CSC411 Project 2

February 23, 2018

Christopher Sheedy 1002362542, Shahin Jafari 1002401634

Contents

Part 1

For this project we are working with the MNIST dataset. The dataset is split up into 10 parts, each corresponding to a digit of 0 through 9. Each part of the dataset is further split into a training and a test set. The training sets generally contain between 5500 and 6800 images, while the test sets contain between 900 and 1100 images. The digits have a fair bit of variance in terms of shape and style, but for the most part are centered in the image. It is worth noting that each image is 28x28 pixels. Some of the digits are unidentifiable even to a person, but this is a relatively small minority of digits. There is very little noise in the images, they are uniformly black except for the digit itself. Overall the dataset is very high quality. Below are 12 random samples from each digit, in which you can see the above observations:

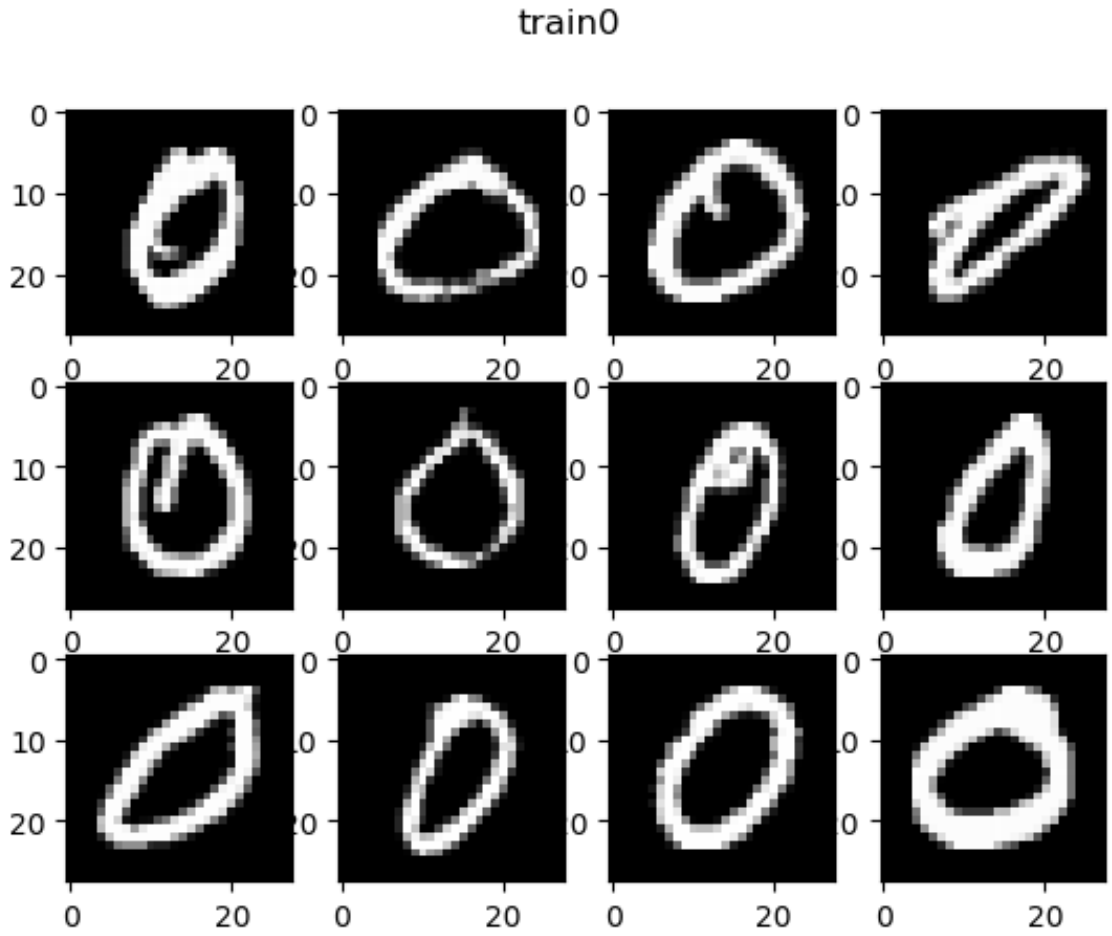


Figure 1: Images for the digit 0

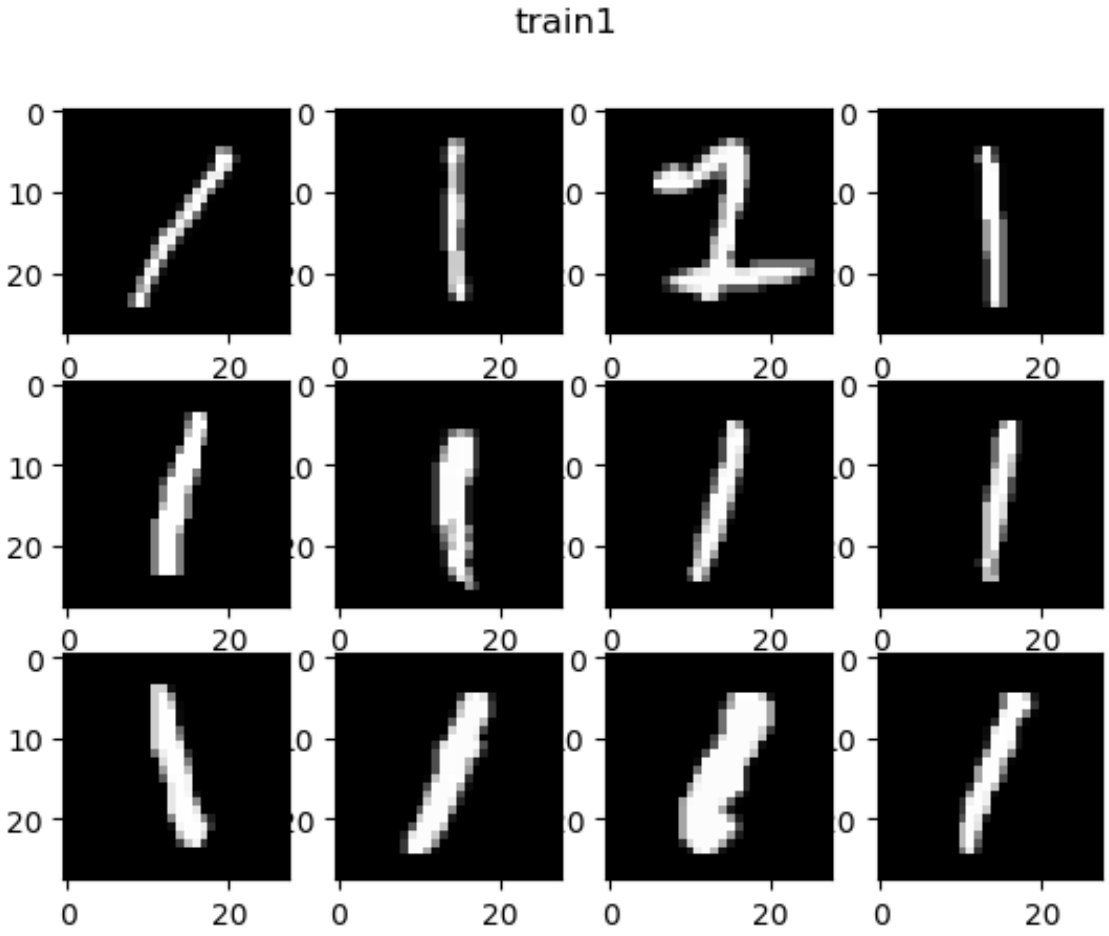


Figure 2: Images for the digit 1

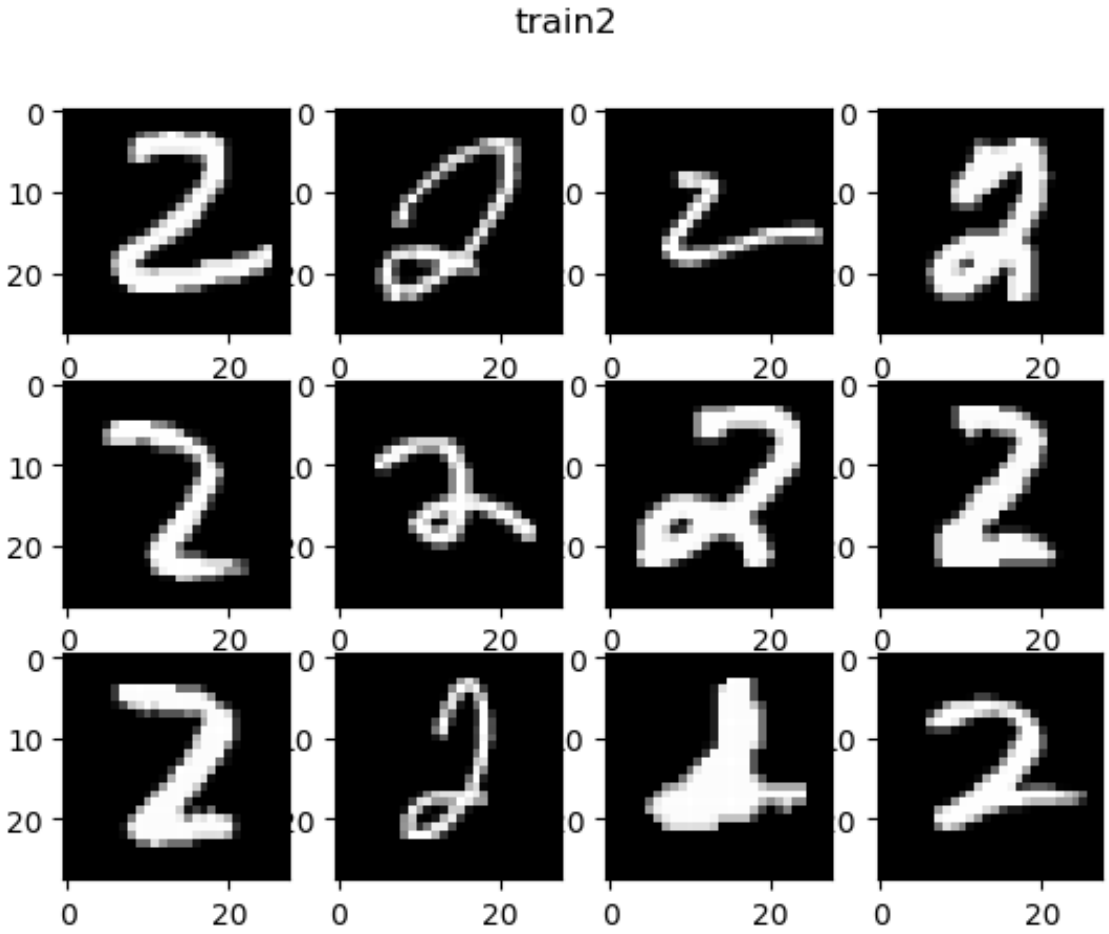


Figure 3: Images for the digit 2

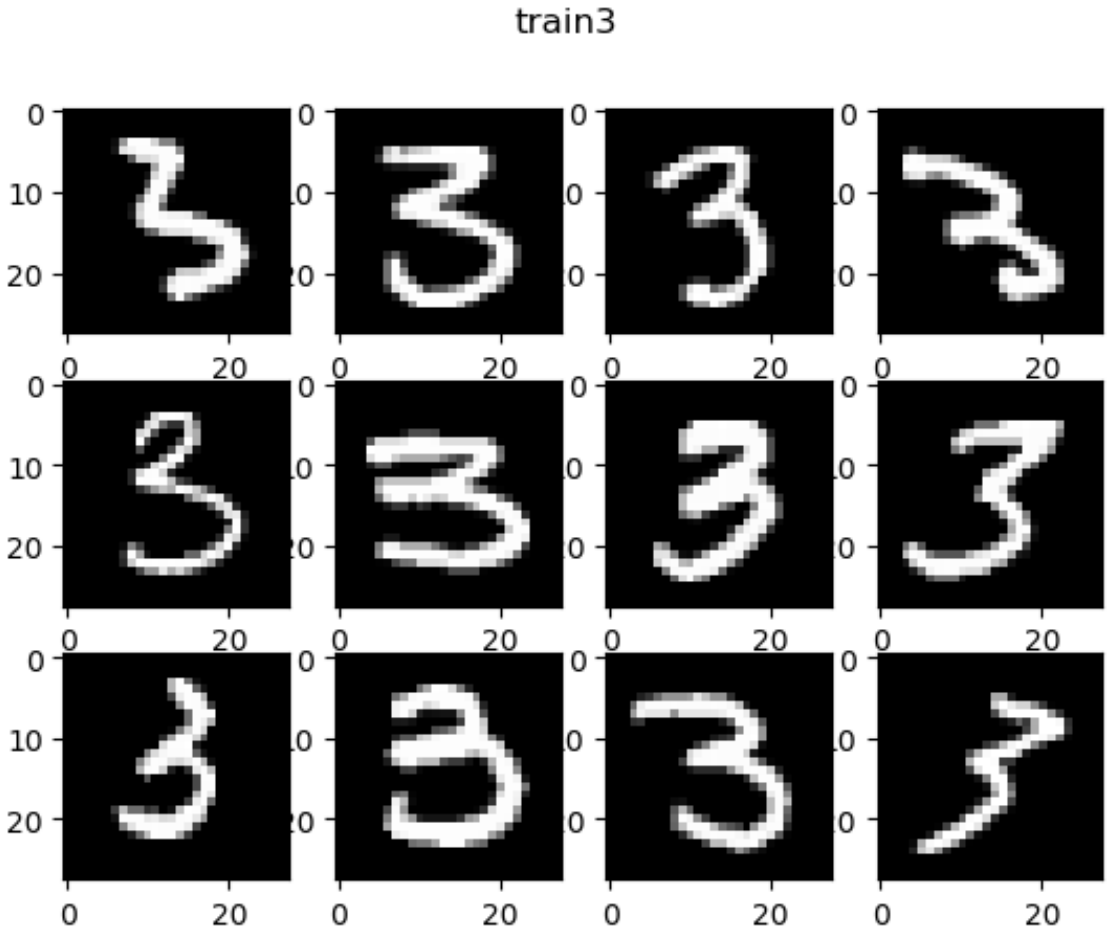


Figure 4: Images for the digit 3

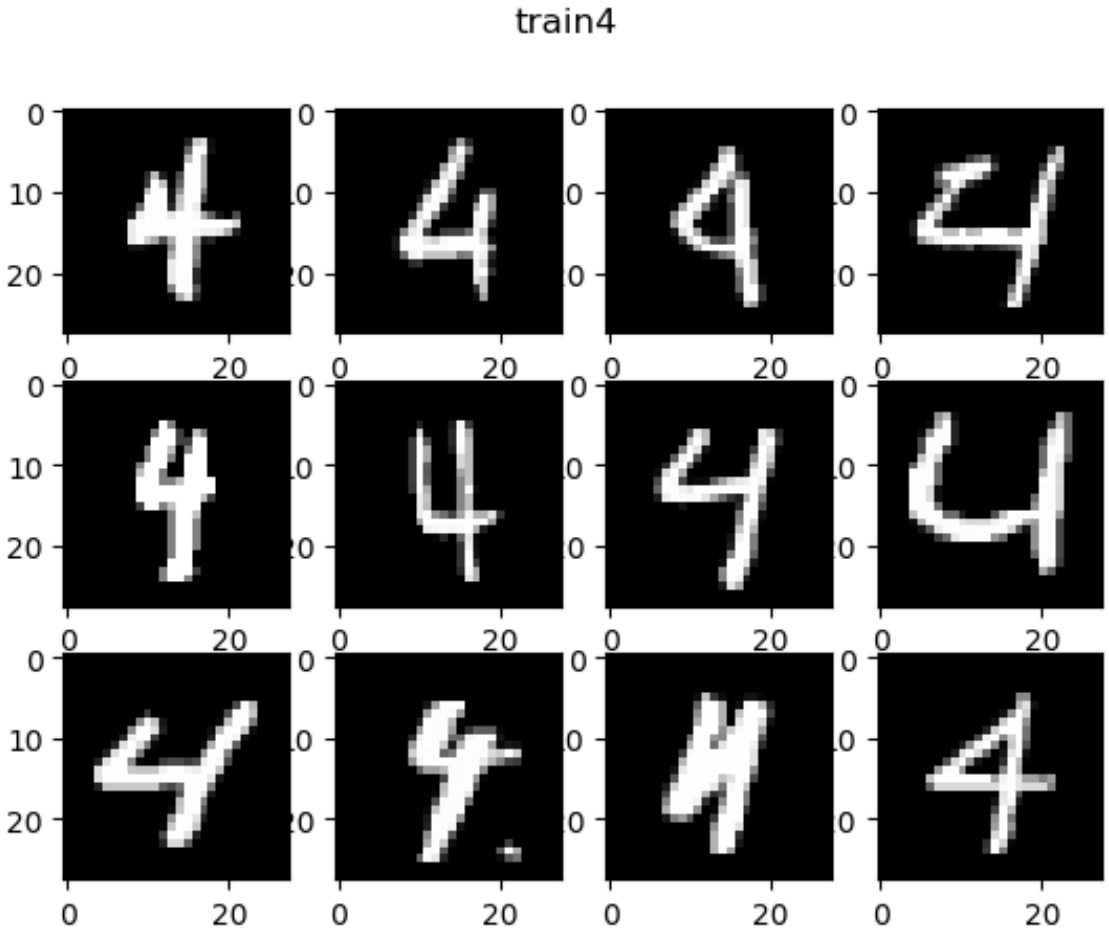


Figure 5: Images for the digit 4

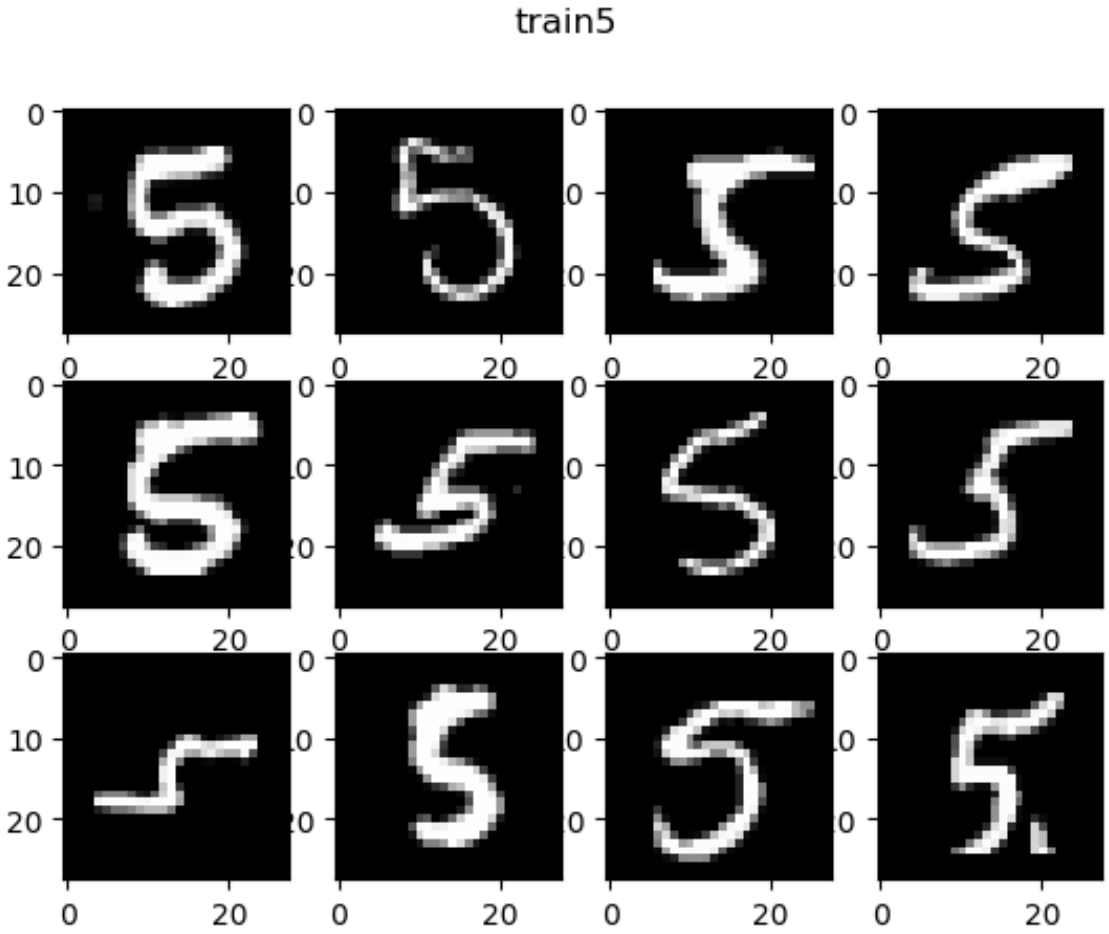


Figure 6: Images for the digit 5

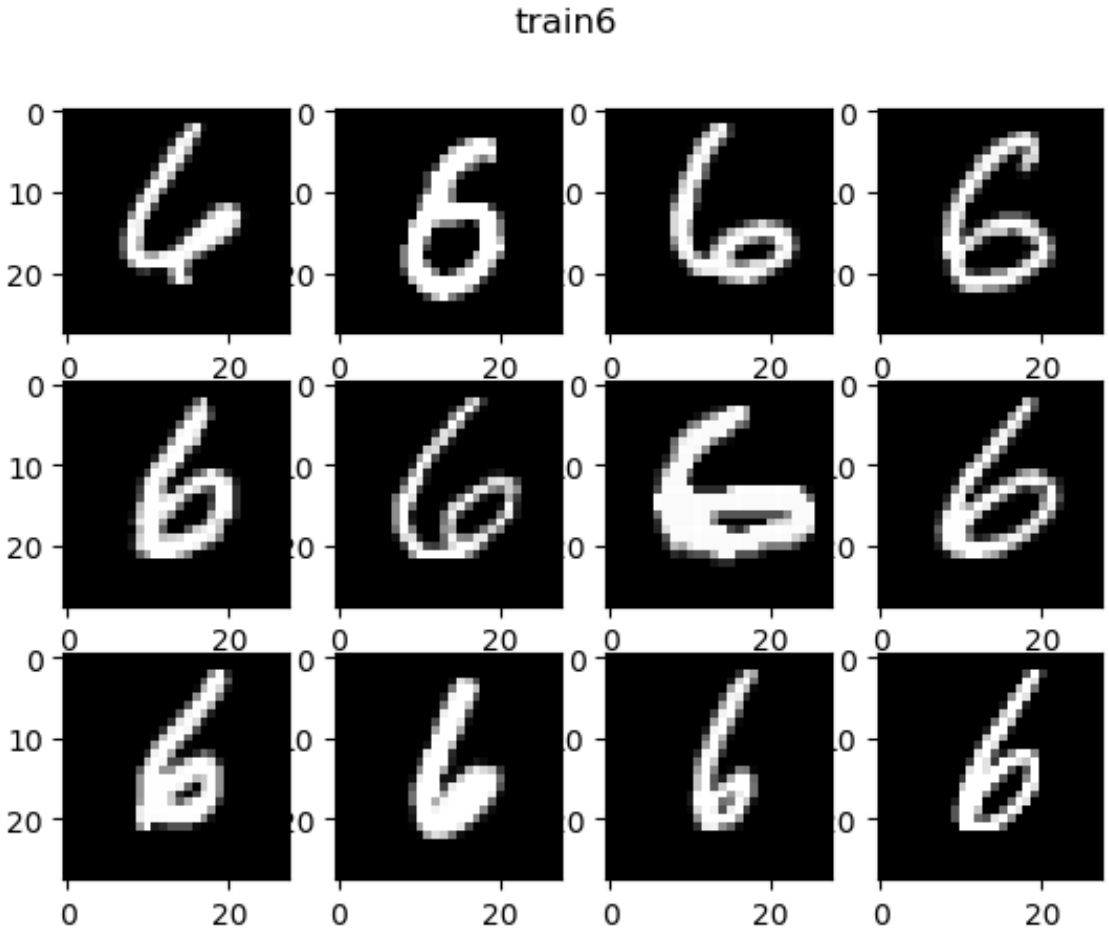


Figure 7: Images for the digit 6

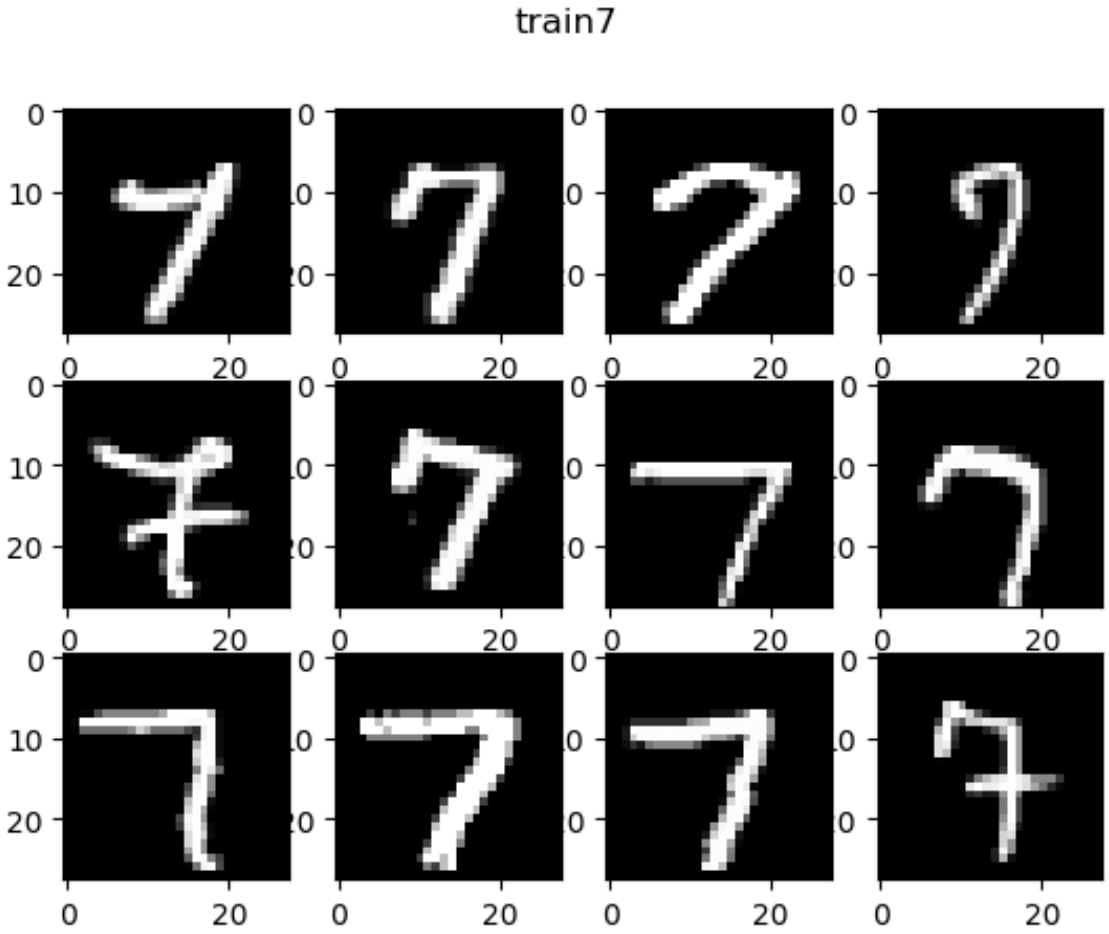


