CSC2515: Assignment 2 - Deep NN

Due on Friday, February 23, 2018

M.Wong, S.Pyda

February 23, 2018

Problem 0

Introductory information and readme instructions

This submissions containts the required files: *digits.py, faces.py, deepfaces.py, deepnn.tex & faces.pdf* in addition to two folders - both containing an previously downloaded pictures for each actor/actress. The downloading code can be found in the source files. Comments about the code are located in each of the source files. Code was written in Python 3.5 with the Anaconda environment - the packages used are outlined at the beginning of the file. To load the training data for Parts 1-6, you must put the mnist_all.mat file into the same directory as the .py files.

Problem 1

Dataset description

In parts 1 through , the MNIST dataset of handwritten digits will be used: *mnist_all.mat*. The data is loaded in the form of a python dictionary. It is already split into training and testing sets - with 60 000 digits in the training and 10 000 in testing, and is also split by digit which can be accessed with *train[digit]*. The number of of each digit in test/train is approximately equal although there are variations between digits. The digit data is in the form of a 784 integer array, representing the flattened pixel intensities of 28 by 28 pixel images.

10 examples of each digit are shown below:

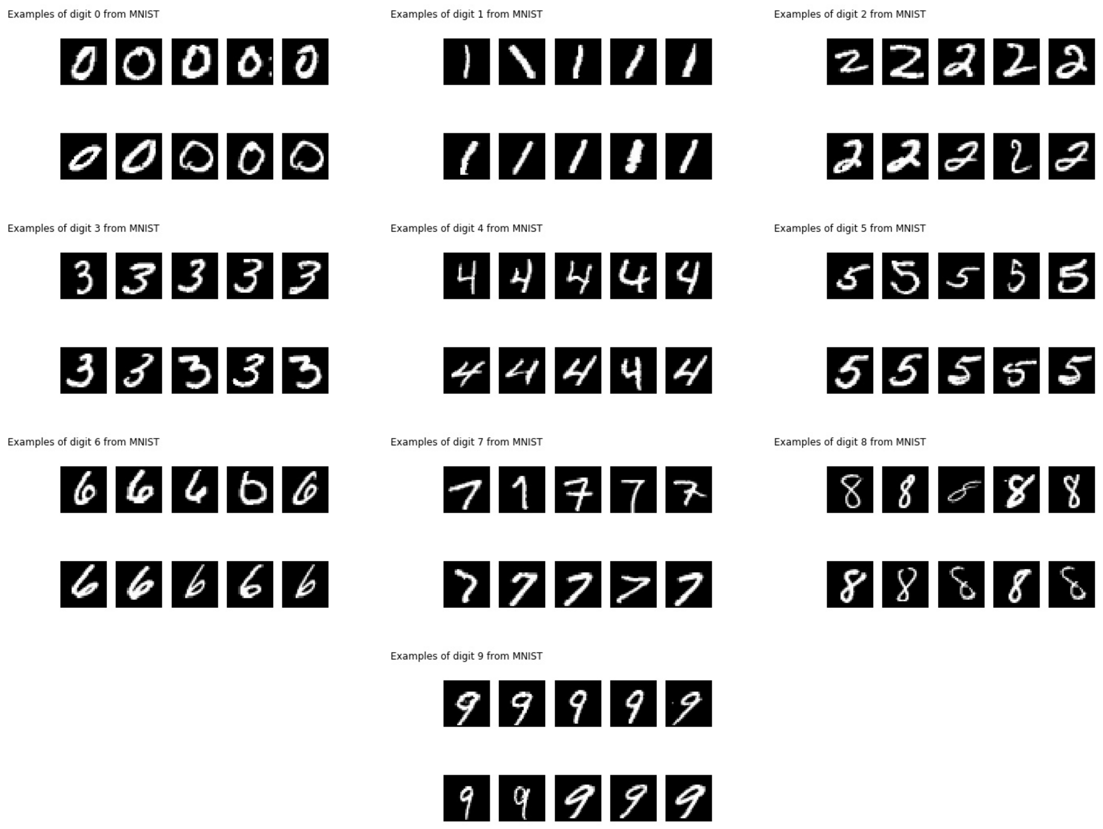
Examples of digit 0 from MNIST
Examples of digit 1 from MNIST
Examples of digit 2 from MNIST
Examples of digit 3 from MNIST
Examples of digit 4 from MNIST
Examples of digit 5 from MNIST
Examples of digit 6 from MNIST
Examples of digit 7 from MNIST
Examples of digit 8 from MNIST
Examples of digit 9 from MNIST

