CS 189 –Spring 2016 — Homework 6 Solutions

Utsav Baral, SID 25694452

DUE: February 10th, 2016

Contents

Problem 1.

Derive the stochastic gradient descent updates for all parameters (V and W) for both mean-squared error and cross-entropy error as your loss function given a single data point (x, y). Use tanh activation function for the hidden layer units and the sigmoid function for the output layer units. To do this, you must compute the partial derivative of J with respect to every Vij and Wij.

Mean Squared Error:

We have the following equation for mean squared error: $J = \frac{1}{2}\sum_{k=1}^{n_{out}}(y_k - z_k(x))^2$. Each row in our weight matrices $\mathbf{W}$ and $\mathbf{V}$ corresponds to the weights vector for some output, or intermediary output neuron. For $\mathbf{W}$, row i corresponds to the weights we apply to the hidden neurons to receive output z_i before we apply the sigmoid function of course. And similarly, for $\mathbf{V}$ row i corresponds to weights on vector point x to get output for h_i, before we apply the hyperbolic tangent function. We can probably vectorize this operation in our code for efficiency but for derivation purposes we can look at the gradient for some arbitrary row i of the weight matrices.

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{W_i}} &= \frac{\partial}{\partial \mathbf{W_i}}\left(\frac{1}{2}\sum_{k=1}^{n_{out}}(y_k - z_k(x))^2\right) \\
&= \frac{1}{2}\sum_{k=1}^{n_{out}}\frac{\partial}{\partial \mathbf{W_i}}(y_k - z_k(x))^2 \\
&= -\sum_{k=1}^{n_{out}}(y_k - z_k(x))\frac{\partial z_k}{\partial \mathbf{W_i}} = -(y_i - z_i(x))\frac{\partial z_i}{\partial \mathbf{W_i}} \\
&= -(y_i - z_i)\cdot z_i(1 - z_i)\mathbf{h}
\end{aligned}$$

Now that we have the gradient, the stochastic gradient rule for W is simply as follows: Let $\mathbf{W_i}$ be the weight vector $\in \mathbb{R}^{n_{hid}+1}$ corresponding to output z_i. For some random sample point, we update

$$\boxed{\mathbf{W_i}_{new} \leftarrow \mathbf{W_i}_{old} - \eta \cdot \nabla_{\mathbf{W_i}} J}, \text{ or: } \boxed{\mathbf{W_i}_{new} \leftarrow \mathbf{W_i}_{old} + \eta \cdot (y_i - z_i)\cdot z_i(1 - z_i)\mathbf{h}}$$

For $\mathbf{V_i}$ we have can similarly apply chain rule, a.k.a, back-propagation to find a similar update rule. With the chain rule we have that (*Note $k = n_{out}$ in this context*):

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{V_i}} = \nabla_{\mathbf{V_i}} J = \frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial \mathbf{V_i}} &= \frac{\partial h_i}{\partial \mathbf{V_i}}\sum_{j=1}^{k}\frac{\partial z_j}{\partial h_i}\frac{\partial J}{\partial z_j} \\
&= (1 - \tanh^2(\mathbf{V_i x}))\cdot\sum_{j=1}^{k}\frac{\partial z_j}{\partial h_i}(z_j - y_j) \\
&= \mathbf{x}\cdot(1 - \tanh^2(\mathbf{V_i x}))\cdot\sum_{j=1}^{k}W_{ij}z_j(1 - z_j)(z_j - y_j)
\end{aligned}$$

For some random sample point, we update:

$$\boxed{\mathbf{V_i}_{new} \leftarrow \mathbf{V_i}_{old} - \eta \cdot \mathbf{x}\cdot(1 - \tanh^2(\mathbf{V_i x}))\cdot\sum_{j=1}^{k}W_{ij}z_j(1 - z_j)(z_j - y_j)}$$

Cross-entropy error:

We will approach this problem the same way as with the mean squared error, just changing up the Cost Function. So for W_i we have that:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{W_i}} &= \frac{\partial}{\partial \mathbf{W_i}} \left(-\sum_{k=1}^{n_{out}} [y_k \ln(z_k) + (1-y_k)\ln(1-z_k)] \right) \\
&= -\sum_{k=1}^{n_{out}} \frac{\partial}{\partial \mathbf{W_i}} y_k \ln(z_k) + \frac{\partial}{\partial \mathbf{W_i}} (1-y_k)\ln(1-z_k) \\
&= -\sum_{k=1}^{n_{out}} y_k \frac{\partial}{\partial \mathbf{W_i}} \ln(z_k) + (1-y_k)\frac{\partial}{\partial \mathbf{W_i}} \ln(1-z_k) \\
&= -\sum_{k=1}^{n_{out}} \frac{\partial z_k}{\partial \mathbf{W_i}} \left(\frac{1-y_k}{1-z_k} - \frac{y_k}{z_k} \right) \\
&= \frac{\partial z_i}{\partial \mathbf{W_i}} \left(\frac{1-y_i}{1-z_i} - \frac{y_i}{z_i} \right) = \frac{\partial z_i}{\partial \mathbf{W_i}} \cdot \frac{z_i - y_i}{z_i(1-z_i)} \\
&= z_i(1-z_i)\mathbf{h} \cdot \frac{z_i - y_i}{z_i(1-z_i)} = \mathbf{h}(z_i - y_i)
\end{aligned}$$

Thus the update equation for $\mathbf{W_i}$:

$$\boxed{\mathbf{W_i}_{new} \leftarrow \mathbf{W_i}_{old} - \eta \cdot (z_i - y_i)\mathbf{h}}$$

For V_i, we can reuse many of the computation we have, so we get (*Note k is initially a free variable, but I change it to $k = n_{hid}$ in the end for consistency*):

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{V_i}} &= \frac{\partial}{\partial \mathbf{V_i}} \left(-\sum_{k=1}^{n_{out}} [y_k \ln(z_k) + (1-y_k)\ln(1-z_k)] \right) \\
&= -\sum_{k=1}^{n_{out}} \frac{\partial}{\partial \mathbf{V_i}} y_k \ln(z_k) + \frac{\partial}{\partial \mathbf{V_i}} (1-y_k)\ln(1-z_k) \\
&= -\sum_{k=1}^{n_{out}} y_k \frac{\partial}{\partial \mathbf{V_i}} \ln(z_k) + (1-y_k)\frac{\partial}{\partial \mathbf{V_i}} \ln(1-z_k) \\
&= \sum_{k=1}^{n_{out}} \frac{\partial z_k}{\partial \mathbf{V_i}} \left(\frac{z_k - y_k}{z_k(1-z_k)} \right) = \sum_{k=1}^{n_{out}} \frac{\partial z_k}{\partial h_i} \frac{\partial h_i}{\partial \mathbf{V_i}} \left(\frac{z_k - y_k}{z_k(1-z_k)} \right) \\
&= \mathbf{x}(1 - \tanh^2(\mathbf{V_i x})) \sum_{k=1}^{n_{out}} W_{ki} \cdot z_k(1-z_k) \left(\frac{z_k - y_k}{z_k(1-z_k)} \right) \\
&= \mathbf{x}(1 - \tanh^2(\mathbf{V_i x})) \sum_{k=1}^{n_{out}} W_{ki} \cdot (z_k - y_k)
\end{aligned}$$

Thus the update equation for $\mathbf{V_i}$ is: $\boxed{\mathbf{V_i}_{new} \leftarrow \mathbf{V_i}_{old} - \eta \cdot \mathbf{x}(1 - \tanh^2(\mathbf{V_i x})) \sum_{j=1}^{n_{out}} W_{ji} \cdot (z_j - y_j)}$

Problem 2.

