vggnet

# example code

|  |
| --- |
| from google.colab import drive  drive.mount('vggnet')  # import package  # model  import torch  import torch.nn as nn  import torch.nn.functional as F  from torchsummary import summary  from torch import optim  from torch.optim.lr\_scheduler import StepLR  # dataset and transformation  from torchvision import datasets  import torchvision.transforms as transforms  from torch.utils.data import DataLoader  import os  # display images  from torchvision import utils  import matplotlib.pyplot as plt  %matplotlib inline  # utils  import numpy as np  from torchsummary import summary  import time  import copy  # specify a data path  path2data = '/content/vggnet/MyDrive/data'  # if not exists the path, make the directory  if not os.path.exists(path2data):      os.mkdir(path2data)  # load dataset  train\_ds = datasets.STL10(path2data, split='train', download=True, transform=transforms.ToTensor())  val\_ds = datasets.STL10(path2data, split='test', download=True, transform=transforms.ToTensor())  # check train\_ds  img, \_ = train\_ds[1]  print(img.shape)  print(len(train\_ds))  print(len(val\_ds))  # To normalize the dataset, calculate the mean and std  train\_meanRGB = [np.mean(x.numpy(), axis=(1,2)) for x, \_ in train\_ds]  train\_stdRGB = [np.std(x.numpy(), axis=(1,2)) for x, \_ in train\_ds]  train\_meanR = np.mean([m[0] for m in train\_meanRGB])  train\_meanG = np.mean([m[1] for m in train\_meanRGB])  train\_meanB = np.mean([m[2] for m in train\_meanRGB])  train\_stdR = np.mean([s[0] for s in train\_stdRGB])  train\_stdG = np.mean([s[1] for s in train\_stdRGB])  train\_stdB = np.mean([s[2] for s in train\_stdRGB])  val\_meanRGB = [np.mean(x.numpy(), axis=(1,2)) for x, \_ in val\_ds]  val\_stdRGB = [np.std(x.numpy(), axis=(1,2)) for x, \_ in val\_ds]  val\_meanR = np.mean([m[0] for m in val\_meanRGB])  val\_meanG = np.mean([m[1] for m in val\_meanRGB])  val\_meanB = np.mean([m[2] for m in val\_meanRGB])  val\_stdR = np.mean([s[0] for s in val\_stdRGB])  val\_stdG = np.mean([s[1] for s in val\_stdRGB])  val\_stdB = np.mean([s[2] for s in val\_stdRGB])  print(train\_meanR, train\_meanG, train\_meanB)  print(val\_meanR, val\_meanG, val\_meanB)  # define the image transformation  # using FiveCrop, normalize, horizontal reflection  train\_transformer = transforms.Compose([                      transforms.ToTensor(),                      transforms.Resize(224),                      transforms.Normalize([train\_meanR, train\_meanG, train\_meanB], [train\_stdR, train\_stdG, train\_stdB]),  ])  # test\_transformer = transforms.Compose([  #                     transforms.ToTensor(),  #                     transforms.Resize(224),  #                     transforms.Normalize([train\_meanR, train\_meanG, train\_meanB], [train\_stdR, train\_stdG, train\_stdB]),  # ])  # apply transformation  train\_ds.transform = train\_transformer  val\_ds.transform = train\_transformer  # display transformed sample images  def show(imgs, y=None, color=True):      # for i, img in enumerate(imgs):      #     npimg = img.numpy()      #     npimg\_tr = np.transpose(npimg, (1, 2, 0))      #     plt.subplot(1, imgs.shape[0], i+1)      #     plt.imshow(npimg\_tr)      npimg = imgs.numpy()      npimg\_tr = np.transpose(npimg, (1, 2, 0))      plt.imshow(npimg\_tr)        # plt.imshow(npimg\_tr)      if y is not None:          plt.title('labels: ' + str(y))  np.random.seed(0)  torch.manual\_seed(0)  # pick a random sample image  rnd\_inds = int(np.random.randint(0, len(train\_ds), 1))  img, label = train\_ds[rnd\_inds]  print('images indices: ', rnd\_inds)  plt.figure(figsize=(20, 20))  show(img)  # create dataloader  train\_dl = DataLoader(train\_ds, batch\_size=32, shuffle=True)  val\_dl = DataLoader(val\_ds, batch\_size=32, shuffle=True)  # VGG type dict  # int : output chnnels after conv layer  # 'M' : max pooling layer  VGG\_types = {      'VGG11' : [64, 'M', 128, 'M', 256, 256, 'M', 512,512, 'M',512,512,'M'],      'VGG13' : [64,64, 'M', 128, 128, 'M', 256, 256, 'M', 512,512, 'M', 512,512,'M'],      'VGG16' : [64,64, 'M', 128, 128, 'M', 256, 256,256, 'M', 512,512,512, 'M',512,512,512,'M'],      'VGG19' : [64,64, 'M', 128, 128, 'M', 256, 256,256,256, 'M', 512,512,512,512, 'M',512,512,512,512,'M']  }  # define VGGnet class  class VGGnet(nn.Module):      def \_\_init\_\_(self, model, in\_channels=3, num\_classes=10, init\_weights=True):          super(VGGnet,self).