CSC321 A2 ; classifying handwritten digits using single and muti-layer neural networks

Maxwell Huang-Hobbs (g4rbage)

Part 1 - Description of Dataset

The dataset is the MNIST digit dataset, consisting of 1000 handwritten characters of each of [0,1,2,3,4,5,6,7,8,9] represented as 28x28 grayscale images.

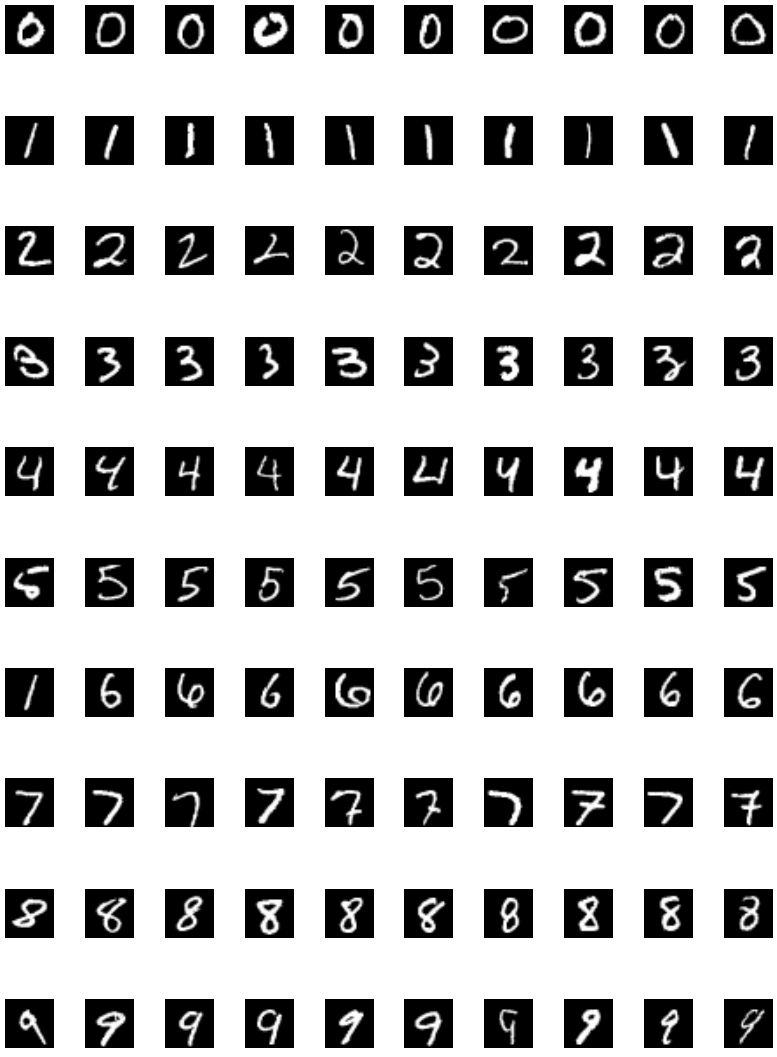

Figure 1: Examples of input data

Part 2 - Implementing the network

Not: The first half of this assignment was implemented by concatenating biases to the front of the weight matrix, concatenating a row of constants to the front of the input matrix

Additionally, N was used as the first matrix index

The network is implemented with the following function:

```
def linear_network(x, W):
    return softmax(dot(x, W))
```

Part 3 - Gradient of the Single Layer Network

The gradient of the network was implemented using the following function

```
def grad_neg_log_likelihood(inp, weights, targets):
    N, _ = inp.shape
    I, O = weights.shape

    probability = softmax(dot(inp.reshape((N, I)), weights)

     = (output)
    pdiffs = (probability - targets)
    pdiffs = tile(pdiffs.reshape((N, 1, O)), (1, I, 1))
    inp_expanded = inp.reshape((N, I, 1))
    inp_expanded = tile(inp_expanded, (1, 1, O))

    return pdiffs * inp_expanded
```

Part 4 - Verifying the Gradient of the Single Layer network.

