A11_submitted_version

February 12, 2019

1 Problem 1

Team member (alphabetical order):
Jin Dong, Liheng Ma, Maximilien Le Clei

```
In [0]: import numpy as np
        import matplotlib.pyplot as plt
        import random
        np.random.seed(1)
```

```
In [2]: from google.colab import drive
        drive.mount('/content/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/gdrive
```

1.1 Building the Model

1. We use two 666- dimensional hidden layers, the number of parameters are $785 * 666 + 667 * 666 + 667 * 10 = 973,702$.
2. We implement a vectorized forward and backward propagation with mini-batch.

```
In [0]: def one_hot(labels, n):
            """labels: m*1 vector
               n: expected classes
               outout: m*n matrix"""
            m = len(labels)
            onehot = np.zeros((m, n))
            onehot[np.arange(m), labels] = 1
            return onehot

        # We re-store the dataset into different npy files
        # label is one-hot encoded by one_hot
```

```
DATA_PATH = r'/content/gdrive/My Drive/Datasets/MNIST'
X_train = np.load(DATA_PATH + '/x_train.npy')
y_train = one_hot(np.load(DATA_PATH + '/y_train.npy'),10)
X_val   = np.load(DATA_PATH + '/x_val.npy')
y_val   = one_hot(np.load(DATA_PATH + '/y_val.npy'),10)
X_test  = np.load(DATA_PATH + '/x_test.npy')
y_test  = one_hot(np.load(DATA_PATH + '/y_test.npy'),10)
```

In [0]:
```
def accuracy(y_pred, y):
    return np.sum(1 * np.argmax(y_pred, axis=1) \
                  == np.argmax(y, axis=1)) * 100.0 / y.shape[0]


def data_iter(data, batch_size):
    X, y = data
    batches = [(X[i:i+batch_size],y[i:i+batch_size]) \
               for i in range(0,X.shape[0],batch_size)]
    random.shuffle(batches)
    for batch in batches:
        yield batch


def glorot(in_dim, out_dim):
    d = np.sqrt(6/(in_dim+out_dim))
    return np.random.uniform(-d,d,(in_dim,out_dim))
```

