CS189/289A – Spring 2017 — Homework 6

Yicheng Chen, SID 26943685

1. Derivations

$$\nabla_{W_i} L = \frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial W_i}$$
$$= \frac{\partial L}{\partial z_i} s'(W_i \cdot h)h$$

Where $s'(W_i \cdot h) = \nabla_{W_i}\sigma(W_i \cdot h)$ and h is the hidden unit output.

$$\nabla_{V_i} L = \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial V_i}$$
$$= \sum_{j=1}^{k} \frac{\partial L}{\partial z_j}\frac{\partial z_j}{\partial h_i} t'(V_i \cdot x)x$$

Where $t'(V_i \cdot x) = \nabla_{V_i} \tanh(V_i \cdot x)$ and x is a random training sample.

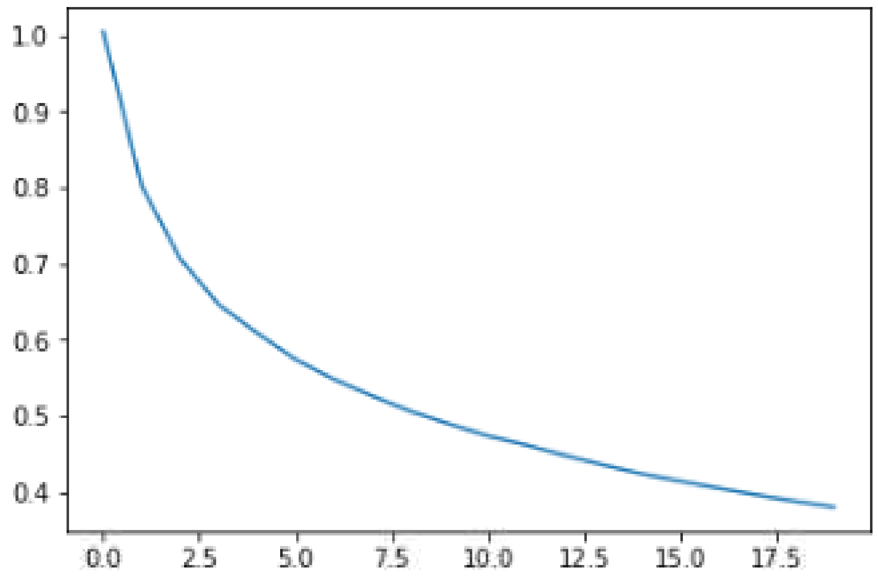
2. Implementation

(a) Hyperparameters tuned: the step size and the number of hidden layer neurons.

(b) Training Accuracy: 0.9508

(c) Validation Accuracy: 0.8578

(d) Plot of the loss value versus number of epochs:

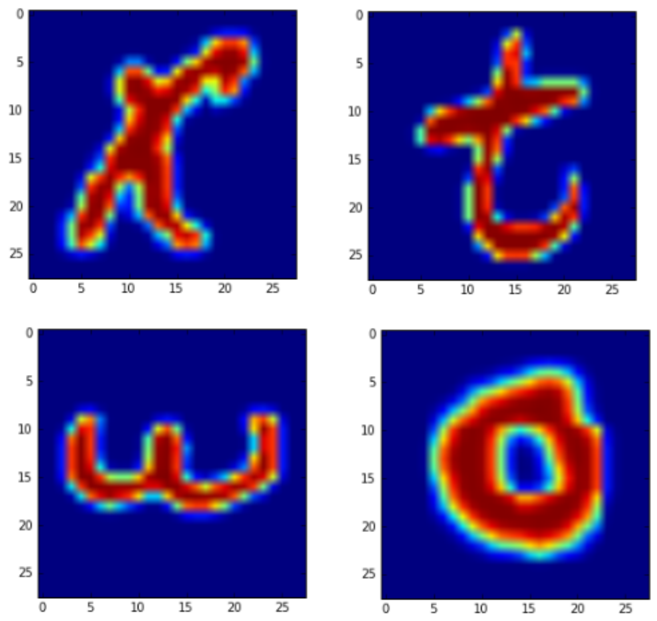


(e) For Kaggle submission, HOG feature is used for model training and predicting. Kaggle Score: 0.91163
Display Name: Yicheng Chen

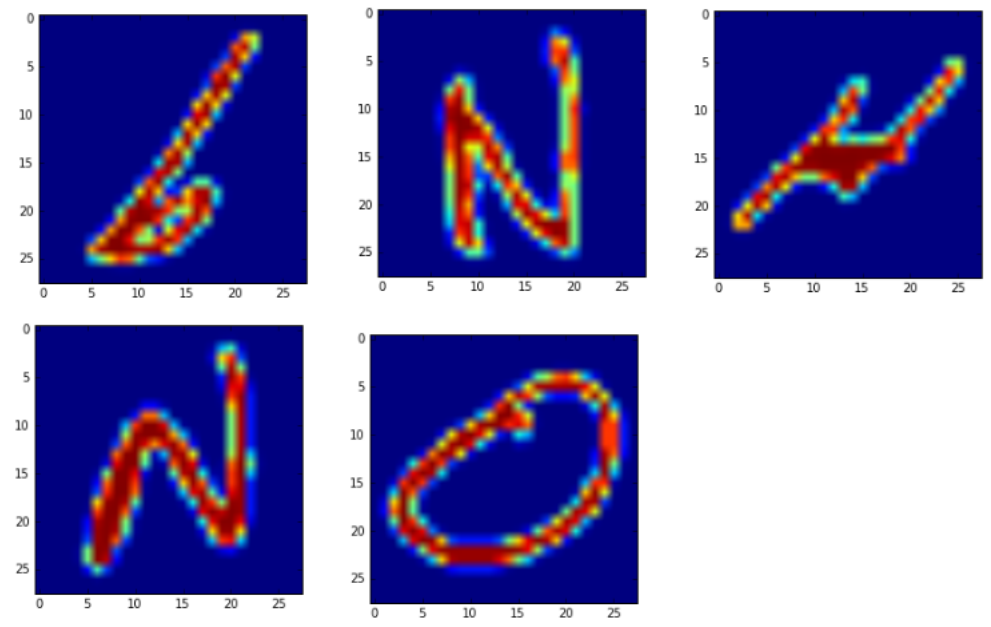
(f) Code: See appendix

3. Visualization

(a) 5 digits in the validation set that are correctly classified:



