CSC411: Assignment #1

Due on Friday, January 29, 2018

Lukas Zhornyak

February 23, 2018

Environment

Parts 1-6 were created with Python 2.7.14 with numpy 1.14.0, scipy 1.0.0, scikit-image 0.13.1, and matplotlib 2.1.1, as well as all associated dependencies. Parts 8-10 were created with Python 3.6.4 with numpy 1.14.1, scipy 1.0.0, scikit-image 0.13.1, and matplotlib 2.1.2, pytorch 0.3.0, torchvision 0.2.0, as well as all associated dependencies.

Part 1

The dataset, provided in assignment, is already split into training and validation sets. About six thousand images are provided in the training set for each digit (ranging from 5421 to 6742) while about one thousand images are provided in the testing set (ranging from 892 to 1135), for a total of about seven thousand images in the combined set for each digit. In total, there are sixty thousand training images and ten thousand testing images. Ten randomly selected images from the combined training and testing sets are shown in fig. 1.

Due to the way the images are selected and displayed, any images selected that were from the validation set would be shown on the second row and right side of the set of images for each digit shown in fig. 1. As cab be seen, there does not seem to be any major systematic variation in the images from the validation set compared to the images from the training set. This is to be expected and is necessary for the proper training of the classifier.

The images provided are typical of digits written quickly and without specific care to how they appear; in other words, they represent digits how they would likely appear in an arbitrary handwritten sample. The digits do not seem specifically drawn for clarity. While the positioning and the size remain approximately consistent between digits, the geometry of the digits is not completely consistent. For example, some two's are written with a loop while other's are not. Additionally, the slant and stroke thickness of the digit varies considerably.

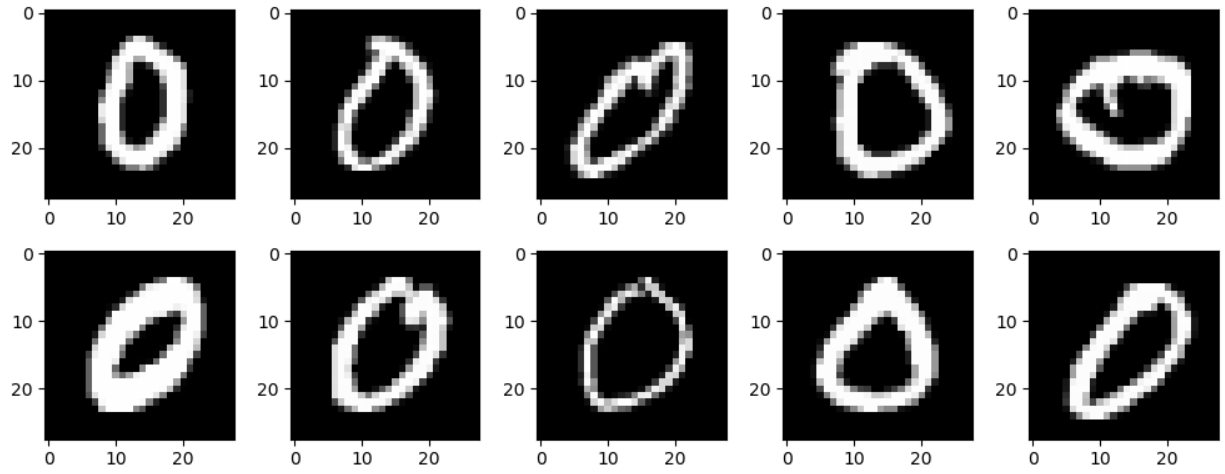

Figure 1: Random sample of each digit selected from the complete dataset.

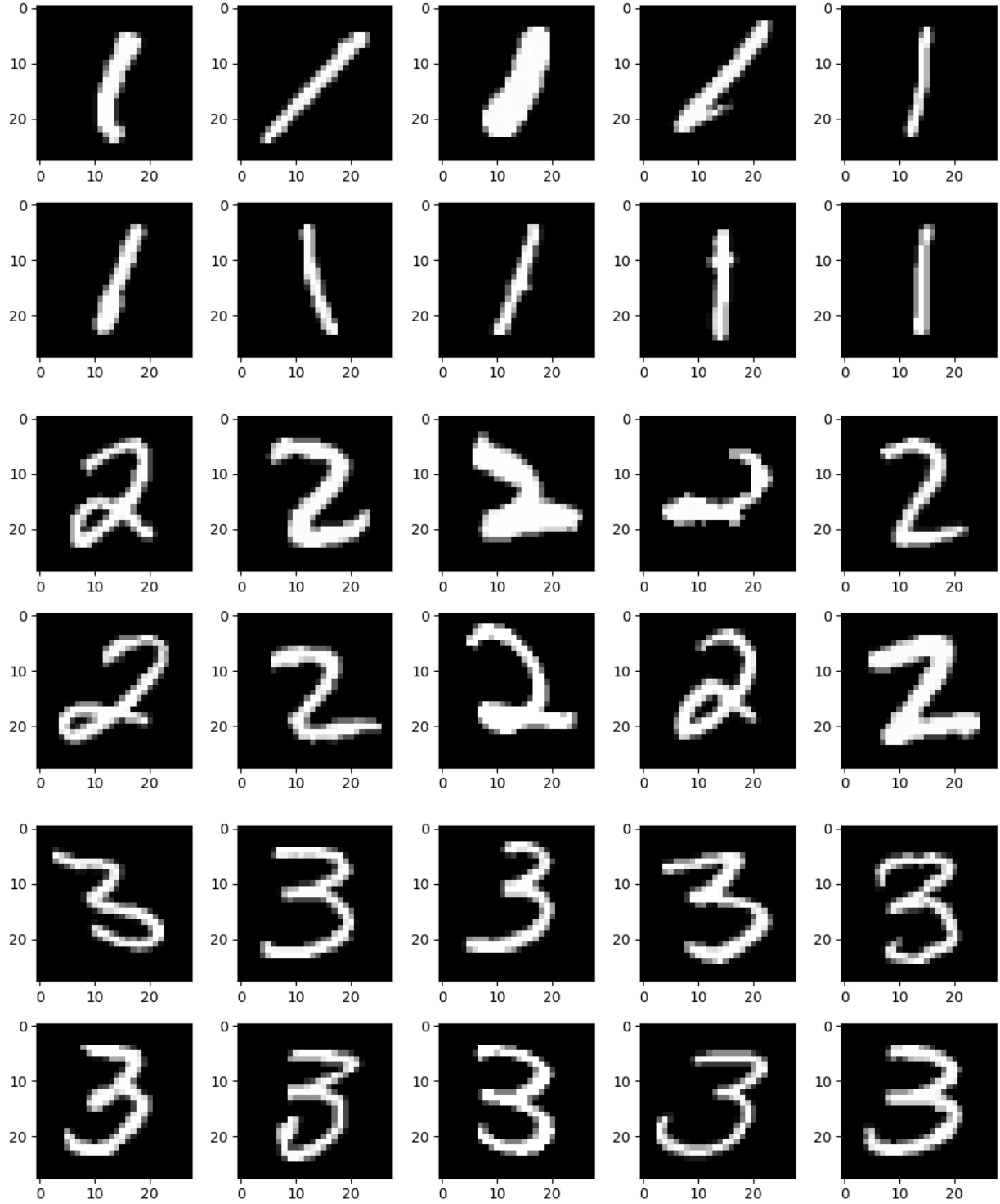

Figure 1: *Cont.* Random sample of each digit selected from the complete dataset.