Figure 8: Images for the digit 7

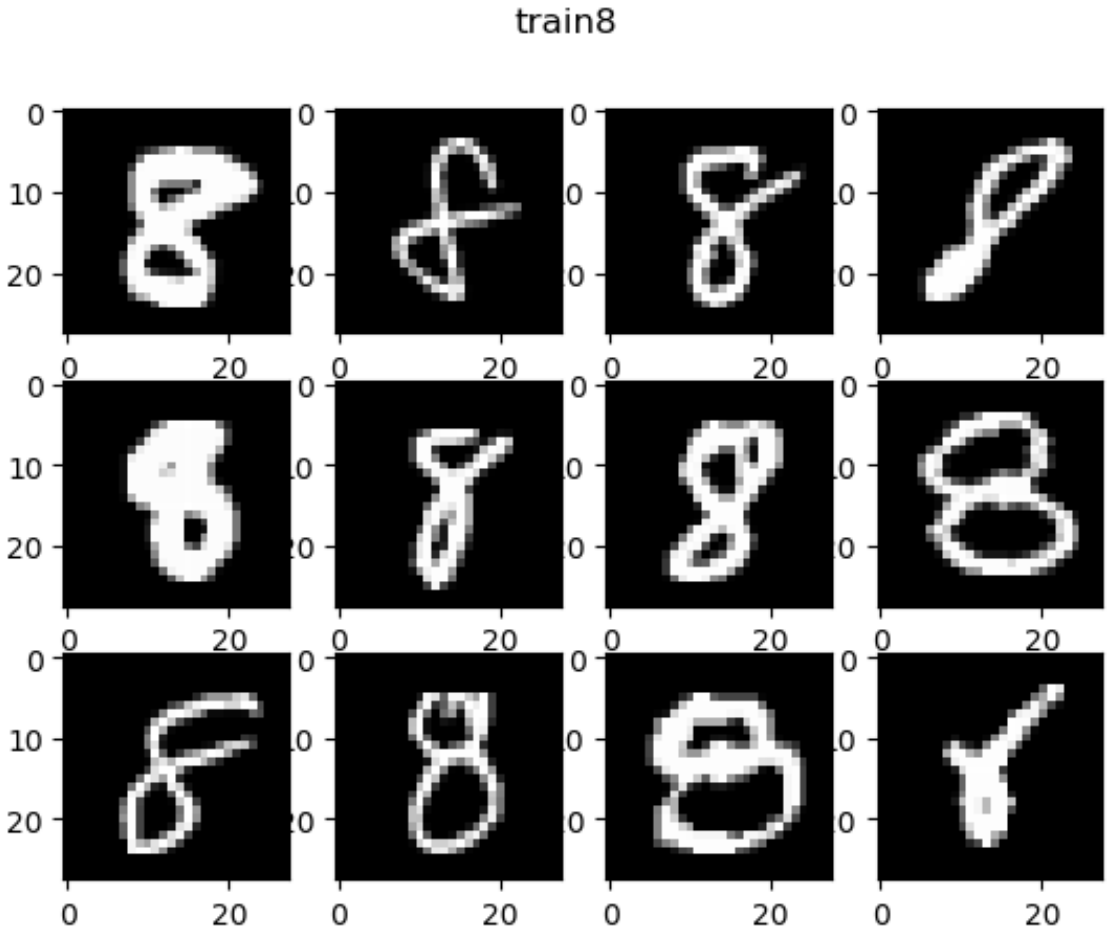


Figure 9: Images for the digit 8

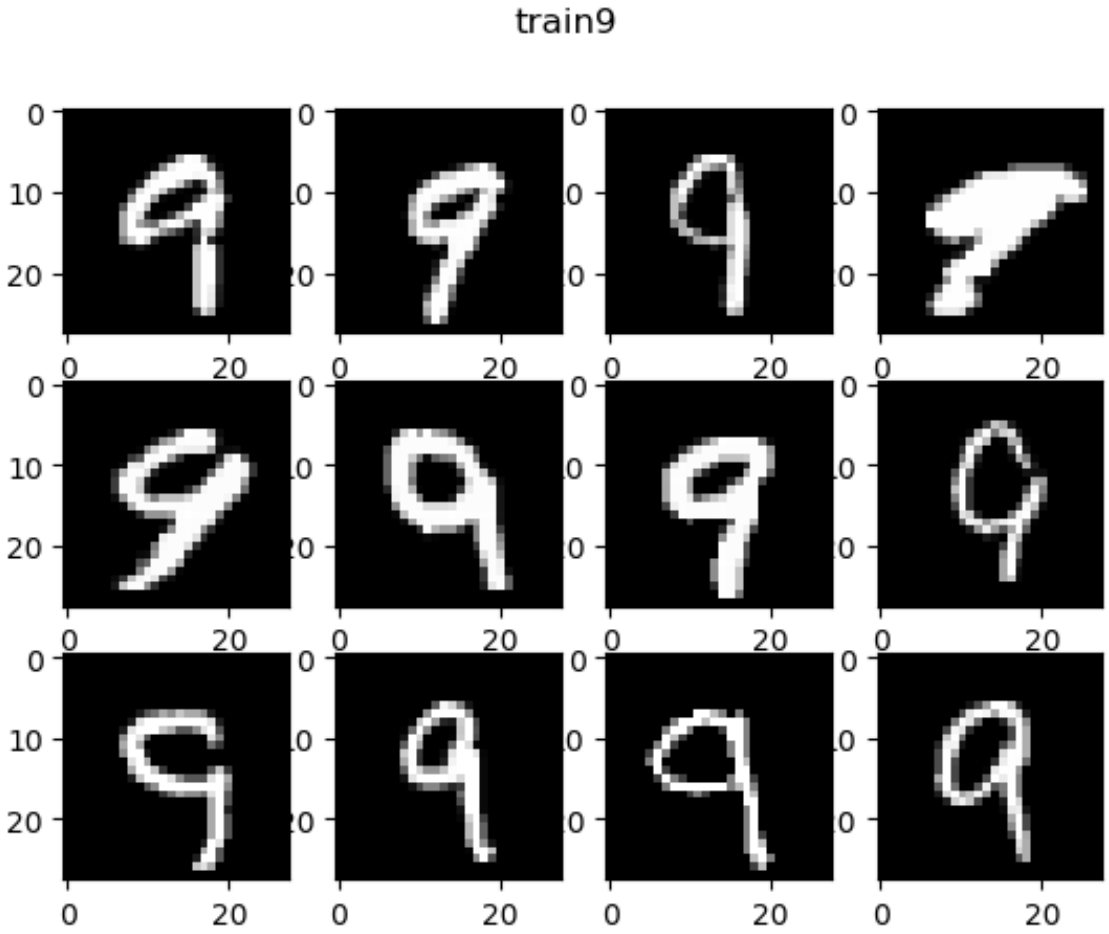


Figure 10: Images for the digit 9

Part 2

The input x, representing a training image, is a vector with dimensions 784×1. We construct a matrix w of dimensions 10×784 that contains all of the weights for the network, and a matrix of biases b of dimensions 10×1:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{784} \end{bmatrix}, w = \begin{bmatrix} w_{0,1} & w_{0,2} & \dots & w_{0,784} \\ w_{1,1} & w_{1,2} & \dots & w_{1,784} \\ \vdots & \vdots & \ddots & \vdots \\ w_{9,1} & w_{0,2} & \dots & w_{0,784} \end{bmatrix}, b = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_9 \end{bmatrix}$$

Note that $w_{i,j}$ represents the weight going from jth pixel, to the ith o, where i is one of the digits. With these matrices, we define a new matrix o whose elements are o_i:

$$o = \begin{bmatrix} o_0 \\ o_1 \\ \vdots \\ o_9 \end{bmatrix} = \begin{bmatrix} w_{0,1} & w_{0,2} & \dots & w_{0,784} \\ w_{1,1} & w_{1,2} & \dots & w_{1,784} \\ \vdots & \vdots & \ddots & \vdots \\ w_{9,1} & w_{0,2} & \dots & w_{0,784} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{784} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_9 \end{bmatrix}$$

We have encoded this as:

```
def compute_o(x,w,b):
    return np.matmul(w,x) + b

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 p2(x,w,b):
    o = compute_o(x,w,b)
    res = softmax(o)
    return res
```

Part 3(a)