The gradient of the network was approximated by changing the weight of each value in the weight matrix by a small step value (0.01) upwards and downwards, and calculating the slope of the cost over that difference

The difference between corresponding values in the approximate and calculated gradients is highly centered around 0, which would suggest that the gradient function is accurate.

The large tails on either side of the distribution could be explained by the approximation function stepping over abrupt changes in the cost function. This could be corrected for by using a smaller step function for approximating the gradient.

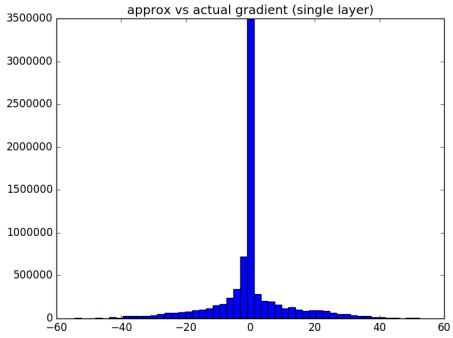

Figure 2: difference ($grad_{approx} - grad_{calc}$) between the approximate and calculated gradient for the linear network

Part 5 - Gradient Descent with the Single Layer Network

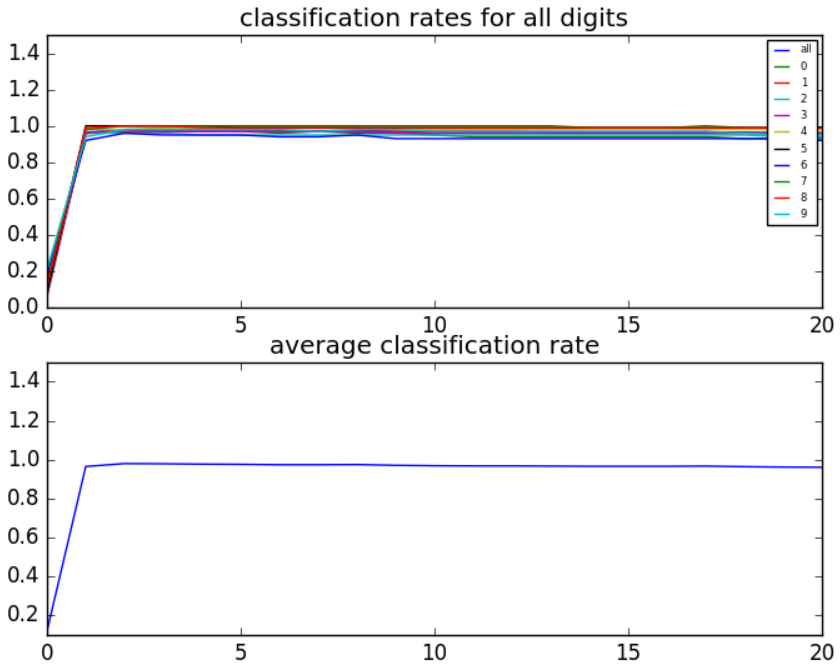

Figure 3: Learning rate for the single layer neural network versus generations of the network

The neural network was trained on the training set with the gradient function from Part 4 using the batch size of 50 samples from each digit (500 total), and a constant learning rate of 0.01.

The final neural network performed with 96.1% accuracy on the testing set.

The images the neural network failed to correctly classify are mostly either drawn with thin strokes, or skewed in some direction.

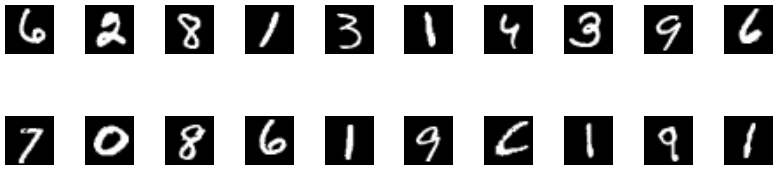

Figure 4: Examples of images the network correctly classifies

Figure 5: Examples of images the network fails to classify

Failure to classify digits with various stroke widths can be explained by the fact that the neural network directly maps each intensity on the input layer to some output on the output layer, so a thinly stroked digit would have low intensity in many of the highly-weighted outputs.