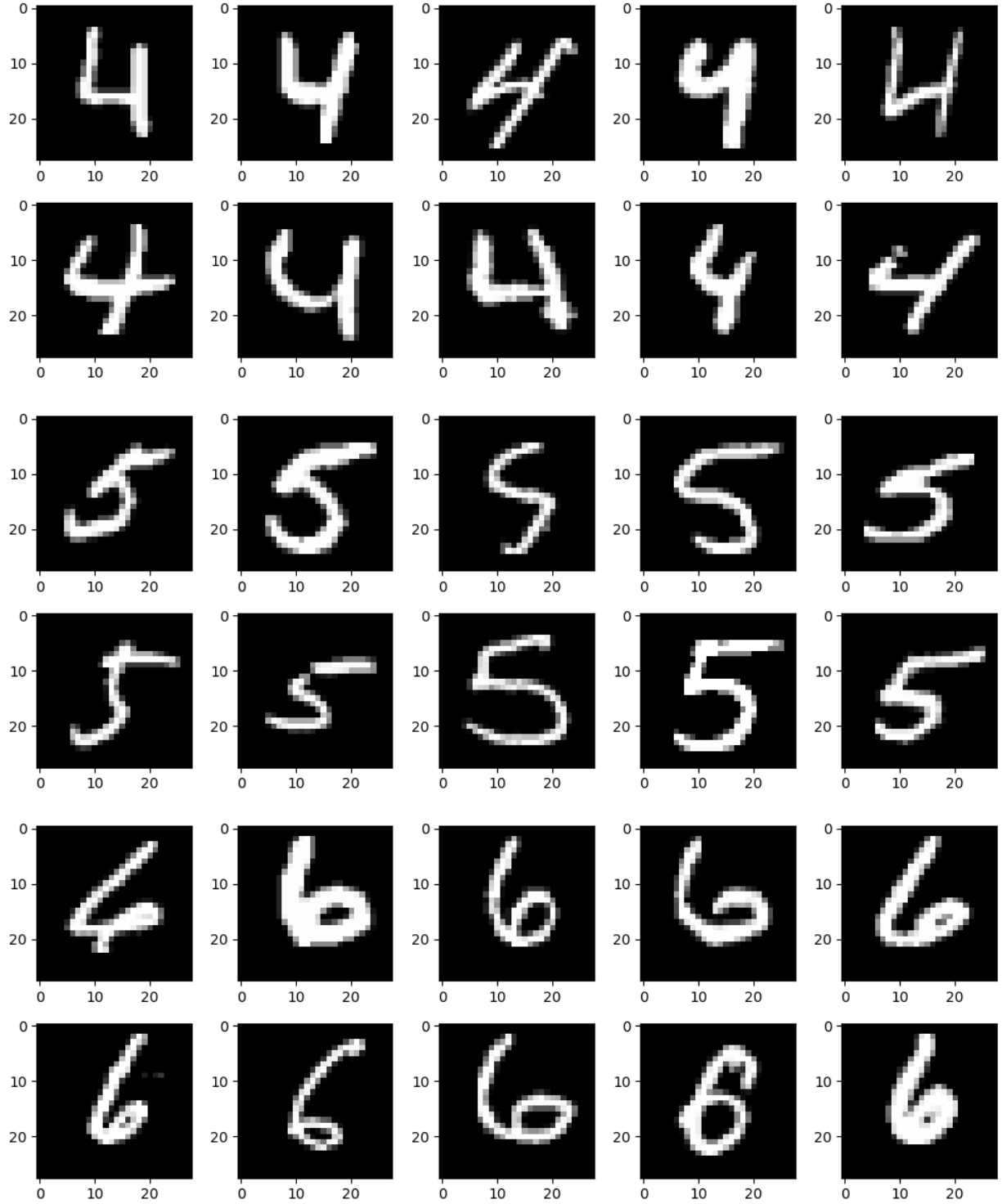

Figure 1: *Cont.* Random sample of each digit selected from the complete dataset.

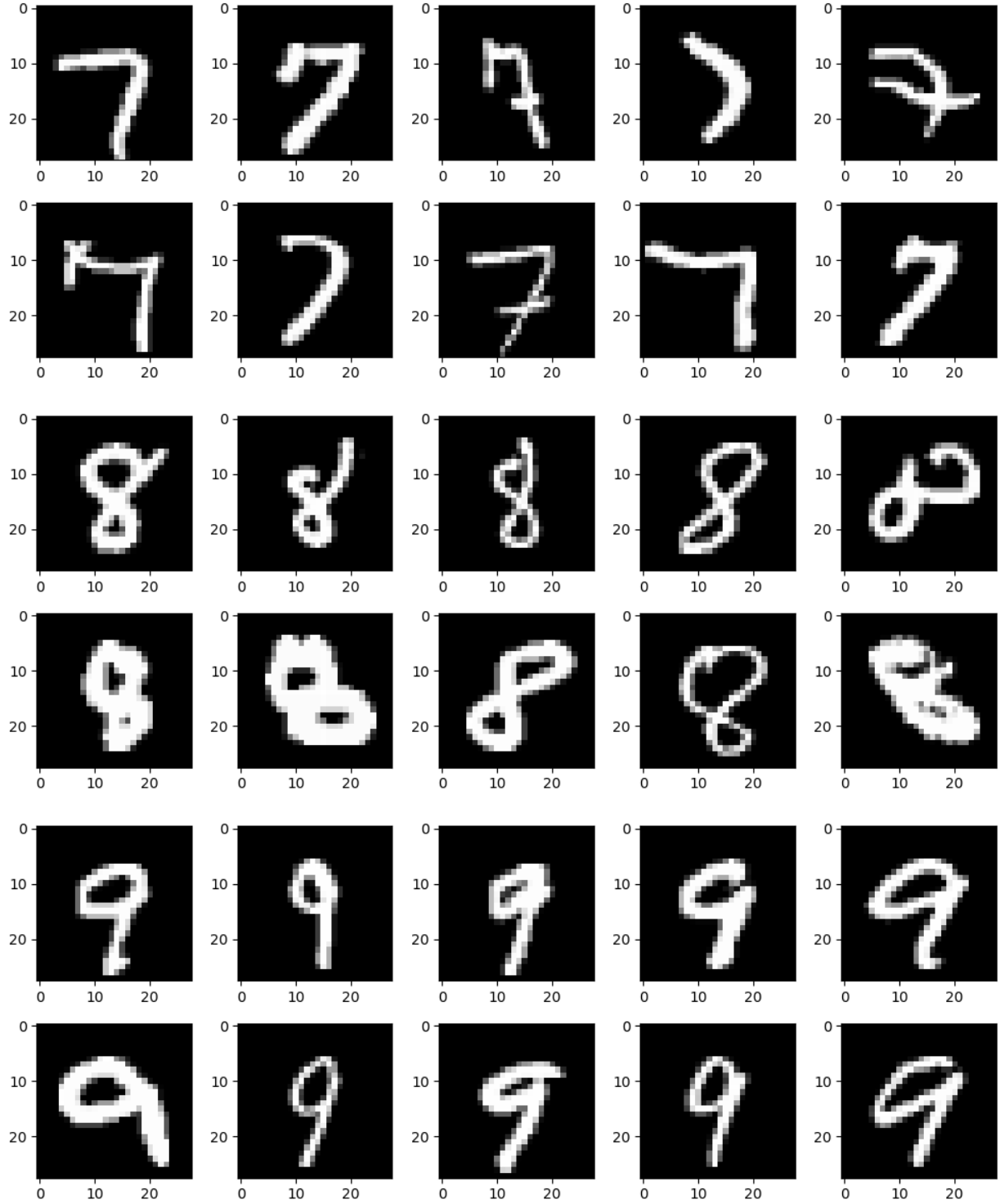

Figure 1: *Cont.* Random sample of each digit selected from the complete dataset.

