CSC411: Project #2

Due on Sunday, February 23, 2018

Hao Zhang & Tyler Gamvrelis

February 22, 2018

Foreword

In this project, neural networks of various depth were used to recognize handwritten digits and faces.

Part 1 provides a description of the MNIST dataset from which the images of handwritten digits were taken. Part 2 implements the computation of a simple network. Part 3 presents the *sum of the negative log-probabilities* as a cost function, and derives the expression for its gradient with respect to one of its weights. Part 4 details the training and optimization procedures for a digit-recognition neural network, and part 5 improves upon the results by modifying gradient descent to use momentum. Learning curves are presented in parts 4 and 5, and weight visualization is presented for part 4. Part 6 analyses the behaviour of the network with respect to two of its weights. Part 7 derives asymptotic upper bounds for the runtime complexity of two different backpropagation computation techniques. Part 8 presents a face recognition network architecture for classifying actors, and uses PyTorch for implementation. Part 9 presents visualizations of hidden unit weights relevant to two of the actors. Part 10 uses activations of AlexNet to train a neural network to perform classification of the actors.

System Details for Reproducibility:

- Python 2.7.14
- Libraries:
  - numpy
  - matplotlib
  - pylab
  - time
  - os
  - scipy
  - urllib
  - cPickle
  - PyTorch
  - hashlib
  - collections
  - re

Part 1

Dataset description
The MNIST dataset is made of thousands of 28 by 28 pixel images of the handwritten digits: 0 to 9. The images are split into training set and test set images labelled 'train0' to 'train9' and 'test0' to 'test9'. The number of images with each label is presented in Table 1, below.

Label	**Number of Images**	**Label**	**Number of Images**
train0	5923	test0	980
train1	6742	test1	1135
train2	5958	test2	1032
train3	6131	test3	1010
train4	5842	test4	982
train5	5421	test5	892
train6	5918	test6	958
train7	6265	test7	1028
train8	5851	test8	974
train9	5949	test9	1009

Table 1: Quantity of each type of image in the MNIST dataset.

Ten images of each number were taken from the training sets and displayed in Figure 1. The correct labels of most of the pictures can be discerned at a glance by humans However, since the digits are handwritten, some of them may not be completely obvious. For example, Figure 1cv is categorized as a 9 but looks like an 8.

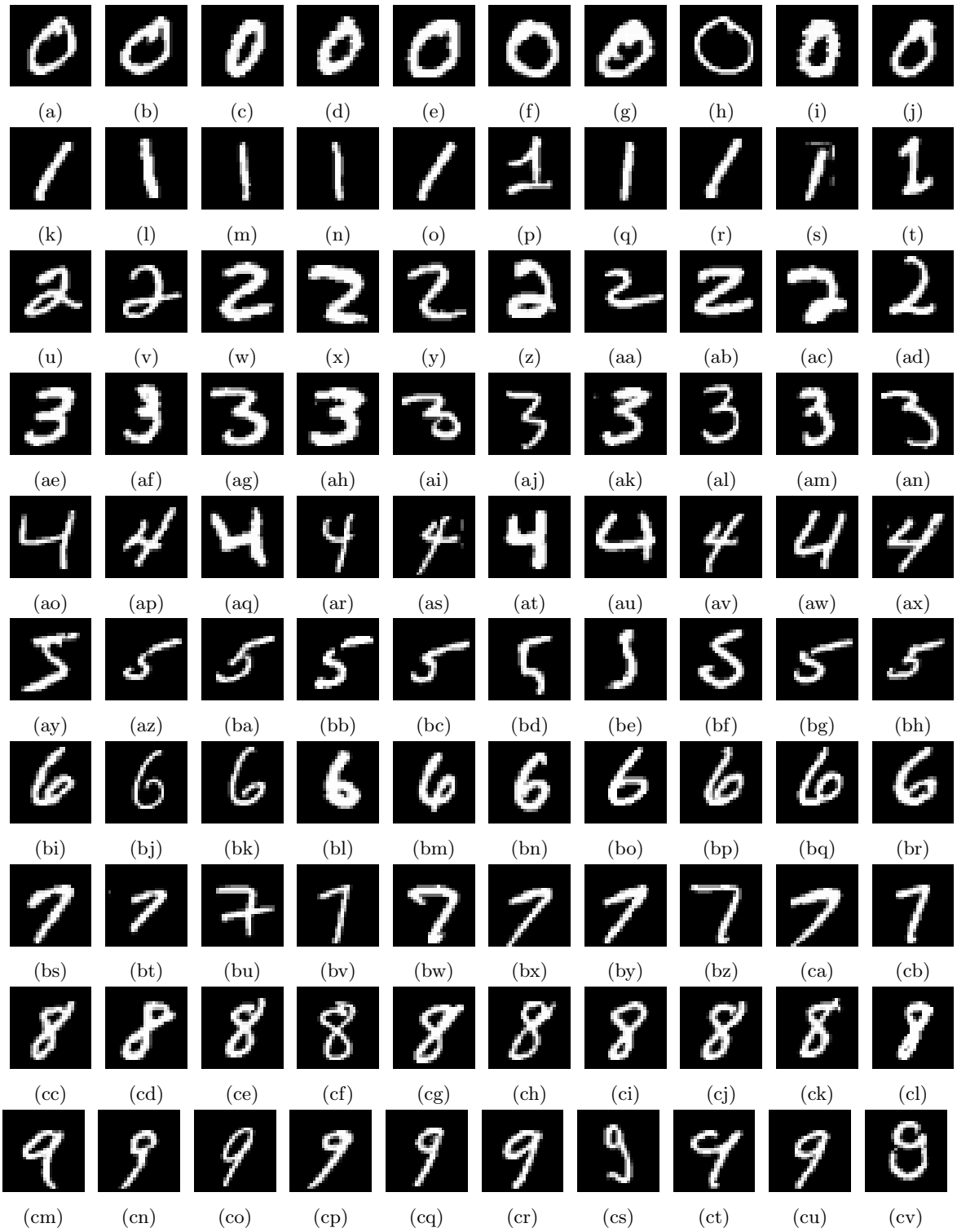


Figure 1: Subset of the MNIST dataset.

Part 2

Computing a Simple Network

In this part, the simple network depicted in Figure 2 was implemented as a function in Python using NumPy, the code listing for which is presented is Figure 3.

Note: It is assumed that the biases are incorporated into the weight matrix and that the input vector is concatenated with an extra $x_{785} = 1$.

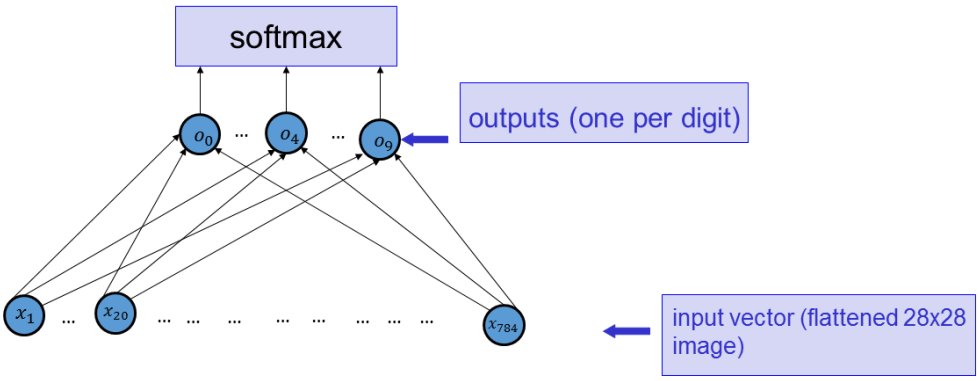


Figure 2: Simple network diagram from project handout.

```
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 SimpleNetwork(W, X):
    '''
    SimpleNetwork returns the vectorized multiplication of the (n x 10)
    parameter matrix W with the data X.
    Arguments:
        W -- (n x 10) matrix of parameters (weights and biases)
        x -- (n x m) matrix whose i-th column corresponds to the i-th training
    image
    '''
    return softmax(np.dot(W.T, X))
```

Figure 3: Python implementation of network using NumPy.

Part 3

Cost Function of Negative Log Probabilities
The Cost function that will be used for the network described in part 2 is:

$$C = -\sum_{q=1}^{m}(\sum_{l=1}^{k} y_l log(p_l))_q$$

Where k is the number of output nodes, m is the number of training samples, $p_l = \frac{e^{o_l}}{\sum_{c=1}^{k} e^{o_c}}$ and y_l is equal to 1 if the training sample is labelled as l and 0 otherwise.

