CSC411: Assignment #2

Due on Friday, February 23, 2018

Kaiyang Chen, Weixin Liu

February 24, 2018

Part 0

Download DataSet

Two datasets are used in this project.

For *Part 1* to *Part 6*, this project works with MNIST dataset. It is a dataset of digits (0 to 9), which is avaiable at: http://yann.lecun.com/exdb/mnist/

For *Part 8* to *Part 10*, this project works with images of actors and actress. It is a subset from the FaceScrub website, which is available at: http://vintage.winklerbros.net/facescrub.html

Part 1

Dataset description

From Part 1 to 6, this project works with a dataset of digits (0 to 9). The total number of digits downloaded from the website is 70, 000. The dataset is pre-split into 60, 000 images in training set and 10, 000 images in the test set. Each image is 28×28-pixel. Those images may vary in boldness, font, and written style.

In Figure 1, 10 images of each of the digits is shown. It is evident that some of the digits may be hard to classify, even to humans. For example, the left-most 4 can be interpreted as a 9, the right-most 8 can be read as a 1. Moreover, the sixth 1 from the left seems to be cut-off from the top. These particular examples may make the classification job harder and lower the accuracy, as we may see in later parts.

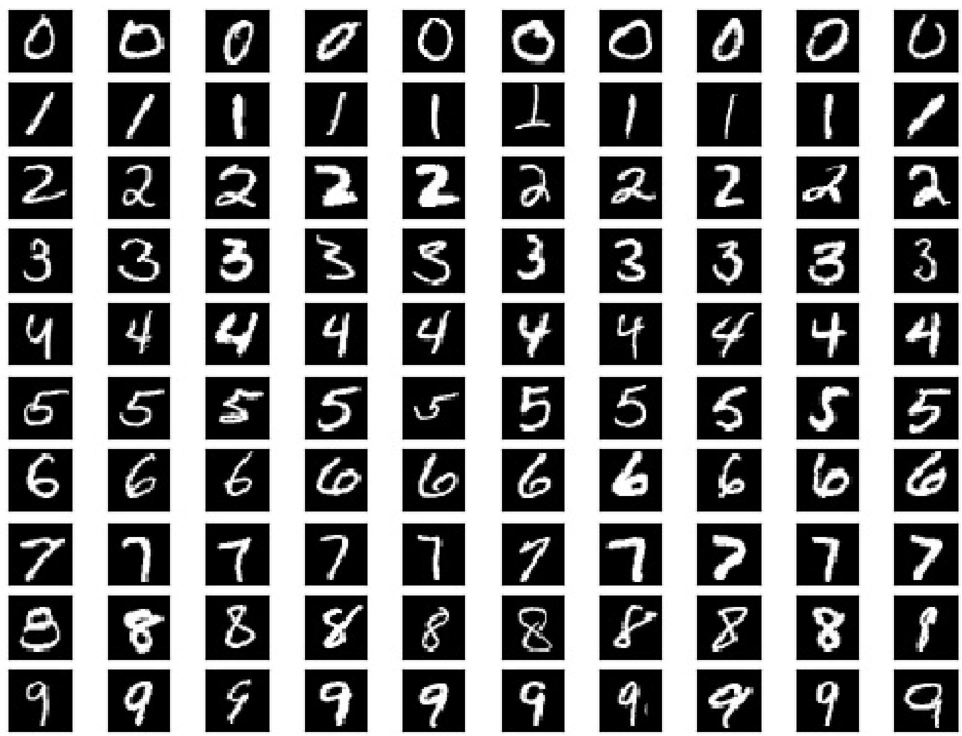


Figure 1: Visualizations of the digits dataset

Part 2

Implement Forward Step

The forward step of the neural network is implemented. The function `softmax(Y)` computes the softmax for the output neuron. The function `forward(X, W0b0)` computes the forward step of the neural network. In this implementation, the activation function is simply the linear combinations of the x's. The listing of the implementation is below:

```
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 forward(X, W0b0):
    '''
    Function takes in X, the input, which is n*784 matrix, n being the
    and W0b0, the weights matrix, which is a 785*10 matrix. The first 784 rows
    corresponds to the neuron weights, the last row is the bias.
    Function returns the Output neurons O and the output after taking the
    softmax function.
    Both of the outputs have dimension 10*n.
    '''
    W0 = W0b0[:-1,:]
    b0 = W0b0[-1:,:].T
    O = dot(W0.T, X.T) + b0
    output = softmax(O)
    return O, output
```

Part 3

Cost function of the neural network

Part 3a)

The gradient of the cost function with respect to the weight w_{ij} is computed as follows. The final result is in Eq. 4:
The Cost function over all the training example is:

$$Cost = -\sum_k \sum_i y_i^{(k)} logp_i^{(k)} \tag{1}$$

where Y is the label, P is the output prediction. k is the k-th data point (image), and i is the i-th pixel in one data point (image).
The output prediction is computed using softmax function.

$$p_i^{(k)} = \frac{e^{o_i^{(k)}}}{\sum_z e^{o_z^{(k)}}} \tag{2}$$

where i,k are defined same as above, O is the output neuron, and z is the z-th output neuron.
The outout is computed using linear activation function.

$$o_i^{(k)} = \sum_j w_{j,i} x_j^{(k)} + b_{i,k} \tag{3}$$

where O is the output neuron, W is the weight, X is the input, b is bias. i, and k are defined same as above
Now, the gradient of the cost function with respect to the weight is computed as follows.

$$\begin{aligned} \frac{\partial Cost}{\partial w_{i,j}} &= \sum_k \frac{\partial Cost}{\partial O^{(k)}} \frac{\partial O^{(k)}}{\partial w_{i,j}} \\ &= \sum_k (\sum_i \frac{\partial Cost}{\partial p_i^{(k)}} \frac{\partial p_i^{(k)}}{\partial o_i^{(k)}}) \frac{\partial o_i^{(k)}}{\partial w_{i,j}} \\ &= \sum_k (p_i^{(k)} - y_i^{(k)}) x_j^{(k)} \end{aligned} \tag{4}$$

Some of the steps above requires further justification by taking the partial derivatives of Eq. 1, 2, and 3, see below:

$$\frac{\partial Cost}{\partial p_i^{(k)}} = -\frac{y_i^{(k)}}{p_i^{(k)}}$$

$$\frac{\partial p_i^{(k)}}{\partial o_i^{(k)}} = p_i^{(k)}(1 - p_i^{(k)})$$

Thus we have:

$$(\sum_i \frac{\partial Cost}{\partial p_i^{(k)}} \frac{\partial p_i^{(k)}}{\partial o_i^{(k)}}) = (p_i^{(k)} - y_i^{(k)})$$

because only one of the y_i is 1 for each k, with all other y_i's being zero. This equation is same as what we devired in class. Finally,

$$\frac{\partial o_i^{(k)}}{\partial w_{i,j}} = x_j^{(k)}$$

Part 3b)