Part 2

With the provided softmax function:

```
def basic_network(W, x, b):
    return softmax(np.dot(W, x) + b)
```

Part 3

3 (a)

We want to find $\frac{\partial C}{\partial \omega_{ij}}$ the derivative of the weight ω_{ij} associated with the connection from the j-th input node x_j to the i-th output node o_i, before the softmax. The derivative can thus be expanded via the chain rule as

$$\frac{\partial C}{\partial \omega_{ij}} = \sum_k \frac{\partial C}{\partial o_i^{(k)}} \frac{\partial o_i^{(k)}}{\partial \omega_{ij}} \tag{1}$$

Where the superscript denotes the k-th member of the set. Since ω_{ij} only influences the value of o_i, the derivatives of other output nodes with respect to ω_{ij} will be zero and are thus not included in the above equation. Each of these derivatives can now be found separately. The second can be found trivially since we assume an identity activation function, giving.

$$\frac{\partial o_i^{(k)}}{\partial \omega_{ij}} = x_j^{(k)} \tag{2}$$

Examining first part of the derivative of the cost function:

$$\begin{aligned} C &= -\sum_k \sum_\ell y_\ell^{(k)} \log\left(p_\ell^{(k)}\right) \\ \Rightarrow \quad \frac{\partial C}{\partial o_i^{(k)}} &= -\sum_k \sum_\ell y_\ell^{(k)} \frac{\partial \log\left(p_\ell^{(k)}\right)}{\partial o_i^{(k)}} \end{aligned} \tag{3}$$

Where p is the output after the softmax and y is the actual label. Dropping the superscript for clarity,

$$\begin{aligned} \log(p_\ell) &= \log\left(\frac{\exp(o_\ell)}{\sum_m \exp(o_m)}\right) \\ &= o_\ell - \log\left(\sum_m \exp(o_m)\right) \\ \Rightarrow \quad \frac{\partial \log(p_\ell)}{\partial o_i} &= \frac{\partial o_\ell}{\partial o_i} - \frac{\exp(o_i)}{\sum_m \exp(o_m)} \\ &= \begin{cases} 1 - p_i & i = j \\ -p & i \neq j \end{cases} \end{aligned} \tag{4}$$

Subbing eq. (4) into eq. (3) thus produces:

$$\begin{aligned} \frac{\partial C}{\partial o_i^{(k)}} &= \sum_k \left(\left(\sum_\ell y_\ell^{(k)} p_i^{(k)} \right) - y_i^{(k)} \right) \\ &= \sum_k \left(p_i^{(k)} - y_i^{(k)} \right) \end{aligned} \tag{5}$$

Where one-hot encoding was assumed, resulting in $\sum_\ell y_\ell^{(k)} = 1$. Substituting eqs. (2) and (5) into eq. (1) produces the complete expression for the gradient.

$$\frac{\partial C}{\partial \omega_{ij}} = \sum_k \left(p_i^{(k)} - y_i^{(k)} \right) x_j^{(k)} \tag{6}$$

This can also be expressed in vector form as

$$\frac{\partial C}{\partial W} = \sum_k \left(p^{(k)} - y^{(k)} \right) \left(x^{(k)} \right)^\mathsf{T} \tag{7}$$

3 (b)

The code used to compute the gradient with respect to both the weights and the biases is shown below. Note that all quantities are transposed compared to the formula shown in eq. (7), hence the small differences.

```
1 def basic_gradient(W, x, b, y):
2     p = basic_network(W, x, b)
3     return np.dot(x.T, p - y), np.sum(p - y, 0)
```

Part 4

The learning curve after ten thousand generations of stochastic gradient descent is shown in fig. 2 and a visualization of the resultant weights is shown in fig. 3. For this optimization, the training sets for each individual digit were combined into one large set and an additional array with the corresponding labels was created.

The initial weights were initialized using Xavier Initialization. Each weight was thus sampled from a normal distribution centred at zero and with a variance of $\frac{1}{140}$, corresponding to $\frac{2}{n_{in}+n_{out}}$. The weights were trained on mini batches of size 1000, selected through trial and error as a compromise between speed and stability The learning rate was selected as 1×10^{-5} as this was the largest rate (to an order of magnitude) that did not result in the runaway growth of the weights.

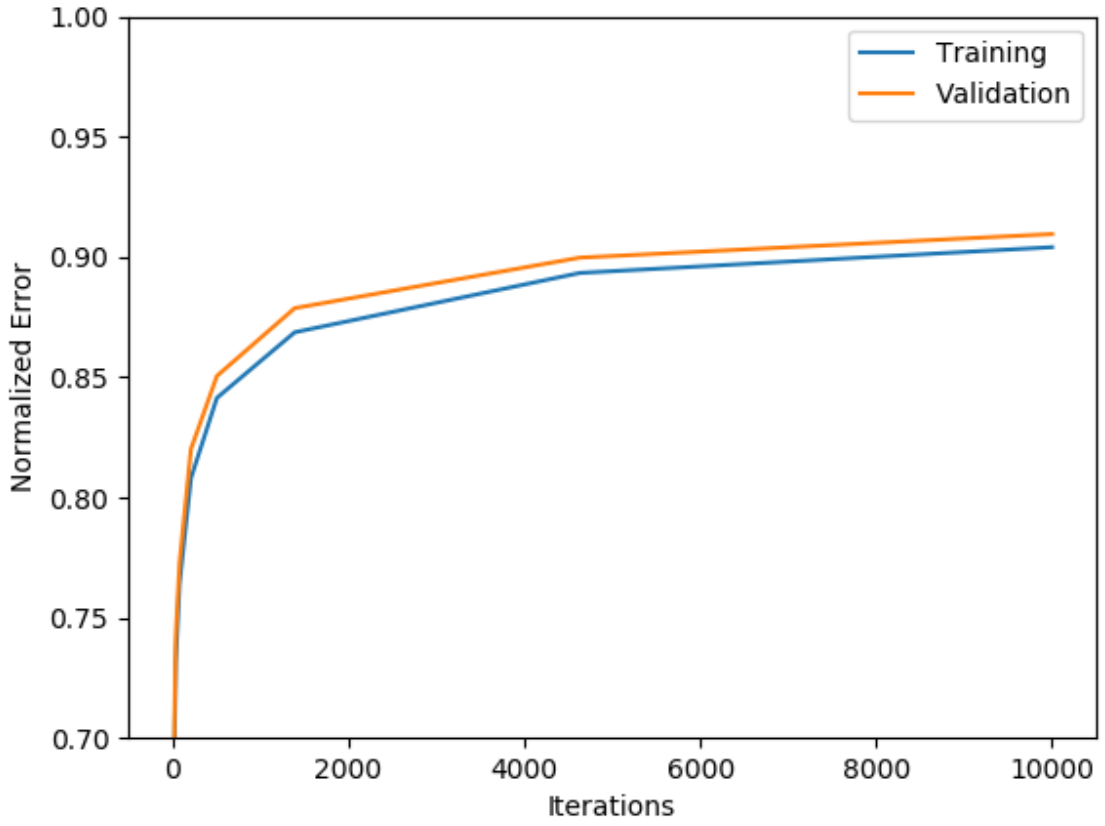

Figure 2: Performance on the provided training and verification sets after ten thousand generations of training on the training set using stochastic gradient descent. Note that "Normalized Error" should read accuracy.

