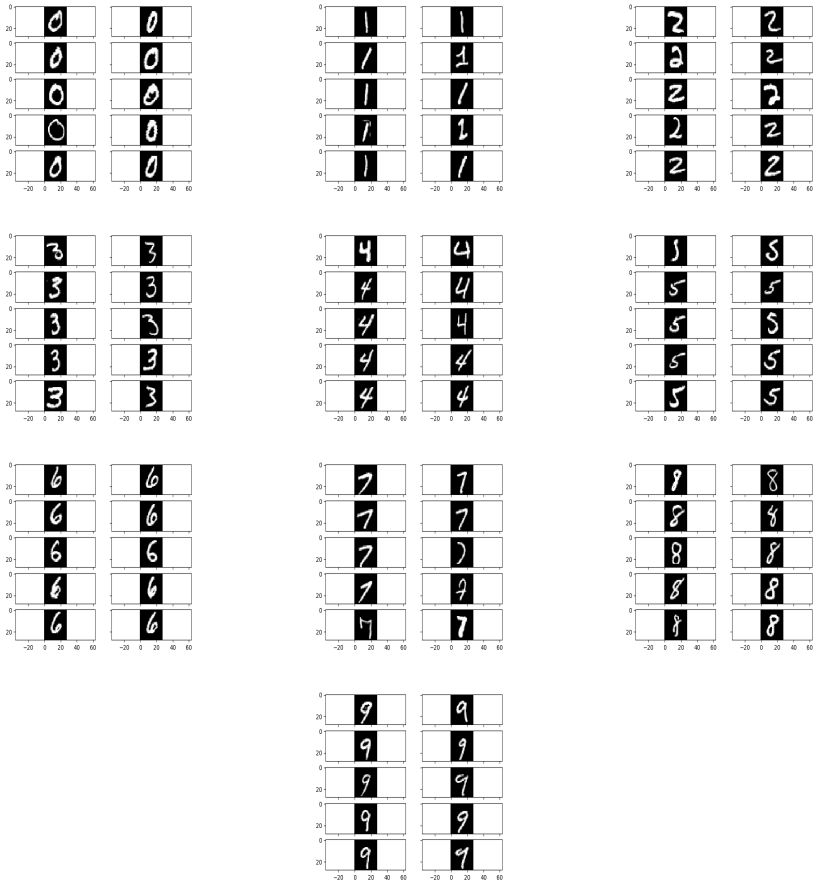
CSC411-Project 2

Amanda Hassoun, Tony Ye

February 18th, 2018

Part 1



Dataset Description

For parts 1-6, we are working with the MNIST dataset widely used in computer science literature, and available at:

`http://yann.lecun.com/exdb/mnist/`

This dataset has images of handwritten digits 0 through 9, with a training set of 60,000 and a test set of 10,000. The quality of this dataset is quite good, because the digits are in their upright position, normalized to the same size, and centered. There is fairly good alignment, and no further processing is required.

A challenge in classifying digits using this dataset could be that some of the digits are very similar in writing. For example, some of the 7's could be mistaken for 1's and vice versa. The 8 at the bottom left corner could be mistaken for a 1 as well.The 3 visually resembles an 8 since it corresponds to half of it. Some of the 5's that are tilted to the right resemble a 6. So there is a lot of overlap between the digits that could cause the classifier to make mistakes.

Part 2

The function that computes the single layer neural network is presented below:

```
def forward(x, W0, b0 ):
    L0 = dot(W0.T, x) + b0
    output = softmax(L0)

    return L0, output

np.random.seed(100)

W0 = np.random.randn(784,10)
b0 = np.random.randn(10,1)

#200 images per digit

x = np.zeros((784,2000))

#Construct matrix
for i in range(10):
    for j in range(200):
        image = M["train" + str(i)][j]/255.0
        x[:,j] = image.flatten().T

L0, output = forward(x, W0, b0)
```

Part 3

a)

Notation: x_j is the jth input unit (pixel) $j \in \{1 \dots 784\}$. w_{ji} is the weight from the jth input to the ith output unit, $j \in \{1 \dots 784\}, i \in \{1 \dots 10\}$. o_i is the ith output unit. p_j is the softmax probability for class $j, j \in \{1 \dots 10\}$.

