CSC411: Project #2

Due on Friday, February 23, 2018

Yu-Chien Chen, Hunter Richards

Friday, February 22, 2018

Part 1

The images below served as the main dataset for the project. They were all 32 × 32 pixels, grayscale, with around 800 to 6000 images per digit. The dataset was pretty diverse with different styles of handwriting and different thickness for the strokes. It was still fairly easy to distinguish the digits since most of the them were clearly written and none of the digits were cropped out. However, there were some digits that were even hard for humans to decipher (for example row 4 column 2, row 6 column 7, and row 10 column 10).

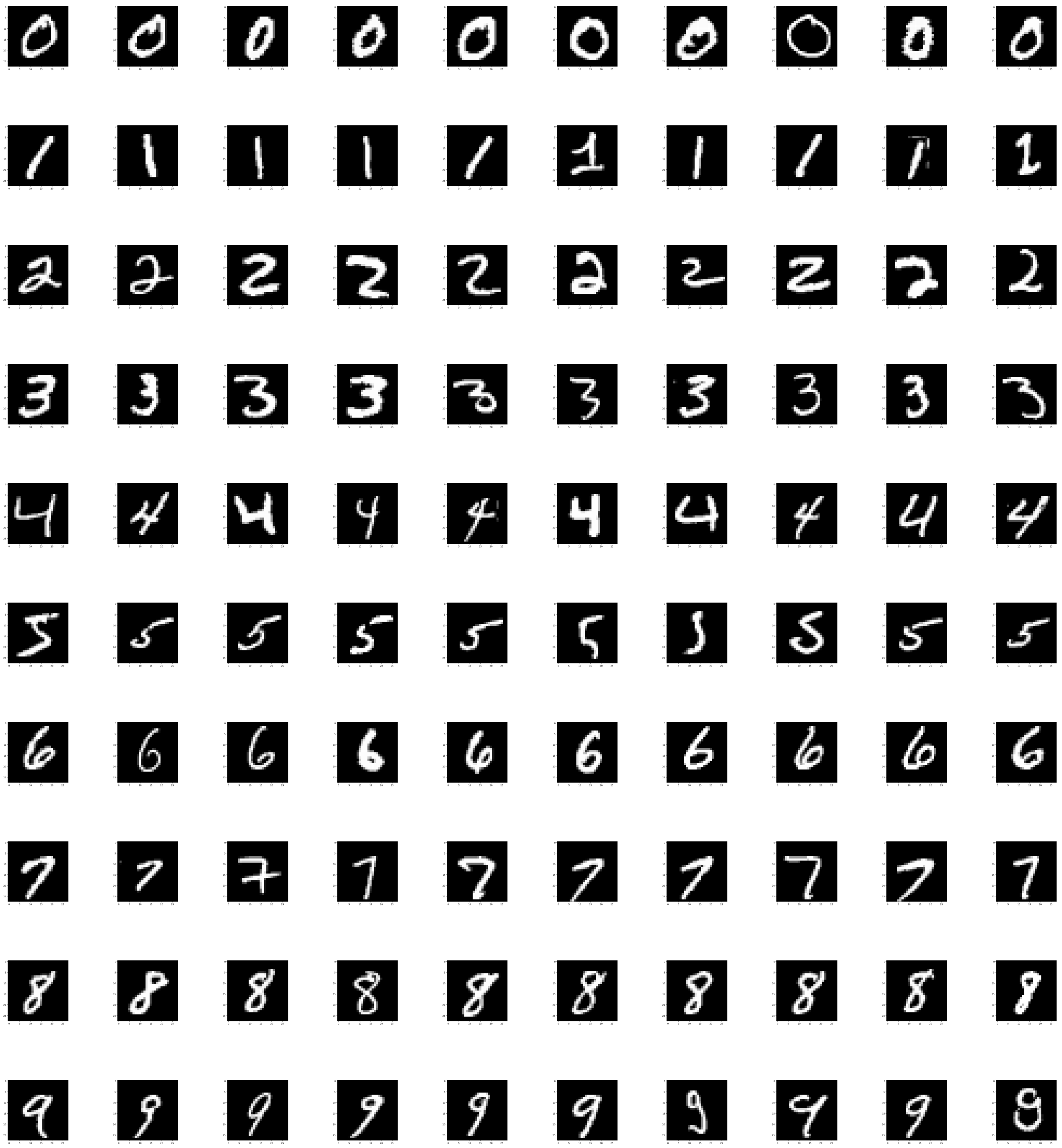


Figure 1: Snippet of the dataset of digits

Part 2

The following code was written* to compute the forward pass of this network

```
def softmax(O):
    return exp(O)/exp(O).sum(axis=1, keepdims=True)

def forward(X, W, b):
    i = np.ones((len(X), 1))
    return softmax(dot(X, W.T) + dot(i, b.T))
```

Note that the code has been implemented such that it accepts an input matrix **X** containing an entire training set, with each row representing one training example**. For future reference, let N be the number of features (pixels), M be the number of training examples, and K be the number of labels**.

