CSC411 Machine Learning
Project 2: Deep Neural Networks

Ariel Kelman
Student No: 1000561368

Gideon Blinick
Student No: 999763000

23 February 2018

1 Introduction

This section provides an introduction to Project 2, including the data used, and notes on reproducing the results. Sections 2-6 discuss the implementation of a simple neural network to classify images from the MNIST database, section 7 is a theoretical anlaysis of the efficiency of backpropogation, and sections 8-10 work with deep neural networks.

1.1 Digits

The following figure shows 10 random images from the training set of each of the digits.

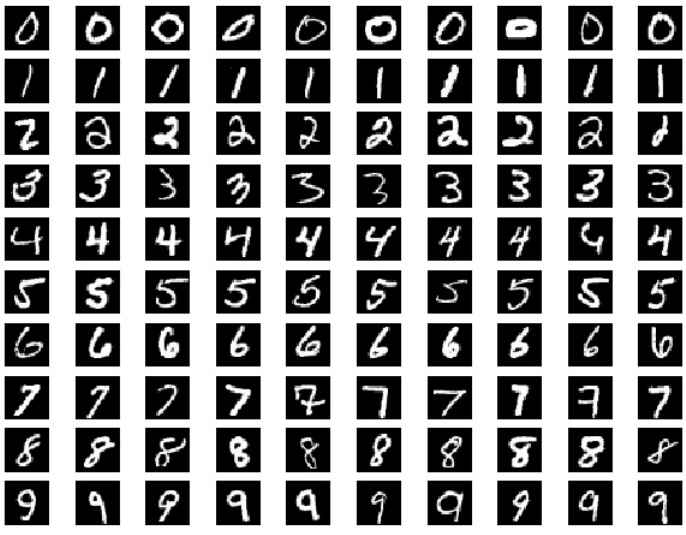

Figure 1: 10 random samples from the training set for each digit. This image was generated using `plot_samples()`.

The data was downloaded from the assignment webpage, and imported into Python using the provided code. The data was already divided into training and testing sets, but the testing set was divided into two sets of 5000 samples each, for a separate validation and test set. The training set contains 60000 samples, divided almost equally between the ten digits.

1.2 Results & Report Reproducibility

All results and plots can be generated using the appropriate python files. All code is python 3, run with Anaconda 3.6.3. Code for sections 1-6 can be found in `digits.py`, for part 8 and 9 in `faces.py`, and for part 10 in `deepfaces.py`. Running the code will save all generated images in the `resources` folder, where they are used by LaTeX. Note that some of the sections require the code for other sections (in the same file) to be run first. Additionally, the files for parts 8 and 10 contain similar functions; it may be useful to restart the python shell when switching files to ensure reproducibility. To reproduce the report, simply run it through LaTeX. This will pull the most recently generated figures from the `resources` folder.

2 The Network

The following function implements a neural network with no hidden layers, with the output passed through a softmax layer to estimated probabilities.

```
def no_hidden_layers(x, W, b):
  '''Compute the network'''
   #the first column of W contains the weights for output 1
   # W is (784x10), x is (784x60000) and b is (10x60000)
   # So L1 is (10x60000)
   L1 = np.dot(W.T, x) + tile(b, (1, np.shape(x)[1] ) )
 return softmax(L1)
```

The network is described by the weights and biases from the 784 = 28*28 inputs (representing pixel intenisites of the input image) to ten "output" nodes (with the identity as the activation function). The output from this layer is what is passed through the sofmax function $p_k = \frac{e^{o_k}}{\sum_q e^{o_q}}$.

The weights are represented as a 784 by 10 matrix W, where w_{ij} (i^{th} row, j^{th} column) represents the weight from the i^{th} input to the j^{th} output. When computing the network on a given sample, the transpose of W is multiplied by the column vector (or matrix when computing on multiple samples) representing the input. The biases are represented by a 10 by 1 vector, one entry for each output. We use the `tile()` function to duplicate the biasses b up to appropirate dimensions (10 by 60000) to be added to the result of $W^T x$.

