CS189/CS289A – Spring 2017 — Homework 5

Yaoyang Zhang, SID 3032114788

Problem 1: Derivation

Consider stochastic gradient descent with one input sample at a time, let input be a column vector x, the hidden layer be a column vector h, and the out put layer be a column vector z. The weights passed from the input layer to the hidden layer is a matrix V and the weights passed from the hidden layer to the output layer is a matrix W.

$$h = tanh(Vx)$$
$$z = s(Vh)$$

where $tanh()$ is the hybolic tangent function and $s()$ is the sigmoid function.
The loss function is the cross entropy:

$$L(z, y) = -\sum_{i=1}^{C} y_i \ln z_i + (1 - y_i) \ln(1 - z_i)$$

where C is the size of the output layer. In this problem $C = 26$.
The derivative w.r.t. z_i is

$$\frac{\partial L}{\partial z_i} = \frac{z_i - y_i}{(1 - z_i)z_i}$$

The derivative w.r.t. W_i, the ith row of W, is

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial W_i} = \frac{z_i - y_i}{(1 - z_i)z_i}(1 - z_i)z_i h = (z_i - y_i)h$$

The gradient w.r.t. W is

$$\nabla_W L = (z - y)h^T$$

The gradient w.r.t. h is

$$\nabla_h L = \sum_{i=1}^{C} \frac{\partial L}{\partial z_i} z_i(1 - z_i)W_i = W^T(z - y)$$

The derivative w.r.t V_i is

$$\frac{\partial L}{\partial V_i} = \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial V_i} = (W^T(z - y))_i(1 - h_i^2)x$$

The gradient w.r.t. V is

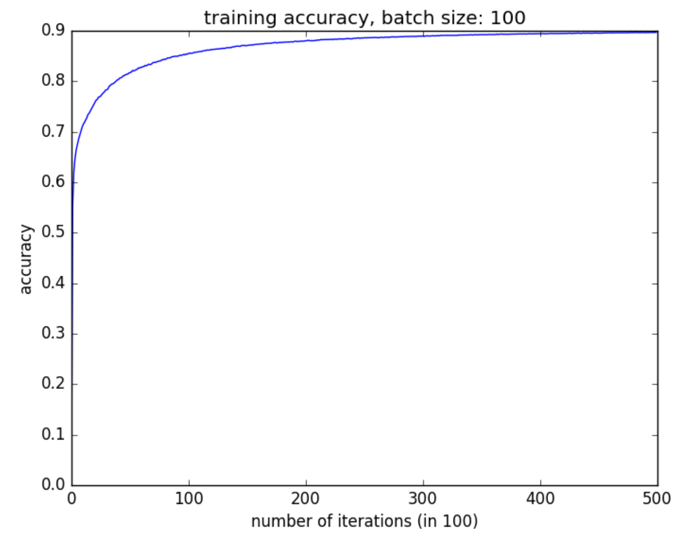
$$\nabla_V L = (W^T(z - y) \circ (1 - h \circ h))x^T$$

where $\circ$ means element-wise multiplication.
The SGD update would be

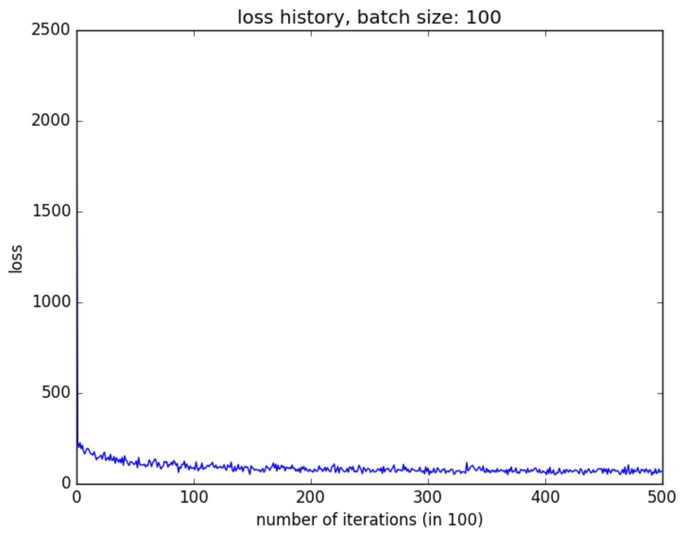
$$W \leftarrow W - \epsilon \nabla_W L$$
$$V \leftarrow V - \epsilon \nabla_V L$$

Problem 2: Implementation

The neural network I implemented has 300 neurons in the hidden layer. It uses tanh as activation function for the hidden layer and sigmoid function as activation function for the output layer. I used mini-batch gradient descent to train the neural network. The batch size is 100, the learning rate is 0.03, the regularization parameter is 0.001.
The training accuracy v.s. number of iterations is shown in the following plot



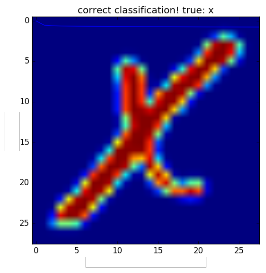
The validation accuracy in the end of training is 0.88.
The loss v.s. number of iterations is shown in the following plot