Figure 1: Digits from MNIST

Problem 2

Computation of the provided Network.

The network provided in Part 2 is shown below:

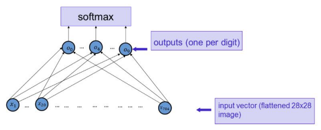

Figure 2: Neural Network to Implement

The source code to compute the assigned network is provided below.

```
def compute(X, W, b):
   hypothesis = np.matmul(W.T, X)
   hypothesis = hypothesis + b
   return hypothesis
```

The source code running a test case of the assigned network is provided below, followed by the output. M is the MNIST dataset provided, W and b are randomly initialized numpy variables.

```
def testPart2():
   # Load Data
   M = loadData()
   train = loadTrain(M)
   # Testing computation function
   # Initialize weights and bias to random normal variables
   W = np.random.normal(0.0, 0.1, (784, 10))
   b = np.random.normal(0, 0.1, (10, 1))
   hypothesis = compute(train, W, b)
   print(softmax(hypothesis))
```

Output (after applying softmax) from the test case:

```
[[ 0.02681158 0.0532326 0.04155301 ..., 0.02401518 0.05461363
  0.09298066]
 [ 0.02934882 0.02945978 0.00866719 ..., 0.02812303 0.05160524
  0.06640991]
 [ 0.07800929 0.14095711 0.05324094 ..., 0.08344358 0.071218
  0.03403779]
 ...,
 [ 0.06811845 0.09068319 0.11408343 ..., 0.09293973 0.23242635
  0.24750088]
 [ 0.42558937 0.28527524 0.36294473 ..., 0.32821263 0.12923604
  0.11150445]
 [ 0.04023265 0.03443131 0.05201543 ..., 0.0454216 0.06806685
  0.08509415]]
```

Problem 3

Gradient derivation and vectorized format

(A) For the network described above, the cost function used will be the sum of negative log-probabilities. This cost function, summed over training examples is:
$C = -\sum_j y_j log p_j$
Where y is the actual target and p is the predicted value.
The output p_i is the calculated the softmax function to o_i,:
$p_i = \frac{e^{o_i}}{\sum_j e^{o_j}}, \quad o_i = \sum_j x_j w_{ij}$
We use the chain rule to find $\frac{dC}{dw_{ij}}$:
$\frac{dC}{dw_{ij}} = \frac{dC}{dp_i}\frac{dp_i}{do_i}\frac{do_i}{dw_{ij}}$
$\frac{dp_i}{do_i} = \frac{e^{o_i}}{\sum_j e^{o_i}} - \left(\frac{e^{o_i}}{\sum_j e^{o_i}}\right)^2 = p_i(1-p_i)$
$\frac{do_i}{dw_{ij}} = x_j$
$\frac{dC}{do_i} = \sum_j \frac{dC}{dp_j}\frac{dp_j}{do_i}$

(B) The source code for computing the gradient is shown below.

```
def grad_NLL_W(y, o, layer):
    p = softmax(o)
    grad = p - y
    grad = np.matmul(grad, np.transpose(layer))
    return np.transpose(grad)


# Computes the gradient of negative log-loss function for biases only
def grad_NLL_b(y, o):
    p = softmax(o)
    grad = p - y
    grad = np.sum(grad, axis=1, keepdims=True)
    return grad
```

