FAET630004: AI-Core and RISC Architecture **(Due: 06/20/21)**

Homework Assignment #3

Instructor: Chixiao Chen *Name:* Chunyu Wang, *FudanID:* 20210860017

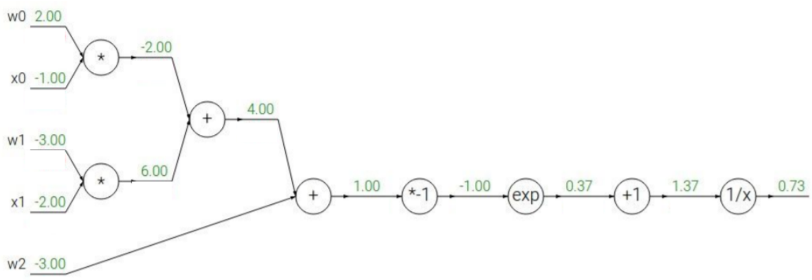
- This HW counts 15% of your final score, please treat it carefully.
- Please submit the electronic copy via mail: `faet_english@126.com` before 06/11/2020 11:59pm.
- It is encouraged to use LaTeX to edit it, the source code of the assignment is available via: `https://www.overleaf.com/read/mrhqrdztsdzs`
- You can also open it by Office Word, and save it as a .doc file for easy editing. Also, you can print it out, complete it and scan it by your cellphone.
- Problem 2 needs python and numpy. If you do not have a local python environment, please use an online version `https://colab.research.google.com/`.
- You can answer the assignment either in Chinese or English

Problem 1: Gradient Computing (30 points)

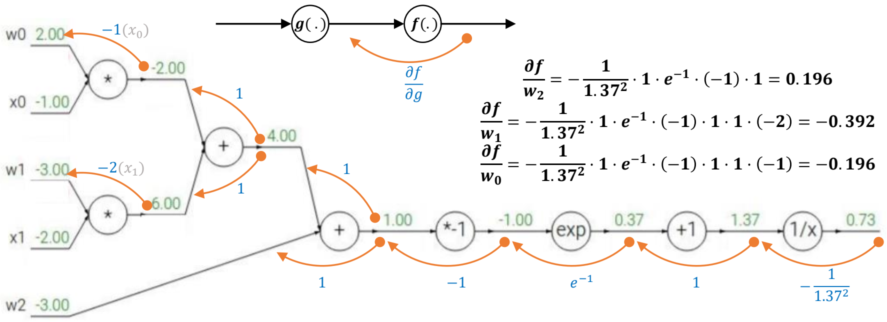
Assuming that one loss function in a classifier has the following output expression:

$$f(x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}},$$

and the current state is shown below:



Please compute all the weight gradients $\frac{\partial f}{\partial w_i}, i = 0, 1, 2$.



$$\frac{\partial f}{w_2} = -\frac{1}{1.37^2} \cdot 1 \cdot e^{-1} \cdot (-1) \cdot 1 = 0.196$$

$$\frac{\partial f}{w_1} = -\frac{1}{1.37^2} \cdot 1 \cdot e^{-1} \cdot (-1) \cdot 1 \cdot 1 \cdot (-2) = -0.392$$

$$\frac{\partial f}{w_0} = -\frac{1}{1.37^2} \cdot 1 \cdot e^{-1} \cdot (-1) \cdot 1 \cdot 1 \cdot (-1) = -0.196$$

Problem 2: Training a two-layer neural network using Numpy (70 points)

Assuming you have a tiny dataset which has 8 inputs, 4 classes and 500 samples. Please design a two-layer neural network as the classifier. Both forward (inference) and backward (training) propagation are required. The first 400 samples are for training, and the last 100 samples are for test. The dataset is available via: `https://cihlab.github.io/course/dataset.txt`. The activation function is ReLU in the case.

The following table is an example interpretation of the dataset file. (The first two lines of the file is illustrated.)

