CSC411: Project 2
Deep Neural Networks for Handwritten Digit and Face Recognition

Due on Sunday, March 5, 2017

Michelle Leung and Tianyi Yu

February 23, 2018

Part 1

Dataset description
The dataset contains handwritten, white-on-black images of digits from 0-9. The numbers are written with various stroke widths, pen weight, and writing styles, but for the most part the orientation of each set of digits is largely the same. The following is a random sampling of 10 images for each digit:

Figure 1: A selection of 0's from the MNIST dataset.

Figure 2: A selection of 1's from the MNIST dataset.

Figure 3: A selection of 2's from the MNIST dataset.

Figure 4: A selection of 3's from the MNIST dataset.

Figure 5: A selection of 4's from the MNIST dataset.

Figure 6: A selection of 5's from the MNIST dataset.

Figure 7: A selection of 6's from the MNIST dataset.

Figure 8: A selection of 7's from the MNIST dataset.

Figure 9: A selection of 8's from the MNIST dataset.

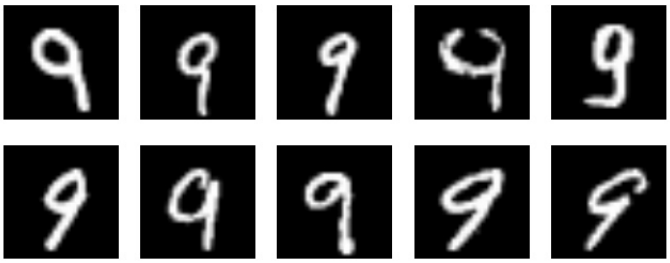


Figure 10: A selection of 9's from the MNIST dataset.

Part 2

Implementation of a Network

The following is an implementation of the given network in Python where the o's are a linear combination of the x's such that

$$o_i = \sum_j w_{ij} x_j + b_i \tag{1}$$

```
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 calculate_output(w, x):
    out = np.dot(x, w[1:785, :]) + w[0, :]
    out = out/len(out)
    return softmax(out)

def part2():
    np.random.seed(0)
    w = 2*np.random.random((785, 10))-1
    x = np.random.random((1, 784))
    print(calculate_output(w, x))
```

Part 3

Using the sum of the negative log-probabilities as the cost function

(a) Given that we would like to use the sum of the negative log-probabilities, the cost function for which we would like to find the gradient for is:

$$C = -\sum_j y_j \log p_j \tag{2}$$

which is the negative log-likelihood, where the likelihood of y is

$$p_i = \frac{e^{o_i}}{\sum_j e^{o_j}} \tag{3}$$

And the outputs o_i with respect to the weight w_{ij}

$$o_i = \sum_j w_{ij} x_j + b \tag{4}$$

Using chain rule, the gradient of the cost function of one training set can be expressed as:

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial p_j} \cdot \frac{\partial p_j}{\partial o_i} \cdot \frac{\partial o_i}{\partial w_{ij}} \tag{5}$$

Starting with the right most product term, we have

$$\frac{\partial o_i}{\partial w_{ij}} = \frac{\partial \sum_j w_{i,j} x_j + b}{\partial w_{i,j}} = x_j \tag{6}$$

since o_i is a linear function with respect to w_{ij} and constant offset b. For the product term in the centre on the right hand side of equation (5), we write:

$$\frac{\partial p_j}{\partial o_i} = \frac{\partial e^{o_i}}{\partial o_i \sum_j e^{o_j}} \tag{7}$$

for which there are two cases: when $i = j$ and when $i \neq j$. In the first case, for $i = j$ we have,

$$\frac{\partial p_j}{\partial o_i} = \frac{\sum_j e^{o_j} e^{o_i} - e^{o_i} e^{o_i}}{(\sum_j e^{o_j})^2} = \frac{e^{o_i}}{\sum_j e^{o_j}} \cdot \frac{\sum_j e^{o_j} - e^{o_i}}{\sum_j e^{o_j}} = p_i \cdot (1 - p_i)|_{i=j} \tag{8}$$