\_\_init\_\_()          self.in\_channels = in\_channels          # create conv\_layers corresponding to VGG type          self.conv\_layers = self.create\_conv\_laters(VGG\_types[model])          self.fcs = nn.Sequential(              nn.Linear(512 \* 7 \* 7, 4096),              nn.ReLU(),              nn.Dropout(),              nn.Linear(4096, 4096),              nn.ReLU(),              nn.Dropout(),              nn.Linear(4096, num\_classes),          )          # weight initialization          if init\_weights:              self.\_initialize\_weights()      def forward(self, x):          x = self.conv\_layers(x)          x = x.view(-1, 512 \* 7 \* 7)          x = self.fcs(x)          return x      # define weight initialization function      def \_initialize\_weights(self):          for m in self.modules():              if isinstance(m, nn.Conv2d):                  nn.init.kaiming\_normal\_(m.weight, mode='fan\_out', nonlinearity='relu')                  if m.bias is not None:                      nn.init.constant\_(m.bias, 0)              elif isinstance(m, nn.BatchNorm2d):                  nn.init.constant\_(m.weight, 1)                  nn.init.constant\_(m.bias, 0)              elif isinstance(m, nn.Linear):                  nn.init.normal\_(m.weight, 0, 0.01)                  nn.init.constant\_(m.bias, 0)        # define a function to create conv layer taken the key of VGG\_type dict      def create\_conv\_laters(self, architecture):          layers = []          in\_channels = self.in\_channels # 3          for x in architecture:              if type(x) == int: # int means conv layer                  out\_channels = x                  layers += [nn.Conv2d(in\_channels=in\_channels, out\_channels=out\_channels,                                       kernel\_size=(3,3), stride=(1,1), padding=(1,1)),                             nn.BatchNorm2d(x),                             nn.ReLU()]                  in\_channels = x              elif x == 'M':                  layers += [nn.MaxPool2d(kernel\_size=(2,2), stride=(2,2))]            return nn.Sequential(\*layers)  # define device  device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  print(device)  # creat VGGnet object  model = VGGnet('VGG16', in\_channels=3, num\_classes=10, init\_weights=True).to(device)  print(model)  # print model summary  summary(model, input\_size=(3, 224, 224), device=device.type)  loss\_func = nn.CrossEntropyLoss(reduction="sum")  opt = optim.Adam(model.parameters(), lr=0.001)  # get learning rate  def get\_lr(opt):      for param\_group in opt.param\_groups:          return param\_group['lr']  current\_lr = get\_lr(opt)  print('current lr={}'.format(current\_lr))  # define learning rate scheduler  # from torch.optim.lr\_scheduler import CosineAnnealingLR  # lr\_scheduler = CosineAnnealingLR(opt, T\_max=2, eta\_min=1e-5)  from torch.optim.lr\_scheduler import StepLR  lr\_scheduler = StepLR(opt, step\_size=30, gamma=0.1)  def metrics\_batch(output, target):      # get output class      pred = output.argmax(dim=1, keepdim=True)        # compare output class with target class      corrects=pred.eq(target.view\_as(pred)).sum().item()      return corrects  def loss\_batch(loss\_func, output, target, opt=None):        # get loss      loss = loss\_func(output, target)        # get performance metric      metric\_b = metrics\_batch(output,target)        if opt is not None:          opt.zero\_grad()          loss.backward()          opt.step()      