Cost is the sum of negative log probabilities, over all the training cases:

$$C = -\sum_h \sum_i {y_i}^h \log {p_i}^h$$

$$\frac{\partial C}{\partial w_{ji}} = \sum_h \frac{\partial C}{\partial o_i^h} \frac{\partial o_i^h}{\partial w_{ji}}$$

Consider cost of one training example,

$$c = -\sum_i y_i \log p_i$$

$$o_i = \sum_j w_{ji} x_j + b_i$$

$$\frac{\partial c}{\partial o_i} = \sum_j \frac{\partial c}{\partial p_j} \frac{\partial p_j}{\partial o_i}$$

$$\frac{\partial c}{\partial p_j} = -\frac{y_j}{p_j}$$

Perform the derivative quotient rules for expression of p_j with respect to o_i:

$$p_j = \frac{e^{o_j}}{\sum_l e^{o_l}}$$

When $i = j$,

$$\begin{aligned} \frac{\partial p_j}{\partial o_i} &= \frac{e^{o_i} \sum_l e^{o_l} - (e^{o_i})^2}{(\sum_l (e^{o_l})^2} \\ &= p_i(1 - p_i) \end{aligned}$$

When $i \neq j$,

$$\begin{aligned} \frac{\partial p_j}{\partial o_i} &= \frac{0 - (e^{o_i})(e^{o_j})}{(\sum_l (e^{o_l})^2} \\ &= -p_i p_j \end{aligned}$$

$$\begin{aligned}\frac{\partial c}{\partial o_i} &= -\frac{y_i}{p_i}p_i(1-p_i) + \sum_{j\neq i} -\frac{y_j}{p_j}(-p_i p_j) \\ &= -y_i + y_i p_i + \sum_{j\neq i} p_i y_j \\ &= -y_i + \sum_j p_i y_j \\ &= -y_i + p_i\end{aligned}$$

The last step follows from the fact that $\sum_j y_j = 1$ because of one hot encoding of the response vector (so one element is equal to 1 and the rest equal to 0).

Lastly: $\frac{\partial o_i}{w_{ji}} = x_j$

Combining the above results, we obtain:

$$\frac{\partial C}{\partial w_{ji}} = \sum_h (p_i^h - y_i^h) x_j^h$$

b)

Vectorized code that computes the gradient of the cost function with respect to the weights and biases of the network:

```
#Helper functions

def make_set(set_size, set_type):
    """Construct matrix based on set size."""

    x = np.zeros((784,10*set_size))
    y = np.zeros((10,10*set_size))
    col = 0

    #Construct set
    for i in range(10):
        for j in range(set_size):
            #image = M[set_type + str(i)][j].reshape((28,28))/255.0
            image = M[set_type + str(i)][j]/255.0
            x[:,col] = image.flatten().T
            col += 1
        y[i][i*set_size:(i+1)*set_size] = 1

    return x,y

def part3_grad(x, y, output):
    return dot(x, (output-y).T), sum(output-y, axis=1)
```

```
# Verify gradient for w
def finite_diffw(x, y_, W0, b0):
    h = np.zeros((784,10))
    step = 1e-9

    shape_h = h.shape
    for i in range(shape_h[0]):
        for j in range(shape_h[1]):
            h[i,j] = step
            L0, f_h = forward(x, W0 + h, b0)
            L1, f_ = forward(x, W0, b0)
            dw, db = part3_grad(x, y_, f_)
            finite = ((NLL(f_h, y_) - NLL(f_, y_))/step)
            print "Finite differences at (%d, %d): %.5f" % (i,j,finite)
            print "Vectorized gradient at (%d, %d): %.5f" % (i,j, dw[i,j])
            h = np.zeros((784,10))

# Verify gradient for b
def finite_diffb(x, y_, W0, b0):
    h = np.zeros((10,1))
    step = 1e-9

    shape_h = h.shape
    for i in range(shape_h[0]):
        for j in range(shape_h[1]):
            h[i,j] = step
            L0, f_h = forward(x, W0, b0 + h)
            L1, f_ = forward(x, W0, b0)
            dw, db = part3_grad(x, y_, f_)
            finite = ((NLL(f_h, y_) - NLL(f_, y_))/step)
            print "Finite differences at (%d, %d): %.5f" % (i,j,finite)
            print "Vectorized gradient at (%d, %d): %.5f" % (i,j, db[i])
            h = np.zeros((10,1))

#Part 3 - Code that runs the functions
#200 images per digit
x , y = make_set(200, "train")
L0, output = forward(x, W0, b0)

finite_diffw(x, y, W0, b0)
finite_diffb(x, y, W0, b0)
```

Some output to verify correctness of vectorized gradient w.r.t to the weights:

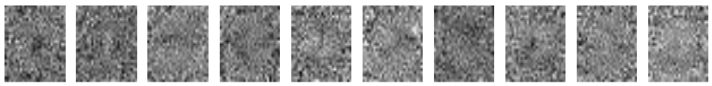
```
Finite differences at (124, 0): -13.25679
Vectorized gradient at (124, 0): -13.26044
Finite differences at (124, 1): 19.27401
Vectorized gradient at (124, 1): 19.27830
Finite differences at (124, 2): -60.07758
Vectorized gradient at (124, 2): -60.07728
Finite differences at (124, 3): -37.11466
Vectorized gradient at (124, 3): -37.11739
```

Some output to verify correctness of vectorized gradient w.r.t to the bias:

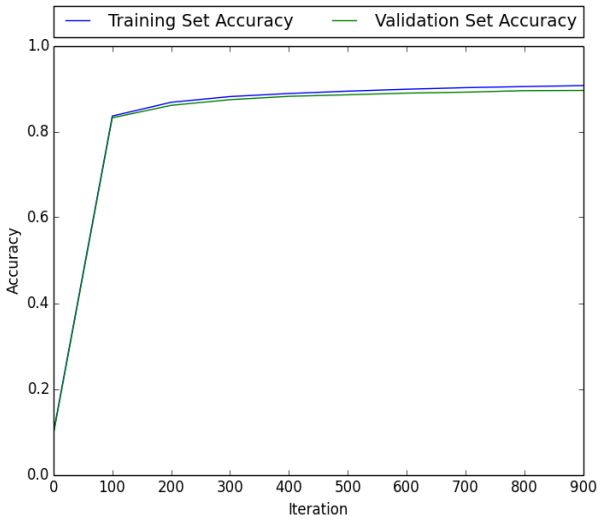
```
Finite differences at (0, 0): -146.07213
Vectorized gradient at (0, 0): -146.09808
Finite differences at (1, 0): 169.34064
Vectorized gradient at (1, 0): 169.35725
Finite differences at (2, 0): -178.71935
Vectorized gradient at (2, 0): -178.70593
Finite differences at (3, 0): -143.71108
Vectorized gradient at (3, 0): -143.68174
Finite differences at (4, 0): -28.29620
Vectorized gradient at (4, 0): -28.26135
Finite differences at (5, 0): -139.05810
Vectorized gradient at (5, 0): -139.03063
Finite differences at (6, 0): -190.37179
Vectorized gradient at (6, 0): -190.38342
Finite differences at (7, 0): 526.49557
Vectorized gradient at (7, 0): 526.56680
Finite differences at (8, 0): 174.69210
Vectorized gradient at (8, 0): 174.70549
Finite differences at (9, 0): -44.48520
Vectorized gradient at (9, 0): -44.46840
```

Part 4

Weights in output units (in sequence from units 0 to 9):



Learning curve for the training and validation set:



The above is a learning curve for the training and test set. In around 100 epochs (iterations, since we're using batch gradient descent), the accuracy is just over 80%. Accuracy begins to flatten out from that point onwards. The training set accuracy continues to increase slightly as the classifier becomes overfitted to the training set. However, overfitting doesn't seem to be a major issue in this case, mostly because we are using such a large dataset for the training. If the training set size is smaller, we would expect a more pronounced discrepancy between the training and validation set accuracy as iterations increase. Overfitting is nevertheless discernible because of the slight increase in the difference between the training set accuracy and the test set accuracy as the number of iterations increases.

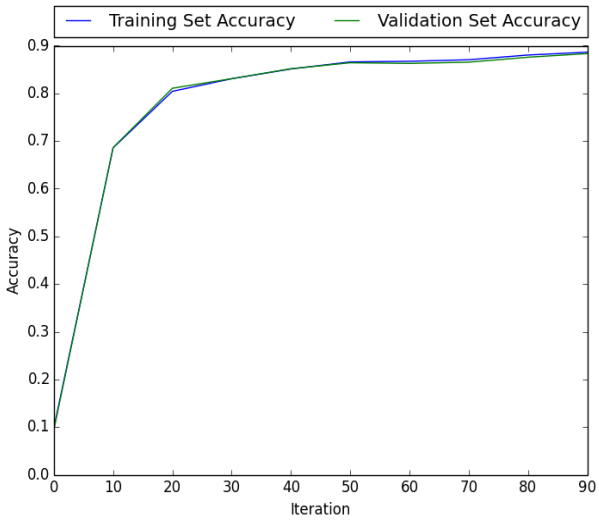
Initially we used a learning rate (α) of 1e-2, which was far to large. The learning rate was reduced to around 1e-5 because we realized that the cost was oscillating intensely. 1e-5 was a suitable rate because the cost and accuracies converged well within a reasonable number of iterations. However we found that what worked best was an adaptive learning rate algorithm, which was used in the end to produce the learning curve. We initialized the learning rate to 1e-5 and at each iteration of gradient descent, a check is done to make sure that error decreased from the previous iteration. If so, then learning rate increases by 10%. If not, then the weights and biases are set to the previous iteration

and learning rate reduced by 50%. This allowed convergence to be very quick and thus computationally efficient.

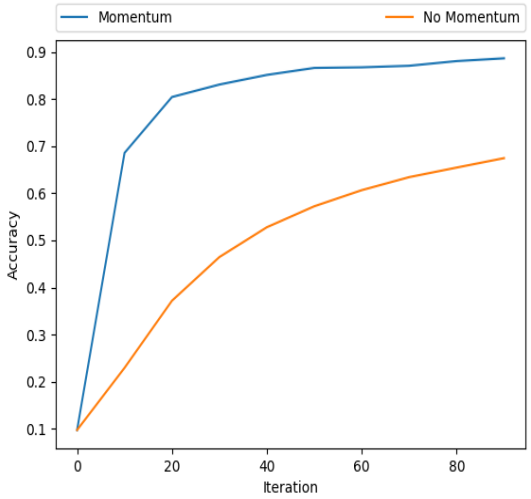
The initial weights were randomly sampled from the a normal distribution with mean 0 and variance 100. This ensured that not all weights will be moved in the same direction during gradient descent.