The following is the mathematical representation of this program

$$\mathbf{O} = \mathbf{X}\mathbf{W}^T + \mathbf{1}\mathbf{b}^T$$

$$\mathbf{P}_{ij} = \frac{\exp(\mathbf{O}_{ij})}{\sum_k \exp(\mathbf{O}_{kj})}$$

where **W** is the weight matrix, **1** is a vector of ones, **b** is the bias vector, **O** is the output matrix, and **P** is the prediction after applying the softmax activation function. The dimensions of each term are as follows

$$\mathbf{W} \in K \times N$$

$$\mathbf{1} \in M \times 1$$

$$\mathbf{b} \in K \times 1$$

$$\mathbf{O} \in M \times K$$

$$\mathbf{P} \in M \times K$$

**All code used in this assignment is implemented in Python 3.6*

***These conventions are used for the entire report*

Part 3

a) The derivative of the cost function for a specific training example i was determined to be

$$\begin{aligned}
\frac{\partial C^{(i)}}{\partial w_{pq}} &= \frac{\partial C^{(i)}}{\partial o_{pq}} \cdot \frac{\partial o_{pq}}{\partial w_{pq}} \\
&= \frac{\partial}{\partial o_{pq}} \left(-\sum_j {y_j}^{(i)} \log {p_j}^{(i)} \right) \cdot \frac{\partial}{\partial w_{pq}} \left(\sum_j w_{ij} {x_j}^{(i)} + b_i \right) \\
&= -\sum_j \frac{\partial}{\partial o_{pq}} \Big({y_j}^{(i)} \log {p_j}^{(i)} \Big) \cdot \sum_j \frac{\partial}{\partial w_{pq}} \Big(w_{ij} {x_j}^{(i)} + b_i \Big) \\
&= \left(-\sum_j {y_j}^{(i)} \frac{1}{{p_j}^{(i)}} \frac{\partial {p_j}^{(i)}}{\partial o_{pq}} \right) \cdot {x_q}^{(i)} \\
&= \left(-{y_q}^{(i)} \Big(1 - {p_q}^{(i)} \Big) - \sum_{j \neq q} {y_j}^{(i)} \frac{1}{{p_j}^{(i)}} \Big(-{p_j}^{(i)} {p_q}^{(i)} \Big) \right) \cdot {x_q}^{(i)} \\
&= \left(-{y_q}^{(i)} + {y_q}^{(i)} {p_q}^{(i)} + \sum_{j \neq q} {y_j}^{(i)} {p_q}^{(i)} \right) \cdot {x_q}^{(i)} \\
&= \left({p_q}^{(i)} \sum_j {y_j}^{(i)} - {y_q}^{(i)} \right) \cdot {x_q}^{(i)} \\
&= \Big({p_p}^{(i)} - {y_p}^{(i)} \Big) \cdot {x_q}^{(i)}
\end{aligned}$$

Line 5 follows because

$$\frac{\partial {p_j}^{(i)}}{\partial o_{pq}} = \frac{\partial}{\partial o_{pq}} \left(\frac{e^{o_j^{(i)}}}{\sum_k e^{o_k^{(i)}}} \right) = \begin{cases} {p_q}^{(i)} \Big(1 - {p_q}^{(i)} \Big) & j = q \\ -{p_j}^{(i)} {p_q}^{(i)} & j \neq q \end{cases}$$

Converting the indices p, q back to the specified i, j and summing over all of the training examples

$$\frac{\partial C}{\partial w_{ij}} = \sum_k \Big({p_i}^{(k)} - {y_i}^{(k)} \Big) \cdot {x_j}^{(k)}$$

We can sum over all of the training examples because of the distributive property of the derivative.

b) The following vectorized code was written to compute the gradient of the cost function

```
def dC_weight(X, Y, P):
    return dot((P-Y).T, X)

def dC_bias(X, Y, P):
    i = np.ones((len(X), 1))
    return dot((P-Y).T, i)
```

The following is the mathematical representation of this program

$$\frac{\partial \mathbf{C}}{\partial \mathbf{W}} = \Big(\mathbf{P} - \mathbf{Y}\Big)^T \mathbf{X}$$

$$\frac{\partial \mathbf{C}}{\partial \mathbf{b}} = \Big(\mathbf{P} - \mathbf{Y}\Big)^T \mathbf{1}$$

Finite difference was calculated on 10 different, random points in the weight matrix and as many as possible for the bias vector (**b** is of size 10, however elements are sometimes zero and cannot be used).

