Assignment-5
20170937 강제순

October 31, 2019

```
In [1]: '''

        Submitter : JESOON KANG, 20170937
        Date : 2019. 10.


            Assignment 4.

        -   -



        '''
        import matplotlib.pyplot as plt
        import math
        import numpy as np
        import random

        import torch
        from torch.utils.data import Dataset, DataLoader
        import torchvision.transforms as transforms

        import torchvision
        import os
```

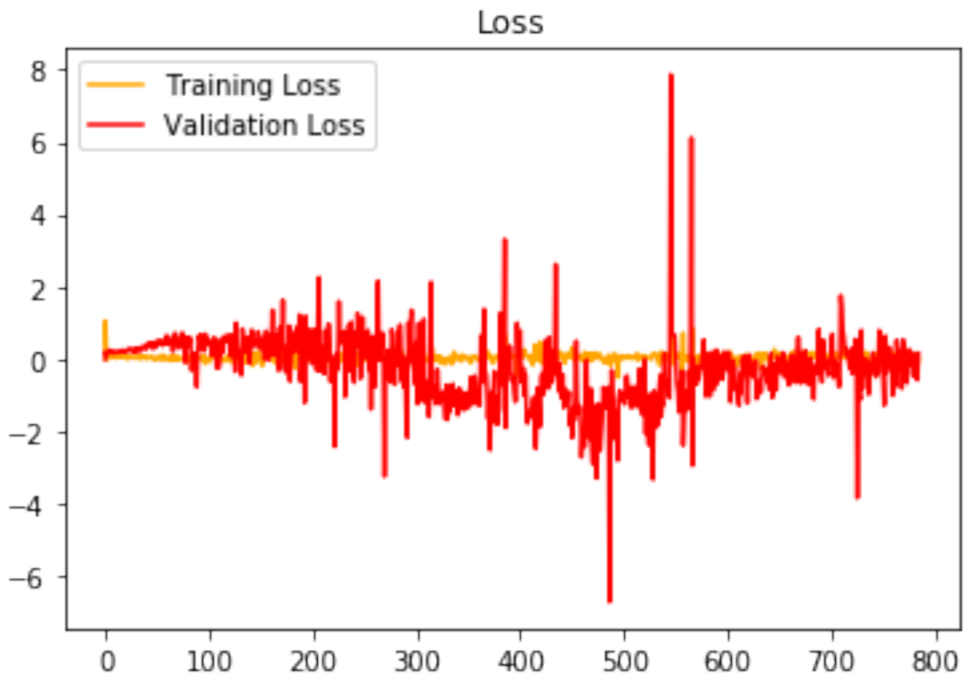
```
In [2]: #### Section 1. #### This Section is bringed Data_import_ex.py file.

        # Image Data import & resize

        transform = transforms.Compose([#transforms.Resize((256,256)),
                                        transforms.Grayscale(),                # the code tran
                                        transforms.ToTensor(),])

        #train_data_path = 'relative path of training data set'
        train_data_path = '../data/horse-or-human/train'
        trainset = torchvision.datasets.ImageFolder(root=train_data_path, transform=transform)
        # change the valuse of batch_size, num_workers for your program
        # if shuffle=True, the data reshuffled at every epoch
        trainloader = torch.utils.data.DataLoader(trainset, batch_size=100, shuffle=True, num_
        validation_data_path = '../data/horse-or-human/validation'
        valset = torchvision.datasets.ImageFolder(root=validation_data_path, transform=transfo
        # change the valuse of batch_size, num_workers for your program
        valloader = torch.utils.data.DataLoader(valset, batch_size=100, shuffle=True, num_worke

        #### Section 1 END ####
```

```
In [3]: # sigmoid Function.
