CSC411: Assignment #2

Due on Monday, February 23, 2018

Chawla Dhruv, Maham Shama Saila

February 23, 2018

Part 1

Dataset Description

The dataset consists of a set of images with each image representing a digit from 0 to 9.

Each image is 28x28 and has a black background with the digit handwritten in white.

Some of the images are straightforward to analyze and decipher while some are more tricky to ascertain what the handwriting is trying to represent (consider the 7th image from the left of the digit 5).

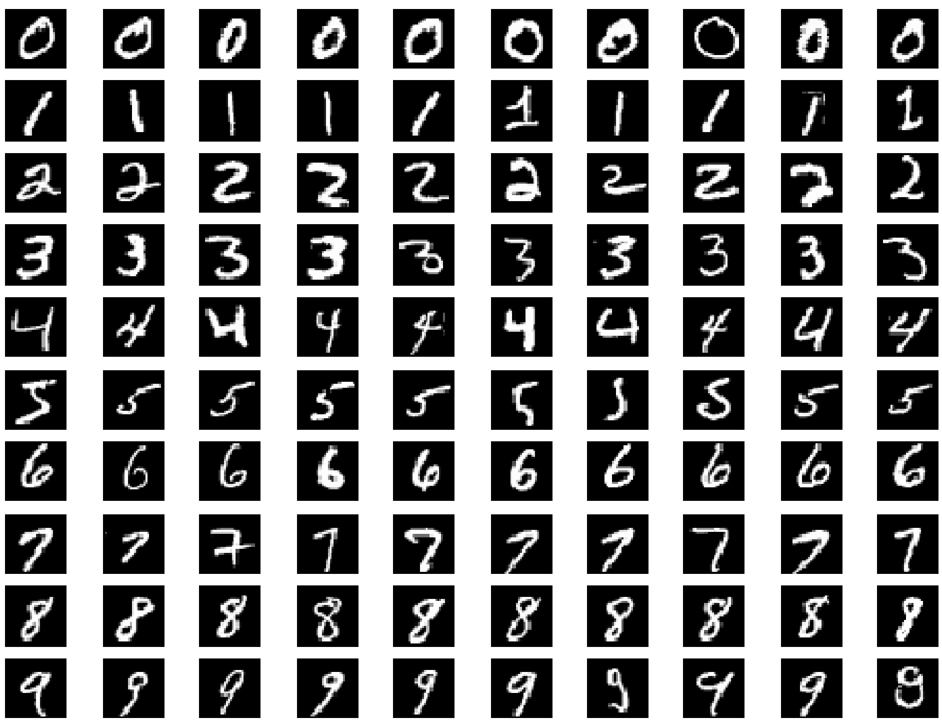


Figure 1: Images from the MNIST dataset

The image is produced from `plot_each_digit()` in `plot.py`.

Part 2

Compute Simple Neural Network

The function for computing simple neural network (no hidden layers) is in `mnist_handout.py` and is reproduced here.

```
def compute_simple_network(x, W, b):
    '''Compute a simple network (with no hidden layers)
    '''
    o = np.dot(W.T, x) + b
    return softmax(o)
```

`compute_simple_network(x, W, b)` returns the output layer of the neural network.

Part 3

Cost Function: Sum of negative log-probabilities of all training cases

Part 3(a)

Compute $\frac{\partial C}{\partial w_{ij}}$ (gradient of cost function with respect to a single weight)

For one training case, the cost function is (from slide 6 of One-Hot Encoding Lecture):

$$C = -\sum_j y_j log p_j \tag{1}$$

For M training examples, the cost functions gets modified to:

$$C = -\sum_{m=1}^{M} \sum_j y_j log p_j \tag{2}$$

We also know that (from slide 7 of One-Hot Encoding Lecture),

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

The partial derivative will be the following:

$$\frac{\partial p_i}{\partial o_j} = \begin{cases} p_i(1-p_i) & i = j \\ -p_i p_j & i \neq j \end{cases} \tag{4}$$