The average relative error between the points calculated by the gradient function and that which was calculated by the finite difference function was 2.54×10^{-5} and 9.28×10^{-6} for the weight matrix and bias vector, respectively. These error values are very small, indicating that the vectorized gradient function is implemented correctly.

Part 4

The following learning curves were obtained from running gradient descent:

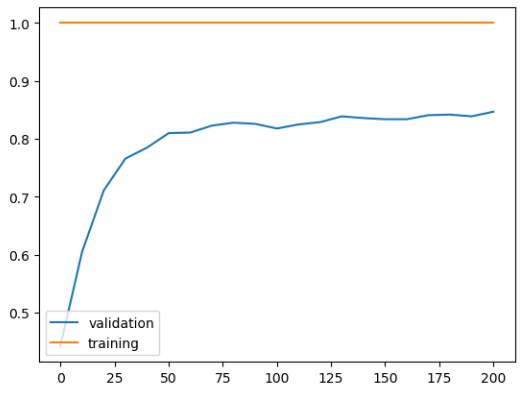


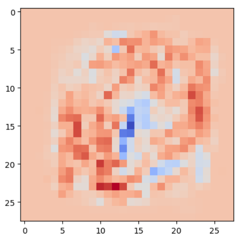
Figure 2: Number of training examples vs. Accuracy

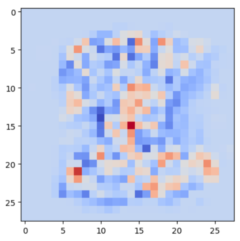
To perform gradient descent, finding the right value for the learning rate α was crucial. The procedure to find an optimal α was through trial and error. If the algorithm did not converge, then α was too large and was passing over the minimum. Smaller and smaller values of α were tested until the algorithm converged on a solution. Finally, α was then increased by very small increments to get close to the threshold where the algorithm stops converging again. It is desired to have as large an α as possible because too small an α causes the algorithm to converge unnecessarily slow.

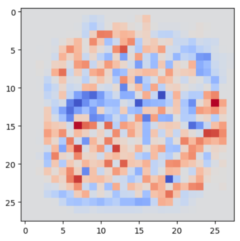
In addition to finding an appropriate learning rate, the maximum number of iterations was chosen such that it was not too small, forcing the algorithm to end too early, and not too large to cause overfitting. Also, the tolerance for when $\boldsymbol{\theta}$ is no longer increasing (the loop invariant) can not be too large or the algorithm will terminate prematurely and not too small or the algorithm will take too long and may overfit. The values for maximum iterations and tolerance are 10000 and 10^{-5} respectively, and were determined using trial and error.

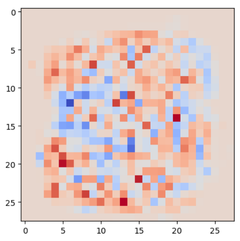
The weight matrix and bias vector were initialized as zero matrix/vector. This gave the fastest convergence because the values start out small (very small), bringing the cost closer to a minimum. Additionally, it helped with preventing the algorithm from diverging because everything amounts to zero initially.

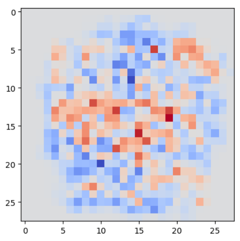
Through this procedure, the following visualization of the weights were obtained:

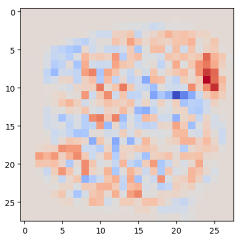

(a) 0

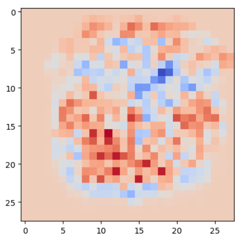

(b) 1

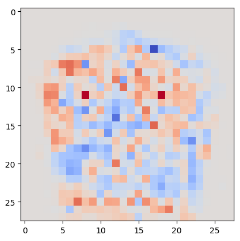

(c) 2

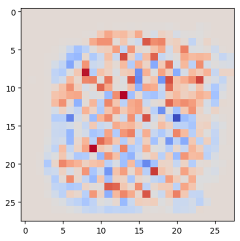

(d) 3


(e) 4


(f) 5


(g) 6


(h) 7


(i) 8

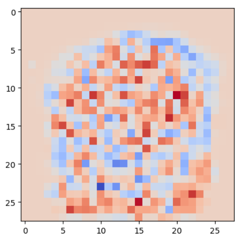

(j) 9

Figure 3: Visualization of the weights

Once the model was properly trained, the weights do not resemble numbers very much anymore. This was the same result obtained in Project 1 were as the model is trained, the weights generalize and look less and less like their corresponding face (or number, as is the case in this report).

Part 5

As expected, the learning curve for gradient descent with momentum (see figure 4) is steeper and the accuracy is slightly better than the one shown in part 4. The reason for why this is expected will be explained in part 6. The vectorized code written for gradient descent with momentum is shown below.

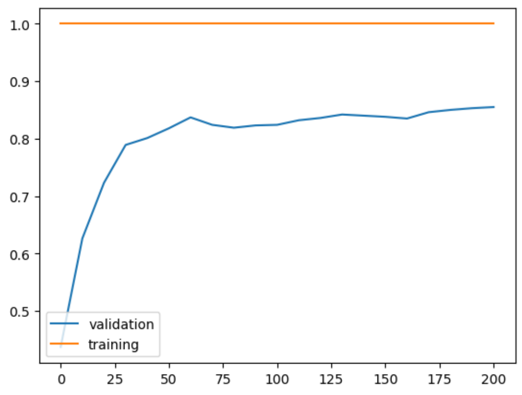


Figure 4: Number of training examples vs. accuracy

```
def part5():
    def f(n):
        n = n if n else 1
        X = genX(M, TRAIN, n)
        Y = genY(M, TRAIN, n)
        W = np.zeros((NUM_LABEL, NUM_FEAT))
        b = np.zeros((NUM_LABEL, 1))

        W, b = gradDescent(X, Y, W, b, momentum=True, out=False)
        P = classify(X, Y, W, b)
        res_train = accuracy(P, Y)

        X = genX(M, TEST, 100)
        Y = genY(M, TEST, 100)
        P = classify(X, Y, W, b)
        res_test = accuracy(P, Y)

        print('({},{})-point␣generated'.format(n, res_test))
        return res_test, res_train

    y1, y2 = list(), list()
    x = np.arange(0, 210, 10)
    for n in x:
        res = f(n)
        y1.append(res[0])
        y2.append(res[1])
    linegraphVec(y1, y2, x, 'pt5_learning_curve_accuracy')
    return
```

```
def gradDescent(X, Y, W0, b0, momentum=False, out=True):
    i = 0
    MR = 0.99
    LR = 1e-4 if momentum else 0.005
    if momentum:
        Z = W0.copy()
        v = b0.copy()
    W = W0.copy()
    b = b0.copy()
    WPrev = W0 - 10*EPS
    bPrev = b0 - 10*EPS

    while (norm(W-WPrev)>EPS or norm(b-bPrev)>EPS) and i<MAX_ITER:
        WPrev = W.copy()
        bPrev = b.copy()
        P = forward(X, W, b)
        if momentum:
            Z = MR * Z + LR * dC_weight(X, Y, P)
            v = MR * v + LR * dC_bias(X, Y, P)
            W -= Z
            b -= v
        else:
            W -= LR * dC_weight(X, Y, P)
            b -= LR * dC_bias(X, Y, P)
        if i % 500 == 0 and out:
            print('Iter', i)
            print('C(Y, P) =', C(Y, P), '\n')
        i += 1
    return W, b
```

Part 6

This part of the project demonstrates the difference between gradient descent with and without momentum. Two weights, w_1 and w_2, were picked from the trained network generated from the previous part. A contour plot was then generated by varying the values of w_1 and w_2 around the values obtained from part 5. In order to learn the difference between the two gradient descent method (with and without momentum), w_1 and w_2 were re-initialized to a value away from the local optimum (observe from the plotted contour plot). Alter the two gradient descent methods from part 5 so that each iteration only updates the two chosen weights. Record the values of w_1 and w_2 from each iteration and plot the trajectory (see Figure5).