        def sigmoid(z) :
            return 1 / (1 + torch.pow(math.e,-z)+0.000001)

        def tanh(z) :
            ret = (2 / (1 + torch.pow(math.e,-2*z) + 0.000001)) - 1
            return ret

        def derv_tanh(z) :
            ret = (1 - (tanh(z)**2))
            return ret

        def relu(z) :
            ret = z.clone().detach()
            #print(ret)
            tmp = torch.zeros_like(ret)
            ret = torch.where(ret<=0,tmp,ret)
            return ret

        def derv_relu(z) :
            ret = z.clone().detach()
            tmp_1 = torch.zeros_like(ret)
            tmp_2 = torch.ones_like(ret)
            ret = torch.where(ret<0,tmp_1,tmp_2)
            #print(ret)
            return ret


        def leakly_relu(z) :
            ret = z.clone().detach()
            tmp1 = torch.ones_like(ret)
            ret = torch.where(ret<0,ret*0.01,ret)
            return ret

        def derv_leakly_relu(z) :
            ret = z.clone().detach()
            tmp1 = torch.ones_like(ret)
            ret = torch.where(ret<0, tmp1*0.01,tmp1)
            return ret

        def get_activation(z,type) :
            if (type == 0) :
                return sigmoid(z)
            elif (type == 1) :
                return tanh(z)
            elif (type == 2) :
                return relu(z)
            elif (type == 3) :
                return leakly_relu(z)

            else :
                print("Error, get_activation")
                return 0

        def get_derv_activation(z,type) :
            if (type == 0) :
                return sigmoid(z)*(1-sigmoid(z))
            elif (type == 1) :
                return derv_tanh(z)
            elif (type == 2) :
                return derv_relu(z)
            elif (type == 3) :
                return derv_leakly_relu(z)
            else :
                print("Error, get_derv_activation")
                return 0
```

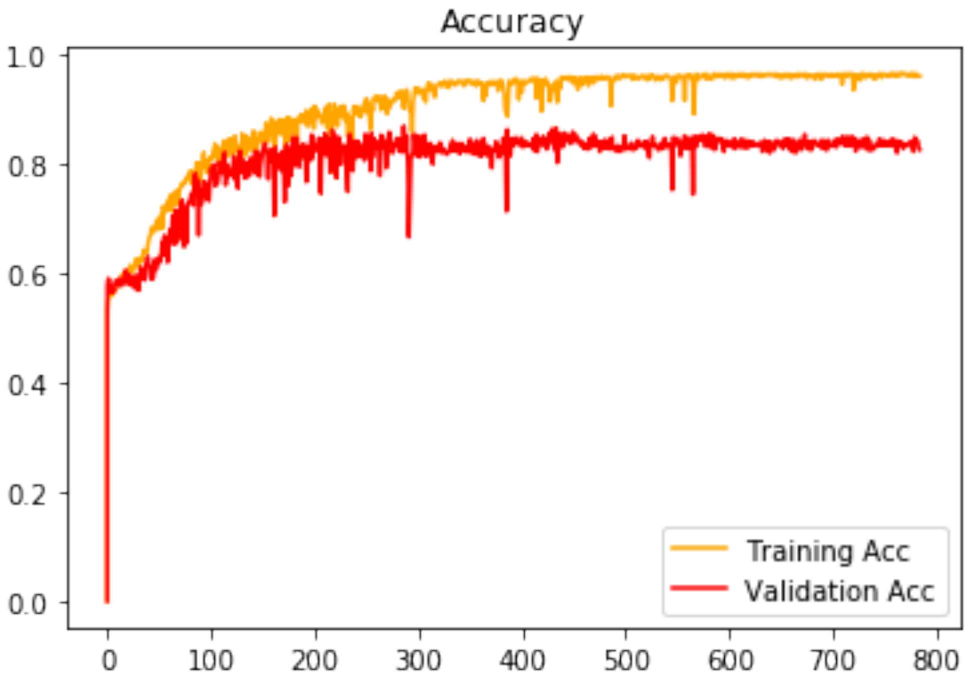
```
In [22]: def get_loss(y,a) :
             #print(y,a)
             ret = -(torch.div(y,a+0.000001) - torch.div(1-y,1-a+0.000001))

             return ret

         BATCH_SIZE = 100
         FEATURE_SIZE = 10000
         class ML :
             def __init__(self,act1,act2,act3,L1_size,L2_size,L3_size,lr,min_loss_diff) :
                 self.act_type_1 = act1
                 self.act_type_2 = act2
                 self.act_type_3 = act3
                 self.l1_size = L1_size
                 self.l2_size = L2_size
                 self.l3_size = L3_size
                 self.feature_size = 10000
                 self.epoch = 0
                 self.lr = lr
```

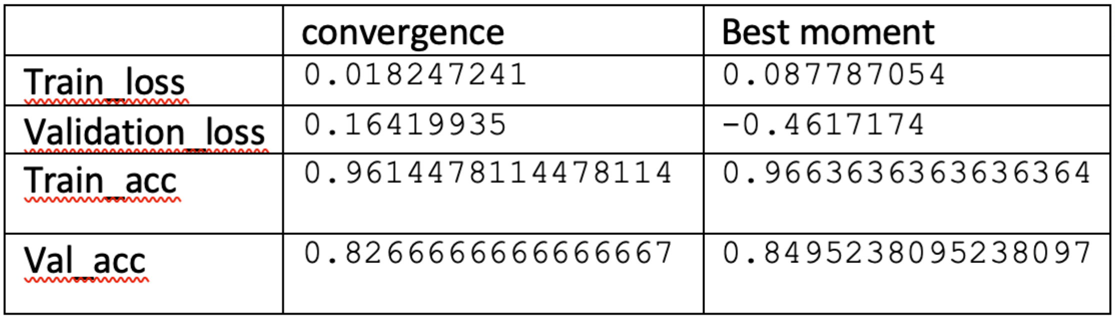
```
        self.min_loss_diff = min_loss_diff

        self.init_weights()

        print("ML Object initialized")
        self.iter = 0

        self.val_acc_log = []
        self.val_loss_log = []
        self.train_acc_log = []
        self.train_loss_log = []
        self.epoch_log = []
        self.val_acc_log.append(0)
        self.val_loss_log.append(0)
        self.train_acc_log.append(0)
        self.train_loss_log.append(0)
        self.epoch_log.append(0)

    def init_weights(self) :
        self.w_1 = torch.FloatTensor(self.l1_size,self.feature_size).uniform_(-1,1)