The source for our test cases, along with the outputs, are also shown below.

```
def testPart3():
    # Test gradient functionality
    y = np.zeros((10, 1))
    y[1, :] = 1

    # Create a test matrix
    M = loadData()
    test0 = ((M["train1"][130].T) / 255.0).reshape((784, 1))
    W = np.random.normal(0, 0.2, (784, 10))
    b = np.random.normal(0, 0.2, (10, 1))
    # print(np.where(test0 != 0))

    # Create a finite difference
    h = 0.00001

    # Weight testing
    finite_W = np.zeros((784, 10))
    finite_W[542, 0] = h
    finite_d = (NLL(y, compute(test0, W + finite_W, b)) - NLL(y, compute(test0, W,
        b))) / (h)
    print("Cost for row 542, column 0: " + str(finite_d))
    gradient = grad_NLL_W(y, compute(test0, W, b), test0)
    print("Gradient for row 542, column 0: " + str(gradient[542, 0]))

    # Bias testing
    finite_b = np.zeros((10, 1))
    finite_b[1, :] = h
    finite_d = (NLL(y, compute(test0, W, b + finite_b)) - NLL(y, compute(test0, W,
        b))) / (h)
    print("Cost for second element in bias: " + str(finite_d))
    gradient = grad_NLL_b(y, compute(test0, W, b))
    print("Gradient matrix: " + str(gradient))

    # Reinitialize test variables for another test
    finite_W = np.zeros((784, 10))
    finite_b = np.zeros((10, 1))
    y = np.zeros((10, 1))
    test1 = ((M["train9"][130].T) / 255.0).reshape((784, 1))
    y[9, :] = 1

    # Weight testing
    finite_W[300, 4] = h
    finite_d = (NLL(y, compute(test1, W + finite_W, b)) - NLL(y, compute(test1, W,
        b))) / (h)
    print("Cost for row 300, column 4: " + str(finite_d))
    gradient = grad_NLL_W(y, compute(test1, W, b), test1)
    print("Gradient for row 300, column 4: " + str(gradient[300, 4]))

    # Bias testing
    finite_b[4, :] = h
    finite_d = (NLL(y, compute(test1, W, b + finite_b)) - NLL(y, compute(test1, W,
        b))) / (h)
    print("Cost for fifth element in bias: " + str(finite_d))
    gradient = grad_NLL_b(y, compute(test1, W, b))
    print("Gradient matrix: " + str(gradient))
```

```
Cost for row 542, column 0: 0.0424078520744
Gradient for row 542, column 0: 0.0424076964593
Cost for second element in bias: -0.996437311773
Gradient matrix: [[ 0.05461597]
 [-0.99643733]
 [ 0.03592502]
 [ 0.32774037]
 [ 0.04444944]
 [ 0.05179153]
 [ 0.08930539]
 [ 0.14572631]
 [ 0.19441422]
 [ 0.05246907]]
```

```
Cost for row 300, column 4: 0.000106382902487
Gradient for row 300, column 4: 0.000106382357955
Cost for fifth element in bias: 0.00010680176743
Gradient matrix: [[ 3.36017608e-03]
 [ 1.57850465e-02]
 [ 1.07051677e-02]
 [ 1.06504072e-01]
 [ 1.06801186e-04]
 [ 6.75277789e-02]
 [ 9.15366897e-02]
 [ 8.16717644e-02]
 [ 6.16811862e-01]
 [ -9.94009358e-01]]
```

Problem 4

Gradient Descent Training

For the Vanilla Gradient Descent algorithm, the weights and biases were sampled from the normal distribution around a mean of 0 and standard deviation of 0.2, using *numpy.random.normal*. The weights and biases were initialized with shapes (784, 10) and (10, 1) respectively.