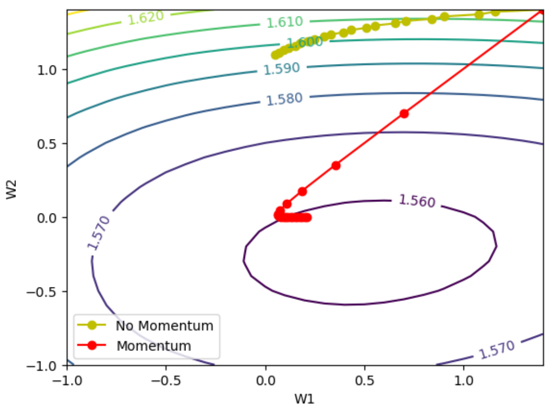


Figure 5: Contour plot with trajectories

Without a doubt, gradient descent with momentum performed better than vanilla gradient descent. It was able to find the local optimum faster and more accurately. Momentum is essentially a moving average of the gradients and it is used to update the weight of the network for each iteration. Exponentially weighed average would "de-noised" the data and create a smoother line that is closer to the real function.

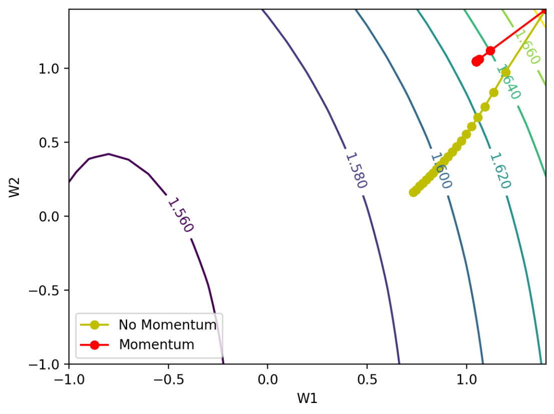
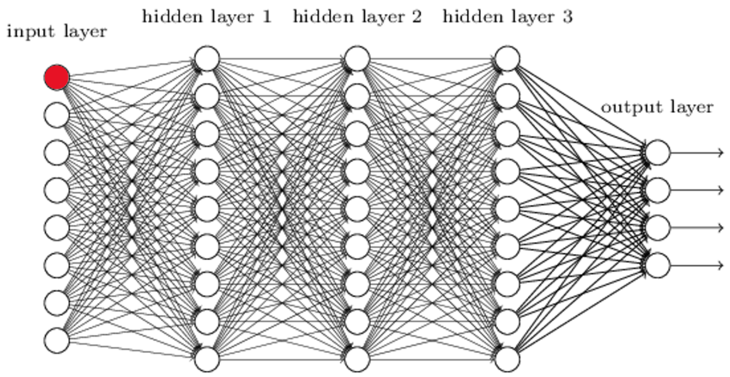


Figure 6: Demonstration for 6e

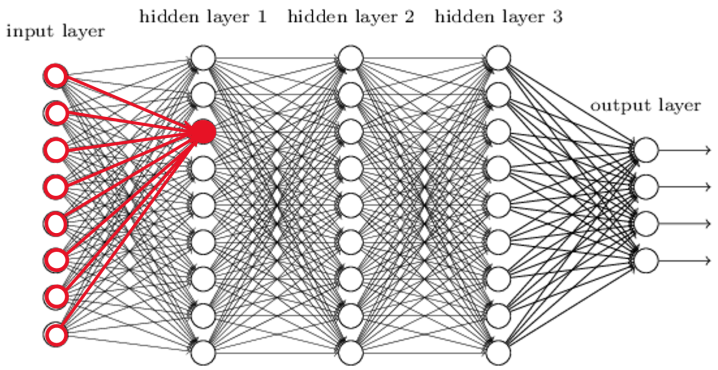
Figure6 above demonstrates that there are cases when it does not benefit when using momentum. This happens when the chosen weights are along the edges. This is because the one with the momentum oscillates too much as it tries to reach to the local optimum. As a result, it slows down the process and never gets to the local optimum.