Train this multi-layer neural network on full training data using stochastic gradient descent. Predict the labels to the test data and submit your results to Kaggle. The paramaters, and the corresponding values I used, for my neural network were:

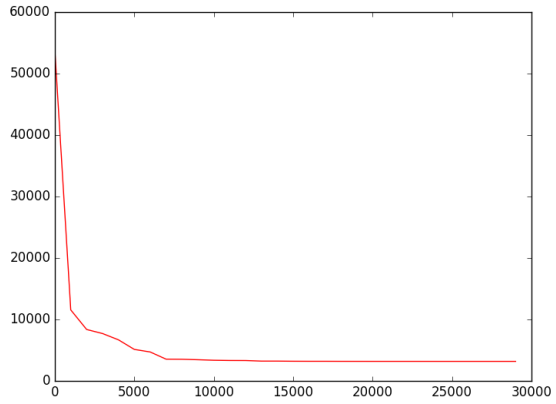
- **Hyperparamaters:** # **of** $\mathbf{n_{hidd}}$ = 100; **batch size** = 10; **learn_rate** = .1;**loss_func** = "cross_entropy"
- With these paramaters, I got a training accuracy of **0.938**, and a validation accuracy of **0.924**.
- **Total Running Time** was around **10 minutes** for **6 epochs**.
- I found that **Cross Entropy Loss** function **outperformed** the **Mean Squared Error Loss** function for every variation of hyperparamters, perhaps because the C.E function had logarithm terms and thus took on much smaller values than the M.S.W which the Neural Net seemed to like.

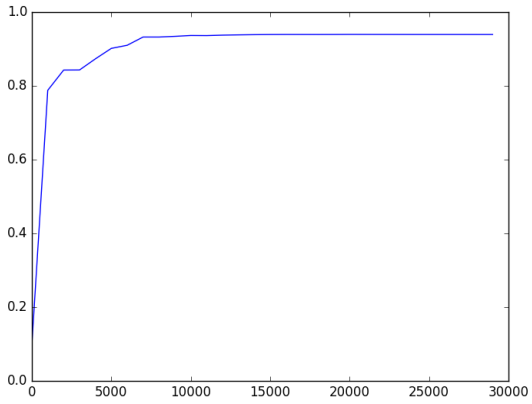
Additional/Extra Things:

- Tweaking the number of **hidden neurons** from 200 down to **100** significantly boosted my accuracy rate ($\sim 5\%$)
- I added **mini-batching** for my gradient descent algorithm, which helped a lot with **batch size = 10**

KAGGLE SCORE: 0.93720

Here are some **plots** of the training process, the first red one is of the loss function and the second blue one is of the training accuracy. Both plots are with respect to every 1000 iterations:





APPENDIX A: CODE LISTINGS

NeuralNetwork.py

```
import numpy as np
import sklearn.utils
from scipy.special import expit
import pickle


class NeuralNetwork:
    def __init__(self, n_in=784, hidden=200, out=10):
        self.n_hidden = hidden
        self.n_in = n_in
        self.n_out = out

    def predict(self, V, W, images):
        images = np.matrix(np.append(images, np.ones((images.shape[0], 1)), axis=1))
        H = np.matrix(np.append(np.tanh(V * images.getT()), np.ones((1, images.shape[0])), 0))
        return expit(W * H)

    # Train Network Algorithm
    def train(self, images, labels, epochs_for_plot, errRates, lossFunc, learning_rate=0.05,
        batch_size=1):
        def mean_squared_update(h, z, b, image, image_label):
            # MSE
            delta_w = learning_rate * (z[:, b] - image_label[b, :].getT())
            delta_w = np.multiply(delta_w, np.multiply(z[:, b], (1 - z[:, b])))
            delta_w = np.multiply(delta_w, np.repeat(h[:, b].getT(), 10, axis=0))

            delta_v = np.multiply(learning_rate * (1 - np.square(h[:-1, b])),
                                  sum([W[j, :-1] * (z[j, b] - image_label[b, j]) * (z[j, b] * (1
                                      - z[j, b])) for j in
                                       range(self.n_out)]).getT())
            delta_v = np.multiply(delta_v, np.repeat(image[b, :], self.n_hidden, axis=0))

            return delta_v, delta_w

        def cross_entropy_update(h, z, b, image, image_label):
            # H, and Z are calculated in the forward pass, here we back-propagate to find update
                rule
            delta_w = learning_rate * (z[:, b] - image_label[b, :].getT())
            delta_w = np.multiply(delta_w, np.matrix(np.repeat(h[:, b].getT(), 10, axis=0)))

            delta_v = np.multiply(learning_rate * (1 - np.square(h[:-1, b])),
                                  sum([W[j, :-1] * (z[j, b] - image_label[b, j]) for j in
                                      range(self.n_out)]).getT())
            delta_v = np.multiply(delta_v, np.repeat(image[b, :], self.n_hidden, axis=0))

            return delta_v, delta_w

        self.n_in = np.size(images, 1)  # number of neurons in the input layer = #features, (we
            don't count the bias)

        # Initialize all weights, V, W at random
        V = np.matrix(np.random.normal(0, 0.01, (self.n_hidden, self.n_in + 1)))
        W = np.matrix(np.random.normal(0, 0.01, (self.n_out, self.n_hidden + 1)))

        epochs, err_rate = 0, 0
        while True:
            images, labels = sklearn.utils.shuffle(images, labels)
            print("epoch# = {0}".format(epochs))
            old_err_rate = err_rate
            ZZZ = self.predict(V, W, images)
            err_rate = (sum(np.argmax(labels, axis=1) == np.argmax(ZZZ.getT(), axis=1))[0, 0]) /
                (labels.shape[0])
            err_diff = np.math.fabs(err_rate - old_err_rate)
```

```
        print("Training Accuracy on epoch {1}: {0}".format(err_rate, epochs))

        if err_diff <= .0000001:
            break

        for i in range((np.size(images, 0) // batch_size) + 1):
            # pick batch data points (x, y) at random from the training set (FOR stochastic
                GD batch_size = 1)
            image = images[i * batch_size:(i + 1) * batch_size, :]
            image_label = labels[i * batch_size:(i + 1) * batch_size, :]

            if image.size == 0:
                continue

            # perform forward pass (computing necessary values for gradient descent update)
            # vectorized for efficiency computation of the outputs on all samples
            image = np.matrix(np.append(image, np.ones((image.shape[0], 1)), axis=1))
            H = (np.append(np.tanh(V * image.getT()), np.ones((1, image.shape[0])), 0))
            Z = expit(W * H)

            for b in range(image.shape[0]):
                del_v, del_w = cross_entropy_update(H, Z, b, image, image_label)
                V -= del_v
                W -= del_w

            if labels.shape[0] == 0 or i % 1000 != 0:
                continue

            useThis = np.matrix(np.append(images, np.ones((images.shape[0], 1)), axis=1))
            H = (np.append(np.tanh(V * useThis.getT()), np.ones((1, useThis.shape[0])), 0))
            Z = expit(W * H)
            ce_loss = .5 * np.sum(np.square(Z[:, :].getT() - labels[:, :]))
            errRates = np.append(errRates,
                                 np.sum(np.argmax(labels, axis=1) == np.argmax(Z.getT(),
                                     axis=1)) / (
                                     labels.shape[0]))
            lossFunc = np.append(lossFunc, ce_loss)
            epochs_for_plot = np.append(epochs_for_plot, i)

        if epochs >= 5:
            with open('CE_Loss_epoch_{0}.pickle'.format(epochs), 'wb') as f:
                pickle.dump({"V": V, "W": W}, f, pickle.HIGHEST_PROTOCOL)

        epochs += 1
        learning_rate *= .1  # anneal the learning rate

    return V, W, epochs_for_plot, errRates, lossFunc
```