Throughout the code, $y\_$ is a matrix representing the correct results for each image; each column is a vector of zeros with a 1 in the place representing the correct digit. y is a matrix representing the output probabilities from the network of the same dimensions as $y\_$ (in the derivations below, as well as some comments on the code, p is used in place of y); each column is a vector representing the probabilities for each digit for a particular sample.

3 Gradient

The cost function (negative log probability) is taken to be $-\sum_k y_k ln(p_k)$ for one sample, where y_k is 1 for the correct class and 0 otherwise, and p_k is the prediction probability for class k. Writing this in vector notation (replacing the sum with a vector dot product) and summing over all the samples gives:

$$C = -\sum_s y^{(s)} ln(p^{(s)})$$

where ln is applied pointwise, and s is an index over the samples. $y^{(s)}$ is a vector of 0's with a 1 in the position representing the correct digit.

The following results will be used in the derivations throughout this section:

$$\frac{\partial p_k}{\partial o_q} = \begin{cases} -p_k p_q & \text{if } k \neq q \\ p_q(1-p_q) & \text{if } k = q \end{cases}$$

These results follow directly from the definition of the softmax function.

3.1 Gradient wrt w_{ij}

First, we derive $\frac{\partial C}{\partial o_q}$ (for one sample), to be used in the derivation of $\frac{\partial C}{\partial w_{ij}}$.

$$\begin{aligned}
\frac{\partial C}{\partial o_q} &= \left[\frac{\partial C}{\partial p_q}\frac{\partial p_q}{\partial o_q} + \sum_{k\neq q}\frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial o_q}\right] \\
&= -\left[y_q \frac{1}{p_q} p_q(1-p_q) + \sum_{k\neq q} y_k \frac{-1}{p_k} p_k p_q\right] \\
&= -\left[y_q(1-p_q) - \sum_{k\neq q} y_k p_q\right] \\
&= \left[-y_q + \sum_k y_k p_q\right] \\
&= p_q - y_q
\end{aligned}$$

The last line follows because $\sum_k y_k = 1$. Using this result, and applying the chain rule gives:

$$\begin{aligned}
\frac{\partial C}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \sum_s y^{(s)} ln(p^{(s)}) \\
&= \sum_s \sum_q \frac{\partial C}{\partial o_q}\frac{\partial o_q}{\partial w_{ij}} \\
&= \sum_s \sum_q (p_q - y_q) x_i \delta_{jq} \\
&= \sum_s (p_j - y_j) x_i
\end{aligned}$$

As the outputs are linear functions of both the inputs and the weights, the derivative with respect to a weight is given by the value of the input to that weight, namely x_i. The Kronecker delta δ_{jq} results from $\frac{\partial o_q}{\partial w_{ij}}$ being nonzero only when $j = q$. The (s) superscript indicating the sample index was omitted throughout the derivation for clarity of presentation.

3.2 Vectorized Gradient Code

The following code computes the gradient with respect to the wieghts and biases (representing all input training images in a matrix X, 784 by the number of samples, in this case 60000):

```
def grad(y_, y, x):
  '''Compute the gradient wrt weights and biases'''
   #y and y_ have dimension (10x60000)
   #x has dimension (784x60000)
```

```
            diff = (y - y_) #y is output of softmax

            grad_W = np.dot( x, diff.T )
            grad_b = np.sum( diff, 1)
            grad_b = np.reshape(10,1)

        return  grad_W, grad_b
```

The above code works by creating a matrix ∇C which is of dimension 784x10 (same as the weight matrix), and where every element contains $\frac{\partial C}{\partial w_{ij}}$ for the corresponding w_{ij}. It does this by implementing $\sum_s (p_j - y_j)x_i$ for each matrix element, where i increases as you go down a column (from 1 to 784) and j increases as you go accross a row (from 1 to 10). Each element itself contains 60000 terms added together to aggregate the information from all samples.

The gradient function was checked using the `finite_diff()` function. Along directions $(i, j) = \{(45, 7), (453, 8), (129, 2), (631, 3)\}$ the difference between the gradient and the finite difference was less than 0.001, with an ϵ of 0.0001. Note that the comparison was done at the point W after training for Part 4. Decreasing ϵ improves the results even further.

