Fundamental of Neural Networks
Final Project 1

Shao Hsuan Hung (1723219)
s.hung@student.tue.nl

January 22, 2023

1 Multi-Layer Perceptron

1.1 Implementation

- Description of the project: (1.) Implemented in the 5LSH0_MLP.py (2) Use the .csv to store the MNIST dataset, I already attached the file in the ./dataset
- The code of the in question 4 (to improve the performance of the NN) is attached in the Improved_MLP.ipynb
- Fully connected layer with bias

```python
import numpy as np
class MLP(object):
    def __init__(self,layer_dimension,layer_act_fun):
        '''
        :parameters
            layer_dimension: list of dimension of each layer, including
            the input/output layer, e.g.: [784,128,64,32,10]
            layer_act_fun: list of string of name of activation function
        '''
        self.layer_dimension = list(layer_dimension)
        self.layer_act_fun = list(layer_act_fun)
        self.num_layer = len(self.layer_dimension)
        # Initialize the weight and bias, neuron and neuron before activation
        self.W = [0]*(self.num_layer)
        self.B = [0]*(self.num_layer)
        self.Z = [0]*(self.num_layer)
        self.A = [0]*(self.num_layer)
        # Derivate of weighs, bias, the output layer
        self.dW = [0]*(self.num_layer)
        self.dB = [0]*(self.num_layer)
        self.dA_output = [0]*(self.num_layer)
        # Random assign the inital weigh and bias
        for layer_idx in range(1,self.num_layer):
            self.W[layer_idx] = np.random.randn(self.layer_dimension[layer_idx],self.
    layer_dimension[layer_idx-1])*0.01
            self.B[layer_idx] = np.zeros((self.layer_dimension[layer_idx],1))
        # Data log
        self.train_acc = []
        self.valid_acc = []
        self.loss = []
```

Listing 1: Implement Class of the Fully connected multi-layer perceptron

- ReLU and Sigmoid activations

```python
@staticmethod
def ReLU(z,df = False):
    if df == False:
        return (0 < z)*z
    else:
        return (0<z)*1
@staticmethod
def sigmoid(z,df = False):
```

```
    f = 1/(1+np.exp(-z)+1e-8) # add 1e-8 to avoid overflow
    if df == False:
        return f
    else:
        return f*(1-f)
```

