CSC411 Project 2

Zikun Chen
1001117882

February 23, 2018

Note:

- Program in this project is implemented in Python 2
- "mnist_all.mat" is not uploaded due to size limit. Please include this file in the working directory.
- Due to size limit, the training images for Part 10 are divided into two zip files "deep1" and "deep2". To reproduce the results, please decompress them and store the images in both folders in a new folder called "deep" in the working directory.
- Folders (zip files) "data" and "deep" store cropped and pre-processed face images that are used in Part 8 and Part 10 respectively. Images of pre-processed Peri Gilpin are already stored in these folders. (see Part 8 for futher explanation on pre-processing)

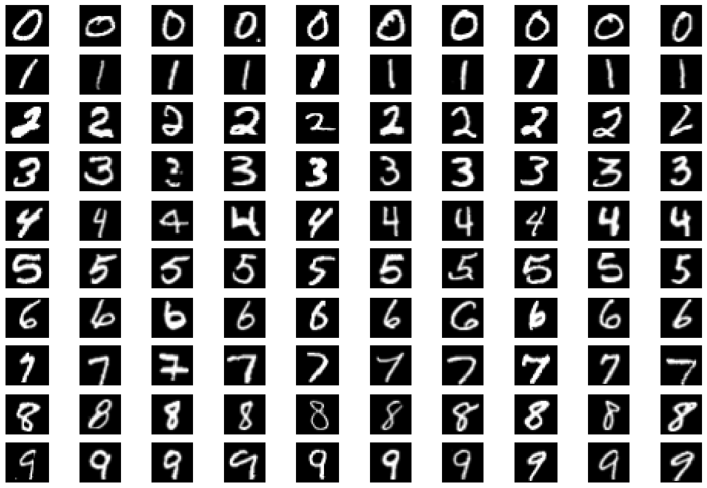
Part 1

Randomly sampling 10 images of each digit, we can see that the digits are normalized and positioned in the centre. There are different writing styles (e.g. 7 with a vertical line) but that is natural and does not interfere with the classification.

Part 2

Code:

```
# number of digits
N = 10
# dimension of images
K = 28*28

def output(W, X, b):
```



```
    M = X.shape[1]
    B = np.tile(b, M)
    O = np.dot(W.T, X) + B
    return O

def softmax(O):
    '''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 np.exp(O)/np.tile(np.sum(exp(O),0), (len(O),1))

def forward(W, X, b):
    O = output(W, X, b)
    P = softmax(O)
    return P
```

Part 3

(a) Note: Instead of i and j, I am choosing p and q for the indices of this particular weight w_{pq}.

First:

$N = 10$ is the number of outputs (digits) to classify

$K = 784$ is the dimension of the flattened input image

M is the number of training examples

$$W_{K\times N} = \begin{bmatrix} w_{11} & \cdots & w_{1N} \\ & \ddots & \vdots \\ w_{K1} & \cdots & w_{KN} \end{bmatrix}, \; X_{K\times M} = \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(M)} \\ \vdots & \ddots & \vdots \\ x_n^{(1)} & \cdots & x_n^{(M)} \end{bmatrix},$$

$$Y_{N\times M} = \begin{bmatrix} y_1^{(1)} & \cdots & y_1^{(M)} \\ \vdots & \ddots & \vdots \\ y_N^{(1)} & \cdots & y_N^{(M)} \end{bmatrix}, \; B_{N\times M} = \begin{bmatrix} b_1 & \cdots & b_1 \\ \vdots & \ddots & \vdots \\ b_N & \cdots & b_N \end{bmatrix}$$

Forward Pass:

$$o_i^{(s)} = \sum_{j=1}^{K} w_{ji} x_j^{(s)} + b_i$$

$$p_i^{(s)} = \frac{e^{o_i^{(s)}}}{\sum_{k=1}^{N} e^{o_k^{(s)}}}$$

$$C = -\sum_{s=1}^{M}\sum_{i=1}^{N} y_i^{(s)} log(p_i^{(s)})$$

Want $\frac{\partial C}{\partial w_{pq}}$ for a particular $p \in 1, \cdots, K$ and $q \in 1, \cdots, N$