4 Training

The neural network was trained on the full training set of 60000 images without momentum. A learning rate of 10^{-4} was used for 1000 iterations, giving a training accuracy of 92.6%, validation accuracy of 92.3%, and a testing accuracy of 91.9%. The weights and biases were initialized using a uniform distribution to values between 0 and 1. To determine the optimal learning rate, the function `optimize_learning()` tested several learning rates, outputting the validation accuracy for each. Learning curves for each learning rate are saved as images in the `resources` directory (corresponding to the list of rates in `digits.py`); and a rate of 10^{-4} gave the best results on the validation set.

The following figure shows the learning curve for the final parameters used; the dips likely come when gradient descent left a local minimum:

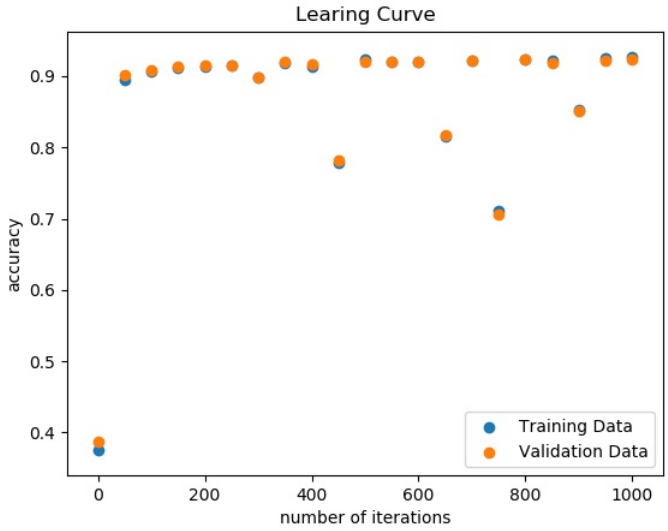

Figure 2: Learning curves with a learning rate of 10^{-4}.

A surprisingly small number of iterations was required (espcially in comparison with Project 1),

with almost no improvement past 1000 iterations. The large size of the training set likely contributed to this.

The following figure, generated using `image_W()` shows the weights input to each output neuron (before the softmax layer), corresponding to the digits 1...10. The general outlines of the digits can be seen; the pattern is clearer for more complex-shaped digits such as 8.

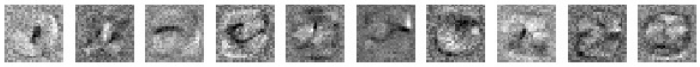

Figure 3: Visualization of the weights. Note that the image was manually cropped to remove whitespace.

5 Training with Momentum

The neural network was again trained on the full training set of 60000 images with momentum, set to 0.8. A learning rate of 10^{-4} was used for 1000 iterations, giving a training accuracy of 93.4%, verfication accuracy of 92% and a testing accuracy of 91.8%. The weights and biases were again initialized using a uniform distribution to values between 0 and 1.

The following figure shows the learning curves:

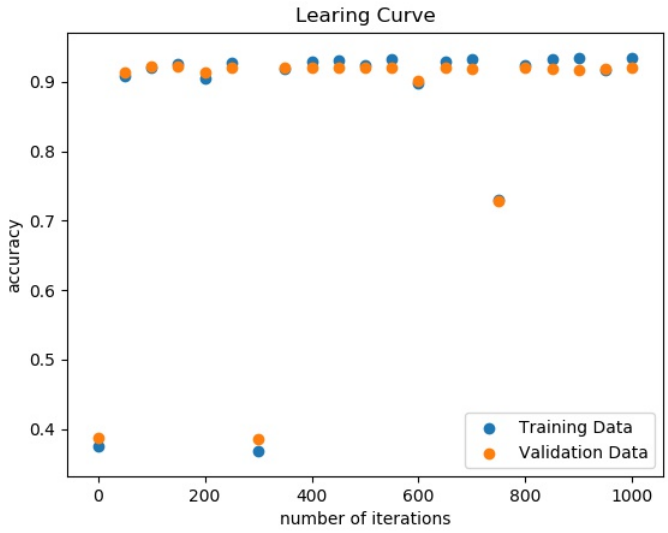

Figure 4: Learning curves for neural network with momentum of 0.8

To determine the best value of momentum, the function `optimize_momentum()` was used. Learning curves for the various tested momenta can be found in the `resources` folder; the value 0.8, though smaller than traditionally used, gave the best results on the validation set. Though ideally one would vary both the learning rate and momentum together, to save on computation time, the optimal rate of 10^{-4} from Part 4 was used. This still allows for the effects of momentum to be clear, as can be seen by comparing the learning curves with and without momentum - using momentum results in much