Partial Derivative of the Cost Function (3a)
Let the matrix $W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1k} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2k} \\ \dots & \dots & \dots & \dots & \dots \\ w_{n1} & w_{n2} & w_{n3} & \dots & w_{nk} \end{bmatrix}$ and let W_j be the j^{th} column of matrix W. This results in $o_j = W_j^T x$ where $x = \begin{bmatrix} x_1 \\ x_2 \\ .. \\ x_n \end{bmatrix}$ and n is equal to the number of pixels with an extra 1 to multiply into the bias. That is, $x_1^{(q)}$ is always taken to be 1 while $x_2^{(q)}, ..., x_n^{(q)}$ are the inputs to the network corresponding to the q^{th} training example.

To calculate the partial derivative of C, we will consider only **a single training sample**: $C = -\sum_{l=1}^{k} y_l log(p_l)$
By the chain rule, the partial derivative with respect to w_{ij} is:

$$\frac{\partial C(W)}{\partial w_{ij}} = -\sum_{l=1}^{k} \frac{\partial C}{\partial p_l}\frac{\partial p_l}{\partial o_j}\frac{\partial o_j}{\partial w_{ij}}$$

The first term is straightforward: $\frac{\partial C}{\partial p_l} = \frac{y_l}{p_l}$.

For the second term, we have $\frac{\partial p_l}{\partial o_j} = \begin{cases} \frac{-e^{o_l}e^{o_j}}{(\sum_{c=1}^{k} e^{o_c})^2} = -p_l p_j, \text{if } l \neq j \\ \frac{e^{o_j}}{\sum_{c=1}^{k} e^{o_c}} - \frac{e^{2o_j}}{(\sum_{c=1}^{k} e^{o_c})^2} = p_j - p_j^2, \text{if } l = j \end{cases}$
From the equation: $o_j = W_j^T x$, we have $o_j = w_{1j}x_1 + w_{2j}x_2 + \dots + w_{nj}x_n$ and it is clear to see that: $\frac{\partial o_j}{\partial w_{ij}} = x_i$.

Putting it all together, we have:

$$\frac{\partial C(W)}{\partial w_{ij}} = -(\sum_{l \neq j}(-\frac{y_l}{p_l}p_l p_j) + \frac{y_j}{p_j}(p_j - p_j^2))x_i = (\sum_{l \neq j}(y_l p_j) + y_j(p_j - 1))x_i$$

Considering that we are using 1-hot encoding, only a single y_l can equal 1. Therefore,

$$\frac{\partial C(W)}{\partial w_{ij}} = \begin{cases} (p_j - 1)x_i, & \text{if } y_j = 1 \\ p_j x_i, & \text{otherwise} \end{cases}$$

Which is equivalent to saying: $\frac{\partial C(W)}{\partial w_{ij}} = (p_j - y_j)x_i$. To extend this to **all training samples**, we just have to sum up each individual contribution, resulting in the equation: $\frac{\partial C(W)}{\partial w_{ij}} = \sum_{q=1}^{m}((p_j - y_j)x_i)_q$.

Vectorized Implementation of Gradient (3b)
To vectorize the gradient, we first define the gradient matrix to be:

$$\frac{\partial C(W)}{\partial W} = \begin{bmatrix} \frac{\partial C}{w_{11}} & \dots & \dots & \dots & \frac{\partial C}{w_{1k}} \\ \frac{\partial C}{w_{21}} & \dots & \dots & \dots & \frac{\partial C}{w_{2k}} \\ \dots & \dots & \dots & \dots & \dots \\ \frac{\partial C}{w_{n1}} & \dots & \dots & \dots & \frac{\partial C}{w_{nk}} \end{bmatrix} = \begin{bmatrix} \sum_{q=1}^{m}((p_1 - y_1)x_1)_q & \dots & \dots & \dots & \sum_{q=1}^{m}((p_k - y_k)x_1)_q \\ \sum_{q=1}^{m}((p_1 - y_1)x_2)_q & \dots & \dots & \dots & \sum_{q=1}^{m}((p_k - y_k)x_2)_q \\ \dots & \dots & \dots & \dots & \dots \\ \sum_{q=1}^{m}((p_1 - y_1)x_n)_q & \dots & \dots & \dots & \sum_{q=1}^{m}((p_k - y_k)x_n)_q \end{bmatrix}$$

This is equivalent to: $X(P-Y)^T$ where:

$$X = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & \dots & x_2^{(m)} \\ \dots & \dots & \dots & \dots & \dots \\ x_n^{(1)} & x_n^{(2)} & x_n^{(3)} & \dots & x_n^{(m)} \end{bmatrix} Y = \begin{bmatrix} y_1^{(1)} & y_1^{(2)} & y_1^{(3)} & \dots & y_1^{(m)} \\ y_2^{(1)} & y_2^{(2)} & y_2^{(3)} & \dots & y_2^{(m)} \\ \dots & \dots & \dots & \dots & \dots \\ y_k^{(1)} & y_k^{(2)} & y_k^{(3)} & \dots & y_k^{(m)} \end{bmatrix} P = \begin{bmatrix} p_1^{(1)} & p_1^{(2)} & p_1^{(3)} & \dots & p_1^{(m)} \\ p_2^{(1)} & p_2^{(2)} & p_2^{(3)} & \dots & p_2^{(m)} \\ \dots & \dots & \dots & \dots & \dots \\ p_k^{(1)} & p_k^{(2)} & p_k^{(3)} & \dots & p_k^{(m)} \end{bmatrix}$$

Where the superscript of each element represents the index of the training example it belongs to. The code used to compute the gradient is shown in Figure 4.

```
def negLogLossGrad(X, Y, W):
    '''
    negLogLossGrad returns the gradient of the cost function of negative log
    probabilities.

    Arguments:
        X -- Input image(s) from which predictions will be made (n x m)
        Y -- Target output(s) (actual labels) (10 x m)
        W -- Weight matrix (n x 10)
    '''

    P = SimpleNetwork(W, X) # Get the prediction for X using the weights W

    return np.dot(X, (P - Y).transpose())
```

Figure 4: Vectorized Implementation of the Negative Log Loss Gradient

Finite Difference Approximation (3b)

To check that the gradient was computed correctly, the gradient with respect to weight w_{ij} was approximated using finite differences for all coordinates in the weight matrix. 10 values for the differential quantity, h, were tested, for which the maximum total error was 2.698 for $h = 1$ and the minimum total error was 3.054×10^{-7} for $h = 1 \times 10^{-7}$. Thus, for sufficiently small h, the finite difference approximation and vectorized computation converged to the same quantity, suggesting the vectorized computation was correct. The code used to obtain these results is shown in Figure 5, and the complete results are summarized in Table 2, below.

Total error	**h**
2.698202291518717	1.0
0.200166359876926	0.1
0.019912135073950066	0.01
0.0019906216532900034	0.001
0.00019905646908402463	0.0001
1.990372311189148e-05	1e-05
1.9841905873896337e-06	1e-06
3.2450363651737035e-07	1e-07
3.0542104362263345e-06	1e-08
2.747131061797692e-05	1e-09

Table 2: Sum of differences between vectorized gradient computation and finite difference approximation for 10 h-values.

```
# Test part 3
X = [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3 ,4],[1, 2, 3, 4]]
X = np.array(X)
W = [[1, 0, 1.2, 3], [1, 2, 0.2, 1.2], [1, 8, 1, 4], [1, -2, 3, 1], [1, 0, 3, 1]]
W = np.array(W)
Y = [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
Y = np.array(Y)
G1 = negLogLossGrad(X, Y, W)
h = 1.0
for i in range(10):
    G2 = negLogLossGrad_FiniteDifference(X, Y, W, h)
    print "Total error: ", sum(abs(G1 - G2)), "h: ", h
    h /= 10
```

Figure 5: Test Case Used to Calculate Error Between Finite Difference Approximation and the Vectorized Gradient

Part 4

Training a Digit Classifier Using Vanilla Gradient Descent

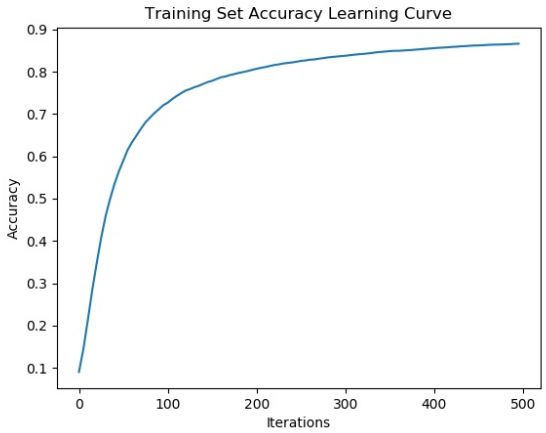
In this section, the neural network from part 2 was trained using vanilla gradient descent. The optimization procedure is described below, and learning curves and visualizations of the weights going into each of the output units are presented in Figure 6 and Figure 7, respectively.

