CSC411: Assignment #2

Due on Friday, February 23, 2018

Zhiyan Deng, Xin Jie Lee

February 23, 2018

Prefix

The code for parts 1 to 6 of this project is written in Python 2.7.
The code for parts 8 to 10 of this project is written in Python 3.6.

Problem 1

***Part 1**: Dataset description*

The MNIST dataset contains 60,000 training images and 10,000 test images of handwritten digits 0-9. Shown Below is the breakdown of the number of images available for each digit:

```
Number of digit 0 training images: 5923
Number of digit 1 training images: 6742
Number of digit 2 training images: 5958
Number of digit 3 training images: 6131
Number of digit 4 training images: 5842
Number of digit 5 training images: 5421
Number of digit 6 training images: 5918
Number of digit 7 training images: 6265
Number of digit 8 training images: 5851
Number of digit 9 training images: 5949
Total Number of Training Images: 60000
Number of digit 0 test images: 980
Number of digit 1 test images: 1135
Number of digit 2 test images: 1032
Number of digit 3 test images: 1010
Number of digit 4 test images: 982
Number of digit 5 test images: 892
Number of digit 6 test images: 958
Number of digit 7 test images: 1028
Number of digit 8 test images: 974
Number of digit 9 test images: 1009
Total Number of Test Images: 10000
```

Shown below are examples of the handwritten digits in the dataset.

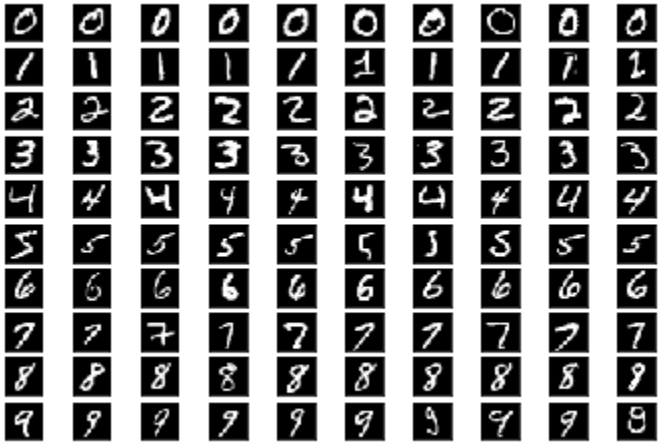


Figure 1: Samples of handwritten digits in the MNIST dataset

Problem 2

***Part 2**: Implement Basic Neural Network*

To generate the output of a single neuron in a neural network layer, the $784 \times M$ matrix of input images is mutiplied with the $n \times 784$ matrix of weights and added to a $N \times 1$ bias vector to produce an $N \times M$ output. The output will be transformed by the softmax function to produce an output between 0 and 1. Shown below are the implementation of the codes for the output of a single neuron and the softmax function.

```
def linear_neuron(X, weights, bias):
    '''
    X: is the input matrix of images - 784xM matrix
    weights: is the matrix of weights - Nx784 matrix
    bias: an Nx1 matrix

    Returns an NxM matrix

    where N is number of outputs (10 for digits) and M is number of images
    '''
    return np.dot(weights, X) + bias

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 (10 for digits
        ), and M
    is the number of cases (number of images)'''
    return exp(y)/tile(sum(exp(y),0), (len(y),1))
```

Problem 3

***Part 3 (a)**: Compute the gradient of the cost function with respect to the weight w_{ij} and bias b_j*

For a single image:

$C = \sum_{j=1}^{N} y_j \log p_j$

$p_k = \frac{e^{o_k}}{\sum_{l=1}^{N} e^{o_l}}$

$o_j = \sum_{m=1}^{M} w_{mj} + b_j$

Taking the partial derivative of the cost with respect to the weight w_{ij}:

$\frac{\partial C}{\partial w_{ij}} = -\sum_{j=1}^{N} \frac{\partial C}{\partial p_k} \frac{\partial p_k}{\partial o_j} \frac{\partial o_j}{\partial w_{ij}}$

$\frac{\partial C}{\partial p_k} = \frac{y_k}{p_k}$

if k = j

$\frac{\partial p_k}{\partial o_j} = \frac{e^{o_j}}{\sum_{l=1}^{N} e^{o_l}} - (\frac{e^{o_j}}{\sum_{l=1}^{N} e^{o_l}})^2$

$= p_j - (p_j)^2$

$= p_j(1 - p_j)^2$

if k $\neq$ j

$\frac{\partial p_k}{\partial o_j} = 0 - \frac{e^{o_k} e^{o_j}}{\sum_{l=1}^{N} e^{o_l}}$

$= -p_k p_j$

$\frac{\partial o_j}{\partial w_{ij}} = x_i$

$\frac{\partial C}{\partial w_{ij}} = -(\frac{y_0}{p_0}(-p_0 p_j) + ... + \frac{y_j}{p_j}(p_j)(1 - p_j) + ... + \frac{y_N}{p_N}(-p_N p_j))x_i$

$= (y_0 p_j + ... + y_j(p_j - 1) + ... + y_N p_j)x_i$

$= (p_j(y_0 + ... + y_N) - y_j)x_i$

$= (p_j - y_j)x_i$

For the entire training set m=1, ... , M, the gradient of the cost function with respect to weights w_{ij}

$\nabla_{w_{ij}} C = \sum_{m=1}^{M} (p_j^{(m)} - y_j^{(m)})x_i^{(m)}$

$=(P - Y)X^T$

where P is the NxM prediction matrix, Y is the NxM one-hot matrix and X is the PxM matrix of input images. N refers to the number of output, M refers to the number input images and P is the number of pixels in an image.