Failure to classify skewed digits could likewise be explained by the structure of the neural network.

Part 6 - Visualizing the weights of the inputs of the neural network

The weights of the inputs to the neural network look like blurred versions of their corresponding digits. They also seem to have some negative noise around the outside of the borders of each digit, which is likely weighting against the other digits.

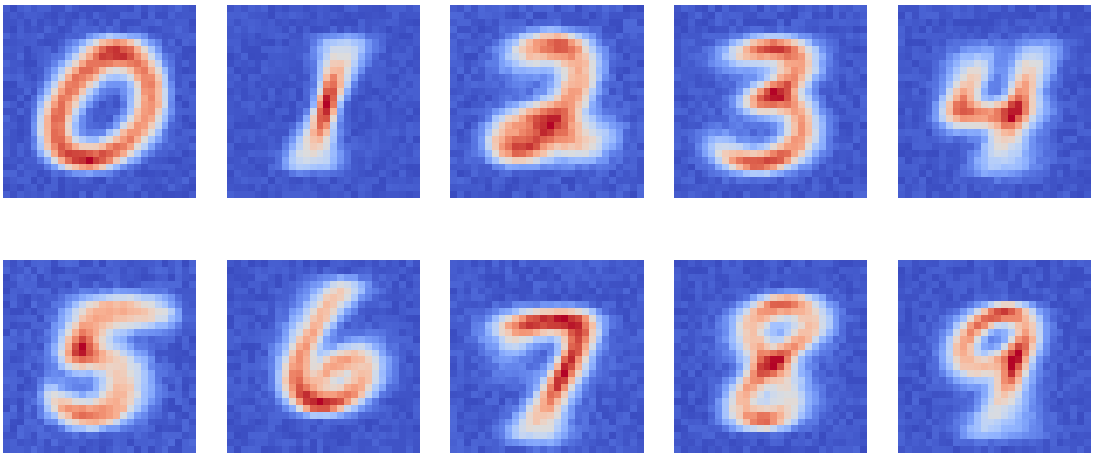

Figure 6: input weights of the neural network

Part 7 - Implementation & Explanation of the Gradient Function

The following is the implementation of the gradient function used in this project. Some matrix reshaping at the beginning is omitted (accounting for the shape of the data)

```
def dtanh(y):
    return 1.0 - (y** 2)

def grad_multilayer((W0, b0), (W1, b1), inp, expected_output):
    # convenience variables
    I, N = inp.shape
    H, O = W1.shape

    # run through the neural network
    L0, L1, prediction = forward(inp, (W0, b0), (W1,b1))

    # partial derivatives
    dCdL1 = (prediction - expected_output)
    dL1dL0 = dtanh(W1)
    dCdL0 = dot(dL1dL0, dCdL1)

    # derivative of layers
    dL1dW1 = dtanh(L0)  # L1 = tanh(dot(W1, L0))
    dL0dW0 = dtanh(inp) # L0 = tanh(dot(W0, inp)

    reshape and do a scalar multiplication instead of tiling and doing a dot product to avoid issues
    gradients_W1 = dCdL1.reshape((1, O, N)) * dL1dW1.reshape((H, 1, N))
    gradients_W0 = dCdL0.reshape((1, H, N)) * dL0dW0.reshape((I, 1, N))

    return (gradients_W0, dCdL0), (gradients_W1, dCdL1), prediction
```