Optimization Procedure

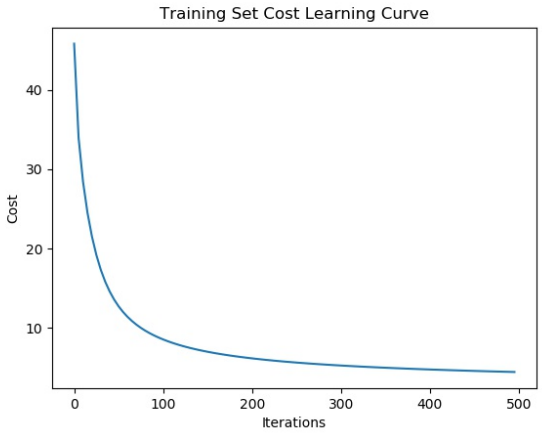
The final gradient descent parameters used were $\epsilon = 1 \times 10^{-5}$, $\alpha = 1 \times 10^{-6}$, and a maximum number of iterations of 500. The initial guess for the weight matrix was randomly initialized using `np.random.rand()`, seeded with the value of 3. The decision to randomly initialize the weight matrix was made arbitrarily at the beginning of the optimization procedure to obtain a distribution of value, and was not changed since acceptable performance resulted. The gradient descent parameters were determined experimentally by observing the performance on the validation set. Values between 500 and 5000 were tested for the maximum number of iterations, but little difference in performance was observed. Thus, 500 was chosen to reduce the risk of over-fitting. The value of ϵ did not play a significant role given this number of iterations, as the cost function was decreasing by a large amount (approximately 10) between iterations. Thus, the stopping criterion was effectively the number of iterations. The value for α was initialized to the same value used in part 3 of project 1, and provided acceptable performance; thus it was not altered.

Learning Curves

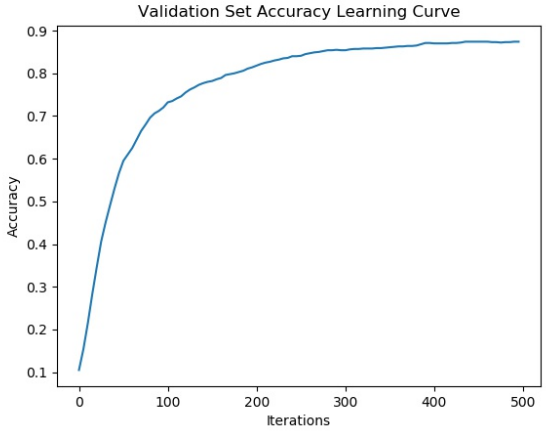
The learning and loss curves followed similar trajectories for the training and validation sets. One key difference is that the validation set was slightly more "bumpy" than the training set.



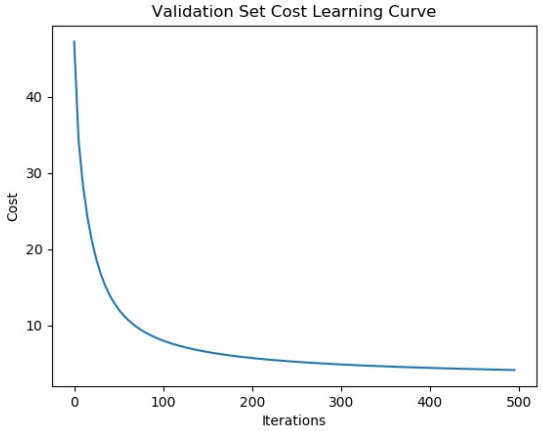
(a) Performance Curve on the Training Set



(b) Loss Curve on the Training Set



(c) Performance Curve on the Validation Set

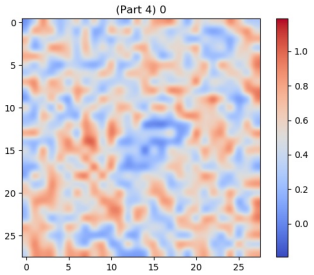


(d) Loss Curve on the Validation Set

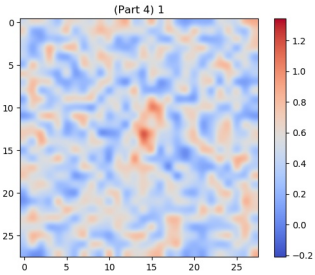
Figure 6: Part 4 Learning Curves

Weights Going Into Output Units

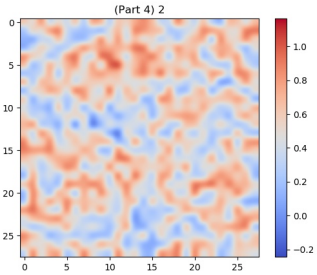
Most of the weights did not bear strong resemblances to the digits they corresponded to. However, one can still observe the iconic circular shape for the digit zero, and the vertical line for the digit one. Other weights such as three and seven share some visual resemblance to their digits, although not obviously.



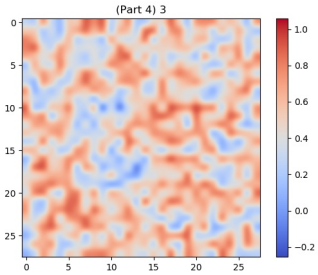
(a) Digit 0



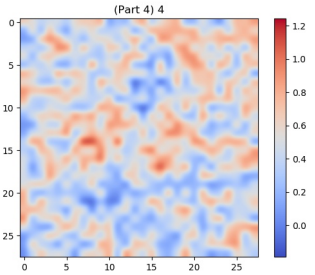
(b) Digit 1



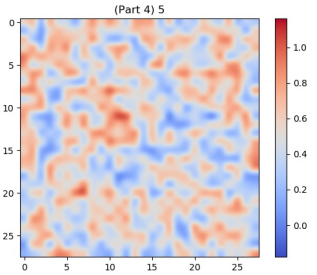
(c) Digit 2



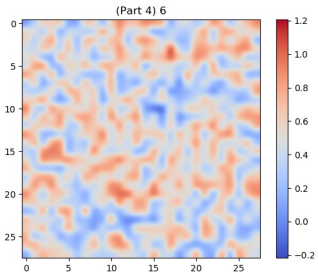
(d) Digit 3



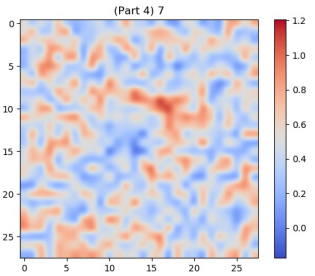
(e) Digit 4



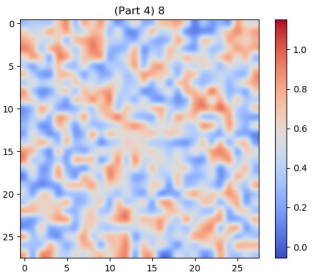
(f) Digit 5



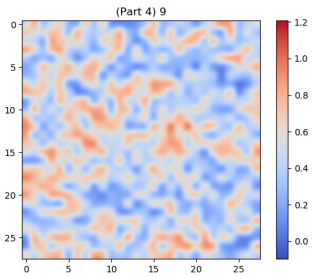
(g) Digit 6



(h) Digit 7



(i) Digit 8



(j) Digit 9

Figure 7: Visualization of the Weights for Digits 0 Through 9 in Part 4

Part 5

Training a Digit Classifier Using Gradient Descent With Momentum

In this section, the neural network from part 2 was trained using gradient descent with momentum. The learning curves are presented in Figure 8, and the vectorized implementation of gradient descent with momentum is presented in Figure 9. The only difference between vanilla gradient descent and gradient descent with momentum is the update rule: momentum accelerates convergence towards the direction most consistently travelled.