The vectorized function to calculate the gradient of the cost function is implemented as follows. The function descriptions are written in the function comments. Function `CostFunction(X, Y, W0b0)`, `Gradient(X, Y, W0b0)` are used to compute the vectorized cost function gradient.

```
def CostFunction(X, Y, W0b0):
    '''
    Use Negative log-probabilities of all training cases as the cost function
    '''
    O, output = forward(X,W0b0)
    return -sum(Y*log(output))

def Gradient(X, Y, W0b0):
    '''
    Return Gradient of the cost function
    The first return matrix is the gradient w.r.t. the W0 matrix (weights), it has
    dimension 784 * 10
    The second return matrix is the gradient w.r.t. the b0 matrix (bias), it has
    dimension 10*1
    '''
    O, output = forward(X, W0b0)
    dy = output - Y
    return dot(dy, X).T, dot(dy, ones((X.shape[0],1)))

def finite_diff(CostFunction, X, Y, W0b0, row, col, h):
    '''
    Function for calculating one component of gradient using finite-difference
    approximation.
    h is the "small step" to take for the finite-difference
    row, col is the coordinate which that we wish to take the finite-differnce at
    '''
    W0b0_h = np.copy(W0b0)
    W0b0_h[row,col] = W0b0_h[row,col] + h
    return (CostFunction(X, Y, W0b0_h) - CostFunction(X, Y, W0b0))/h

def Check_diff(X, Y, W0b0, row, col ,h, gradW0, gradb0):
    '''
    This function prints the difference between gradient calculated using the
    finite difference method and vectorized gradient function
    '''
    finiteDiff = finite_diff(CostFunction, X, Y, W0b0, row, col ,h)
    if row == 784:
        print('The difference on Gradient_of_b0[%i, 0] is %010.10f' %(col, \
            abs(finiteDiff - gradb0[col,0])))
    else:
        print('The difference on Gradient_of_W0[%i,%i] is %010.10f' %(row, \
            col, abs(finiteDiff - gradW0[row,col])))

def part3b():
    '''
    Main function for part3b, in order to check the accuracy of the vectorized
    gradient function by comparing to finite method
```

```
    7 Random points for each of the weights and the bias are selected from a
    normal distribution of scale 0.0001.
    By setting row = 784, we are checking the gradients of the bias
    '''
    np.random.seed(1)
    W0 = np.random.normal(scale = 0.0001, size = (784,10))
    b0 = np.random.normal(scale = 0.0001, size = (10,1))
    W0b0 = np.vstack((W0, b0.T))
    row_ = np.random.randint(0,784,7)
    #row = 784 # This indicates that we are checking for the gradient for b
    col_ = np.random.randint(0,10,7)
    h = 10**(-7)
    gradW0, gradb0 = Gradient(X_train, Y_train, W0b0)
    for i in range(7):
        Check_diff(X_train, Y_train, W0b0, row_[i], col_[i] ,h, \
            gradW0, gradb0)
        #Check_diff(X_train, Y_train, W0b0, row, col_[i] ,h, \
        #    gradW0, gradb0) #This indicates that we are checking for the gradint for b
```

The function `finite_diff(CostFunction, X, Y, W0b0, row, col, h)`, `Check_diff(X, Y, W0b0, row, col ,h, gradW0, gradb0)` and `part3b()` are then used to verify the accuracy by comparing the results from vectorized gradient and finite-method.
The Gradients of the Weights are checked:

```
>>> part3b()
The difference on Gradient_of_W0[575,0] is 0.0001926810
The difference on Gradient_of_W0[165,3] is 0.0001806322
The difference on Gradient_of_W0[218,7] is 0.0000338211
The difference on Gradient_of_W0[627,2] is 0.0006195966
The difference on Gradient_of_W0[38,8] is 0.0005302735
The difference on Gradient_of_W0[378,3] is 0.0001368954
The difference on Gradient_of_W0[457,8] is 0.0002568140
```

The Gradients of the bias unit are checked:

```
>>> part3b()
The difference on Gradient_of_b0[1, 0] is 0.0003257430
The difference on Gradient_of_b0[5, 0] is 0.0006381659
The difference on Gradient_of_b0[3, 0] is 0.0007544559
The difference on Gradient_of_b0[6, 0] is 0.0002933402
The difference on Gradient_of_b0[5, 0] is 0.0006381659
The difference on Gradient_of_b0[9, 0] is 0.0004366353
The difference on Gradient_of_b0[0, 0] is 0.0006255623
```

It is evident that these differences are smaller than 10^{-3}. Therefore it is verified that the vectorized gradient function is accurate.

Part 4

Training Neural Network without Momentum

Choosing parameters

The initial weights of the neural network is set to random variables in a scaled standard normal distribution with scale 0.0001. The initial weights cannot be all zeros, nor should it be too far away from zero. Therefore, a scaled standard normal distribution is chosen.
The maximum iteration is set to be 1500. This is an arbitrary choice, but it is proven under 1500 iteration, the neural network has converged.(i.e. the performance has reached its plateau.)
The learning rate α is chosen to be 1×10^{-5}. We chose this number because any alpha below or equal to 1×10^{-4} makes the cost function diverge, i.e. the cost "blows up". Moreover, as in figure 2, the learning rate of 1×10^{-5} is shown to be the one with lowest cost, i.e. it converges to minimum the fastest. Please note here is maximum number of iteration is 100 which is different from the maximum iteration of the training, 1500 because it is only use to justify the choice of α.

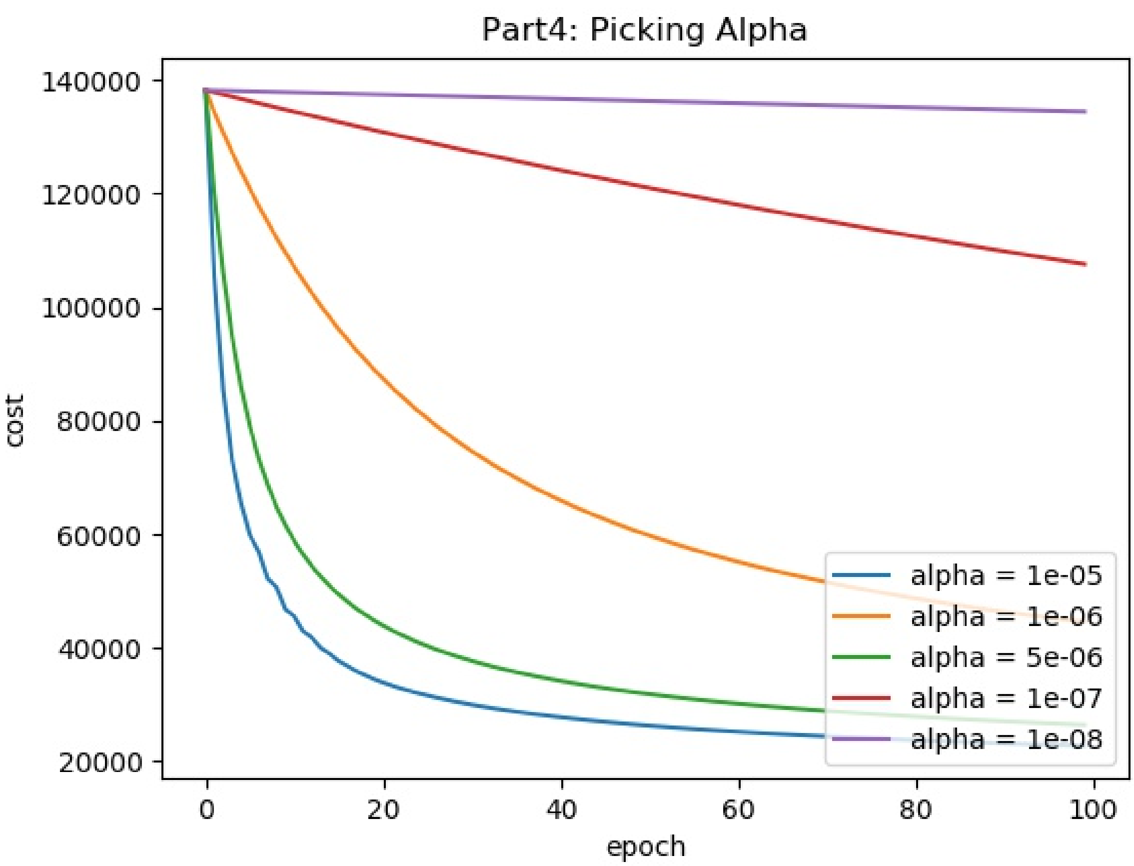


Figure 2: Cost function with different learning rate

Performance of neural network

The learning curve of the training and testing set is shown in figure 3. The neural network "learns" in about 200 iterations And both of the training and testing set has a performance about 90%