The optimization procedure for gradient descent involved testing: the type of distribution used in weight and bias initializiation, the learning rate, and the number of iterations. The reported results proved to be optimial. The network converges relatively quickly at 500 iterations - although itwas tested up to approximately 5000 iterations, however this had little accuracy on validation performance. The learning curve is shown below.

By trial and error, a learning rate of 0.00001 was selected. The learning rate was chosen to optimize convergence time - a low learning rate resulted in slow convergence time, and a high learning rate resulted in fluctuations and hindered convergence.

The log loss is plotted against the number of iterations.

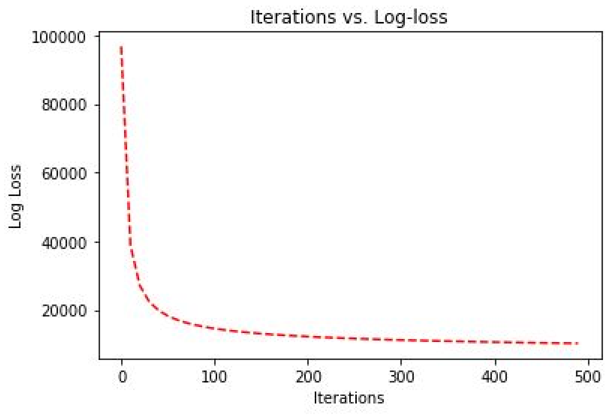

Figure 3: Contour Plot of Cost Function demonstrating effective Momentum

Problem 5

Gradient Descent With Momentum

Momentum is used in gradient descent to dampen oscillation and include an directional accumulation term to increase speed. The parameter update rule for momentum, as provided in lecture notes is:

$$\nu \leftarrow \gamma\nu + \alpha\frac{dC}{dW}\ W \leftarrow W - \nu$$

Where γ is the momentum term and α is the learning rate. C is the cost, and W is the parameter (weight) being updated. The momentum rate was set to 0.5 based on some initial tests, however the number could be optimized further for better/faster performance. This momentum rate was sufficient to demonstrate its effect. The function for vectorized momentum is shown below.

```
\# MOMENTUM GRADIENT DESCENT

def grad_descent_w(NLL, grad_NLL_W, grad_NLL_b, x, y, init_w, init_b, alpha,
    iterations):
   iters = list()
   costy = list()

   EPS = 1e-5
   prev_w = init_w - 10 * EPS
   prev_b = init_b - 10 * EPS
   w = init_w.copy()
   b = init_b.copy()
   max_iter = iterations
   iter = 0

   v = np.zeros((784, 10))
   v[:, :] = 0.000001 # Initialize
   momentum_rate = 0.5

   while np.linalg.norm(w - prev_w) > EPS and iter < max_iter and np.linalg.norm(b -
       prev_b) > EPS:
      prev_w = w.copy()
      prev_b = b.copy()
      grad_desc = alpha * (grad_NLL_W(y, compute(x, w, b), x))
      mv = np.multiply(momentum_rate, v)
      v = np.add(mv, grad_desc)
      w -= v
      b -= alpha * (grad_NLL_b(y, compute(x, w, b)))
      if iter % 10 == 0:
         print("Iteration: " + str(iter))
         print("Cost: " + str(NLL(y, compute(x, w, b))))
         cost_i = NLL(y, compute(x, w, b))
         iters.append(iter)
         costy.append(cost_i)
      iter += 1
```

The figure below shows the learning curve on train data for gradient descent with and without momentum.

It can be observed that momentum increases convergence speed.