Similarly, when $i \neq j$ we have,

$$\frac{\partial p_j}{\partial o_{i \neq j}} = \frac{0 - e^{o_i} e^{o_i}}{(\sum_j e^{o_j})^2} = -p_i p_j|_{i \neq j} \tag{9}$$

Lastly, for the leftmost term on the right hand side of equation (5), we write:

$$\frac{\partial C}{\partial p_j} = \frac{\partial [-\sum_j y_j \log p_j]}{\partial p_j} = -\sum_j y_j \frac{1}{p_j} \tag{10}$$

Combining the three partial derivatives using chain rule for the gradient function and rearranging,

$$\frac{\partial C}{\partial w_{ij}} = [x_j] \cdot [\sum_{i \neq j} (-y_j \frac{1}{p_j})(-p_i p_j) - y_i \frac{1}{p_i} p_i (1 - p_i)] = x_j [p_i (\sum_{i \neq j} y_j + y_i) - y_i] \tag{11}$$

Since $\sum_{i \neq j} y_j + y_i = 1$, this simplifies to:

$$\frac{\partial C}{\partial w_{ij}} = x_j \cdot (p_i - y_i) \tag{12}$$

(b) Code for computing the gradient in vectorized form:

```
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 cross_entropy(y, y_):
    # cross entropy cost function
    return -sum(y_*log(y))

def calculate_output(w, x):
    x = vstack( (ones((1, x.shape[1])), x))
    out = dot(w.T, x)
    return softmax(out)

def f(x, y_, w):
    # cost function y_ * log(softmax(y))
    p = calculate_output(w, x)
    return cross_entropy(p, y_)

def df(x, y_, w):
    p = calculate_output(w, x)
    x = vstack( (ones((1, x.shape[1])), x))
    return dot(x,(p-y_).T)

def part3():
    #Set up the vectors to compute the gradient
    n = 500
    x = M["train0"][0:n]
    x = vstack((x, M["train1"][0:n]))
    x = vstack((x, M["train2"][0:n]))
    x = vstack((x, M["train3"][0:n]))
    x = vstack((x, M["train4"][0:n]))
    x = vstack((x, M["train5"][0:n]))
    x = vstack((x, M["train6"][0:n]))
    x = vstack((x, M["train7"][0:n]))
    x = vstack((x, M["train8"][0:n]))
    x = vstack((x, M["train9"][0:n]))
    x = x.T/255.0

    y_ = []
    for i in range(0, n):
        y_.append([1,0,0,0,0,0,0,0,0,0])
    for i in range(0, n):
        y_.append([0,1,0,0,0,0,0,0,0,0])
    for i in range(0, n):
        y_.append([0,0,1,0,0,0,0,0,0,0])
    for i in range(0, n):
        y_.append([0,0,0,1,0,0,0,0,0,0])
    for i in range(0, n):
        y_.append([0,0,0,0,1,0,0,0,0,0])
    for i in range(0, n):
```

```
        y_.append([0,0,0,0,0,1,0,0,0,0])
    for i in range(0, n):
        y_.append([0,0,0,0,0,0,1,0,0,0])
    for i in range(0, n):
        y_.append([0,0,0,0,0,0,0,1,0,0])
    for i in range(0, n):
        y_.append([0,0,0,0,0,0,0,0,1,0])
    for i in range(0, n):
        y_.append([0,0,0,0,0,0,0,0,0,1])

    y_ = array(y_).T

    np.random.seed(0)
    w_init = np.random.random((785, 10))
    w = df(x, y_, w_init)
```

Comparison of the gradient with finite differences at several points:

```
@[350,2]
Finite Difference:
49.0703075229
Direct derivative calculation:
49.0703074999

@[150,7]
Finite Difference:
73.2969259843
Direct derivative calculation:
24.2266184493
```

Part 4

Training and Optimizing the Neural Network The network was trained using gradient descent with a learning rate of 0.00002. The initial weights and bias terms are random variables from a normal sample with $\mu = 0$ and $\sigma = 0.001$ using a seed of 1234. 500 images of each digit were used as the training set and another 50 images of each digit for the test set. The network learned for 3000 iterations of gradient descent, reaching a test set performance of 90.6%. The accuracy starts to fluctuate around 2500 iterations, possibly indicating the start of overfitting.