Part 5

The below graph shows the learning curves for gradient descent with momentum:



If we compare this graph to the learning curves for gradient descent without momentum, we see that the accuracy for both the test and training sets converge a lot quicker when we use momentum.



As a better means for comparison, the above graph shows the learning curve on the training set for gradient descent using momentum and without using momentum. Evidently, momentum allows for much quicker convergence. We see that within around 20 iterations, gradient descent with momentum yields an accuracy of 80% already while for gradient descent without momentum, the accuracy is still very low at 40%.

We added an additional parameter to the gradient descent method, where if momentum is set to 'True', update rule for momentum will be in effect. Vectorized code for implementing momentum:

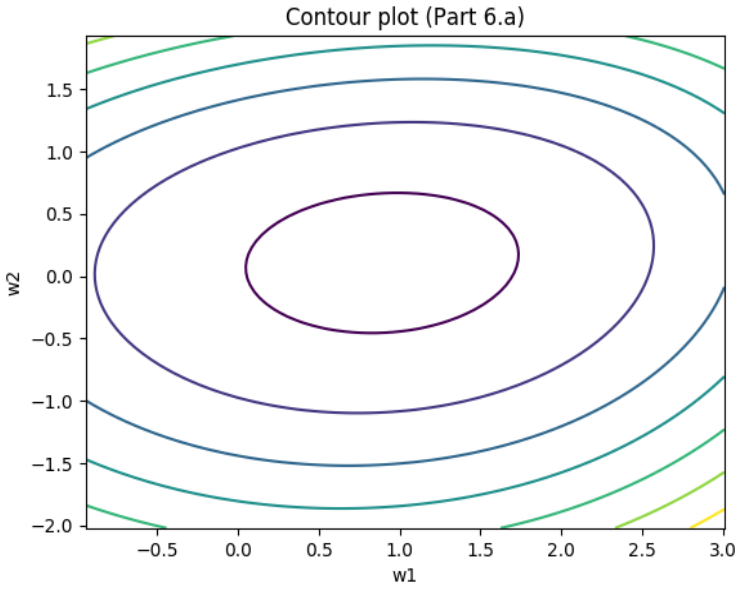
```
def grad_descent(x, y, init_weights, init_bias,
                 alpha, max_iter, momentum=False):
    # x: your image pixel intensity matrix
    # y: actual labels

    EPS = 1e-10
    prev_t = init_weights-10*EPS
    w = init_weights.copy()
    b = init_bias.copy()
    perf_valid = []
    perf_train = []
    iterations = []
    iter_  = 0
    firstTime = True
    v1 = np.zeros_like(w)
    v2 = np.zeros_like(b)

    #Generate validation set
    x_valid , y_valid = make_set(800, "test")

    while iter_ < max_iter and norm(w - prev_t) >  EPS:
        prev_w = w.copy()
        prev_b = b.copy()
        L_, output = forward(x, prev_w, prev_b)
        dw_prev, db_prev = part3_grad(x, y, output)

        L_, output = forward(x, w, b)
        dw, db = part3_grad(x, y, output)

        if (firstTime):
            v1 = np.zeros_like(dw)
            v2 = np.zeros_like(db)
            firstTime = False

        if momentum:
            v1 = 0.99*v1 + alpha*dw
            w -= v1
            v2 = 0.99*v2 + alpha*np.squeeze(array([db]))
            b -= array([v2]).T
        else:
            w -= alpha*np.squeeze(array([dw]))
```

```
            b -= alpha*array([db]).T

        if iter_ % 1 == 0:
            train_accuracy = getPredictionAccuracy(x,y,w,b)
            valid_accuracy = getPredictionAccuracy(x_valid, y_valid, w, b)
            perf_train.append(train_accuracy)
            perf_valid.append(valid_accuracy)
            iterations.append(iter_)

            print "Iter", iter_
            print "Cost %.2f " % (f(x,y,w,b))
        iter_ += 1

    plot_perfomance(iterations,perf_train, perf_valid)

    return w, b, iterations, perf_train
```