Vanilla gradient descent update rule:

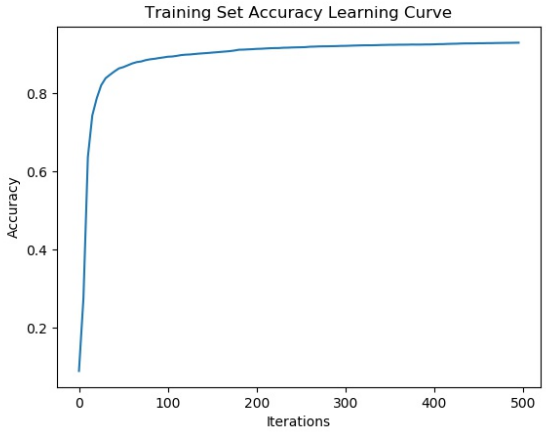
$$W \leftarrow W - \alpha \frac{\partial C}{\partial W}$$

Gradient descent with momentum update rule:

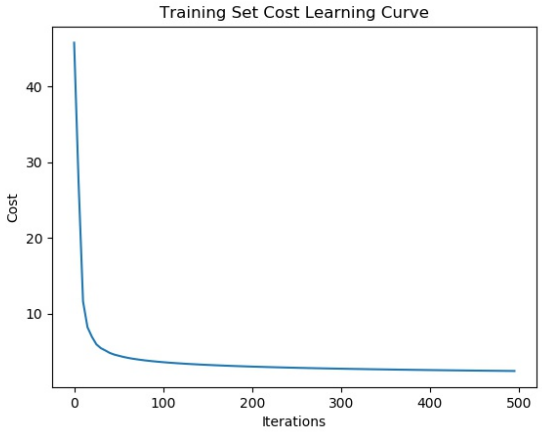
$$\nu \leftarrow \gamma\nu + \alpha \frac{\partial C}{\partial W}$$

$$W \leftarrow W - \nu$$

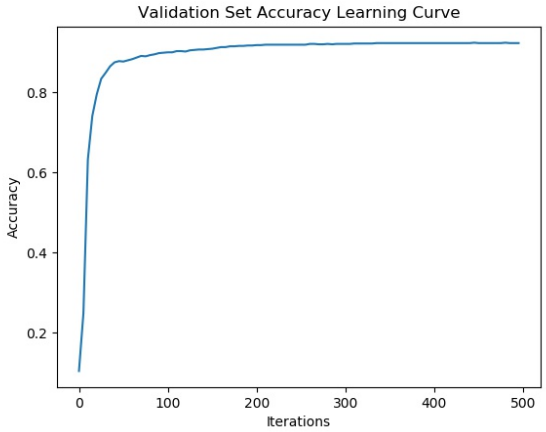
Comparing Figure 6 and Figure 8, it is clear that compared to vanilla gradient descent, gradient descent with momentum results in better performance (in terms of both accuracy and cost) in fewer iterations. One might say that gradient descent with momentum produces "sharper" or more "rapid" learning curves. Indeed, vanilla gradient descent took nearly 200 iterations to reach 80% accuracy on the validation set, while gradient descent with momentum took approximately 25 iterations to reach this performance. As was mentioned before, the reason for this is that the update rule tends to steer the gradient towards the most consistently travelled direction, which (more often than not) results in accelerated convergence when properly tuned.



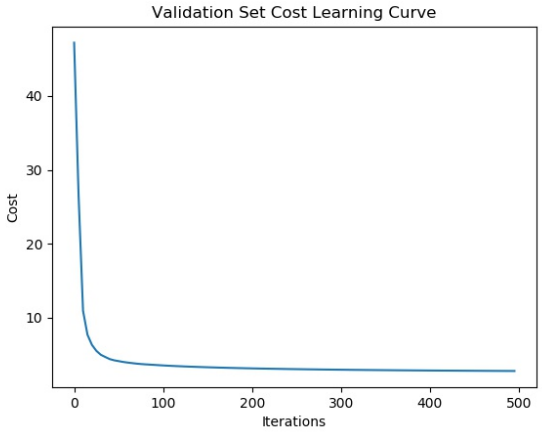
(a) Performance Curve on the Training Set



(b) Loss Curve on the Training Set



(c) Performance Curve on the Validation Set



(d) Loss Curve on the Validation Set

Figure 8: Part 5 Learning Curves

```python
def part5_gradient_descent(X, Y, init_W, alpha, eps, max_iter, init_v, momentum):
    '''
    part5_gradient_descent finds a local minimum of the hyperplane defined by
    the hypothesis dot(W.T, X). The algorithm terminates when successive
    values of W differ by less than eps (convergence), or when the number of
    iterations exceeds max_iter. This function uses momentum

    Arguments:
        X -- input data for X (the data to be used to make predictions)
        Y -- input data for X (the actual/target data)
        init_W -- the initial guess for the local minimum (starting point)
        alpha -- the learning rate; proportional to the step size
        eps -- used to determine when the algorithm has converged on a solution
        max_iter -- the maximum number of times the algorithm will loop before
        terminating
        init_v -- the initial momentum value
        momentum -- the momentum parameter in range (0, 1)
    '''

    iter = 0
    previous_W = 0
    current_W = init_W.copy()
    previous_v = 0
    current_v = init_v # Initial momentum...(not strictly necessary)
    firstPass = True
    history = list()
    Whistory = list()

    m = Y.shape[1]

    # Do-while...
    while(firstPass or
            (np.linalg.norm(current_W  - previous_W) > eps and
            iter < max_iter)):
        firstPass = False

        previous_W = current_W.copy() # Update the previous W value
        previous_v = current_v # Update previous momentum

        current_v = momentum * current_v + alpha * p3.negLogLossGrad(X, Y, current_W)
        current_W = current_W - current_v
    #As opposed to current_W = current_W - alpha * p3.negLogLossGrad(X, Y, current_W)
        if(iter % (max_iter // 100) == 0):
            # Print updates every so often and save cost into history list
            cost = p3.NLL(p2.SimpleNetwork(current_W, X), Y)
            history.append((iter, cost))
            Whistory.append(current_W)
            print("Iter: ", iter, " | Cost: ", cost)

        iter += 1

    return(Whistory, history)
```

Figure 9: Vectorized Code for Gradient Descent with Momentum

Part 6

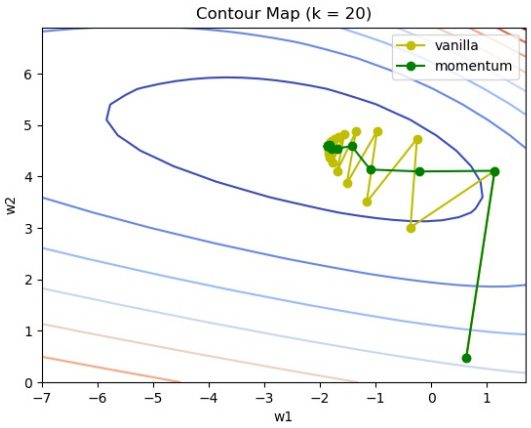
Comparison Between Momentum and Vanilla Gradient Descent

The effect of momentum on gradient descent was studied by taking two different weights, treating them as coordinates and plotting their path over 20 iterations on a contour plot. Gradient descent was run two times for each plot, using the same initial weight matrices and only changing the two chosen weights while keeping the other weights constant. The coordinates of the weights studied were $W[200, 2]$ and $W[201, 2]$. Figure 10a shows a plot where the two methods were comparable in terms of performance.

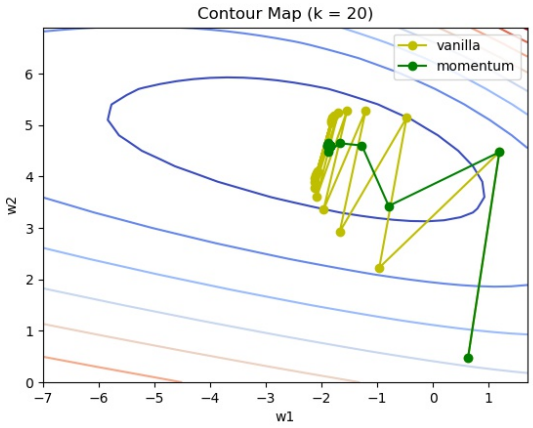
Results

It was found that the effects of momentum could be seen most clearly when the learning rate was slightly increased, shown in Figure 10b. This is because in the vanilla method, the weights would travel mostly vertically, due to the shape of the contour, while in the momentum method, the vertical components of the weights would cancel each other out over consecutive steps. Therefore, the momentum method travels further towards the optimum with each step than the vanilla method.

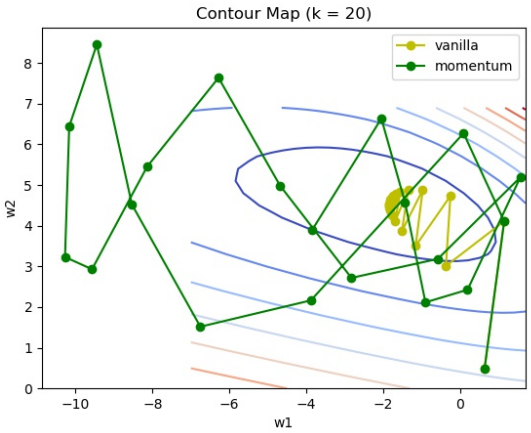
However, when momentum is too high, the contributions from the previous steps would have too much of an effect, resulting in the weights overshooting the minimum like in Figure 10c.



