ECE421 A2 Report

Helper Functions

1. Relu ()

```
# s(l) - sigma -> x(l)
def relu(x):
    sigma = np.maximum(x,0) #Element-wise maximum of array elements.
    return sigma
```

2. Softmax()

```
def softmax(x):
    # In order to prevent overow while computing exponentials
    # subtract the maximum value of z from all its elements.
    x = x - np.amax(x, axis=1, keepdims=True)
    numerator = np.exp(x)
    demoninator = (np.sum(np.exp(x), axis=1, keepdims=True))
    res = numerator/demoninator
    return res
```

3. Compute()

```
def compute(x, w, b):
    # This function will accept 3 arguments: a weight matrix, an input vector, and a
    # bias vector and return the product between the weights and input, plus the biases
    res = np.matmul(x,w)+b
    return res
```

4. Average_ce()

```
def average_ce(target, prediction):
    #target yk(n)
    #prediction pk(n)
    matrix = target*np.log(prediction) #point_wise multiply y*log(p)
    res = (-1/target.shape[0])*np.sum(matrix)
    return res
```

5. Grad_ce()

Apply chain rule to the loss function to compute $\frac{dL}{do}$

$$\mathcal{L} = -\sum_{k=1}^{K} y_k \log p_k$$

for data n: $y = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$ $k \rightarrow$ $\quad p = \begin{bmatrix} \; \end{bmatrix}$ $k \rightarrow$ ← probability for class k $\quad o \in \mathbb{R}^k \quad p \in \mathbb{R}^k$

$$p = \text{softmax}(\vec{o}) = \frac{e^{\vec{o}}}{\sum_{k=1}^{K} e^{o_k}}$$

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

Compute $\frac{dL}{dp}$

$$\frac{\partial \mathcal{L}}{\partial p} = -\begin{bmatrix} \frac{y_1}{p_1} \\ \vdots \\ \frac{y_k}{p_k} \end{bmatrix}$$

Now try to compute $\frac{dp}{do}$

$$\frac{\partial P}{\partial O} = \begin{bmatrix} \frac{\partial P_1}{\partial O_1} & \cdots & \frac{\partial P_1}{\partial O_K} \\ \vdots & & \\ \frac{\partial P_K}{\partial O_1} & \cdots & \frac{\partial P_K}{\partial O_K} \end{bmatrix}$$

$$\frac{\partial P_i}{\partial O_j}: \quad \text{if } i=j \quad P_i = \frac{e^{O_i}}{\sum_{k=1}^{K} e^{O_k}} \quad \frac{\partial P_i}{\partial O_i} = \frac{e^{O_i}\left(\sum_{k=1}^{K} e^{O_k}\right) - e^{O_i} \cdot e^{O_i}}{\left(\sum_{k=1}^{K} e^{O_k}\right)^2}$$

$$= \frac{e^{O_i}}{\left(\sum_{k=1}^{K} e^{O_k}\right)} \cdot \left(1 - \frac{e^{O_i}}{\left(\sum_{k=1}^{K} e^{O_k}\right)}\right)$$

$$= P_i \cdot (1 - P_i)$$

$$\text{if } i \neq j \quad P_i = \frac{e^{O_i}}{\sum_{k=1}^{K} e^{O_k}} \quad \frac{\partial P_i}{\partial O_j} = - \frac{e^{O_i} \cdot e^{O_j}}{\left(\sum_{k=1}^{K} e^{O_k}\right)^2}$$

$$= - P_i \cdot P_j$$

Compute $\frac{dL}{do}$

$$\frac{\partial L}{\partial O} = \frac{\partial P}{\partial O} \cdot \frac{\partial L}{\partial K}$$