Part 7

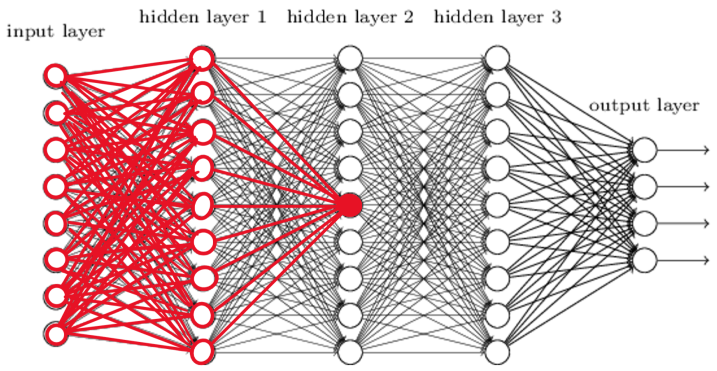
When computing the gradient with respect to each weight individually, the further into the network that one travels the more computationally intensive the process becomes. This is because the chain rule must be propagated up the network towards the input layer. The following images* demonstrate this process:



(a) Gradient for unit in the input layer



(b) Gradient for unit in hidden layer 1



(c) Gradient for unit in hidden layer 2

Figure 7: The number of derivatives computed grows exponentially with depth.

**neuralnetworksanddeeplearning.com/chap6.html*

Each of the solid circles is the unit for which the gradient is being computed. The lines demonstrate all of the derivatives that need to be computed as part of the chain rule. For the first (input) layer, only one derivative needs to be calculated. For the second layer, K derivatives need to be calculated (there are K units per layer). For the third layer, K^2 derivatives are calculated because for each unit in the second layer, K derivatives need to be calculated for each weight in the first layer. This continues until the N^{th} (output) layer where K^{N-1} calculations are made. Summing over all the layers gives

$$K^0 + K^1 + K^2 + \cdots + K^{N-2} + K^{N-1} \in O(K^N)$$

Therefore, the time complexity for computing the gradient with respect to each weight individually without caching is $O(K^N)$.

Backpropagation greatly improves on this result. With backpropagation, the gradients for each layer are captured in vectorized multiplication. Let the units in the i^{th} layer be represented by the vector $\mathbf{h}^{(i)}$. The corresponding weight matrix for this layer is $\mathbf{W}^{(i)}$. The output of i^{th} layer (ignoring the bias) can therefore be represented by the equation

$$\mathbf{h}^{(i)} = \mathbf{W}^{(i)}\mathbf{h}^{(i-1)}$$

For a network with N layers, the steps for backpropagation are

$$\begin{aligned}
\overline{\mathbf{h}}^{(N)} &= \mathbf{1} \\
\overline{\mathbf{W}}^{(N)} &= \overline{\mathbf{h}}^{(N)}\mathbf{h}^{(N-1)^T} \\
\overline{\mathbf{h}}^{(N-1)} &= \mathbf{W}^{(N-1)^T}\overline{\mathbf{h}}^{(N)} \\
\overline{\mathbf{W}}^{(N-1)} &= \overline{\mathbf{h}}^{(N-1)}\mathbf{h}^{(N-2)^T} \\
&\vdots \\
\overline{\mathbf{W}}^{(3)} &= \overline{\mathbf{h}}^{(3)}\mathbf{h}^{(2)^T} \\
\overline{\mathbf{h}}^{(2)} &= \mathbf{W}^{(2)^T}\overline{\mathbf{h}}^{(3)} \\
\overline{\mathbf{W}}^{(2)} &= \overline{\mathbf{h}}^{(2)}\mathbf{h}^{(1)^T} \\
\overline{\mathbf{h}}^{(1)} &= \mathbf{W}^{(1)^T}\overline{\mathbf{h}}^{(2)}
\end{aligned}$$

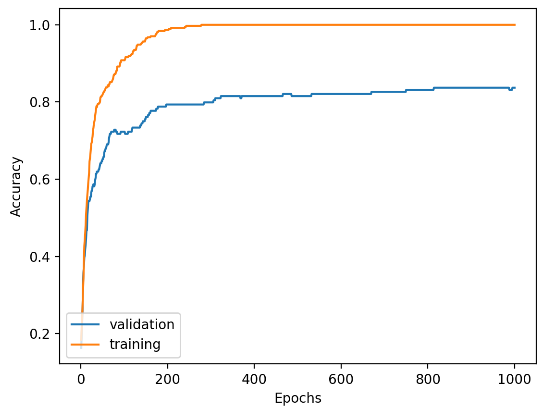
where $\overline{(\cdot)}$ represents a derivative in the chain rule. Given that each layer has K units, the dimensions of $\mathbf{W}^{(i)}$ is $K \times K$ and of $\mathbf{h}^{(i)}$ is $K \times 1$. Therefore, the time complexity of a single matrix-vector multiplication is quadratic with K (i.e. $O(K^2)$). Multiplying by $2N$, considering that their are two matrix calculations per layer, the overall time complexity of backpropagation is $2NK^2 \in O(NK^2)$.

