ECE421 Introduction of Machine Learning

Assignment 2: Neural Networks

Zetong Zhao

1 Neural Networks with Numpy

Network structure :

a) 3 layers - 1 input, 1 hidden with ReLU activation and 1 output with Softmax

b) Cross Entropy Loss :$\mathcal{L} = -\sum_{k=1}^{K} y_k \log(p_k)$, where $\mathbf{y} = [y_1, y_2, \ldots, y_k]^T$ is the one-hot coded vector of the label.

1.1 Helper Functions

1) ReLU(): The activation equation is ReLU(x) = max(x,0).

```
def relu(x):
    return np.maximum(x,0)
```

Figure1: numpy relu function

2) softmax(): $\sigma(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{K}\exp(z_k)}$, j=1, ... , K for K classes. To prevent overflow while computing exponential, input numpy array should subtract the max value first.

```
def softmax(x):
    x = x - np.amax(x, axis=1, keepdims=True)
    return np.exp(x)/(np.sum(np.exp(x), axis=1, keepdims=True))
```

Figure2: numpy softmax function

3) compute():

```
def computeLayer(X, W, b):
    return np.matmul(X,W)+b
```

Figure3: numpy cumputeLayer function

4) averageCE():

Average CE = $= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)} \log(p_k^{(n)})$. N is the number of examples, $y_k^{(n)}$ is the one-hot label for sample n, $p_k^{(n)}$ is the softmax output for sample n in k^{th} class.

```
def CE(target, prediction):
    return (-1/target.shape[0])*np.sum(target*np.log(prediction))
```

Figure4: numpy relu function

5) gradCE():

$\mathcal{L} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)} \log(p_k^{(n)})$, where $\mathbf{p}$= softmax($\mathbf{o}$) $= \frac{\exp(\mathbf{o})}{\sum_{n=1}^{K}\exp(o_n)}$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{p}} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\frac{y_k}{p_k}\frac{\partial p}{\partial o}$$

$$\frac{\partial p_k}{\partial o_n} = \frac{\exp(o_n)}{\sum_{n=1}^{N}\exp(o_n)} - \frac{\exp(2o_n)}{(\sum_{n=1}^{N}\exp(o_n))^2} = \frac{\exp(o_n)}{\sum_{n=1}^{N}\exp(o_n)}\left(1 - \frac{\exp(o_n)}{\sum_{n=1}^{N}\exp(o_n)}\right) = p_k(1-p_k) \text{ (if } n = k),$$

$$\frac{\partial p_k}{\partial o_n} = -\frac{\exp(o_k+o_n)}{(\sum_{n=1}^{N}\exp(o_n))^2} = -p_n p_k \text{ (if } n \neq k)$$

Apply chain rule:

$$\frac{\partial \mathcal{L}}{\partial o} = \frac{\partial \mathcal{L}}{\partial p}\frac{\partial p}{\partial o}$$

$$= \frac{1}{N}\sum_{n=1}^{N}\frac{\partial \mathcal{L}}{\partial p_n}\frac{\partial p_n}{\partial o_n} = \frac{1}{N}\sum_{n=1}^{N} - \sum_{k=1}^{K}\frac{y_k}{p_k}\frac{\partial p_k}{\partial o_n}$$

$$= -\frac{1}{N}\sum_{n=1}^{N}\left(\frac{y_n}{p_n}\frac{\partial p_n}{\partial o_n} + \sum_{k\neq n}^{K}\frac{y_k}{p_k}\frac{\partial p_k}{\partial o_n}\right)$$

$$= -\frac{1}{N}\sum_{n=1}^{N}[y_n(1-p_n)) - \sum_{k\neq n}^{K} y_k p_n]$$

$$= -\frac{1}{N}\sum_{n=1}^{N}[y_n - p_n\sum_{k=1}^{K} y_k] = -\frac{1}{N}\sum_{n=1}^{N}(y_n - p_n)$$

$$= \frac{1}{N}(\mathbf{p} - \mathbf{y})$$

```
def gradCE(target, prediction):
    return (softmax(prediction) - target)/target.shape[0]
```

Figure5: numpy gradCE function

1.2 Backpropagation Derivation

$\mathbf{x_o}$ is the input layer, $\mathbf{w}_h$ and b_h are weight and bias of the hidden layer. $s_h = \mathbf{w}_h\mathbf{x}_o + b_h$, $s_o = \mathbf{w}_o\mathbf{x}_h + b_o$. The structure is shown in figure below.

$$x_i \xrightarrow[b_h]{W_h} s_h \xrightarrow{ReLU} x_h \xrightarrow[b_o]{W_o} s_o \xrightarrow{Softmax} \mathcal{L}$$

Figure6: general structure of the network

With gradient of cross-entropy calculated above, let $\mathbf{G}_{CE} = \frac{\partial \mathcal{L}}{\partial s_o} = \mathbf{x_o} - \boldsymbol{y}^T$