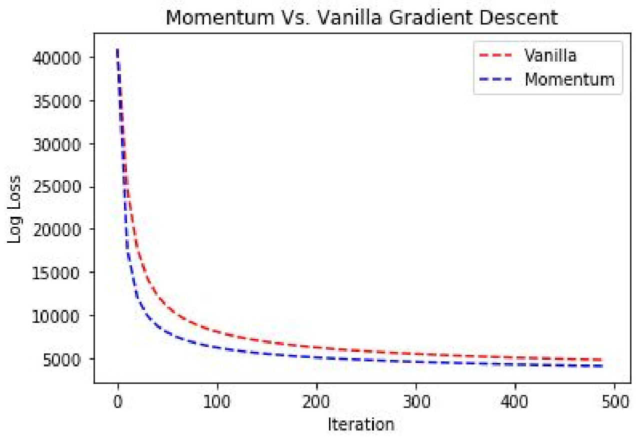

Figure 4: Comparison of Gradient Descent with and without Momentum

Problem 6

Contours of Gradient Descent

In this section, the effectiveness of gradient descent moving towards a minima will be demonstrated on contour plots. The weights chosen were at $init\_w[490, 3]$ and $init\_w[490, 3]$. These weights were chosen because by checking the corresponding values on the digits, to ensure that they were not all 0s would not have an effect on the output. When the weights were allowed to vary around the vales obtained in Part 5, without gradient descent, they did not move towards the local minima. The weights were moved $\pm$ 0.15 from their trained values. The learning rate was increased to 0.017, and 20 iterations were run.

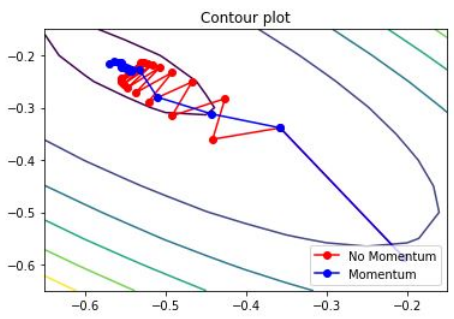

Figure 5: Contour Plot of Cost Function demonstrating effective Momentum

As shown in the figure momentum results in a faster path towards the minimum with fewer oscillations. The momentum term in the update introduces memory and allows the learning process to accelerate in consistent directions.

An example of momentum not being effective, would be as stated above, where the weights correspond to pixels of 0 or close to 0. In the example below, the weights chosen were changed to at $init\_w[10, 3]$ and $init\_w[11, 3]$, which may have been closer to the sides. It can be seen from the contours that they are more linear and less effected by the weights. Momentum could be less effective, or very similar to vanilla gradient descent, when there is already a 'smooth' path towards the minima.

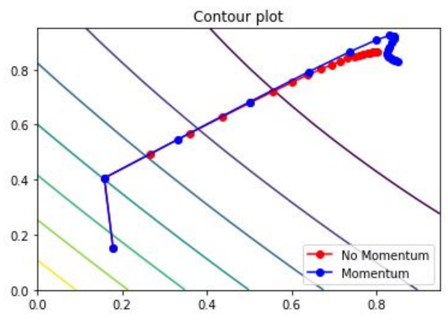

Figure 6: Contour Plot of Ineffective momentum

Problem 7

For a network of N layers and K neurons, if we implement a fully vectorized backpropagation algorithm for gradient descent, the computational complexity is limited by only matrix multiplications and additions - and therefore, the number of neurons, K becomes less relevant, and the complexity depends more so on the number of layers. Matrix multiplication is approximately $O(n^2)$, and therefore, the complexity of fully vectorized backpropagation would be also $O(n^2)$ (which would make sense in this case - e.g. the weights' computational cost is linear). Without caching the weights, each neuron, in each layer, would need to propagate the gradient to its connected neuron in the preceeding layer. Without the ability to simply look up the weights associated at each neuron and furthermore, without the linearized computational cost in the weights, the computational time complexity would scale with both K and N, and therefore, the computational time complexity would be approximately vary between $O(n^3$ and $O(n^4)$. Indeed, this is the case with some other neural network optimization techniques (such as Newton's method).