Part 6

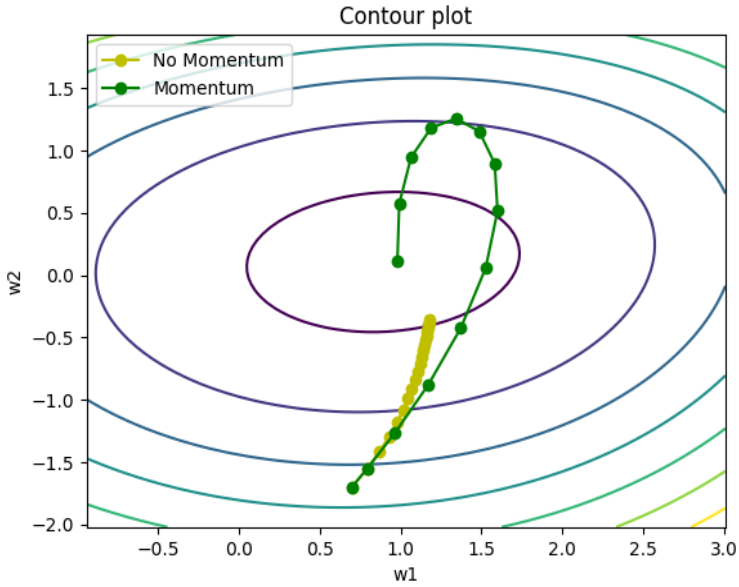
a)

We picked the weights between input pixel 400 and output unit 2, and pixel 300 and output unit 6. The following is a contour plot of the cost function when these two weights are varied from the optimum at approximately $(1.059, -0.022)$:

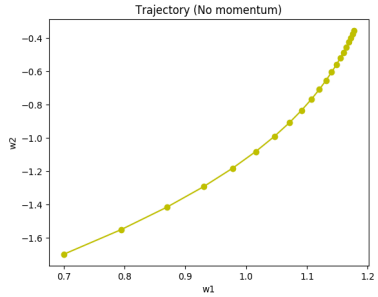


b,c,d)

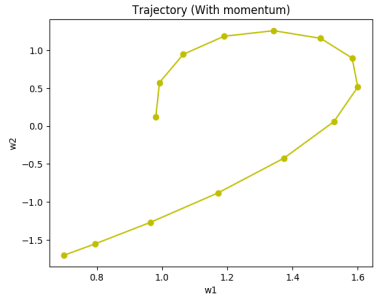
The following is the contour plot and trajectory by running gradient descent without momentum and with momentum. The individual trajectories follow.



The following is the trajectory by running vanilla gradient descent without momentum $K = 20$. We see that it is moving slowly toward the local minimum of $(1.059, -0.022)$.



The following is the trajectory by running gradient descent with momentum $K = 14$. We see that it has reached close to the local minimum of $(1.059, -0.022)$.

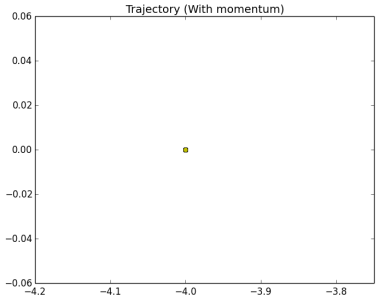


For gradient descent without momentum, we see that it moves a lot slower

toward the local minimum, whereas for momentum, it moves rather quickly to optimum. The difference is accounted for by the fact that momentum ensures movement in the direction of local minimum by using the gradient from the previous iteration. This shows that momentum is beneficial and allows for faster convergence than gradient descent without momentum.

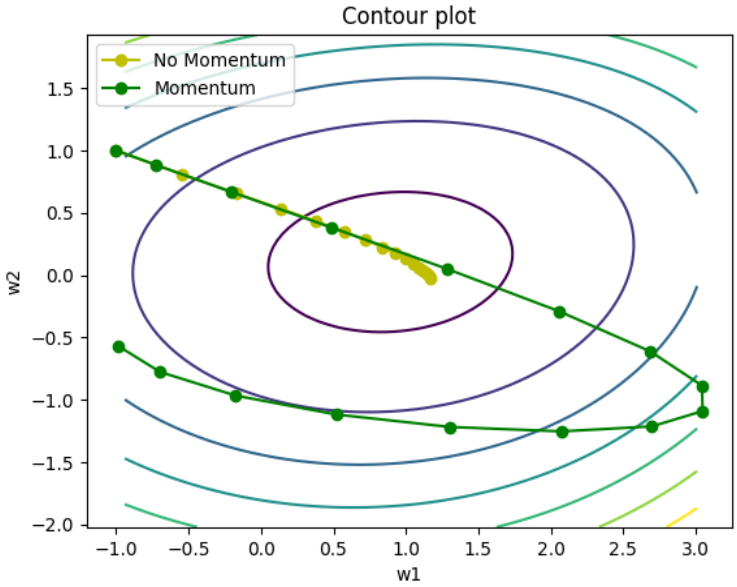
e)

In order to find appropriate weights, we chose the ones that were located near the centre of the image as suggested. Here is the trajectory of the gradient for weights located near the corners (where the pixels are all black):



As demonstrated by the trajectory in the graph, the weight barely changes for the dark pixels (there's actually 20 iterations of gradient descent in the graph). We first chose weights that belonged to the same output unit and ran into some trouble because they were too close to each other (contour plot was a bunch of straight lines). So we then decided to pick weights that were further apart (and specifically that weren't for the same output unit). The other variable we needed to adjust was alpha: we had to set it to a smaller value when using gradient descent with (or without) momentum or else the cost kept oscillating.