1) $\frac{\partial \mathcal{L}}{\partial w_o}$:

$$\frac{\partial s_o}{\partial w_o} = \mathbf{x}_h{}^T$$

Apply chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_o} = \frac{\partial \mathcal{L}}{\partial s_o}\frac{\partial s_o}{\partial w_o} = \mathbf{x}_h{}^T\mathbf{G}_{CE}$$

2) $\frac{\partial \mathcal{L}}{\partial \mathrm{b_o}}$:

$$\frac{\partial \mathrm{s_o}}{\partial \mathrm{w_o}} = \mathbf{1}$$

Apply chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathrm{b_o}} = \frac{\partial \mathcal{L}}{\partial \mathrm{s_o}} \frac{\partial \mathrm{s_o}}{\partial \mathrm{b_o}} = \mathbf{G}_{\mathrm{CE}}$$

3) $\frac{\partial \mathcal{L}}{\partial \mathrm{w_h}}$:

$$\frac{\partial \mathrm{s_o}}{\partial \mathrm{x_h}} = \mathbf{w_o}^T$$

$$\frac{\partial \mathrm{x_h}}{\partial \mathrm{s_h}} = \begin{cases} 1\,, if\ \mathrm{z_h} > 0 \\ 0, \qquad else \end{cases}$$

$$\frac{\partial \mathrm{s_h}}{\partial \mathrm{w_h}} = \mathbf{x_i}$$

Apply chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathrm{w_h}} = \frac{\partial \mathcal{L}}{\partial \mathrm{s_o}} \frac{\partial \mathrm{s_o}}{\partial \mathrm{x_h}} \frac{\partial \mathrm{x_h}}{\partial \mathrm{s_h}} \frac{\partial \mathrm{s_h}}{\partial \mathrm{w_h}} = \mathbf{G}_{\mathrm{CE}} \mathbf{w_o}^T \frac{\partial \mathrm{x_h}}{\partial \mathrm{s_h}} \mathbf{x_i}$$

4) $\frac{\partial \mathcal{L}}{\partial \mathrm{b_h}}$:

$$\frac{\partial \mathrm{s_o}}{\partial \mathrm{x_h}} = \mathbf{w_o}^T$$

$$\frac{\partial \mathrm{x_h}}{\partial \mathrm{s_h}} = \begin{cases} 1\,, if\ \mathrm{z_h} > 0 \\ 0, \qquad else \end{cases}$$

$$\frac{\partial \mathrm{s_h}}{\partial \mathrm{w_h}} = \mathbf{1}$$

Apply chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathrm{w_h}} = \frac{\partial \mathcal{L}}{\partial \mathrm{s_o}} \frac{\partial \mathrm{s_o}}{\partial \mathrm{x_h}} \frac{\partial \mathrm{x_h}}{\partial \mathrm{s_h}} \frac{\partial \mathrm{s_h}}{\partial \mathrm{w_h}} = \mathbf{G}_{\mathrm{CE}} \mathbf{w_o}^T \frac{\partial \mathrm{x_h}}{\partial \mathrm{s_h}}$$

According to the calculations above, backpropagation is implemented by:

```
def backprop(xi, xh, w, target, prediction):
    gradce = gradCE(target, prediction)
    dwo = np.dot(np.transpose(xh),gradce) #10000,1000   100000,10
    dbo = np.transpose(sum(gradce)).reshape(1, 10)
    dwh = np.dot(np.transpose(xi),np.where(xh > 0, 1, 0)*np.matmul(gradce,np.transpose(w)))
    dbh = sum(np.where(xh > 0, 1, 0) * np.dot(gradce, np.transpose(w))).reshape(1, 1000)
    return dwo,dbo,dwh,dbh
```

Figure7: numpy backprop function

1.3 Learning

Construct the following neural network for training:

- Weight matrices follows Xaiver initialization scheme (zero-mean Gaussian with variance $\frac{2}{units\ in+out}$)
- Bias are assigned to zeroes
- Optimization: Gradient Descent with momentum

$$\mathbf{\upsilon}_{\text{new}} \leftarrow \gamma \mathbf{\upsilon}_{\text{old}} + \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \mathbf{\upsilon}_{\text{new}}$$

where γ is set to 0.9, and α is set to 0.1.