Backword Pass:

$$\frac{\partial o_q^{(s)}}{\partial w_{pq}} = x_p^{(s)}$$

when $q = i$:

$$\frac{\partial p_i^{(s)}}{\partial o_q^{(s)}} = \frac{e^{o_i^{(s)}} \sum_{k=1}^{N} e^{o_k^{(s)}} - e^{o_i^{(s)}} e^{o_i^{(s)}}}{(\sum_{k=1}^{N} e^{o_k^{(s)}})^2} = \frac{e^{o_i^{(s)}}}{\sum_{k=1}^{N} e^{o_k^{(s)}}} \frac{(\sum_{k=1}^{N} e^{o_k^{(s)}} - e^{o_i^{(s)}})}{\sum_{k=1}^{N} e^{o_k^{(s)}}}$$

$$= p_i^{(s)}(1 - p_i^{(s)}) = p_q^{(s)}(1 - p_q^{(s)})$$

when $q \neq i$:

$$\frac{\partial p_i^{(s)}}{\partial o_q^{(s)}} = \frac{-e^{o_i^{(s)}} e^{o_q^{(s)}}}{(\sum_{k=1}^{N} e^{o_k^{(s)}})^2} = -p_i^{(s)} p_q^{(s)}$$

So:

$$\frac{\partial C}{\partial o_q^{(s)}} = -\sum_{s=1}^{M}\sum_{i=1}^{N} \frac{y_i^{(s)}}{p_i^{(s)}} \frac{\partial p_i^{(s)}}{\partial o_q^{(s)}} = -\sum_{s=1}^{M} [\frac{y_q^{(s)}}{p_q^{(s)}} p_q^{(s)}(1 - p_q^{(s)}) - \sum_{i\neq q} \frac{y_i^{(s)}}{p_i^{(s)}} p_i^{(s)} p_q^{(s)}]$$

$$= \sum_{s=1}^{M} [\sum_{i\neq q} y_i^{(s)} p_q^{(s)} - y_q^{(s)}(1 - p_q^{(s)})] = \sum_{s=1}^{M} [\sum_{i\neq q} y_i^{(s)} p_q^{(s)} - y_q^{(s)} + y_q^{(s)} p_q^{(s)}]$$

$$= \sum_{s=1}^{M} [p_q^{(s)} \sum_{i=1}^{N} y_i^{(s)} - y_q^{(s)}] = \sum_{s=1}^{M} (p_q^{(s)} - y_q^{(s)})$$

Note: $\sum_{i=1}^{N} y_i^{(s)} = 1$ since it is a one hot encoded vector for the labels

Therefore:

$$\frac{\partial C}{\partial w_{pq}} = \frac{\partial C}{\partial o_q^{(s)}} \frac{\partial o_q^{(s)}}{\partial w_{pq}} = \sum_{s=1}^{M} (p_q^{(s)} - y_q^{(s)}) x_p^{(s)}$$

(b) **Vectorized Code:**

```
def gradient(X, Y, W, b):
    '''Incomplete function for computing the gradient of the cross-entropy
    cost function w.r.t the parameters of a neural network'''
    P = forward(W, X, b)
    dCdO = P - Y
    dCdW = np.dot(X, dCdO.T)
    # change to dCdb
    # dCdB = np.tile(np.sum((y - y_), 1), (M, 1)).T
    dCdb = np.sum(dCdO, 1).reshape(N, 1)
    return dCdW, dCdb
```

Gradient Check:

```
def check_gradient(X, Y, W, b, i, j, h=1e-7):
    deltaij = np.zeros(W.shape)
    deltaij[i, j] = h

    P = forward(W, X, b)
    cost = NLL(P, Y)

    P_h = forward(W+deltaij, X, b)
    cost_h = NLL(P_h, Y)

    dCdWij = gradient(X, Y, W, b)[0][i, j]
    finite_diff = (cost_h - cost)/h

    #dj = dJ(X, Y, W)[i, j]

    print 'W[%1d, %1d]'%(i, j)
    print '--------------------------------------'
    print 'dCdwij: %.7f\nfinite-difference approximation: %.7f' \
        % (dCdWij, finite_diff)
    print 'absolute difference: %.7f\n' % (abs(dCdWij - finite_diff))
```