For a single image, taking the partial derivative of the cost with respect to the bias b_j:

$\frac{\partial C}{\partial b_j} = -\sum_{j=1}^{N} \frac{\partial C}{\partial p_k} \frac{\partial p_k}{\partial o_j} \frac{\partial o_j}{\partial b_j}$

$\frac{\partial o_j}{\partial b_j} = 1$

$$\begin{aligned}\frac{\partial C}{\partial b_j} &= -(\frac{y_0}{p_0}(-p_0 p_j) + ... + \frac{y_j}{p_j}(p_j)(1-p_j) + ... + \frac{y_N}{p_N}(-p_N p_j)) \\ &= (y_0 p_j + ... + y_j(p_j - 1) + ... + y_N p_j) \\ &= (p_j(y_0 + ... + y_N) - y_j) \\ &= (p_j - y_j)\end{aligned}$$

For the entire training set m=1, ... , M, the gradient of the cost function with respect to the biases b_j

$$\begin{aligned}\nabla_{b_j} C = &\sum_{m=1}^{M} (p_j^{(m)} - y_j^{(m)}) x_i^{(m)} \\ =&(P - Y)(1...1)^T\end{aligned}$$

where P is the NxM prediction matrix, Y is the NxM one-hot matrix and $(1...1)^T$ is a vector of N "1". N refers to the number of output, M refers to the number input images and P is the number of pixels in an image.

***Part 3 (b)**: Write vectorized codes that compute the gradient of the cost function with respect to the weight w_{ij} and bias b_j*

The code for computing the gradient of the cost function with respect to the weights w_{ij} and the biases b_j is shown below

```
def gradient(X, weights, bias, Y_):
    '''
    X: the input matrix of images - 784xM matrix
    weights: the matrix of weights - Nx784 matrix
    bias: an Nx1 matrix
    Y_: the one-hot encoding matrix -NxM matrix

    Returns
    grad_w: an Nx784 matrix (same dimensions as weights) for gradient w.r.t. weights
    grad_b: an Nx1 matrix (same dimensions as weights) for gradient w.r.t. bias

    where N is number of outputs (10 for digits) and M is number of images
    '''
    L = linear_neuron(X, weights, bias)
    Y = softmax(L)
    grad_w = np.dot((Y-Y_), X.T)

    grad_b = np.dot((Y-Y_), np.ones((Y.shape[1], 1)))  #Check!!!

    return grad_w, grad_b
```

The gradient of the cost function with respect to both weights w_{ij} and the biases b_j can be approximated using finite differences. The code for implementing this approach is shown below.

```
def finite_difference(X, weights, bias, Y_, h):
    '''
    X: the input matrix of images: 784xM matrix
    weights: the matrix of weights: Nx784 matrix
    bias: an Nx1 matrix
    Y_: the one-hot encoding matrix: NxM matrix

    returns an Nx784 matrix (same dimensions as weights) for gradient w.r.t. weights
    and an Nx1 matrix (same dimensions as weights) for gradient w.r.t. bias

    where N is number of outputs (10 for digits) and M is number of images
    '''
    finite_grad_w = np.zeros(weights.shape)
    for i in range(weights.shape[0]):
        for j in range(weights.shape[1]):
            H = np.zeros(weights.shape)
            H[i][j] = h
            Y_plusH = softmax(linear_neuron(X, weights+H, bias))
            Y_minusH = softmax(linear_neuron(X, weights-H, bias))
            finite_grad_w[i][j] = (NLL(Y_plusH, Y_) - NLL(Y_minusH, Y_))/(2*h)
    finite_grad_bias = np.zeros(weights.shape[0])
    for i in range(weights.shape[0]):
```

```
        H = np.zeros(weights.shape[0])
        H[i] = h
        H = H.reshape(bias.shape[0], bias.shape[1])
        Y_plusH = softmax(linear_neuron(X, weights, bias+H))
        Y_minusH = softmax(linear_neuron(X, weights, bias-H))
        finite_grad_bias[i] = (NLL(Y_plusH, Y_) - NLL(Y_minusH, Y_))/(2*h)
    return finite_grad_w, finite_grad_bias.reshape(3,1)
```

Using h of 0.00001 and randomly generated input X, one-hot Y, weights and bias matricies, the results of the gradient with respect to weights and bias are shown below. Since the results generated usung the *gradient* function and the *finite_difference* function are virtually identical, we conclude that the gradient function is correctly implemented.

