CS289A HW4

David Winer

October 27, 2016

Problem 1

Part A: W update

Notation: Throughout this section, superscripts will refer to the layer of neural network. So, for example, $x_j^{(2)}$ would refer to the j^{th} unit in the second layer (output layer) of the neural network. Similarly, $x_j^{(1)}$ would refer to the j^{th} unit in the hidden layer and $x_j^{(0)}$ would refer to the j^{th} unit in the input layer.

We are given $W = n_{out}$-by-$n_{hid} + 1$. Thus, W_{ij} is the weight connecting the j^{th} unit in the hidden layer to the i^{th} unit in the output layer. Our stochastic gradient descent update rule will take on the following form:

$$W_{ij} \leftarrow W_{ij} - \alpha \frac{\partial J}{\partial W_{ij}}$$

Where α is the learning rate and J is the cross-entropy loss. We can decompose the cross-entropy derivative:

$$\begin{aligned}\frac{\partial J}{\partial W_{ij}} &= \frac{\partial J}{\partial z_i^{(1)}} \frac{\partial z_i^{(1)}}{\partial W_{ij}} \\ &= \delta_i \times x_j^{(1)}\end{aligned}$$

Note that:

$$z_i^{(1)} = \sum_{j=0}^{n_{hid}+1} W_{ij} x_j^{(1)}$$

We then just need to derive the expression for δ_i:

$$\begin{aligned}\delta_i &= \frac{\partial}{\partial z_i^{(1)}} \left[-\sum_{k=0}^{n_{out}} y_k \ln g_k(\mathbf{z^{(1)}}) \right] \quad \text{where } g \text{ is the softmax function} \\ &= -\sum_{k=0}^{n_{out}} \frac{y_k}{g_k(\mathbf{z^{(1)}})} \frac{\partial g_k(\mathbf{z^{(1)}})}{\partial z_i} \\ &= \sum_{k \neq i} y_k g_i(\mathbf{z^{(1)}}) - y_i(1 - g_i(\mathbf{z^{(1)}})) \\ &= g_i(\mathbf{z^{(1)}}) \sum_{k \neq i} y_k - y_i(1 - g_i(\mathbf{z^{(1)}})) \\ &= \left[x_i^{(2)} \sum_{k \neq i} y_k \right] - y_i(1 - x_i^{(2)}) \\ &= x_i^{(2)} - y_i\end{aligned}$$

Thus, we have the consolidated update rule:

$$W_{ij} \leftarrow W_{ij} - \alpha \left[\delta_i \times x_j^{(1)} \right]$$

Where:

$$\delta_i = x_i^{(2)} - y_i$$

Part B: V update

We are given $V = n_{hid}$-by-$n_{in} + 1$. Thus, V_{ij} is the weight connecting the j^{th} unit in the input layer to the i^{th} unit in the hidden layer. Our stochastic gradient descent update rule will take on the following form:

$$V_{ij} \leftarrow V_{ij} - \alpha \frac{\partial J}{\partial V_{ij}}$$

Where α is the learning rate and J is the cross-entropy loss. Similar to the approach in Part A, we can decompose the cross-entropy derivative:

$$\begin{aligned} \frac{\partial J}{\partial V_{ij}} &= \frac{\partial J}{\partial z_i^{(0)}} \frac{\partial z_i^{(0)}}{\partial V_{ij}} \\ &= \gamma_i \times x_j^{(0)} \end{aligned}$$

Note that:

$$z_i^{(0)} = \sum_{j=0}^{n_{in}+1} V_{ij} x_j^{(0)}$$

We then just need to derive the expression for γ_i:

$$\begin{aligned} \gamma_i &= \sum_{k=0}^{n_{out}} \frac{\partial J}{\partial z_k^{(1)}} \times \frac{\partial z_k^{(1)}}{\partial x_i^{(1)}} \times \frac{\partial x_i^{(0)}}{\partial z_i^{(0)}} \\ &= \sum_{k=0}^{n_{out}} \delta_k \times W_{ki} \times g'(z_i^{(0)}) \\ &= g'(z_i^{(0)}) \sum_{k=0}^{n_{out}} \delta_k \times W_{ki} \end{aligned}$$

Note that here g is the activation function for the hidden layer, $g(z) = \max(0, z)$. For this function:

$$g'(x) = \begin{cases} 0 \text{ for } x \leq 0 \\ 1 \text{ for } x > 0 \end{cases}$$

Thus we have the following consolidated update rule:

$$V_{ij} \leftarrow V_{ij} - \alpha \left[\gamma_i \times x_j^{(0)} \right]$$

Where:

$$\gamma_i = \begin{cases} \sum_{k=0}^{n_{out}} \delta_k \times W_{ki} & \text{when } z_i^{(0)} > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$z_i^{(0)} = \sum_{j=0}^{n_{in}+1} V_{ij} x_j^{(0)}$$

Note that δ_k is the same as defined in Part A.