Problem 8

Training a single-hidden-layer fully connected network

The provided source was modified to train a single-hidden layer, fully-connected neural network to classify actors and actresses from Project 1, assuming the original list of actors and actresses were the following:

```
act =['Lorraine Bracco', 'Peri Gilpin', 'Angie Harmon', 'Alec Baldwin', 'Bill Hader',
    'Steve Carell']
```

The neural network architecture involved a fully connected layer (Linear), followed by ReLU activation (activation), followed by another fully connected layer (Linear) with input dimensions of 32*32*3 (e.g. 32 $\times$ 32 $\times$ 3 representing the pixel areas and colour channels), 20 hidden neurons, and finally a output layer consisting of 6 output units. The Cross Entropy cost was minimized as the cost function. Furthermore, experimentation involving the learning rate, α, number of iterations, as well as batch size was conducted, leading to an optimal learning rate of 0.0001, 5000 iterations, and a batch size of approximately 200 (out of a total 450 training examples). After training, the accuracy for the training set, validation set and test set were 87.3%, 84.9% and 79.2% respectively. A plot of the training & validation accuracies throughout the training are shown below.

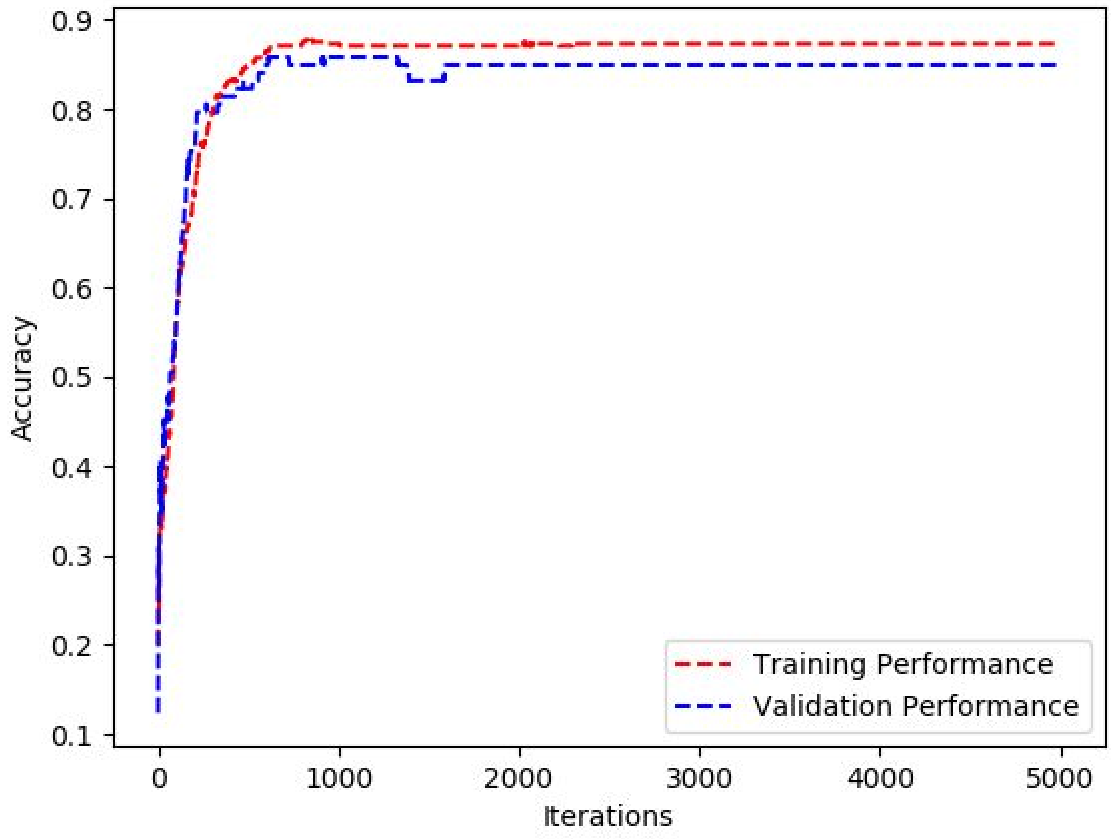

Figure 7: Training & Validation performance for Part 8

Problem 9

Visualizing weights of Actors

The figure below shows the output of the weights selected for two actors (Carell and Hader). Note that the negative and positive weights are plotted separately.