Output:

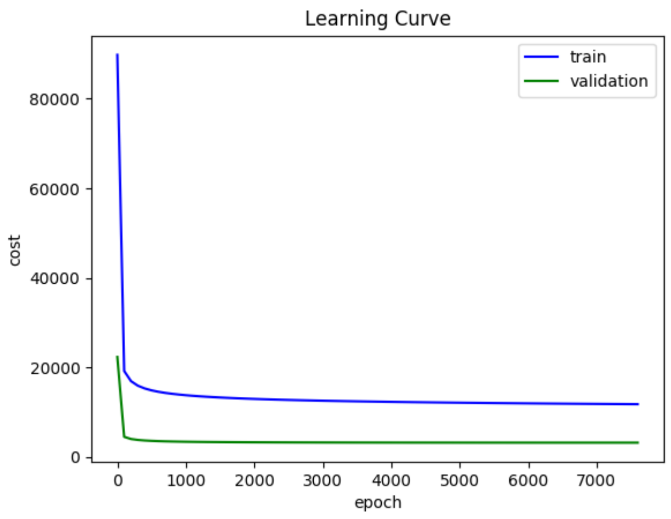
```
W[456, 3]
--------------------------------------
dCdwij: 1068.5822088
finite-difference approximation: 1068.5822053
absolute difference: 0.0000035

W[47, 0]
--------------------------------------
dCdwij: 1.0524879
finite-difference approximation: 1.0523945
absolute difference: 0.0000934

W[209, 5]
--------------------------------------
dCdwij: 130.6694340
finite-difference approximation: 130.6699414
absolute difference: 0.0005074

W[653, 9]
--------------------------------------
dCdwij: 642.8394354
finite-difference approximation: 642.8391498
absolute difference: 0.0002856

W[200, 4]
--------------------------------------
dCdwij: 6.9406903
finite-difference approximation: 6.9406815
absolute difference: 0.0000088
```

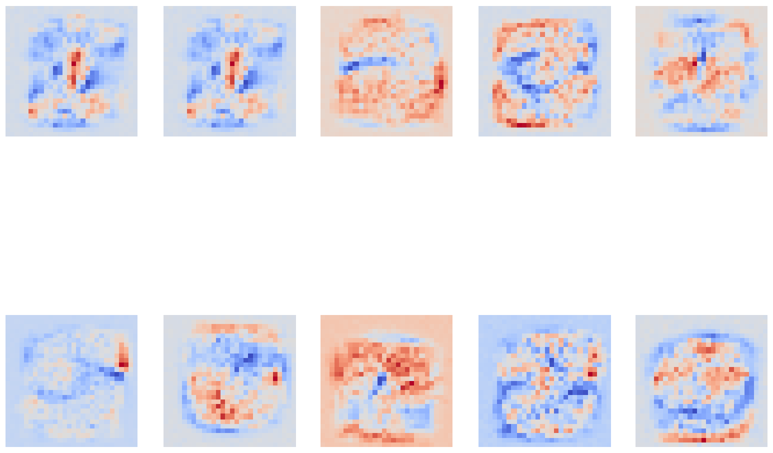
Part 4

Learning Curve:
Weights Visualization:
Optimization Procedure: To initialize the weights, I loaded and randomly sampled weights and biases from the "snapshot50.pkl" file provided in the handout. I used seed and np.random.shuffle for reproducible results.

10^{-5} seems to be the optimal learning rate since both 10^{-4} and 5×10^{-5} cause training to diverge.