Our goal is to compute the gradient, which has partial derivatives of the cost function with respect to the weight $w_{i,j}$. We will begin by using the chain rule:

$$\frac{\partial C}{\partial w_{i,j}} = \frac{\partial C}{\partial O_i}\frac{\partial O_i}{\partial w_{i,j}}$$

where

$$\frac{\partial C}{\partial O_i} = \sum_k \frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial O_i}$$

Note that p represents the output of the softmax function, or the probabilities and each change in o, will have an effect on other os, since there is a sum on the denominator of the softmax function. Since $O_i = x_j w_{i,j} + b_i$

$$\frac{\partial O_i}{\partial w_{i,j}} = x_j$$

which by substitution gives us:

$$\frac{\partial C}{\partial w_{i,j}} = \left(\sum_k \frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial O_i}\right) x_j$$

Next we want to solve for the partial derivative of cost with respect to p_k, with $C = -\log(p_k) y_k$:

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

Next, we want to examine $\frac{\partial p_k}{\partial O_i}$. In particular we note that:

$$p_k = \frac{e^{O_k}}{\sum_l e^{O_l}}$$
$$\log p_k = O_k - \log \sum_l e^{O_l}$$
$$\frac{\partial \log p_k}{\partial O_i} = \frac{1}{p_k}\frac{\partial p_k}{\partial O_i}$$

where in the last step we took the derivative of the left side. Now we take the derivative of the right side and equate the sides.
If $k = i$:

$$\frac{1}{p_k}\frac{\partial p_k}{\partial O_i} = 1 - \frac{1}{\sum_l e^{O_l}} e^{O_k} \Rightarrow \frac{\partial p_k}{\partial O_i} = p_i\left(1 - p_i\right)$$

if $k \neq i$:

$$\frac{1}{p_k}\frac{\partial p_k}{\partial O_i} = 0 - \frac{1}{\sum_l e^{O_l}} e^{O_i} \Rightarrow \frac{\partial p_k}{\partial O_i} = -p_k p_i$$

Now, we can find:

$$\begin{aligned}\sum_k \frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial O_i} &= -\frac{y_i}{p_i}\left(p_i\left(1-p_i\right)\right) + \sum_{k\neq i}\frac{y_k}{p_k}p_k p_i \\ &= -y_i + y_i p_i + \sum_{k\neq i} y_k p_i \\ &= p_i\left(\sum_k y_k\right) - y_i\end{aligned}$$

and since all elements of y are zero except for one which is 1:

$$\sum_k \frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial O_i} = p_i - y_i$$

Thus giving us our final result as:

$$\frac{\partial C}{\partial w_{i,j}} = \left(\sum_k \frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial O_i}\right) x_j = \left(p_i - y_i\right) x_j$$

now, since we have a training set with many different images, we can generalize this to:

$$\frac{\partial C}{\partial w_{i,j}} = \sum_t^{trainingset}\left(p_i^{(t)} - y_i^{(t)}\right) x_j^{(t)}$$

We also need to take the derivative of the cost function with respect to the bias vector. By inspecting the formula for calculating O_i, we can notice that the steps to take the derivative with respect to the bias vector are the same except for the very last step:

$$\begin{aligned}&\frac{\partial O_i}{\partial b_i} = 1 \\ &\Rightarrow \frac{\partial C}{\partial b_i} = \sum_t^{trainingset}\left(p_i^{(t)} - y_i^{(t)}\right)\end{aligned}$$

Part 3(b)

Our gradient should have the following form:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_{0,1}} & \frac{\partial C}{\partial w_{0,2}} & \cdots & \frac{\partial C}{\partial w_{0,784}} \\ \frac{\partial C}{\partial w_{1,1}} & \frac{\partial C}{\partial w_{1,2}} & \cdots & \frac{\partial C}{\partial w_{1,784}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial w_{9,1}} & \frac{\partial C}{\partial w_{0,2}} & \cdots & \frac{\partial C}{\partial w_{0,784}} \end{bmatrix}$$

We calculated this earlier as:

$$\frac{\partial C}{\partial w_{i,j}} = \sum_{t}^{trainingset} \left(p_i^{(t)} - y_i^{(t)}\right) x_j^{(t)}$$

Now, x is expressed as the following matrix (where n is the size of our training set):

$$x = \begin{bmatrix} x_1^{(0)} & x_1^{(1)} & \cdots & x_1^{(n)} \\ x_2^{(0)} & x_2^{(1)} & \cdots & x_2^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{784}^{(0)} & x_{784}^{(1)} & \cdots & x_{784}^{(n)} \end{bmatrix}$$

while p and y are of the following form:

$$p = \begin{bmatrix} p_0^{(0)} & p_0^{(1)} & \cdots & p_0^{(n)} \\ p_1^{(0)} & p_1^{(1)} & \cdots & p_1^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ p_9^{(0)} & p_9^{(1)} & \cdots & p_9^{(n)} \end{bmatrix}, y = \begin{bmatrix} y_0^{(0)} & y_0^{(1)} & \cdots & y_0^{(n)} \\ y_1^{(0)} & y_1^{(1)} & \cdots & y_1^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ y_9^{(0)} & y_9^{(1)} & \cdots & y_9^{(n)} \end{bmatrix}$$

Thus to find our gradient:

$$\begin{bmatrix} \frac{\partial C}{\partial w_{0,1}} & \frac{\partial C}{\partial w_{0,2}} & \cdots & \frac{\partial C}{\partial w_{0,784}} \\ \frac{\partial C}{\partial w_{1,1}} & \frac{\partial C}{\partial w_{1,2}} & \cdots & \frac{\partial C}{\partial w_{1,784}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial w_{9,1}} & \frac{\partial C}{\partial w_{0,2}} & \cdots & \frac{\partial C}{\partial w_{0,784}} \end{bmatrix} =$$

$$\left(\begin{bmatrix} p_0^{(0)} & p_0^{(1)} & \cdots & p_0^{(n)} \\ p_1^{(0)} & p_1^{(1)} & \cdots & p_1^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ p_9^{(0)} & p_9^{(1)} & \cdots & p_9^{(n)} \end{bmatrix} - \begin{bmatrix} y_0^{(0)} & y_0^{(1)} & \cdots & y_0^{(n)} \\ y_1^{(0)} & y_1^{(1)} & \cdots & y_1^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ y_9^{(0)} & y_9^{(1)} & \cdots & y_9^{(n)} \end{bmatrix} \right) \begin{bmatrix} x_1^{(0)} & x_1^{(1)} & \cdots & x_1^{(n)} \\ x_2^{(0)} & x_2^{(1)} & \cdots & x_2^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{784}^{(0)} & x_{784}^{(1)} & \cdots & x_{784}^{(n)} \end{bmatrix}^T$$

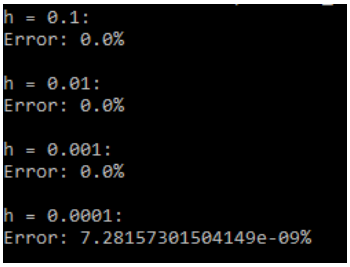
We can also express this as:

$$\nabla C = (p - y)\, x^T$$

For gradient descent, we also need to calculate the gradient of the cost function with respect to the bias, in order to update the bias after each step. This was already done in 3(a):

$$\frac{\partial C}{\partial b_i} = \sum_{t}^{trainingset} \left(p_i^{(t)} - y_i^{(t)} \right)$$

We can check the validity of our gradient by comparing it to the finite difference approximation. Here are the results with various sizes of h:



With this sanity check, we can now proceed to use the derived formulas to train our network. Below is the code for this part:

```
def gradient_w(x,y,w,b):
    '''
    input: xs and ys, weights and biases
    output: derivative w.r.t weights
    y is the label of the training set, each case is a column vector
    x is the training set, each case is a column vector
    '''
    o_s = compute_o(x,w,b)
    p = softmax(o_s)
    p_y = p - y
    grad = np.matmul(p_y,x.T)
    return grad

def gradient_b(x,y,w,b):
    '''
    input: xs and ys, weights and biases
    output: derivative w.r.t biases
    '''
    o_s = compute_o(x,w,b)
    p = softmax(o_s)
    p_y = p-y
    grad = np.sum(p_y, axis = 1)
    return grad
```

Part 4

We used vanilla gradient descent for this section. Our weights were initialized as uniform 0's. Our learning rate was 0.0001 which we found experimentally. The experiment was conducted to find the optimum learning rate and maximum iterations, while allowing the program to run in a reasonable time. In doing so, we limited ourselves to a training set of 100 images and a test set of 50 images randomly chosen from the available images. Below is the result of the experiment:

Table 1: Optimum Parameters Experiment

Learning Rate	Max Iterations	Time of Optimization(s)	Performance on Training Set(%)	Performance on Test Set(%)
0.01	100	0.40199995	100	82
0.01	1000	4.148000002	100	82.6
0.01	10000	40.47900009	100	82.8
0.01	100000	442.655	100	82
0.001	100	0.417999983	97.6	85
0.001	1000	4.157000065	100	83.4
0.001	10000	41.82799983	100	83
0.001	100000	419.323	100	83
0.0001	100	0.419000149	89.6	80.6
0.0001	1000	4.204999924	97.6	85.4
0.0001	10000	41.81200004	100	83.4
0.0001	100000	419.474	100	83
0.00001	100	0.426000118	78.9	73
0.00001	1000	4.187999964	89.6	80.6
0.00001	10000	42.26599979	97.6	85.4
0.00001	100000	419.4190001	100	83.4

From the results above, we decided to use the learning rate of 0.0001 and we set the maximum iterations to be 3000, in order to allow it to run a bit more than 1000, but not too much due to the data collection time constraint.

Below we have the graph of the learning curve:

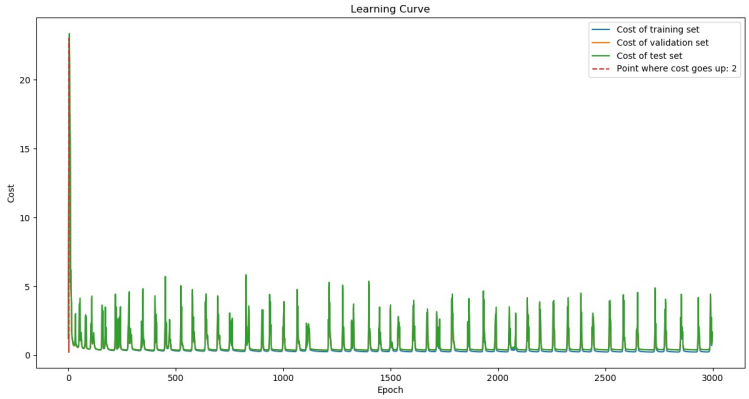
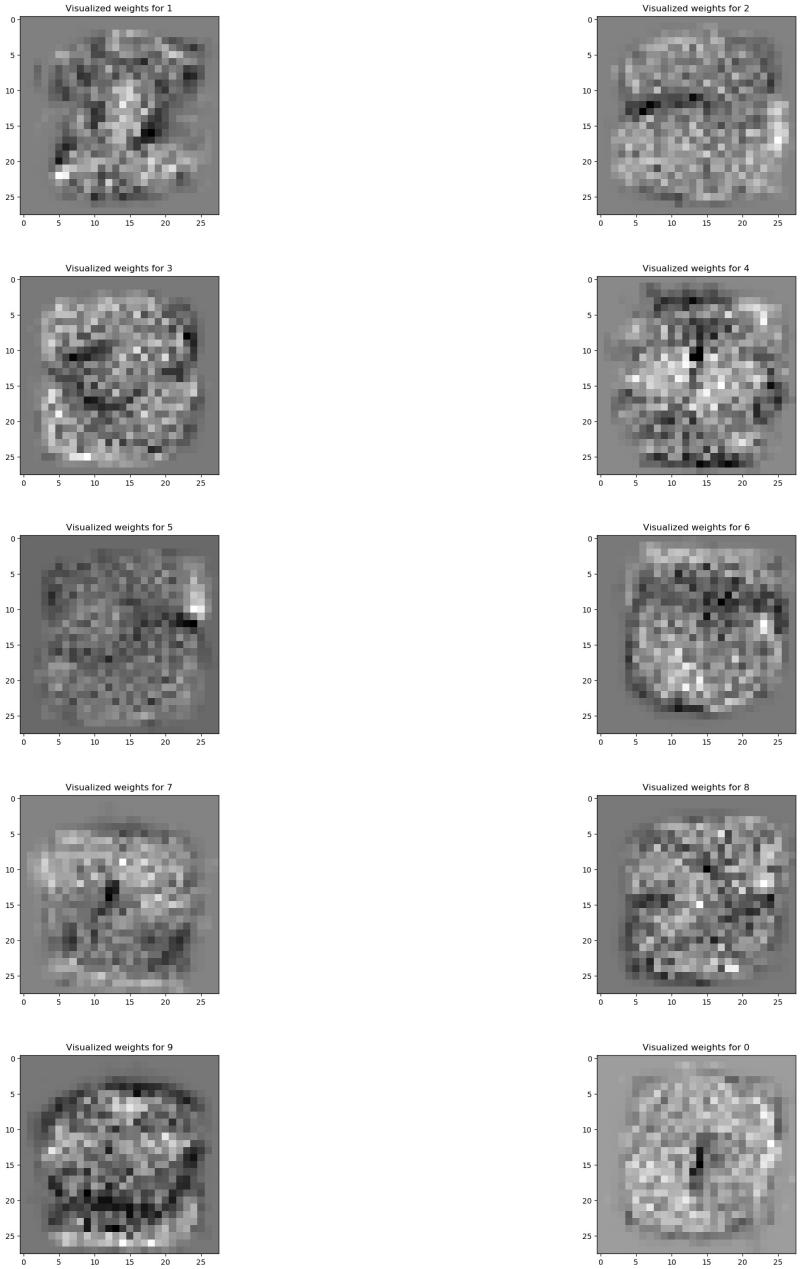


Figure 11: Vanilla Gradient Descent Learning Curve

The graph above was produced using the training set of 80 % of all available images, with the rest being the validation set.The test set was consisted of all images available. The learning rate was 0.0001, as described above and the maximum number of iterations was 3000.

We can see from the graph that as the number of iterations increases, the cost of the the training set generally decreases. There are spikes in the graph, which can be explained by the fact that minimizing the cost is not a very trivial task if the learning rate is not very small. This will cause the function to overshoot the minimum by some amount, making the cost going up and in the next epoch, it will reduce the cost.

The results of the visualizations are shown in the next page.



All of them have gray edges, which was expected, since most of the test set has gray edges. Although the visualizations are not clearly distinguishable, but they inherit some features of their class. For example, number 3 visualization shows that there are two small spaces in the middle of the image, where the pixels are likely to be zero. Another example is number 0, where a big hole is represented in the middle of the image.

Part 5

We first wrote a function to determine the optimized β value, while keeping the learning rate and the maximum iterations the same as the previous part. Again, to reduce the computation time, we used 100 images for the training set and 50 images for the test set. Below are the results of the optimization:

Table 2

β	Time of Optimization	Performance on Training Set	Performance on Test Set
0.9	12.2019999	100	83.6
0.99	13.20600009	100	83
0.999	4.457999945	10	10
0.9999	2.83100009	10	10

With the momentum term set to 0.9, we were able to reduce the time of weight optimization, but the performance decreased a bit. Now using 0.9 for β, 0.0001 for α and setting maximum iterations to 3000, we were able to run a function to display the learning curve. Below is the graph of the learning curve with the momentum term.

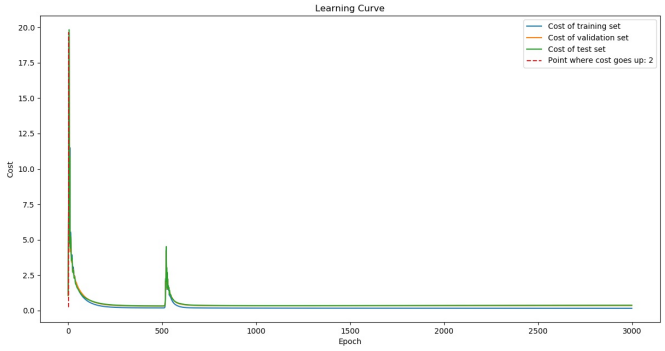


Figure 13: Learning curve with momentum

From the graph of the learning curves, we can notice that the gradient descent with momentum resulted in a faster minimization of the cost. In terms of the cost on the sets, they are very similar to each other, where the test set has a higher cost than training set, which was also apparent in the vanilla gradient descent learning curve. The spikes are also greatly reduced, and the learning curve is smoother, which is due to the fact that with momentum, we do not overshoot the optimum points as much as we do with vanilla gradient descent.

```
def grad_descent_momentum(dfw, dfb, x, y, init_w, init_b, alpha, beta):
    EPS = 1e−5 #EPS = 10**(−5)
    max_iter = 3000
    iter = 0

    prev_w = init_w−10∗EPS
    w = init_w.copy()
    sub_w = np.zeros(init_w.shape)
    v_w = 0

    prev_b = init_b−10∗EPS
    b = init_b.copy()
    sub_b = np.zeros(init_b.shape)
    v_b = 0

    while (norm(w − prev_w) > EPS or norm(b − prev_b) > EPS) and iter < max_iter:
        prev_w = w.copy()
        sub_w = dfw(x, y, w, b).reshape(sub_w.shape)
        prev_v_w = v_w
        v_w = beta ∗ prev_v_w + alpha ∗ sub_w
        w −= v_w

        prev_b = b
        sub_b = dfb(x, y, w, b).reshape(sub_b.shape)
        prev_v_b = v_b
        v_b = beta ∗ prev_v_b + alpha ∗ sub_b
        b −= v_b

        iter += 1
    return w, b
```

Part 6(a)

In this part w_1 was defined to be the weight of the pixel[13][14] and w_2 was defined to be the weight of the pixel[15][14] going to the output of the digit 5. Digit 5 was chosen since it shares lots of patterns with other numbers, such as its half circle which is common with number 8 and 9 and 6. Its vertical line is also common to number 1 and 4 and its horizontal line is common with 7. By defining w_1 and w_2 as such, we were able to get very different costs, for even a small variation in them. An exhaustive search was performed to find the cost function. In this search, w1 and w2 could range from [-10,10] with intervals of 0.01, but the other weights and biases were held constant to the initial gradient descent results. The optimum value for w_1 was 0.9017 and the optimum value for w_2 was 0.3272, as found in part 4 with the full training set. Since the computation time for this search was extremely large, the number of training images per digit was set to be 400, and a vanilla gradient descent was performed on them (with learning rate of 0.0001 and maximum iterations of 3000). The resultant w and b were stored. The contour plot of the search is shown below:

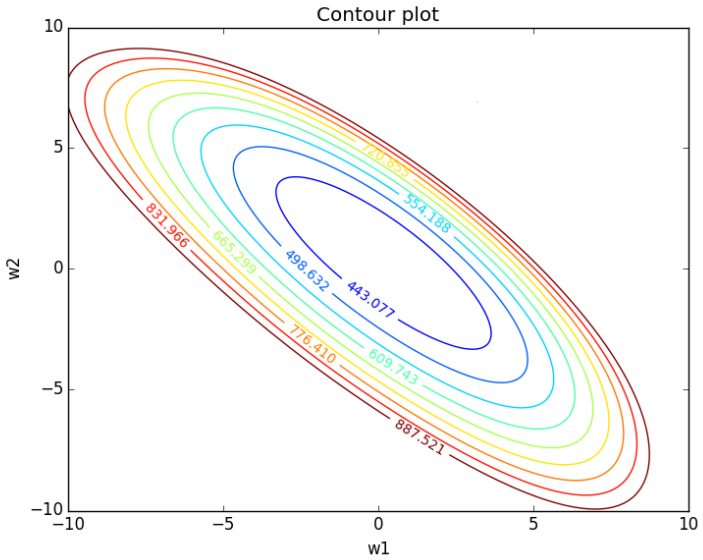


Figure 14: Contour plot of the cost function

One difference between this contour plot and the learning curve is that the costs are different. The reason for this difference is that the number of training set images were different, and as a result, the matrix w and b were different, resulting in different costs. It is also worth noting that due to the very big volume of data and the computation time for this part, the results were saved as a Numpy object, and they were used in the later parts.