Another situation under which the benefit of momentum is not clear is demonstrated below:



This is when the initial point is set to $(w_1 = -1, w_2 = 1)$ and we perform gradient descent with momentum and without momentum for 15 iterations. Clearly, without momentum, it converged close to the optimum, whereas for momentum, it went straight through the optimum. This is because the gradient in the direction of the local minimum was very large, so there is 'too much' momentum when doing gradient descent, causing it to overshoot the local optimum. Also the learning rate could have been a bit too large, which caused this issue for gradient descent with momentum.

Part 7

We will be using a big-O analysis for the vectorized backpropagation, where the gradients are cached (only computed once), and compare it to the case where the gradients are computed individually for each weight.

Case 1: Vectorized Backpropagation

Notation

Following is the notation we'll use in the analysis:
$\vec{h_n}$ - units in layer n, $\forall n = 0 \dots N$
$\vec{w_n}$ - weights for units in layer n to units in layer $n+1$ $\forall n = 0 \dots N-1$
$\vec{b_n}$ - bias terms for units in layer n $\forall n = 1 \dots N$
$\vec{y}$ is the target vector
$C = \sum(\vec{h}_N - \vec{y})^2$

Analysis

We assume that matrix transpose is constant time and therefore irrelevant for analysis. We further make the simplification that for matrix multiplication between two matrices of dimension $K \times K$, the complexity is $O(K^2)$ (which is reasonable with advanced matrix multiplication algorithms). To simplify analysis we will assume the activation function applied to each unit in a layer is linear. Other activation functions involve the chain rule with the derivative, which would not actually change the overall complexity. We also assume that the loss function is the sum of squared loss function.

$$\frac{\partial C}{\partial \vec{h_N}} = 2(\vec{h_N} - \vec{y})$$

$$\vec{h}_n = \vec{w}_{n-1}\vec{h}_{n-1} + \vec{b}_n$$

Define a recursive formula:

$$\vec{U}_n = \frac{\partial C}{\partial \vec{h_n}}$$

$$\vec{U}_n = \vec{U}_{n+1}\frac{\partial \vec{h}_{n+1}}{\partial \vec{h}_n} = \vec{U}_{n+1}\vec{w}_n$$

Then for the gradient of the weights:

$$\frac{\partial \vec{C}}{\partial \vec{w}_n} = \frac{\partial \vec{C}}{\partial \vec{h}_{n+1}} \cdot \frac{\partial \vec{h}_{n+1}}{\partial \vec{w}_n} = \vec{U}_{n+1} \cdot \vec{h}_n$$

For the gradient of the biases:

$$\frac{\partial \vec{C}}{\partial \vec{b_n}} = \frac{\partial \vec{C}}{\partial \vec{h_n}} \cdot \frac{\partial \vec{h_n}}{\partial \vec{b_n}} = \vec{U_n}$$

For the first step, matrix subtraction is $O(K)$. Then to compute $\vec{U_n}$, the matrix multiplication step is $O(K^2)$. In total, for $\vec{U_n}, \forall n \in \{1 \dots N-1\}$, we have N terms and therefore the complexity is $O(NK^2)$. To compute the gradients the dot (elementwise) multiplication will be bounded by $O(K^2)$ as that is the size of the matrix so in total the complexity for computing the gradient of the weight matrix of all layers is $O(NK^2)$. The gradient of biases is equal to $\vec{U_n}$ so no computation is required. Hence, in total, we can say that the complexity for one pass of backpropagation with vectorization and caching of intermediate results is $O(NK^2)$, or linear in the number of parameters of the deep neural network.

Case 2: Individual Gradient Computation

Notation

Following is the notation we'll use in the analysis:

${h_i}^n$ - unit i in layer n, $\forall n = 0 \dots N$, $\forall i = 1 \dots K$

${w_{ij}}^n$ - weight between unit i in layer n and unit j in layer $n+1$
$\forall n = 0 \dots N-1$ $\forall i, j = 1 \dots K$

$+{b_i}^n$ - bias term of unit i in layer n $\forall n = 1 \dots N$, $\forall i = 1 \dots K$

y_i is the target for the output of unit i, $\forall i = 1 \dots K$

Analysis

$$C = \sum_{i=1}^{K} ({h_i}^N - y_i)^2$$

$${h_j}^N = \sum_{i=1}^{K} {w_{ij}}^{n-1} {h_i}^{n-1} + {b_j}^n, \forall j \in \{1, \dots, K\}$$