faster convergence to near the minimum, followed by small oscillations as it settles in, and the effects of momentum disperse.

The backpropogation algorithm from part 4 was altered for this part to accomodate momentum. A momentum term was added, along with the corresponding update equations for W and b that make use of the momentum term. Thus, our new backprogation function was implemented as:

```
def backprop(x_train, y_train, x_val, y_val, W, b, rate, max_iter, mom=0, filename
=''):
    iter_acc = []
    train_acc = []
    test_acc = []

    nu_W = np.zeros( np.shape(W) )
    nu_b = np.zeros( np.shape(b) )

    iter = 0
    while iter <= max_iter:
        y = no_hidden_layers(x_train, W, b)

        grad_W, grad_b = grad(y_train, y, x_train)
        nu_W = mom*nu_W + rate*grad_W
        nu_b = mom*nu_b + rate*grad_b
        W -= nu_W
        b -= nu_b

        if iter%50 == 0:
            iter_acc += [iter]

            y = no_hidden_layers(x_train, W, b)
            res = check_results(y_train, y)
            train_acc += [ res.count(1)/len(res) ]

            y = no_hidden_layers(x_val, W, b)
            res = check_results(y_val, y)
            test_acc += [ res.count(1)/len(res) ]
        iter += 1

    #Plot the learning curves
    if filename:
        plt.scatter(iter_acc, train_acc, label='Training Data')
        plt.scatter(iter_acc, test_acc, label='Test Data')
        plt.title('Learing Curve')
        plt.xlabel('number of iterations')
        plt.ylabel('accuracy')
        plt.legend()
        plt.savefig('resources/' + filename)
        plt.close()

    return W, b
```

6 Analysis of Momentum

6.1 Contour Plot

For the analysis in this section, the weights $w_1 = w_{375,2}$ and $w_2 = w_{475,4}$ (chosen randomly) were used (recall that W is a 784 by 10 matrix). After training the network as described in the previous section, these weights had the values $w_{375,2} = 1.55$ and $w_{475,4} = 1.3$. These values correspond the the 0's in the plots below (note that only the training set cost was used in this section).

The following plot shows the variation of the cost function (negative log loss) around the values for $w_{375,2}$ and $w_{475,4}$ chosen above.

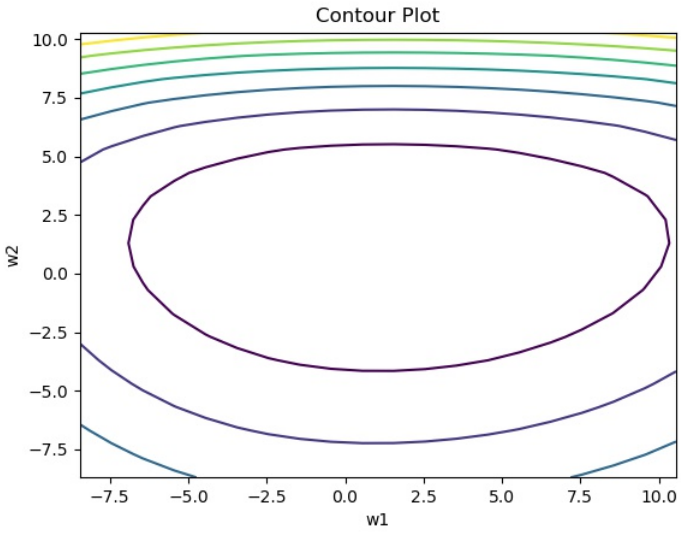

Figure 5: Contour Plot of the Cost around optimal values for w_1 and w_2.