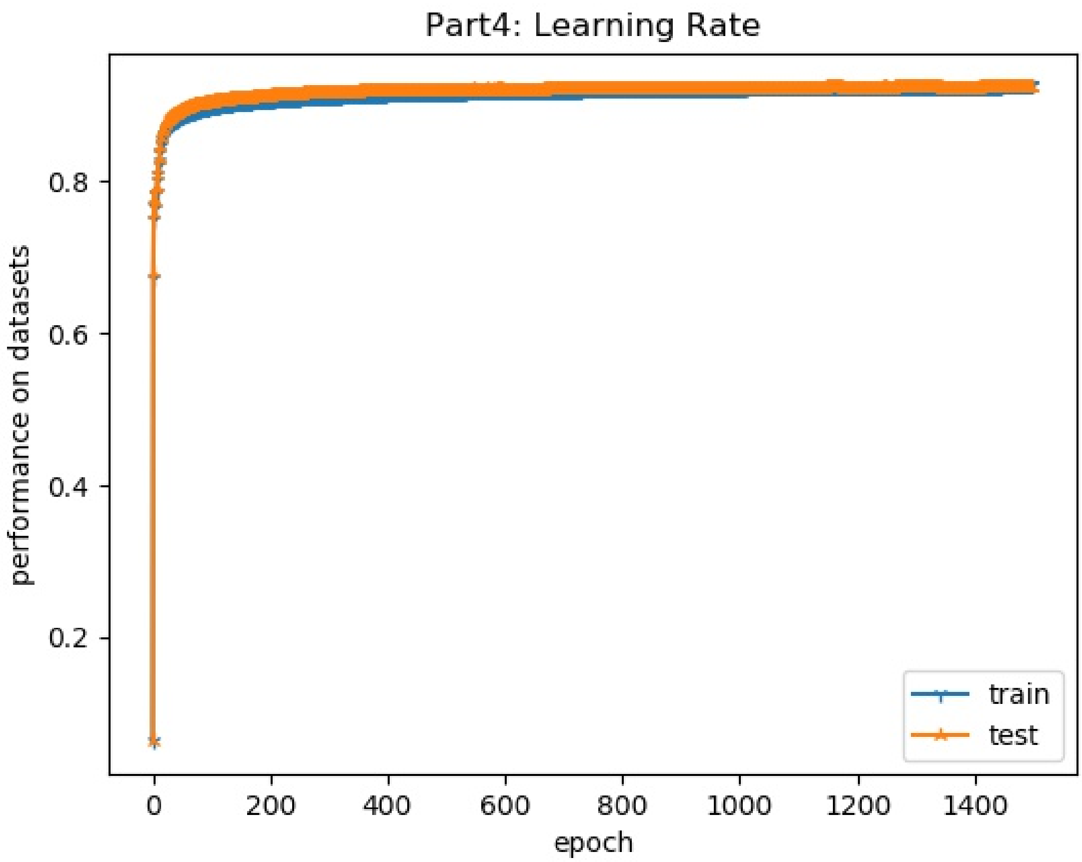
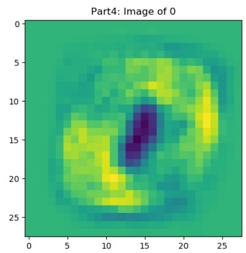


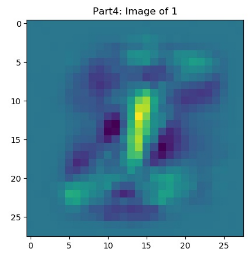
Figure 3: Learning curve of the neural network

Visualize Weights

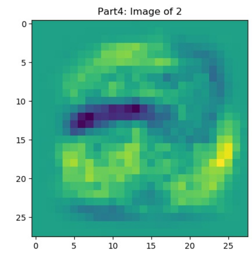
The weights going into each of the output units (i.e. digits) are shown below in Figure 4.



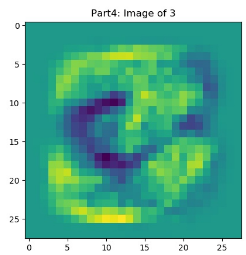
(a) Weights of Output Neuron 0



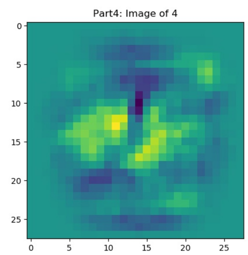
(b) Weights of Output Neuron 1



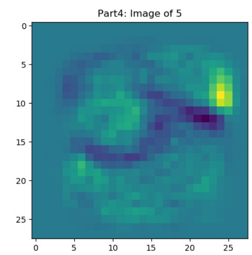
(c) Weights of Output Neuron 2



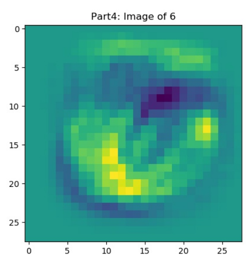
(d) Weights of Output Neuron 3



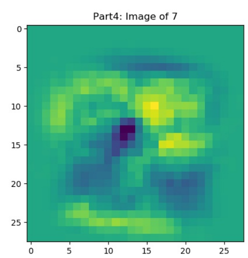
(e) Weights of Output Neuron 4



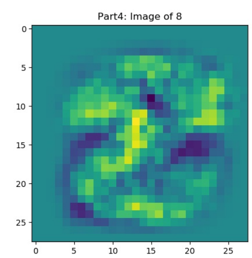
(f) Weights of Output Neuron 5



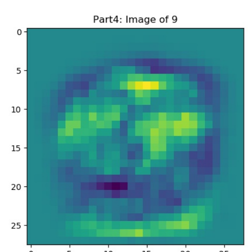
(g) Weights of Output Neuron 6



(h) Weights of Output Neuron 7



(i) Weights of Output Neuron 8



(j) Weights of Output Neuron 9

Figure 4: Visualization of output units

Part 5

Training Neural Network with Momentum

In this section, a momentum of 0.9 is used. The parameter update rule for momentum is:

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

$$W \leftarrow W - v$$

Where γ is the momentum term, α is the learning rate and W is the weights.
The code implementation is below. Note that line 23 and 24 is the implementation of momentum. The function `CostFunction()` and `Gradient` are the same as the two functions in *Part 3b)* of the same name.

```
1  def grad_descent_part5(f, df, x, y, init_t, alpha, EPS=1e-5, max_iter=80000, \
2   plotLR = False, gamma = 0.9):
3       '''
4       This function is same as the grad_descent function in part 4, except we update
5       the weights with momentum term gamma.
6       '''
7       prev_t = init_t - 10 * EPS
8       t = init_t.copy()
9       iter = 0
10      v = 0
11      cost_func_ = []
12      performanceTrain = []
13      performanceTest = []
14      while norm(t - prev_t) > EPS and iter < max_iter:
15           costFunc = f(x, y, t)
16           gradW0, gradb0 = df(x, y, t)
17           grad = np.vstack((gradW0,gradb0.T))
18
19           cost_func_.append(costFunc)
20           performanceTrain.append(performance(x, y, t))
21           performanceTest.append(performance(X_test,Y_test, t))
22           prev_t = t.copy()
23           v = gamma*v + alpha*grad
24           t -= v
25           if iter % 100 == 0:
26                print "Iter", iter
27                print "Cost", costFunc
28                print "Gradient: ", grad, "\n"
29           iter += 1
30
31      if plotLR:
32           fig = plt.figure(40)
33           plt.title("Part5: Learning Rate")
34           plt.xlabel('epoch')
35           plt.ylabel('performance on datasets')
36           plt.plot(range(iter),performanceTrain,'-1', label = 'train')
37           plt.plot(range(iter),performanceTest,'-2', label = 'test')
38           plt.legend(loc='lower right')
39           fig.savefig(dirpath + '/part5_LearningRate.jpg')
40           plt.show()
```

```
    return t, cost_func_, performanceTrain, performanceTest

def part5():
    '''
    Train the neural networking using gradient desecent.
    The initial weights is set to be a scaled standard normal with scale 0.0001
    The alpha is set to be 1e-5
    Maximum iteration is set to be 1500
    '''
    alpha = 1e-5
    np.random.seed(1)
    W0 = np.random.normal(scale = 0.0001, size = (784,10))
    b0 = np.random.normal(scale = 0.0001, size = (10,1))
    W0b0 = np.vstack((W0, b0.T))
    W0b0_part5, cost_func_part5, performanceTrain_part5, performanceTest_part5 = \
        grad_descent_part5(CostFunction, Gradient, X_train, Y_train, W0b0, alpha, \
        plotLR = True, max_iter = 1500)

    return W0b0_part5, cost_func_part5,     performanceTrain_part5, performanceTest_part5
```

The initialization of weights and setting of learning rate are the same as in part 4. The learning curve of training using momentum is shown below in Figure 5. To compare it with the previous training in *Part 4* without momentum, a plot with smaller epoch is presented in Figure 6. From the plots, it is evident that training with momentum trains the neural network faster.

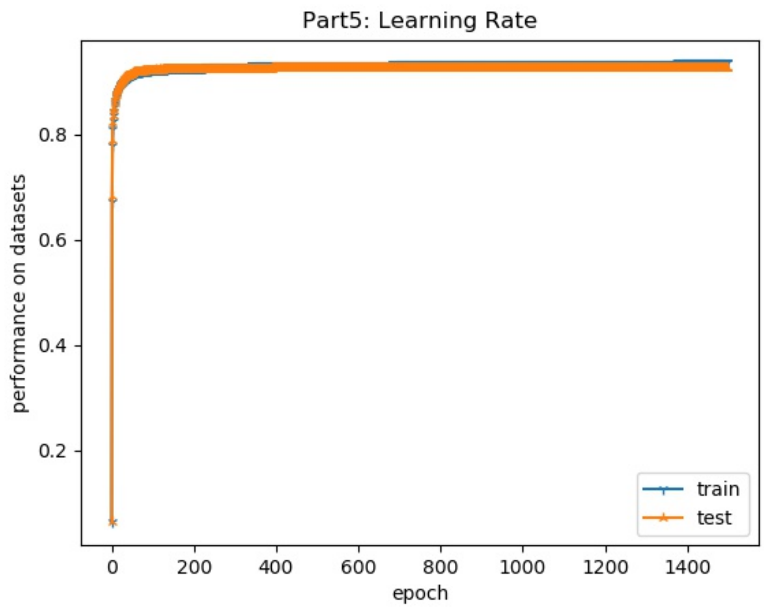


Figure 5: Learning curve of the neural network, with momentum

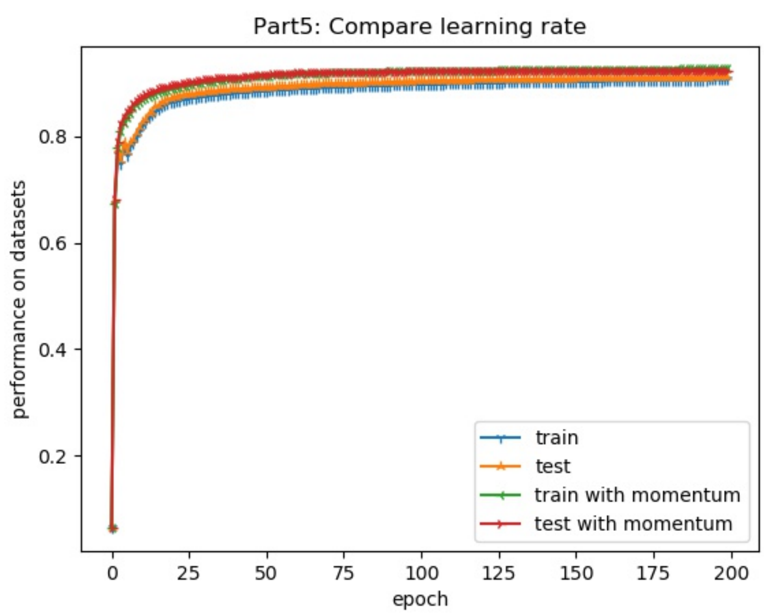


Figure 6: Comparison of learning curves between with and without momentum

Part 6

Part 6a) Contour Plot of the Cost Function

To visualize cost function with different weights w_1 and w_2, a contour plot is produced in Figure 7. The two weights chosen are the 249^{th} and 265^{th} pixel of the 9^{th} neuron, i.e. the digit 8. A range from -2 to 2 is chosen for both of the weights.

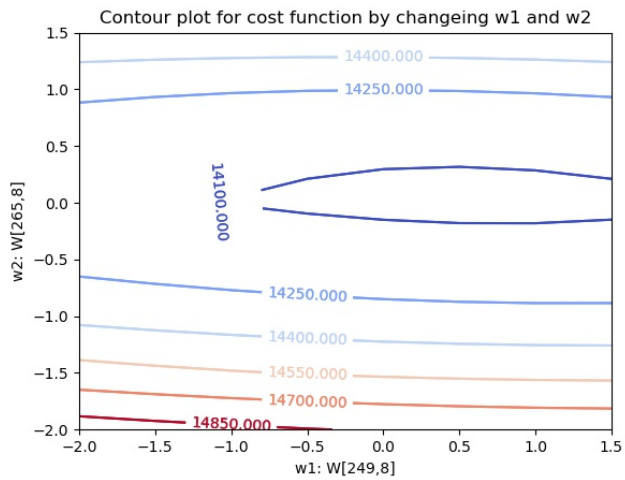


Figure 7: Contour Plot of the Cost Function

Part 6b), c) Training using momentum

To prove the use of momentum in some cases gives a better result. One specific example is given below in Figure 8. We use the 300^{th} and 407^{th} pixel of the third neuron (i.e. digit 2). The initialization of w_1 is -2 and w_2 is 0. The learning rate for training without momentum is 3.8×10^{-3} and the training with momentum is 4×10^{-4}. And the number of iteration is 10.