Part 6(b,c,d)

Again, a gradient descent was performed initially to get the optimum ws and bs. The parameters were kept the same as the previous part, to make everything consistent and comparable. w_1 and w_2 were initialized to be -5.5 and 5 respectively, and 30 steps of vanilla and momentum gradient descent was performed with learning rate of 0.001 and 0.003 and β of 0.9. To choose the learning rate, we also tried 0.0001, but since it was very small, the trajectory was barely visible. Another version was also produced with learning rate of 0.003 and β of 0.99 for comparison. The contour plot results were imported from the previous parts, and the results are shown below:

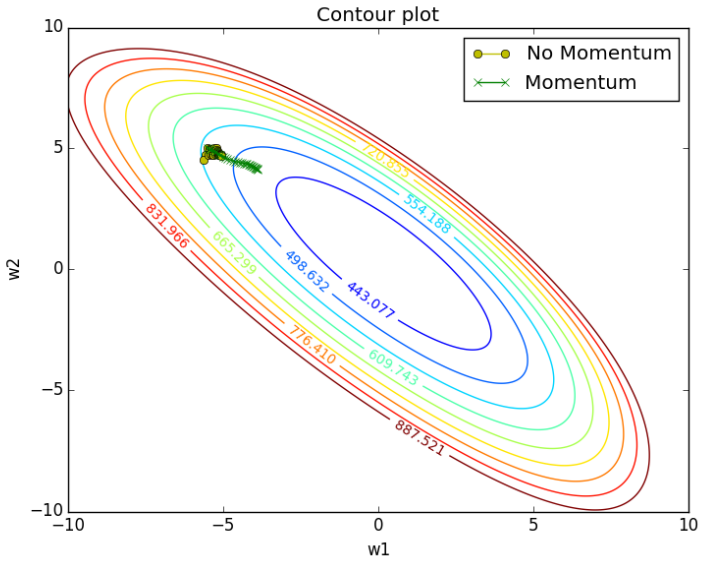


Figure 15: Trajectory of the gradient descent, $\alpha = 0.001$ and $\beta = 0.9$

From this figure, one can notice that the momentum term helped the network to converge to a smaller cost, in the same amount of steps with the same learning rate.

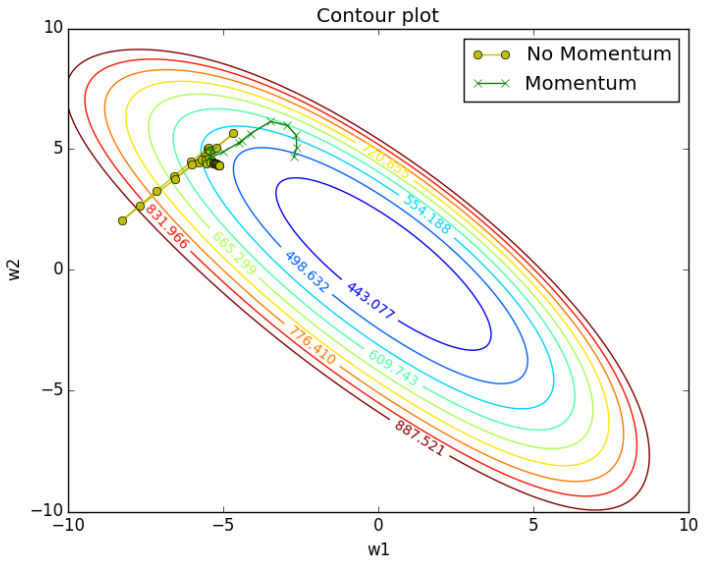


Figure 16: Trajectory of the gradient descent, $\alpha = 0.003$ and $\beta = 0.9$

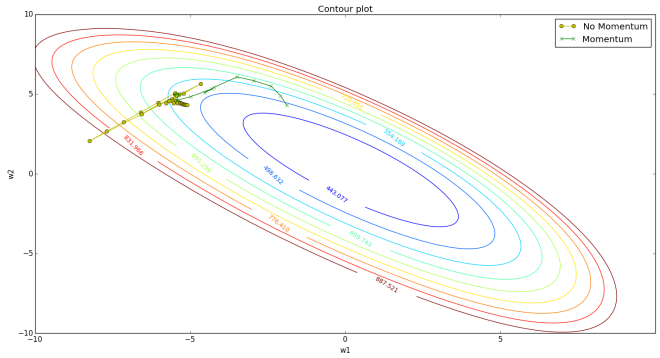


Figure 17: Trajectory of the gradient descent, $\alpha = 0.003$ and $\beta = 0.99$

These graphs also prove the fact that the momentum will make the network converge faster, but the higher the β does not necessarily improve the performance, since the distance between each step becomes larger, and again overshoot will become a possibility.

Part 6(e)

In order to produce a visualization that is neutral to having the momentum component, we set w_1 and w_2 to be pixel[0][0] and pixel[1][0], going to the digit 1. The rationale behind it was to choose pixels that affect the neurons the least. Digit 1 is usually a vertical line in the middle of the images, and the edge pixels neither affect digit 1, nor any other digits in a significant way. Unfortunately, we couldn't get a contour plot, since the cost did not change with any of the combinations of w_1 and w_2.

In the second try, we set w_1 and w_2 to be weights associated with the upper-middle part of the picture, going to the output of digit 1,pixel[3][14] and pixel[3][15].

Below are the results:

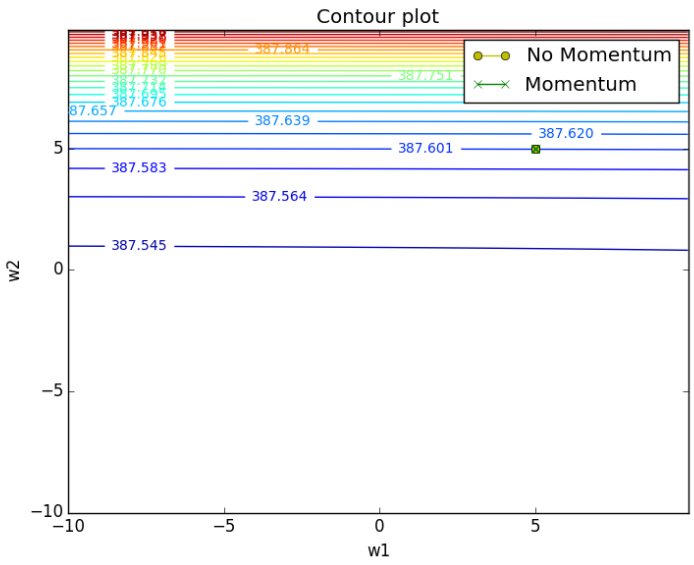


Figure 18: Trajectory of the gradient descent, $\alpha = 0.001$ and $\beta = 0.9$

As can be seen from the plot, the points are very densely located at the contour with the cost of 387.601. Clearly, momentum does not help the network at all in this case. To speculate this, we zoomed in to that region to observe the trajectory better.

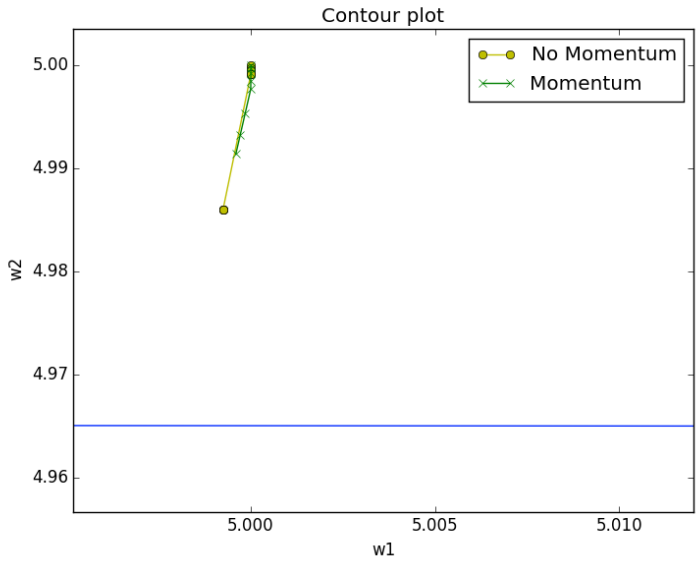


Figure 19: Trajectory of the gradient descent, $\alpha = 0.003$ and $\beta = 0.99$

As can be seen from this plot, the momentum and vanilla trajectories are almost the same, and vanilla descent has a better trajectory, in terms of minimizing the cost.

To conclude, momentum decreases the runtime of the network greatly, but it does not necessarily make the network perform better, since it does not help when the sign of the gradient changes in each iteration.

Part 7

There are two approaches to calculate the gradient with respect to each weight. Although they share the same procedure, their performances are very different. Both follow the following procedure:

1. For a given training example, propagate forward. That is, find the output of the neural network, given the training example.
2. Compare the output of the neural network with the correct answer. That is, compare each output neuron with what they should have been, if the network was perfect in classification.
3. Propagate backwards. That is, take the difference of the output neurons, and calculate the the cost function. Then take the derivative of each output neuron with respect to the previous layer, and continue this operation using the chain rule, to find the derivative with respect to a single weight.

Now if we take a naive approach, for every weight, we must propagate forward, and propagate backward using the chain rule. There are N layers and K neurons in each layer, which means each neuron in the layers except for the last one, is connected to K other neurons. Consider the first layer; the total number of connections are:

$$k \times k = k^2 \tag{1}$$

The second layer also follows the same equations for the number of connections. Now if our network was only consisted of 3 layers, we would have:

$$k \times k \times k = k^3$$

paths to consider, from the last layer to the first layer. The second layer has

$$k^2$$

paths to consider, going from the last layer to it. The last layer or the output, if it is connected to the softmax function, also adds paths to consider, which means that in order to compute the gradient with respect to every weight and bias, the total complexity of the network, as a function of complexity of the propagations(p) and derivative calculations(d) are:

$$p + d(k^3) + p + d(k^2) + p + d(k) = \sum_{i=1}^{3} p + d(k^i)$$

If we generalize the above calculation for a network with N layers and K neurons in each layer, we will obtain the following equation for the number of propagations and derivative calculations:

$$p + d(k^N) + p + d(k^{N-1}) + ... + p + d(k) = \sum_{i=1}^{N} p + d(k^i) = Np + d \times \frac{K(K^N - 1)}{K - 1} = O(N + K^N)$$

We can incorporate biases as another neuron in each layer, since we only have 1 bias for every layer. The total complexity of this approach will then become:

$$O(N + (K + 1)^N) = O(K^N)$$

Clearly, this exponential runtime is not desirable for a large network. As a result, backpropagation was introduced to reduce this exponential runtime by inheriting techniques of dynamic programming. It follows the same procedure, but it propagates forward once. The algorithm is as follows:

1. Propagate the training example once.
2. Calculate the derivate of the output with respect to all the weights in the previous layer.
3. Store the results by hashing them.
4. Since each derivative of a weight, is a multiplication of x by the derivative of the neuron that is connected to it and each derivative of a bias, is the multiplication of 1 by the derivative of the neurons that it is connect to them, we only need to multiply the stored hashes by a value.
5. Repeat this procedure to get all the derivatives.

Let us calculate the complexity of this algorithm. If the network had no hidden layers, that is, it is only consisted of the input, output and the softmax function, we only needed to propagate once, and calculate the derivative of the output and the inputs, assuming the multiplication has a cost of c:

$$p + d(K) + c(K^2) + 1(K)$$

where the first term is the cost of forward propagation; the second term is the representative of the derivative calculation of the output term with respect to the softmax function; the third term represents the derivative calculation of the cost function with respect to each weight of the input, and the last term is the derivative the bias, which is the multiplication of 1 by the derivative of the neuron that is connected to it. Note that we consider the bias as a neuron in the previous case, because in that method we needed to calculate the derivatives of the other layers every time to get to the bias, but in this case, we only need to get the derivative of other layers and using our hash, O(1), we can multiply them by 1, O(1), for all the neurons in the next layer, since the bias will affect every single one of them. The total complexity of this algorithm is:

$$p+d(K)+c(K^2)+1(K)+c(K^2)+(K)+...+c(K^2)+1(K) = p+d(K)+c\sum_{i=1}^{N-1} K^2 = p+d(K)+cK^2(N-1)$$

$$= O(NK^2)$$

As a result of this hashing, we are able to reduce the runtime complexity of the problem from exponential to polynomial time.

Part 8

In particular, describe how you preprocessed the input and initialized the weights, what activation function you used, and what the exact architecture of the network that you selected was.

The input to the network for this part was from the same dataset as in project 1. We processed the images in the same fashion. First, they were cropped to specific bounding boxes given in the dataset. Then the images were set to grayscale, and finally, reduced to either 32x32 or 64x64 resolution. Finally, they were converted into a numpy array and fed into the network. The network was a single hidden layer neural network. For a resolution of 32x32 there would be 1024 input neurons. These were fully connected to the first hidden layer with 20 neurons. These neurons used the ReLU activation function, after which they were sent to the output layer, again fully connected. The output layer had 6 neurons, which were fed into a MAX function to determine the actual class. The weights were initialized by pytorch by default.
To choose the correct parameters we iterated through a number of variations to see which gave the best results. 100 images per actor were used in the training set for all iterations. Our testing methodology was straightforward. Fix all variables except the one which we are trying to determine. Narrow down the parameter to a few likely values based on sample tests, then compare those by taking an average of several tests. The reason for this is that the sets were created randomnly, and so different numpy.random.seed() would give a different result.

32x32

The first step was to find the learning rate:

Table 3: Initial attempt for learning rate

Learning Rate	Training Set	Test Set	Runtime
0.1	20.0	16.6	41.1s
0.01	20.0	16.6	41.4s
0.001	100	84.1	38.5s
0.0001	100	83.3	36.9s
0.00001	85.0	75.8	33.6s

Based on these results we decided to investigate a learning rate of 0.001 or 0.0001. To do this we took a weighted average of several numpy.random.seed()s:

Table 4: Learning Rate averages

Learning Rate	Training Set	Test Set	Runtime
0.001	99.8	82.2	31s
0.0001	99.5	84.2	21.8s

From this experiment we decided to use a learning rate of 0.0001. Next up was to determine the batch size. With the learning rate set to 0.0001 based on our last result. we got the following result:

Table 5: Initial Attempt for Batch Size

Batch size	Training Set	Test Set	Runtime
32	100	84.2	429.84s
64	100	81.6	194.5s
128	100	80.1	72.1s
256	100	79.2	54.8s
512	100	84.1	27.9s
600	100	83.33	35.1s

Based on this result we decided to investigate 32 and 64 image batch sizes. Although 512 performed well, this is technically not a batch since the data set has only 600 images.