6.2 Weight Trajetories & Discussion

A modified version of the backpropogation function was created to only modify the two weights under consideration. As efficiency was not a huge concern for the small network, and to avoid changing all the weights other than w_1 and w_2, the `finite_diff()` function was used to determine the derivate with respect to the two weights, with an ϵ of 0.01.

The following figure shows how the weights move back to their optimal values after being initialized some distance away. A learning rate of 0.01 was used for 80 steps, and the momentum coefficient for the trajectory with momentum was 0.8.

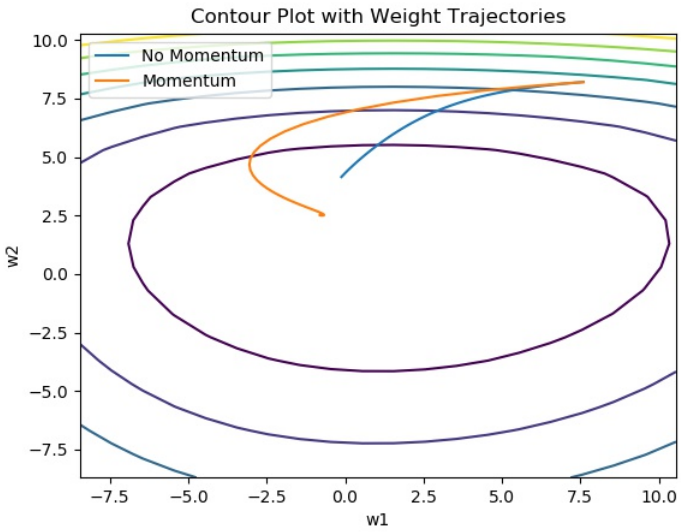

Figure 6: Contour Plot with Weight Trajectories.

The momentum trajectory moves much faster than the trajectory without momentum (this is even more clear when the process is run for fewer - e.g. 40 - iterations).

The wieghts chosen above were deliberately chosen to be central in the weight matrix W (though taken randomly other than that), so that the weights would match an input that is significant in terms of identifying digits - i.e. close to the center of the image. Taking wieghts close to the edge does not give a good contour plot, as the weights there (after training) are all very close to zero (or zero exactly), and so have no effect on the predictions - and therefore no effect on the cost.

7 Efficiency of Vectorized Backpropogation

Consider a fully connected neural network with N layers each containing K neurons. By convention, the input layer will not be counted toward N layers, but the final output layer will be counted (thus there are N sets of weights and biases that must be learned). Any function (such as softmax) applied to the final layer is ignored; none of these conventions will affect the asymptotic runtime. We also assume that the results of the forward pass of the network are cached, to be used for both vectorized backpropogation and computing the gradient with respect to each weight independenly. We can also ignore the biases; including them could be accomodated by adding an extra neuron to every layer (except the last) with a constant value of 1. This would be equivalent to increasing K by 1. As a final note, any functions (such as softmax) applied to the output are ignored; these would have no asymptotic effect on N and would increase linearly with K.

Denote each layer by the numbers $1, 2...N$ (we can refer to the input layer as layer 0). The efficiency of computing the gradient with respect to a weight to/from a particular layer does not depend on which weight it is, so w^a can stand for an arbitrary weight from layer $a-1$ to a. A particular neuron in layer n will be denoted by σ^n with a subscript to denote that one specific neuron in the layer is intended (the one which the weight under consideration leads to).

Then the following holds:

$$\frac{\partial C}{\partial w^N} = \frac{\partial C}{\partial \sigma_q^N}\frac{\partial \sigma_q^N}{\partial w^N}$$

$$\frac{\partial C}{\partial w^{N-1}} = \sum_q^K \frac{\partial C}{\partial \sigma_q^N}\frac{\partial \sigma_q^N}{\partial \sigma_m^{N-1}}\frac{\partial \sigma_m^{N-1}}{\partial w^{N-1}}$$

$$\frac{\partial C}{\partial w^{N-2}} = \sum_q^K \sum_m^K \frac{\partial C}{\partial \sigma_q^N}\frac{\partial \sigma_q^N}{\partial \sigma_m^{N-1}}\frac{\partial \sigma_m^{N-1}}{\partial \sigma_h^{N-2}}\frac{\partial \sigma_h^{N-2}}{\partial w^{N-2}}$$

Again, in each case, the derivative of the last neuron ($\sigma_q, \sigma_m, \sigma_h$ respectively in the equalities above) with respect to a weight leading to that layer refers to the neuron where that weight terminates. Since there are K neurons in each layer, each sum goes up to K.