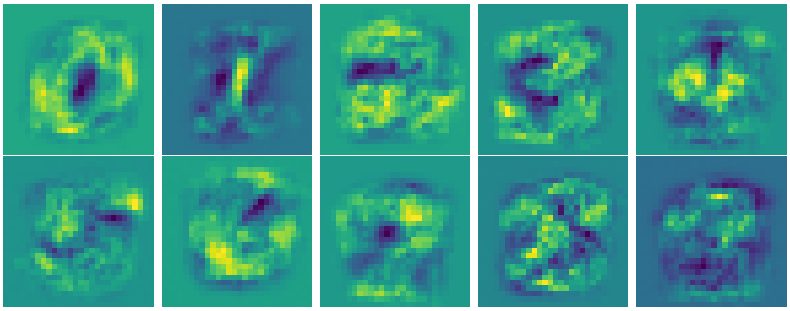


Figure 11: Visualization of the weights going into each of the output units for each digit
Top: 0-4; Bottom: 5-9

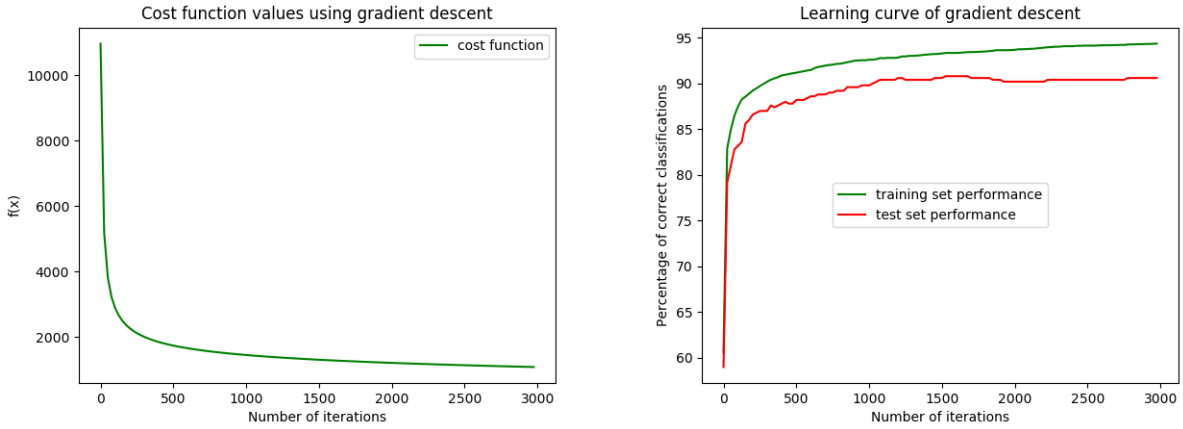


Figure 12: *(Left)* Graph; *(Right)*

Part 5

Classification using Logistic Regression vs Linear Regression

The advantage of using logstic regression is that the cost function doesn't change drastically due to outliers in the training set. Using linear regression, one sample that is very different from others with the same label will cause large changes in the cost function, leading to overfitting on the outlier sample. Logistic regression avoids this by applying the logarithm, which greatly reduces the cost function due to outliers in the sample.