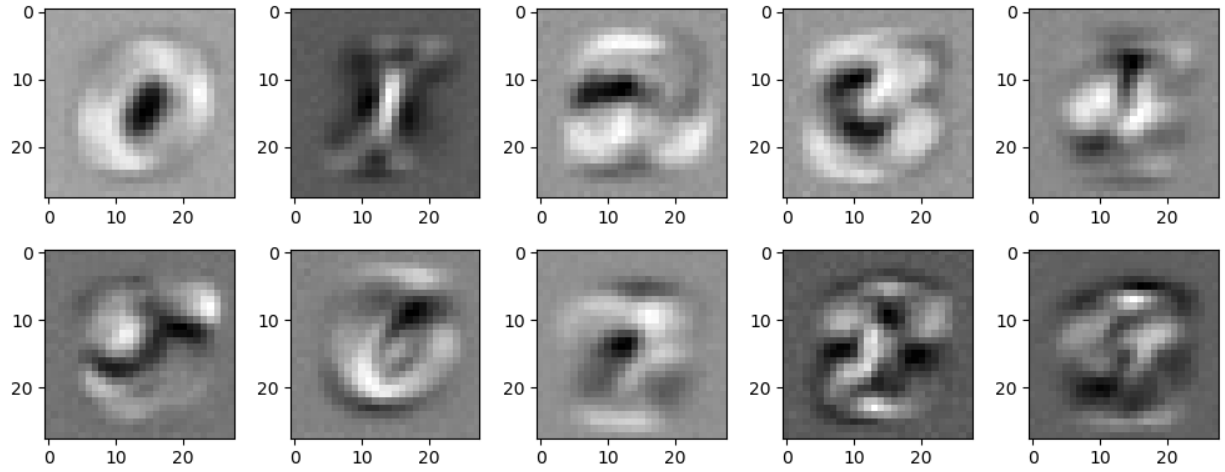

Figure 3: Visualization of the weights going into each output node after stochastic gradient descent. Visualization is constructed so that the location of each weight corresponds to the location of the input node that it corresponds to. Note the resemblance to generic numerals.

Part 5

The function used to perform gradient descent with momentum is given below. Note that several lines related to recording and plotting the error and progress are removed for brevities sake. The resulting learning curve and weights after 10 000 (the same as in problem 4) are shown in figs. 4 and 5, respectively.

```
def gradient_descent(W, x, b, y, xv, yv, learning_rate=1e-5, epsilon=1e-8,
    max_iter=1000, momentum=0.9, save_file=None):
    W = W.copy()  # ensure that passed in value not changed
    b = b.copy()
    last = np.zeros_like(W)
    i = 0
    dW, db = 0, 0
    while np.linalg.norm(W - last) > epsilon and i < max_iter:
        last = W.copy()
        dW_n, db_n = basic_gradient(W, x, b, y)
        dW = dW * momentum - dW_n * learning_rate
        db = db * momentum - db_n * learning_rate
        W += dW
        b += db
        i += 1
    return W, b
```

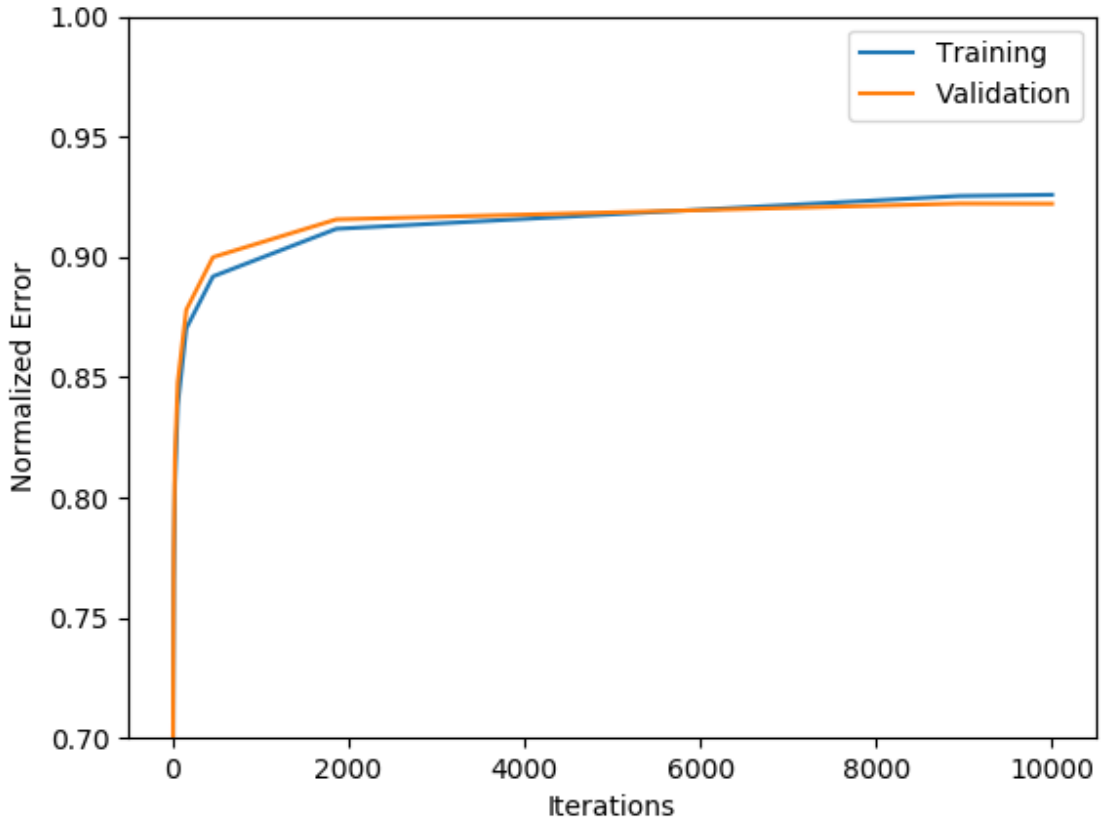

Figure 4: Performance on the provided training and verification sets after ten thousand generations of training on the training set using stochastic gradient descent with momentum.