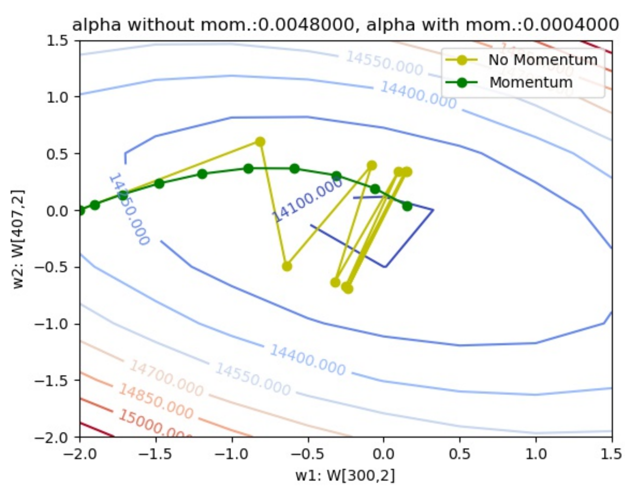


Figure 8: Comparing training with and without momentum

Part 6d) Difference between two trajectories

The trajectory without momentum is very "bouncy", it has trouble going in the direction that can minimize the cost. On the other hand, the trajectory with momentum is following the path that can minimize the cost better. The trajectory is smoother.
This is because the cost function contour in this situation is a flat ellipse and the learning rate is big. It means when we start at a point closer to the major axis, in this case point (-2,0), the direction of no momentum training trajectory will keep changing because every movement changes it to a point with almost opposite gradient. On the other hand, the momentum training trajectory follows a smoother path because in each iteration, it moves further towards the minimum because of the "momentum".

Part 6e) When momentum fails to work

Previously, in *Part 6c)* and *Part 6d)*, it is highlighted that a flat ellipse cost function contour with a initialization closer to the major axis which gives a big gradient change can demonstrate the effectiveness of momentum.
However, momentum can fail to work under certain situations such in Figure 9 below. In this case, the initialization of w_1 is -0.5 and w_2 is -2. The learning rate for training without momentum is 2×10^{-3} and the training with momentum is 2×10^{-4}. And the number of iteration is 10. We see that the no momentum trajectory goes into the minimum cost perfectly, whereas the momentum trajectory overshoots the minimum. This is because the initialization is along the minor access of the ellipse, thus the gradient is changes less, and points to the minimum more directly.

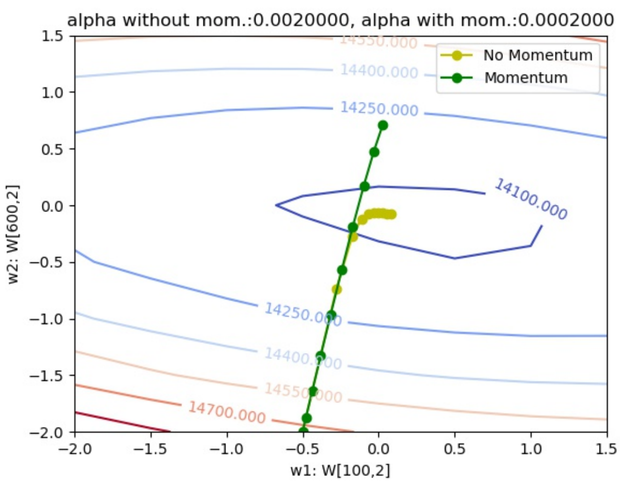


Figure 9: Momentum Overshoot

Moreover, it is important to note that we deliberately avoid choosing pixels that are on the edges of the images because such pixels are usually dark and do not have any feature associated with them. As a result, the gradients on these pixels are small and there will not be any trajectories because the weights will not change at all.

Part 7

Say we have a neural network with N layers each of which contains K neurons.
First, we define our symbols:

- $\mathbf{W_i}$: K×K matrix that represents the weights between the neurons on i-th layer and the neurons on (i+1)-th layer.
- $\mathbf{b_i}$: K × 1 vector that represents the bias of the i-th layer.
- g_i: the activation function used in neurons on i-th layer.
- $\mathbf{h_i}$: K × 1 vector that represents the neurons on i-th layer.
- $\mathbf{h_{i+1}} = g_{i+1}(\mathbf{W_i^T h_i} + \mathbf{b_i})$

So we have:

- the gradient between (i+1)-th layer and i-th layer: $\frac{\partial \mathbf{h_{i+1}}}{\partial \mathbf{h_i}} = \mathbf{W_i} g_{i+1}^{'}(\mathbf{W_i^T h_i} + \mathbf{b_i})$
- the gradient between a layer and its weight matrix: $\frac{\partial \mathbf{h_i}}{\partial \mathbf{W_i}} = \mathbf{h_i} g_{i+1}^{'}(\mathbf{W_i^T h_i} + \mathbf{b_i})$
- the gradient between a layer and its bias vector: $\frac{\partial \mathbf{h_i}}{\partial \mathbf{b_i}} = g_{i+1}^{'}(\mathbf{W_i^T h_i} + \mathbf{b_i})$

Let's assume the complexity of element-wise operations(e.g. addition and multiplication) takes $O(1)$, so we have the complexity of matrix-wise operations:

- matrix addition of two a×b matrix is $O(ab)$
- matrix multiplication of a a×b matrix and a b×c matrix is $O(abc)$
- element-wise operations on a a×b matrix is $O(ab)$

Start to evaluate the complexity of calculating gradients:

- the gradient between a layer and its bias vector: $\frac{\partial \mathbf{h_i}}{\partial \mathbf{b_i}} = g_{i+1}^{'}(\mathbf{W_i^T h_i} + \mathbf{b_i})$
  It requires a matrix multiplication of a K×K matrix and a K×1 matrix, a vector addition of two K×1 matrices, and a element-wise operation with a K×K matrix. Thus, the complexity is $O(K^2) + O(K) + O(K) = O(K^2)$.
  Note that this expression appears on other gradients, so we can store it and reuse.
- the gradient between (i+1)-th layer and i-th layer, $\frac{\partial \mathbf{h_{i+1}}}{\partial \mathbf{h_i}} = \mathbf{W_i} g_{i+1}^{'}(\mathbf{W_i^T h_i} + \mathbf{b_i})$
  It requires a matrix multiplication of a K×K matrix and a K×1 matrix. Thus the complexity is $O(K^2)$. (Note that $g_{i+1}^{'}(\mathbf{W_i^T h_i} + \mathbf{b_i})$ is already calculated.)
- the gradient between a layer and its weight matrix: $\frac{\partial \mathbf{h_i}}{\partial \mathbf{W_i}} = \mathbf{h_i} g_{i+1}^{'}(\mathbf{W_i^T h_i} + \mathbf{b_i})$
  It requires a matrix multiplication of a K×1 matrix and a 1×K matrix. Thus the complexity is $O(K^2)$. (Note that $g_{i+1}^{'}(\mathbf{W_i^T h_i} + \mathbf{b_i})$ is already calculated.)

In back propagation, assuming we stores all the intermediate values, we only need to compute: $\frac{\partial \mathbf{h_{i+1}}}{\partial \mathbf{b_i}}$ and $\frac{\partial \mathbf{h_i}}{\partial \mathbf{W_i}}$ N-1 times for N-1 connections between N layers, and $\frac{\partial \mathbf{h_{i+1}}}{\partial \mathbf{h_i}}$ N-2 times between N layers except for the input layer. Only consider one gradient calculation:

$$(N-1)(O(K^2) + O(K^2)) + (N-2)O(K^2) = O(NK^2)$$

If compute each gradient individually without store intermediates steps, the cost of calculating gradients becomes:

$$\sum_{i=1}^{N} \frac{\partial J}{\partial \mathbf{W_i}}$$

For the sake of convenience, ignore the bias term and activation function:

$$
\begin{aligned}
\frac{\partial J}{\partial W_{i,j,k}} &= \frac{\partial J}{\partial h_{i+1,j}} {h_{i,j}}' \\
&= \sum_{j_{i+2}}^{K} \frac{\partial J}{\partial h_{i+2,j_{i+2}}} W_{i+1,j_{i+1},j_{i+2}} h_{i,j}^{'} \\
&= \sum_{j_{i+2}}^{K} \sum_{j_{i+3}}^{K} \frac{\partial J}{\partial h_{i+3,j_{i+3}}} W_{i+2,j_{i+2},J_{i+3}} W_{i+1,j_{i+1},j_2} h_{i,j}^{'} \\
&\text{......} \\
&= \sum_{j_{i+2}}^{K} \sum_{j_{i+3}}^{K} ... \sum_{j_N}^{K} \frac{\partial J}{\partial h_{N,j_N}} W_{i+2,j_{i+2},j_{N-1}} ... W_{i+1,j_{i+1},j_{i+2}} h_{i,j}^{'}
\end{aligned}
$$