The actual gradient with respect to weights is
[[-0.36193322 -1.10805724 -0.4295545 -0.64038512 -0.90915913]
[-0.61200997 0.44645638 -0.51100466 0.0366156 -0.26232004]
[0.97394319 0.66160085 0.94055916 0.60376952 1.17147916]]
Using finite difference, the estimated gradient with respect to weights is
[[-0.36193322 -1.10805724 -0.4295545 -0.64038512 -0.90915913]
[-0.61200997 0.44645638 -0.51100466 0.0366156 -0.26232004]
[0.97394319 0.66160085 0.94055916 0.60376952 1.17147916]]
The actual gradient with respect to bias is
[[-0.85824383]
[-0.63481023]
[1.49305407]]
Using finite difference, the estimated gradient with respect to bias is
[[-0.85824383]
[-0.63481023]
[1.49305407]]

Problem 4

***Part 4**: Training the basic neural network with "vanila" gradient descent*

A basic neural network with no hidden layer was trained to classify the MNIST digits. The training set of images contains 48,000 images, while the validation set contains 12,000 images. The trained network would then be tested against a test set containing 10,000 images to verify its performance. To find the optimal initializations of the learning rate, a grid search was conducted. The weights matrix was initialized as an $N \times 784$ matrix of 0 since the network is small and there are no hidden layers. The biases was initialized as a vector of 1. The gradient descent algorithm would update the weights and biases at every iteration for a maximum of 5,000 iterations. The results of the grid search are shown below:

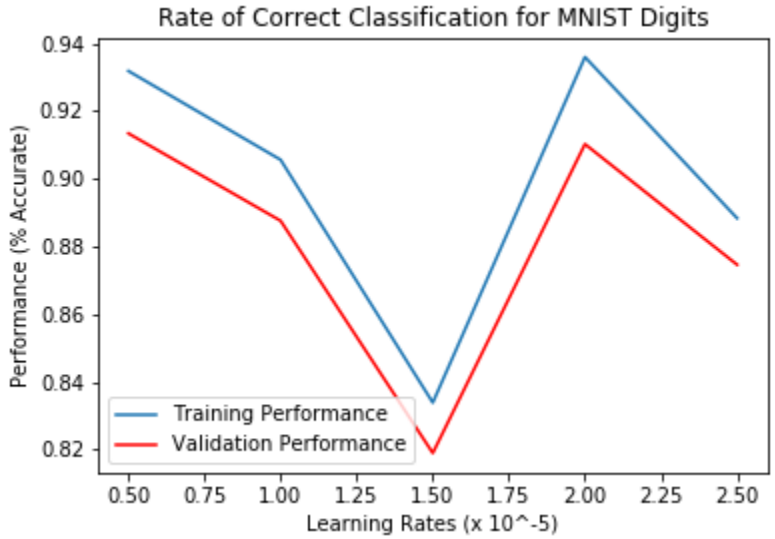


Figure 2: Classification accuracy vs learning rate

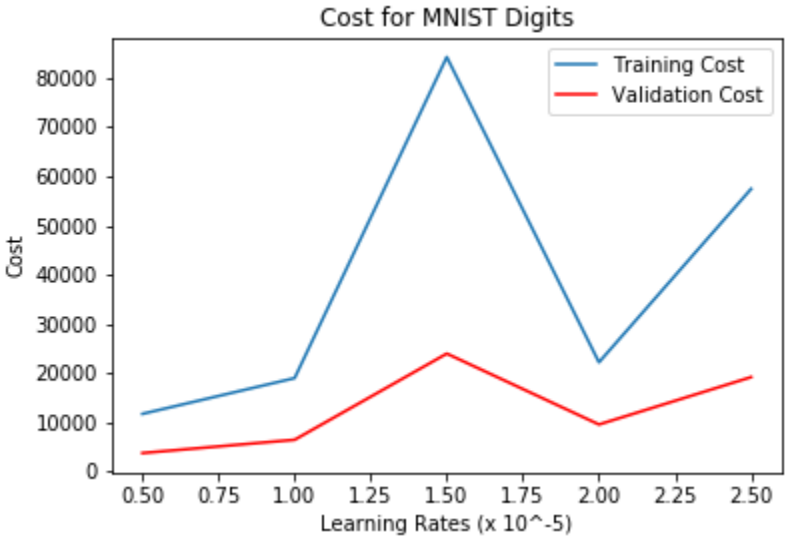


Figure 3: Classification cost vs learning rate

From the grid search, it was observed that using a learning rate of 0.000005 would generate the best accuracy and lowest cost. Using the optimal learning rate of 0.000005, the final results of the network's performance are shown below. Classification accuracy for both the training and validation images climbed above 90% after approximately 750 iterations of gradient descent. After 1000 iterations, both classification accuracy and cost exhibit small gradual improvements. The final classification accuracies are over 90% for the training, validation and test images.

```
Final Training Set Performance: 0.927742473174
Final Training Set Cost: 12607.4046517
Final Validation Set Performance: 0.918284048313
Final Validation Set Cost: 3380.84023078
Final Test Set Performance: 0.9221
Final Test Set Cost: 2727.79293586
```