Train accuracy	Valid accuracy	Train loss	Valid loss
0.9768	0.9128	0.11	0.3048

Table1: training results from the neural network

```
def learning():
    trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
    trainData = trainData.reshape((trainData.shape[0], -1))
    validData = validData.reshape((validData.shape[0], -1))
    testData = testData.reshape((testData.shape[0], -1))
    newtrain, newvalid, newtest = convertOneHot(trainTarget, validTarget, testTarget)

    epoch=200
    H=1000
    F=trainData.shape[1]
    gamma=0.9
    alpha=0.1
    xi = trainData
    wo = np.random.normal(0, np.sqrt(2/(H+10)), (H,10))
    wh = np.random.normal(0, np.sqrt(2/(F+H)), (F,H))
    bo = np.zeros((1,10))
    bh = np.zeros((1,H))
    train_loss = []
    valid_loss = []
    train_acc = []
    valid_acc = []
    test_acc = []

    dwh = np.zeros((F,H))
    dwo = np.zeros((H,10))
    dbh = np.zeros((1,H))
    dbo = np.zeros((1,10))

    vwh = np.full((F,H),1e-5)
    vwo = np.full((H,10),1e-5)
    vbh = np.full((1,H),1e-5)
    vbo = np.full((1,10),1e-5)

    sh = np.zeros((10000,1000))
    so = np.zeros((10000,10))
    sh_ = np.zeros((6000,1000))
    so_ = np.zeros((6000,10))
    for i in range(epoch):

        sh = computeLayer(xi, wh, bh)
        xh = relu(sh)
        so = computeLayer(xh, wo, bo)
        yo = softmax(so)
        train_loss.append(CE(newtrain,yo))
        compare = np.equal(np.argmax(yo,axis=1),np.argmax(newtrain,axis=1))
        train_accuracy = np.sum((compare==True))/(trainData.shape[0])
        train_acc.append(train_accuracy)
        print("epoch", i, ": accuracy = ",train_accuracy)
```

Figure8: learning() function implementation - part1

```
        sh_  =  computeLayer(validData,  wh,  bh)
        xh_  =  relu(sh_)
        so_  =  computeLayer(xh_,  wo,  bo)
        valid_pre  =  softmax(so_)
        valid_loss.append(CE(newvalid,valid_pre))
        compare_valid  =  np.equal(np.argmax(valid_pre,axis=1),np.argmax(newvalid,axis=1))
        valid_accuracy  =  np.sum((compare_valid==True))/(validData.shape[0])
        valid_acc.append(valid_accuracy)
        dwo,dbo,dwh,dbh  =  backprop(xi,  xh,  wo,  newtrain,  so)

        if  (i==epoch-1):
            print("train_acc  is",train_accuracy)
            print("train_loss  is",train_loss)
            print("valid_acc  is",valid_accuracy)
            print("valid_loss  is",valid_loss)

        vwh  =  gamma*vwh  +  alpha*dwh
        vwo  =  gamma*vwo  +  alpha*dwo
        vbh  =  gamma*vbh  +  alpha*dbh
        vbo  =  gamma*vbo  +  alpha*dbo
        wo  =  wo  -  vwo
        wh  =  wh  -  vwh
        bo  =  bo  -  vbo
        bh  =  bh  -  vbh

    plt.plot(range(epoch),train_loss)
    plt.plot(range(epoch),valid_loss)
    plt.ylabel('Loss')
    plt.xlabel('Epochs')
    plt.title('Train  and  Validation  Loss',  fontsize=16)
    plt.show()

    print(train_acc)
    plt.plot(range(epoch),train_acc)
    plt.plot(range(epoch),valid_acc)
    plt.ylabel('Accuracy')
    plt.xlabel('Epochs')
    plt.title('Train  and  Validation  Accuracy',  fontsize=16)
    plt.show()
```

Figure9: learning() function implementation - part2

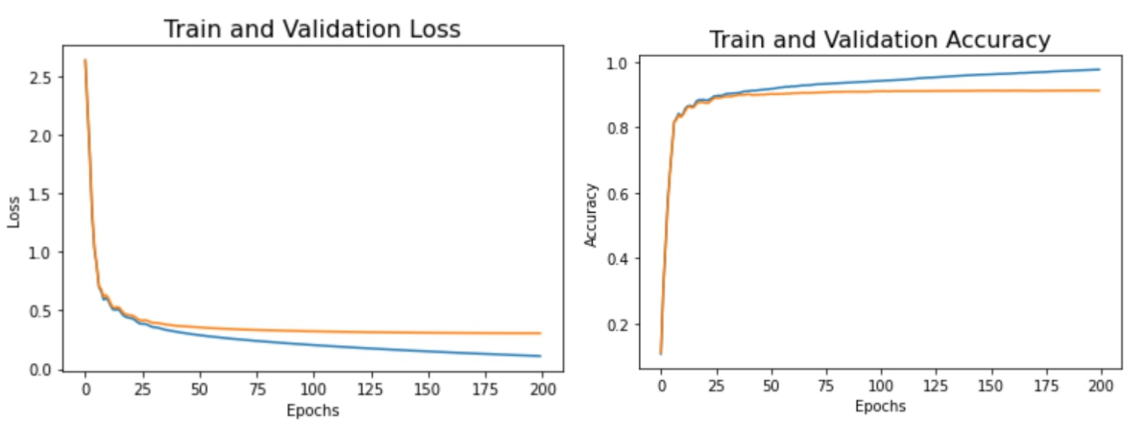

Figure10: training results: loss (left), accuracy (right)