I also implemented early stopping into the learning algorithm. It counts the number of the last few elements in the list of validation losses, which are recorded every few hundred epoch. If the count exceeds the limit, then the training will be stopped since over-fitting is starting to occur. The limiting





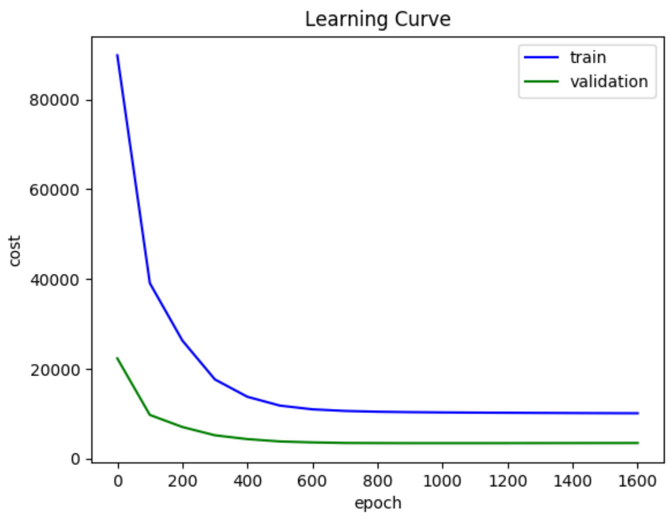
weights going into each of output units

count and frequency of recording the losses can be customized in the training function as parameters. In the code, the limiting count is set to 5 and the frequency is set to 100.

The training will stop early if weights converge and the norm if the weights stop changing (using small quantity EPS) or exceeds maximum iteration limit.

Part 5

Learning Curve:



Comparing with vanilla gradient descent: The momentum method makes training faster according to the learning curves. The training and validation cost while using the momentum method reaches the same level as those with vanilla gradient descent using a fifth of the time (1600 epoch vs. 8000 epoch)

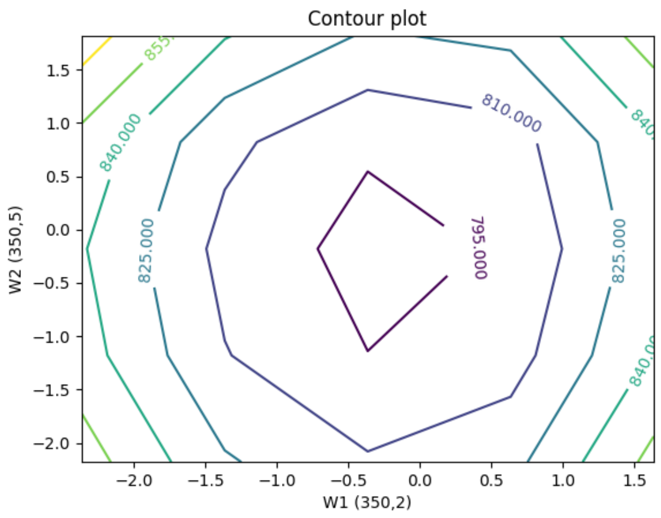
New Function for Momentum Training:

```
def momentum_train(X_train, X_valid, Y_train, Y_valid, init_W, init_b, alpha = 1e-6, \
                   gamma = 0.99, EPS=1e-5, count = 5, freq=100, max_iter=10000):

    prev_W = init_W-10*EPS
    W = init_W.copy()

    prev_b = init_b-10*EPS
    b = init_b.copy()

    global W_mom
global b_mom


change_W = np.zeros(W.shape)
change_b = np.zeros(b.shape)

iter  = 0


# gradient descent with momentum
while np.linalg.norm(W - prev_W) > EPS and iter < max_iter \
    and np.count_nonzero(np.diff(valid_costs_mom)[-5:]>0) < count:
    prev_W = W.copy()
    prev_b = b.copy()

    prev_change_W = change_W
    prev_change_b = change_b

    P = forward(W, X_train, b)

    dCdW, dCdb = gradient(X_train, Y_train, W, b)

    change_W = gamma * prev_change_W + alpha * dCdW
    change_b = gamma * prev_change_b + alpha * dCdb

    W -= change_W
    b -= change_b

    W_mom = W
    b_mom = b


    if iter % freq == 0:
        P_train = forward(W, X_train, b)
        t_cost = NLL(P_train, Y_train)
        dCdW, dCdb = gradient(X_train, Y_train, W, b)

        print "Epoch ", iter

        # print gradient
        print "dCdw =  ", dCdW, "\n"
        print "dCdb =  ", dCdb, "\n"

        # report costs
        P_valid = forward(W, X_valid, b)
        v_cost = NLL(P_valid, Y_valid)
        train_costs_mom.append(t_cost)
        valid_costs_mom.append(v_cost)

        print '\ntraining cost: %.7f\nvalidation cost: %.7f\n'% (t_cost, v_cost)

        epoch_mom.append(iter)
    iter += 1
```

Part 6

Note: The indices of weights that are allowed to vary are included in the axis lables.