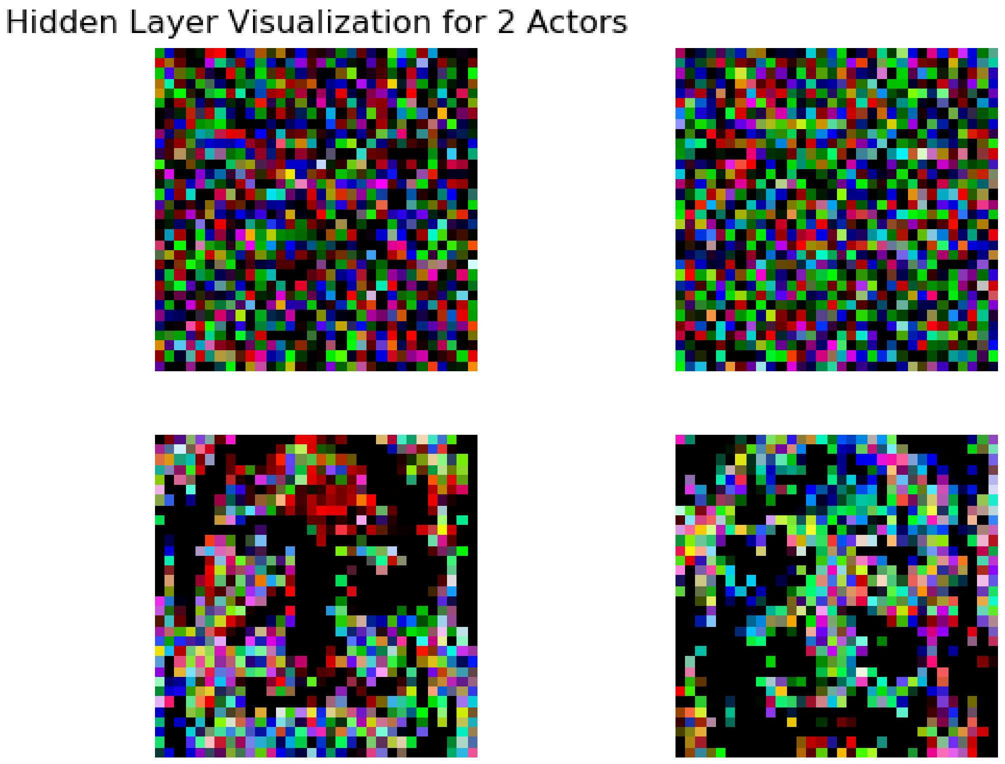

Figure 8: Visualized weights for two actors - positive on the left, negative on the right.

It appears that the images shown above can resemble faces, as outlined in the assignment specification. The weights were selected using a trial and error approach, given our neural network's architecture (e.g. 20 hidden layers involved many runs of attempting to visual weights that had actual data).