Computing the cost function with respect to the output:

$$\frac{\partial C}{\partial o_i} = \sum_j \frac{\partial C}{\partial p_j} \frac{\partial p_j}{\partial o_i} \tag{5}$$

$$= \frac{\partial C}{\partial p_i} \frac{\partial p_i}{\partial o_i} - \sum_{j \neq i} \frac{\partial C}{\partial p_j} \frac{\partial p_j}{\partial o_i} \tag{6}$$

$$= -y_i(1-p_i) + \sum_{j \neq i} y_j p_j \tag{7}$$

$$= -y_i + p_i \sum_{j \neq i} y_j \tag{8}$$

$$= p_i - y_i \tag{9}$$

Computing the O with respect to weight

$$o_i = \sum_j w_{ji} x_j + b \tag{10}$$

$$\frac{\partial o_i}{\partial w_{ij}} = \sum_j x_j \tag{11}$$

Computing the cost function with respect to the weight

$$\frac{\partial C}{\partial w_{ij}} = \sum_j \frac{\partial C}{\partial o_i} \frac{\partial o_i}{\partial w_{ij}} \tag{12}$$

we will get

$$\frac{\partial C}{\partial w_{ij}} = x_j(p_i - y_i) \tag{13}$$

Part 3(b)

Compute Gradient of Cost Function with respect to Weight

The function for computing the gradient with respect to weight is in `mnist_handout.py` and is reproduced here.

```
def gradient_simple_network_w(x, W, b, y):
    p = compute_simple_network(x, W, b)

    p_minus_y = np.subtract(p, y)
    gradient_mat = np.matmul(x, p_minus_y.T)

    return gradient_mat
```

Compute Gradient of Cost Function with respect to Bias

The function for computing the gradient with respect to bias is in `mnist_handout.py` and is reproduced here.

```
def gradient_simple_network_b(x, W, b, y):
    p = compute_simple_network(x, W, b)

    return np.sum((p - y), axis=1).reshape((10, 1))
```

Finite difference check

The gradient with respect to weight at several different coordinates was computed and displayed along with the finite difference values at the same coordinates.

As can be seen, the gradient is accurate up to 4-5 decimal places.

```
Index: 245.0, 9.0
Actual Gradient Value:    0.0508788452877
Finite Difference Value: 0.0508927423062

Index: 241.0, 4.0
Actual Gradient Value:    0.030247790924
Finite Difference Value: 0.030294025573

Index: 686.0, 4.0
Actual Gradient Value:    0.0288074199276
Finite Difference Value: 0.0288493552549

Index: 571.0, 7.0
Actual Gradient Value:    -0.0403508424702
Finite Difference Value: -0.040346477939

Index: 180.0, 3.0
Actual Gradient Value:    -0.0110842325284
Finite Difference Value: -0.0110820650203
```

```
Index: 408.0, 6.0
Actual Gradient Value:   0.00594562385543
Finite Difference Value: 0.00595885681642

Index: 216.0, 6.0
Actual Gradient Value:   0.0111480447289
Finite Difference Value: 0.0111945692614
```

This is done in `check_grad_w(x, W, b, y, h, coords)` in `mnist_handout.py`.

Part 4

Train the neural network using Gradient Descent

The code to train the neural net is included in `mnist_handout.py`.

```
def train_nn(f, df_W, df_b, x_train, y_train, x_test, y_test, init_W, init_b, alpha, max_iter =
                                                        2000):
    x = x_train
    y = y_train

    epoch, train_perf, test_perf = [], [], []

    EPS = 1e-10
    prev_W = init_W - 10 * EPS
    prev_b = init_b - 10 * EPS
    W = init_W.copy()
    b = init_b.copy()
    itr = 0

    while norm(W - prev_W) > EPS and norm(b - prev_b) > EPS and itr < max_iter:
        prev_W = W.copy()
        prev_b = b.copy()

        W -= alpha * df_W(x, W, b, y)
        b -= alpha * df_b(x, W, b, y)

        if itr % 500 == 0 or itr == max_iter - 1:
            epoch_i = itr
            train_perf_i = performance(x_train, W, b, y_train)
            test_perf_i = performance(x_test, W, b, y_test)

            epoch.append(epoch_i)
            train_perf.append(train_perf_i)
            test_perf.append(test_perf_i)

            print("Epoch: " + str(epoch_i))
            print("Training Performance:   " + str(train_perf_i) + "%")
            print("Testing Performance:    " + str(test_perf_i) + "%\n")

        itr += 1

    return W, b, epoch, train_perf, test_perf
```

The following optimization procedue was followed:

1. Weights were initialized using the saved weights from `snapshot50.pkl` file. These performed better than randomly initialized weights. I suspect these are pre-trained weights that give a better starting point than random weights.