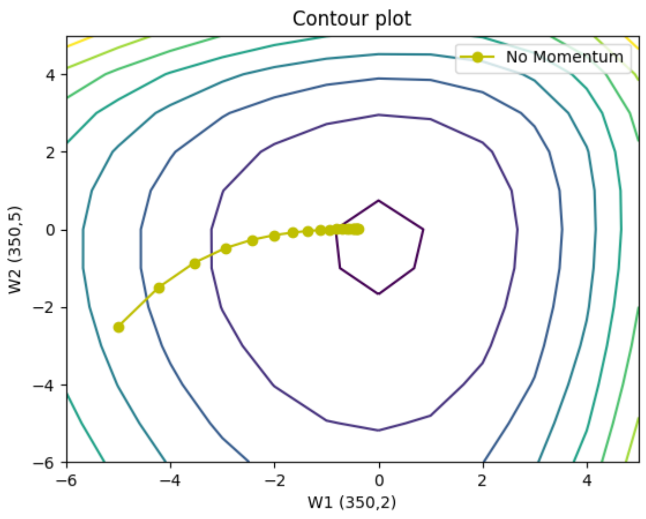
(a) **Contour Plot of Cost Function:**

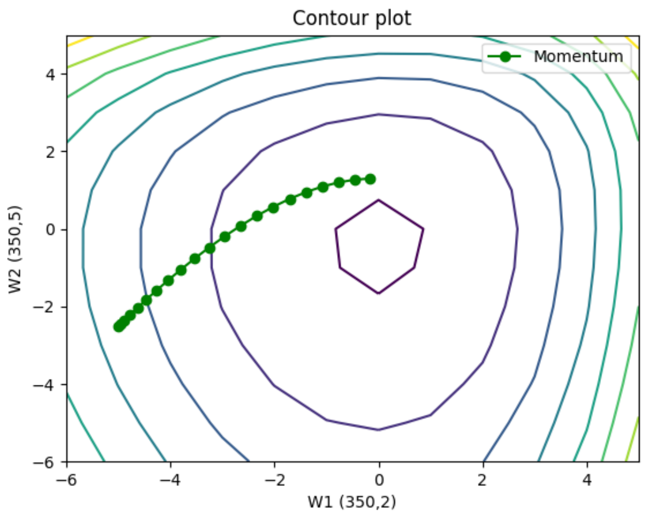


(b) **Vanilla Gradient Descent:**

(c) **Gradient Descent with Momentum:**

(d) The momentum method has a trajectory that is slanted towards the side whereas the vanilla gradient descent has a more direct path towards the minimum. This is due to the fact that momentum takes into the account of the previous changes in weights and these changes are being aggregated together. It is also faster, the learning rates for momentum and vanilla are 10^{-3} and 2×10^{-2} respctively. The momentum method had a much slower learning rate but was able to get to a similar place on the contour plot as the vanilla gradient descent.





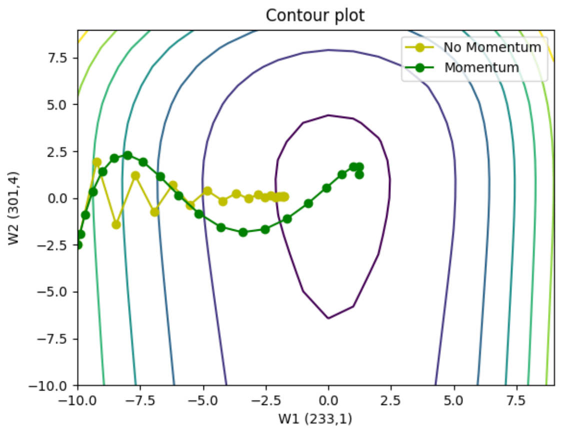
(e) To find settings where momentum works better, we want to try different elements of the trained weight matrix associated with the center of the digits and look at the contour plots of the lost function with only the chosen weights varying.