In this part, data is generated from the same set as previous part but 10% of the training set for each digit is purposefully mislabeled with an incorrect y value. This simulates having extreme outliers in the training set. The code for this part is shown below:

```
# cost functions using linear/logistic regression
# and their gradients
def f_lin(x, y, theta):
    # cost function
    x = vstack( (ones((1, x.shape[1])), x))
    return sum( (dot(theta.T,x).T - y) ** 2)

def df_lin(x, y, theta):
    # derivative of cost function
    x = vstack( (ones((1, x.shape[1])), x))
    return 2*dot(x, ((dot(theta.T, x).T-y)))

def f_log(x, y_, w):
    # cost function y_ * log(softmax(y))
    p = calculate_output(w, x)
    return cross_entropy(p, y_)

def df_log(x, y_, w):
    p = calculate_output(w, x)
    x = vstack( (ones((1, x.shape[1])), x))
    return dot(x,(p-y_).T)

def grad_descent_lin(cost, grad, x, y, init_t, alpha):
    EPS = 1e-5   #EPS = 10**(-5)
    prev_t = init_t-10*EPS
    t = init_t.copy()
    max_iter = 500
    iter  = 0
    while norm(t - prev_t) >  EPS and iter < max_iter:
        prev_t = t.copy()
        t -= alpha*grad(x, y, t)
        if iter % 25 == 0:
            print ("Iter", iter)
            print("Lin f(x) = ", cost(x, y, t))
        iter += 1
    return t

def grad_descent_log(cost, grad, x, y, init_t, alpha):
    EPS = 1e-5   # EPS = 10**(-5)
    prev_t = init_t-10*EPS
```

```
    t = init_t.copy()
    max_iter = 500
    iter = 0
    while norm(t - prev_t) >  EPS and iter < max_iter:
        prev_t = t.copy()
        t -= alpha*grad(x, y, t)
        if iter % 25 == 0:
            print ("Iter", iter)
            c = cost(x, y, t)
            print ("Log f(x) = ", c)
        iter += 1
    return t

# generate some mislabels
for i in range(nT):
    # each onehot y value is shifted over to the next digit
    y_[i] = [0,1,0,0,0,0,0,0,0,0]
    y_[i+500] = [0,0,1,0,0,0,0,0,0,0]
    y_[i+1000] = [0,0,0,1,0,0,0,0,0,0]
    y_[i+1500] = [0,0,0,0,1,0,0,0,0,0]
    y_[i+2000] = [0,0,0,0,0,1,0,0,0,0]
    y_[i+2500] = [0,0,0,0,0,0,1,0,0,0]
    y_[i+3000] = [0,0,0,0,0,0,0,1,0,0]
    y_[i+3500] = [0,0,0,0,0,0,0,0,1,0]
    y_[i+4000] = [0,0,0,0,0,0,0,0,0,1]
    y_[i+4500] = [0,0,0,0,0,0,0,0,0,0]

y_ = array(y_).T

# initial weights, bias is w[0]
w_init = random.normal(0, 0.001, (785, 10))

# learning step
delta = 0.000005
w_log = grad_descent_log(f_log, df_log, x, y_, w_init, delta)
w_lin = grad_descent_lin(f_lin, df_lin, x, y_.T, w_init, delta)

# check accuracy of log
p = calculate_output(w_log, xT).T
correct = 0
incorrect = 0;
for i in range(500):
    if (argmax(p[i]) == argmax(yT[i])):
        correct += 1
    else:
        incorrect += 1
print("Logistic, ", correct / (correct + incorrect))

# check accuracy of linear
p = dot(w_lin[1:].T, xT).T + w_lin[0]
correct = 0.
incorrect = 0.
for i in range(500):
```

```
    if (argmax(p[i]) == argmax(yT[i])):
        correct += 1
    else:
        incorrect += 1
print("Linear: ", correct / (correct + incorrect))
```

Running both the networks with the same randomized initial weights and bias, using the same learning rate and number of iterations of gradient descent leads to a 21% decrease in error rate, showing that logistic regression has much better performance when the data has outliers. Below are the performances for logistic and linear regression:

```
Logistic:  0.858
Linear:  0.82
```

Part 6

Comparison of vectorized backpropagation and computing the gradient individually with respect to speed

For a network with N layers containing K neurons in each layer, we are interested in determining how much faster fully-vectorized backpropagation is compared to computing the gradient with respect to each weight individually, assuming all layers are fully-connected.