With K neurons in each layer, there are K^2 weights between layers; thus the efficiency of finding $\frac{\partial C}{\partial w^N}$ for a specific weight is $O(1)$ with respect to N and K, but for all the weights is $O(K^2)$. For a specific weight into layer $N-1$, the efficiency would be $O(K)$; scaling this to all the weights is $O(K^3)$, since the results for other weights are not cached. Continuing this trend backward (each sum adds a factor of K), the efficiency of finding the derivative of all the weights into the second hidden layer is $O(K^N)$. If we make the simplifying assumption that there are K input neurons, then the efficiency of finding all the derivatives for the weights in the first layer is $O(K^{N+1})$. Taking only the highest order, gives an overall efficiency of $O(K^N)$.

The exponential dependence on N also follows intuitively - as adding and extra layer exponentially increases the number of paths from the cost to the weight, and to find the derivative, each of these paths must be followed.

For vectorized backpropogation, the derivative of the cost with respect to each layer can be computed using the cached values and matrix multiplication. Ignoring the complexity of carrying the derivative through the activation function, the derivative with respect to each weight between layers $N-1$ and N requires just one matrix multiplication. Moving backward through each layer adds one more matrix multiplication. With each layer having K neurons, these matrices have dimension K by K. The complexity of matrix multiplicaion is polynomial in their dimensions; multiplying two K by K matrices has complexity $O(K^3)$ (there are algorithms with slightly lower powers, but we'll take 3 as a baseline here). Therefore, to compute the gradient with respect to all the weights from the input to the first hidden layer is $O(NK^3)$, and if this is done one matrix at a time (i.e. propogating backwards one layer at a time), the derivative with respect to all the intermediate weights falls out at no extra cost (they are the result of already computed matrix multiplications). Thus the complexity of vectorized backpropogation with caching is $O(NK^3)$ for a network with N layers (excluding the input layer) each of K neurons.

For large neural networks, particularly those with several hidden layers, fully vectorized and cached backpropogation reduces the asympotic runtime from to exponential in N to being linear in N, a huge speedup.

8 Classifying Faces

The data for this section was downloaded using the file `get_data.py`. Carrying over a blacklist for bad images from Project 1, and adding in a check to ensure the hashes were correct, the images were downloaded, converted to grayscale, and cropped according to the given bounding box. The resulting image was rescaled to 32 by 32 pixels. While over 90 images were downloaded for Lorraine Bracco, Angie Harmon, Alec Baldwin, Bill Hader, and Steve Carell, only 57 images were downloaded for Peri Gilpin. Therefore, 70 images were used in the training set for each actor, while 37 were used for Gilpin, leaving 20 images (per actor) for the validation and test sets. Rather than use an additional 10 images

for the test set (for 20 test images per actor), and possibly even doing the same for the validation set, cross-validation (i.e. choosing the training/validation/testing set differently) was used as described below.

The supplied code was modified to create a fully connected, single hidden layer (with RELU activation function) neural network. The data was loaded and formatted using the `format_data()` function, which creates a dictionary similar to the format in which the MNIST data was loaded: a key for the training, validation, and testing set for each actor, with the value holding a `numpy` array holding the appropriate data. The function `get_set_data()` then retrieves these arrays for use in the neural network, and creates the arrays holding the correct classification for each image.

The network was trained using the entire training set, divided into random mini-batches at the start of each epoch (6 mini-batches per epoch), using `Adam` as the optimizer. The weights and biases were initialized randomly using torche's `randn()` function. The following figure shows the learning curve for the training and validation sets for the training with the final parameters: learning rate of 10^{-3}, 25 hidden neurons, and 5 epochs each with 6 mini-batches of 1000 iterations.