Table 6: Batch Size Averages

Batch size	Training Set	Test Set	Runtime
32	99.3	84	429.84s
64	96.5	82.2	194.5s

Based on this result, we chose a batch size of 32. Next was to determine the maximum number of epochs. again, there were 20 neurons in the hidden layer. This was our result:

Table 7: Initial Attempt for Max Epoch

max epoch	Training Set	Test Set	Runtime
10	41.7	32.5	0.17s
100	78	66.66	2.557s
1000	100	84.2	18.3s
10000	100	82.5	323.5s

Based on these results, we decided to use 1000 as our max epoch as it had a significantly shorter computation time than 10000. This thus gave us our final results for 32x32

Table 8: Final Results

Training Set	Test Set	Runtime
99.1	82.8	35.1s

The learning curve for the parameters shown above is in the following figure:

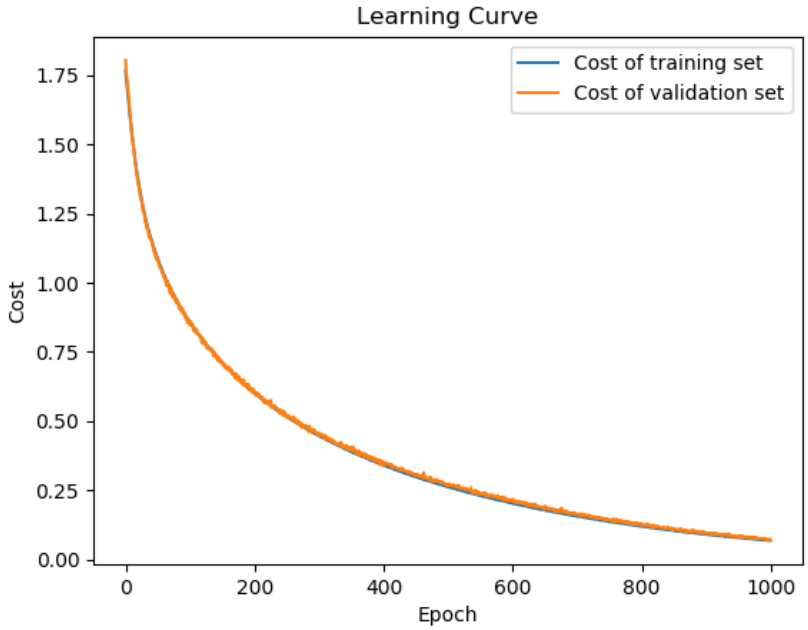


Figure 20: Learning Curve

64x64

It is possible that using 64x64 could yield slightly more accuracy, but it would be at a fairly significant trade off in computation time. Using the same parameters from the 32x32 images, we got the following result from an average of 10 experiments:

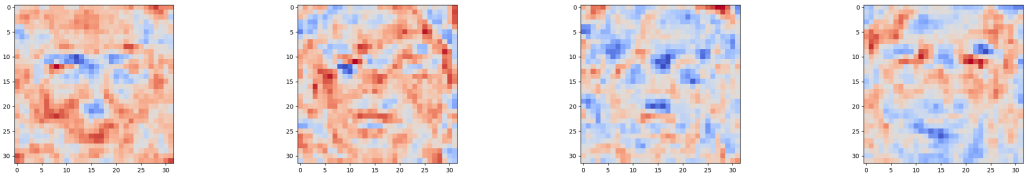
Table 9: 64x64 Averages

Training Set	Test Set	Runtime
99.9	82	90.2s

It is possible that using 64x64 could eventually yield better results, since even without any fine tuning we got the same accuracy as on the 32x32 images. However, it is likely that the improvements would only be marginal, especially given the quality of the database. Many of the images are not entirely lined up, and there is a lot of background noise and variation in the pose of the actors. Thus, given the increased computation time for the larger images we decided not to pursue this and so our final result is that given in the section for 32x32

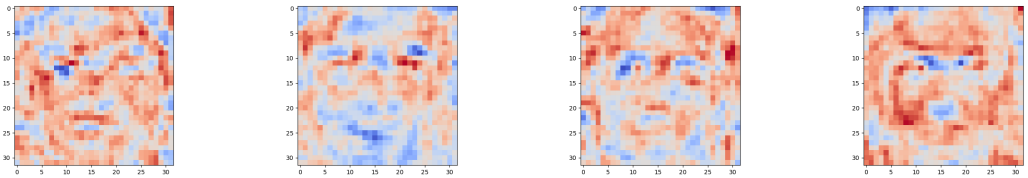
Part 9

We decided to look at the output neurons corresponding to Alec Baldwin and Lorraine Bracco. To determine which neurons to visualize, we examined the weights of the output neurons. Unfortunately, of the 20 hidden neurons, there was no single neuron that was a clear indicator of a particular actor being chosen. However, there were groups of neurons that had significantly higher weights than the others. For both Baldwin and Bracco there were 4 neurons that had weights of higher magnitude. These are displayed below:



(a) Unit 0 (b) Unit 10 (c) Unit 11 (d) Unit 6

Figure 21: Visualizations of hidden units useful for identifying Alec Baldwin



(a) Unit 10 (b) Unit 6 (c) Unit 8 (d) Unit 9

Figure 22: Visualizations of hidden units useful for identifying Lorraine Bracco

It is interesting to see that some of the important units for both Bracco and Baldwin are common. There are many possible reasons for this. For Baldwin, unit 6 has a weight of 0.706, but for Bracco unit 6 has a weight of -0.832. Perhaps for Bracco unit 6 tells the classifier that this picture is *not* Baldwin, and so the activation for that output would be lowered.

Part 10

We decided to use the same linear classifier as in part 8, and see if using the features extracted from AlexNet would improve the results without changing the structure of the classifier. The first step in this was to process the images. We scaled the images to 227x227 pixels, and limited them to 3 RGB channels. They were then processed in the same way as in the provided sample code from the course website. That is to say, each image would have a mean of 0 and have values lying between -1 and 1. This was done using the code below:

```
im = im − np.mean(im.flatten())
im = im/np.max(np.abs(im.flatten()))
im = np.expand_dims(np.transpose(im), axis=0)
```

The next step was to extract the features. To do this, we used the modified version of AlexNet provided on the course website, and simply removed the linear classifier from it. By doing so we effectively changed the forward pass to return the features computed by the convolutional part of AlexNet. The code for this modification is shown below:

```
    def __init__(self, num_classes=6):
        super(MyAlexNet, self).__init__()

        self.features = nn.Sequential(
            #Layer 1
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            #Layer 2
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            #Layer 3
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            #Layer 4
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            #Layer 5
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )

        self.load_weights()

    ## FORWARD PASS

    def forward(self, x):
        x = self.features(x)
```

```
#This part flattens the output from the convolutional layers to input for fully connected
    ↪ layer
x = x.view(x.size(0), 256 * 6 * 6)
return x
```

As mentioned earlier, the linear classifier was similar to the one used in part 8, albeit with 9216 input units rather than 1024. However, we decided to retune all the parameters to see if that would improve performance. We used the same method of tuning as in part 8:

Table 10: Initial attempt for learning rate

Learning Rate	Training Set	Test Set	Runtime
0.1	20.0	16.6	198.1s
0.01	1	83.3	172.5s
0.001	100	90	173.7s
0.0001	100	95	130.7s
0.00001	100	94.2	111.4s

Based on these results we decided to investigate a learning rate of 0.0001 or 0.00001. To do this we took a weighted average of several numpy.random.seed()s:

Table 11: Learning Rate averages

Learning Rate	Training Set	Test Set	Runtime
0.0001	100	94.67	120.7s
0.00001	100	94.41	115.1s

From this experiment we decided to use a learning rate of 0.0001. Next up was to determine the batch size. With the learning rate set to 0.0001 based on our last result. we got the following result:

finding batch size:

Table 12: Initial Attempt for Batch Size

Batch size	Training Set	Test Set	Runtime
32	100	94.1	1273.1s
64	100	94.2	307.3s
128	100	94.2	244.7s
256	100	94.2	141.6s
512	100	94.2	94.6s
600	100	94.2	103.2s

Interestingly, there were little to no differences in performance between the different batch sizes. We still want to continue doing batch gradient descent, so we decided not to investigate batch size of 512, or 600. We decided to focus on 64 and 128 since they took a lot less time than 32.

Table 13: Batch Size Averages

Batch size	Training Set	Test Set	Runtime
64	100	94.9	441.2s
128	100	94.9	249.2s

The next step was to again, determine epoch size:
1000: (1.0, 0.95, 23.14035201072693), 10000: (1.0, 0.9583333333333334, 263.3401291370392),

Table 14: Initial Attempt for Max Epoch

max epoch	Training Set	Test Set	Runtime
10	96.9	91.7	0.37s
100	100	94.2	1.9s
1000	100	95	23.1s
10000	100	95.8	263.4s

The long run time for 10000 made us decide to stick with 1000 for max epoch. Thus, our final result is:

Table 15: Final Result

Training Set	Test Set	Runtime
100	94.3	25.7s

We also found the learning curve for the classifier:

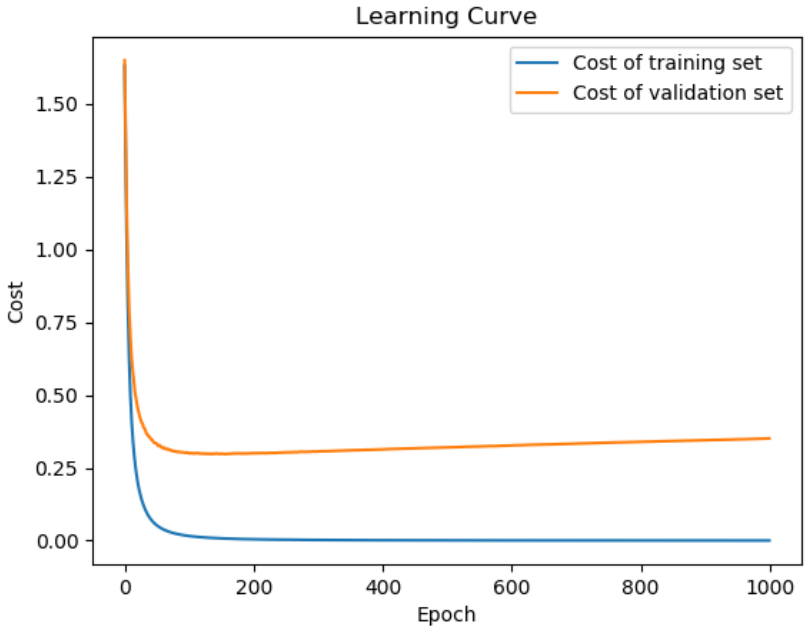


Figure 23: Learning Curve

Code

digits.py

This section includes all code for classifying digits

faces.py

This section includes all code used for classifying faces using a linear classifier using pytorch

deepfaces.py

This section includes all code used for classifying faces based on features extracted from AlexNet