(a) Alpha = 0.5, Momentum = 0.3



(b) Alpha = 0.55, Momentum = 0.3



(c) Alpha = 0.5, Momentum = 0.9

Figure 10: Contour Plots

Part 7

The Effect of Caching on the Time Required to Compute the Gradient

A neural network consists of several layers of *neurons* connected to each other by edges which each have their own *weight*. In order to optimize a neural network, we need to "learn" the values for the weights that minimize the cost function. The *gradient* of the cost function can be computed with respect to each weight in the network; this matrix-valued quantity indicates (up to a proportionality constant) how much each of the weights should be changed to achieve the target output. Traditionally, there are two algorithms that come to mind for how to compute the gradient, both of which make excessive use of the chain rule from calculus:

(1) Naïve approach: brute-force computation of the gradient, where each entry in the gradient matrix is evaluated separately.

(2) Better approach: "backpropagation", a computation of the gradient that uses caching to store each partial derivative so that no term ever has to be computed twice. Terms are accessed from a table in $O(1)$ time once they have been computed, and are added to the table otherwise.

This section presents an analysis of the runtime complexity of these two algorithms, ultimately deriving asymptotic upper bounds for each in the case of a fully-connected network with N layers and K neurons per layer. Note that only one training example will be considered, while in general the gradient would be an average over all training examples. The variables and dimensions to be used are explained as follows:

- C is the cost function
- $W^{(i,j,k)}$ is the weight corresponding to the edge connecting the j^{th} neuron in the $(i-1)^{\text{th}}$ layer to the k^{th} neuron in the i^{th} layer
  - $i \in [1, N]$
  - $j, k \in [1, K]$
- $a_k^{(i)}$ is the output from the k^{th} neuron in the i^{th} layer. Note that the neurons $a_k^{(N)}$ form the output layer and the neurons $a_k^{(0)}$ form the input.
- N is the number of layers in the network
- K is the number of neurons per layer

Also, the following reasonable assumptions are made to simplify the analysis:

1. Element-wise multiplication is $O(1)$
2. Addition is negligible
3. An N-layer neural network includes the output layer in the count; that is, it has $N-1$ hidden layers
4. Activation functions and non-linearities such as softmax at the output layer can be ignored
5. Partial derivatives in tables (i.e. previously-solved subproblems) take negligible time to retrieve

Characterization of Naïve Approach (Brute Force)

We begin with the naïve approach as this will make the analysis for the "better" approach more simple due to the terminology developed. We begin by writing the derivative of the cost function with respect to an arbitrary weight:

$$\frac{\partial C}{\partial W^{(i,j,k)}}$$

Next, we consider the fact that the cost function is evaluated at the output neurons, $a_k^{(N)}$; that is, $C = C(a_1^{(N)}, ..., a_K^{(N)})$. Each $a_k^{(N)}$ is in turn computed by first evaluating the inner product $\sum_{j=1}^{K} W^{(N-1,j,k)} a_j^{(N-1)}$, which uses the outputs from layer $N-1$ and the weights connecting these outputs to neuron k in layer N. Each $a_j^{(N-1)}$ can also be computed in a similar fashion using the outputs and weights from the previous layer, until the input layer is reached. What this means is that any $a_k^{(i)}$ is technically an input to the cost function, so it makes sense to write expressions such as $\frac{\partial C}{\partial a_k^{(i)}}$. In fact, we can relate this to the partial derivative with respect to an arbitrary weight in the network using the chain rule from calculus:

$$\frac{\partial C}{\partial W^{(i,j,k)}} = \frac{\partial C}{\partial a_k^{(i)}} \frac{\partial a_k^{(i)}}{\partial W^{(i,j,k)}}$$

But we can expand $\frac{\partial C}{\partial a_k^{(i)}}$ in terms of the a_k's that are affected in the next layer by an infinitesimal change in $a_k^{(i)}$ (note that we sum from 1 to K because we are studying a fully-connected network and hence $a_k^{(i)}$ connects to K neurons in the next layer):

$$\frac{\partial C}{\partial a_k^{(i)}} = \sum_{l=1}^{K} \Big(\frac{\partial C}{\partial a_l^{(i+1)}} \frac{\partial a_l^{(i+1)}}{\partial a_k^{(i)}} \Big)$$

This expansion gives yet a larger expression for the derivative of the cost function with respect to an arbitrary weight:

$$\frac{\partial C}{\partial W^{(i,j,k)}} = \frac{\partial C}{\partial a_k^{(i)}} \frac{\partial a_k^{(i)}}{\partial W^{(i,j,k)}} = \Big(\sum_{l=1}^{K} \Big(\frac{\partial C}{\partial a_l^{(i+1)}} \frac{\partial a_l^{(i+1)}}{\partial a_k^{(i)}} \Big) \Big) \frac{\partial a_k^{(i)}}{\partial W^{(i,j,k)}} = \frac{\partial a_k^{(i)}}{\partial W^{(i,j,k)}} \sum_{l=1}^{K} \Big(\frac{\partial C}{\partial a_l^{(i+1)}} \frac{\partial a_l^{(i+1)}}{\partial a_k^{(i)}} \Big)$$

Consider the case where we are interested in $\frac{\partial C}{\partial W^{(1,j,k)}}$, that is, the derivative of the cost function with respect to a weight connecting the input layer and layer 1. In this case, the expansion has N summation terms involving weights in the next layers:

$$\frac{\partial C}{\partial W^{(1,j,k)}} = \frac{\partial a_k^{(1)}}{\partial W^{(1,j,k)}} \sum_{l_1=1}^{K} \Big(\frac{\partial a_{l_1}^{(2)}}{\partial a_k^{(1)}} \sum_{l_2=1}^{K} \Big(\frac{\partial a_{l_2}^{(3)}}{\partial a_{l_1}^{(2)}} \sum_{l_3=1}^{K} \Big(\dots \quad \dots \sum_{l_{N-1}=1}^{K} \Big(\frac{\partial a_{l_{N-1}}^{(N-1)}}{\partial a_{l_{N-2}}^{(N-2)}} \sum_{l_N=1}^{K} \Big(\frac{\partial a_{l_N}^{(N)}}{\partial a_{l_{N-1}}^{(N-1)}} \Big) \Big) \Big) \Big) \Big) \Big)$$

Since there are N sums, and each runs from 1 to K, this computation of the gradient with respect to a weight between the input layer and the first layer involves K^N operations, provided that the previously-stated assumptions hold. Thus, it follows that for a weight connecting the first and second layers, there are K^{N-1} computations (since the indices in the previous expression would all begin at layer 2 instead of layer 1). Indeed, for the gradient with respect to any weight connecting neurons in the $(i-1)^{\text{th}}$ and i^{th} layers, there are $K^{N-(i-1)}$ computations. Thus, for weights connecting the $N-1^{\text{th}}$ and N^{th} layers, there are then $K^{N-(N-1)} = K$ computations.

The total number of operations is then the product of the number of nodes per layer (K) and the number of operations to compute the gradient with respect to a weight in each layer:

$$K(K^N + K^{N-1} + ... + K) = K^{N+1} + K^N + ... + K^2 \in O(NK^{N+1})$$

So the naïve algorithm has a runtime complexity that is asymptotically bounded above by $O(NK^{N+1})$.

Characterization of Backpropagation
Now we will analyse the "better" solution, backpropagation. Intuitively, we expect that after computing each partial derivative once, we should never need to compute it again as this exemplifies an *overlapping subproblem*. To illustrate this, note that the partial derivative of the cost function in the first layer (implicitly) uses the partial derivatives of the cost function with respect to the outputs and weights in the second layer, and so forth. These latter partial derivatives should be retrievable in constant time if they were computed earlier.

Note the following:

1. Fully connected implies $K \times K = K^2$ weights per inter-layer junction
2. N layers implies $N - 1$ inter-layer junctions
3. Each partial derivative should be constant time to compute
4. $(N - 1)K^2$ total weights in the network

Now, in the final layer of the network, there is 1 partial derivative to compute for each $a_k^{(N)}$, which is simply $\frac{\partial C}{\partial a_k^{(N)}}$. However, in the previous inter-layer junction (i.e. connecting the $(N-1)^{\text{th}}$ layer to the N^{th} layer), there are 2 partial derivatives to compute, as shown below:

$$\frac{\partial C}{\partial W^{(i,j,k)}} = \frac{\partial C}{\partial a_k^{(N)}} \frac{\partial a_k^{(N)}}{\partial W^{(N,j,k)}}$$

But $\frac{\partial C}{\partial a_k^{(N)}}$ is computed when considering the derivative of the cost function with respect to the output layer. So, if we work backwards through the neural network, we can see we can save time on computation since we don't need to keep computing partial derivatives whose values we already know. That is, in the case shown above, there were two partial derivatives that needed to be computed, but only one of them would need to be computed if the derivatives in the previous layer were already solved.