Listing 2: Implement ReLU and Sigmoid activation function

- Mini-batch Gradient Descent (SGD) and Cross-Entropy loss.

```
class MLP ( object ):
    def train(self,X,Y,X_valid,Y_valid,epochs,lr,mini_batch_size):
        m = X.shape[1]
        for one_epoch in range(epochs):
            cost = 0
            # generate the new mini-batch every epoch
            mini_batch_idx_list = self.mini_batch_shuffle(m,mini_batch_size)
            for idx in range(0,len(mini_batch_idx_list)):
                X_mini = X[:,mini_batch_idx_list[idx]]
                Y_mini = Y[:,mini_batch_idx_list[idx]]
                self.forward_propogation(X_mini)
                loss = self.loss_function(Y_mini) #Cross-Entropy loss
                self.back_propogation()
                self.update_parameters(lr) # SGD
                cost = cost+loss
            # Do the inference to estimate the accuracy
            self.train_acc.append(self.acc_calculation(X,np.argmax(Y).reshape((1,-1))))
            self.valid_acc.append(self.acc_calculation(X_valid,Y_valid))
            self.loss.append(cost)
            # In end of the epoch:
            if(one_epoch%10 == 0):
                print("Epoch",one_epoch,", loss:",cost,\
                    ",Train Acc:",self.train_acc[one_epoch],\
                    ",Valid Acc:",self.valid_acc[one_epoch])
    def loss_function(self,Y):
        output = self.A[self.num_layer-1] # Get the output's value
        # Calculate the cross entropy loss
        if self.layer_act_fun[self.num_layer-1]=="Softmax":
            loss = -np.sum(np.sum(Y*np.log(output+1e-8),keepdims=True))/(Y.shape[0])
            dA = Y/(output+1e-8)# calcualte the dA for backpropogation
        else: # logistic cost
            loss = -np.sum(Y*np.log(output+1e-8)+(1-Y)*np.log(1-output))/Y.shape[0]
            dA = (-(Y/output)+((1-Y)/(1-output)))
        self.dA_output[self.num_layer-1] = dA
        return loss
    def update_parameters(self,lr): #SGD
        for layer in range(1,self.num_layer):
            self.W[layer] = self.W[layer]-lr*self.dW[layer]
            self.B[layer] = self.B[layer]-lr*self.dB[layer]
    def forward_propogation(self,x):
        self.A[0] = x
        for layer_idx in range(1,self.num_layer):
            self.Z[layer_idx],self.A[layer_idx] =  self.layer_forward_calculation(\
            self.A[layer_idx-1],self.W[layer_idx],self.B[layer_idx],\
            self.layer_act_fun[layer_idx])
        return self.A[self.num_layer-1]
    def back_propogation(self):
        for i in reversed(range(1,self.num_layer)):
            self.dA_output[i-1],self.dW[i],self.dB[i] = self.layer_backward_calculation(\
            self.W[i],self.B[i],self.Z[i],self.A[i-1],self.A[i],self.dA_output[i],\
            self.layer_act_fun[i])
    def layer_forward_calculation(self,x,w,b,acti_fun):
        Z = np.dot(w,x)+b
        if acti_fun == "ReLU":
            A = self.ReLU(Z)
        elif acti_fun == "Sigmoid":
            A = self.sigmoid(Z)
```

```
        elif acti_fun == "Softmax":
            A = self.softmax(Z)
        else:
            A = self.softmax(Z)
        return Z,A
    def layer_backward_calculation(self,w,b,z,A_pervious,A,dA,activation):
        if activation=="ReLU":
            G = self.ReLU(z,df=True)
            dZ = dA*G
        elif activation=="Sigmoid":
            G = self.sigmoid(z,df=True)
            dZ = dA*G
        elif activation=="Softmax":
            dZ = A*(1-dA)
        dW = np.dot(dZ,A_pervious.transpose())/z.shape[0]
        dB = np.sum(dZ,keepdims=True)/z.shape[0]
        dA_previous = np.dot(w.transpose(),dZ)
        return dA_previous,dW,dB
```

Listing 3: Mini-batch Gradient Descent and Cross-Entropy loss

1.2 Training and testing

(1). Train the MLP and report the training and test accuracies. Explain why there are differences in training and test performance?

To train the MLP, as shown bebelow, selected the learning rate equal to 0.01, mini batch size 32, and train for 200 epochs.

```
layer_dim = [784,128,64,32,10]
avti=[None,"ReLU","ReLU","ReLU","Softmax"]
network = MLP(layer_dim,avti)
network.train(train_img,train_label,valid_img,valid_label,\
              epochs=200,lr=0.01,mini_batch_size=32)
```

Listing 4: Build the MLP instance

The Figure 1 shows the loss and accurcay of the model after training for 200 epochs. The training dataset contains 60,000 MNIST images, the testing dataseet contains 42,000 MNIST images. The training accuracy is 99.6%, the testing accuracy is 97.1%

It shows that the training accuracy is slightly larger than the testing accuracy. Since the neurons and parameters are trained and updated for the train dataset. When feeding the testing set in the neural network, instead of learning from the testing data (update parameter),the network only output the predicted result. So the model would have lower accuracy on the testing dataset.

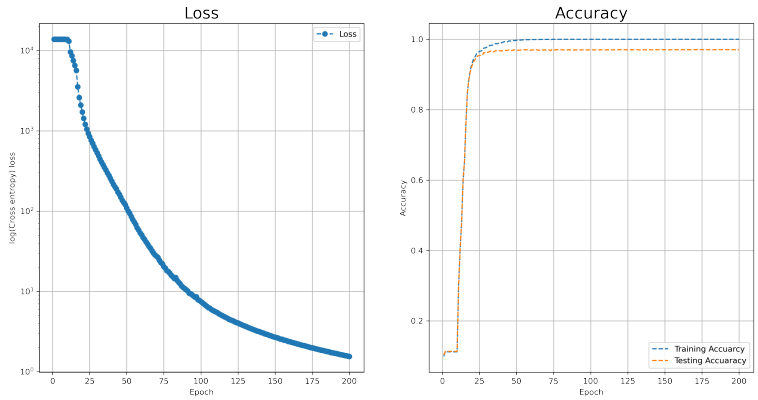

Figure 1: Performance of the self-implement MLP. (a) loss, (b) Accuracy of training and testing set

(2). Calculate the number of trainable parameters of your model (you can implement it or calculate by hand) and explain how you have estimated it?