Assume all step takes $O(1)$, $\frac{\partial J}{\partial W_{i,j,k}}$ is $O(K^{N-i-1})$. Thus, the total complexity is:

$$
\begin{aligned}
KO(\frac{\partial J}{\partial h_{N,j_N}}) + K^2 O(\frac{\partial J}{\partial W_{N-1,i,j}}) + K^2 O(\frac{\partial J}{\partial W_{N-2,i,j}}) + ... + K^2 O(\frac{\partial J}{\partial W_{1,i,j}}) \\
= O(K) + O(K^2) + O(K^2) + O(K^3) + ... + O(K^N) = O(K^N)
\end{aligned}
$$

So backpropagation has complexity of $O(NK^2)$ while non-vectorized, non-storing calculation of gradient has complexity of O(K^N)

Part 8

Training Faces Data using PyTorch

For *Part 8*, *Part 9* and *Part 10*, the data set used is same as the ones used in *Assignment 1*. The images used is a subset from the FaceScrub website, which is available at:
http://vintage.winklerbros.net/facescrub.html

Description of the System

The architecture of the neural network is the following. There are 1024 input neurons because we re-process the images to 32×32 pixel grey-scale. The first and only hidden layer has 12 hidden units. The activation function on this hidden layer is *tanh*. There are 6 output neurons, because we have 6 labels. There is no activation function on the output layer, only linear function is used. Finally, a softmax function is applied to the outputs. The optimizer function used is `torch.optim.Adam()`. The loss function used is cross-entropy.
The input data is a randomly selected mini-batch of the whole training set after removing inappropriate images. The size of the batch is 10 images. Again, the inputs are 32×32 pixel grey-scaled images. Therefore the input is a $n \times 1024$ matrix, with flattened images, not counting the bias unit.
The initial weights of the neural network is random variables in a standard normal with a scale of 0.01 (i.e. mean 0, standard deviation 0.01). The initial weights cannot be all zeros, nor should it be too far away from zero. Therefore, a scaled standard normal distribution is chosen.
The learning rate α is 3×10^{-4} and the maximum number of epoch is 1500.
The learning curves is shown in 10 below. Note that the training set performance is approaching to 100% and the validation and testing set is about 85%. The final classfication performance on the test set is 86.67%

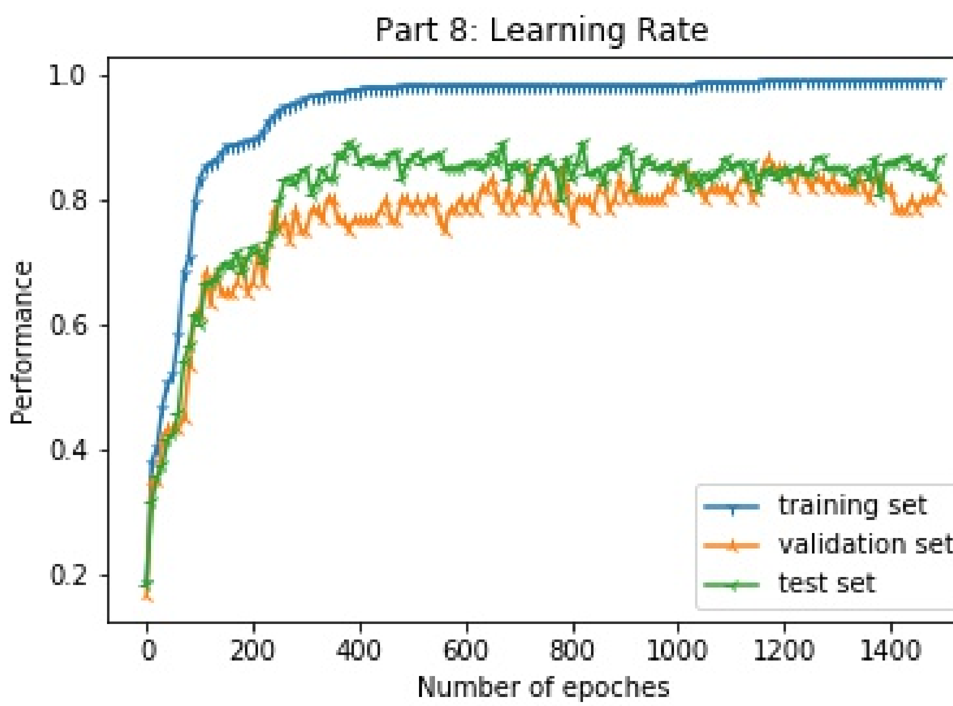
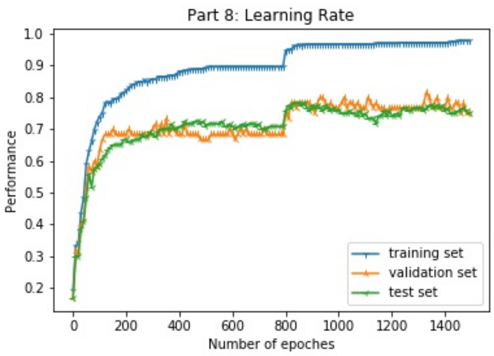


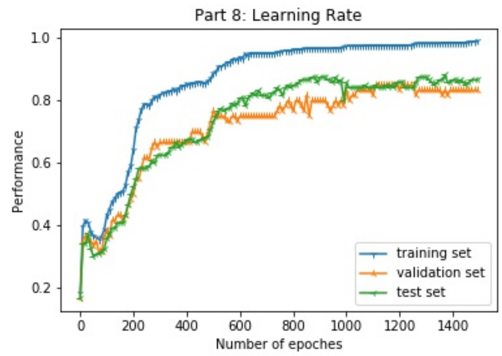
Figure 10: Learning curve of the neural network

Choice of parameter

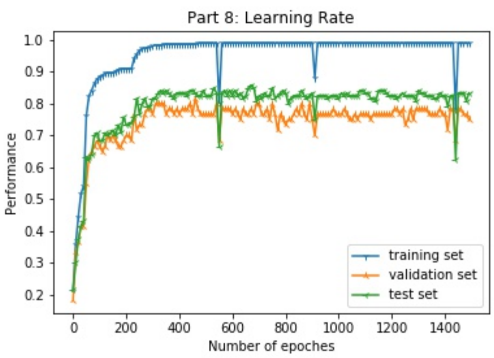
Figures of learning rate using different parameter is shown in Figure 11 on the next page.
The activation function used was *tanh* because *ReLU* gives a bad result as shown in Figure 11a under all other parameters kept same.
The learning rate α is choose to be 3×10^{-4} because it gives the best result compared to $alpha = 1 \times 10^{-3}$ in Figure 11b (this suggests that α could be larger) and $alpha = 5 \times 10^{-3}$ in Figure 11c (this suggests that α might be too large because we see unstable behaviour).
The size of mini-batch is 10 because with a mini-batch of 20 in Figure 11d, the learning curve is the same. Also, we see that with a mini-batch of size 10, the training set performance is reaching 100%. But smaller mini-batch size is unreasonable because we have six distinct labels.
The number of hidden neuron is chosen to be 12 because a 30 hidden neuron produces a similar performance, as shown in Figure 11e. Increasing number of neuron increases the training time. Therefore we chose 12 hidden neurons to compromise between computation time and learning performance.
The input photo is chosen to be 32×32 pixel because the input of 6464 pixel gives slightly worse result, under all other parameter kept the same as the original setting, as shown in Figure 11f.
For the optimizer function, we have tried different optimizer functions, but they give similar performance.
Finally, note that all other parameters, maximum epoch, the standard deviation of normal distribution initialization are educated choices. It is proven that these parameters gives a good performance, therefore no further tuning on these parameters.