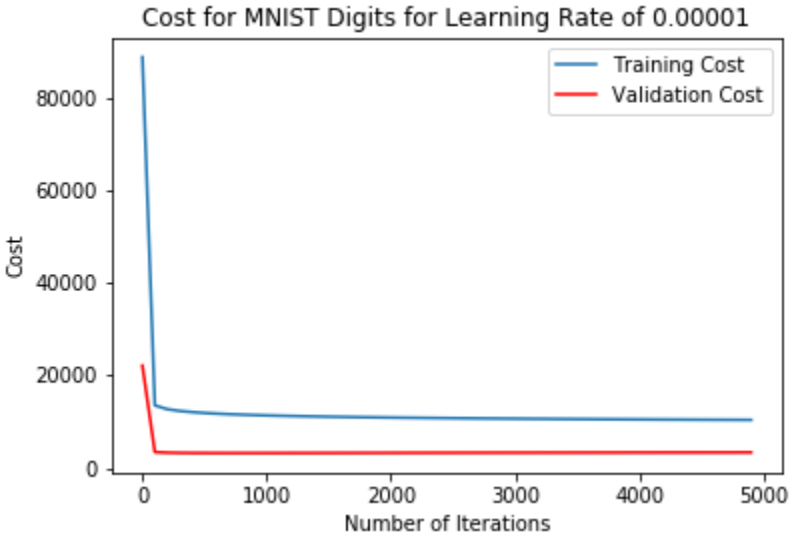


Figure 4: Classification accuracy vs iterations

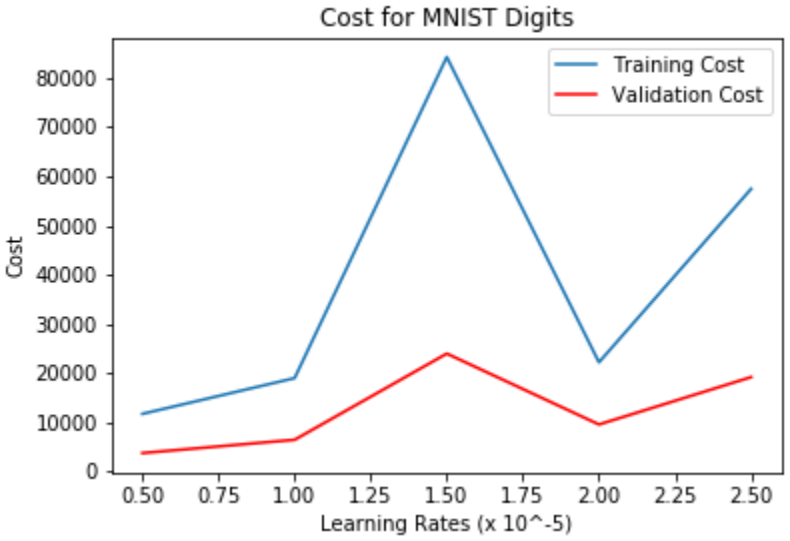
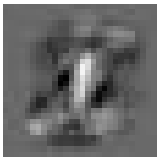


Figure 5: Classification cost vs iterations

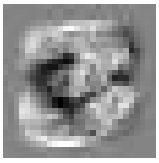
The weights connecting the input layer to the output layer are shown below. Each of the weight clearly resembles each of the 10 digits. These weights are extracted at the end of gradient descent.

(a) Weight for digit 0

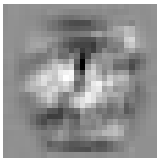


(b) Weight for digit 1

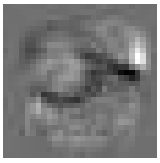
(c) Weight for digit 2



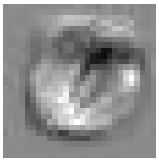
(d) Weight for digit 3



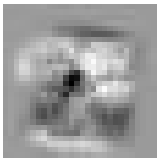
(e) Weight for digit 4



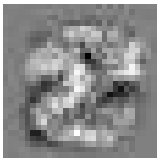
(f) Weight for digit 5



(g) Weight for digit 6



(h) Weight for digit 7



(i) Weight for digit 8

(j) Weight for digit 9

Figure 6: Visualizations of the weights connected to the output layer

Problem 5

***Part 5**: Training the basic neural network with momentum gradient descent*

Momentum gradient descent was implemented to help improve the speed of gradient computation in the network. Using momentum gradient descent would help to accerlerate gradient computation in the right direction. In every iteration of the momentum gradient descent, the gradient would be updated according to the formulas below:

$v = \gamma v + \alpha \frac{\partial C}{\partial W}$
$W = W - v$

where γ is the momentum term and is set to 0.9 in our implementation of the momentum gradient descent. The learning rate was set at 0.000005 as in part 4. The code for implementing momentum gradient descent is shown below:

```
def mom_gradient_descent(X, weights, bias, Y_, alpha, X_valid, Y_valid, max_iter, eps
    =1e-5, gamma=0.9):
    Vw = np.zeros(weights.shape)
    Vb = np.zeros((weights.shape[0], 1))

    diff_weights = np.array([eps + 1.0])  # make sure the initial loop goes through
    i = 0
    train_perf_record = []
    valid_perf_record = []
    train_cost_record = []
    valid_cost_record = []
    iterations = []
    while np.linalg.norm(diff_weights) > eps and i < max_iter:
        prev_weights = weights.copy()
        grad_w, grad_b = gradient(X, weights, bias, Y_)
        Vw = gamma*Vw + alpha*grad_w
        weights -= Vw
        Vb = gamma*Vb + alpha*grad_b
        bias -= Vb
        if i % 100 == 0:
            train_perf_record.append(performance(X, weights, bias, Y_))
            valid_perf_record.append(performance(X_valid, weights, bias, Y_valid))
            train_cost_record.append(cost(X, weights, bias, Y_))
            valid_cost_record.append(cost(X_valid, weights, bias, Y_valid))
            iterations.append(i)
        i += 1
        diff_weights = weights - prev_weights
    return weights, bias, train_perf_record, valid_perf_record, train_cost_record,
        valid_cost_record, iterations
```

The use of momentum gradient descent helped improved the speed of convergence in the computation of the gradient as compared to the "vanilla gradient descent". There is a slight improvememt to performance of the digit classification as well. The final performance of the network is shown below.