Trainable parameters of the model are: weight and bias. The dimension of the model trained in the previous question are:

- Input layer: 784
- hidden layer1: 128
- hidden layer2: 64
- hidden layer3: 32
- Output layer: 10

For previous layer pass to the hidden layers, there is one weight between two neurons. So there are $784 \times 128 + 128 \times 64 + 64 \times 32 + 32 \times 10 = 110,912$ weights. Except for the input layer, each neuron has one biases. So there are $128 + 64 + 32 + 10 = 234$ biases.

(3). How would the number of parameters vary if you use a convolution layer with batch normalization layer instead of a fully connected layer (explain)? What is the relation between a convolution and fully-connected layer, when are they the same/different?

There are 2 differences from parameters perspective When using the convolution layer with bath normalization layer. First, the neurons in the convolution layer are not "densely" connected together, not all input nodes connect all output nodes. The connected nodes are determined by the convolution process and size of the convolution kernel. So the number of weights in the convolution layer is a lot smaller than the fully connected layer. Secondly, if using the batch normalization layer, there would have 4 type of parameters more, which are 2 learnable parameters (β and γ) and 2 non-learnable parameters (mean moving average and variance moving average).

On the other hand, there are 2 difference between a convolution layer and the fully-connected(FC) layer. First, in the FC layer, every neuron in the output is connected to every input neuron, while in the convolution layer, the neurons are not densely connected but are connected only to neighboring neurons within the width of the convolution kernel. Seconldy, the convolution layer share the weight, but the FC layer doesn't. In an FC layer, every output neuron is connected to every input neuron through a specific weight. However, in a convolution layer, the weights are shared among different neurons by the convolution kernel.

(4).Use the knowledge acquired in the course or sources online to improve the performance on the MNIST dataset. Try to achieve an accuracy of more than what a standard CNN can achieve (say more than 98.5-99.0%). You can use PyTorch for this question.

(a) Explain in detail the improvements and changes that you have added. Well reasoned answers will get more/full points. I add the following methods to improve the performance of model: (1) convolution layer, (2) batch normalization in every layer, (3) increase the batch size (128) combine with drop out layer (p = 0.5) to improve the performance of model, and (4) data augmentation on the Mnist dataset by randomly rotating the training set by -15 to 15 degree. The final result validation accuracy reach 99.56%.

The reason that the convolution layer can improve the accuracy in image processing is that the data type are 2D image, it helps to extract more spatial information. For 2D image, each pixel have strong correlation with the nearby pixels. The correlation will reduce with respect to the euclidean distance of the two pixels. Therefore, the convolution can improve

Second, since the batch normalization help to utilize the information from other batched, it would help improve the accuracy.

Third, since I add more layer in the model, so I add the drop out layer and increase the batch size to make the computation more efficient and also avoid the overfitting issue.

Four, I did the data augmentation on the training dataset by randomly rotating the angle of the image from -15 to 15 image to make the dataset close to the real world case (in the real application, the hand-written digits may be rotated by some angle.

(b) Given that you have unlimited resources for computation and no extra data, how would you improve

performance on the MNIST dataset? Hint: Try data augmentations, different networks and/or losses. Check for tricks online!

1. Data Augmentation: I can use data augmentation like cropping, rotating, distortion, etc, to have more and more data to ensure the generalization.

2. Transfer learning: I think another good way is to use other dataset with **similar features.** Then the model can learned transfer knowledge from other data set or models like human learning process.

3. Parallel computing: If I have unlimited computation resources, I would tuning as much as hyper parameters at the same time parallel, to find the optimal model.