(b) 5 digits in the validation set that are not correctly classified:



4. Bells and Whistles

Due to the time constraint, no bells and whistles were implemented.

Appendix: Code

```
class neural_network():

    def __init__(self, layer_sizes=[784, 200, 26], step_size=0.1):
        self.layer_sizes = layer_sizes
        self.step_size=step_size #??
        # support classical 3-layer network only
        self.V = np.random.normal(loc=0, scale=1/np.sqrt(layer_sizes[0]+1), size=(layer_siz
        self.W = np.random.normal(loc=0, scale=1/np.sqrt(layer_sizes[1]+1), size=(layer_siz

    # activation function: tanh
    def tanh(self, x):
        x = np.minimum(500, x)
        x = np.maximum(-500, x)
        a = np.exp(x)
        b = np.exp(-x)
        return (a-b)/(a+b)

    # derivative of tanh function
    def tanh_d(self, x):
        t = self.tanh(x)
        return 1 - np.multiply(t, t)

    # activation function: sigmoid
    def sigmoid(self, x):
        return 1/(1+np.exp(-x))

    # derivative of sigmoid function
    def sigmoid_d(self, x):
        s = self.sigmoid(x)
        return np.multiply(s, 1-s)

    # activation function: ReLU
    def relu(self, x):
        return np.maximum(0, x)

    # derivative of ReLU function
    def relu_d(self, x):
        return (x>0).astype(float)

    # activation function: softmax
    def softmax(self, x):
        e = np.exp(x)
        return e / np.sum(e)

    # loss function: cross-entropy
    def cross_entropy(self, z, y):
        return -np.sum(np.multiply(y, np.log(z+1e-20)) + np.multiply((1-y), np.log(1-z+1e-2

    # derivative of cross-entropy loss function
    def cross_entropy_d(self, z, y):
        return -np.divide(y, z+1e-20) + np.divide((1-y), (1-z+1e-20))

    # calcluate h = s(Vx) and z = s(Wh), aka forward propagation
    def forward_prop(self, x):
        x1 = np.append(x, 1) # 1 at the end
        h = self.sigmoid(self.V.dot(x1)) # tanh for hidden layer
        h1 = np.append(h, 1) # 1 at the end
        z = self.sigmoid(self.W.dot(h1)) # sigmoid for output layer
        return x1, h1, z

    # calculate the gradient of weight matrices, aka backward propagation
    def backward_prop(self, x1, h1, z, y):
        dLdz = self.cross_entropy_d(z, y)
        s2d = self.sigmoid_d(self.W.dot(h1))
        # the gradient of W
        dW = np.outer(np.multiply(dLdz, s2d), h1)

        s1d = self.sigmoid_d(self.V.dot(x1))
        dLdh1 = np.sum(np.dot(np.diag(np.multiply(dLdz, s2d)), self.W), axis=0)
        dLdh = dLdh1[:-1] # don't include the 1
        # the gradient of V
        dV = np.outer(np.multiply(dLdh, s1d), x1)
        return dV, dW

    # vectorize data label
    def vectorize_y(self, y):
        M = np.max(y)
        N = len(y)
        yv = np.zeros([N, M])
        for i in range(N):
            yv[i, y[i]-1] = 1
        return yv

    # transform vectorized y back to labels
    def labelize_y(self, y):
        return np.argmax(y, axis=1) + 1

    # train NN using stochastic gradient descent
    # X: Nxd data matrix
    # y: Nx1 data label, range 1~26
    def train_sgd(self, X, y, N_epoch=10):
        all_loss = []
        yv = self.vectorize_y(y)
        for epoch in range(0, N_epoch):
            print('epoch #%d' % epoch)
            for i in range(X.shape[0]):
                xr = X[i, :]
                yr = yv[i, :]
                # FP
                x1, h1, z = self.forward_prop(xr)
                # BP
                dV, dW = self.backward_prop(x1, h1, z, yr)
                # GD
                self.V -= self.step_size * dV
                self.W -= self.step_size * dW
            print('  Training Accuracy: %f' % self.accuracy(X, y))
            all_loss.append(self.loss(X, yv))
        return all_loss

    def predict(self, X):
        z = np.zeros([len(X), len(self.W)])
        for i, x in enumerate(X):
            _, _, z[i, :] = self.forward_prop(x)
        z = np.argmax(z, axis=1) + 1
        return z

    def accuracy(self, X, y):
        z = self.predict(X)
        N = len(y)
        return np.sum(z == y) / float(N)

    def loss(self, X, y):
        loss = 0
        z = np.zeros([len(X), len(self.W)])
        for i, x in enumerate(X):
            _, _, z[i, :] = self.forward_prop(x)
        for i in range(len(z)):
            yi = y[i, :]
            zi = z[i, :]
            loss += self.cross_entropy(zi, yi)
        return loss / len(z)
```