Final Training Set Performance: 0.938056047505
Final Training Set Cost: 10850.963966
Final Validation Set Performance: 0.91928363182
Final Validation Set Cost: 3378.91846084
Final Test Set Performance: 0.9243
Final Test Set Cost: 2727.15554639

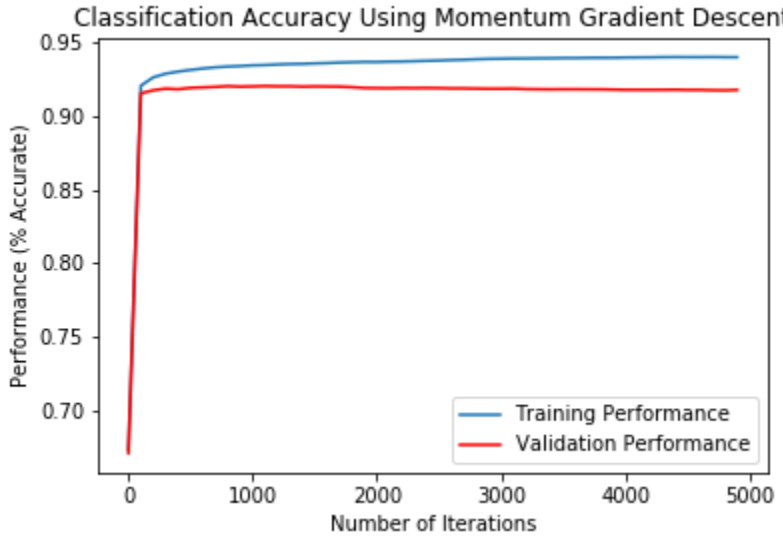


Figure 7: Classification accuracy vs iterations

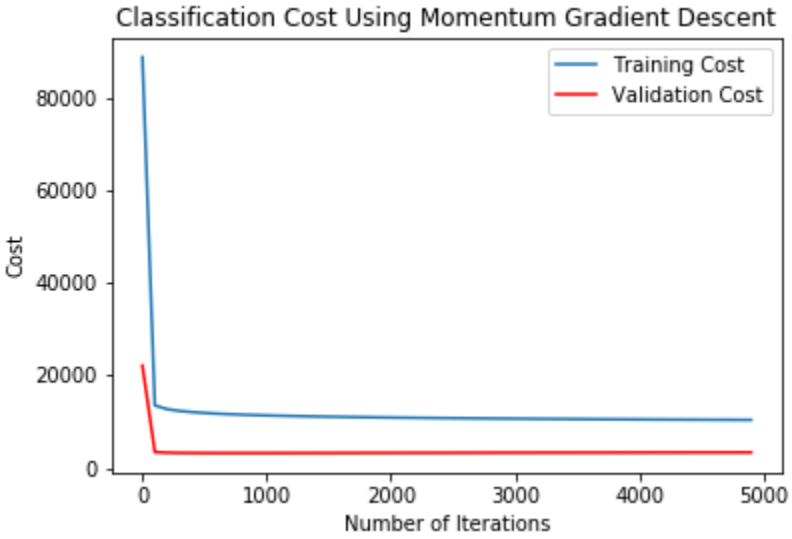


Figure 8: Classification cost vs iterations

Problem 6

***Part 6**: Demonstration of momentum gradient descent*

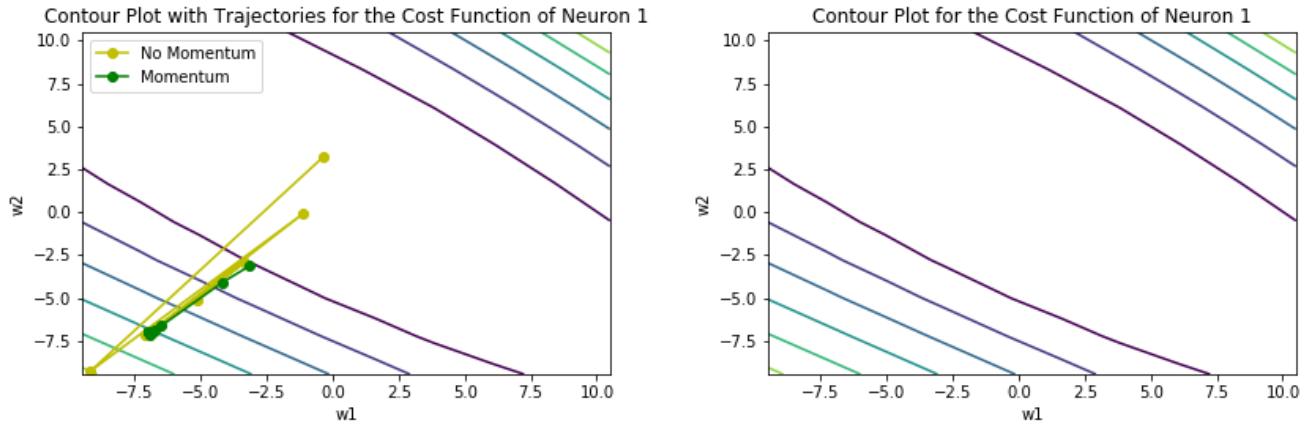


Figure 9: Contour plot for the centre 2 weights with and without trajectories

part6(d)
As we can see from the trajectory of the vanilla gradient descent, the gradient is of high curvature. Hence the the vanilla gradient descent oscillate a lot. However, the momentum gradient descent of these high curvature directions cancel each other out and hence provided a more stable progression towards the local minimum.

part6(e)
The learning rate for the vanilla gradient descent was set to 0.0026, the learning rate for the momentum gradient descent was 0.0034, k (number of iterations) was set to 15. If the learning rate for the momentum gradient descent was too low, the same as vanilla 0.0026 for example, the progression will be too slow to visualize, since the momentum canceled out the oscillation. All the points will be really close to each other. Hence in order to visualize the how momentum stablize the gradient desecnt, we need to increase the learning rate. Another bad example would be if we set both learning rate too low or too high, the points will be either really close to each other that we can't observe the differences, or the gradient descent will be oscillating too much. The learning rates combining with k in part 6(c) and (d) were tested to fit the width and height for the graph the best.

Note: The weights used in this visualization were obtained before the introduction of random seed in the code. Hence reproducing the csv files in part 5 would generate different plots as shown above

Problem 7

***Part 7**: Comparing Backpropagation versus computing the gradient with respect to each weight*

Assuming a network with N layers, K neurons per hidden layer and M outputs for the last layer. For simplicity, let's assume M ≈ K. Starting from the last layer, the gradient of the Cost with respect to each weight is:

$\frac{\partial C}{\partial w^{(N,j,k)}} = \frac{\partial C}{\partial o_k}\frac{\partial o_k}{\partial w^{(N,j,k)}}$
where $i = 1, ..., N$ and $j, k = 1, ..., K$