$$= - \begin{bmatrix} \frac{\partial P_1}{\partial O_1} & \cdots & \frac{\partial P_1}{\partial O_K} \\ \vdots & & \\ \frac{\partial P_K}{\partial O_1} & \cdots & \frac{\partial P_K}{\partial O_K} \end{bmatrix} \begin{bmatrix} \frac{y_1}{P_1} \\ \vdots \\ \frac{y_K}{P_K} \end{bmatrix}$$

$$\frac{\partial L}{\partial O}(i) = - \sum_{k=1}^{K} \frac{\partial P_i}{\partial O_k} \cdot \frac{y_k}{P_k}$$

$$= \sum_{k=1}^{K} (P_i \cdot P_k) \frac{y_k}{P_k} - P_i \cdot P_i \frac{y_i}{P_i} - P_i(1-P_i)\frac{y_i}{P_i}$$

$$= \sum_{k=1}^{K} P_i \cdot y_k - y_i$$

$$= P_i - y_i$$

$$\therefore \frac{\partial L}{\partial O} = P - y$$

```
def grad_ce(target, logits):
    # gradient of the cross entropy loss withrespect to the inputs to the softmax function
    p = softmax(logits)
    res = (p - target)/target.shape[0]
    return res
```

Backpropagation Derivation

Structure of the layers

$$x \longrightarrow S_h \xrightarrow{\text{ReLu}} X_h \longrightarrow S_o \xrightarrow{\text{Soft max}} \hat{y}$$

$$sh = W_h^T x + b_h \qquad S_o = W_o^T xh + b_o$$

$F = 784$ $\qquad$ $H = 1000$ $\qquad$ $K = 10$

1. $\frac{\partial L}{\partial Wo}$

$$\frac{\partial \mathcal{L}}{\partial W_o} = \frac{\partial \mathcal{L}}{\partial S_o} \cdot \frac{\partial S_o}{\partial W_o}$$

From grade_CE : $\frac{\partial \mathcal{L}}{\partial S_o} = \text{grade CE}$

$$\frac{\partial S_o}{\partial W_o} = X_h^T$$

$$\therefore \frac{\partial \mathcal{L}}{\partial W_o} = X_h^T \cdot \text{grade CE}$$

```
def DL_Dwo(xh, target, prediction):
    softmax_ce = grad_ce(target, prediction)
    xh_transpose = np.transpose(xh)
    Dwo = np.matmul(xh_transpose,softmax_ce)
    return Dwo
```

2. $\frac{\partial L}{\partial bo}$

$$\frac{\partial \mathcal{L}}{\partial b_o} = \frac{\partial \mathcal{L}}{\partial S_o} \cdot \frac{\partial S_o}{\partial b_o}$$

$$\frac{\partial S_o}{\partial b_o} = 1$$

$$\therefore \frac{\partial \mathcal{L}}{\partial W_o} = 1 \cdot \text{grade CE}$$

```
def DL_Dbo(target, prediction):
    softmax_ce = grad_ce(target, prediction)
    Dbo = np.transpose(sum(softmax_ce)).reshape(1, 10) #K=10
    return Dbo
```

3. $\frac{\partial L}{\partial Wh}$

$$\frac{\partial \mathcal{L}}{\partial Wh} = \frac{\partial \mathcal{L}}{\partial So} \cdot \frac{\partial So}{\partial Xh} \cdot \frac{\partial Xh}{\partial Sh} \cdot \frac{\partial Sh}{\partial Wh}$$

$$\frac{\partial \mathcal{L}}{\partial So} = \text{grade CE}$$

$$\frac{\partial So}{\partial Xh} = Wo^T$$

$$\frac{\partial Xh}{\partial Sh} = \begin{cases} 1 & sh > 0 \rightarrow Xh > 0 \\ 0 & sh < 0 \rightarrow Xh < 0 \end{cases}$$

$$\frac{\partial Sh}{\partial Wh} = X$$

$$\frac{\partial \mathcal{L}}{\partial Wh} = \text{gradeCE} \cdot W_o^T \cdot \frac{\partial X_h}{\partial Sh} \cdot X$$

```
def DL_Dwh(xi, xh, wo, target, prediction):
    DL_Dso = grad_ce(target, prediction)
    Dso_Dxh = np.transpose(wo)
    Dxh_Dsh = np.where(xh > 0, 1, 0)
    Dsh_Dwh = np.transpose(xi)
    dwh = np.matmul(Dsh_Dwh,Dxh_Dsh*np.matmul(DL_Dso,Dso_Dxh))
    return dwh
```