If the contour plot shows a eclipse, then we know the gradient descent trajectory will swing back and forth in the ravine, in which case, the momentum will perform better in training.

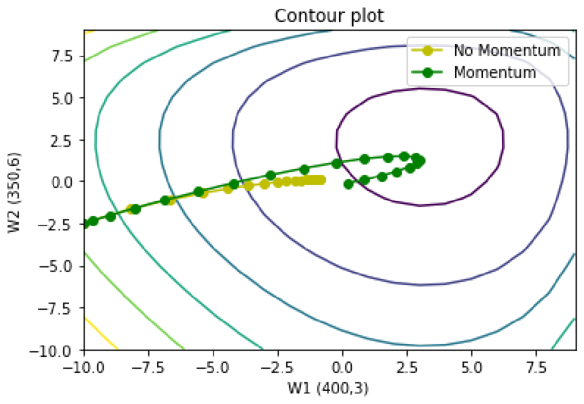
It is harder to produce visualization because we have to look for the weights appropriate to the two scenarios and plot both trajectories with different algorithms and learning rates.

The trajectory of vanilla gradient descent has a zig-zag pattern because each step of the gradient descent points to the direction of the steepest descent of the cost function regardless of the previous step. On the other hand, the momentum method takes into account the previous changes. If the gradient direction is very different from the previous step, the momentum will produce a step that smooths the trajectory and dampens oscillation. Otherwise, if the gradient direction is very similar to the previous steps, it's going to speed up the descent by aggregating those previous directions.

Momentum working better:



Vanilla working better:



Part 7

Architecture and Assumptions:

The neural network has N layers, with the first layer being the input layer with dimension M and last layer being the output layer with dimension S. Other hidden layers have dimension K. Assume all layers are fully-connected. The S units in the last layer then go through a softmax activation function and the probabilities are aggregate through a cross entropy loss function. We assume all other layers are linear have no activation functions. In addition, let the number of training examples be d. Assume that the complexity for a multiplication between matrix $A_{n\times m}$ and matrix $B_{m\times p}$ is $O(nmp)$ since there are m additions and multiplications for each of the $(n \times p)$ components in the result matrix. Further, assume that we use vectorized gradient computation for weights in any layer for both calculation with and without back propagation. Now, let's compare the complexity of computing vectorized gradients for a single epoch with and without back propagation.

Complexity with Back Propagation:

In the derivation of a single layer neural network in Part 3, we can see that the vectorized gradient for the weights are matrix multiplication with $O(NMK)$. Similarly, the cost of computing the gradient matrix of weights in the last layer for our network is $O(KSd)$. For weights in a hidden layer, the cost of computing the gradient matrix is $O(KKd)$. For the input layer, the complexity is $O(KMd)$

Since we are caching the gradients from the previous layers in back propagation, the cost of computing vectorized gradients for all the weights in the network is:

$$\begin{aligned} &O(KSd + KKd + \cdots + KKd + KMd) \\ &= O(KSd + (N-2)KKd + KMd) \\ &= O(Kd(s + (N-2)K + M)) \end{aligned}$$

Complexity without Back Propagation:

In this case, for any layer in the network, we cannot utilize the cached weights from the previous layer, so we have to compute the vectorized gradient for the weights in the previous layer and its previous layer and so on. This means that the computation cost will be:

$$O(KSd{+}(KSd{+}KKd){+}(KSd{+}KKd{+}KKd){+}\cdots{+}(KSd{+}(N{-}2)KKd){+}KMd)$$

$$\begin{aligned} &= O((N-1)KSd + [1 + \cdots + (N-2)]KKd + KMd) \\ &= O((N-1)KSd + \frac{(N-2)(N-1)}{2}KKd + KMd) \\ &= O(Kd[(N-1)S + \frac{(N-2)(N-1)}{2}K + M]) \end{aligned}$$