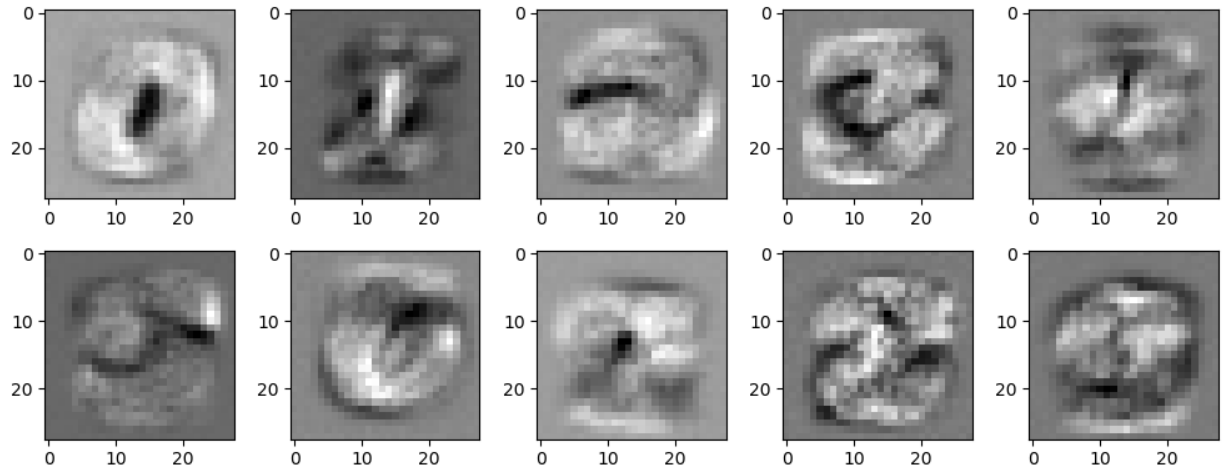

Figure 5: Visualization of the weights going into each output node after stochastic gradient descent with momentum. Visualization is constructed so that the location of each weight corresponds to the location of the input node that it corresponds to. Note the resemblance to generic numerals. Note that "Normalized Error" should read "Accuracy".

Part 6

The contour plot and trajectories of the two weights are shown in fig. 6. This figure shows that using momentum properly will allow for faster convergence than without. Specifically, the use of momentum significantly reduces the oscillations associated with going down a valley as one bounces from one wall to another while at the same time increases the rate that one moves down the valley. This is because the momentum term, in a sense, pushes the gradient descent to keep moving in the same direction that it was going. This compounds the effects of gradients pointing in similar direction while reducing the effects of gradients pointing in opposite directions.

To produce these visualization, the pixels located at (13, 13) in the "0" weights and the "1" weights were chosen as w_1 and w_2, respectively. A pixel near the centre was chosen as the weights associated with it would be necessary to distinguish numbers, producing large enough gradients. Additionally, to show the effects of momentum a contour that is considerably narrower one way than the other is also desirable as such situations are where the momentum term really shows its effect. A learning rate of 6×10^{-3} was selected through trial and error and used for both types of gradient descent to highlight the differences of just the momentum term.

Figure 7 shows an identical figure to that shown in fig. 6, except the momentum has been set to 0.8 rather than 0.25. This higher momentum means that the gradient descent overshoots the actual minimum and spends longer circling around the minimum. This happens because the large momentum term means that the descent is less able to react to sharp changes in the slope of the gradient, the momentum term acting as a form of exponential smoothing.

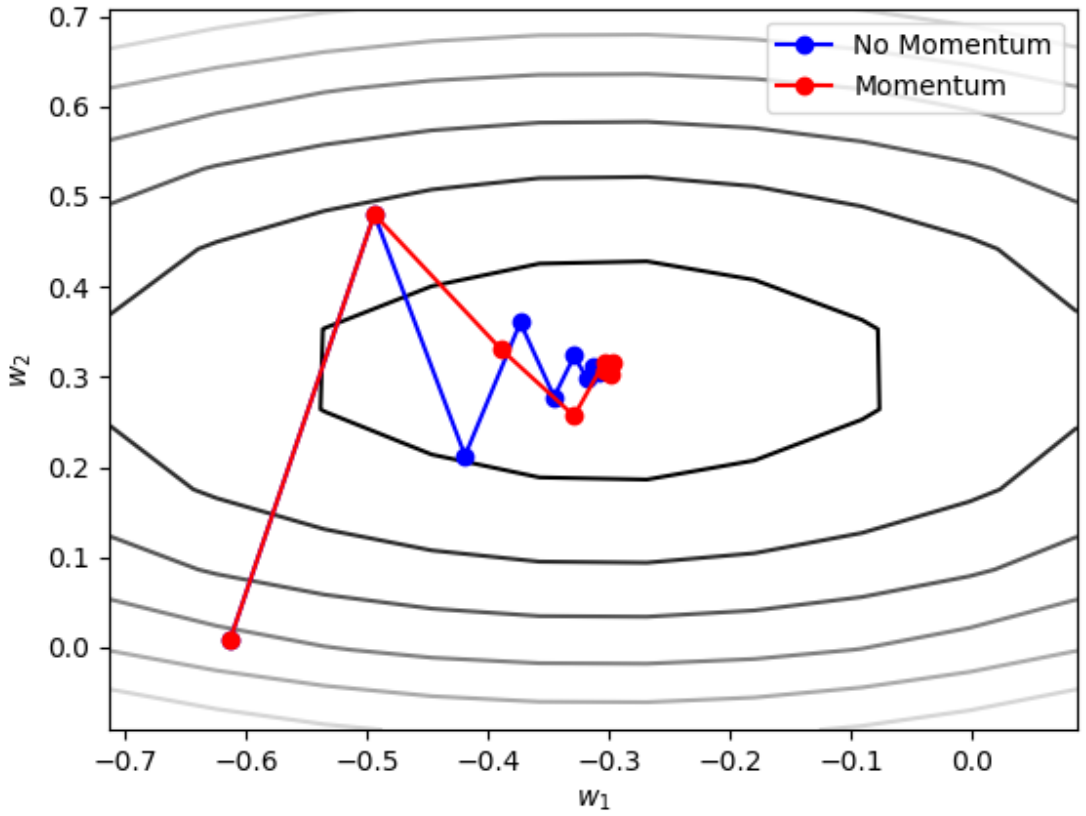

Figure 6: Visualization of the path of the gradient descent without momentum and with a momentum of 0.25.