2 Loss function

1. Derive the derivative of the output y with respect to logits z.
Assume z_i and z_j are arbitrary elements in the K elements-vector z, output y_i and y_j are the output value of the corresponding nueron, so the derivative of the output y_j with respect to z_i is:

$$\frac{\partial y_j}{\partial z_i} = \frac{\partial \sigma(z_i)}{\partial z_i} = \frac{\partial}{\partial z_j}\left(\frac{e^{z_j}}{\sum_{x=1}^{K} e^{z_x}}\right) \tag{1}$$

Let $f(z) = e^{z_j}$, $g(z) = \sum_{x=1}^{K} e^{z_j}$ so the output y_j with respect to z_i can be represented into:

$$\frac{\partial y_j}{\partial z_i} = \frac{\partial}{\partial z_j}\left(\frac{f(z)}{g(z)}\right) = \frac{f(z)'g(z) - f(z)g(z)'}{g(z)^2} \tag{2}$$

The $f(z)'$ have different derivative depends on whether the index (i,j) of the output y and logits z are the same:

$$f(z') = \begin{cases} \frac{\partial e^{z_j}}{\partial z_i} &= e^{z_i}, \quad \text{When } i = j \\ \frac{\partial e^{z_j}}{\partial z_i} &= 0, \quad \text{When } i \neq j \end{cases} \tag{3}$$

$$g(z') = \frac{\partial(\sum_{x=1}^{K} e^{z_x})}{\partial z_i} = e^{z_i} \tag{4}$$

So for $i = j$:

$$\frac{\partial y_i}{\partial z_i} = \frac{e^{z_i}(\sum_{x=1}^{K} e^{z_x}) - e^{z_i}e^{z_i}}{(\sum_{x=1}^{K} e^{z_x})^2} = \left(\frac{e^{z_i}}{(\sum_{x=1}^{K} e^{z_x})}\right)\left(\frac{(\sum_{x=1}^{K} e^{z_x}) - e^{z_i}}{(\sum_{x=1}^{K} e^{z_x})}\right) = \sigma(z_i)(1 - z_i) \tag{5}$$

For $i \neq j$:

$$\frac{\partial y_j}{\partial z_i} = \frac{0 \cdot (\sum_{x=1}^{K} e^{z_x}) - e^{z_j} \cdot e^{z_i}}{(\sum_{x=1}^{K} e^{z_x})^2} = \frac{-e^{z_i}}{\sum_{x=1}^{K} e^{z_x}} \cdot \frac{e^{z_j}}{\sum_{x=1}^{K} e^{z_x}} = -\sigma(z_i)\sigma(z_j) \tag{6}$$

So the derivative of the output y with respect to logits z is :

$$\frac{\partial y_j}{\partial z_i} = \begin{cases} \sigma(z_i)(1 - \sigma(z_i)) &= (y_i)(1 - y_i) \quad , \quad \text{When } i = j \\ -\sigma(z_i)\sigma(z_j) &= -y_i y_j \quad , \quad \text{When } i \neq j \end{cases} \tag{7}$$

2. Use condition when i = j, and derive the derivative of the loss function L_{CE} with respect to the logits z.

$$\frac{\partial L_{CE}}{\partial z_i} = \frac{\partial L_{CE}}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_i} = \frac{\partial(-t_j \log(y_j) - (1 - t_j)\log(1 - y_i))}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_i} \tag{8}$$

Where t is one hot encoded vector for the labels, N is the number of the classes. Since the t_j is constant, so for the derivative of the loss to the y_j is just derivative of the $log(y_i)$:

$$\frac{\partial L_{CE}}{\partial z_i} = \left(\frac{-t_j}{y_j} + \frac{1 - t_j}{1 - y_j}\right) \cdot \frac{\partial y_j}{\partial z_i} = \begin{cases} \left(\frac{-t_j}{y_j} + \frac{1-t_j}{1-y_j}\right) \cdot (y_j)(1 - y_j) &= y_i - t_i \quad , \quad \text{When } i = j \\ \left(\frac{-t_j}{y_j} + \frac{1-t_j}{1-y_j}\right) \cdot (-y_i y_j) &= \frac{y_i(t_j - y_j)}{1-y_j} \quad , \quad \text{When } i \neq j \end{cases} \tag{9}$$

So for the condition $i = j$, the derivative of the cross entropy loss to the legits z is:

$$\frac{\partial L_{CE}}{\partial z} = y - t \tag{10}$$

3.Show that $\frac{\partial L_{dice}}{\partial z} = \frac{y-1}{2y} \times (1 - L_{dice})^2$