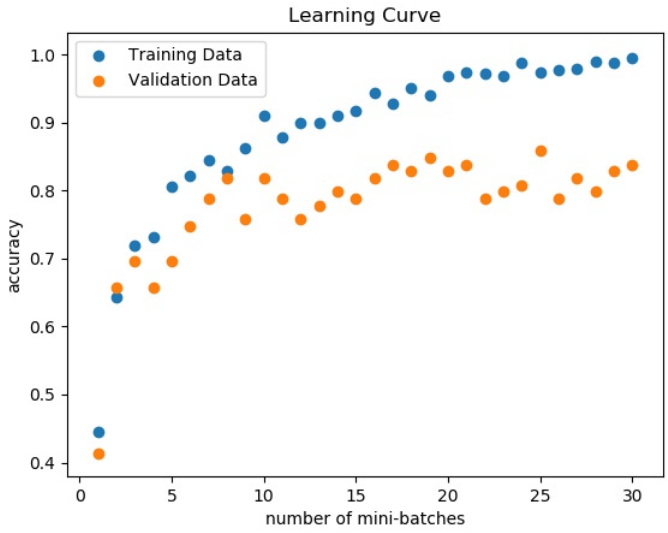

Figure 7: Learning Curve - the oscillations are due to changing the training set using mini-batches. There are six mini-batches to an epoch.

Because of the large number of hyperparameters (learning rate, number of epochs, size of the training/validation sets, number of neurons in the hidden layer, etc.), it was difficult to optimize the network. Rather than iteratively fully exploring a subspace of possible hyperparameters (which would grow extremely quickly even for few choices for each parameter), the function `optimize_params()` was used to sample several options, and was run several times with different divisions into training/validation/testing sets (changed by setting the seed in `format_data()` to 0, 1, 2, and 3). The validation results for each set of parameters were then approximately averaged, and that result led to a choice of trial 2 (as defined in the `optimize_params()` function), i.e. the parameters given above. Small changes around these values (particularly increasing the number of epochs) had small effects on the results.

These parameters give a final training accuracy of $> 99\%$, validation accuracy of 81%, and testing accuracy $of 83.3\%$ when averaged over all the seeds. While the validation set did have an effect on training through the choice of hyper-parameters, this is mitigated by the cross-validation; taking it to be part of the testing set gives an accuracy of 82.15%.

To further improve this result, experimentation with color images, or vastly increasing the number of hidden neurons would be a good place to start. With more hidden neurons, there may be greater

ability for the network to abstract properties of each actor. However, testing this with 200 hidden neurons (trial 6 in `optimize_params()`) gave validation accuracy of 76.8% and testing accuracy of 75%, for a data seed of 1. This was the lowest accuracy of any of the `optimize_params()` trials with that seed, but was not tested for other divisions of the data due to the long runtime (the number of epochs was increased to allow more time for all the neurons to converge).

9 Visualizing Weights

Figure 8 shows all the weights from the input layer (1024 inputs) to the first (only) hidden layer in Part 8 (25 hidden units).

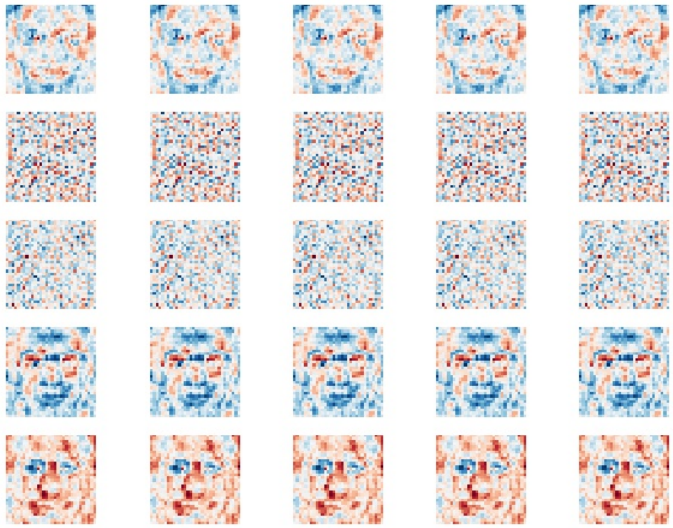

Figure 8: Visualization of the weights into the hidden layer (the color is artificial).