ANN_Digit_Classifier.py

```
import numpy as np
import scipy.io
import sklearn.preprocessing
import sklearn.cross_validation
import NeuralNetwork as nn
import pickle
import matplotlib.pyplot as plt


def fix_labs(val):
    label_vec = np.zeros(10)
    label_vec[val] = 1
    return label_vec


def centroid_calc(N, imgs):
    imgs = np.array(imgs.reshape((1, len(imgs))))
    imgs.reshape([N, 28, 28])
    imgtotalsum = np.sum(np.sum(imgs, axis=0), axis=0)

    indices = np.arange(28)
    X, Y = np.meshgrid(indices, indices)
    X = X.reshape((1, 784))
    Y = Y.reshape((1, 784))
    centroidx = np.sum(np.sum(imgs * X, axis=0), axis=0) / imgtotalsum
    centroidy = np.sum(np.sum(imgs * Y, axis=0), axis=0) / imgtotalsum
    return centroidx, centroidy


def custom_feature_extractor(imagetrue):
    featureslist = np.array([item for sublist in imagetrue for item in sublist])
    ctr_x, ctr_y = centroid_calc(1, featureslist)
    h = np.size(imagetrue, 0)
    w = np.size(imagetrue, 1)
    featureslist = np.append(featureslist,
                             [np.average(imagetrue[h / 2:, w / 2:]), np.average(imagetrue[h /
                                 2:, :w / 2]),
                              np.average(imagetrue[:h / 2, w / 2:]), np.average(imagetrue[:h /
                                  2, :w / 2])])

    topclosed = False
    prevrow = (False, False, [], [], [])
    numholes = 0
    edgecount = 0
    for row in imagetrue:
        for i in range(np.size(row) - 1):
            if row[i] != 0 and row[i + 1] == 0:
                edgecount += 1
            elif row[i] == 0 and row[i + 1] != 0:
                edgecount += 1

        nonzero_elems = np.nonzero(row)[0]
        if len(nonzero_elems) > 0:
            separation = np.ediff1d(nonzero_elems) - 1
            nonzerodiffs = np.nonzero(separation)[0]
            startnonzero = nonzero_elems[0]
            endnonzero = nonzero_elems[-1]
            if len(nonzerodiffs) > 0:  # has gaps
                gaps_in_row = []
                for block in nonzerodiffs:
                    gap_start_index = nonzero_elems[block]
                    gap_end_index = nonzero_elems[block] + separation[block] + 1
                    gaps_in_row.append((gap_start_index, gap_end_index))

                # do stuff here, that has gaps of 0pixels, figure out how to detect holes
                if prevrow[0]:
```

```
                    if prevrow[1]:  # prevrow has Gaps, this row has gaps
                        for prevGap in prevrow[2]:
                            leakfromtop = shortcircuit = True
                            for ii in range(len(gaps_in_row)):
                                for p in range(gaps_in_row[ii][0], gaps_in_row[ii][1] + 1):
                                    shortcircuit = shortcircuit and prevrow[4][p] != 0
                                    if not shortcircuit:
                                        break
                                if shortcircuit:
                                    topclosed = True

                                if prevGap[0] >= startnonzero and prevGap[1] <= endnonzero:
                                    leakfromtop = False
                                    break
                            if leakfromtop:
                                topclosed = False
                                leakfromtop = False
                    else:  # prevrow no gaps, this row has gaps
                        for gap in gaps_in_row:
                            if not prevrow[3][0] <= gap[0] + 1 or not prevrow[3][1] >= gap[1] -
                                1:
                                topclosed = False

                else:  # zero pixels filled row
                    topclosed = False
                prevrow = (True, True, gaps_in_row, (startnonzero, endnonzero), row)
            else:  # only stuff here has no zero gaps in it
                if not prevrow[0]:  # prev row was all zeros and this row is all solid
                    topclosed = True
                else:  # prev row not all zeros
                    if prevrow[1]:  # prevrow has gaps, this row solid
                        for gap in prevrow[2]:
                            if startnonzero - 1 <= gap[0] and gap[1] <= endnonzero + 1:
                                if topclosed:
                                    numholes += 1
                    else:  # prevrow solid, this row solid