Assume the condition $i = j$, the derivative of the dice loss function to the z becomes the derivative of the dice loss function to the y and derivative of output y to the z:

$$\frac{\partial L_{dice}}{\partial z} = \frac{\partial L_{dice}}{\partial y} \cdot \frac{\partial y}{\partial z} \tag{11}$$

The derivative of the dice loss function to the y and the derivative of output y to the z (assume i = j) are shown in below respectively:

$$\frac{\partial L_{dice}}{\partial y} = \frac{(-2t)(y+t) - (-2yt)}{(y+t)^2} = \frac{-2t^2}{(y+t)^2} \tag{12}$$

$$\frac{\partial y}{\partial z} = y(y-1) \tag{13}$$

So the derivative of the dice loss function to the z becomes:

$$\frac{\partial L_{dice}}{\partial z} = \frac{-2t^2 y(y-1)}{(y+t)^2} = (\frac{1}{2})(\frac{y-1}{y})\frac{(4y^2t^2)}{(y+t)^2} = \frac{y-1}{2y} \cdot (1 - L_{dice})^2 \tag{14}$$

4. Expand the generic to the binary version of focal loss.

Based on the reference in the question sheet, the focal loss is basically adding a factor $(1 - p_t)^\gamma$ on the cross entropy loss, where the p_t is the model's estimated probability for the class, γ is a tunable parameter. Thus, a binary focal loss is the binary cross entropy with the factor,(When $\gamma = 0$, then the binary focal loss would become to binary cross entropy loss):

$$L_{binary\,focal} = -\left[(1-y)^\gamma \cdot t \cdot \log(y) + (y)^\gamma (1-t)\log(1-y)\right] \tag{15}$$

5. Show that

$$\frac{\partial L_{focal}}{\partial y} = \frac{-1}{y(1-y)}(t(1-y)^\gamma(\gamma\epsilon_y + 1 - y) + (t-1)y^\gamma(\gamma\epsilon_{1-y} + y)) \quad \text{Where} \epsilon_p = -p\log(p)$$

Taking the derivative of the binary focal loss to the model output y:

$$\frac{\partial L_{focal}}{\partial y} = \frac{\partial(-(1-y)^\gamma \cdot t \cdot \log(y) - (y)^\gamma(1-t)\log(1-y))}{\partial y} = \frac{\partial(f(y) + g(y))}{\partial y} = \frac{\partial f(y)}{\partial y} + \frac{\partial g(y)}{\partial y} \tag{16}$$

Where $f(y)$ and $g(y)$ are shown in the equation 17 and 18 in order to simplify the derivative:

$$f(y) = -(1-y)^\gamma \cdot t \cdot \log(y) \tag{17}$$

$$g(y) = -(y)^\gamma \cdot (1-t)\log(1-y) \tag{18}$$

The derivative of the $f(y)$ and $g(y)$ to the output y is :

$$\frac{\partial f(y)}{\partial y} = \gamma(1-y)^{\gamma-1} \cdot t \cdot \log(y) - (1-y)^\gamma \cdot \frac{t}{y} = \frac{-1}{y(1-y)}\left[t \cdot (1-y)^\gamma \cdot (-\gamma y log(y) + 1 - y)\right] \tag{19}$$

$$\frac{\partial g(y)}{\partial y} = -\gamma y^{\gamma-1}(1-t)\log(1-y) + y^\gamma\frac{(1-t)}{1-y} = \frac{-1}{y(1-y)}\left[(t-1)y^\gamma(-\gamma(1-y)\log(1-y)) + y)\right] \tag{20}$$

Substitute the $-y\log(y)$ and the $-(1-y)\log(1-y)$ with the ϵ_y and ϵ_{1-y}, then the derivative of the binary focal loss to the model output y:

$$\frac{\partial L_{focal}}{\partial y} = \frac{-1}{y(1-y)}\left[t(1-y)^\gamma(\gamma\epsilon_y + 1 - y) + (t-1)y^\gamma(\gamma\epsilon_{1-y} + y)\right] \tag{21}$$

6. Under what condition is

$$\frac{\partial L_{focal}}{\partial y} = \frac{\partial L_{ce}}{\partial y}$$

What happens when γ value is too high or low in focal loss? Explain in detail how it would impact performance?