4. $\frac{\partial L}{\partial bh}$

$$\frac{\partial \mathcal{L}}{\partial bh} = \frac{\partial \mathcal{L}}{\partial So} \cdot \frac{\partial So}{\partial Xh} \cdot \frac{\partial Xh}{\partial Sh} \cdot \frac{\partial Sh}{\partial bh}$$

$$\frac{\partial \mathcal{L}}{\partial S_o} = \text{gradeCE}$$

$$\frac{\partial So}{\partial Xh} = W_o^T$$

$$\frac{\partial Xh}{\partial Sh} = \begin{cases} 1 & sh > 0 \rightarrow xh > 0 \\ 0 & sh < 0 \rightarrow xh < 0 \end{cases}$$

$$\frac{\partial sh}{\partial bh} = 1$$

$$\frac{\partial \mathcal{L}}{\partial bh} = \text{gradeCE} \quad Wo^T \quad \frac{\partial Xh}{\partial Sh}$$

```
def DL_Dbh(xi, xh, wo, target, prediction):
    DL_Dso = grad_ce(target, prediction)
    Dso_Dxh = np.transpose(wo)
    Dxh_Dsh = np.where(xh > 0, 1, 0)
    Dsh_Dwh = 1
    dbh = sum(Dxh_Dsh * np.dot(DL_Dso, Dso_Dxh)).reshape(1, 1000)#H
    return dbh
```

Back propagation function:

```
def backprop(xi, xh, w, target, prediction):
    dl_dwo = DL_Dwo(xh, target, prediction)
    dl_dbo = DL_Dbo(target, prediction)
    dl_dwh = DL_Dwh(xi, xh, w, target, prediction)
    dl_dbh = DL_Dbh(xi, xh, w, target, prediction)
    return dl_dwo,dl_dbo,dl_dwh,dl_dbh
```

Learning

Processing data from load_data

```
def Process_Data():
    # 10000*28*28 6000*28*28  2720*28*28
    #[trainData,  validData,  testData, trainTarget, validTarget, testTarget
    dataList = load_data()
    dataList = list(dataList)

    # 10000*784   6000*784   2720*784
    # [trainData, validData, testData]
    for i, data in enumerate(dataList[:3]):
        dataList[i] = data.reshape(len(data), -1)

    trainData  = dataList[0] #10000*784
    validData  = dataList[1] #6000*784
    testData   = dataList[2] #2720*784

    trainTarget= dataList[3]
    validTarget= dataList[4]
    testTarget = dataList[5]

    #one hot encoding of Target
    TargetList = convert_onehot(trainTarget, validTarget, testTarget)
    TargetList = list(TargetList)

    trainTarget= TargetList[0]
    validTarget= TargetList[1]
    testTarget = TargetList[2]



    print("trainData:",trainData.shape)
    print("validData:",validData.shape)
    print("testData:",testData.shape)

    print("trainTarget:",trainTarget.shape)
    print("validTarget:",validTarget.shape)
    print("testTarget:",testTarget.shape)


    return trainData,validData,testData,trainTarget,validTarget,testTarget
```