In [0]:
```
INPUT_DIM = 784
OUTPUT_DIM = 10

class NN(object):


    def __init__(self,hidden_dims=[666,666], n_hidden=2,
                 init='Normal',activate='relu'):
        self.dims = [INPUT_DIM,] + hidden_dims + [OUTPUT_DIM,]
        self.weights = []
        self.biases  = []
        self.init = init
        self.activate = activate

        self.initialize_weights()


    def initialize_weights(self):
        '''
        init weights of all layers according to self.init
        '''
```

```python
        init_method = None
        if self.init == 'Zero':
            init_method = lambda x, y: np.zeros((x,y))
        elif self.init == 'Normal':
            init_method = lambda x, y: np.random.randn(x,y)
        elif self.init == 'Glorot':
            init_method = glorot
        else:
            raise Exception('Choose right initialization method.')

        for (inputs, outputs) in zip(self.dims[:-1], self.dims[1:]):
            self.weights.append(init_method(inputs, outputs))
            self.biases.append(np.zeros(outputs))


    def activation(self, inputs):
        if self.activate == 'relu':
            inputs[inputs < 0] = 0
            return inputs
        if self.activate == 'sigmoid':
            return 1.0/(1.0+np.exp(-inputs))


    def loss(self, pred, labels):
        '''
        cross entropy loss
        '''
        ls = np.nan_to_num(np.log(pred+1e-8))
        ls = - np.sum(labels * ls)
        return ls / pred.shape[0]


    def forward(self, inputs, labels):
        a_k = None
        h_k = inputs
        a   = []
        h   = [h_k]
        for (W, b) in zip(self.weights[:-1], self.biases[:-1]):
            a_k = np.dot(h_k, W) + b
            h_k = self.activation(a_k)
            a.append(a_k)
            h.append(h_k)

        a_k = np.dot(h_k, self.weights[-1]) + self.biases[-1]
        h_k = self.softmax(a_k)
        a.append(a_k)
        h.append(h_k)
```

```python
        ls = self.loss(h_k, labels)
        cache = (a, h)

        return h_k, ls, cache


    def backward(self,cache,labels,lss):
        """
        Input: cache: (as, hs)
                    as: preactivate values
                    hs: activated values
                    lss: loss
        output: grads: (grads_w, grads_b)
        """
        as_ = cache[0]
        hs_ = cache[1]

        nabla_w = [np.zeros_like(w) for w in self.weights]
        nabla_b = [np.zeros_like(b) for b in self.biases]

        # nabla l -> softmax -> pre-softmax
        nabla_a = -(labels - hs_[-1])
        nabla_b[-1] = np.sum(nabla_a, axis=0)
        nabla_w[-1] = np.dot(hs_[-2].T, nabla_a)
        # for each preactivate -> activation layer
        for layer in range(2, len(self.dims)):
            nabla_h = np.dot(nabla_a, self.weights[-layer+1].T)
            nabla_a = nabla_h * self.activate_grad(as_[-layer])

            nabla_b[-layer] = np.sum(nabla_a, axis=0)
            nabla_w[-layer] = np.dot(hs_[-layer-1].T, nabla_a)

        nabla_w = [x / labels.shape[0] for x in nabla_w]
        nabla_b = [x / labels.shape[0] for x in nabla_b]
        return (nabla_w,nabla_b)


    def update(self,grads,lr):
        grads_w, grads_b = grads
        for i in range(len(self.weights)):
            self.weights[i] -= lr * grads_w[i]
            self.biases[i] -= lr * grads_b[i]


    def train(self, data, epochs, batch_size, lr, lambd=0.0, test_data=None):
        l_acc = []
        l_ls  = []
        for ep in range(1, epochs+1):
```

```
            for (batch_x, batch_y) in data_iter(data, batch_size):
                y_pred, ls, cache = self.forward(batch_x, batch_y)
                grads = self.backward(cache, batch_y, ls)
                self.update(grads, lr)
            if test_data:
                acc, ls = self.test(test_data)
                l_acc.append(acc)
                l_ls.append(ls)
                print('Epoch %i (acc, loss):(%.4f,%.4f)' % (ep, acc, ls))
        return l_acc, l_ls


    def test(self, data):
        x, y = data
        outputs, ls, _ = self.forward(x, y)
        return accuracy(outputs, y), ls


    def activate_grad(self,inputs):
        '''
        derivatives of activation function
        '''
        if self.activate == 'relu':
            inputs[inputs > 0] = 1
            inputs[inputs < 0] = 0
            return inputs
        elif self.activate == 'sigmiod':
            return self.activation(inputs) * (1 - self.activation(inputs))


    def softmax(self,inputs):
        inputs = inputs - np.max(inputs, axis=1).reshape(inputs.shape[0],1)
        outputs = np.exp(inputs)
        return outputs / (np.sum(outputs, axis=1).reshape(inputs.shape[0],1))
```

1.2 Initialization

The results show that:

1. Networks with zero initialization learn nothing. Becasuse corresponding weights, intermediate variables are also zeros, and will be used to compute gradients, most gradients will be also zero which causes the network learns nothing.
2. Networks with normal initialization need smaller learning rate. If we use a large learning rate and the network was initialized to a bad point. Some of the predicted probabilites of the right classes will be close to 0, causing the cross entropy loss move to infinity.

3. Networks with Glorot initialization are more stable when we use relatively large learning rate., and it has the lowest loss.