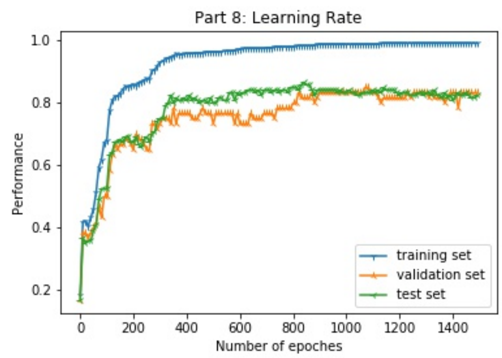
(a) Using ReLU as activation function



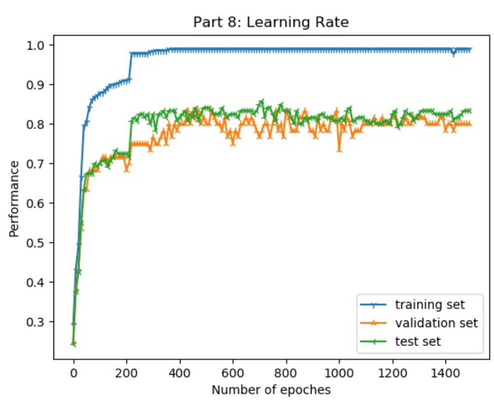
(b) Learning rate = 0.0001



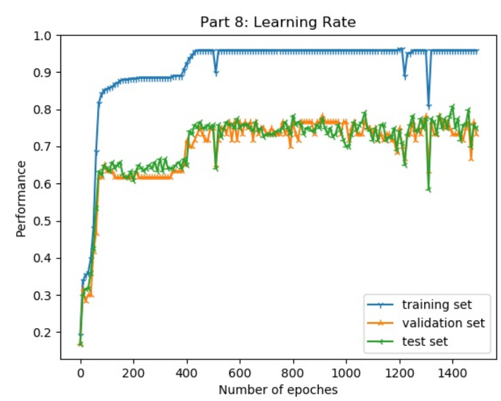
(c) Learning rate = 0.0005



(d) Mini-batch size 20



(e) Hidden Neuron 30



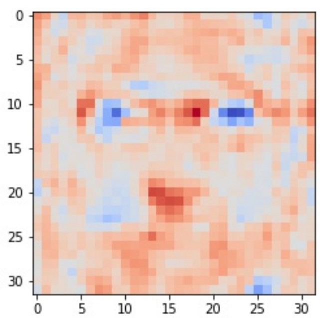
(f) Input photo 64×64 pixel

Figure 11: Learning curves with different parameters

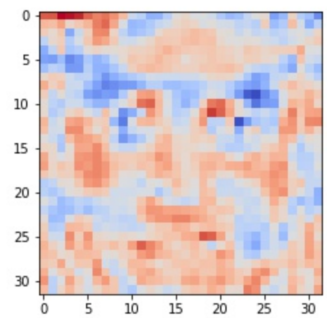
Part 9

Visualize Weights for useful hidden units

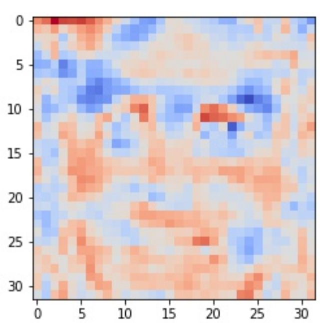
For this part, we visualize the weights of hidden neurons that are useful for classifying Lorraine Bracco and Peri Gilpin. In order to accomplish this, we pick the four largest positive weights from hidden layer that is going into each of the two output neuron. After finding these positive weights and its associated neuron in the hidden layer, we then visualize them by plotting the weights of these hidden neurons.
Figure 12 shows the four most useful neurons for classifying Lorraine Bracco, which are: Neuron 3, 7, 10 and 11.
Figure 13 on next page shows the four most useful neurons for classifying Peri Gilpin, which are: Neuron 1, 2, 3 and 4.



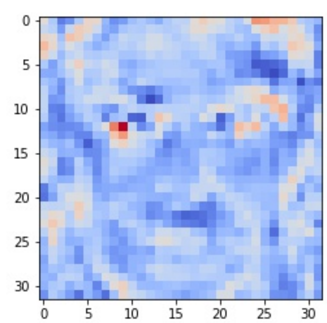
(a) Hidden Neuron 3



(b) Hidden neuron 7

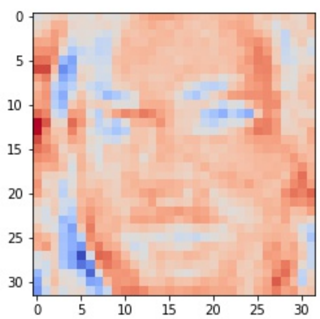


(c) Hidden neuron 10

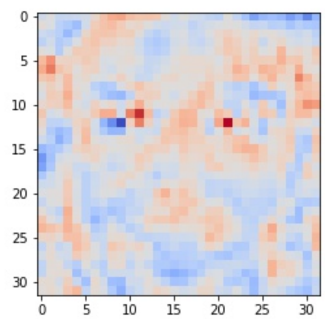


(d) Hidden neuron 11

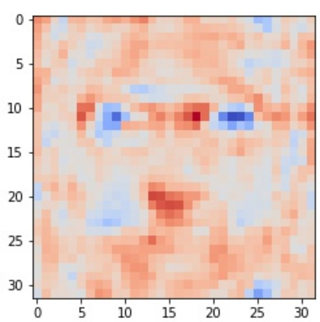
Figure 12: Neurons useful for classifying Lorraine Bracco