Initialization of data

```
def initialize_weight(F,H,K):
    wo = np.random.normal(0, np.sqrt(2/(H+K)), (H,K))
    bo = np.zeros((1,K))
    wh = np.random.normal(0, np.sqrt(2/(F+H)), (F,H))
    bh = np.zeros((1,H))
    return wo,bo,wh,bh

def initialize_V_matrix(F,H,K):
    #initialize them to the same size as the hidden and output layer weight matrix sizes
    #with a very small value (e.g. 1e-5).
    V_wo = np.full((H,K),1e-5)
    V_bo = np.full((1,K),1e-5)
    V_wh = np.full((F,H),1e-5)
    V_bh = np.full((1,H),1e-5)
    return V_wo,V_bo,V_wh,V_bh

def initialize_sh_so(N,F,H,K):
    sh = np.zeros((N,H))
    so = np.zeros((N,K))
    return sh, so
```

Helper functions

```
def foward_prop(xi,wh,bh,wo,bo):
    sh = compute(xi, wh, bh)
    xh = relu(sh)
    so = compute(xh, wo, bo)
    yo = softmax(so)
    return yo,xh,so

def compute_loss_and_accuracy(Target,predicton,Data):
    loss  = average_ce(Target,predicton)
    #check if the prediction is same as target
    compare = np.equal(np.argmax(predicton,axis=1),np.argmax(Target,axis=1))
    accuracy = np.sum((compare==True))/(Data.shape[0])
    return loss, accuracy
```