Problem 2

See code snippet for implementation.

Problem 3

Data normalization and weight initialization

I normalized my data by, for each pixel, subtracting the global mean of the data (across all examples/features) and dividing by 255.

I initialized my W and V vectors by choosing from the univariate Gaussian distribution with mean 0 and standard deviation 0.01.

Parameters, initialization, stopping criteria, and training time

The main two parameters I tuned were α, the learning rate, and β, my rate of decay. I ultimately chose $\alpha = 0.01$ and $\beta = 0.9$, with β applied after every two epochs of stochastic gradient descent. I chose these parameters by constructing a grid of (α, β) pairs and seeing which produced the best test accuracy after 4 epochs of SGD. I tried $\alpha = 1, 0.1, 0.01$ and $\beta = 0.9, 0.7, 0.5$.

I chose to stop the gradient descent algorithm after four epochs (200,000 iterations on training data). I made this choice by examining the training loss and accuracy after every 1000 iterations as I was training. I saw that after five epochs, I was getting approximately 99% test accuracy, which felt sufficient. In total, training took 6 minutes.

Final accuracy and training time

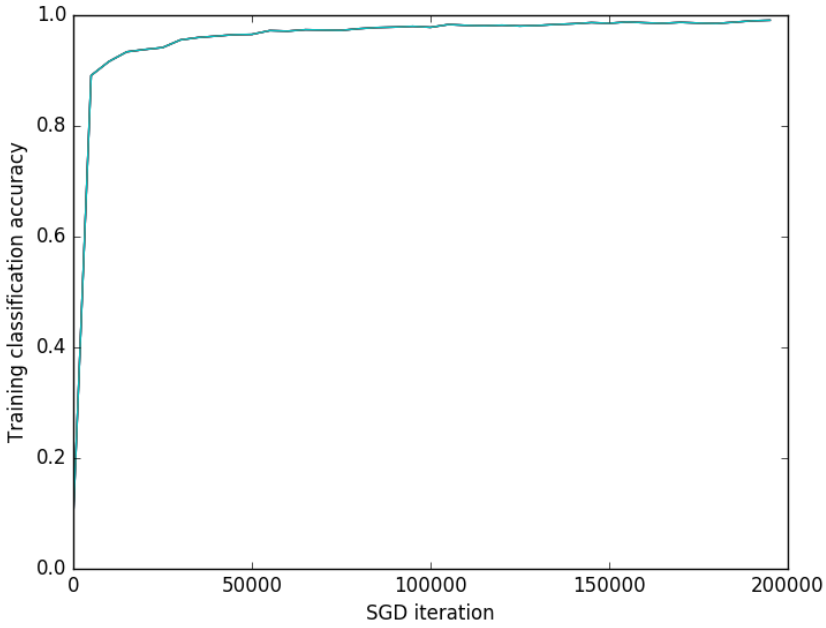
My training accuracy was 99.08%. My validation accuracy was 97.3%.