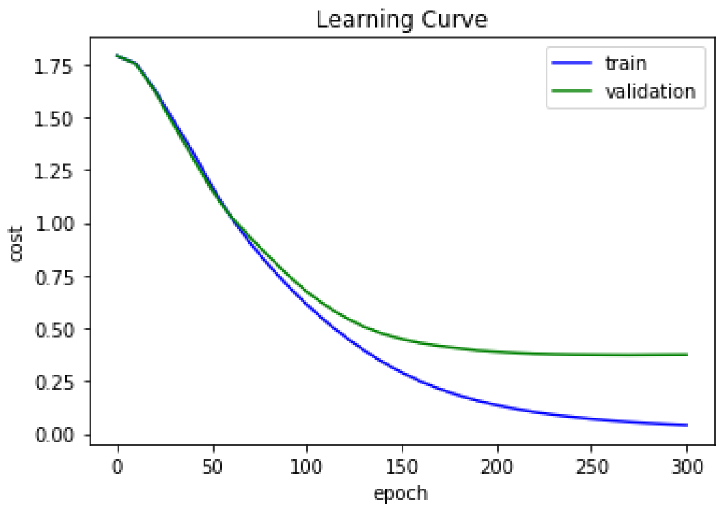
Therefore, when the network is deep and N is large, the cost of computing gradients are very high if back propagation is not being performed and the gradient matrices are not cached for computing gradients in layers further down the network.

Part 8

Final Performance:

```
--------- Final Performance ---------

training accuracy: 100.00%
validation accuracy: 85.09%
test accuracy: 77.39%
```

Learning Curve:



To preprocess the images, I compared the SHA-256 strings of the downloaded images to the line in the text file to skip bad images. I stored them in the folder "uncropped" and cropped them in "data" folder if the SHA-256 strings are consistent. However, for Peri Gilpin, I did not use this method because it doesn't produce enough images for her. So I downloaded her images to another folder called "gilpin" without checking the SHA-256 strings and cropped them in the folder "gilpin_cropped". And then I moved them into the "data" folder with other actors. After the SHA-256 prepossessing described above, there are no additional bad images to be removed.

Because of the lack of images for Peri Gilpin, while splitting the data sets for training/validation/test, I have proportionally decrease the number of images for Peri Gilpin in the three sets according to the split of other actors (87 in total for Gilpin). They have equal number of images in the three sets (percisely, 79 for training, 20 for validation and testing for each actor).

For png images with 4 channels, I selected the first 3 channels which correspond to RGB channels. For black and white images with 2 dimensions, I copied the image in the gray channel three times to create the 3rd dimension.

I chose to initialize the weights and biases by using a random normal distribution with np.random.seed for reproducing the results and learning curves.

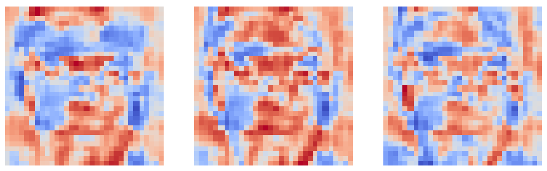
The architecture of the neural network consists of a fully connected network with a single hidden layer with 21 units . The activation functions I used is ReLU, and the output activation is softmax. The loss function is cross entropy loss. The learning algorithm is Adam. Resolution of the face images is 32×32.

This setting produces the best performances on the validation and test sets after experimenting with different optimizer(Adagrad, SGD, Adam), learning rate, activation functions, weight initializations (changing variance of the Normal distribution) and different number of hidden units. Similar to part 4, I also implemented early stopping to prevent over-fitting.

Part 9

To find the hidden units that are most useful for classifying an actor, I first converted the image into a torch variable and sent it through the network to make sure it classifies the image correctly. Then I indexed the model to extract the output of the hidden units to figure out which hidden unit contains the largest value, in order words, which hidden units contributes the most for classifying the image correctly. After that, I picked out the weights associated to that hidden unit and visualize the weights by displaying each RGB channel separately.