Problem 10

Using a Pre-trained AlexNet CNN

The weights from the Pre-trained AlexNet Convolution Neural Network were obtained by modifying the source code provided on the CSC2515 website, and the code is shown below.

```
class AnotherAlexNet(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

      classifier_weight_i = [1, 4]
      for i in classifier_weight_i:
         self.classifier[i].weight = an_builtin.classifier[i].weight
         self.classifier[i].bias = an_builtin.classifier[i].bias

   def __init__(self, num_classes=1000):
      super(AnotherAlexNet, self).__init__()
      self.features = nn.Sequential(
         nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
         nn.ReLU(inplace=True),
         nn.MaxPool2d(kernel_size=3, stride=2),
         nn.Conv2d(64, 192, kernel_size=5, padding=2),
         nn.ReLU(inplace=True),
         nn.MaxPool2d(kernel_size=3, stride=2),
         nn.Conv2d(192, 384, kernel_size=3, padding=1),
         nn.ReLU(inplace=True),
         nn.Conv2d(384, 256, kernel_size=3, padding=1),
         nn.ReLU(inplace=True),
         nn.Conv2d(256, 256, kernel_size=3, padding=1),
         nn.ReLU(inplace=True),
      )
      self.classifier = nn.Sequential(
         nn.Dropout(),
         nn.Linear(256 * 6 * 6, 4096),
         nn.ReLU(inplace=True),
         nn.Dropout(),
         nn.Linear(4096, 4096),
         nn.ReLU(inplace=True),
      )

      self.load_weights()

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

Based on the code snippet above, the result after feeding a $3 \times 227 \times 227$ image into the convolution neural network was a $k \times 256 \times 13 \times 13$ Pytorch Tensor, where k was the number of training examples. The neural network architecture we implemented was similar to the one of Part 8:

1. A Linear layer with shape 449×43264
2. A ReLU activation
3. Output layer of 6 units (one for each actor/actress)

The Cross-Entropy loss was selected as the cost function. The optimizer used was the Adam optimizer. We attempted modification of the optimizer, number of iterations and the number of hidden neurons in the hidden layer, but unfortunately, due to the high computational cost of running the entire training set through the AlexNet, limited trial runs were conducted. Despite this however, we were able to obtain 100% accuracy on both the training and validation sets and approximately 60% accuracy on the test set (using a 80:20 split on the training set for training and validation, and 120 images in the test set). Based on our test set, we observed that the training set was not proportional in the actors. The script we wrote could download up to 130 images for each other; and while there were 130 images for some actors, there was less than 100 for others (e.g. Gilpin). In retrospect, it would have been preferable to normalize the number of each actor in the training set, thereby potentially increasing performance on the test set (i.e. the training set distribution was not representative of the test set distribution).

We attempted to test this behaviour by modifying our script only split the dataset into training and validatio - here, we observed a 100% training accuracy and 98% validation accuracy, potentially indicating that there may have been something wrong with the original split function. A plot of the learning curve for this section (for the training set only, due to computational limitations) is shown below.

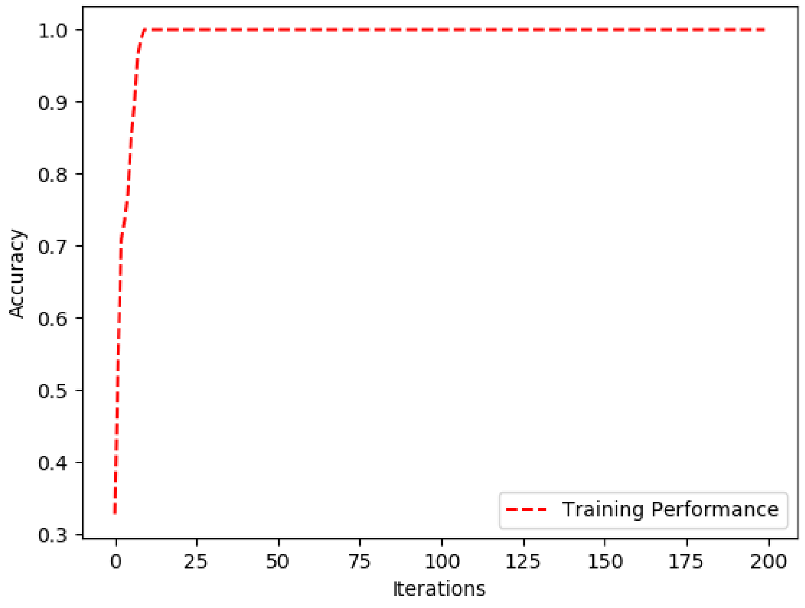

Figure 9: Learning Curve for the extracted AlexNet and the implemented neural network