To identify the faces shown in these images, they were passed through the neural network, to see the predictions. Based on this (and ignoring the images in rows 2 and 3), Hader is most present in images 2,3, and 20, and Harmon in images 4 and 19. The rest of the images are most identified with the other actors (see the code at the end of `faces.py`, where one can simply identify each image with the list of actors). As each image is a representation of the weights into a neuron in the hidden layer, the images noted above are associated with the same neurons in the network.

10 AlexNet

To download the data, `get_data.py` was run with slightly different parameters, to retain the images in color, and resize the images to 227 by 227. The number of images downloaded varied slightly from the numbers in Part 8. These images are stored in the `resources` folder. The validation and test sets were 10 images per actor, to maximize the number of images in the training set. As in Part 8, fewer training images were used for Peri Gilpin than for the rest of the actors.

Starting from the supplied AlexNet code, a new function `conv_output()` was defined, to return the activations from the final convolutional layer. Using `format_data()` to generate a dictionary of filenames with training, verification, and testing data for each of the six actors (same actors as in Part 8). This was then passed through `get_activations()`, which ran each image through AlexNet, retrieved the 9216 activations after the final convolutional network, and returned a list containing these

activations for each image. Once all the activations are retrieved, appropriate arrays are generated with the correct classification for each set of activations.

In the function `train()`, a torch neural network is defined, and trained using mini-batches. Using a fully connected network with one hidden layer of 100 neurons, a learning rate of 0.001, and 5 epochs with 6 mini-batches of 100 iterations each, gave perfect training accuracy, and 81.7% and 80% accuracy for the verification and training sets respectively. Due to the significant runtime, we did not preform cross-validation on the division of data into training, verification, and testing data.

Due to the lengthy runtime and large space for the hyperparameters, a similar approach to that in Part 8 was adopted to optimize the hyperparameters: varying several of the parameters independently, and continuing to optimize based on the results. These variations included extending the network to two hidden layers, but that change did not improve results (at least not with the other settings for the hyperparameters): trying both 50 and 100 neurons per hidden layer (with learing rate 0.001, 5 epochs as above) gave no better validation results (78.3% and 81.7% respectively), though the validation results had saturated. Even increasing to three hidden layers did not improve this result.

The final parameters that were used that gave reasonable accuracy (though not significantly better than Part 8) were those given above (doubling the number of epochs slightly increased accuracy, but at large computational cost), giving a perfect training accuracy, a validation accuracy of 81.7%, and a testing accuracy of 80%.

The following figure shows the learning curves with these settings. Note the convergence of the validation and test accuracy, showing that further iterations does not improve results.

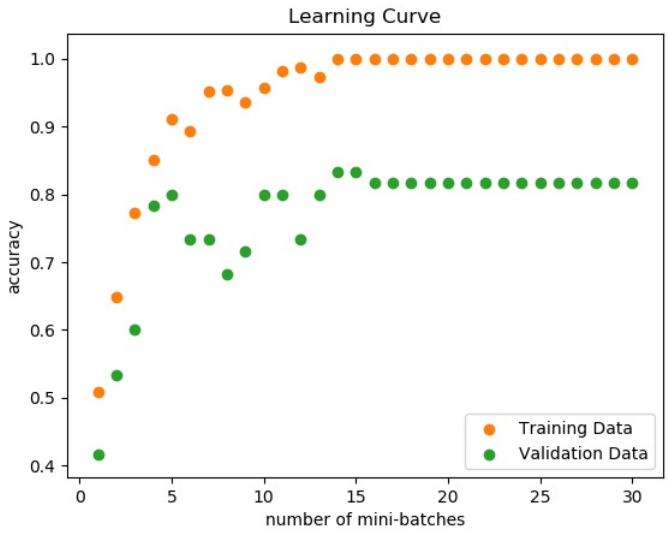

Figure 9: Learning Curves for Part 10.