Suppose the running time for a higher layer with respect to each neuron is C_{i+1}/K (where C_{i+1} represents the cost of computing layer $i+1$), then to get to the layer immediately below that the running time will be $K * K = K^2$ for all neurons. Since we are interested in the time complexity required to compute the gradient with respect to each weight individually, we must sum up the time required to get to the bottom most level of the network from each layer of weights, written as:

$$K^2 + K^3 + ... + K^{N-1} + K^N \tag{13}$$

for a fully-connected network with N layers of K neurons each, resulting in a time complexity of $O(K^N)$.

In vectorized backpropagation, the gradient of the cost function with respect to a weight for each layer k is:

$$\frac{\partial C_k}{\partial w_{ijk}} = \frac{\partial C_k}{\partial x_{ik}} \cdot \frac{\partial x_{ik}}{\partial w_{ijk}} \tag{14}$$

Thus, the time required to compute each layer is $K * K = K^2$ for each entire layer with K neurons, and since the derivative is stored for each layer, the running time would be written as:

$$K^2|_{\text{layer 1}} + K^2|_{\text{layer 2}} + ... + K^2|_{\text{layer N-1}} + K^2|_{\text{layer N}} \tag{15}$$

resulting in a time complexity of $O(NK^2)$ for N layers.

Therefore, the time complexity of vectorized backpropagation in a fully-connected network of N layers with K neurons in each layer is $O(NK^2)$ and the time complexity of computing the gradient with respect to each weight individually for the same network is $O(K^N)$, and as such, for networks with large N the former would be much faster than the latter. *Assumptions made: time complexity of matrix multiplication is negligible, or on the order of O(1) (nearly constant regardless of sizes of matrices), compared to the overall scale of time complexity for computing the weights with respect to each neuron in a layer with respect to all the layers in the network*

Part 7

Single Hidden-layer Neural Network for Face Recognition

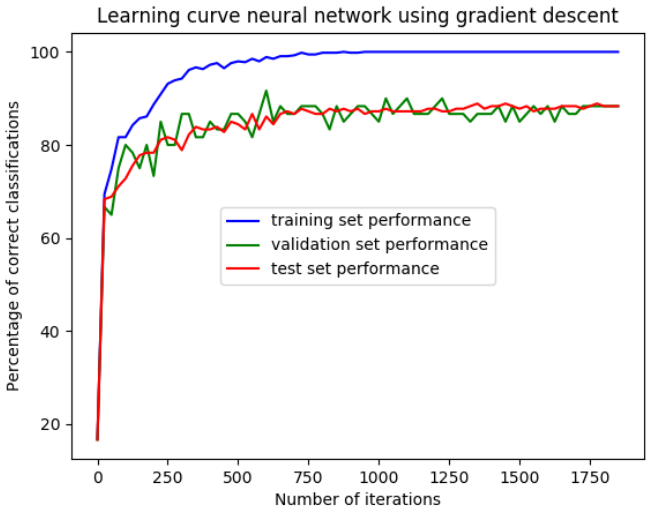
The base code for a single hidden layer neural network was adapted for face recognition. The input images are downloaded using the same script as Project 1 with the addition of `SHA256` checksum verification, then cropped and resized and grayscaled to 32 x 32 pixels. These are then flattened and added to a dictionary with a training set, validation set and test set for each actors/actresses. There are 90 images in the training set, 10 images in the validation set and 30 images in the test set.