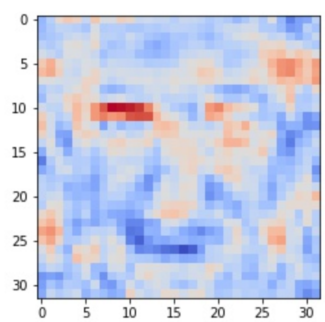
(a) Hidden Neuron 1



(b) Hidden neuron 2



(c) Hidden neuron 3



(d) Hidden neuron 4

Figure 13: Neurons useful for classifying Peri Gilpin

Part 10

Training using AlexNet

In this part, the value of activation at the last layer of the AlexNet is extracted and used as features to perform face classification. These modifications are performed inside the `class MyAlexNet(nn.Module)`, and it is included below.
In particular, in line 10-14, we initialize the weights of the classifier to a random variables of a normal distribution with mean 0 and standard deviation 0.01.
In line 33-37, we add an additional fully-connected layer, which the activation function is just linear. This layer, which takes the activation values from the last layer of the AlexNet, is going to be our neural network that gets trained.

```
1  class MyAlexNet(nn.Module):
2      def load_weights(self):
3          an_builtin = torchvision.models.alexnet(pretrained=True)
4  
5          features_weight_i = [0, 3, 6, 8, 10]
6          for i in features_weight_i:
7              self.features[i].weight = an_builtin.features[i].weight
8              self.features[i].bias = an_builtin.features[i].bias
9  
10         # Initilize weights
11         classifier_weight_i = [0]
12         for i in classifier_weight_i:
13             self.classifier[i].weight.data.normal_(0.0,0.01)
14             self.classifier[i].bias.data.normal_(0.0,0.01)
15 
16     def __init__(self, num_classes=1000):
17         super(MyAlexNet, self).__init__()
18         self.features = nn.Sequential(
19             nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
20             nn.ReLU(inplace=True),
21             nn.MaxPool2d(kernel_size=3, stride=2),
22             nn.Conv2d(64, 192, kernel_size=5, padding=2),
23             nn.ReLU(inplace=True),
24             nn.MaxPool2d(kernel_size=3, stride=2),
25             nn.Conv2d(192, 384, kernel_size=3, padding=1),
26             nn.ReLU(inplace=True),
27             nn.Conv2d(384, 256, kernel_size=3, padding=1),
28             nn.ReLU(inplace=True),
29             nn.Conv2d(256, 256, kernel_size=3, padding=1),
30             nn.ReLU(inplace=True),
31             nn.MaxPool2d(kernel_size=3, stride=2),
32         )
33         # Addition of the extra layer, the linear function
34         self.classifier = nn.Sequential(
35             nn.Linear(256 * 6 * 6, 6),
36             nn.Softmax()
37         )
38 
39         self.load_weights()
40 
```

```
    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        x = self.classifier(x)
        return x

    # Exrtact the activation on the last layer
    def process_X(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        return x.data.numpy()
```

The extraction of the activation value of the last layer of the AlexNet is done through the function `get_data()`.

```
def get_data(S, act, model, grayed = False, s = 0):
    act_rawdata = {}
    act_data = {}
    for i in act:
        act_rawdata[i] = np.empty((0,9216))
        act_data[i] = [0, 0, 0]

    for act_type in ['actors', 'actresses']:
        if grayed:
            dataset_path = dirpath + "/facescrub_%s_%i_grayed/" %(act_type, S[0])
        else:
            dataset_path = dirpath + "/facescrub_%s_%i/" %(act_type, S[0])
        for root, dirs, files in os.walk(dataset_path):
            dirs.sort()
            files.sort()
            for filename in files:

                # extracting activation data of AlexNet
                im = imread(dataset_path + filename)[:,:,:3]
                im = im - np.mean(im.flatten())
                im = im/np.max(np.abs(im.flatten()))
                im = np.rollaxis(im, -1).astype(np.float32)
                im = Variable(torch.from_numpy(im).unsqueeze_(0), requires_grad=False)
                im = model.process_X(im)
                for i in act:
                    if i in filename:
                        act_rawdata[i] = np.vstack((act_rawdata[i], im))

    # randomly shuffle act_rawdata
    np.random.seed(s)
    for i in act:
        act_rawdata[i] = act_rawdata[i][np.random.permutation(act_rawdata[i].shape[0]),:]

    for i in act:
        act_data[i][0] = act_rawdata[i][:min(70, act_rawdata[i].shape[1] - 30),:]
        act_data[i][1] = act_rawdata[i][-30:-20,:]
        act_data[i][2] = act_rawdata[i][-20:,:]
    return act_data
```

It is important to note that we are only training the weights for output of AlexNet, i.e. the activation values. To make this point clear, we labeled these values as `self.classifier(x)`, as in line 44 of the first code snippet. And we train it using the code below by only calling `model.classifier(x)`:

```
for Epoch in range(nEpoch):
    for iMinibatch, Minibatch in enumerate(dataloader):
        x = Variable(Minibatch[:,:-1], requires_grad = False).type(dtype_float)
        y = Variable(Minibatch[:,-1], requires_grad = False).type(dtype_long)
        y_pred = model.classifier(x)
        loss = loss_fn(y_pred, y)
        model.classifier.zero_grad()  # Zero out the previous gradient computation
        loss.backward()    # Compute the gradient
        optimizer.step()   # Use the gradient information to make a step
```

Description of the system

As previously mentioned, this neural network takes the activation values of AlexNet on the last layer. The inputs to the AlexNet is $227 \times 227 \times 3$ pixel RGB image. The activation values of AlexNet output has dimension $256 \times 6 \times 6 = 9126$. These values are used to train a fully connected neural network without hidden layers. A linear function is used to compute values for each of 6 output neurons, because we have 6 labels, each for one actor/actress. A softmax function is applied on top of the output neurons to reach the final values. (See line 34-37 of the first code snippet).
The optimizer used is `torch.optim.Adam`. The loss function used is cross-entropy.
The learning rate used is 3×10^{-4}, and the batch-size is 10, which are the same as those parameters in $Part8$ because we wish to compare the performance. The maximum number of iteration is 600, which is sufficient for this neural network to reach its best performance.
The weights are initialized to a random variables of a normal distribution with mean 0 and standard deviation 0.01 because we try to avoid 0 weights but not having weights too far away from 0.

Performance of the System

The performance is shown below in the learning curves in Figure 14. Compare to the performance of the previous neural network without using AlexNet in Figure 10, both the validation and test set performance has increased. The validation accuracy is high as 100% and test accuracy is around 95%, compared to the around 85% accuracy previously.

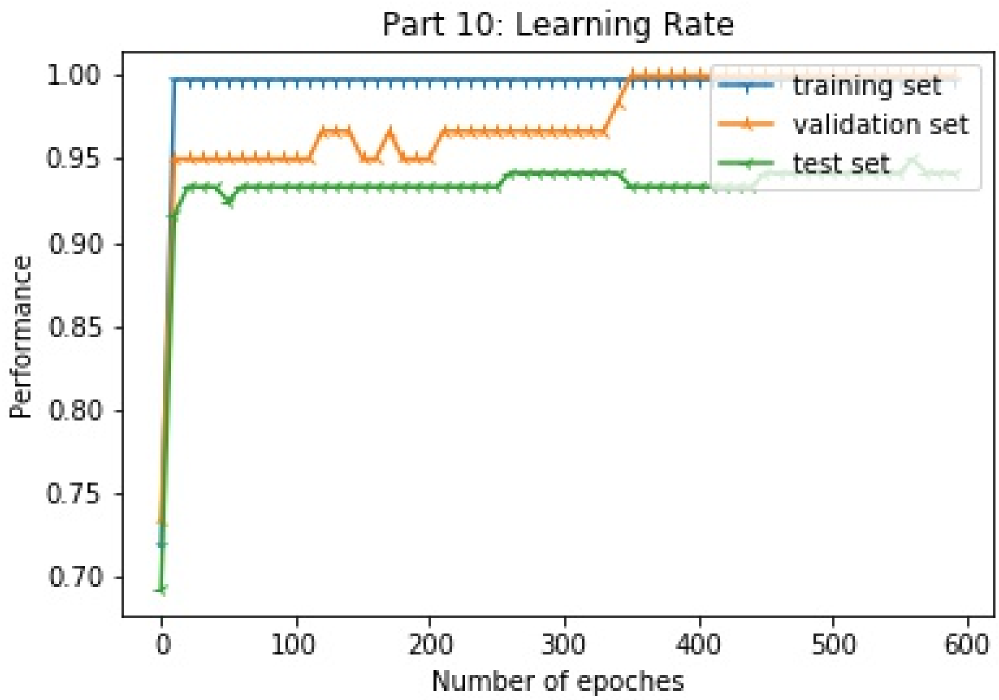


Figure 14: Learning curve of the neural network with AlexNet