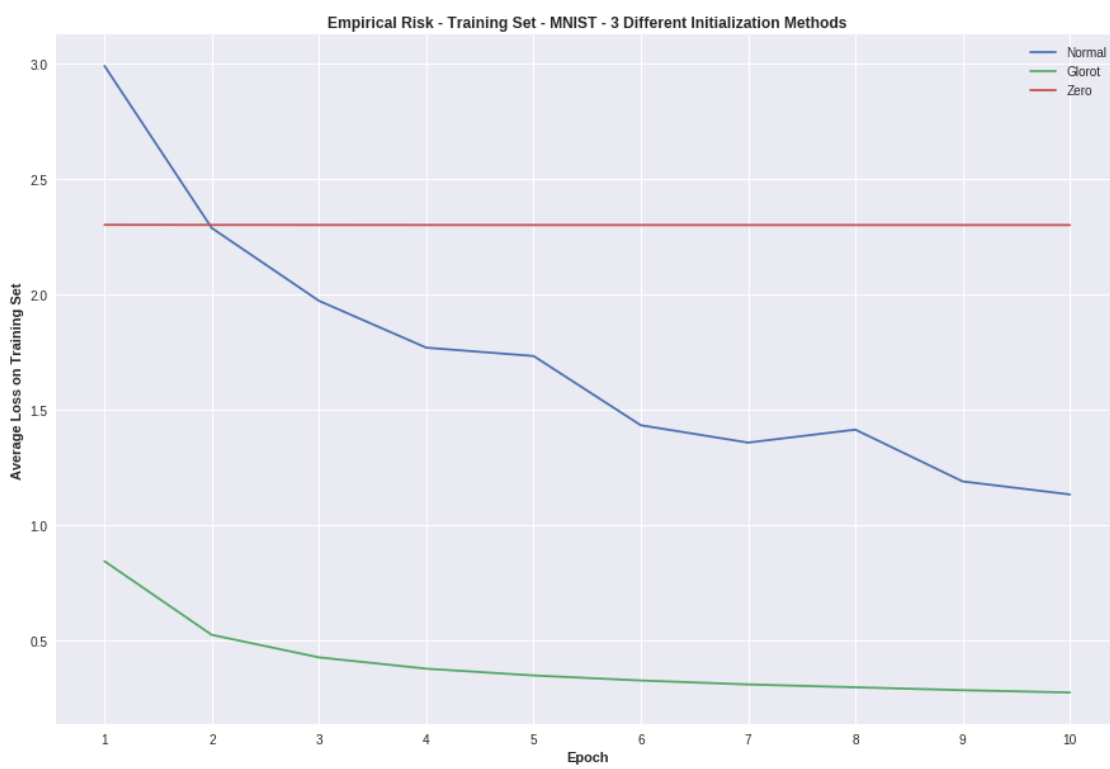
We use the architecture as below in this section:

1. Architecture (dimensions) : 784 -> 666 -> 666 -> 10.
2. parameters: 785*666+667*666+667*10 = 973,702 parameters.
3. Nonlinearity : ReLU
4. Learning rate : 0.01(zero init), 0.001(normal init), 0.01(glorot init)
5. Batch size : 200
6. Numpy random seed : 0
7. Epochs: 10

In [6]:
```
nn1 = NN(hidden_dims=[666,666],n_hidden=2,init='Zero')
zero_acc, zero_ls = nn1.train((X_train,y_train), epochs=10, batch_size=200,
                              lr=0.01, test_data=(X_train,y_train))

nn2 = NN(hidden_dims=[666,666],n_hidden=2,init='Normal')
normal_acc, normal_ls = nn2.train((X_train,y_train), epochs=10, batch_size=200,
                                  lr=0.001, test_data=(X_train,y_train))

nn3 = NN(hidden_dims=[666,666],n_hidden=2,init='Glorot')
glorot_acc, glorot_ls = nn3.train((X_train,y_train), epochs=10, batch_size=200,
                                  lr=0.01, test_data=(X_train,y_train))

from pylab import rcParams
rcParams['figure.figsize'] = 15, 10

plt.xticks(np.arange(0, 11, step=1))
plt.xlabel('Epoch', weight='bold')
plt.ylabel('Average Loss on Training Set', weight='bold')
tlt = 'Empirical Risk - Training Set - MNIST - 3 Different Initialization Methods'
plt.title(tlt, weight='bold')

plt.plot(np.arange(1, 11, step=1), normal_ls, label='Normal')
plt.plot(np.arange(1, 11, step=1), glorot_ls, label='Glorot')
plt.plot(np.arange(1, 11, step=1), zero_ls, label='Zero')
plt.legend()
plt.show()
```

```
Epoch 1 (acc, loss):(11.3560,2.3020)
Epoch 2 (acc, loss):(11.3560,2.3016)
Epoch 3 (acc, loss):(11.3560,2.3013)
Epoch 4 (acc, loss):(11.3560,2.3012)
Epoch 5 (acc, loss):(11.3560,2.3011)
Epoch 6 (acc, loss):(11.3560,2.3011)
Epoch 7 (acc, loss):(11.3560,2.3011)
Epoch 8 (acc, loss):(11.3560,2.3010)
Epoch 9 (acc, loss):(11.3560,2.3010)
```

```
Epoch 10 (acc, loss):(11.3560,2.3010)
Epoch 1 (acc, loss):(83.6440,2.9903)
Epoch 2 (acc, loss):(87.4520,2.2880)
Epoch 3 (acc, loss):(89.1960,1.9726)
Epoch 4 (acc, loss):(90.2840,1.7698)
Epoch 5 (acc, loss):(90.4860,1.7337)
Epoch 6 (acc, loss):(92.1260,1.4338)
Epoch 7 (acc, loss):(92.5160,1.3586)
Epoch 8 (acc, loss):(92.1840,1.4147)
Epoch 9 (acc, loss):(93.4440,1.1905)
Epoch 10 (acc, loss):(93.7220,1.1340)
Epoch 1 (acc, loss):(82.9940,0.8446)
Epoch 2 (acc, loss):(87.0940,0.5257)
Epoch 3 (acc, loss):(88.7600,0.4281)
Epoch 4 (acc, loss):(89.6600,0.3792)
Epoch 5 (acc, loss):(90.2960,0.3498)
Epoch 6 (acc, loss):(90.7920,0.3282)
Epoch 7 (acc, loss):(91.2300,0.3109)
Epoch 8 (acc, loss):(91.5620,0.2986)
Epoch 9 (acc, loss):(91.8940,0.2860)
Epoch 10 (acc, loss):(92.1920,0.2761)
```