When $\gamma = 0$, the binary focal loss would become to binary cross entropy loss, since the $y^{\gamma} = 1$, $\gamma\epsilon_y = 0$ and $\gamma\epsilon_{1-y} = 0$. In the focal loss, γ controls the shape of the curve in the below figure 2. The higher the value of γ, the lower the loss for well-classified examples, and the loss of the hard-to-classified examples would become relative larger,thus we could turn the attention of the model more towards 'hard-to-classify examples. The higher γ extends the range in which an example receives low loss. Therefore, when the value of γ is too high (meaning that the focal loss is too low at the same time), that means the loss of the classes that have extremely small dataset would be large. In the training process, the model would mainly focus on hard-to-classified classes. On the other hand, when the value of γ is low, that means the behavior of the focal loss would be similar to the cross entropy loss.

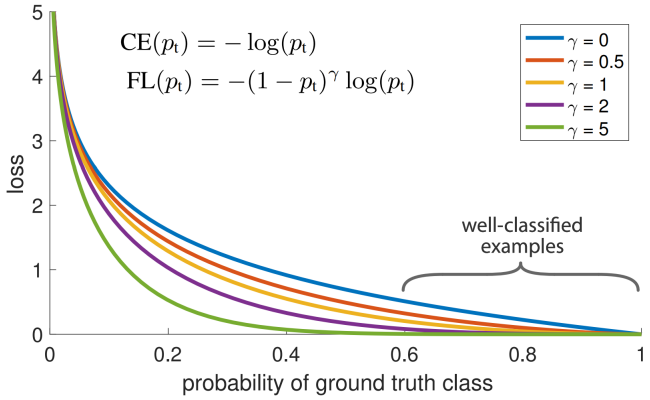

Figure 2: Loss versus probability of ground truth class with different value γ[1]

7. List the applications where each loss function would be useful and explain why?

Cross entropy loss are widely use in the classification in the neuronal network / convolution neuronal network. Compare with the mean square error loss, the cross entropy loss have larger gradient when on the error surface. Thus in the optimization prospective, the cross entropy always help the model converge to the minimum.

The dice loss is often applied in the image or instance segmentation problem, since in the image or instance, there are less foreground and more background in the dataset. If one use cross entropy loss, the algorithm may predict most of the pixel as background even when they are not and still get low errors. In this case, the dice loss consider the intersection and the union of the output predictions, regardless the effect accuracy of true negative (well classified the background), which we can reflect the true performance of the accuracy on the foreground.

Focal loss is often applied when there exist relative low number of the data in a class (class imbalance), like dense object in the object detection. By tuning the parameter γ, this loss function can make the well-classified classes having lower loss value, and make the hard-to-classified classes having relative high loss value, so the model can focus on the hard-to-classified classes more when tuning parameters.

8. Discuss (in detail) the behavior of precision and recall for each of the listed loss functions above. Use the proofs from questions 2, 3 and 5 to support your answers.

Generally speaking, for all the loss function listed above, for both true positive (TP) and false negative (FN), their label are both equal to 1 (t = 1) for all loss function. For $t = 1$, and the output y is close to zero, the TP is close to zero, then the precision and the recall are zero. For $t = 1$ and y close to 1, the false positive goes to zero, the precision would close to 1. For $t = 0$, and y is close to zero, the false positive is zero, leading the precision become 1. For $t = 0$ and y is close to 1, the true positive is zero, the precision would be zero.

For different loss function, the probability that the model predict the TP, TN, FP, and FN would be different when the class imbalance issue happened. For image case, the number of foreground is always less than the

number of background, which case the class imbalance, the cross entropy loss would have issue with having many false negatives and many true positive because it can get most answer right by just predicting there is no object in the image. So for the cross-entropy loss, the recall score would be very low. On the other hand, the dice loss function helps improve upon cross entropy by penalizing false negatives more, so as focal loss function. So the dice loss and focal loss would result a higher recall score, but possibly have a lower precision because the network will learn to make more predictions.

References

[1] Tsung-Yi Lin et al. "Focal Loss for Dense Object Detection". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2999–3007. DOI: 10.1109/ICCV.2017.324.