Partial Derivatives

1. `dCdL1 = (prediction - expected_output)`
   $\frac{\delta C}{\delta L_1}$ = prediction − expected_output given in the slides on one-hot encoding
2. `dL1dL0 = dtanh(W1)`
   Let M be the output of the linear layer $L_0 \to W_1 \to M$
   $\frac{\delta L_1^i}{\delta L_0^j} = \frac{\delta L_1^i}{\delta M}\frac{\delta M}{\delta L0}$
   $\frac{\delta M_1^i}{\delta L_0^j} M_1^i = \frac{\delta M_1^i}{\delta L_0^j} \sum_J W_1^{i,J} L_0^j = (W_1^{i,0} * 0 + .. + W_1^{i,j} + .. + W_1^{i,300} * 0) = W_1^{i,j}$
   $\frac{\delta L_1^i}{\delta M_1^i} = \frac{\delta}{\delta M_1^i} tanh(M_1^i) = dtanh(M_1^i)$
   Combining these partial derivatives we get,
   $\frac{\delta L_1^i}{\delta L_0^j} = dtanh(M_1^i) W_1^{i,j}$
   TODO FIX IN CODE
3. `dCdL0 = dot(dL1dL0, dCdL1)`
   By the chain rule, and parts (1.) and (2.)

Derivatives of Layers

1. `dL1dW1 = dtanh(L0)`

   $L_1^i = tanh(\sum_J (W_1^{i,j}, L_0^j))$ $\frac{\delta}{\delta W_1^{i,j}} L_1^i = \frac{\delta L_1^i}{\delta M^i} \frac{\delta M^i}{\delta W_1^{i,j}}$

   $\frac{\delta L_1^i}{\delta W_1^{i,j}} = dtanh(M^i) \frac{\delta M^i}{\delta W_1^{i,j}} \sum_J W_1^{i,J} * L_0^J$

   $\frac{\delta L_1^i}{\delta W_1^{i,j}} = dtanh(M^i)(0 * L_0^0 + .. + L_0^j + .. + 0 * L_0^3 00)$

   $\frac{\delta L_1^i}{\delta W_1^{i,j}} = dtanh(M^i) L_0^j$

   TODO fix in code

2. `dL0dW0 = dtanh(inp)`

   By the same process as (1.),

   $\frac{\delta L_0^i}{\delta W_0^{i,j}} = dtanh(M) inp^j$

   TODO fix in code

3. `gradients_W1 = dCdL1.reshape((1, O, N)) * dL1dW1.reshape((H, 1, N))`
   `gradients_W0 = dCdL0.reshape((1, H, N)) * dL0dW0.reshape((I, 1, N))`

   By the chain rule, $\frac{C}{\delta W_1} = \frac{\delta C}{\delta L_1} \frac{\delta L_1}{\delta W_1}$

   $\frac{C}{\delta W_0} = \frac{\delta C}{\delta L_0} \frac{\delta L_0}{\delta W_0}$

   A multiplication between matrices of mismatched dimensions broadcasts the arrays accross each other, which is equivalent to tiling and doing a dot product between the arrays

   TODO does this even work?

Part 8 - Approximation to the Gradient

The difference between the actual and expected gradients was mostly promising (centered around 0, low spread). However, the implementation of the was likely inaccurate in some areas, as the distribution of the gradients of the offset matrix b_1 was unpatterned and widely spread.