It can be seen from this result that backpropagation is much faster than computing the gradient with respect to each weight individually.

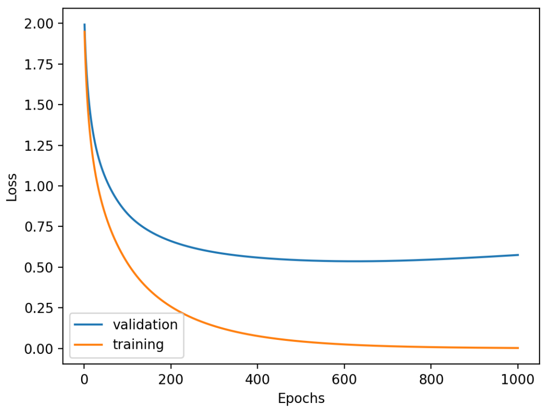
Part 8

An accuracy of 82.6% on the test set was able to obtain through trial and error. The final network has one hidden layer with 30 hidden units and the resolution ended up using was 32×32. The initial weights were Xavier uniform and the initial bias were 0.1. Several activation functions were tried during this process and it was found that Tanh produced the highest accuracy.

Mentioned in the previous project, there were a few non-face images in the dataset. Unlike project 1, the non-face images had to be removed. SHA-256 hashes was used to remove band image. If the hash values generated using SHA-256 is different from the hash value in the facescrub.txt file, the image will be discarded. Similar to project 1, the training sets (images) were first converted NumPy arrays and stack together vertically to create an $N \times M$ matrix where N is the number of images and M is the number of features. However, for this project, the pixels were converted to (-1, 1) instead of (0, 1).



(a) Epochs vs. Accuracy



(b) Epochs vs. Loss

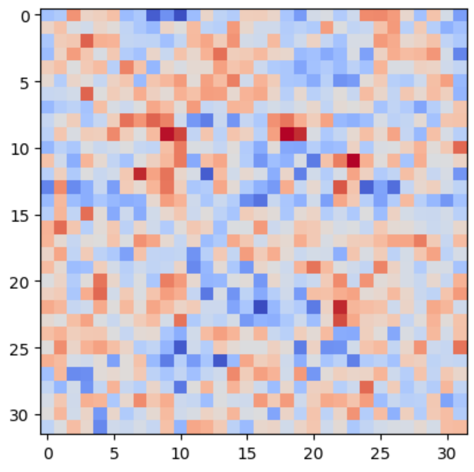
Figure 8: Learning Curves

The final network was fairly accurate in classifying the faces. Figure 8 shows the learning curve for the training and validation sets. As expected, the network predicts the faces 100% accurately on the training sets with loss of 0. The accuracy on the validation set is around 80 to 85% and the loss is fairly low. The accuracy of the network can be further improve by implementing AlexNet, which will be discuss in Part10.

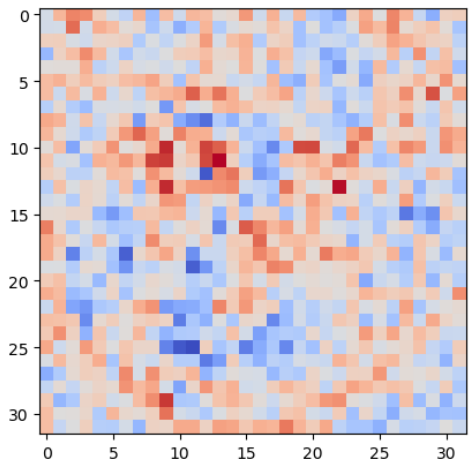
Part 9

In order to find out which hidden units are useful for classifying input photos as those particular actors, one would feed in images of an actor into the network, check the values of the hidden layer by computing model.forward(X).data.numpy, and see which hidden unit responds to the actor the most(see snippet of code below). The neuron would be activated the most if the input looks like the weight matrix. For example, when put in Bracco's images into the network, it outputs a array of hidden units that each picture activated the most. It was found that hidden unit 11 activated the most for this Bracco and hidden unit 4 for Carell (Figure9).

```
def part9():
    dtype_float = torch.FloatTensor

    train_set, _, _ = getSets(act, (60, 20, 0), 'processed/32x32')
    train_x = genX(train_set, 3072, 'processed/32x32')

    model = loadObj('model')
    model = nn.Sequential(model[0], model[1])

    img = train_x[:84,:]
    X = Variable(torch.from_numpy(img), requires_grad = False).type(dtype_float)
    out = model.forward(X).data.numpy()
    activation = np.argmax(out, 1).tolist()
    print('Lorraine␣Bracco:␣{}'.format(max(set(activation), key=activation.count)))
```