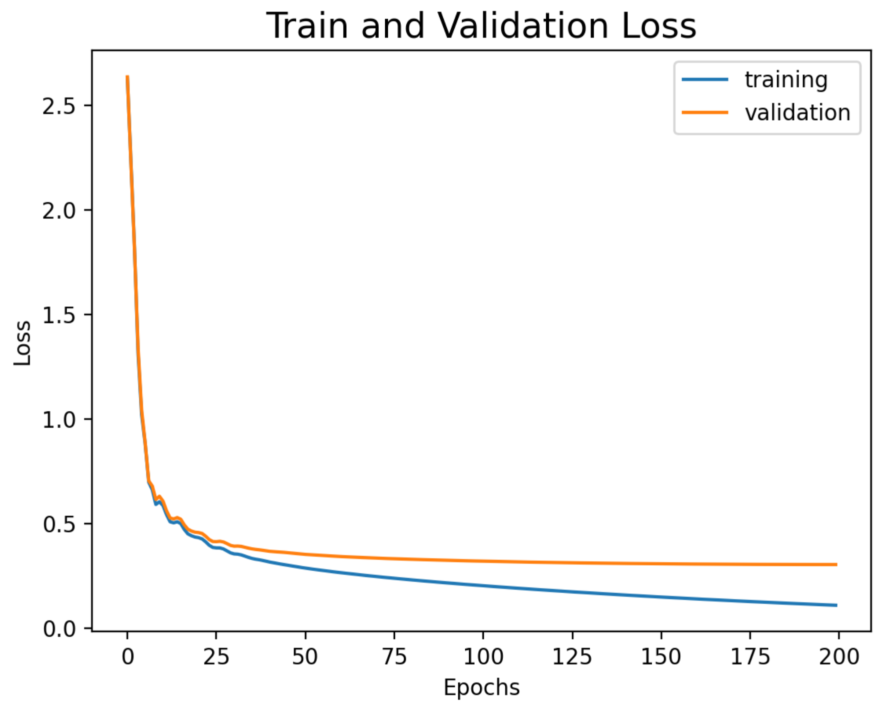
Learning Function

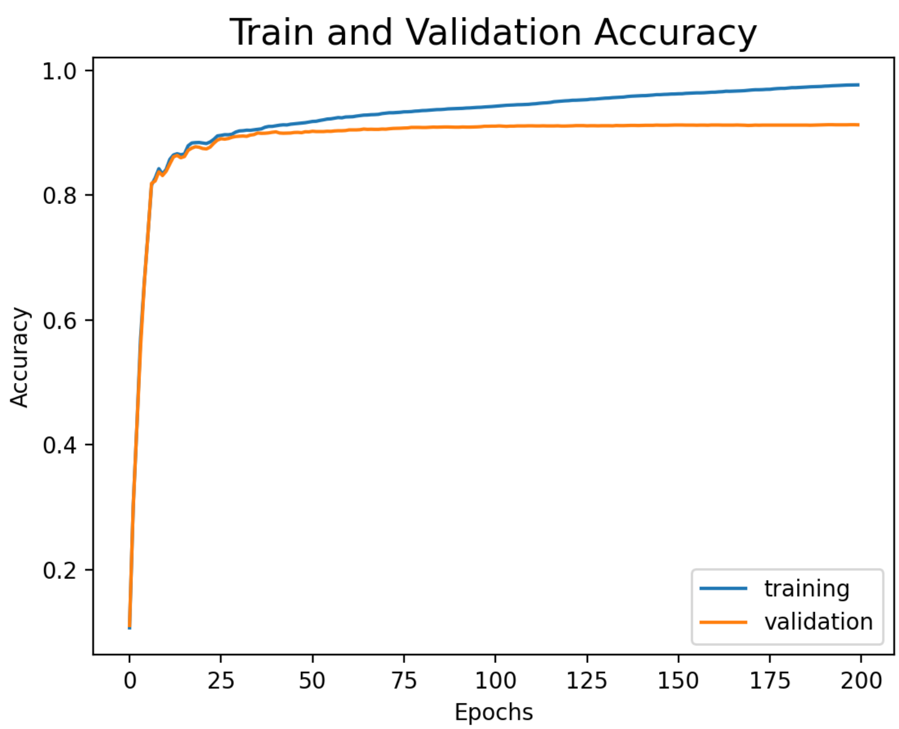
```
#================================== global Variable  ===================================#
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []

epoch=200

# define Dimension
F= 784
H= 1000
K = 10

# define learning rate and momentum
momentum =0.9
learn_rate=0.1
```

```python
#================================= Learning  ===================================#
def Train_Network():
  trainData,validData,testData,trainTarget,validTarget,testTarget = Process_Data()

  wo,bo,wh,bh = initialize_weight(F,H,K)
  V_wo,V_bo,V_wh,V_bh = initialize_V_matrix(F,H,K)
  Train_sh,Train_so = initialize_sh_so(10000,F,H,K)
  Valid_sh,Valid_so = initialize_sh_so(6000,F,H,K)


  for i in range(epoch):
    #training set foward
    yo,xh,so = foward_prop(trainData,wh,bh,wo,bo)
    loss, accuracy = compute_loss_and_accuracy(trainTarget,yo,trainData)
    train_loss.append(loss)
    train_acc.append(accuracy)

    #validation set forward
    yo_,xh_,so_ = foward_prop(validData,wh,bh,wo,bo)
    loss, accuracy = compute_loss_and_accuracy(validTarget,yo_,validData)
    valid_loss.append(loss)
    valid_acc.append(accuracy)

    #print("epoch =",i, "train loss=",train_loss[-1],"train acc=",train_acc[-1])

    #back propagation
    dwo,dbo,dwh,dbh = backprop(trainData, xh, wo, trainTarget, so)

    V_wo = momentum *V_wo + learn_rate*wo
    wo = wo - V_wo

    V_bo = momentum *V_bo + learn_rate*dbo
    bo = bo - V_bo

    V_wh = momentum *V_wh + learn_rate*wh
    wh = wh - V_wh

    V_bh = momentum *V_bh + learn_rate*dbh
    bh = bh - V_bh
```

Loss and Accuracy

Train Loss	Validation Loss	Train Accuracy	Validation Accuracy
0.110056	0.304798	0.9768	0.91283


Train and Validation Loss
training
validation
Loss
Epochs
0.0
0.5
1.0
1.5
2.0
2.5
0
25
50
75
100
125
150
175
200


Train and Validation Accuracy
training
validation
Accuracy
Epochs
0.2
0.4
0.6
0.8
1.0
0
25
50
75
100
125
150
175
200