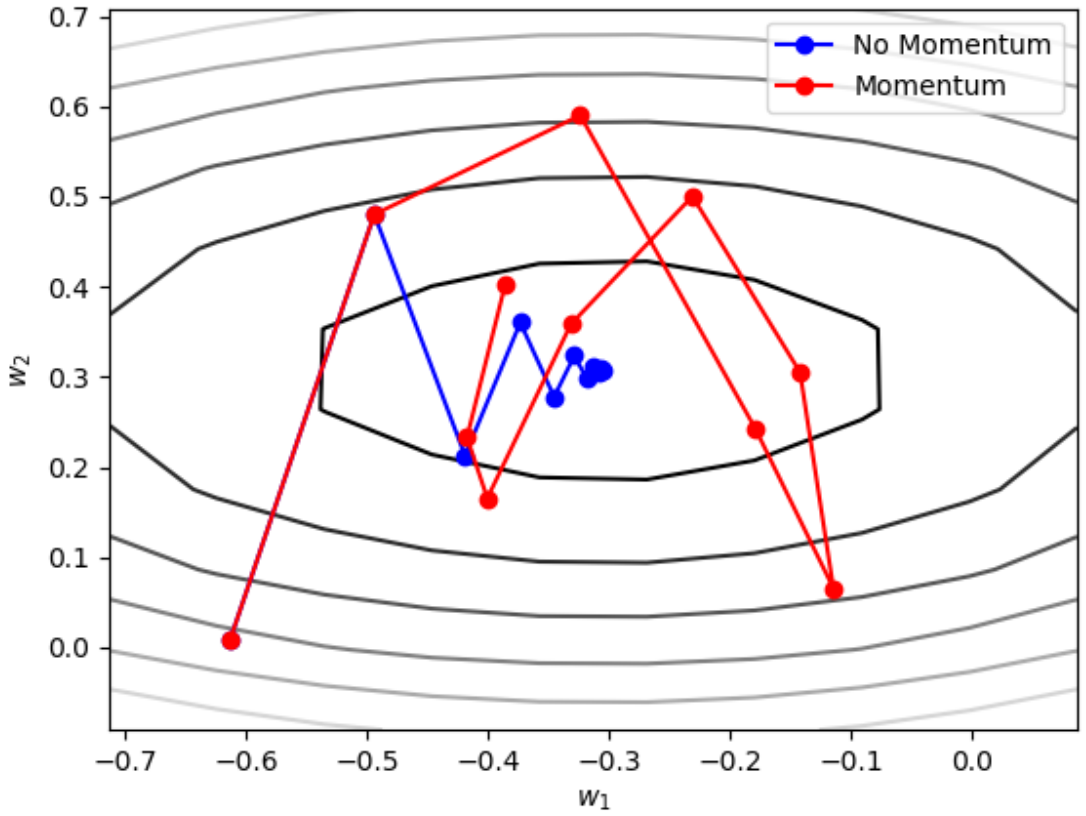

Figure 7: Visualization of the path of the gradient descent without momentum and with a momentum of 0.8.

Part 7

Let $y_i^{(\ell)}$ be the i-th node in the *ell*-th layer, where $y^{(1)}$ is the input layer and $y^{(N)}$ is the output layer for a total of N layers with K nodes in each layer. Let $\omega_{ij}^{(\ell)}$ be the weight associated with the connection of node j in the layer ℓ to node i of the layer $\ell+1$. Then, the formula expressing one layer in terms of the previous, for $\ell > 1$ and assuming no biases, can be expressed as

$$y_i^{(\ell)} = a\left(\sum_{j=1}^{K} \omega_{ij}^{(\ell-1)} y_i^{(\ell-1)}\right)$$

where a is the activate function. The derivative of $y_i^{(\ell)}$ with respect to a weight connecting a node in the previous layer to it can thus be expressed as

$$\frac{\partial y_i^{(\ell)}}{\partial \omega_{im}^{(\ell-1)}} = a'\Big(y_i^{(\ell)}\Big) y_m^{(\ell)} \tag{8}$$

Similarly, the derivative of $y_i^{(\ell)}$ with respect to the node m in layer p, $p < \ell$, can be expressed as

$$\frac{\partial y_i^{(\ell)}}{\partial y_m^{(p)}} = a'\Big(y_i^{(\ell)}\Big)\left(\sum_{j=1}^{K} \omega_{ij}^{(\ell-1)} \frac{\partial y_j^{(\ell-1)}}{\partial y_m^{(p)}}\right) \tag{9}$$

requiring the computation of an additional K derivatives.

Assume that basic multiplication takes time t_m, addition takes time t_+, find $a'(y)$ for some y takes time t_a, and finding the derivative of the cost function C with respect to one of the output nodes takes time t_C. Additionally, assume all values $y_i^{(\ell)}$, produced as a result of forward propagation, are cached. Then, the cost of eq. (8) can be expressed as

$$T\left(\frac{\partial y_i^{(\ell)}}{\partial \omega_{im}^{(\ell-1)}}\right) = t_a + t_m$$

while the cost of eq. (9) can be expressed as, assuming $K \gg 1$ and $N \gg 1$,

$$\begin{aligned}
T\left(\frac{\partial y_i^{(\ell)}}{\partial y_m^{(p)}}\right) &= t_a + t_m + (K-1)t_+ + \sum_{j=1}^{K}\left(t_m + T\left(\frac{\partial y_j^{(\ell-1)}}{\partial y_m^{(p)}}\right)\right) \\
&= t_a + (K+1)t_m + (K-1)t_+ + \sum_{j=1}^{K} T\left(\frac{\partial y_j^{(\ell-1)}}{\partial y_m^{(p)}}\right) \\
&= \sum_{q=1}^{\ell-p} K^{q-1}(t_a + (K+1)t_m + (K-1)t_+) \\
&= (t_a + (K+1)t_m + (K-1)t_+)\frac{1-K^{(\ell-p)}}{1-K} \\
&= O\big(K^{\ell-p}\big)
\end{aligned}$$

Combining all this together and assuming that $N > p > 0$, the time taken for the derivative of the cost function with respect to a single weight is

$$\begin{aligned}
T\Big(\frac{\partial C}{\partial \omega_{nm}^{(p)}}\Big) &= T\left(\sum_{i=1}^{K} \frac{\partial C}{\partial y_i^{(N)}} \frac{\partial y_i^{(N)}}{\partial y_n^{(p+1)}} \frac{\partial y_n^{(p+1)}}{\partial \omega_{nm}^{(p)}}\right) \\
&= K\big(t_c + t_a + 3t_m + O\big(K^{N-p-1}\big)\big) + (K-1)t_+ \\
&= O\big(K^{N-p}\big)
\end{aligned}$$

The total cost to compute the gradients without cached backpropagation is thus

$$\begin{aligned} T\left(\frac{\partial C}{\partial W}\right) &= \sum_{i=1}^{N-1}\sum_{j=1}^{K}\sum_{n=1}^{K} O\big(K^{N-i}\big) \\ &= \sum_{i=1}^{N-1} O\big(K^{N-i+2}\big) \\ &= O\left(K^{N+1}\sum_{i=0}^{N-2} K^{-i}\right) \\ &= O\big(K^{N+1}\big) \end{aligned} \tag{10}$$

However, through the use of backpropagation and caching, this value can be significantly reduced. First, note that the derivative of the cost function with respect to a given weight can be expressed as

$$\begin{aligned} \frac{\partial C}{\partial \omega_{nm}^{(p)}} &= \sum_{i=1}^{K} \frac{\partial C}{\partial y_i^{(p+2)}} \frac{\partial y_i^{(p+2)}}{\partial y_n^{(p+1)}} \frac{\partial y_n^{(p+1)}}{\partial \omega_{nm}^{(p)}} \\ &= \left(\sum_{i=1}^{K} \frac{\partial C}{\partial y_i^{(p+2)}} \omega_{in}^{(p+1)}\right) \frac{\partial y_n^{(p+1)}}{\partial \omega_{nm}^{(p)}} \end{aligned}$$

if $p+1 < N$. If the quantity $\sum_{i=1}^{K} \frac{\partial C}{\partial y_i^{(p+2)}} \omega_{in}^{(p+1)} = \frac{\partial C}{\partial y_n^{(p+1)}}$ is cached, it can then be used to speed up the calculation of the next layer. Thus, the cost to compute an individual gradient (assuming the previous result is cached) reduces to

$$\begin{aligned} T\left(\frac{\partial C}{\partial \omega_{nm}^{(p)}}\right) &= T\left(\left(\sum_{i=1}^{K} \frac{\partial C}{\partial y_i^{(p+2)}} \omega_{in}^{(p+1)}\right) \frac{\partial y_n^{(p+1)}}{\partial \omega_{nm}^{(p)}}\right) \\ &= t_a + (K+2)t_m + (K-1)t_+ \\ &= O(K) \end{aligned}$$

while the cost to compute the full gradient is

$$\begin{aligned} T\left(\frac{\partial C}{\partial W}\right) &= \sum_{i=1}^{N-1}\sum_{n=1}^{K}\sum_{m=1}^{K} O(K) \\ &= O(NK^3) \end{aligned} \tag{11}$$

Comparing eqs. (10) and (11), the benefits of using backpropagation are clear. The time to find all the weights individually shows polynomial growth to a large order $(N+1)$ with respect to the number of nodes in a layer and *exponential* growth with respect to the number of layers. Meanwhile, backpropagation shows quadratic growth with respect to the number of nodes in a layer and only linear growth with respect to the number of layers. In other words, backpropagation is $O(K^{N-3}/N)$ times faster than finding the weights individually.