2. Learning rate was set to 1^{-5} and iterations to 2000. This was done with experiments and trying to see which combination gave the lowest cost on the testing set.

The performance of the NN on training set is 93.24% and on the testing set is 92.57%. This is good for a neural net with no hidden layers on the MNIST dataset according to literature available on the web.

Performance of the learning curves can be seen in Figure 2.

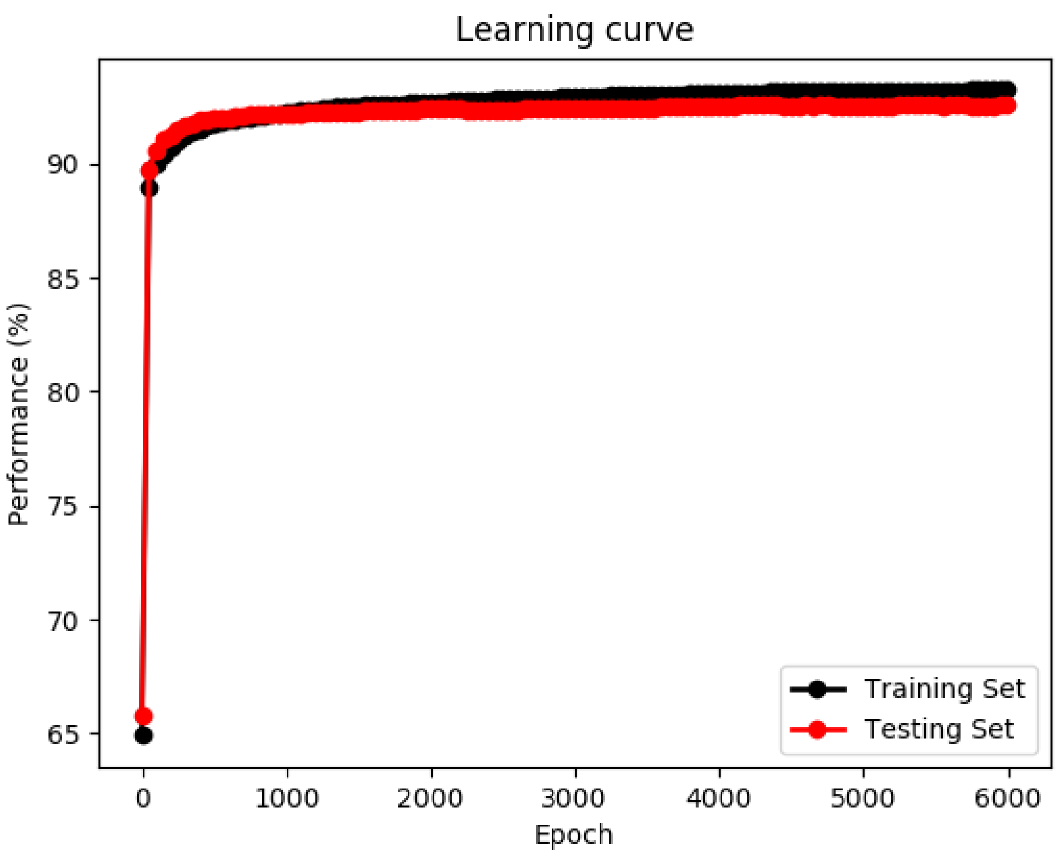
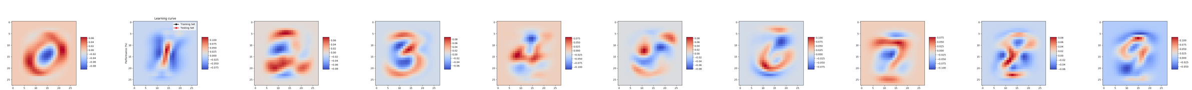


Figure 2: Learning curve for neural network using Gradient Descent

The weights corresponding to each digit are plotted below:



(a) 0 (b) 1 (c) 2 (d) 3 (e) 4 (f) 5 (g) 6 (h) 7 (i) 8 (j) 9

Figure 3: Displaying weights of each of the digits

The images can be reproduced by calling `part4()` in `digits.py`.