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	Class Label
0.4812	0.7790	0.8904	0.7361	0.9552	0.2119	0.7992	0.2409	4
0.4472	0.5985	0.7859	0.5035	0.6912	0.4038	0.0787	0.2301	1

Please submit you code and a brief report with the loss function definition, the final accuracy results, the neuron number in the hidden layers, etc. Also include your strategy for batch size and learning rate. (Hint: It is encouraged to use python and numpy (`https://www.numpy.org/`). You can refer to the slides 34 in the lecture 7 notes. The problem does not encourage you to use Tensorflow/caffe/pytorch, but if you have no idea about numpy, you can also using these frameworks.)

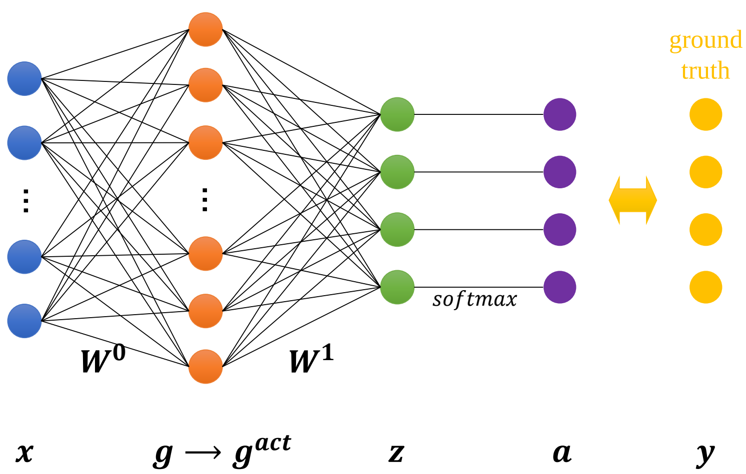


Figure 1: Network Structure

Network Structure A two-layer neural network as shwon in Fig.1 is implemented using pure `numpy`. I will give the definition of this network and the specific parameter settings. It has 32 neurons in the hidden layer (g). The entire network can be written in the following form:

$$\boldsymbol{a} = F(\boldsymbol{x}) = (\sigma(\boldsymbol{x}\boldsymbol{W}^{(0)} + \boldsymbol{b}^{(0)})\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)})$$

where, $\boldsymbol{x} \in \mathbb{R}^{1\times 8}$ is the input, $\boldsymbol{a} \in \mathbb{R}^{1\times 4}$ is the output of F. $\boldsymbol{y}$ is the ground truth in the form of onehot. $\boldsymbol{W}^{(0)} \in \mathbb{R}^{8\times H}, \boldsymbol{W}^{(1)} \in \mathbb{R}^{H\times 4}, \boldsymbol{b}^{(0)} \in \mathbb{R}^{1\times H}, \boldsymbol{b}^{(1)} \in \mathbb{R}^{1\times 4}$ are weights and bias, H indicates the number of neurons in hidden layer($H = 32$ in our experiment). $\sigma(\cdot)$ is sigmoid function or ReLU as activation function.

For ease of representation, we have the following definition:

$$\begin{aligned}
\boldsymbol{g} &= \boldsymbol{x}\boldsymbol{W}^{(0)} + \boldsymbol{b}^{(0)} \\
\boldsymbol{g}^{act} &= \sigma(\boldsymbol{g}) \\
\boldsymbol{z} &= \boldsymbol{g}^{act}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)} \\
\boldsymbol{a} &= S(\boldsymbol{z}) = softmax(\boldsymbol{z})
\end{aligned}$$

Loss Function we use cross entropy loss as the loss function:

$$L = -\sum_i \boldsymbol{y}_i \log \boldsymbol{a}_i$$

Gradient Calculation

$$
\begin{aligned}
&\frac{\partial L}{\partial \boldsymbol{z}_i} = \boldsymbol{a}_i - \boldsymbol{y}_i \\
\longrightarrow &\frac{\partial L}{\partial \boldsymbol{W}_{ij}^{(1)}} = \frac{\partial L}{\partial \boldsymbol{z}_j}\frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{W}_{ij}^{(1)}} = \frac{\partial L}{\partial \boldsymbol{z}_j}\boldsymbol{g}_i^{act} \\
\longrightarrow &\frac{\partial L}{\partial \boldsymbol{g}_i^{act}} = \sum_j \frac{\partial L}{\partial \boldsymbol{z}_j}\frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{g}_i^{act}} = \sum_j \frac{\partial L}{\partial \boldsymbol{z}_j}\boldsymbol{W}_{ij}^{(1)} \\
\longrightarrow &\frac{\partial L}{\partial \boldsymbol{g}_i} = \frac{\partial L}{\partial \boldsymbol{g}_i^{act}}\frac{\partial \boldsymbol{g}_i^{act}}{\partial \boldsymbol{g}_i} = \frac{\partial L}{\partial \boldsymbol{g}_i^{act}}\sigma_i^{-1} \\
\longrightarrow &\frac{\partial L}{\partial \boldsymbol{W}_{ij}^{(0)}} = \frac{\partial L}{\partial \boldsymbol{g}_j}\frac{\partial \boldsymbol{g}_j}{\partial \boldsymbol{W}_{ij}^{(0)}} = \frac{\partial L}{\partial \boldsymbol{g}_j}\boldsymbol{x}_i \\
\longrightarrow &\frac{\partial L}{\partial \boldsymbol{b}_i^{(1)}} = \frac{\partial L}{\partial \boldsymbol{z}_i}, \quad \frac{\partial L}{\partial \boldsymbol{b}_i^{(0)}} = \frac{\partial L}{\partial \boldsymbol{g}_i}
\end{aligned}
$$

Parameters Update

We use gradients of mini batch with `batch-size=16` to update parameter $\boldsymbol{\theta}$ ($\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(0)}, \boldsymbol{b}^{(0)}, \boldsymbol{b}^{(1)}$):

$$\boldsymbol{\theta}_i := \boldsymbol{\theta}_i - \alpha \frac{1}{m}\frac{\partial L}{\partial \boldsymbol{\theta}}_i$$

where m is `batch_size=16`, α is learning rate. We used an initial learning rate of 1 ($\alpha = 1$), and the learning rate is reduced to its half for every 100 epoch.

$$\alpha := \frac{1}{2}\alpha$$

Experiments & Results

The original dataset was randomly shuffled and then divided into 70% training set and 30% test set. During the training process, the average loss of each epoch on the test set and training set and the accuracy rate on the test set are shown in the figure below.

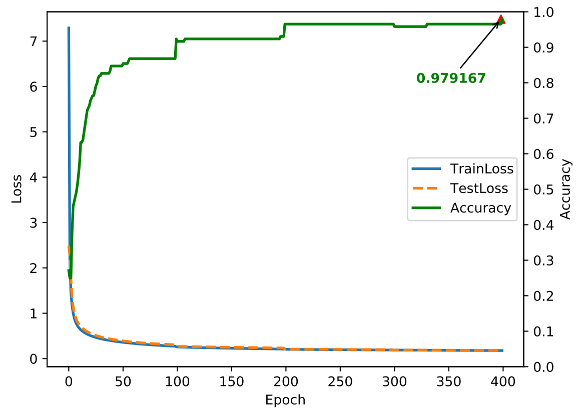


Figure 2: $\sigma = sigmoid$

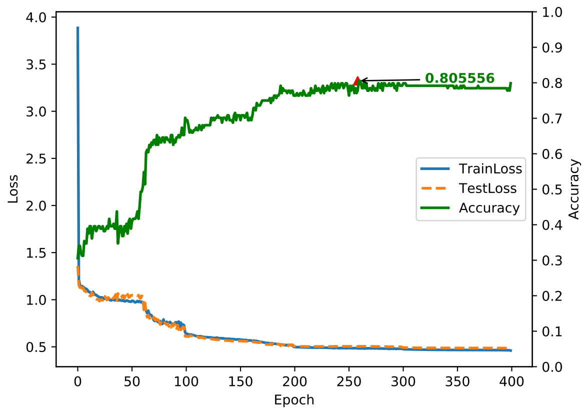


Figure 3: $\sigma = ReLU$

σ	hidden-size	batch-size	init-lr	accuracy
sigmoid	32	16	1.0	**0.979167**
ReLU	32	16	1e-1	0.805556

Sigmoid activation function performs better than ReLU under the same conditions, and the final accuracy reaches **0.979167**. Sigmoid performs better on shallow networks such as this task, while ReLU ss widely used in deeper networks with the ability of sparsity and handling vanishing gradient problem.