Appendix

digits.py

```
from pylab import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import time
import matplotlib.image as mpimg
from scipy.ndimage import filters
import urllib
from numpy import random
import time

import pickle as cPickle

import os
from scipy.io import loadmat

PIC_SIZE = 28
LBL_SIZE = 10

def NLL(y, y_):
    '''
    y is p, what the network is giving us
    y_ is the actual y, what we want the network to give us
    '''
    return -sum(y_*log(y))

#Part 1
def get_all_training_valid__test_set():
    '''
    input : nothing
    output: the training set with 80% of available training images, validation set with 20% of available training
        ↪ images, test set with all of the available test images
    some images are excluded (10) to make sure there are no overlaps
    '''

    train_label = ["train0","train1","train2","train3","train4","train5","train6","train7","train8","train9"]
    test_label = ["test0","test1","test2","test3","test4","test5","test6","test7","test8","test9"]
    i = 0
    train_set = np.array([[]])
    train_set_label = np.array([[]])
    valid_set = np.array([[]])
    valid_set_label = np.array([[]])
```

```
for digit in train_label:
    up_t = int(round(0.8*M[digit].shape[0])) - 10
    low_v = up_t + 10
    size_v = M[digit].shape[0]-low_v
    x = np.zeros([PIC_SIZE*PIC_SIZE,up_t])
    labels_t = np.zeros([LBL_SIZE,up_t])
    v = np.zeros([PIC_SIZE*PIC_SIZE,size_v])
    labels_v = np.zeros([LBL_SIZE,size_v])

    np.random.seed(0)
    indices_t = np.random.random_integers(up_t, size = x.shape[1])
    np.random.seed(0)
    indices_v = np.random.random_integers(low_v,M[digit].shape[0]-1, size = v.shape[1])
    j = 0
    for index in indices_t:
        x[:,j:j+1] = M[digit][index].T.reshape((PIC_SIZE*PIC_SIZE,1)) / 255.0
        j += 1
    j = 0
    for index in indices_v:
        v[:,j:j+1] = M[digit][index].T.reshape((PIC_SIZE*PIC_SIZE,1)) / 255.0
        j += 1
    labels_t[i,:] = 1
    labels_v[i,:] = 1
    i += 1
    if digit == 'train0':
        train_set = x
        train_set_label = labels_t
        valid_set = v
        valid_set_label = labels_v
    else:
        train_set = np.hstack((train_set, x))
        train_set_label = np.hstack((train_set_label, labels_t))
        valid_set = np.hstack((valid_set, v))
        valid_set_label = np.hstack((valid_set_label, labels_v))

i = 0
test_set = np.array([[]])
test_set_label = np.array([[]])
for digit in test_label:
    up = M[digit].shape[0]
    np.random.seed(0)
    test = np.zeros([PIC_SIZE*PIC_SIZE,up])
    labels_test = np.zeros([LBL_SIZE,up])
    indices_test = np.random.random_integers(up-1, size = test.shape[1])
    j = 0
    for index in indices_test:
```

```
            test[:,j:j+1] = M[digit][index].T.reshape((PIC_SIZE*PIC_SIZE,1)) / 255.0
            j += 1
        labels_test[i,:] = 1
        i += 1
        if digit == 'test0':
            test_set = test
            test_set_label = labels_test
        else:
            test_set = np.hstack((test_set, test))
            test_set_label = np.hstack((test_set_label, labels_test))


    return train_set,train_set_label,valid_set, valid_set_label, test_set,test_set_label


def get_training_set(train_set_size):
    '''
    input: desired size of the training set
    output: the training set
    '''
    np.random.seed(0)
    indices = np.random.random_integers(1000, size = (train_set_size))
    train_label = ["train0","train1","train2","train3","train4","train5","train6","train7","train8","train9"]
    x = np.zeros([PIC_SIZE*PIC_SIZE,train_set_size*len(train_label)])
    labels = np.zeros([LBL_SIZE,train_set_size*len(train_label)])
    i = 0
    j = 0
    for digit in train_label:
        for index in indices:
            x[:,j:j+1] = M[digit][index].T.reshape((PIC_SIZE*PIC_SIZE,1)) / 255.0
            j += 1
        labels[i,train_set_size*(i):train_set_size*(i+1)] = 1
        i += 1
    return x, labels

def get_valid_set(valid_set_size):
    '''
    input: desired size of the validation set
    output: the validation set
    '''
    np.random.seed(0)
    indices = np.random.random_integers(1001,2000, size = (valid_set_size))
    valid_label = ["train0","train1","train2","train3","train4","train5","train6","train7","train8","train9"]
    x = np.zeros([PIC_SIZE*PIC_SIZE,valid_set_size*len(valid_label)])
    labels = np.zeros([LBL_SIZE,valid_set_size*len(valid_label)])
    i = 0
```

```
    j = 0
    for digit in valid_label:
        for index in indices:
            x[:,j:j+1] = M[digit][index].T.reshape((PIC_SIZE*PIC_SIZE,1)) / 255.0
            j += 1
        labels[i,valid_set_size*(i):valid_set_size*(i+1)] = 1
        i += 1
    return x, labels

def get_figures_1():
    '''
    input : none
    output: 12 random images of each digit
    '''
    np.random.seed(0)
    indices = np.random.random_integers(5000, size = (12))
    train_label = ["train0","train1","train2","train3","train4","train5","train6","train7","train8","train9"]
    for digit in train_label:
        plt.figure()

        for i in range(0,12):
            plt.subplot(3,4, i+1)
            plt.imshow(M[digit][indices[i]].reshape((28,28)), cmap=cm.gray)

        plt.suptitle(digit)
        plt.savefig(digit, bbox_inches='tight')
        plt.show()

def get_test_set(test_set_size):
    '''
    input: desired size of the test set
    output: the test set
    '''
    np.random.seed(0)
    indices = np.random.random_integers( size = (test_set_size))
    test_label = ["test0","test1","test2","test3","test4","test5","test6","test7","test8","test9"]
    x = np.zeros([PIC_SIZE*PIC_SIZE,test_set_size*len(test_label)])
    labels = np.zeros([LBL_SIZE,test_set_size*len(test_label)])
    i = 0
    j = 0
    for digit in test_label:
        for index in indices:
            x[:,j:j+1] = M[digit][index].T.reshape((PIC_SIZE*PIC_SIZE,1)) / 255.0
            j += 1
        labels[i,test_set_size*(i):test_set_size*(i+1)] = 1
        i += 1
```

```
    return x, labels

#Part 2
def compute_o(x,w,b):
    '''
    input: xs and weights and biases
    output: the output of the network
    '''
    return np.matmul(w,x) + b

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 p2(x,w,b):
    '''
    input: xs weights and biases
    output: output of the network, with softmax
    '''
    o = compute_o(x,w,b)
    res = softmax(o)
    return res

#Part 3
def gradient_b(x,y,w,b):
    '''
    input: xs and ys, weights and biases
    output: derivative w.r.t biases
    '''
    o_s = compute_o(x,w,b)
    p = softmax(o_s)
    p_y = p-y
    grad = np.sum(p_y, axis = 1)
    return grad

def gradient_w(x,y,w,b):
    '''
    input: xs and ys, weights and biases
    output: derivative w.r.t weights
    y is the label of the training set, each case is a column vector
    x is the training set, each case is a column vector
    '''
    o_s = compute_o(x,w,b)
    p = softmax(o_s)
```

```
    p_y = p − y
    grad = np.matmul(p_y,x.T)
    return grad

def finite_dif(func,xs,ys,w,h):
    '''
    finite difference computation of the gradient
    input: function and h
    output: gradient
    '''
    gradient = np.zeros((thetas.shape))
    for i in range(w.shape[0]):
        new_w = w.copy()
        new_w[i,:] += h
        u1 = func(xs,ys,new_w)
        u2 = func(xs,ys,w)
        derivative = (func(xs,ys,new_w) − func(xs,ys,w)) / float(h)
        gradient[i,:] = derivative
    return gradient

def max_dif(t2, t1):
    return "Error: " + str(np.amax(t2−t1)∗100.0) + "%"

def finite_diff_test():
    xs, ys = get_test_set(100)
    init_w = np.zeros((LBL_SIZE,PIC_SIZE∗PIC_SIZE))
    init_b = np.zeros((LBL_SIZE,1))
    learning_rate = 0.0001
    ws, bs = grad_descent(gradient_w, gradient_b, xs, ys,init_w, init_b, learning_rate)
    h0 = 0.1
    h1 = 0.01
    h2 = 0.001
    h3 = 0.0001
    gradient = gradient_w(xs,ys,ws,bs)
    finite_diff0 = finite_dif_w(cost_function,xs,ys,ws,bs,h0)
    finite_diff1 = finite_dif_w(cost_function,xs,ys,ws,bs,h1)
    finite_diff2 = finite_dif_w(cost_function,xs,ys,ws,bs,h2)
    finite_diff3 = finite_dif_w(cost_function,xs,ys,ws,bs,h3)
    print "h = 0.1:"
    print max_dif(gradient, finite_diff0)
    print ''
    print "h = 0.01:"
    print max_dif(gradient, finite_diff1)
    print ''
    print "h = 0.001:"
    print max_dif(gradient, finite_diff2)
```

```
    print ''
    print "h = 0.0001:"
    print max_dif(gradient, finite_diff3)
    print ''

#Part 4
def grad_descent(dfw, dfb, x, y, init_w, init_b, alpha):
    '''
    input: derivative functions, xs and ys, initial weights and biases, learning rate
    output: optimized weights and biases, using gradient descent with momentum
    '''
    EPS = 1e-5 #EPS = 10**(-5)
    prev_w = init_w-10*EPS
    w = init_w.copy()
    prev_b = init_b-10*EPS
    b = init_b.copy()
    max_iter = 3000
    iter = 0
    sub_w = np.zeros(init_w.shape)
    sub_b = np.zeros(init_b.shape)

    while (norm(w - prev_w) > EPS or norm(b - prev_b) > EPS) and iter < max_iter:
        prev_w = w.copy()
        prev_b = b.copy()
        sub_w = alpha*dfw(x, y, w, b).reshape(sub_w.shape)
        sub_b = alpha*dfb(x, y, w, b).reshape(sub_b.shape)
        w -= sub_w
        b -= sub_b
        iter += 1
    return w, b #w is the fitted weights

def grad_descent_param(dfw, dfb, x, y, init_w, init_b, alpha, iter):
    '''
    input: derivative functions, xs and ys, initial weights and biases, learning rate and max iter
    output: optimized weights and biases, using vanilla descent
    this is the same function as grad_descent, except it takes one more argument, maximum iterations
    '''
    EPS = 1e-5 #EPS = 10**(-5)
    prev_w = init_w-10*EPS
    w = init_w.copy()
    prev_b = init_b-10*EPS
    b = init_b.copy()
    max_iter = iter
    iter = 0
    sub_w = np.zeros(init_w.shape)
    sub_b = np.zeros(init_b.shape)
```

```
    while (norm(w − prev_w) > EPS or norm(b − prev_b) > EPS) and iter < max_iter:
        prev_w = w.copy()
        prev_b = b.copy()
        sub_w = alpha∗dfw(x, y, w, b).reshape(sub_w.shape)
        sub_b = alpha∗dfb(x, y, w, b).reshape(sub_b.shape)
        w −= sub_w
        b −= sub_b
        iter += 1
    return w, b #w is the fitted weights

def train(training_set, label_set):
    '''
    input: training set
    output: the matrices of w and b for network
    '''
    init_w = np.zeros((LBL_SIZE,PIC_SIZE∗PIC_SIZE))
    init_b = np.zeros((LBL_SIZE,1))
    learning_rate = 0.0001
    x = training_set
    y = label_set
    weights, bias = grad_descent(gradient_w, gradient_b, x, y,init_w, init_b, learning_rate)
    return weights, bias

def train_momentum(training_set, label_set, beta):
    '''
    input: training set
    output: the matrices of w and b for network
    '''
    init_w = np.zeros((LBL_SIZE,PIC_SIZE∗PIC_SIZE))
    init_b = np.zeros((LBL_SIZE,1))
    learning_rate = 0.0001
    x = training_set
    y = label_set
    weights, bias = grad_descent_momentum(gradient_w, gradient_b, x, y,init_w, init_b, learning_rate, beta)
    return weights, bias



def train_param(training_set, label_set, alpha, iterations):
    '''
    input: training set
    output: the matrices of w and b for network
    this is the same function as train, except this will take alpha annd iterations as well
    '''
    init_w = np.zeros((LBL_SIZE,PIC_SIZE∗PIC_SIZE))
```

```
    init_b = np.zeros((LBL_SIZE,1))
    learning_rate = alpha
    x = training_set
    y = label_set
    weights, bias = grad_descent_param(gradient_w, gradient_b, x, y,init_w, init_b, learning_rate,iterations)
    return weights, bias

def test_performance(x, label, w, b):
    '''
    input: a set and its labels, the weight matrix and bias matrix
    output: percentage correct classified
    '''
    correct = 0
    total = 0
    for i in range(x.shape[1]):
        o_test = np.matmul(w,x[:,i:i+1]) + b
        y_test = label[:,i]
        if (argmax(o_test) == argmax(y_test)):
            correct += 1
        total += 1

    return correct/float(total)*100

def plot_learning_curve(momentum='off'):
    '''
    input: (optional) learning curve of gradient descent with momentum
    output: the learning curve of gradient descent, using all the images
    '''

    train_set,train_set_label,valid_set, valid_set_label, test_set,test_set_label = get_all_training_valid__test_set()

    init_w = np.zeros((LBL_SIZE,PIC_SIZE*PIC_SIZE))
    init_b = np.zeros((LBL_SIZE,1))
    alpha = 0.0001
    beta = 0.9
    x = train_set
    y = train_set_label

    EPS = 1e-5 #EPS = 10**(-5)
    prev_w = init_w-10*EPS
    w = init_w.copy()
    prev_b = init_b-10*EPS
    b = init_b.copy()
    max_iter = 3000
    iter = 0
    sub_w = np.zeros(init_w.shape)
```

```
sub_b = np.zeros(init_b.shape)
v_w = 0
v_b = 0
performance = np.zeros((max_iter,4))
i = 0
prev_cost = None
roc = 0
flag_inc = False
if(momentum == 'off'):
    while (norm(w − prev_w) > EPS or norm(b − prev_b) > EPS) and iter < max_iter:
        prev_w = w.copy()
        prev_b = b.copy()
        sub_w = alpha∗gradient_w(x, y, w, b).reshape(sub_w.shape)
        sub_b = alpha∗gradient_b(x, y, w, b).reshape(sub_b.shape)
        w −= sub_w
        b −= sub_b
        performance[i:i+1,0] = iter
        performance[i:i+1,1] = NLL(softmax(compute_o(train_set,w,b)),train_set_label)/train_set.shape[1]
        performance[i:i+1,2] = NLL(softmax(compute_o(valid_set,w,b)),valid_set_label)/valid_set.shape[1]
        performance[i:i+1,3] = NLL(softmax(compute_o(test_set,w,b)),test_set_label)/test_set.shape[1]
        iter += 1
        if flag_inc == False and prev_cost != None and NLL(softmax(compute_o(valid_set,w,b)),
            ↪ valid_set_label)/valid_set.shape[1] > prev_cost:
            roc = iter
            flag_inc = True
        else:
            prev_cost = NLL(softmax(compute_o(valid_set,w,b)),valid_set_label)/valid_set.shape[1]
        i += 1
else:
    while (norm(w − prev_w) > EPS or norm(b − prev_b) > EPS) and iter < max_iter:
        prev_w = w.copy()
        sub_w = gradient_w(x, y, w, b).reshape(sub_w.shape)
        prev_v_w = v_w
        v_w = beta ∗ prev_v_w + alpha ∗ sub_w
        w −= v_w
        prev_b = b
        sub_b = gradient_b(x, y, w, b).reshape(sub_b.shape)
        prev_v_b = v_b
        v_b = beta ∗ prev_v_b + alpha ∗ sub_b
        b −= v_b
        performance[i:i+1,0] = iter
        performance[i:i+1,1] = NLL(softmax(compute_o(train_set,w,b)),train_set_label)/train_set.shape[1]
        performance[i:i+1,2] = NLL(softmax(compute_o(valid_set,w,b)),valid_set_label)/valid_set.shape[1]
        performance[i:i+1,3] = NLL(softmax(compute_o(test_set,w,b)),test_set_label)/test_set.shape[1]
        iter += 1
```

```
            if flag_inc == False and prev_cost != None and NLL(softmax(compute_o(valid_set,w,b)),
                ↪ valid_set_label)/valid_set.shape[1] > prev_cost:
                roc = iter
                flag_inc = True
            else:
                prev_cost = NLL(softmax(compute_o(valid_set,w,b)),valid_set_label)/valid_set.shape[1]
            i += 1

    #need to plot
    plt.figure()
    plt.plot(performance[:,0],performance[:,1]) #train set plot
    plt.plot(performance[:,0],performance[:,2]) #valid set plot
    plt.plot(performance[:,0],performance[:,3]) #test set plot
    plt.plot(roc*np.ones(performance[:,0].shape),performance[:,1], '--')
    plt.title('Learning Curve')
    plt.xlabel('Epoch')
    plt.ylabel('Cost')
    plt.legend(['Cost of training set', 'Cost of validation set','Cost of test set','Point where cost goes up: ' + str
        ↪ (roc)])
    plt.show()

    return performance

def disp_weights(w):
    '''
    input: the weight matrix
    output: visualization of the weights
    '''
    for i in range(10):
        vis = w[i:i+1,:].reshape((PIC_SIZE,PIC_SIZE))
        figure()
        imshow(vis, cmap = cm.gray)
        title("Visualized weights for " + str(i))
    show()

def get_optimum_param():
    '''
    input: none
    output: an array with the performances of different learning rates and max iterations
    '''
    size_per_digit = 100
    test_set_size = 50
    test_set, test_set_label = get_test_set(test_set_size)
    train_set, train_set_label = get_training_set(size_per_digit)
    alphas = [0.01, 0.001, 0.0001, 0.00001]
    iterations = [100, 1000, 10000, 100000]
```

```
    performance = np.zeros((len(alphas)*len(iterations),5))
    i = 0
    for l_r in alphas:
        for iter in iterations:
            start_time = time.time()
            w, b = train_param(train_set, train_set_label, l_r, iter)
            performance[i,2] = float(time.time() - start_time)
            performance[i,0] = l_r
            performance[i,1] = iter
            performance[i,3] = test_performance(train_set, train_set_label, w, b)
            performance[i,4] = test_performance(test_set, test_set_label, w, b)
            i += 1
            print('done_1_nested_loop')
            print(performance)
    return performance

#Part 5
def get_optimum_beta():
    '''
    input: none
    output: an array with the performances of different betas
    '''
    size_per_digit = 100
    test_set_size = 50
    test_set, test_set_label = get_test_set(test_set_size)
    train_set, train_set_label = get_training_set(size_per_digit)
    betas = [0.9, 0.99, 0.999, 0.9999]
    performance = np.zeros((len(betas),4))
    i = 0
    for beta in betas:
        start_time = time.time()
        w, b = train_momentum(train_set, train_set_label, beta)
        performance[i,1] = float(time.time() - start_time)
        performance[i,0] = beta
        performance[i,2] = test_performance(train_set, train_set_label, w, b)
        performance[i,3] = test_performance(test_set, test_set_label, w, b)
        print performance
        i += 1
    return performance

def grad_descent_momentum(dfw, dfb, x, y, init_w, init_b, alpha, beta):
    '''
    input: derivative functions, xs and ys, initial weights and biases, learning rate and momentum term
    output: optimized weights and biases, using gradient descent with momentum
    '''
    EPS = 1e-5 #EPS = 10**(-5)
```

```
    max_iter = 3000
    iter = 0

    prev_w = init_w−10∗EPS
    w = init_w.copy()
    sub_w = np.zeros(init_w.shape)
    v_w = 0

    prev_b = init_b−10∗EPS
    b = init_b.copy()
    sub_b = np.zeros(init_b.shape)
    v_b = 0

    while (norm(w − prev_w) > EPS or norm(b − prev_b) > EPS) and iter < max_iter:
        prev_w = w.copy()
        sub_w = dfw(x, y, w, b).reshape(sub_w.shape)
        prev_v_w = v_w
        v_w = beta ∗ prev_v_w + alpha ∗ sub_w
        w −= v_w

        prev_b = b
        sub_b = dfb(x, y, w, b).reshape(sub_b.shape)
        prev_v_b = v_b
        v_b = beta ∗ prev_v_b + alpha ∗ sub_b
        b −= v_b

        iter += 1
    return w, b
#Part 6
def plot_contour(disp='off'):
    '''
    input: (optional) display the contour or no
    output: w1 and w2 axis of the plot, z of the plot (contour)
    it also saves w1 and w2 and z into numpy files
    THIS TOOK 5 HOURS TO RUN, TEST ON YOUR OWN RISK (PATIENCE)
    '''
    train_set, train_set_label = get_training_set(400)
    w, b = train(train_set, train_set_label)
    w1_pos = (5,13∗14) #weight 1 going to number 5, best near 0.7
    w2_pos = (5,15∗14) #weight 1 going to number 5, best near 1.1
    w1_list = np.arange(−10,10,0.01)
    w2_list = np.arange(−10,10,0.01)
    w1z , w2z = np.meshgrid(w1_list,w2_list)
    w_copy = w.copy()
    cost_data = np.zeros((len(w2_list),len(w1_list)))
    for i,w1 in enumerate(w1_list):
```

```
        for j,w2 in enumerate(w2_list):
            w_copy[w1_pos[0],w1_pos[1]] = w1
            w_copy[w2_pos[0],w2_pos[1]] = w2
            cost = get_diff_loss(train_set,train_set_label,b,w,w_copy)
            cost_data[j,i] = cost
            print 'w1: ' + str(w1) + ', w2: ' + str(w2) + ', Cost: ' + str(cost)
    z = cost_data
    if disp == 'on':
        figure()
        CS = contour(w1z, w2z, z,levels = np.arange(z.min(),z.max(),abs(z.max()-z.min())/25))
        clabel(CS, inline=1, fontsize=10)
        title('Contour_plot')
        xlabel('w1')
        ylabel('w2')
        show()

    i = 0
    saved = False
    while (saved == False):
        string_w1 = 'contour_w1_v'+str(i)+'.npy'
        string_w2 = 'contour_w2_v'+str(i)+'.npy'
        string_cost = 'contour_cost_v'+str(i)+'.npy'
        try:
            temp = np.load(string_w1)
            i += 1
        except IOError:
            np.save(string_w1[:-4],w1z)
            np.save(string_w2[:-4],w2z)
            np.save(string_cost[:-4],z)
            saved = True

    return w1z, w2z, z


def grad_descent_k_step(dfw, dfb, x, y, init_w, init_b, alpha, k, w1_pos,w2_pos,w1_val,w2_val,beta,type_of_descent =
    ↪ 'Vanilla'):
    '''
    input: derivative functions, xs and ys, initial weights and biases, learning rate and beta, k steps, location of the
        ↪ weights and their value, type of descent (optional)
    output: w and b, a list of w1 and w2 across the steps
    '''
    EPS = 1e-5 #EPS = 10**(-5)
    prev_w = init_w-10*EPS
    w = init_w.copy()
    prev_b = init_b-10*EPS
    b = init_b.copy()
```

```
    max_iter = k
    iter = 0
    sub_w = np.zeros(init_w.shape)
    sub_b = np.zeros(init_b.shape)
    w_list = [(w1_val,w2_val)]


    v_w = 0
    v_b = 0

    if type_of_descent == 'Vanilla':
        while iter < max_iter:
            prev_w = w.copy()
            prev_b = b.copy()
            sub_w = alpha*dfw(x, y, w, b).reshape(sub_w.shape)
            sub_b = alpha*dfb(x, y, w, b).reshape(sub_b.shape)
            w -= sub_w
            b -= sub_b
            iter += 1
            w_list.append((w[w1_pos[0],w1_pos[1]],w[w2_pos[0],w2_pos[1]]))
    else:
        while iter < max_iter:
            prev_w = w.copy()
            sub_w = dfw(x, y, w, b).reshape(sub_w.shape)
            prev_v_w = v_w
            v_w = beta * prev_v_w + alpha * sub_w
            w -= v_w
            prev_b = b
            sub_b = dfb(x, y, w, b).reshape(sub_b.shape)
            prev_v_b = v_b
            v_b = beta * prev_v_b + alpha * sub_b
            b -= v_b
            iter += 1
            w_list.append((w[w1_pos[0],w1_pos[1]],w[w2_pos[0],w2_pos[1]]))

    return w, b,w_list #w is the fitted weights

def get_diff_loss(train_set,train_set_label,b,w,w_new):
    '''
    input: training set, w and b, new w
    output: difference between the costs of the old w and new w
    '''
    cost_old = NLL(softmax(compute_o(train_set,w,b)),train_set_label)
    cost_new = NLL(softmax(compute_o(train_set,w_new,b)),train_set_label)
    return abs(cost_new - cost_old)
```

```
def plot_trajectory(version,read_from_file = 'on'):
    '''
    input: version of the numpy file to read (for the contour)
    output: the trajectory
    '''
    #get training set and optimum weights
    train_set, train_set_label = get_training_set(400)
    w, b = train(train_set, train_set_label)

    w1_pos = (5,13*14) #weight 1 going to number 5
    w2_pos = (5,15*14) #weight 2 going to number 5

    #initialize weights away from optimum
    # w1_init = w[w1_pos[0],w1_pos[1]]+sign(w[w1_pos[0],w1_pos[1]])*0.03
    # w2_init = w[w2_pos[0],w2_pos[1]]+sign(w[w2_pos[0],w2_pos[1]])*0.03
    print 'optimum_w1:_' + str(w[w1_pos[0],w1_pos[1]])
    print 'optimum_w2:_' + str(w[w2_pos[0],w2_pos[1]])
    w1_init = -0.5
    w2_init = -0.5
    beta = 0.9
    w_copy = w.copy()
    w_copy[w1_pos[0],w1_pos[1]] = w1_init
    w_copy[w2_pos[0],w2_pos[1]] = w2_init
    w_k,b_k,w_steps = grad_descent_k_step(gradient_w,gradient_b,train_set,train_set_label,w_copy,b,0.001,30,w1_pos,
        ↪ w2_pos,w1_init,w2_init,beta,'Vanilla')

    w_k_m,b_k_m,w_steps_m = grad_descent_k_step(gradient_w,gradient_b,train_set,train_set_label,w_copy,b
        ↪ ,0.001,30,w1_pos,w2_pos,w1_init,w2_init,beta,'Momentum')

    if(read_from_file == 'on'):
        string = 'contour_w1_v' + str(version) + '.npy'
        cnt_w1 = np.load(string)
        string = 'contour_w2_v' + str(version) + '.npy'
        cnt_w2 = np.load(string)
        string = 'contour_cost_v' + str(version) + '.npy'
        cnt_data = np.load(string)
    else:
        cnt_w1,cnt_w2,cnt_data = plot_contour(disp='off')

    w_copy = w.copy()

    #display
    figure()
    CS = contour(cnt_w1, cnt_w2, cnt_data,levels = np.arange(cnt_data.min(),cnt_data.max(),(cnt_data.max()-
        ↪ cnt_data.min())/25))
    clabel(CS, inline=1, fontsize=10)
```

```
    title('Contour_plot')
    xlabel('w1')
    ylabel('w2')
    plt.plot([a for a, b in w_steps], [b for a,b in w_steps], 'yo-', label="No_Momentum")
    plt.plot([a for a, b in w_steps_m], [b for a,b in w_steps_m], 'gx-', label="Momentum")
    plt.legend(loc='top_left')
    show()
    return


if (__name__ == "__main__"):

    # train_set, train_set_label = get_training_set(50)
    # w, b = train(train_set, train_set_label)
    # test_set, test_set_label = get_test_set(10)
    # print test_performance(test_set,test_set_label,w,b)

    # results = plot_learning_curve()
    # disp_weights()
    # results = get_optimum_param()

    # result = get_optimum_beta()

    # performance_momentum = plot_learning_curve('on')

    # q,w,r = plot_contour('on')

    # figure()
    # CS = contour(q, w, r,levels = np.arange(0,1,0.01))
    # clabel(CS, inline=1, fontsize=10)
    # title('Contour plot')
    # xlabel('w2')
    # ylabel('w1')
    # show()


    # plot_trajectory(0)
```