This concept generalizes across the network; thus, since there is one new computation required per weight, and there are K^2 weights per inter-layer junction, the total number of computations required to solve the gradient of the network with respect to all weights is:

$$K^2 + K^2 + ... + K^2 \qquad N \text{ times}$$

This gives an asymptotic upper bound on the runtime complexity of $O(K^2N)$.

Part 8

Implementation of a Fully Connected Neural Network in PyTorch
A fully connected neural network was implemented in PyTorch with a single hidden layer and 6 output nodes. The network was trained on the faces of the 6 actors from project 1: Lorraine Bracco, Peri Gilpin, Angie Harmon, Alec Baldwin, Bill Hader and Steve Carell.

Image Processing
The number of photos used per actor for each set was 70 for the training set and 20 for both validation and test sets. To obtain each set, the uncropped photos from project 1 were processed using the `make3Sets()` function found in `image_processing.py`, which creates the 3 sets of colored photos at the desired resolution. The function also compares the hashes of each photo to their expected hashes given in `actors.txt`, but some of the valid photos were wrongly skipped so they were hardcoded back in.

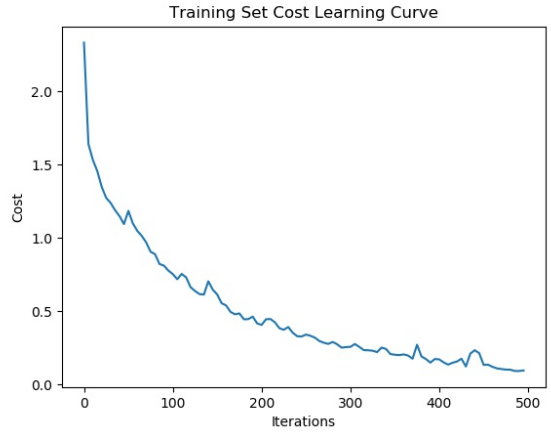
Creation of Input Matrices
To create the input matrices for each set, each color channel of a photo was flattened into a 1D vector and concatenated along the 1D axis. This input vector was then stacked on top of a matrix which contained the input vectors of the other images in the set. In the resulting matrix, each row represented a different image while each column represented one of 3 colour values of a single pixel. The label matrices of each set were created by one-hot encoding 1 by 6 vectors and stacking them on top of each other.

Final Results
The properties of the final and best performing network created can be seen in Table 3. The network's final performance was 82.25% correct on the validation set and 80.5% correct on the test set. The learning curves at these settings for both test and validation sets are shown in Figure 11.

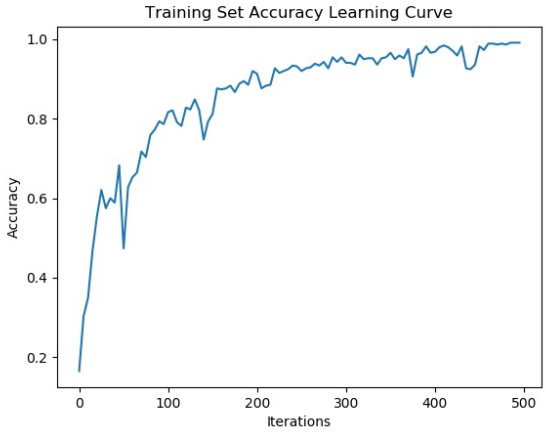
Property	**Value**
Input Resolution	28x28
Input Layer Size	$28 * 28 * 3$
Hidden Layer Size	70
Output Layer Size	6
Hidden Layer Activation Function	Tanh
Loss Function	Cross Entropy Loss
Optimizer	Adam
Learning Rate	1e-3
Batch Size	60
Steps	500
Initial Weights	Default Values

Table 3: Properties of the Best Performing Neural Network

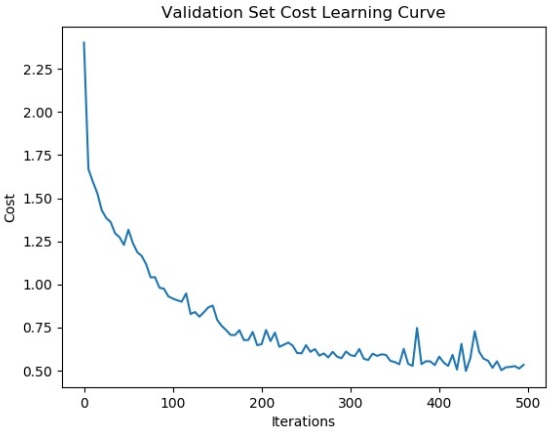
Experimental Observations for Part 8
The Effect of Resolution on Performance
The network was trained on square images at varying resolutions by making the input layer a function of the resolution. The resolutions used include: 64x64, 32x32, 28x28, 24x24 and 14x14. It was found that images close to 32x32 in size resulted in the best performance while those at resolutions of 14x14 and 64x64 gave slightly lower performances. It was also found that a larger resolution such as 64x64 required a much larger hidden layer to optimize than lower hidden layers.



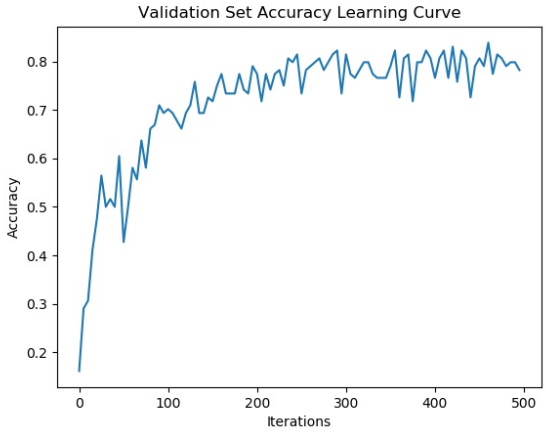
(a) Performance Curve on the Training Set



(b) Loss Curve on the Training Set



(c) Performance Curve on the Validation Set



(d) Loss Curve on the Validation Set

Figure 11: Part 8Learning Curves

The Effect of Activation Functions on Performance
Different activation functions for the hidden layer were also tested and it was found that as long as the hidden layer is large enough, (ie. ≈50 nodes), most activation functions only made a few percentage difference. However, certain activation functions such as Softmin and Softmax would perform poorly no matter the size of the hidden layer. When the hidden layer was small,(ie. ≈6 nodes), it was found that the best activation functions were relatively linear functions that could take both positive and negative values such as linear, Tanhshrink and Softshrink.

Weight Initialization
Different methods were tried to initialize the weights using functions such as `normal_()`, `uniform_()` and `zero()` with various parameters. It was found that if the weights were too large then the performance stayed constant at 1/6. The weights that gave the best performance were the default weights.

Different loss functions and optimizers were also attempted but it was found that most loss functions required a different format of input so it was kept as `CrossEntropyLoss()` while Adam was confirmed to be the best optimizer.

Part 9

Visualizing the Weights of "Useful" Hidden Units

The single-hidden-layer fully-connected network presented in part 8 was explored in greater depth in this part of the project. Consider the task of classifying images of just one actor, for example, Angie Harmon. Since the label vector, y, used a one-hot encoding scheme, a "perfect" classification of an image of Harmon would have resulted in the output $[0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0]^T$ from the neural network. Note that only one output neuron is activated; let this output neuron corresponding to the actor be called o_i. One can then inquire about which hidden units were the most important, or "useful" in computing this classification correctly. This part of the project explored a way of determining which hidden units were the most "useful" for classifying input photos of Angie Harmon and Peri Gilpin, and visualizing the weights going into these "useful" hidden units from the input layer.

Defining "Useful"

It was assumed that there were 2 "useful" hidden units for each actor:

(1) The hidden unit with the *maximum* input value to o_i. This hidden unit promotes the activation of o_i the most.

(2) The hidden unit with the *minimum* input value to o_i. This hidden unit inhibits the activation of o_i the most.

Note that *input value* refers neither to the activation value of the hidden neuron nor the weight of the edge connecting the hidden neuron to o_i; rather, it refers to the product of these two quantities. This product represents the *actual* contribution from the hidden unit.