$$\frac{\partial C}{\partial {h_j}^N} = 2({h_j}^N - y_i)$$

By multivariate chain rule,

$$\frac{\partial C}{\partial {h_j}^n} = \sum_{i_0=1}^{K} \sum_{i_1=1}^{K} \sum_{i_2=1}^{K} \cdots \sum_{i_{N-n}=1}^{K} \frac{\partial C}{\partial {h_{i_0}}^N} \frac{\partial {h_{i_0}}^N}{\partial {h_{i_1}}^{N-1}} \frac{\partial {h_{i_1}}^{N-1}}{\partial {h_{i_2}}^{N-2}} \cdots \frac{\partial {h_{i_{(N-n)}}}^{n+1}}{\partial {h_j}^n}$$

$$\frac{\partial C}{\partial {w_{ij}}^n} = \frac{\partial C}{\partial {h_j}^{n+1}} \frac{\partial {h_j}^{n+1}}{\partial {w_{ij}}^n} = \frac{\partial C}{\partial {h_j}^{n+1}} {h_i}^n$$

$$\frac{\partial C}{\partial {b_j}^n} = \frac{\partial C}{\partial {h_j}^n} \frac{\partial {h_j}^n}{\partial {b_j}^n} = \frac{\partial C}{\partial {h_j}^n}$$

To compute $\frac{\partial C}{\partial {h_j}^n}$ requires $O((N-n)K^{N-n})$ multiplication/addition steps due to the $N-n$ summations of K terms each, and each term is the multiplication of $N-n$ terms. Computing $\frac{\partial C}{\partial {w_{ij}}^n}$ is one multiplication step. So for one pass of backpropagation, for gradients of biases and weights in all layers and between all units ($\forall i, j = 1, \ldots, K, \forall n = 1, \ldots, N$), the complexity is $O(NK^N)$. The complexity of one backpropagation pass is exponential in the number of layers, much worse than the vectorized case with caching as the depth and width of the neural network increases because Case 1 is always linear in the number of learning parameters.

Part 8

For image processing, we used the Python file `fetch_data.py`. For network training, the file is `faces.py`. In this part, the inaccurate images from project 1 were pruned by checking SHA-256 hashes for consistency, and also removing certain images manually. The input was then preprocessed by the same procedure as project 1: using the `imread` command and then flattening it, normalizing the pixel intensities to 0 and 1.

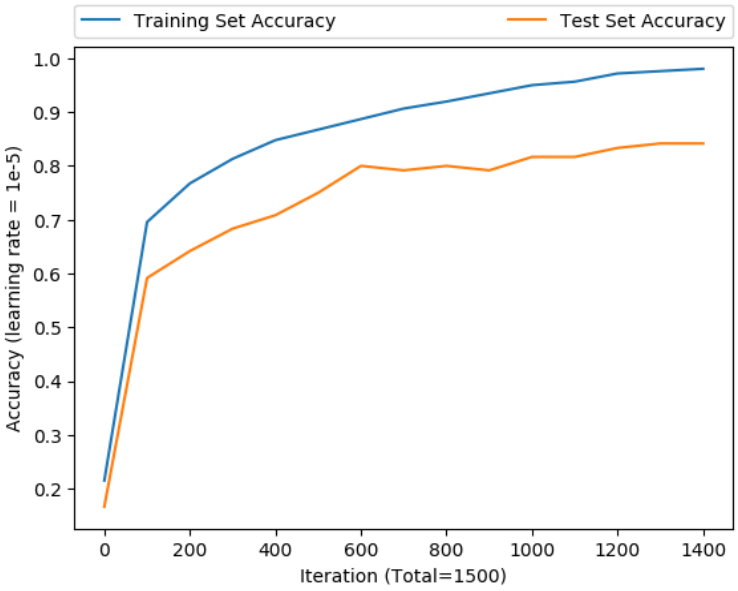
The network consists of a single-hidden layer with Relu activation function applied, and six output units that correspond to each of the six actors. The Cross entropy loss function is applied for the outputs to determine the most probable classification given an input. We varied the number of hidden units between 5 to 30 and found that 18 units produced a good accuracy result.

We found that 64 by 64 images yielded slightly higher accuracies than 32 by 32 images, so we used 64 by 64 for training the classifer. We tested using various optimizers, and we found Adam to be a good choice in terms of speed to convergence. Training with stochastic gradient descent was performed with a mini-batch size of 16. At first, we initialized the weights by sampling from a normal distribution with mean 0 and variance 1. However, we found that the Xavier normal method for initialization allowed optimization to converge even faster with better accuracy results so we used this in the final model.

At first, we trained the model with 100 epochs with a learning rate of 1e-2. The mdoel produced a very low test set accuracy. We then decreased the learning rate to 1e-4 and subsequently 1e-5, which yielded better accuracy results. Increasing the epochs also improved performance, so we decided to train for 1000 epochs in the end.

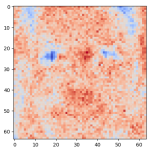
The test set included 20 images per actor, and the rest of the images in the dataset were used for training.