The weights are initialized using tf.truncated_normal with a standard deviation of 0.01 and a random seed of 1234, which are small enough and random enough for our purposes. The network consists of 300 hidden units and 6 output units corresponding to the 6 actors/actresses.

The tanh activation function was used for the hidden layer and identity was used for the output layer. After experimenting with relu and sigmoid on the hidden and output layers, tanh was found to have the best performance on the test set overall.

We also used L2 regularization with $\lambda = 0.0001$, this had a small but noticeable improvement on performance.

Finally, the output layer was fed through softmax and the cross entropy cost function was minimized using tensorflow's `AdamOptimizer` with a learning rate of 0.0004 and over 1850 iterations. Every 25 iterations the performance on the training, validation and test sets are printed and logged in an array, which is shown below.



The final performance on the test set is listed below:

```
i= 1850
Test: 0.883333
Validation: 0.883333
Train: 1.0
Penalty: 0.025702
Final performance on test set:  0.883333
```

Part 8

Regularization
Lambda (λ) penalty is a regularization method that can be employed to prevent overfitting with an over-trained network. A particular scenario where using regularization is necessary in the context of faces is when the training set contains many outliers, included faces turned significantly to one side, have big obstructions (shadows, hair, hands, glasses, etc.) covering a significant portion of the face, or have exaggerated non-standard facial obstructions. The following is a sample of such outlier images that were chosen for this special scenario:

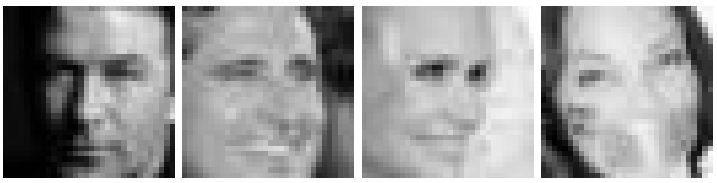


Figure 13: A selection of outlier faces in the dataset

To test regularization, a special data set containing 50 images per actor, 25 of which were selected based on the outlier criteria outlined previously in order to generate a data set that has many non standard images and needs regularization to improve performance. Overall, the final performance of this dataset was seen to have improved from 75.6% (when $\lambda = 0$) to 82.2% (when $\lambda = 0.065$). Listed below are samples of the final performances with various lambda values:

```
---special scenario---
Lambda Value:  0.0
Hidden Units:  300
Final performance on test set: 0.755556

Lambda Value:  0.0015
Hidden Units:  300
Final performance on test set: 0.763333

Lambda Value:  0.003
Hidden Units:  300
Final performance on test set: 0.763333

Lambda Value:  0.0045
Hidden Units:  300
Final performance on test set: 0.777778

Lambda Value:  0.0065
Hidden Units:  300
Final performance on test set: 0.822222
```

Regularization by changing the lambda penalty value was also tested on the general training set without special selection, with minimal improvements observed. Performances fluctuate from 87.8% to 90.6% for λ between 0.0001 to 0.1, with the best performance observed occuring at $\lambda = 0.0075$. Listed below are samples of the final performances with various lambda values:

```
---unselected training set---
Lambda Value:  0.0
Hidden Units:  300
Final performance on test set:  0.877778

Lambda Value:  0.0015
Hidden Units:  300
Final performance on test set:  0.9

Lambda Value:  0.003
Hidden Units:  300
Final performance on test set:  0.888889

Lambda Value:  0.0075
Hidden Units:  300
Final performance on test set:  0.905556
```

Part 9

Hidden Unit Visualization

To visualize the weights of the hidden units, we select the most influential neurons for each actor/actress by performing argmax on the output layer's weights W1 after training. This determines which of the hidden units are most representative of each actor/actresses. The code to generate those hidden units as well as the visualizations themselves for Baldwin and Drescher are given below.

```
weights = W0.eval(session=sess)
bias = b0.eval(session=sess)
w = weights.reshape((32, 32, 300))
w_out = W1.eval(session=sess)

# select most influential neuron for each actor
for i in range(6):
    name = act[i]
    neuron = argmax(w_out.T[i])
    im = w[:,:,neuron] + bias[neuron]
    imsave(name + "neuron" + str(neuron) + ".jpg", im)
```