Angie Harmon:

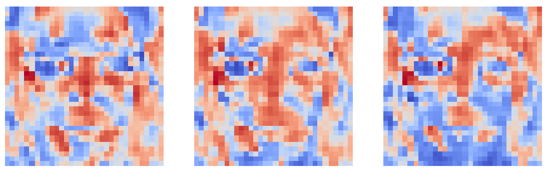


Alec Baldwin:

Part 10

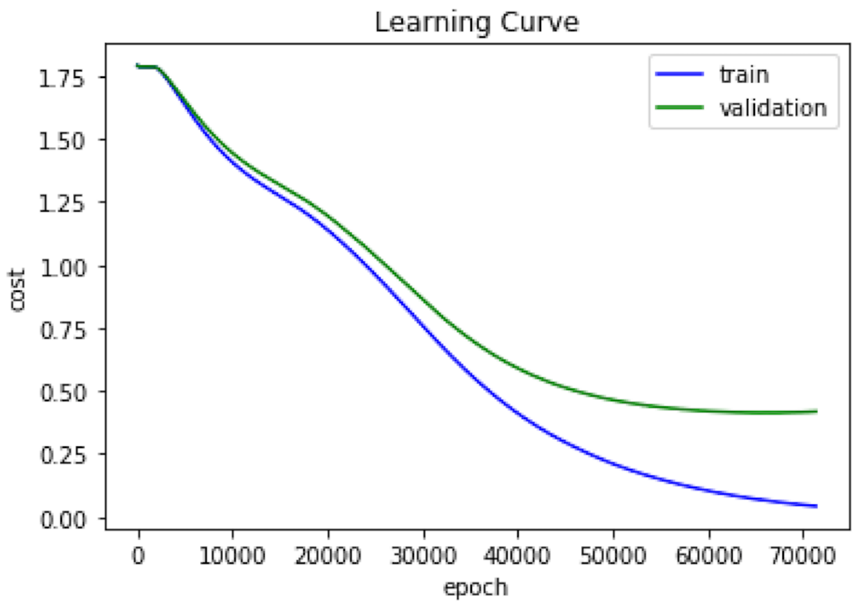
Final Costs and Performance:

```
Epoch  71400
```



```
training cost: 0.0426604
validation cost: 0.4183668


--------- Final Performance ---------

training accuracy: 100.00%
validation accuracy: 85.22%
test accuracy: 87.83%
```

Learning Curve:



Similar to Part 8, to preprocess the images, I downloaded and cropped images of Peri Gilpin individually without SHA-256 hashes. The preprocessing for other actors remain the same. The only difference with Part 8 is that I cropped the images as 227 × 227. In addition, as suggested, when constructing the data sets for training, I applied this transformation to each image im:

```
    im = im - np.mean(im.flatten())
    im = im/np.max(np.abs(im.flatten()))
    im = np.rollaxis(im, -1).astype(np.float32)
    im = im.reshape(1, 3, 227, 227)
```

Similar to Part 8, because of the lack of images for Peri Gilpin, I have proportionally decrease the number of images for Peri Gilpin in the three sets according to the split of other actors. Although this time, there are 88 total images for Gilpin and for other actors, 78 for training, 20 for validation and testing.

This is to match the input dimension of the AlexNet for extracting features from the convolution layer.

Similar to Part 8, I chose to initialize the weights and biases by using a random normal distribution with np.random.seed for reproducing the results and learning curves.

The architecture consists of a fully connected 2-layer network with 1 hidden layer with 50 units (model_full). The activation for the hidden layer is ReLU. This is on top of the AlexNet architecture up to and including Conv4 and its activation ReLU function. The weights in the AlexNet portion of the network are loaded from the trained model and fixed throughout the learning. The output activation is softmax and the loss function is cross entropy loss. The learning algorithm is Adam. Similar to part 4, I also implemented early stopping to prevent over-fitting. I fed the entire data set once through the AlexNet (model_alex) once to extract features before training. Then I used the extracted features as the new data set for training through the full connected network.

Upon considering and experimenting different activation functions, number of hidden units and learning algorithms, weight internalization (with different variances) and learning rates. This setting with 50 hidden units with learning rate 10^-5 performs the best. Comparing to the performances of several attempts with the model in Part 8, feature extraction performs better on the validation and test sets although requiring more time to train.