Procedure to Determine the Most "Useful" Hidden Units

(1) Pick the actor whose "useful" hidden units are to be determined. This specifies an output neuron o_i whose connections to the hidden layer will be studied

   (a) Feed an input of the actor from the validation set into the neural network

   (b) Observe the weights connecting the hidden layer to o_i

   (c) Observe the activation values from the neurons in the hidden layer

   (d) For each neuron in the hidden layer, multiply its activation value by the weight connecting it to o_i. This product represents the actual contribution from the hidden unit

   (e) Record the indices for the "useful" hidden units (as per the definition of "useful" above

(2) Repeat the previous step for all images of the actor

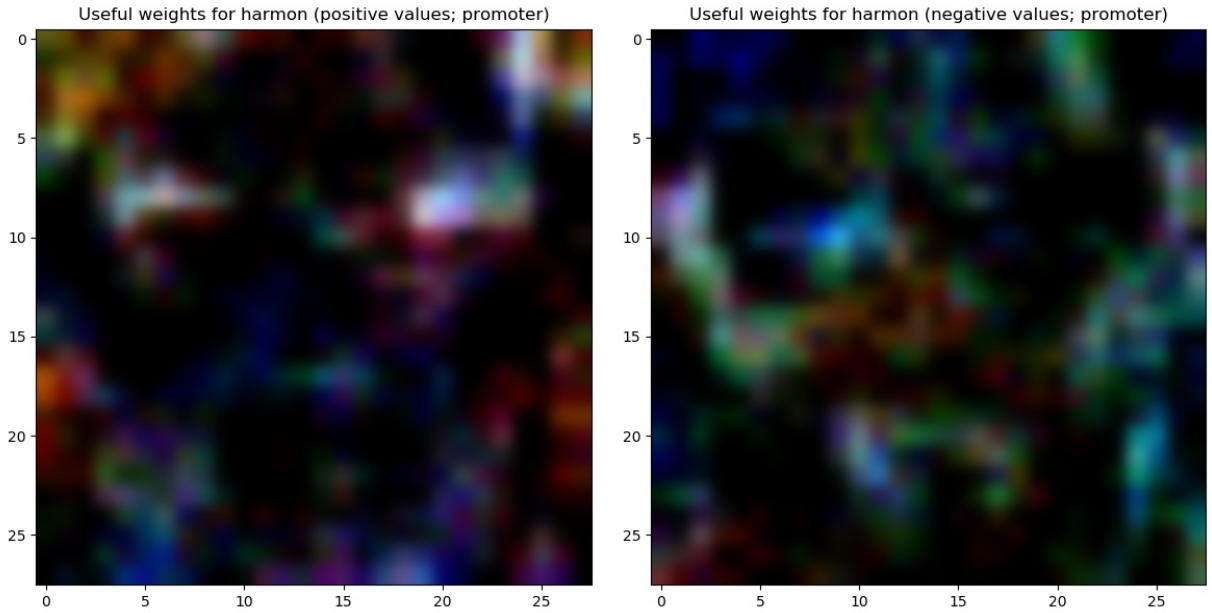
(3) Pick the most frequently-occurring promoter neuron, and the most frequently-occurring inhibitor neuron. Continue to the next step, considering just these two neurons to be "useful"

(4) Visualize the weights connecting the "useful" hidden units to the input layer

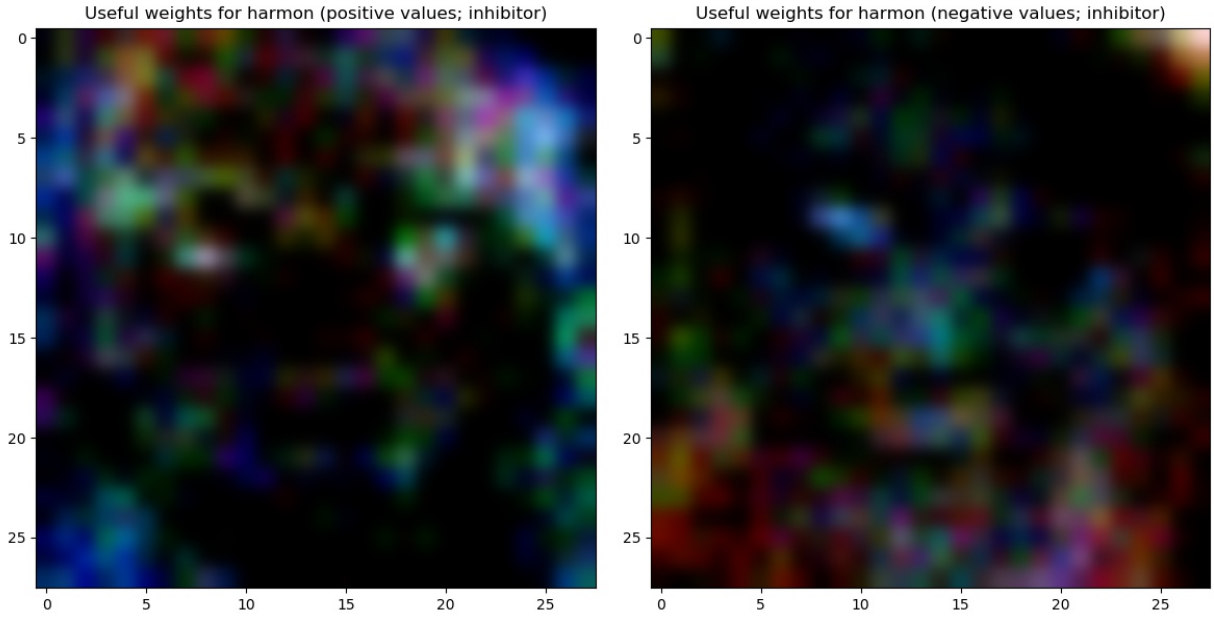
(5) Repeat for the next actor

Comments on Weight Visualization

The weights connecting the hidden layer to the most "useful" hidden units are presented below, with the weights for Angie Harmon in Figure 12, and the weights for Peri Gilpin in Figure 13. Since it is possible for weights to be both positive and negative, the positive and negative parts of the weights were plotted as two separate images; below, the positive parts of the weights are on the left while the negative parts are on the right. The omitted values were set to 0 for each plot. To obtain the weight visualizations from the weight vectors, the portions of the weight vectors representing the red, green, and blue channels of the input images were reshaped back to a 2-D color image.

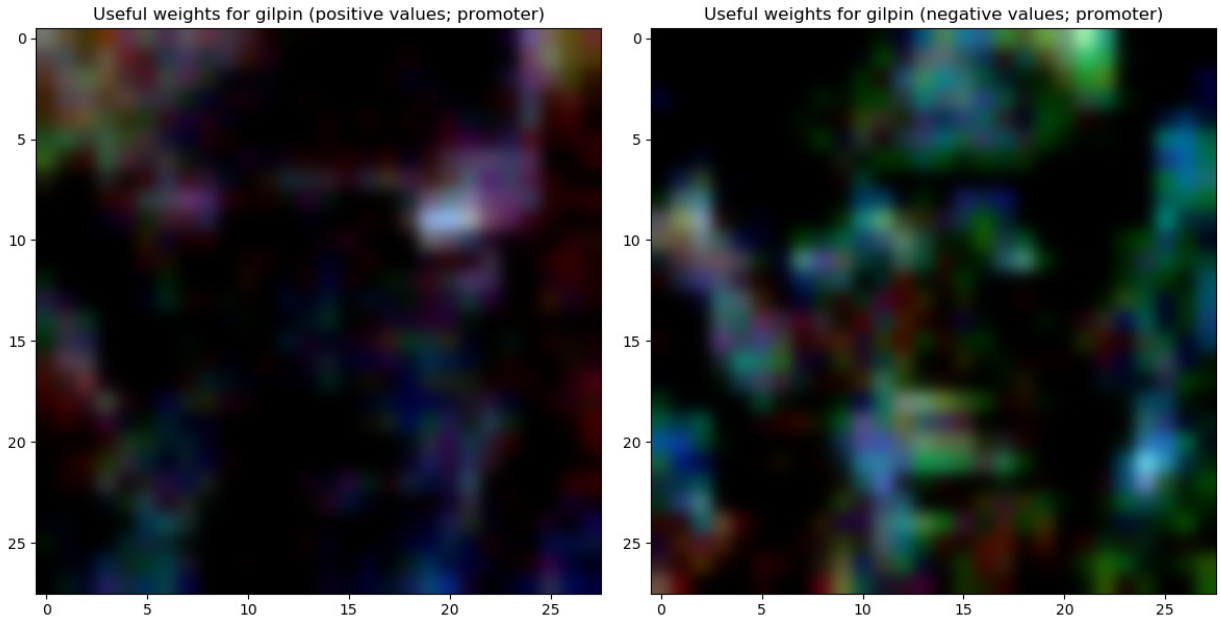


(a) The hidden unit promoting the activation of Harmon the most.