Part 8

The general structure of the network used takes in a flattened black and white image of some size, gets linearly transformed then has a non linear transfer function applied to reach the hidden layer, then gets linearly transformed again to produce the six outputs the result in one hot encoding. The weights for the network are initialised using Xavier initialization. The images used were downloaded from the internet using the same function as in the last assignment, checking for valid hashes and bounding boxes to ensure that only faces were captured. Unfortunately, not enough Peri Gilpin images were downloaded this way, so the size of the associated sets were scaled down appropriately resulting in 54 training images, 16 validation images, and 16 testing images. The remaining actors were tested with 64 training images, 20 validation images, and 20 testing images.

The hyperparameters varied in this model are the image size, size of the hidden layer, number of epochs, batch size, activation function, and initial learning rate for the optimizer. A grid search was performed on these parameters, with the combination producing the highest accuracy on the validation set selected as the optimal set. The parameters were selected from the following, chosen to represented the likely parameters based on previously conducted trial and error:

$$\begin{aligned}
\text{Image Size} &\in \{32, 64\} \\
\text{Size of Hidden Layer} &\in \{10, 20, 50\} \\
\text{Number of Epochs} &\in \{200, 300, 500\} \\
\text{Batch Size} &\in \{20, 50, 100\} \\
\text{Activation Function} &\in \{ReLU, LeakyReLU(0.01)\} \\
\text{Initial Learning Rate} &\in \{5 \times 10^{-4}, 1 \times 10^{-3}\}
\end{aligned}$$

A more thorough examination would examine more parameters.

An image size of 64×64 pixels with a hidden layer of twenty units, initial learning rate of 1×10^{-3}, batch size of one hundred, and using the LeakyReLU activation function with negative slope of 0.01 optimized over two hundred epochs produced the best result, achieving a classification accuracy on the training set of 0.7759. Note that this set was not victorious by a significant margin, and its superiority may simply be due to how the random numbers were generated. Other sets should be examined in more detail for a more thorough analysis. The learning curve generated with the selected hyperparameters is shown in fig. 8.

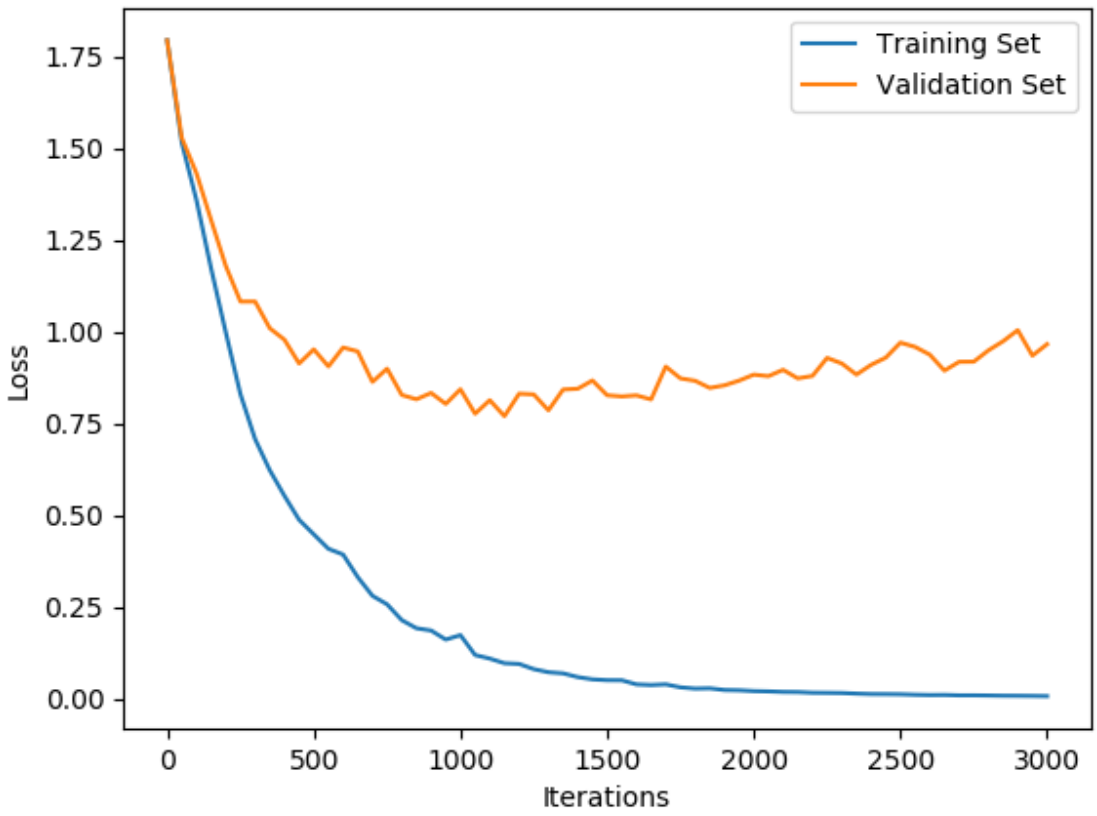

Figure 8: Learning curve for the training and validation set with an image size of 64×64 pixels with a hidden layer of twenty units, initial learning rate of 1×10^{-3}, batch size of 30, and using the ReLU activation function optimized over 3000 iterations.

Part 9

The visualization of the weights important for classifying an image as either Alec Baldwin or Angie Harmon are shown in fig. 9. The group were selected by first finding the average activation of the hidden layer across a 100 images of each person. These activations were then multiplied with the weights feeding into the output associated with each person to obtain the contribution of each hidden unit to the classification of person into a category. The percentage difference between the contribution of each hidden unit to the classification of an image as one person or the other was found, and those hidden units with the largest percentage difference were selected as the most important for the classification and are shown below.