(a) Hidden Unit 4-Carell



(b) Hidden Unit 11-Bracco

Figure 9: Visualization of the weights

Part 10

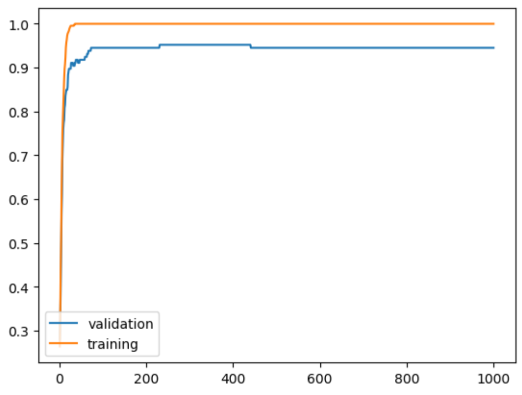
To extract the values of the activations of AlexNet, the `classifier` sequential layer was removed from the AlexNet class. This only left the `features` layer with five convolution network ”sub”-layers. Each of FaceScrub images were then converted to 227×227 resolution and then fed, one-by-one, through AlexNet to produce a entirely new data set of PyTorch `Variable`'s. This data set was then saved as NumPy arrays using Pickle.

A new network similar to the one constructed in part 8 was learned and then used to classify the images. Its structure remains the same, one hidden layer with 30 units, fully connected, and Tanh as the activation function. The following code snippet demonstrates this structure:

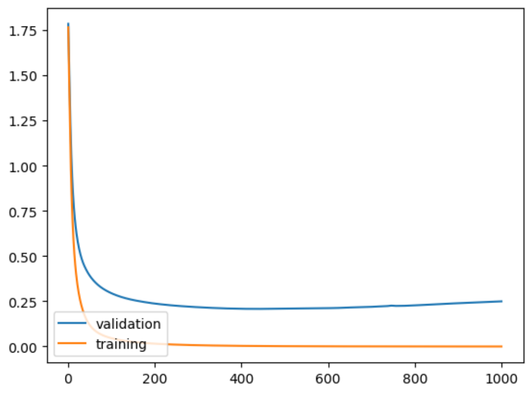
```
class Classify(nn.Module):
    def init_weights(self):
        nn.init.xavier_uniform(self.classifier[0].weight.data)
        nn.init.constant(self.classifier[0].bias, 0.1)

        nn.init.xavier_uniform(self.classifier[2].weight.data)
        nn.init.constant(self.classifier[2].bias, 0.1)

    def __init__(self):
        super(Classify, self).__init__()
        self.classifier = nn.Sequential(
            nn.Linear(9216, 30),
            nn.Tanh(),
            nn.Linear(30, 6)
        )

        self.init_weights()

    def forward(self, x):
        return self.classifier(x)
```

The weight matrices were initiated using PyTorch's `Xavier_uniform` because this gave the best results (i.e. highest accuracy on the test set) for parts 8 and 10. Additionally, the bias vectors were initialized as constant values of 0.1 for each element. Tanh was used as the activation function because it also gave the best results when compared to other activation functions such as softmax, sigmoid, and ReLU.

The constructed network was trained on top of the AlexNet layer by feeding the new dataset of NumPy arrays as input, identical to the procedure followed in part 8. Minibatches of size 16 are fed into the forward function for a total of 1000 epochs. This produced the following learning curves:



(a) Epochs vs. Accuracy



(b) Epochs vs. Loss

Figure 10: Learning Curves

The performance of the network on the test set was 93.8% accuracy, a considerable improvement from part 8. Figure 10 shows the learning curve for the training and validation sets. As expected, the network predicts the faces 100% accurately on the training sets with loss of 0. The accuracy on the validation set is now up to 90 to 95% and the loss is very low, lower than in part 8.

As a next step, the accuracy of the network could be even further improved by tinkering with the AlexNet layers (by turning some layers off and others on) or by inputting higher resolution images.