The below is the learning curve for the training and test set. As evident, 1500 epochs is sufficient to train the model because the validation set accuracy is stagnant at around 1400 epochs and onwards. The training set accuracy is close to 100%, whereas for test set it's around 85%.

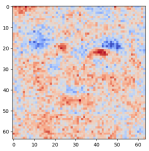

Training Set Accuracy
Test Set Accuracy
Accuracy (learning rate = 1e-5)
Iteration (Total=1500)
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
200
400
600
800
1000
1200
1400

Part 9

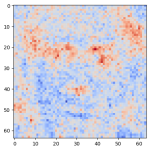
To find which hidden units were important for classifying actor k, we examined the value of $W^T x$, where W is the $H \times n$ weight matrix for the input layer and the first hidden layer (H is number of hidden units and n is the number of pixels, in our case 4096 for 64 by 64 sized images), x is the pixel intensity for one image of actor k. We considered three images of each actor and determined which two of the 18 hidden units were the most activated (highest average value). We found that for Harmon, the hidden unit 11 and 14 were the highest, and for Hader hidden units 1 and 18 were the highest.



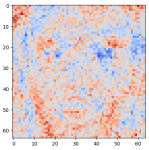
(a) Hidden unit 11 (Harmon)



(b) Hidden unit 14 (Harmon)



(c) Hidden unit 1 (Hader)



(d) Hidden unit 18 (Hader)

Part 10

For network training, the file is `deepfaces.py` As in part 8, the inaccurate images from project 1 were pruned by checking SHA-256 hashes for consistency, and also removing certain images manually. The input was then preprocessed by the same procedure as project 1: using the `imread` command and then flattening it, normalizing the pixel intensities to -1 and 1. Because AlexNet takes 227×227 coloured images, we resized the images to match this dimension requirement.

In this part we extracted the activations of the last convolutional layer of AlexNet. To extract this layer, we wrote an additional class method `getActivations(input)` that computes the forward phase up to the last layer for the AlexNet class implementation provided to us. Then, the activations can be extracted by simply calling `model.getActivations(input)`, where `model` is an instance of the AlexNet model with pre-trained weights and `input` is the pre-processed coloured image that is 227×227. We did this to obtain the activations for each image in our training and test set, and these activations were used as inputs to our neural network.

We trained a network consisting of a single-hidden layer with Relu activation function applied, and six output units that correspond to each of the six actors. The cross entropy loss function is applied for the outputs to determine the most probable classification given an input. We varied the number of hidden units between 5 to 500 and found that 32 units produced a good accuracy result.

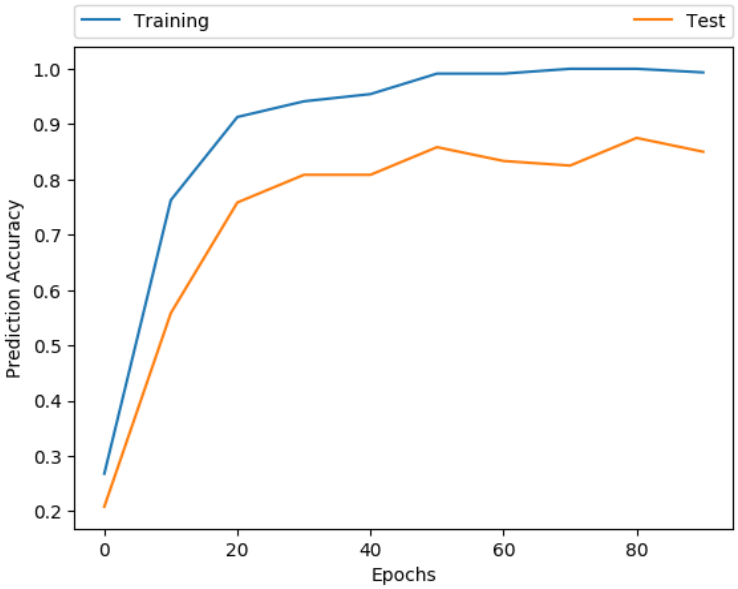
We tested using various optimizers, and we found Adam to be a good choice in terms of speed to convergence. Training with stochastic gradient descent was performed with a mini-batch size of 32. At first, we initialized the weights by sampling from a normal distribution with mean 0 and variance 1. However, we found that the Xavier normal method for initialization allowed optimization to converge even faster with better accuracy results so we used this in the final model.

To avoid overfitting and large weights, we applied dropout to the network as well as L2 regularization with a weight decay parameter of 1e-4.

At first, we trained the model with 100 epochs with a learning rate of 1e-2. The model produced a very low test set accuracy. We then decreased the learning rate to 5e-2 and subsequently 6e-3, which yielded better accuracy results.

The test set included 20 images per actor, and the rest of the images in the dataset were used for training.

The below is the learning curve for the training and test set. The training set accuracy is close to 100%, whereas for test set it's 88%. This is a slight improvement over the first network we constructed in Part 8.


Training
Test
Prediction Accuracy
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
20
40
60
80
Epochs