Source Code https://github.com/yiwangchunyu/AI-RISC/tree/hw3/HW3

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator

def ReLU(x):
    return np.maximum(x,0)

#The derivative of ReLU
def ReLU_d(x):
    return np.where(x > 0, 1, 0)

def sigmoid(x):
    return 1/(1+np.exp(-x))

#The derivative of sigmoid
def sigmoid_d(x):
    return sigmoid(x)*(1-sigmoid(x))

def softmax(z):
    t = np.exp(z)
    a = np.exp(z) / np.sum(t, axis=1).reshape(-1,1)
    return a

class DataLoader():
    def __init__(self,filename='dataset.txt',batch_size=16, shuffle=True, train=True,
     train_ratio=0.7):
        self.filename=filename
        self.batch_size=batch_size
        self.inputs,self.labels=self.load_data()
        if shuffle:
            self.shuffle()
        if train:
            self.inputs, self.labels = self.inputs[:int(self.inputs.shape[0]*train_ratio)],\
                                    self.labels[:int(self.labels.shape[0]*train_ratio)]
        else:
            self.inputs, self.labels = self.inputs[int(self.inputs.shape[0] * train_ratio):], \
                                    self.labels[int(self.labels.shape[0] * train_ratio):]
        self.length = self.inputs.shape[0] // self.batch_size


    def shuffle(self):
        shuffle_ix = np.random.permutation(np.arange(len(self.labels)))
        self.inputs = self.inputs[shuffle_ix]
        self.labels = self.labels[shuffle_ix]

    def load_data(self):
        inputs = []
        labels = []
        with open(self.filename, 'r') as f:
            for line in f:
                row = line.split()
                labels.append(int(row[-1]))
                inputs.append(list(map(float, row[:-1])))
        return np.array(inputs), np.array(labels)-1

    def __iter__(self):
        self.id = 0
        return self

    def __next__(self):
        if self.id<self.length:
            inputs = self.inputs[self.id*self.batch_size:(self.id+1)*self.batch_size]
```

```
            labels = self.labels[self.id * self.batch_size:(self.id + 1) * self.batch_size]
            self.id+=1
            return inputs,labels
        else:
            raise StopIteration


class Net():
    def __init__(self, input_size=8, hidden_size=32, bias=True, num_class=4, lr = None,
         act='sigmoid'):
        self.hidden_size = hidden_size
        self.input_size=input_size
        self.num_class=num_class
        self.bias=bias
        self.lr=lr
        # self.W0=np.zeros((self.input_size, self.hidden_size))
        # self.W1 = np.zeros((self.hidden_size,self.num_class))
        self.W1 = np.random.rand(self.hidden_size,self.num_class)
        self.W0 = np.random.rand(self.input_size, self.hidden_size)

        # self.b0=np.zeros((1,self.hidden_size))
        # self.b1=np.zeros((1,self.num_class))
        self.b0 = np.random.rand(1, self.hidden_size)
        self.b1 = np.random.rand(1, self.num_class)
        if act=='sigmoid':
            self.activate=sigmoid
            self.activate_d = sigmoid_d
            if not self.lr:
                self.lr=1.0
        elif act=='relu':
            self.activate = ReLU
            self.activate_d = ReLU_d
            if not self.lr:
                self.lr=1e-1
        else:
            exit(-1)