faces.py

```
from pylab import *
import torch
from torch.autograd import Variable
import numpy as np
import matplotlib.pyplot as plt
```

```
import os
import urllib
import hashlib
from scipy.misc import imread
from scipy.misc import imresize
from scipy.misc import imsave
import matplotlib.image as mpimg
from scipy.io import loadmat
import timeit



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 hardcode_image_removal():
    '''remove images that do not contain a face, or generate an error later
    in the code'''
    remove_list = ['baldwin30.jpg', 'baldwin89.jpg', 'bracco87.jpg', 'bracco131.jpg', 'bracco104.jpg','bracco124.jpg','
        ↪ bracco138.jpg', 'carell54.jpg', 'carell121.jpg', 'carell146.jpg', 'hader80.jpg', 'hader122.jpg', 'hader15.jpg',
        ↪  'hader6.jpg', 'harmon49.jpg', 'harmon35.png']

    for image in remove_list:
        os.remove("faces_uncropped/"+image)
        os.remove("faces_cropped_32/"+image)
        os.remove("faces_cropped_64/"+image)

def rgb2gray(rgb):
    '''Return the grayscale version of the RGB image rgb as a 2D numpy array
    whose range is 0..1
    Arguments:
    rgb -- an RGB image, represented as a numpy array of size n x m x 3. The
    range of the values is 0..255
    '''


    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray/255.

def get_set():
    '''Download all images, process them and save for later use'''
    for actor in ['Alec_Baldwin', 'Lorraine_Bracco', 'Peri_Gilpin', 'Angie_Harmon', 'Bill_Hader', 'Steve_Carell']:
```

```
name = actor.split()[1].lower()
i = 0

for line in open("subset_faces.txt"):

    if actor in line:
        #download file
        filename = name+str(i)+'.jpg'

        try:
            urllib.urlretrieve(line.split()[4], "faces_uncropped/"+filename)
        except IOError:
            print 'couldnt_retrieve'
        except:
            print "uh_oh"
        if not os.path.isfile("faces_uncropped/"+filename):
            continue

        #convert to numpy array
        try:
            im = imread("faces_uncropped/"+filename)
        except IOError:
            print 'cannot_read_'+filename
            continue

        #crop, gray
        try:
            coord = line.split()[5].split(',')
            box = [int(x) for x in coord]
            im = im[box[1]:box[3], box[0]:box[2]]
            im = rgb2gray(im)
        except IndexError:
            print 'already_gray'

        #scale, save
        try:
            im_32 = imresize(im, (32,32))
            imsave('faces_cropped_32/'+filename, im_32)
        except ValueError:
            print 'image_too_small'

        try:
            im_64 = imresize(im, (64,64))
            imsave('faces_cropped_64/'+filename, im_64)
        except ValueError:
            print 'image_too_small'
```

```
                print filename
                i += 1

def get_test_train_valid(res, valid_size = 0):
    '''from the downloaded images, generate test, train and (optional) validation
    set'''
    np.random.seed(0)

    actors = ['baldwin', 'bracco', 'gilpin', 'harmon', 'hader', 'carell']
    dataset = {'baldwin':177, 'bracco':145, 'gilpin':121, 'harmon':136, 'hader':175, 'carell':183}
    remove = {'baldwin':[30,89], 'bracco':[87,131,104,124,138], 'gilpin':[], 'harmon':[49,35], 'hader':[80,122,15,6], '
        ↪ carell':[54,121,146]}

    test_x = np.zeros((0, res*res)) #store test set
    test_y = np.zeros( (0, 6)) #store labels
    train_x = np.zeros((0, res*res)) #store test set
    train_y = np.zeros( (0, 6)) #store labels

    if valid_size != 0:
        valid_x = np.zeros((0, res*res)) #store test set
        valid_y = np.zeros( (0, 6)) #store labels

    folder = "faces_cropped_" + str(res) + "/"

    for actor in actors:
        index_list = np.random.permutation(dataset[actor])

        index_list = index_list.tolist()

        for index in remove[actor]:
            index_list.remove(index)

        index_list = np.array(index_list)

        for i in range(20):
            filename = actor + str(index_list[i]) + ".jpg"

            im = imread(folder + filename).reshape(1,-1)/255.0
            test_x = np.concatenate((test_x, im))

            one_hot = np.zeros(6)
            one_hot[actors.index(actor)] = 1
            test_y = np.concatenate((test_y, one_hot.reshape(1,-1)))

        if valid_size == 0:
```

```
            for i in range(20, dataset[actor] - len(remove[actor])):
                filename = actor + str(index_list[i]) + ".jpg"

                im = imread(folder + filename).reshape(1,-1)/255.0
                train_x = np.concatenate((train_x, im))

                one_hot = np.zeros(6)
                one_hot[actors.index(actor)] = 1
                train_y = np.concatenate((train_y, one_hot.reshape(1,-1)))

        else:
            for i in range(20, 20+valid_size):
                filename = actor + str(index_list[i]) + ".jpg"

                im = imread(folder + filename).reshape(1,-1)/255.0
                valid_x = np.concatenate((train_x, im))

                one_hot = np.zeros(6)
                one_hot[actors.index(actor)] = 1
                valid_y = np.concatenate((train_y, one_hot.reshape(1,-1)))

            for i in range(20+valid_size, dataset[actor] - len(remove[actor])):
                filename = actor + str(index_list[i]) + ".jpg"

                im = imread(folder + filename).reshape(1,-1)/255.0
                train_x = np.concatenate((train_x, im))

                one_hot = np.zeros(6)
                one_hot[actors.index(actor)] = 1
                train_y = np.concatenate((train_y, one_hot.reshape(1,-1)))


    if valid_size == 0:
        return test_x, test_y, train_x, train_y

    else:
        return test_x, test_y, train_x, train_y, valid_x, valid_y

def optimize_and_evaluate(test_x, test_y, train_x, train_y, res, dim_h, learning_rate, batch_size, max_epoch, s,
    ↪ learning_curve = False, validation_x = None, validation_y = None, get_weights = False):
    '''Train the network. If learning_curve is True, require a validation set to
    generate a learning curve comparing training set with validation set. If
    get_weights is True, return the model to examine the weights later'''
    np.random.seed(s)
    start_time = timeit.default_timer()
```

```
dim_x = res*res
dim_out = test_y.shape[1]
batch_num = train_x.shape[0]/batch_size #number of batches in an epoch
dtype_float = torch.FloatTensor
dtype_long = torch.LongTensor

if learning_curve == True:

    axis_data = []
    y_val_data = []
    y_train_data = []

    val_x = Variable(torch.from_numpy(validation_x), requires_grad=False).type(dtype_float)
    val_y = Variable(torch.from_numpy(np.argmax(validation_y, 1)), requires_grad=False).type(dtype_long)
    full_train_x = Variable(torch.from_numpy(train_x), requires_grad=False).type(dtype_float)
    full_train_y = Variable(torch.from_numpy(np.argmax(train_y, 1)), requires_grad=False).type(dtype_long)

#define the model, loss function, optimizer

model = torch.nn.Sequential(
    torch.nn.Linear(dim_x, dim_h),
    torch.nn.ReLU(),
    torch.nn.Linear(dim_h, dim_out),
)

loss_fn = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

##Optimize

for i in range(max_epoch):
    #randomize and subsample for each step

    train_idx = np.random.permutation(range(train_x.shape[0]))

    for j in range(batch_num):
        try:
            subsample = train_idx[j * batch_size:(j+1)*batch_size]
        except(IndexError):
            continue

        x = Variable(torch.from_numpy(train_x[subsample]), requires_grad=False).type(dtype_float)
        y_classes = Variable(torch.from_numpy(np.argmax(train_y[subsample], 1)), requires_grad=False).type(
            ↪ dtype_long)
```

```
            #take one step:
            y_pred = model(x)
            loss = loss_fn(y_pred, y_classes)

            model.zero_grad() # Zero out the previous gradient computation
            loss.backward() # Compute the gradient
            optimizer.step() # Use the gradient information to
                             # make a step


        if learning_curve == True:
            y_val_pred = model(val_x)
            val_loss = loss_fn(y_val_pred, val_y).data.numpy()
            y_train_pred = model(full_train_x)
            train_loss = loss_fn(y_train_pred, full_train_y).data.numpy()

            axis_data.append(i)
            y_val_data.append(val_loss)
            y_train_data.append(train_loss)

    if get_weights == True:
        return model
    run_time = timeit.default_timer() - start_time

    ## Evaluate
    if learning_curve == False:
        x1 = Variable(torch.from_numpy(train_x), requires_grad=False).type(dtype_float)
        y_pred1 = model(x1).data.numpy()

        performance_train = np.mean(np.argmax(y_pred1, 1) == np.argmax(train_y, 1))

        x2 = Variable(torch.from_numpy(test_x), requires_grad=False).type(dtype_float)
        y_pred2 = model(x2).data.numpy()

        performance_test = np.mean(np.argmax(y_pred2, 1) == np.argmax(test_y, 1))

        return performance_train, performance_test, run_time
    else:
        return axis_data, y_val_data, y_train_data


def single_param_test(x,y,z,w, res, dim_h, learning_rate, batch_size, max_epoch):
    '''Run 10 tests on a set of parameters, changing the seed and taking
    the average'''
    performance_test = 0
    performance_train = 0
```

```
    time = 0

    for i in range(10):
        p_train, p_test, t = optimize_and_evaluate(x,y,z,w, res, dim_h, learning_rate, batch_size, max_epoch, i)
        performance_test += p_test
        performance_train += p_train
        time += t
        print i
    print "param_test"
    print "average_performance_on_the_training_set: "
    print performance_train/10.0
    print "average_performance_on_the_test_set: "
    print performance_test/10.0
    print "average_time: "
    print time/10.0

def multi_param_test():
    '''Experiment with parameters to find the optimal set. WARNING will take a
    very long time to run'''
    ## resolution = 32
    info_1 = {}
    x, y, z, w = get_test_train_valid(32)
    ##Finding learning rate

    # dimh = 20, batch_size = 600, max_epoch = 10000
    #lr = 1e-1
    performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-1, 600, 10000, 0)
    info_1[1e-1] = (performance_train, performance_test, time)
    print "#"
    #lr = 1e-2
    performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-2, 600, 10000, 0)
    info_1[1e-2] = (performance_train, performance_test,time)
    print "##"
    #lr = 1e-3
    performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-3, 600, 10000, 0)
    info_1[1e-3] = (performance_train, performance_test,time)
    print "###"
    #lr = 1e-4
    performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 600, 10000, 0)
    info_1[1e-4] = (performance_train, performance_test,time)
    print "####"
    #lr = 1e-5
    performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-5, 600, 10000, 0)
    info_1[1e-5] = (performance_train, performance_test,time)
    print "finding_learning_rate:"
    print info_1
```

```
#lr = 1e-3
print "1e-3_test"
single_param_test(x,y,z,w, 32, 20, 1e-3, 32, 1000)


#lr = 1e-4
print "1e-4_test"
single_param_test(x,y,z,w, 32, 20, 1e-4, 32 , 1000)

##Finding batch size

#dim_h = 20, lr = 1e-4, max_epoch = 10000
info_2 = {}
#batch_size = 32
performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 32 , 10000, 0)
info_2[32] = (performance_train, performance_test,time)
print "#"
#batch_size = 64
performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 64 , 10000, 0)
info_2[64] = (performance_train, performance_test,time)
print "##"
#batch_size = 128
performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 128 , 10000, 0)
info_2[128] = (performance_train, performance_test,time)
print "###"
#batch_size = 256
performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 256 , 10000, 0)
info_2[256] = (performance_train, performance_test,time)
print "####"
#batch_size = 512
performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 512 , 10000, 0)
info_2[512] = (performance_train, performance_test,time)
print "#####"
#batch_size = 600
performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 600 , 10000, 0)
info_2[600] = (performance_train, performance_test,time)
print "finding_batch_size:"
print info_2

#batch_size = 32
print "32_test"
single_param_test(x,y,z,w, 32, 20, 1e-4, 32 , 1000)

#batch_size = 64
print "64_test"
```

```
    single_param_test(x,y,z,w, 32, 20, 1e-4, 64 , 1000)

    ##Finding epoch size

    #dim_h = 20, lr = 1e-4, batch_size = 32
    info_3 = {}
    #max_epoch = 10
    performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 32 , 10, 0)
    info_3[10] = (performance_train, performance_test,time)
    print "#"
    #max_epoch = 100
    performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 32 , 100, 0)
    info_3[100] = (performance_train, performance_test,time)
    print "##"
    #max_epoch = 1000
    performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 32 , 1000, 0)
    info_3[1000] = (performance_train, performance_test,time)
    print "###"
    #max_epoch = 10000
    performance_train, performance_test, time = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 32 , 10000, 0)
    info_3[10000] = (performance_train, performance_test,time)
    print "####"
    print "finding_epoch_size:"
    print info_3

    #max_epoch = 1000
    print "1000_test"
    single_param_test(x,y,z,w, 32, 20, 1e-4, 32 , 1000)



    ## resolution = 64
    x, y, z, w = get_test_train_valid(64)
    print "resolution_64_test"
    single_param_test(x,y,z,w, 64, 20, 1e-4, 32, 1000)

def get_learning_curve():
    '''Generate a learning curve '''
    x, y, z, w, val_x, val_y = get_test_train_valid(32, 20)
    axis_data, y_val_data, y_train_data = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 32, 1000, 0, True, val_x,
        ↪ val_y)

    axis_data = np.array(axis_data)
    y_val_data = np.array(y_val_data)
    y_train_data = np.array(y_train_data)
```

```
    np.save("axis_data", axis_data)
    np.save("y_val_data", y_val_data)
    np.save("y_train_data", y_train_data)

    plt.plot(axis_data, y_train_data) #train set plot
    plt.plot(axis_data, y_valid_data) #valid set plot
    plt.title('Learning_Curve')
    plt.xlabel('Epoch')
    plt.ylabel('Cost')
    plt.legend(['Cost_of_training_set', 'Cost_of_validation_set'])
    plt.savefig("learning_curve_8")

def visualise_weights():
    '''Using the optimize and evaluate function, extract the complete model,
    determine which weights are important, and visualize them'''
    x, y, z, w = get_test_train_valid(32)
    model = optimize_and_evaluate(x,y,z,w, 32, 20, 1e-4, 32, 1000, 0, False, None, None, True)
    out_1 = np.absolute(model[2].weight.data.numpy()[0])
    index_1 = np.argpartition(out_1, -4)[-4:]
    np.save("indices_1", index_1)
    for index in index_1:
        weight_vis = model[0].weight.data.numpy()[index, :].reshape(32,32)
        np.save("baldwin"+str(index), weight_vis)

        #plt.imshow(weight_vis, cmap = plt.cm.coolwarm)
        #plt.savefig("baldwin"+index+".png")

    out_2 = np.absolute(model[2].weight.data.numpy()[1])
    index_2 = np.argpartition(out_2, -4)[-4:]
    np.save("indices_2", index_2)

    for index in index_2:
        weight_vis = model[0].weight.data.numpy()[index, :].reshape(32,32)
        np.save("bracco"+str(index), weight_vis)

        #plt.imshow(weight_vis, cmap = plt.cm.coolwarm)
        #plt.savefig("bracco"+index+".png")

    print model[2].weight.data.numpy()[0]
    print model[2].weight.data.numpy()[1]

if __name__ == "__main__":
    #Part 8
    get_set()
    multi_param_test()
    get_learning_curve()
```

```
    #Part 9
    visualise_weights()
```