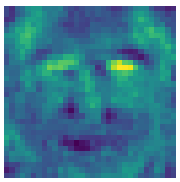


Figure 14: Baldwin: Neuron 168

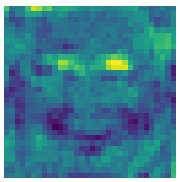


Figure 15: Drescher: Neuron 207

Part 10

AlexNet for Face Recognition

To use AlexNet for face recognition, first the face images had to be the same dimensions as the inputs of AlexNet. The images were redownloaded and this time resized to 227 by 227 and not grayscaled. The blue and red channels were also reversed to adhere to input format. Any images with only one channel for color (not in rbg format) was removed. Like part7, there are 90 images in the training set, 10 in the validation set and 30 in the test set for each actor/actress. The activation values of the conv4 layer are then extracted as such:

```
train_cv4 = sess.run(conv4, feed_dict={x:x_train})
valid_cv4 = sess.run(conv4, feed_dict={x:x_valid})
test_cv4 = sess.run(conv4, feed_dict={x:x_test})
cv4_out = tf.placeholder(tf.float32, (None, 13, 13, 384))
```

Using the activations of conv4 as inputs, a fully-connected neural network was constructed. The input weights has the same dimensions as the outputs of the conv4 layer: (13, 13, 384). Training this network is very slow and using 300 hidden units is too computationally intensive, thus the number of hidden units was reduced to 30. The same truncated normal distribution with standard deviation of 0.01 was used to generate random weights and bias for both layers.

ReLU was used instead of tanh in this part along with the same $\lambda = 0.0001$, since it led to better performance overall.

The AdamOptimizer was only run for 300 iterations this time. Although the performance flattened out at around 50 iterations, we wanted to make sure of the optimality of the accuracy by running for longer. Using only the fully-connected layer on top of the conv4 layer, the final performance on the test set of 180 images reached 95.56%, a 62% reduction in error rate. The learning curve is shown below.

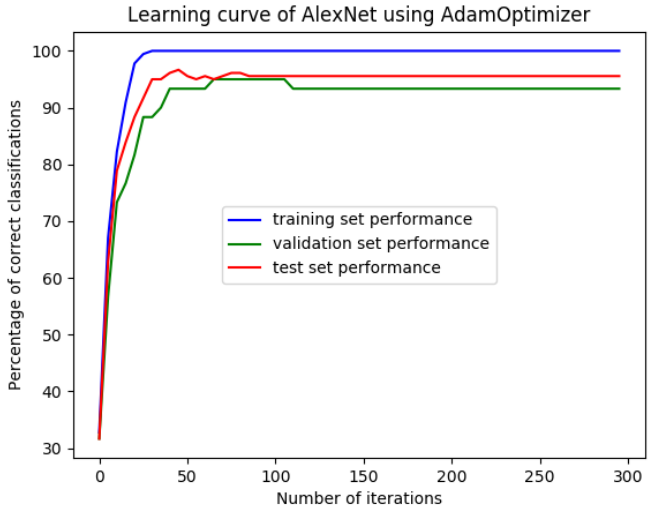


Figure 16

Part 11

Visualization of the Hidden Units

We can visualize what the hidden units are doing in a variety of ways. One way is to backpropagate the output using the trained weights and biases to visualize the input. Some of the interesting filter outputs of AlexNet are visualized below.

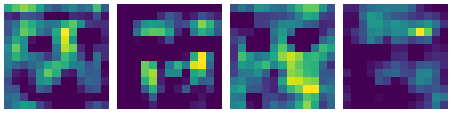


Figure 17: A visualization of some of the more interesting hidden units that were trained on top of the AlexNet conv4 features.