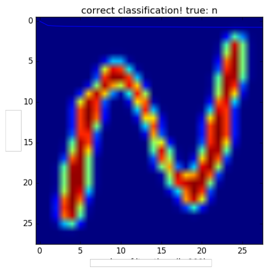


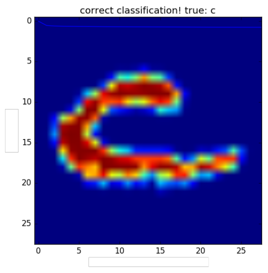
My Kaggle score is 0.87894 and my display name is YaoyangZhang

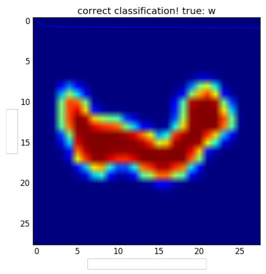
Problem 3: Visualization

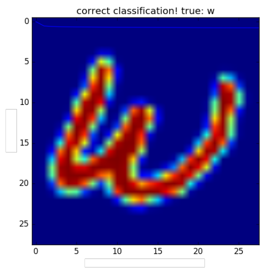
Here are 5 digits that my NN correctly classified



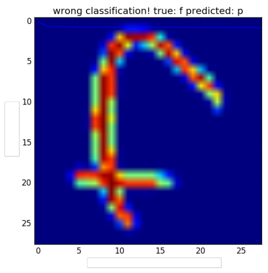


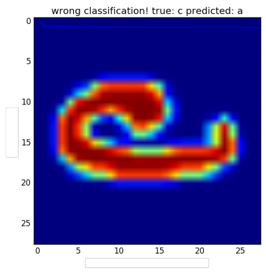


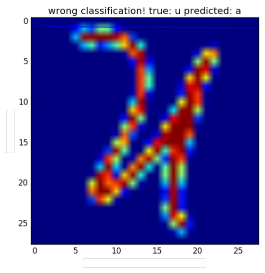


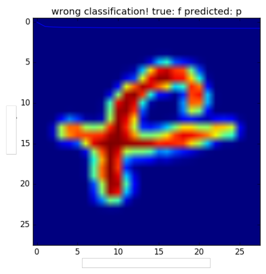


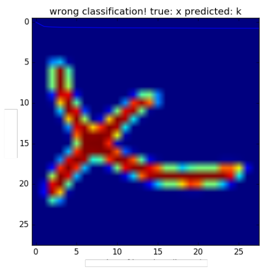
Here are 5 digits that my NN fail to classify











Problem 4: Bells and Whistles

Here are the things I implemented in addition to the basic neural network:
1. I used a decaying learning rate. The learning rate is multiplied by 0.9 after every 2 epochs.
2. I used 300 units in the hidden layer.
3. I used L2 regularization.
4. I implemented the mini-batch gradient descent.

Appendix

```
import numpy as np


class NeuralNet:
    def __init__(self, in_size, hidden_size, out_size):
        self.in_size = in_size
        self.out_size = out_size
        self.hidden_size = hidden_size
        self.V = np.random.normal(0, 0.05, (self.hidden_size, self.in_size))
        self.W = np.random.normal(0, 0.2, (self.out_size, self.hidden_size))
        self.b1 = 0.01 * np.random.uniform(size=(self.hidden_size,1))
        self.b2 = 0.01 * np.random.uniform(size=(self.out_size,1))

    def train(self, X, y, eps=0.001, max_iter=10000,
                            batch_size = 100, reg = 0.001):
        N, _ = X.shape
        y_one_hot = np.zeros((N, self.out_size))

        for i in range(N):
            y_one_hot[i, int(y[i])] = 1

        loss = []
        train_acc = []
        count = 0
        n_batch = int(N / batch_size)

        while count < max_iter:
            count += 1
            rand_ind = np.random.choice(np.arange(N), batch_size)
            cur_X = X[rand_ind, :].T
            cur_y = y_one_hot[rand_ind, :].T

            # forward prop

            h = np.tanh(np.dot(self.V, cur_X) + self.b1)
            z =self.sigmoid(np.dot(self.W, h) + self.b2)

            # decay learning rate every 2 epoch2
            if count % batch_size == 1:
                predicted_Y = self.predict(X)
                acc = self.accuracy(predicted_Y, y)
                print(acc)
                train_acc.append(acc)
                l = np.sum(-cur_y * np.log(z) - (1 - cur_y) * np.log(1 - z))
                loss.append(l)
            if count % (2 * n_batch) == 0:
                print('another two epoch')
                eps *= 0.9




            dv = ((self.W.T.dot(z - cur_y)) * (1 - h ** 2)).dot(cur_X.T)
                                                + reg * self.V
            dw = np.dot(z - cur_y, h.T) + reg * self.W
            db1 = np.sum(((self.W.T.dot(z - cur_y)) * (1 - h ** 2)),
                                                axis=1, keepdims=True)
            db2 = np.sum(z - cur_y,axis=1, keepdims=True)


            self.W -= eps * dw / batch_size
            self.V -= eps * dv / batch_size
            self.b1 -= eps * db1 / batch_size
            self.b2 -= eps * db2 / batch_size
        print(loss)
        return loss, train_acc


    def predict(self, X):

        x = X.T
        h = np.tanh(np.dot(self.V, x) + self.b1)
        z = self.sigmoid(np.dot(self.W, h) + self.b2)
        label = np.argmax(z, axis=0)
        return label


    def accuracy(self, y_predicted, y_true):
        score = 0
        for i in range(y_true.shape[0]):
            if int(y_predicted[i]) == int(y_true[i]):
                score += 1
        return score / y_true.shape[0]

    def sigmoid(self, z):
        return 1/(1+np.exp(-z))
```