deepfaces.py

```
from scipy.misc import imread
from scipy.misc import imresize
from scipy.misc import imsave
import torch
import torchvision.models as models
import torchvision
from torch.autograd import Variable
import urllib
import os
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import imread, imresize
import timeit
import torch.nn as nn

#All part 10

# We modify the torchvision implementation so that the features
# after the final pooling layer is easily accessible by calling
# net.features(...)
# If you would like to use other layer features, you will need to
# make similar modifications.
class MyAlexNet(nn.Module):
    def load_weights(self):
        an_builtin = torchvision.models.alexnet(pretrained=True)

        features_weight_i = [0, 3, 6, 8, 10]
        for i in features_weight_i:
            self.features[i].weight = an_builtin.features[i].weight
            self.features[i].bias = an_builtin.features[i].bias

    def __init__(self, num_classes=6):
        super(MyAlexNet, self).__init__()

        self.features = nn.Sequential(
            #Layer 1
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            #Layer 2
            nn.MaxPool2d(kernel_size=3, stride=2),
```

```
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            #Layer 3
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            #Layer 4
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            #Layer 5
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )


        self.load_weights()

    ## FORWARD PASS

    def forward(self, x):
        x = self.features(x)
        #This part flattens the output from the convolutional layers to input for fully connected layer
        x = x.view(x.size(0), 256 * 6 * 6)
        return x


'''
# model_orig = torchvision.models.alexnet(pretrained=True)
model = MyAlexNet()
#set model to eval mode (We will want to train it first?
model.eval()

##USE AS GUIDE FOR IMAGE PROCESSING
# Read an image
im = imread('kiwi227.png')[:,:,:3]
im = im - np.mean(im.flatten())
im = im/np.max(np.abs(im.flatten()))
im = np.rollaxis(im, -1).astype(float32)

# turn the image into a torch variable
im_v = Variable(torch.from_numpy(im).unsqueeze_(0), requires_grad=False)

# run the forward pass AlexNet prediction
softmax = torch.nn.Softmax()
all_probs = softmax(model.forward(im_v)).data.numpy()[0]
```

```
sorted_ans = np.argsort(all_probs)

for i in range(-1, -6, -1):
    print("Answer:", class_names[sorted_ans[i]], ", Prob:", all_probs[sorted_ans[i]])

ans = np.argmax(model.forward(im_v).data.numpy())
prob_ans = softmax(model.forward(im_v)).data.numpy()[0][ans]
print("Top Answer:", class_names[ans], "P(ans) = ", prob_ans)
'''
def download_images():

    for actor in ['Alec_Baldwin', 'Lorraine_Bracco', 'Peri_Gilpin', 'Angie_Harmon', 'Bill_Hader', 'Steve_Carell']:
        name = actor.split()[1].lower()
        i = 0

        for line in open("subset_faces.txt"):

            if actor in line:
                #download file
                filename = name+str(i)+'.jpg'

                try:
                    urllib.urlretrieve(line.split()[4], "faces_uncropped/"+filename)
                except:
                    print 'couldnt_retrieve'

                if not os.path.isfile("faces_uncropped/"+filename):
                    continue

                #convert to numpy array
                try:
                    im = imread("faces_uncropped/"+filename)[:,:,:3]
                except IOError:
                    print 'cannot_read_'+filename
                    continue
                except IndexError:
                    print 'not_colour_image'
                    continue

                #crop, gray
                try:
                    coord = line.split()[5].split(',')
                    box = [int(x) for x in coord]
                    im = im[box[1]:box[3], box[0]:box[2]]
                except:
                    print "whoops"
```

```
                #scale, save
                try:
                    im_227 = imresize(im, (227,227))
                    imsave('faces_cropped_227/'+filename, im_227)
                except ValueError:
                    print 'image_too_small'



                print filename
                i += 1

def hardcode_image_removal():
    #remove images that do not contain a face
    remove_list = ['baldwin0.jpg', 'baldwin28.jpg','baldwin31.jpg','baldwin90.jpg','baldwin104.jpg','baldwin129.jpg'
        ↪ ,'baldwin139.jpg', 'baldwin159.jpg', 'baldwin7.jpg', 'bracco4.jpg','bracco13.jpg','bracco17.jpg','bracco89
        ↪ .jpg','bracco90.jpg','bracco112.jpg','bracco135.jpg', 'bracco131.jpg', 'bracco141.jpg', 'gilpin4.jpg','
        ↪ gilpin7.jpg','gilpin52.jpg','gilpin71.jpg','gilpin88.jpg','gilpin117.jpg','hader6.jpg','hader15.jpg','hader40.
        ↪ jpg','hader73.jpg','hader75.jpg','hader85.jpg','hader94.jpg','hader95.jpg','hader103.jpg','hader108.jpg','
        ↪ hader117.jpg','hader129.jpg','hader133.jpg','hader134.jpg','hader137.jpg','hader143.jpg','harmon8.jpg','
        ↪ harmon12.jpg','harmon45.jpg','harmon52.jpg','harmon54.jpg','harmon75.jpg','harmon90.jpg','harmon95
        ↪ .jpg','harmon103.jpg','harmon119.jpg', 'carell16.jpg','carell38.jpg','carell44.jpg','carell65.jpg','carell79.
        ↪ jpg','carell84.jpg','carell103.jpg','carell104.jpg','carell128.jpg','carell133.jpg','carell137.jpg','carell142.jpg
        ↪ ','carell160.jpg','carell161.jpg']

    for image in remove_list:
        if os.path.isfile("faces_cropped_227/"+image):
            os.remove("faces_cropped_227/"+image)

def get_set():
    np.random.seed(0)

    actors = ['baldwin', 'bracco', 'gilpin', 'harmon', 'hader', 'carell']
    dataset = {'baldwin':182, 'bracco':154, 'gilpin':132, 'harmon':147, 'hader':190, 'carell':198}


    test_x = np.zeros((0,3,227,227)) #store test set
    test_y = np.zeros( (0, 6)) #store labels
    train_x = np.zeros((0,3,227,227)) #store test set
    train_y = np.zeros( (0, 6)) #store labels
    valid_x = np.zeros((0,3,227,227)) #store test set
    valid_y = np.zeros( (0, 6)) #store labels

    folder = "faces_cropped_227/"
```

```
for actor in actors:
    print actor
    index_list = np.random.permutation(dataset[actor])

    i = 0
    while test_x.shape[0] < 20*(actors.index(actor)+1):
        filename = actor + str(index_list[i]) + ".jpg"
        try:
            im = imread(folder + filename)
        except IOError:
            i += 1
            continue
        im = im - np.mean(im.flatten())
        im = im/np.max(np.abs(im.flatten()))
        im = np.expand_dims(np.transpose(im), axis=0)
        #im = im.astype(float32)
        test_x = np.concatenate((test_x, im),0)

        one_hot = np.zeros(6)
        one_hot[actors.index(actor)] = 1
        test_y = np.concatenate((test_y, one_hot.reshape(1,-1)))
        i += 1

    while valid_x.shape[0] < 20*(actors.index(actor)+1):
        filename = actor + str(index_list[i]) + ".jpg"
        try:
            im = imread(folder + filename)
        except IOError:
            i += 1
            continue

        im = im - np.mean(im.flatten())
        im = im/np.max(np.abs(im.flatten()))
        im = np.expand_dims(np.transpose(im), axis=0)
        #im = im.astype(float32)
        valid_x = np.concatenate((valid_x, im),0)

        one_hot = np.zeros(6)
        one_hot[actors.index(actor)] = 1
        valid_y = np.concatenate((valid_y, one_hot.reshape(1,-1)))
        i += 1

    for j in range(i, dataset[actor]):
        filename = actor + str(index_list[j]) + ".jpg"

        try:
```

```
            im = imread(folder + filename)
        except IOError:
            continue

        im = im - np.mean(im.flatten())
        im = im/np.max(np.abs(im.flatten()))
        im = np.expand_dims(np.transpose(im), axis=0)
        #im = im.astype(float32)
        train_x = np.concatenate((train_x, im),0)

        one_hot = np.zeros(6)
        one_hot[actors.index(actor)] = 1
        train_y = np.concatenate((train_y, one_hot.reshape(1,-1)))

    return test_x, test_y, train_x, train_y, valid_x, valid_y

def get_features(test_x, test_y, train_x, train_y, valid_x, valid_y, model):
    test_X = Variable(torch.from_numpy(test_x), requires_grad=False).type(torch.FloatTensor)
    test_features_x = model.forward(test_X)
    test_x_np = test_features_x.data.numpy()

    train_X = Variable(torch.from_numpy(train_x), requires_grad=False).type(torch.FloatTensor)
    train_features_x = model.forward(train_X)
    train_x_np = train_features_x.data.numpy()

    valid_X = Variable(torch.from_numpy(valid_x), requires_grad=False).type(torch.FloatTensor)
    valid_features_x = model.forward(valid_X)
    valid_x_np = valid_features_x.data.numpy()

    np.savez("features", test_x=test_x_np, test_y=test_y, train_x=train_x_np, train_y=train_y, valid_x=valid_x_np,
        ↪ valid_y=valid_y)

def optimize_and_evaluate(test_x, test_y, train_x, train_y, dim_h, learning_rate, batch_size, max_epoch, s,
    ↪ learning_curve = False, validation_x = None, validation_y = None, get_weights = False):
    np.random.seed(s)
    start_time = timeit.default_timer()
    dim_x = 256*6*6
    dim_out = test_y.shape[1]
    batch_num = train_x.shape[0]/batch_size #number of batches in an epoch
    dtype_float = torch.FloatTensor
    dtype_long = torch.LongTensor

    if learning_curve == True:

        axis_data = []
        y_val_data = []
```

```
    y_train_data = []

    val_x = Variable(torch.from_numpy(validation_x), requires_grad=False).type(dtype_float)
    val_y = Variable(torch.from_numpy(np.argmax(validation_y, 1)), requires_grad=False).type(dtype_long)
    full_train_x = Variable(torch.from_numpy(train_x), requires_grad=False).type(dtype_float)
    full_train_y = Variable(torch.from_numpy(np.argmax(train_y, 1)), requires_grad=False).type(dtype_long)

#define the model, loss function, optimizer

model = torch.nn.Sequential(
    torch.nn.Linear(dim_x, dim_h),
    torch.nn.ReLU(),
    torch.nn.Linear(dim_h, dim_out),
)

loss_fn = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

##Optimize

for i in range(max_epoch):
    #randomize and subsample for each step

    train_idx = np.random.permutation(range(train_x.shape[0]))

    for j in range(batch_num):
        try:
            subsample = train_idx[j * batch_size:(j+1)*batch_size]
        except(IndexError):
            continue

        x = Variable(torch.from_numpy(train_x[subsample]), requires_grad=False).type(dtype_float)
        y_classes = Variable(torch.from_numpy(np.argmax(train_y[subsample], 1)), requires_grad=False).type(
            ↪ dtype_long)

        #take one step:
        y_pred = model(x)
        loss = loss_fn(y_pred, y_classes)

        model.zero_grad() # Zero out the previous gradient computation
        loss.backward() # Compute the gradient
        optimizer.step() # Use the gradient information to
                         # make a step
```

```
        if learning_curve == True:
            y_val_pred = model(val_x)
            val_loss = loss_fn(y_val_pred, val_y).data.numpy()
            y_train_pred = model(full_train_x)
            train_loss = loss_fn(y_train_pred, full_train_y).data.numpy()

            axis_data.append(i)
            y_val_data.append(val_loss)
            y_train_data.append(train_loss)

    if get_weights == True:
        return model
    run_time = timeit.default_timer() - start_time

    ## Evaluate
    if learning_curve == False:
        x1 = Variable(torch.from_numpy(train_x), requires_grad=False).type(dtype_float)
        y_pred1 = model(x1).data.numpy()

        performance_train = np.mean(np.argmax(y_pred1, 1) == np.argmax(train_y, 1))

        x2 = Variable(torch.from_numpy(test_x), requires_grad=False).type(dtype_float)
        y_pred2 = model(x2).data.numpy()

        performance_test = np.mean(np.argmax(y_pred2, 1) == np.argmax(test_y, 1))

        return performance_train, performance_test, run_time
    else:
        return axis_data, y_val_data, y_train_data

def single_param_test(x,y,z,w, dim_h, learning_rate, batch_size, max_epoch):
    performance_test = 0
    performance_train = 0
    time = 0

    for i in range(10):
        p_train, p_test, t = optimize_and_evaluate(x,y,z,w, dim_h, learning_rate, batch_size, max_epoch, i)
        performance_test += p_test
        performance_train += p_train
        time += t
        print i
    print "param_test"
    print "average_performance_on_the_training_set:"
    print performance_train/10.0
    print "average_performance_on_the_test_set:"
    print performance_test/10.0
```

```
    print "average_time: "
    print time/10.0

def multi_param_test():
    features = np.load("features.npz")
    test_x = features["test_x"]
    test_y = features["test_y"]
    train_x = features["train_x"]
    train_y = features["train_y"]
    valid_x = features["valid_x"]
    valid_y = features["valid_y"]

    info_1 = {}
    ##Finding learning rate

    # dimh = 20, batch_size = 600, max_epoch = 10000
    #lr = 1e-1
    performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-1, 600,
        ↪ 10000, 0)
    info_1[1e-1] = (performance_train, performance_test, time)
    print "#"
    #lr = 1e-2
    performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-2, 600,
        ↪ 10000, 0)
    info_1[1e-2] = (performance_train, performance_test,time)
    print "##"
    #lr = 1e-3
    performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-3, 600,
        ↪ 10000, 0)
    info_1[1e-3] = (performance_train, performance_test,time)
    print "###"
    #lr = 1e-4
    performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 600,
        ↪ 10000, 0)
    info_1[1e-4] = (performance_train, performance_test,time)
    print "####"
    #lr = 1e-5
    performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-5, 600,
        ↪ 10000, 0)
    info_1[1e-5] = (performance_train, performance_test,time)
    print "finding_learning_rate:"
    print info_1

    #lr = 1e-4
    print "1e-4_test"
    single_param_test(test_x,test_y,train_x,train_y, 20, 1e-4, 600, 10000)
```

```
#lr = 1e-5
print "1e-5_test"
single_param_test(test_x,test_y,train_x,train_y, 20, 1e-5, 600, 10000)


##Finding batch size
print "batch_size"
#dim_h = 20, lr = 1e-4, max_epoch = 10000
info_2 = {}
#batch_size = 32
performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 32 ,
    ↪ 10000, 0)
info_2[32] = (performance_train, performance_test,time)
print "#"
#batch_size = 64
performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 64 ,
    ↪ 10000, 0)
info_2[64] = (performance_train, performance_test,time)
print "##"
#batch_size = 128
performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 128
    ↪ , 10000, 0)
info_2[128] = (performance_train, performance_test,time)
print "###"
#batch_size = 256
performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 256
    ↪ , 10000, 0)
info_2[256] = (performance_train, performance_test,time)
print "####"
#batch_size = 512
performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4,512 ,
    ↪  10000, 0)
info_2[512] = (performance_train, performance_test,time)
print "#####"
#batch_size = 600
performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 600
    ↪ , 10000, 0)
info_2[600] = (performance_train, performance_test,time)
print "finding_batch_size:"
print info_2


#batch_size = 64
print "64_test"
```

```
    single_param_test(test_x,test_y,train_x,train_y, 20, 1e-4, 64 , 10000)

    #batch_size = 128
    print "128_test"
    single_param_test(test_x,test_y,train_x,train_y, 20, 1e-4, 128 , 10000)

    ##Finding epoch size

    #dim_h = 20, lr = 1e-4, batch_size = 32
    info_3 = {}
    #max_epoch = 10
    performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 128
        ↪ , 10, 0)
    info_3[10] = (performance_train, performance_test,time)
    print "#"
    #max_epoch = 100
    performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 128
        ↪ , 100, 0)
    info_3[100] = (performance_train, performance_test,time)
    print "##"
    #max_epoch = 1000
    performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 128
        ↪ , 1000, 0)
    info_3[1000] = (performance_train, performance_test,time)
    print "###"
    #max_epoch = 10000
    performance_train, performance_test, time = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e-4, 128
        ↪ , 10000, 0)
    info_3[10000] = (performance_train, performance_test,time)
    print "####"
    print "finding_epoch_size:"
    print info_3

    #max_epoch = 1000
    print "1000_test"
    single_param_test(test_x,test_y,train_x,train_y, 20, 1e-4, 128 , 1000)

    #max_epoch = 10000
    print "10000_test"
    single_param_test(test_x,test_y,train_x,train_y, 20, 1e-4, 128 , 10000)

def get_learning_curve():
    '''Generate a learning curve '''
    features = np.load("features.npz")
    test_x = features["test_x"]
    test_y = features["test_y"]
```

```
    train_x = features["train_x"]
    train_y = features["train_y"]
    valid_x = features["valid_x"]
    valid_y = features["valid_y"]
    axis_data, y_val_data, y_train_data = optimize_and_evaluate(test_x,test_y,train_x,train_y, 20, 1e−4, 128 , 1000,
        ↪ 0, True, valid_x, valid_y)

    axis_data = np.array(axis_data)
    y_val_data = np.array(y_val_data)
    y_train_data = np.array(y_train_data)

    np.save("axis_data1", axis_data)
    np.save("y_val_data1", y_val_data)
    np.save("y_train_data1", y_train_data)

    plt.plot(axis_data, y_train_data) #train set plot
    plt.plot(axis_data, y_valid_data) #valid set plot
    plt.title('Learning_Curve')
    plt.xlabel('Epoch')
    plt.ylabel('Cost')
    plt.legend(['Cost_of_training_set', 'Cost_of_validation_set'])
    plt.savefig("learning_curve_10")

if __name__ == "__main__":
```