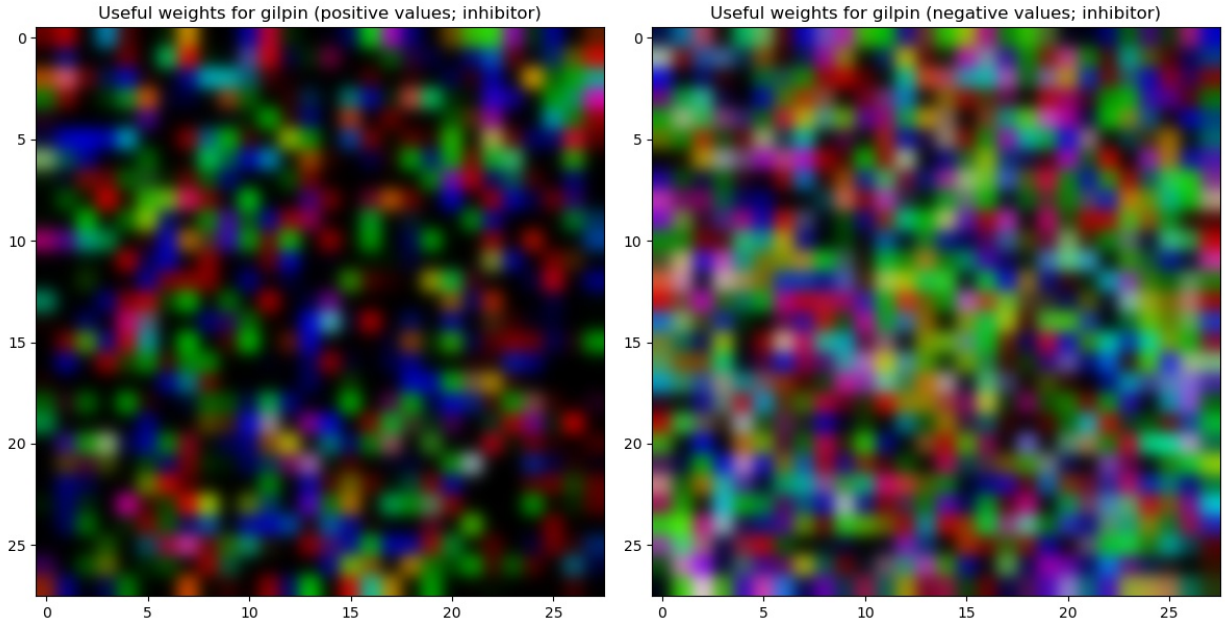


(b) The hidden unit inhibiting the activation of Harmon the most.

Figure 12: Weight visualizations for Angie Harmon.



(a) The hidden unit promoting the activation of Gilpin the most.



(b) The hidden unit inhibiting the activation of Gilpin the most.

Figure 13: Weight visualizations for Peri Gilpin.

Part 10

Implementing a Fully Connected Neural Network on Top of MyAlexNet
To increase the accuracy of classification, the images were put into MyAlexNet and outputs from its features layers were taken and connected to a fully connected network with 1 hidden layer.

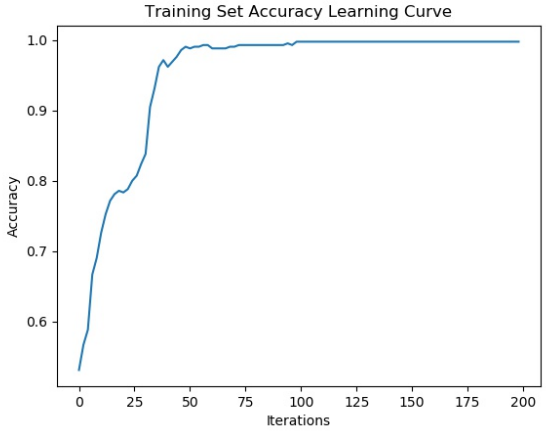
Modifying MyAlexNet
To access the Top feature layer of MyAlexNet, the classifier in the forward function was commented out and the output was reshaped into a 1D vector. To access deeper layers, they are commented out in the $self.features$ section of the class. For every convolutional layer that's commented out, we also comment out an in dex in the $features\_weight\_i$ array to prevent the program from trying to load weights into a non-existent layer.

Computing the Input Matrix for the Fully Connected Layers
After MyAlexNet is modified to provide the desired output, we take each image in a set and put it into MyAlexNet using the sample code given, obtain its corresponding 1D vector and compile it into a set matrix. In the resulting matrix, each row represents an image in that set while each column represents an output of MyAlexNet corresponding to that image. The label matrices are created in the same way as in part 8.
Since only the fully connected layers are trained, the training set matrix from MyAlexNet will always stay the same. Therefore, we can use the output of MyAlexNet as the new inputs to the the fully connected layers without having to recalculate them every time. The input layer of the fully connected layers also has to be resized to match the number of outputs from MyAlexNet.

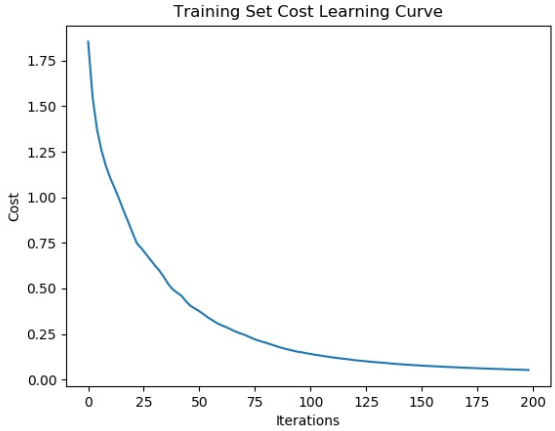
Final Results
The properties of the optimal configuration are shown in Table 4. The outputs from conv4 after the MaxPool layer and from conv2 after the ReLU layer were found to provide the best results but the latter, while having more outputs, gave slightly better performance. The final performance on the validation set was 96.6% while the final performance on the test set was 98.33%. This represents approximately a 70% drop in the error rate from part 8. The learning curves of the test and validation sets are disaplayed in Figure 14

Property	**Value**
Input Resolution	227x227
MyAlexNet Output Layer	conv2 after ReLU
Input Layer Size	64896
Hidden Layer Size	50
Output Layer Size	6
Hidden Layer Activation Function	Tanh
Loss Function	Cross Entropy Loss
Optimizer	Adam
Learning Rate	1e-3
Batch Size	50
Steps	200
Initial Weights	Default Values

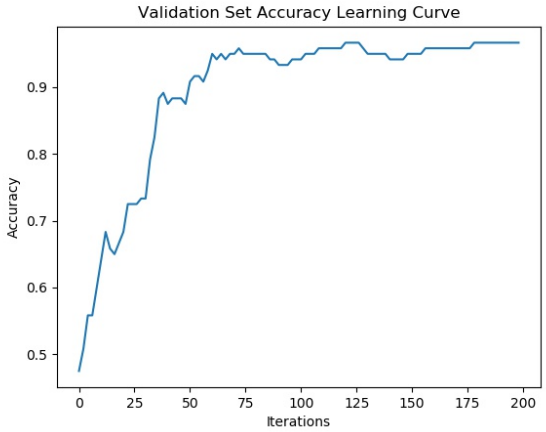
Table 4: Properties of the Best Performing Neural Network



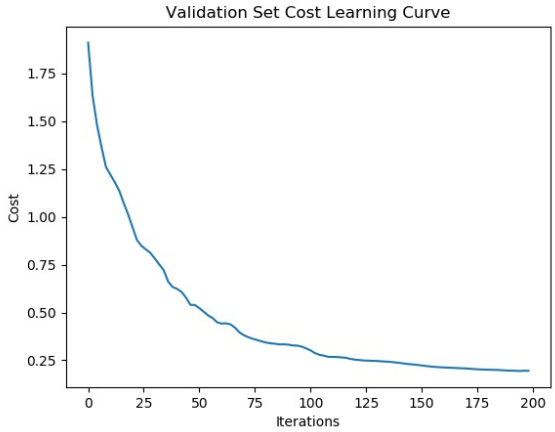
(a) Performance Curve on the Training Set



(b) Loss Curve on the Training Set



(c) Performance Curve on the Validation Set



(d) Loss Curve on the Validation Set

Figure 14: Part 10 Learning Curves