Assuming the computation of a partial derivative takes constant time, or $O(1)$ complexity, The above calculation would be $O(2) \approx O(1)$. Hence, computing the gradient of the cost function with respect to every weight in the layer will take $O(M) \approx O(K)$.

For the second last layer, the gradient of the Cost with respect to every weight is:

$$\begin{aligned}\frac{\partial C}{\partial w^{(N-1,j,k)}} &= \frac{\partial C}{\partial o_1}\frac{\partial o_1}{\partial \partial h_{N-1,j}}\frac{\partial h_{N-1,j}}{\partial w^{(N-1,j,k)}} + ... + \frac{\partial C}{\partial o_M}\frac{\partial o_M}{\partial \partial h_{N-1,j}}\frac{\partial h_{N-1,j}}{\partial w^{(N-1,j,k)}} \\ &= (\sum_{l=1}^{M}\frac{\partial C}{\partial o_l}\frac{\partial o_l}{\partial \partial h_{N-1,j}}\frac{\partial h_{N-1,j}}{\partial w^{(N-1,j,k)}})\frac{\partial h_{N-1,j}}{\partial w^{(N-1,j,k)}}\end{aligned}$$

Hence, the computation of the gradient with respect to each weight would be $O(M) \approx O(K)$. Since there are K neurons in this layer, the computation of the gradient with respect to each weight for the entire layer would be $O(K^2)$.

For the thrid last layer, the gradient of the Cost with respect to every weight is:

$$\frac{\partial C}{\partial w^{(N-2,j,k)}} = (\sum_{j=1}^{M}(\sum_{l=1}^{M}\frac{\partial C}{\partial o_l}\frac{\partial o_l}{\partial \partial h_{N-1,j}})\frac{\partial h_{N-1,j}}{\partial h_{N-2,j}})\frac{\partial h_{N-2,j}}{\partial w^{(N-2,j,k)}}$$

Hence, the computation of the gradient with respect to each weight individually would be $O(M*K) \approx O(K^2)$. Since there are K neurons in this layer, the computation of the gradient with respect to each weight individually for the entire layer would be $O(K^3)$.

Following this trend, we note that for an N layer network, the computation of the gradient with respect to each weight individually would be approximately $O(K^N)$

On the other hand, let us now consider a fully vectorized backpropagation algorithm that stores the gradient of the weight at every layer. Once again, we assume that the number of outputs, M, is approximately to the number of neurons, K, at each layer for simplicity. At every layer $N, N-1, ..., 2$ backpropagation algorithm, we compute:

$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l)$

where δ^l is the $K \times 1$ matrix containing the error of all K neurons in l layer, w^{l+1} is the $K \times K$ matrix of weights in the l layer, $\odot$ is the element-wise product, also known as Hadamard product, σ is the activation function and $z^l = w^l a^{l-1} + b^l$. a^{l-1} is the activation from the pevious layer and $\sigma'(z^l)$ is a $K \times 1$ matrix. Hence, the computation of δ^l produces a $K \times K$ output matrix and has $O(K^3)$ complexity.