1.3 Parameter Search

We achieved more than 97% accuracy on validation dataset by using below hyperparameters

1. Architecture (dimensions) : 784 -> 666 -> 666 -> 10.
2. parameters: 785*666+667*666+667*10 = 973,702 parameters.
3. Nonlinearity : ReLU
4. Learning rate : 0.1
5. Batch size : 100
6. Numpy random seed : 0
7. Initialization : Glorot
8. Epochs: 10

```
In [7]: nn4 = NN(hidden_dims=[666,666],n_hidden=2,init='Glorot')
        best_acc, best_ls = nn4.train((X_train,y_train), epochs=10, batch_size=100,
                                      lr=0.1, test_data=(X_val,y_val))
```

```
Epoch 1 (acc, loss):(93.5900,0.2212)
Epoch 2 (acc, loss):(95.7800,0.1518)
Epoch 3 (acc, loss):(96.7000,0.1251)
Epoch 4 (acc, loss):(96.6000,0.1179)
Epoch 5 (acc, loss):(97.1600,0.0952)
Epoch 6 (acc, loss):(97.6300,0.0848)
Epoch 7 (acc, loss):(97.6000,0.0840)
Epoch 8 (acc, loss):(97.5800,0.0820)
Epoch 9 (acc, loss):(97.8000,0.0770)
Epoch 10 (acc, loss):(97.8600,0.0730)
```

1.4 Validate Gradients using Finite Dierence

We tried 10 N values (1, 5, 10, 50, 1000, 5000, 100000, 500000). The diagram show that, with N increasing (ϵ decreasing), the difference between the finite difference gradient approximation and gradient obtained from back propagation is becoming smaller till 0, which means the finite difference gradient approximation is becoming more accurate, till close to the real gradient.

