptotic runtime. We also assume that the results of the forward pass of the network are cached, to be used for both vectorized backpropogation and computing the gradient with respect to each weight independenly. We can also ignore the biases; including them could be accomodated by adding an extra neuron to every layer (except the last) with a constant value of 1. This would be equivalent to increasing K by 1.

Denote each layer by the numbers $1, 2...N$ (we can refer to the input layer as layer 0). The efficiency of computing the gradient with respect to a weight to/from a particular layer does not depend on which weight it is, so w^a can stand for an arbitrary weight from layer $a-1$ to a. A particular neuron in layer n will be denoted by σ^n with a subscript to denote that one specific neuron in the layer is intended (the one which the weight under consideration leads to).

Then the following holds:

$$\frac{\partial C}{\partial w^N} = \frac{\partial C}{\partial \sigma_q^N}\frac{\partial \sigma_q^N}{\partial w^N}$$

$$\frac{\partial C}{\partial w^{N-1}} = \sum_q^K \frac{\partial C}{\partial \sigma_q^N}\frac{\partial \sigma_q^N}{\partial \sigma_m^{N-1}}\frac{\partial \sigma_m^{N-1}}{\partial w^{N-1}}$$

$$\frac{\partial C}{\partial w^{N-2}} = \sum_q^K \sum_m^K \frac{\partial C}{\partial \sigma_q^N}\frac{\partial \sigma_q^N}{\partial \sigma_m^{N-1}}\frac{\partial \sigma_m^{N-1}}{\partial \sigma_h^{N-2}}\frac{\partial \sigma_h^{N-2}}{\partial w^{N-2}}$$

Again, in each case, the derivative of the last neuron ($\sigma_q, \sigma_m, \sigma_h$ respectively in the equalities above) with respect to a weight leading to that layer refers to the neuron where that weight terminates. Since there are K neurons in each layer, each sum goes up to K.

With K neurons in each layer, there are K^2 weights between layers; thus the efficiency of finding $\frac{\partial C}{\partial w^N}$ for a specific weight is $O(1)$, but for all the weights is $O(K^2)$. For a specific weight into layer $N-1$, the efficiency would be $O(K)$; scaling this to all the weights is $O(K^3)$, since the results for other weights are not cached. Continuing this trend backward, the efficiency of finding the derivative of all the weights into the second hidden layer is $O(K^N)$ JUSTIFY IN MORE DETAIL. If we make the simplifying assumption that there are K input neurons, then the efficiency of finding all the derivatives

for the weights in the first layer is $O(K^{N+1})$. Taking only the highest order, gives an overall efficiency of $O(K^N)$.

The exponential dependence on N also follows intuitively - as adding N layers exponentially increases the number of paths from the cost to the weight, and to find the derivative, each of these paths must be followed.

For vectorized backpropogation, the derivative of the cost with respect to each layer can be computed using the cached values and matrix multiplication. Ignoring the complexity of carrying the derivative through the activation function, the derivative with respect to each weight between layers $N-1$ and N requires just one matrix multiplication. Moving backward through each layer adds one more matrix multiplication. With each layer having K neurons, these matrices have dimension K by K. The complexity of matrix multiplicaion is polynomial in their dimensions; multiplying two K by K matrices has complexity $O(K^3)$ (there are algorithms with slightly lower powers, but we'll take 3 as a baseline here). Therefore, to compute the gradient with respect to all the weights from the input to the first hidden layer is $O(NK^3)$, and if this is done one matrix at a time (i.e. propogating backwards one layer at a time), the derivative with respect to all the intermediate weights falls out at no extra cost (they are the result of already computed matrix multiplications). Thus the complexity of vectorized backpropogation with caching is $O(NK^3)$ for a network with N layers (excluding the input layer) each of K neurons.

SUMMARY