The gradient of the cost with respect to each weight is computed as follows:

$\frac{\partial C}{\partial w^{l,j,k}} = a^{l-1,k}\delta^{l,j}$

Since there are K^2 weights in every layer, the computation of the gradient of the cost with repect to weight will have $O(K^2)$ complexity. The most computationally expensive step of backpropagration takes $O(K^3)$, hence backpropagation has complexity of $O(K^3)$, which is significantly less expensive as compared to computing the gradient with respect to each weight individually for a network with 4 or more layers.

Problem 8

***Part 8**: Facial classification with single-hidden layer*

We will now perform the task of facial classification with a single-hidden layer with Pytorch for the following actors/actresses: Lorraine Bracco, Peri Gilpin, Angie Harmon, Alec Baldwin, Bill Hader and Steve Carell. The images were obtained from facescrub, and the URLs used to download the images are contained in the files **facescrub_actors.txt** and **facescrub_actresses.txt**. Bad images were removed by comparing the SHA-256 codes of the images with the SHA-256 codes provided in the files **facescrub_actors.txt** and **facescrub_actresses.txt** . 70 images were used from each actor as training images, with the exception of Peri Gilpin where we used 37 images. This was due to insufficient images available for Peri Gilpin. For each actor, 10 images were included in the validation set and 10 images were used for the test set. The images were converted into greyscale and resized to size 64 by 64. We observed that using size 64 by 64 images provided better results as compared to using size 32 by 32 images. Gradient descent was optimized using the Adam optimizer and using mini-batches of training images. Experiments were conducted to find the optimal learning rate, mini-batch size, number of hidden neurons and activation function. Shown below are the range of parameters that were tried:

Learning rates: 0.01, 0.001 and 0.0001
Number of Hidden Units: 64, 128, 512, 1024 and 4096
Mini-batch size: 32, 64, 128
Activation function: ReLU, Tanh

The optimal network uses a learning rate of 0.0001, 128 hidden units, mini-batch size of 128 and the ReLU activation function. Shown below is the performance of the network:

Final Training Set Performance: 1.0
Final Validation Set Performance: 0.75
Final Test Set Performance: 0.8

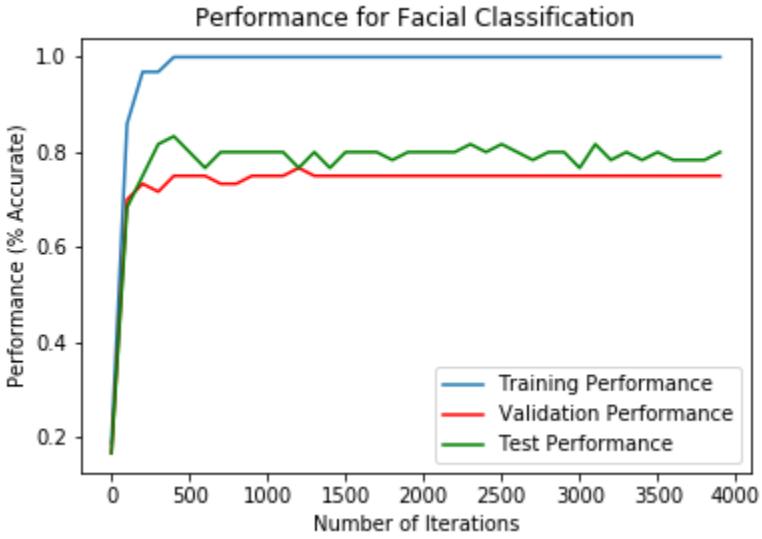


Figure 10: Classification accuracy vs iterations

Problem 9

***Part 9**: Visualizations of the hidden layer weights*

For this section, we will visualize the weights of the hidden layer that are most useful for classifying input photos for the actor Steve Carell and actress Lorraine Bracco. The three most useful hidden weights for each actor will be visualized.

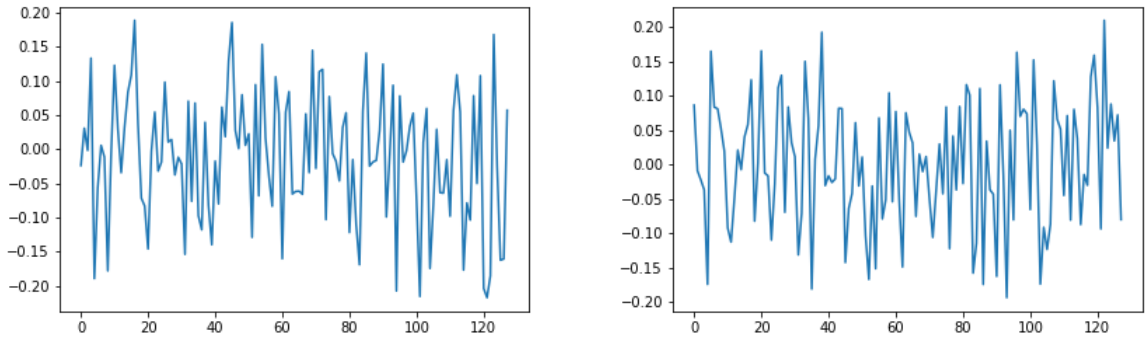


Figure 11: Weights for the connection from hidden layer to output for Bracco and Carell

We extracted the weights from the hidden layer to the output and picked the hidden units corresponding to the 3 most significant weights, since these weights indicated that the corresponding units contributed the most to the output. Since we used one-hot decoding for the output, by selecting Bracco and Carell seperately, each connection from hidden layer to Bracco and Carell is just simply a linear combination, which can be visualized as above. The following are the visualizaion for the useful hidden units.