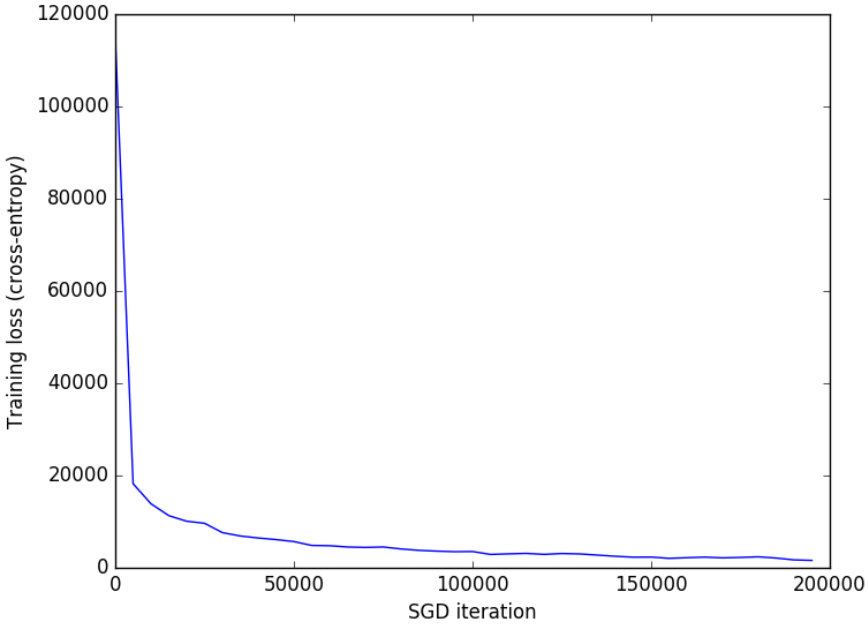
Plots

Below are plots of my training loss and classification accuracy on the training set. Note that the total training loss was computed as:

$$\sum_{i=1}^{n_{examples}} J_i$$

where J_i is the cross-entropy loss on one example. Both training loss and accuracy were calculated after every 5000 iterations of SGD.


1.0
0.8
0.6
0.4
0.2
0.0
Training classification accuracy
0
50000
100000
150000
200000
SGD iteration


120000
100000
80000
60000
40000
20000
0
Training loss (cross-entropy)
0
50000
100000
150000
200000
SGD iteration

Kaggle score

My Kaggle score was 0.9750.