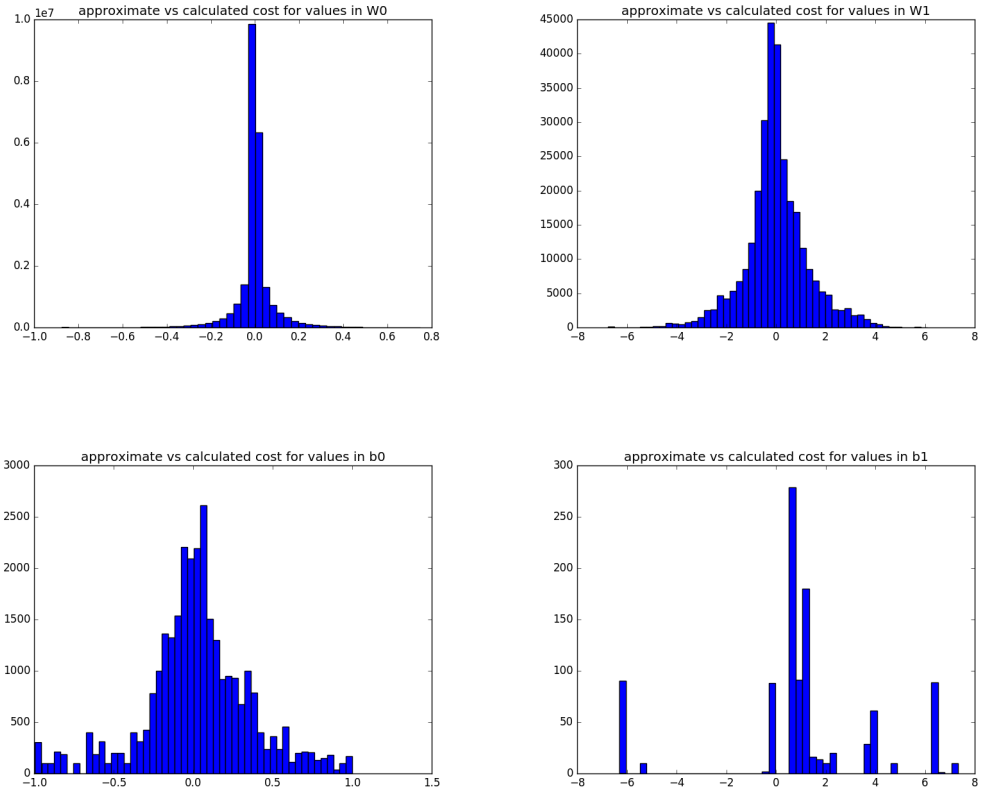

Figure 7: difference approximated - calculated for all values in the gradients of each of W_0 (top left), W_1 (top right), b_0(bottom left), and b_1(bottom right)

Part 9 - Training with the multilayer gradient

As one would expect with a partially incorrect implementation of the gradient function, the neural network did not perform well.

It hovers briefly around 44% accuracy, before quickly falling off to around 10% accuracy. This would suggest that the network is memorizing the features of only one of the classes of digits, and always reporting the digit to b that output

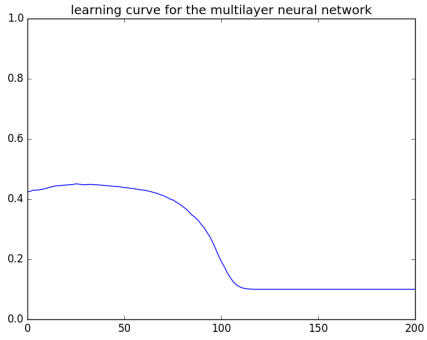

Figure 8: Learning curve of the multilayer neural network

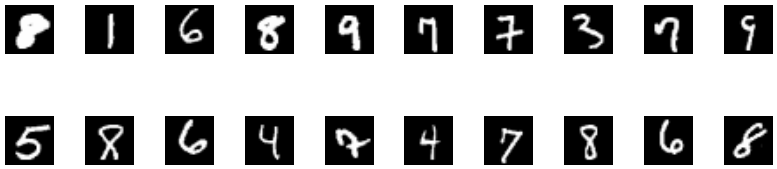

Figure 9: Examples of images the network fails to classify

Figure 10: Examples of images the network fails to classify

Part 10 - Visualizing the input layer of the Neural Network

The input layer to the multilayer neural network are mostly just random noise, though some of them appear to be fitting to parts of different digits

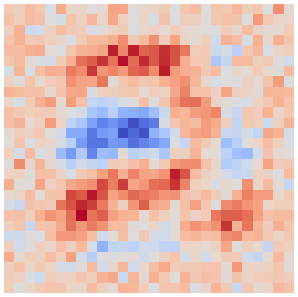

Figure 11: A slice of the first layer in the neural network

This slice of the input layer seems to be fitting to the character 2, and against the space immediately around 2

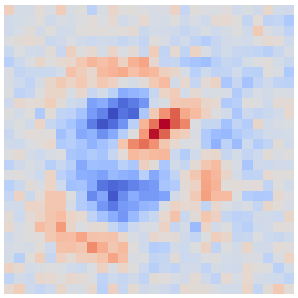

Figure 12: Another slice of the first layer in the neural network

This slice of the input layer seems to be fitting the areas guaranteed to be in the character 3, and against other areas of the image. The break in the layer is possibly because that stroke in the character 3 is often varied