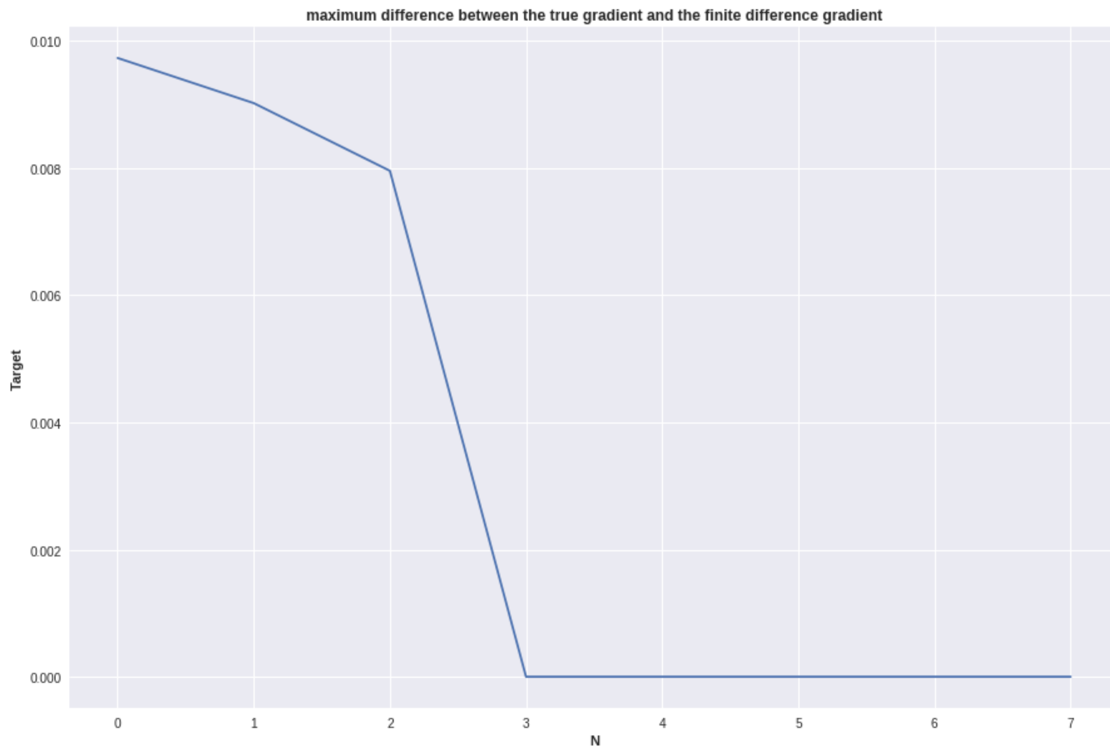
We also fount sometimes the result was a horizon line, which means no differences at all. By checking the input data, we found the reason is that the corresponding pixel value of that weight is zero, which means there is no gradients at these weights if we input that image.

```
In [21]: nn5 = NN(hidden_dims=[666,666],n_hidden=2,init='Glorot')
         i_value = [0,1,3,5]
         k_value = [1,5]
         N_value = [k*10**i for i in i_value for k in k_value]
         print('N values used:', N_value)
         p = 10

         # obtain one data point, fix random.seed to reproduce the result.
         np.random.seed(100)
         data_number = np.random.randint(0,X_train.shape[0])
```

```python
    X, y = X_train[data_number,:].reshape(1,-1), y_train[data_number,:].reshape(1,-1)

    # compute forward and backward prop and the gradient
    y_hat, ls, cache = nn5.forward(X, y)
    grad_W, grad_b = nn5.backward(cache, y, ls)
    # Use the second layer to validate gradient.
    grad_theta = grad_W[1][:p,0]

    res = []
    for N in N_value:
        epsilon = 1 / N
        grad_diff = np.zeros(p)
        for i in range(p):
            # compute L(+epsilon)
            nn5.weights[1][i,0] += epsilon
            _, L_plus, _ = nn5.forward(X,y)
            # compute L(-epsilon)
            nn5.weights[1][i,0] -= 2*epsilon
            _, L_minus,_ = nn5.forward(X,y)
            # recover weight
            nn5.weights[1][i,0] += epsilon
            # Finite Difference
            grad_diff[i] = (L_plus/epsilon-L_minus/epsilon) / 2
        res.append(np.max(np.abs(grad_theta - grad_diff)))

    plt.xlabel('N', weight='bold')
    plt.ylabel('Target', weight='bold')
    plt.title('maximum difference between the true gradient and the nite difference gradi
    plt.plot(np.arange(len(res)), res)
```

```
N values used: [1, 5, 10, 50, 1000, 5000, 100000, 500000]
```

```
Out[21]: [<matplotlib.lines.Line2D at 0x7f065ed4ff60>]
```



where x=(0, 1, 2, 3, ,4 ,5, 6, 7) corresponds to N= (1, 5, 10, 50, 1000, 5000, 100000, 500000), respectively.