Part 5

Train the neural network using Gradient Descent with Momentum

The code to train the neural net is included in `mnist_handout.py`.

```
def train_nn_M(f, df_W, df_b, x_train, y_train, x_test, y_test, init_W, init_b, alpha, gamma = 0.
                                                    9, max_iter = 6000):
    x = x_train
    y = y_train

    epoch, train_perf, test_perf = [], [], []

    EPS = 1e-10
    prev_W = init_W - 10 * EPS
    prev_b = init_b - 10 * EPS
    W = init_W.copy()
    b = init_b.copy()
    itr = 0
    v_W = 0
    v_b = 0

    while norm(W - prev_W) > EPS and norm(b - prev_b) > EPS and itr < max_iter:
        prev_W = W.copy()
        prev_b = b.copy()

        #update velocities
        v_W = gamma * v_W + alpha * df_W(x,W,b,y)
        v_b = gamma * v_b + alpha * df_b(x,W,b,y)
        #update parameters with momentum
        W = W - v_W
        b = b - v_b

        if itr % 50 == 0 or itr == max_iter - 1:
            epoch_i = itr
            train_perf_i = performance(x_train, W, b, y_train)
            test_perf_i = performance(x_test, W, b, y_test)

            epoch.append(epoch_i)
            train_perf.append(train_perf_i)
            test_perf.append(test_perf_i)

            print("Epoch: " + str(epoch_i))
            print("Training Performance:   " + str(train_perf_i) + "%")
            print("Testing Performance:    " + str(test_perf_i) + "%\n")

        itr += 1

    return W, b, epoch, train_perf, test_perf
```

Performance of the learning curves can be seen in Figure 4.

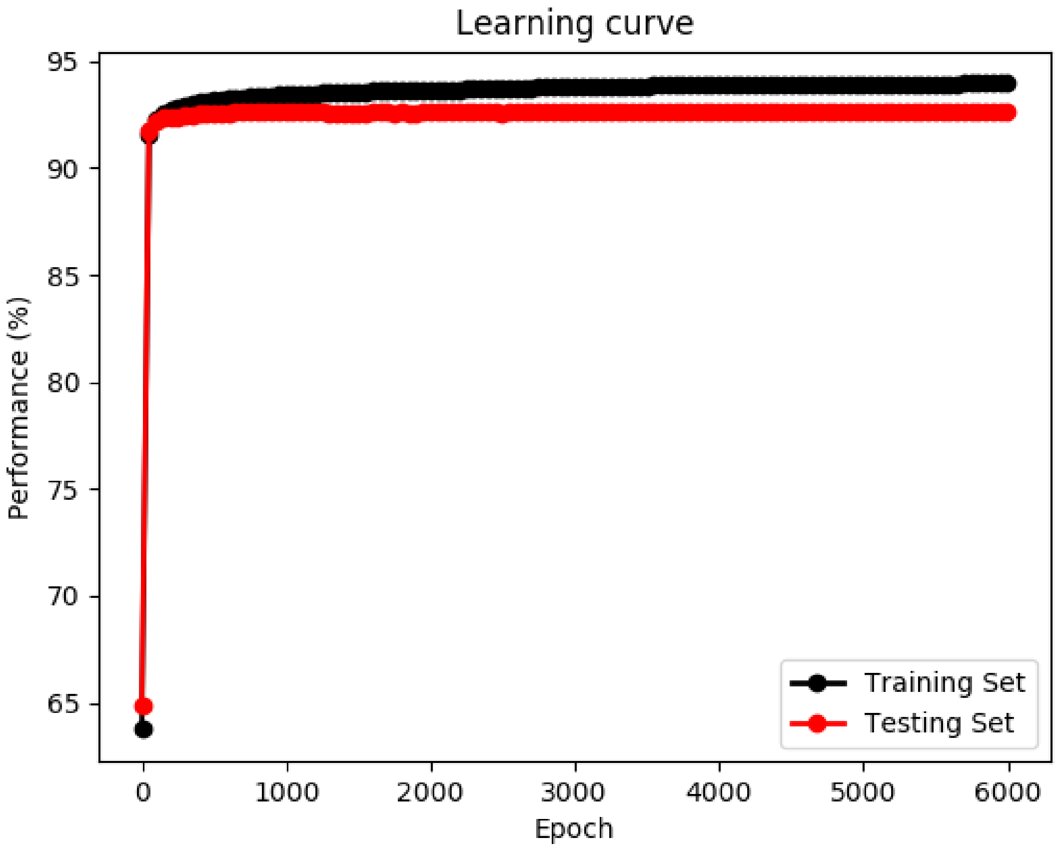


Figure 4: Learning curve for neural network using Gradient Descent with Momentum

It can be observed that with momentum, we reach the peak performance faster than without. In addition, the performance of the algorithm (in terms of accuracy on the test and training set are not sacrificed).