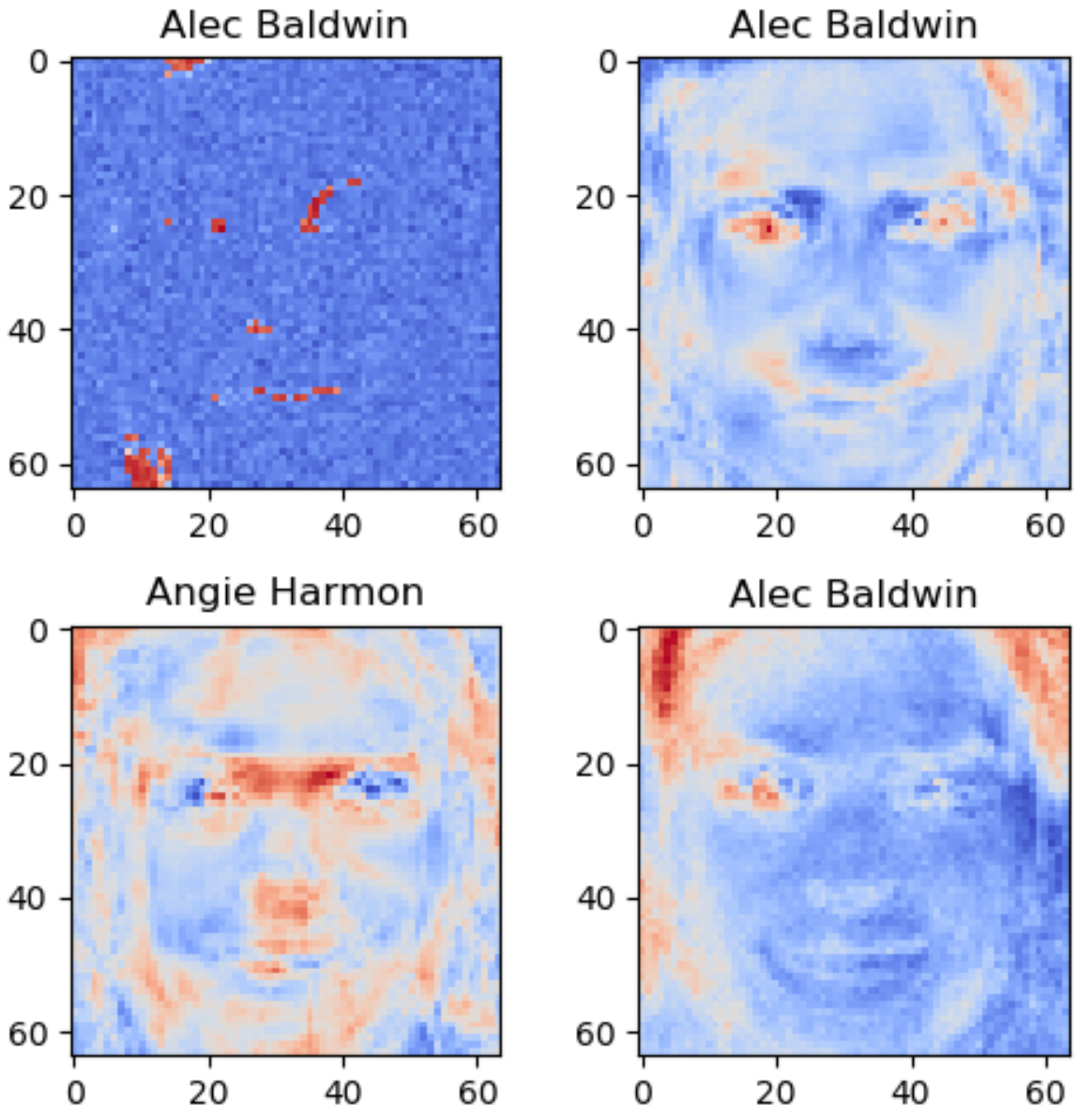

Figure 9: Visualization of the weights important for classifying an image as Alec Baldwin or Angie Harmon. The title above each image denotes the actor/actress that a larger positive activation of the associated hidden unit signifies. Note the resemblance of several of the images to faces.

Part 10

The activations of the fourth convolution layer were extracted by pulling the first 9 layers out of the pretrained AlexNet implementation (corresponding to convolution layer 1 through 4) then applied onto the input. These activations were flattened, stored, and used as the input for the fully-connected neural net built on top of it. This neural network is identical in its basic structure to the neural network described in question 8, is initialised in the same way, and is fed images the same way. The images are 277 by 277 pixels and are preprocessed in the same way as in the sample provided.

Based on the experience obtained in question 8, the hyper parameters were selected. These obtained exceptional performance, so it was decided that no further tuning was required. The neural net had one hundred nodes in its hidden layer and uses the LeakyReLU activation function with negative slope of 0.01. This network was trained with a batch size of one hundred units over one hundred epochs, achieving a final accuracy of 95.69%. The learning curve is visualized in fig. 10. Note that the obtained activations were first detached from AlexNet so that the gradient would not be computed over them.

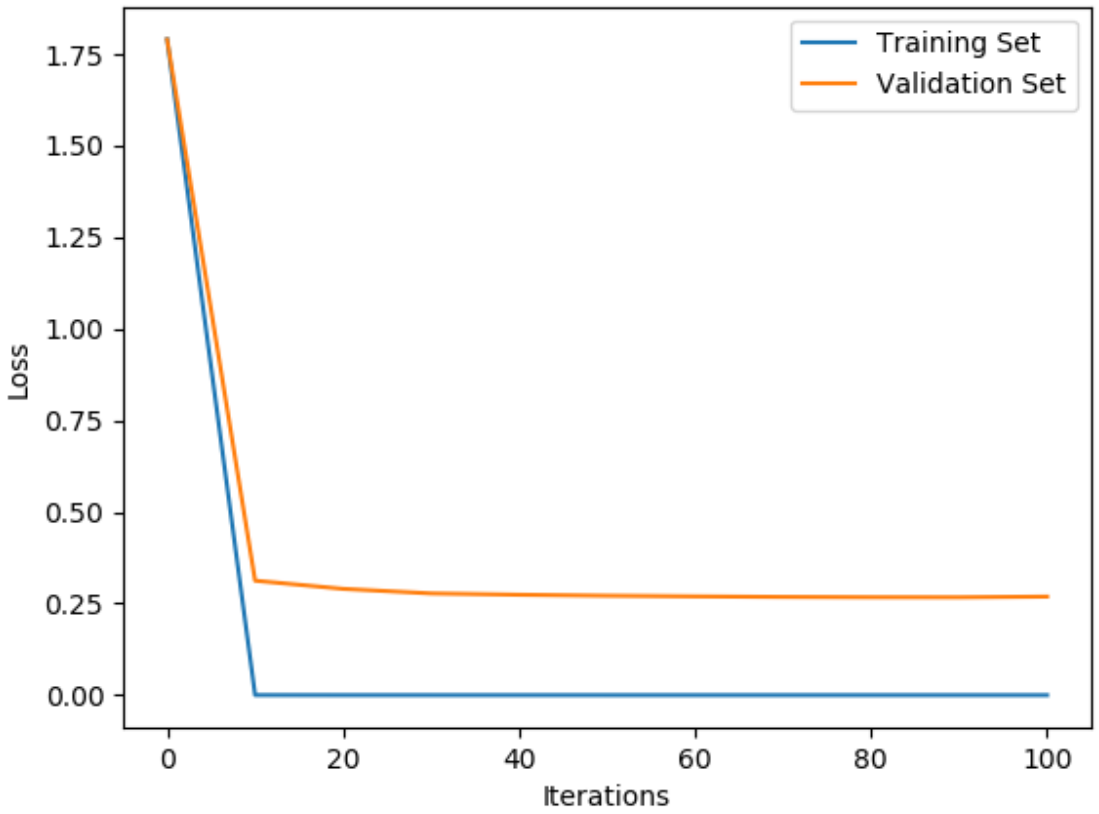

Figure 10: Learning curve for fully-connected neural network built on top of the activations of the conv4 layer of AlexNet.