Figure 12: The 3 most useful hidden units for detecting Bracco

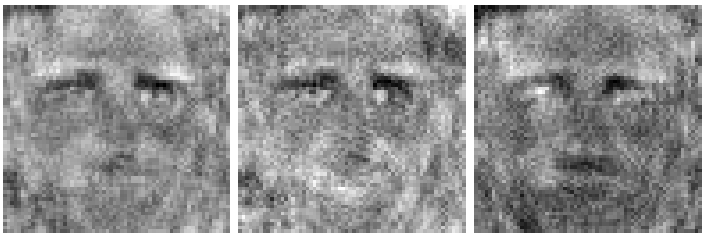


Figure 13: The 3 most useful hidden units for detecting Carell

Problem 10

***Part 10**: Facial Classification with AlexNet*

AlexNet, a convolutional neural network, was used to perform facial classification for the same actors and actresses. For this task, we used rgb coloured version of the original images and resized them to 227 by 227 by 3. As with part 8, 70 images were used from each actor as training images, with the exception of Peri Gilpin where we used 37 images. For each actor, 10 images were included in the validation set and 10 images were used for the test set.

To extract the values of the activations of AlexNet after the $4th$ convolution layer, we set the features for AlexNet to only contain the first 4 layers, and define the forward function to only run the features. Hence when we input the images to AlexNet and run forward, the output will be the activations from the $4th$ convolution layer.

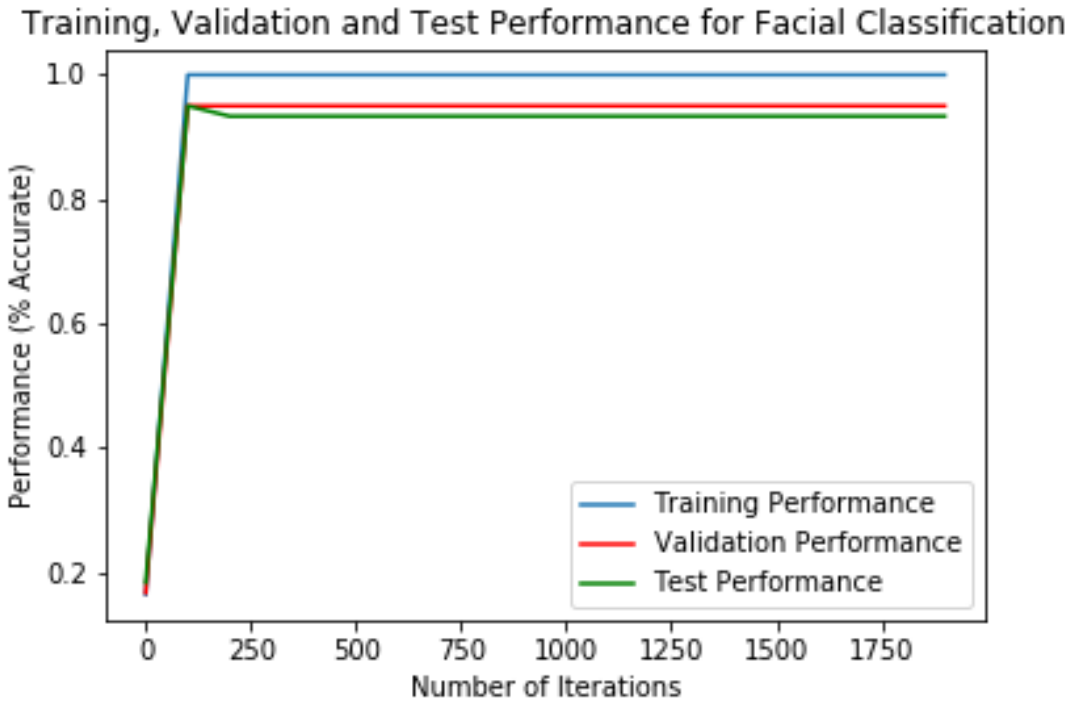


Figure 14: Performance of the full connection after AlexNet

We used the same system as part 8, but set all the inputs from the training, validation and the test set to run through our MyAlexNet (only includes the first 4 convolution layer from AlexNet), and let the output of these to be the input of the system from part 8. In addition, we changed the dimension of the input to 9217 ($256 * 6 * 6 + 1$). We experimented with different parameters and found that the optimal parameters are as follows:

Learning rate: 0.0001
Number of hidden units: 64
Mini-batch size: 128

Activation function: ReLU

As we can see from the above figure, AlexNet vastly outformed the single hidden-layer network used in part 8. The final performance of the AlexNet is shown below:

```
Final Training Set Performance: 1.0
Final Validation Set Performance: 0.95
Final Test Set Performance: 0.933333333333
```