        self.b_1 = torch.FloatTensor(1,self.l1_size).uniform_(-1,1)

        #torch.FloatTensor(a, b).uniform_(r1, r2)

        self.w_2 = torch.FloatTensor(self.l2_size,self.l1_size).uniform_(-1,1)
        self.b_2 = torch.FloatTensor(1,self.l2_size).uniform_(-1,1)

        self.w_3 = torch.FloatTensor(self.l3_size,self.l2_size).uniform_(-1,1)
        self.b_3 = torch.FloatTensor(1,self.l3_size).uniform_(-1,1)




    def training(self) :

        epoch = 0
        for epoch in range(0,1000000):
            train_acc_log_tmp = []
            train_loss_log_tmp = []
            val_acc_log_tmp = []
            val_loss_log_tmp = []


        # load training images of the batch size for every iteration
            for i, data in enumerate(trainloader):

                # inputs is the images
                # labels is the class of the image
                inputs, labels = data
                inputs, labels = inputs.float() , labels.float()
                # if you don't change the image size, it will be [batch_size, 1, 100,

                # if labels is horse it returns tensor[0,0,0] else it returns tensor[

                self.t_data_batch = torch.squeeze(torch.FloatTensor(data[0]),3)

                self.t_data_batch = self.t_data_batch.view(len(data[0]),FEATURE_SIZE)
                self.z_1 = torch.matmul(self.t_data_batch,self.w_1.T) + self.b_1
                #print(self.z_1)
                self.a_1 = get_activation(self.z_1,self.act_type_1)
                #print(self.a_1)

                self.z_2 = torch.matmul(self.a_1,self.w_2.T) + self.b_2

                self.a_2 = get_activation(self.z_2,self.act_type_2) # 1027 x 50

                self.z_3 = torch.matmul(self.a_2,self.w_3.T) + self.b_3
                #print(self.z_3)
                self.a_3 = get_activation(self.z_3,self.act_type_3) # 1027 x 1

                #print(self.a_3)
                self.t_yh_batch = data[1].float().unsqueeze(1)

                #print(self.a_3,self.t_yh_batch)
                acc = self.get_acc(self.a_3,self.t_yh_batch)
                loss = np.array(get_loss(self.t_yh_batch, self.a_3)).mean()

                train_acc_log_tmp.append(acc)
                train_loss_log_tmp.append(loss)

                self.update_weights(self.t_yh_batch,self.a_3)



            train_acc = np.array(train_acc_log_tmp).mean()
            train_loss = np.array(train_loss_log_tmp).mean()

            print("epoch : %s, loss : %s, tra_acc : %s"%(epoch,train_loss,train_acc))



            # load validation images of the batch size for every iteration
            for i, data in enumerate(valloader):

                # inputs is the image
                # labels is the class of the image
                inputs, labels = data
                inputs, labels = inputs.float() , labels.float()
                # if you don't change the image size, it will be [batch_size, 1, 100,


                # if labels is horse it returns tensor[0,0,0] else it returns tensor[

                self.v_data_batch = torch.squeeze(torch.FloatTensor(data[0]),3)

                self.v_data_batch = self.v_data_batch.view(len(data[0]),FEATURE_SIZE)