                        topclosed = True
                prevrow = (True, False, [], (startnonzero, endnonzero), row)
        else:  # row is all zeros
            prevrow = (False, False, [], (), row)

    binary = [0, 0, 0]
    if numholes > 2:
        numholes = 2
    binary[numholes] = 1

    return np.append(featureslist, binary + [edgecount, ctr_x, ctr_y])

#
# mat = scipy.io.loadmat("dataset/train.mat")
# train_images = np.array(mat['train_images'])
# # testing = np.array(mat['test_images'])
# train_labels = np.array(mat["train_labels"])
# numTrainImages = np.size(train_images, 2)
# # numTestImages = np.size(testing, 2)
#
# numSamples = numTrainImages
# print("total training samples = " + str(numTrainImages))
# print("number samples using = " + str(numSamples))
#
# images = np.matrix(np.empty((numSamples, len(custom_feature_extractor(train_images[:, :,
    0])))))
# # test_images = np.matrix(np.empty((numSamples, len(custom_feature_extractor(testing[:, :,
    0])))))
# for i in range(numSamples):
#     images[i] = np.matrix(custom_feature_extractor(train_images[:, :, i]))
# labels_matrix = np.matrix([fix_labs(item) for item in train_labels])
#
```

```
# persistentStore = {"images": images, "labels_matrix": labels_matrix}  #
    "test_images":test_images}
# with open('store training structures.pickle', 'wb') as f:
#     # Pickle the 'data' dictionary using the highest protocol available.
#     pickle.dump(persistentStore, f, pickle.HIGHEST_PROTOCOL)
#
# hidden = 100
# batch_size = 10
# learning_rate = .1
# loss_func = "cross_entropy"
#
# with open('store training structures.pickle', 'rb') as f:
#     # The protocol version used is detected automatically, so we do not
#     # have to specify it.
#     data = pickle.load(f)
#     labels_matrix, images = data["labels_matrix"], data["images"]
#
# neural_net = nn.NeuralNetwork(hidden=hidden)
#
# images = np.matrix(sklearn.preprocessing.normalize(images, norm="l2", axis=1))
#
# X_train, X_test, y_train, y_test = sklearn.cross_validation.train_test_split(images,
    labels_matrix, test_size=0)
# print("training = " + str(np.shape(X_train)))
# print("labels = " + str(np.shape(y_train)))
#
# plot_x = np.empty((0, 0))
# plot_y = np.empty((0, 0))
# plot_y2 = np.empty((0, 0))
#
# V, W, plot_x, plot_y, plot_y2 = neural_net.train(X_train, y_train, plot_x, plot_y, plot_y2,
    learning_rate=learning_rate,
#                                                  batch_size=batch_size)
#
# mat = scipy.io.loadmat("dataset/test.mat")
# test_images = np.array(mat['test_images'])
# images = np.matrix(np.empty((test_images.shape[0], len(custom_feature_extractor(test_images[0,
    :, :])))))
#
# for i in range(test_images.shape[0]):
#     images[i] = np.matrix(custom_feature_extractor(test_images[i, :, :]))
#
# images = np.matrix(sklearn.preprocessing.normalize(images, norm="l2", axis=1))
#
# persistentStore = {"V": V, "W": W, "batch_size": batch_size, "learning_rate": learning_rate,
#                    "hidden_layers": hidden, "lossfunction": loss_func, "plot_x": plot_x,
    "plot_y": plot_y,
#                    "plot_y2": plot_y2, "images": images}
#
# with open('FinishedNN.pickle', 'wb') as f:
#     # Pickle the 'data' dictionary using the highest protocol available.
#     pickle.dump(persistentStore, f, pickle.HIGHEST_PROTOCOL)

with open('FinishedNN.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
    V, W, hidden, plot_x, plot_y, plot_y2, images = data["V"], data["W"], data["hidden_layers"],
        data["plot_x"], data[
        "plot_y"], data["plot_y2"], data["images"]

neural_net = nn.NeuralNetwork(hidden=hidden)
predicted = np.argmax(neural_net.predict(V, W, images).getT(), axis=1)

f = open("output.csv", 'w')
f.write("Id,Category\n")
for i in range(predicted.shape[0]):
    f.write(str(i + 1) + "," + str(predicted[i][0, 0]) + "\n")
```

```
print("DONE!")

plt.plot(plot_x, plot_y, "r-")
#plt.plot(plot_x, plot_y2)
plt.ylabel('Accuracy Rate')
plt.xlabel('Iterations Number')
plt.show()

# Z_train = neural_net.predict(V, W, X_train)
# Z_test = neural_net.predict(V, W, X_test)
#
# train_err = (sum(np.argmax(y_train, axis=1) == np.argmax(Z_train.getT(), axis=1))[0, 0]) /
    (y_train.shape[0])
# print("Training Accuracy is: {0}".format(train_err))
#
# test_err = (sum(np.argmax(y_test, axis=1) == np.argmax(Z_test.getT(), axis=1))[0, 0]) /
    (y_test.shape[0])
# print("Testing Accuracy is: {0}".format(test_err))
```