Part 6

Part 7

Part 8

Part 9

Part 10

Modify AlexNet for FaceScrub face classification

Extracting activation values from MyAlexNet

The values of activation of the AlexNet Conv4 layer were extracted by modifying the features of `MyAlexNet`. We chose Conv4 as it the convolutional layer closest to the output and thus is more likely to have high-level features better for detecting faces.

1. Features after Conv4 layer were commented out

2. Classifer call was removed in `forward(x)` method so that it returns the activation values instead of classification output

Final (modified) code for `MyAlexNet` is reproduced here.

```
class MyAlexNet(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, 6]
        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(MyAlexNet, 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),
            #nn.MaxPool2d(kernel_size=3, stride=2),
        )
        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),
            nn.Linear(4096, num_classes),
        )

        self.load_weights()
```

```
    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 13 * 13)
        # x = self.classifier(x)
        return x
```

This code can be found in `myalexnet.py`

The activation value for an image can be found out by calling `model.forward(x)` and storing it in a `numpy` array.

```
x = Variable(torch.from_numpy(img), requires_grad=False).type(torch.FloatTensor)
activation = model.forward(x).data.numpy()
```

The image array is formed by loading in an RGB image of dimensions 227x227, normalizing the values to be in range [-1, 1] and reshaping it in array of shape (1, 3, 227, 227).

Getting activations for train and test set

Similar to Part 8, each actor has 20 images in test set and the rest in training set. The train-test split is in `build_sets_part10(actor)` in `faces.py`.

The images are obtained from the FaceScrub dataset using the function, `get_and_crop_images(act, 227)` in `faces.py`. The images are checked for SHA-256 checksums and remaining bad images are filtered out using `remove_bad_images(227)`.

Note: The images are stored in cropped227/ folder which can be obtained by unzipping cropped227.zip. Alternatively, comment out the lines in part10() that call for get_and_crop_images(act, 227) and remove_bad_images(227)

The activations for each image is obtained and stored in `train_activation` and `test_activation` `numpy` arrays respectively. This is done in `alexNetFaceScrub()` in `myalexnet.py`. The process is done 100 images at a time due to CDF machine's memory constraints and will have to be modified should there be more than 600 images in the training set or more than 120 in the test set.

Plugging the activations into the new neural net

Once we have the activation values, we plug them into a neural net which is similar to the one used in Part 8 except the input dimension has been modified to fit the activation layer size.

Similar to part 8, the parameters used to gain better performance were:

1. Using ReLU for the hidden layer activation function. ReLU gave better performace than Tanh or Sigmoid.
2. Using 600 neurons in the hidden layer. The number of neurons were increased until they were not bringing the cost down for corresponding epochs (perhaps due to overfitting).
3. Using Adam optimizer. Other optimizers were tried (SGD, AdaGrad) but Adam gave the best performance.
4. Using learning rate of 1^{-4} and 400 iterations. Using more iterations increased cost (due to overfitting). The learning rate was low enough to achieve good performance while not having a lot of iterations.

The labels were stored as one-hot encoding for the actors. The new neural net is in `alexNetFaceScrub()` in `myalexnet.py`.

Performance

The final model had a 92.5% accuracy on the test set. This cut down the error rate in Part 8 by more than 30%. The performance of the model on training and testing set along with the epochs is shown here.

```
Epoch: 0
Training Set Performance: 25.2873563218%
Testing Set Performance:  21.6666666667%

Epoch: 50
Training Set Performance: 95.2380952381%
Testing Set Performance:  86.6666666667%

Epoch: 100
Training Set Performance: 100.0%
Testing Set Performance:  90.0%

Epoch: 150
Training Set Performance: 100.0%
Testing Set Performance:  90.8333333333%

Epoch: 200
Training Set Performance: 100.0%
Testing Set Performance:  91.6666666667%

Epoch: 250
Training Set Performance: 100.0%
Testing Set Performance:  92.5%

Epoch: 300
Training Set Performance: 100.0%
Testing Set Performance:  92.5%

Epoch: 350
Training Set Performance: 100.0%
Testing Set Performance:  92.5%

Epoch: 399
Training Set Performance: 100.0%
Testing Set Performance:  92.5%
```