                self.z_1 = torch.matmul(self.v_data_batch,self.w_1.T) + self.b_1

                self.a_1 = get_activation(self.z_1,self.act_type_1)
                #print(self.a_1)

                self.z_2 = torch.matmul(self.a_1,self.w_2.T) + self.b_2
                self.a_2 = get_activation(self.z_2,self.act_type_2) # 1027 x 50

                self.z_3 = torch.matmul(self.a_2,self.w_3.T) + self.b_3
                self.a_3 = get_activation(self.z_3,self.act_type_3) # 1027 x 1

                self.v_yh_batch = data[1].float().unsqueeze(1)
                acc = self.get_acc(self.a_3,self.v_yh_batch)


                loss = np.array(get_loss(self.v_yh_batch, self.a_3)).mean()
                val_acc_log_tmp.append(acc)
                val_loss_log_tmp.append(loss)
            val_acc = np.array(val_acc_log_tmp).mean()
            val_loss = np.array(val_loss_log_tmp).mean()

            print("epoch : %s, loss : %s, val_acc : %s"%(epoch,val_loss,val_acc))

            self.train_acc_log.append(train_acc)
            self.train_loss_log.append(train_loss)
            self.val_acc_log.append(val_acc)
            self.val_loss_log.append(val_loss)
            self.epoch_log.append(epoch)
            epoch += 1

            tmp_idx = len(self.train_loss_log)-1
            if ( abs(self.train_loss_log[tmp_idx]-self.train_loss_log[tmp_idx-1]) < se
                print("Learning is terminated.")
                break


    def update_weights(self,t_y,a_3) :
        error_wb3 = -(torch.div(t_y,a_3+ 0.00011) - torch.div(1.0-t_y,1.0-a_3+ 0.00001
        d_z_3 = error_wb3*get_derv_activation(self.z_3,self.act_type_3) #
        d_w_3 = torch.matmul(d_z_3.T,self.a_2)
        d_b_3 = torch.sum(d_z_3, dim=0, keepdim=True) / self.a_2.shape[0] # mean

        #########

        error_wb2 = torch.matmul(d_z_3,self.w_3)

        d_z_2 = error_wb2*get_derv_activation(self.z_2,self.act_type_2)

        d_w_2 = torch.matmul(d_z_2.T,self.a_1)
        d_b_2 = torch.sum(d_z_2, dim=0,keepdims=True) / self.a_1.shape[0]


        error_wb1 = torch.matmul(d_z_2,self.w_2)
        d_z_1 = error_wb1*get_derv_activation(self.z_1,self.act_type_1)
        d_w_1 = torch.matmul(d_z_1.T,self.t_data_batch)
        d_b_1 = torch.sum(d_z_1, dim=0,keepdims=True) / self.t_data_batch.shape[0]


        self.w_3 += -self.lr*d_w_3
        self.b_3 += -self.lr*d_b_3

        self.w_2 += -self.lr*d_w_2
        self.b_2 += -self.lr*d_b_2
        self.w_1 += -self.lr*d_w_1
        self.b_1 += -self.lr*d_b_1
        #print(b_3,b_2,b_1)

    def get_acc(self,yhat,y) :
        count = 0

        for a,b in zip(yhat,y) :
            if a >= 0.5 :
                if b == 1 :
                    count+=1
            else :
                if b == 0:
                    count +=1

        return count / len(yhat)

    def show_loss(self) :
        #print(self.train_loss_log)
        #print(self.val_loss_log)
        #print(self.epoch_log)
```

```
In [41]: machine.show_loss()
```



```
In [42]: machine.show_acc()
```





	convergence	Best moment
Train_loss	0.018247241	0.087787054
Validation_loss	0.16419935	-0.4617174
Train_acc	0.9614478114478114	0.9663636363636364
Val_acc	0.8266666666666667	0.8495238095238097