    # compute cross entropy loss
    def loss(self,a,y,reduction='mean'):
        self.y = np.eye(self.num_class)[y] # onehot
        loss = -np.sum(self.y*np.log(a),axis=1)
        if reduction=='mean':
            loss = np.sum(loss) / self.y.shape[0]
        return loss

    def forword(self,x): # x: batch_size x 8
        self.x=x
        self.g = self.x.dot(self.W0)+self.b0
        self.g_act = self.activate(self.g)
        self.z = self.g_act.dot(self.W1) + self.b1
        self.a = softmax(self.z)
        return self.a

    # calculate gradients
    def backword(self,y):
        self.y = np.eye(self.num_class)[y] # onehot
        self.grad_z=self.a-self.y
        self.grad_W1=self.g_act.T.dot(self.grad_z)/self.x.shape[0]
        self.grad_g_act=self.grad_z.dot(self.W1.T)
        self.grad_g=self.activate_d(self.g)*self.grad_g_act
        self.grad_W0=self.x.T.dot(self.grad_g)/self.x.shape[0]
        if self.bias:
            self.grad_b1=self.grad_z.copy()
```

```
            self.grad_b0 = self.grad_g.copy()

    # update params (gradient descent)
    def step(self, lr_shrink=1):
        lr=lr_shrink*self.lr
        self.W0=self.W0-lr*self.grad_W0
        self.W1 = self.W1 - lr * self.grad_W1
        if self.bias:
            self.b0=self.b0-lr*self.b0
            self.b1 = self.b1 - lr * self.b1

    def __call__(self, x):
        return self.forword(x)

def train():
    train_Loader=DataLoader()
    test_Loader = DataLoader(train=False)
    net=Net()
    train_losses,test_losses=[],[]
    pos,best_acc,accs=0,0,[]
    lr_shrink = 1
    for epoch in range(n_epoch):
        train_loss,test_loss=0,0
        if (epoch+1)%100==0:
            lr_shrink*=0.5
        for i,(inputs,labels) in enumerate(train_Loader):
            outputs=net(inputs)
            loss=net.loss(outputs,labels)
            train_loss+=loss
            net.backword(labels)
            net.step(lr_shrink)
        # test
        correct=0
        for i,(inputs,labels) in enumerate(test_Loader):
            outputs=net(inputs)
            loss=net.loss(outputs,labels)
            test_loss+=loss
            preds = np.argmax(outputs,axis=1)
            correct += (preds==labels).sum()

        # logging...
        train_loss/=train_Loader.length
        test_loss/=test_Loader.length
        train_losses.append(train_loss)
        test_losses.append(test_loss)
        acc=correct/(test_Loader.length*test_Loader.batch_size)
        if acc>best_acc:
            best_acc=acc
            pos=len(accs)-1
        accs.append(acc)
        print("epoch:%d, train_loss:%f, test_loss:%f, acc=%f (%d,%d), best_acc:%f" % (
            epoch, train_loss, test_loss,
            acc, correct, test_Loader.length*test_Loader.batch_size,best_acc))

    print('best accuracy:', best_acc)

    #plot
    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    plot11=ax1.plot(np.arange(0,len(train_losses)),train_losses
                ,linewidth = '2',label='TrainLoss')
    plot12=ax1.plot(np.arange(0, len(test_losses)), test_losses
                ,linewidth = '2',linestyle='--', label='TestLoss')
```

```
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')

    ax2 = ax1.twinx()
    plot2=ax2.plot(np.arange(0, len(accs)), accs
                  ,color='g',linewidth = '2',linestyle='-', label='Accuracy')
    ax2.set_ylabel('Accuracy')
    ax2.set_ylim(0,1)
    y_major_locator = MultipleLocator(0.1)
    ax2.yaxis.set_major_locator(y_major_locator)
    ax2.annotate('%f'%(best_acc),(pos,best_acc)
                ,xytext=(n_epoch*0.8,0.8),weight='heavy',color='g',
                arrowprops=dict(arrowstyle='->'))
    ax2.scatter(pos,best_acc,color='r',marker='^')
    lines=plot11+plot12+plot2
    ax1.legend(lines, [l.get_label() for l in lines],loc='center right')
    plt.savefig('loss.pdf', dpi=300)
    plt.show()

n_epoch=400
if __name__=="__main__":
    train()
```