Code snippet

```
from mnist import MNIST
import numpy as np
import sklearn.metrics as metrics
import sklearn
import matplotlib.pyplot as plt
import csv

N_HID = 200
N_OUT = 10

def train_neural_network(X, labels, step_size, decay, num_iter, outfile):

    mu, sigma = 0, 0.01
    n_dim = X.shape[1]
    n_ex = X.shape[0]

    W = np.random.normal(mu, sigma, (N_OUT, N_HID + 1))
    V = np.random.normal(mu, sigma, (N_HID, n_dim + 1))

    step = step_size

    iterations = []
    losses = []
    accuracies = []

    for i in range(num_iter):
        if (i % 100000 == 0):
            step = step * decay
        ex_num = i % n_ex
        Y = np.matrix(labels[ex_num, :])

        # Pull X_0 layer and append one for bias term
        X_0 = np.matrix(np.insert(X[ex_num, :], 0, 1))

        # Forward propagate from input to hidden layer, again adding
            bias term
        Z_0 = np.dot(V, X_0.T)
        X_1 = np.maximum(Z_0, 0) # ReLU
        X_1 = np.insert(X_1.T, 0, 1)

        # Forward propagate from hidden layer to output layer
        Z_1 = np.dot(W, X_1.T).T
        X_2 = np.apply_along_axis(softmax, 1, Z_1, np.arange(10))

        # W grad calculations
        y_sums = np.tile(1, 10) - Y
        deltas = np.multiply(X_2, y_sums) - np.multiply(Y, (1 - X_2))
```

```
            grad_W = np.dot(deltas.T, X_1)

            # V grad calculations
            W_trim = W[:, 1:] # don't pass bias term along to next layer down
            gammas = np.dot(W_trim.T, deltas.T)
            multiplier = np.maximum(Z_0, 0)/Z_0
            gammas = np.multiply(multiplier, gammas)
            grad_V = np.dot(gammas, X_0)

            # W and V updates
            W = W - step * grad_W
            V = V - step * grad_V

            if (i % 1000 == 0):
                print("Iteration {0}".format(i))

            if (i % 5000 == 0):
                pred_labels = predict_neural_network(X, V, W)
                true_labels = np.argmax(labels, axis = 1)
                accuracy_score = metrics.accuracy_score(true_labels,
                    pred_labels)
                print("Train accuracy: {0}\n".format(accuracy_score))
                outfile.write("Train accuracy: {0}\n".format(accuracy_score))
                training_loss = 0
                for k in range(n_ex):
                    myX = np.matrix(np.insert(X[k, :], 0, 1))
                    myY = np.matrix(labels[k, :])
                    training_loss += x_entropy_loss(myX, myY, W, V)
                print("Training loss is {0}".format(training_loss))
                iterations = np.append(iterations, i)
                losses = np.append(losses, training_loss)
                accuracies = np.append(accuracies, accuracy_score)

        np.save("W_save_1.npy", W)
        np.save("V_save_1.npy", V)
        np.save("iterations_1.npy", iterations)
        np.save("losses_1.npy", losses)
        np.save("accuracies_1.npy", accuracies)
        return V, W

    def x_entropy_loss(X, Y, W, V):
        # Input X already has bias term appended
        # Forward propagate from input layer to hidden layer
        Z_0 = np.dot(V, X.T)
        X_1 = np.maximum(Z_0, 0) # ReLU
        X_1 = np.insert(X_1.T, 0, 1)

        # Forward propagate from hidden layer to output layer
        Z_1 = np.dot(W, X_1.T).T
        X_2 = np.apply_along_axis(softmax, 1, np.matrix(Z_1), np.arange(10))
```

```
    log_output = np.log(X_2)
    return -1 * np.sum(np.multiply(Y, log_output))

def predict_neural_network(X, V, W):
    n_ex = X.shape[0]

    # Pull X_0 layer and append one for bias term
    X_0 = np.insert(X, 0, np.ones(n_ex), axis=1)
    Z_0 = np.dot(V, X_0.T)

    # Forward propagate from input to hidden layer, again adding bias
        term
    X_1 = np.maximum(Z_0, 0) # ReLU
    X_1 = np.insert(X_1, 0, np.ones(n_ex), axis=0)
    Z_1 = np.dot(W, X_1)

    # Forward propagate from hidden layer to output layer
    X_2 = np.apply_along_axis(softmax, 0, Z_1, np.arange(10))

    # Return highest prediction
    return np.argmax(X_2, axis = 0)

def softmax(z, j):
    # Uses numerical stability trick
    b = np.max(z)
    denom = np.sum(np.exp(z - b))
    num = np.exp(z[j] - b)
    return num/denom

def load_dataset():
    mndata = MNIST('./data/')
    X, labels = map(np.array, mndata.load_training())

    # Shuffle data
    X_shuf, labels_shuf = sklearn.utils.shuffle(X, labels, random_state
        = 40)

    # Split data into training and validation sets
    X_train = X_shuf[0:50000, :]
    labels_train = one_hot(labels_shuf[0:50000])
    X_val = X_shuf[50000:, :]
    labels_val = one_hot(labels_shuf[50000:])

    # The test labels are meaningless,
    # since you're replacing the official MNIST test set with our own
        test set
    X_test, _ = map(np.array, mndata.load_testing())

    # Center and normalize data
```

```
    X_train = standardize(X_train)
    X_val = standardize(X_val)
    X_test = standardize(X_test)

    # Save for later use, which is faster than doing all this
        preprocessing every time
    np.save("X_test.npy", X_test)
    np.save("X_train.npy", X_train)
    np.save("labels_train.npy", labels_train)

    return X_train, labels_train, X_val, labels_val, X_test

def standardize(X):
    global_mean = np.sum(X)/X.size
    return (X - global_mean)/255

def one_hot(labels_train):
    z = np.zeros((labels_train.shape[0], N_OUT))
    for i in range(len(labels_train)):
        digit = labels_train[i]
        z[i, digit] = 1
    return z

X_train, labels_train, X_val, labels_val, X_test = load_dataset()

# These commented out lines were used when I didn't want to reload every
    time
# I ran training
# X_train = np.load("X_train.npy")
# labels_train = np.load("labels_train.npy")

# Tested other alpha and decay values in a grid (see write-up); these
    are the ones I landed on
alphas = [0.01]
decays = [0.9]

outfile = open("ResultsFile9.txt", "w")

for alpha in alphas:
    for decay in decays:
        print("##### alpha = {0}, decay = {1} ######".format(alpha,
            decay))
        outfile.write("##### alpha = {0}, decay = {1}
            ######\n".format(alpha, decay))
        V, W = train_neural_network(X_train, labels_train, alpha, decay,
            200000, outfile)

outfile.close()
predict_neural_network(X_train, V, W)
```

```
# Plotting
iterations = np.load("iterations.npy")
losses = np.load("losses.npy")
accuracies = np.load("accuracies.npy")

fig = plt.figure(0)
plt.plot(iterations, accuracies)
plt.xlabel("SGD iteration")
plt.ylabel("Training classification accuracy")

W = np.load("W_save.npy")
V = np.load("V_save.npy")

# Prediction on validation set
pred_labels_val = predict_neural_network(X_val, V, W)
true_labels_val = np.argmax(labels_val, axis = 1)
accuracy_score = metrics.accuracy_score(true_labels_val, pred_labels_val)
print("Validation accuracy: {0}\n".format(accuracy_score))

# Prediction on test set
pred_labels_test = predict_neural_network(X_test, V, W)

outfile = open('./output-data.csv', 'w')
writer = csv.writer(outfile)
writer.writerow(['Id', 'Category'])
for i in range(len(pred_labels_test)):
    writer.writerow([int(i+1), int(pred_labels_test[i])])
outfile.close()
```