return loss.item(), metric\_b  def loss\_epoch(model,loss\_func,dataset\_dl,sanity\_check=False,opt=None):      running\_loss=0.0      running\_metric=0.0      len\_data=len(dataset\_dl.dataset)      for xb, yb in dataset\_dl:          # move batch to device          xb=xb.to(device)          yb=yb.to(device)          output = model(xb)          # Five crop : bs, crops, chnnel, h, w          # making dimmension (bs, c, h, w)          # bs, ncrops, c, h, w = xb.size()          # output\_=model(xb.view(-1, c, h, w))          # output = output\_.view(bs, ncrops, -1).mean(1)            # get loss per batch          loss\_b,metric\_b=loss\_batch(loss\_func, output, yb, opt)            # update running loss          running\_loss+=loss\_b            # update running metric          if metric\_b is not None:              running\_metric+=metric\_b          # break the loop in case of sanity check          if sanity\_check is True:              break        # average loss value      loss=running\_loss/float(len\_data)        # average metric value      metric=running\_metric/float(len\_data)        return loss, metric  def train\_val(model, params):      # extract model parameters      num\_epochs=params["num\_epochs"]      loss\_func=params["loss\_func"]      opt=params["optimizer"]      train\_dl=params["train\_dl"]      val\_dl=params["val\_dl"]      sanity\_check=params["sanity\_check"]      lr\_scheduler=params["lr\_scheduler"]      path2weights=params["path2weights"]        # history of loss values in each epoch      loss\_history={          "train": [],          "val": [],      }        # histroy of metric values in each epoch      metric\_history={          "train": [],          "val": [],      }        # 가중치를 저장할 때, 코랩 GPU 오류나서 생략했습니다.      # a deep copy of weights for the best performing model      # best\_model\_wts = copy.deepcopy(model.state\_dict())        # initialize best loss to a large value      best\_loss=float('inf')        # check start time      start\_time = time.time()      # main loop      for epoch in range(num\_epochs):          # get current learning rate          current\_lr=get\_lr(opt)          print('Epoch {}/{}, current lr={}'.format(epoch, num\_epochs - 1, current\_lr))            # train model on training dataset          model.train()          train\_loss, train\_metric=loss\_epoch(model,loss\_func,train\_dl,sanity\_check,opt)          # collect loss and metric for training dataset          loss\_history["train"].append(train\_loss)          metric\_history["train"].append(train\_metric)            # evaluate model on validation dataset          model.eval()          with torch.no\_grad():              val\_loss, val\_metric=loss\_epoch(model,loss\_func,val\_dl,sanity\_check)              # store best model          if val\_loss < best\_loss:              best\_loss = val\_loss              best\_model\_wts = copy.deepcopy(model.state\_dict())                # # store weights into a local file              # torch.save(model.state\_dict(), path2weights)              # print("Copied best model weights!")            # collect loss and metric for validation dataset          loss\_history["val"].append(val\_loss)          metric\_history["val"].append(val\_metric)            # learning rate schedule          lr\_scheduler.step()          print("train loss: %.6f, dev loss: %.6f, accuracy: %.2f, time: %.4f s" %(train\_loss,val\_loss,100\*val\_metric, time.time()-start\_time))          print("-"\*10)      ## load best model weights      # model.load\_state\_dict(best\_model\_wts)        return model, loss\_history, metric\_history  # definc the training parameters  params\_train = {      'num\_epochs':10,      'optimizer':opt,      'loss\_func':loss\_func,      'train\_dl':train\_dl,      'val\_dl':val\_dl,      'sanity\_check':False,      'lr\_scheduler':lr\_scheduler,      'path2weights':'./models/weights.pt',  }  # create the directory that stores weights.pt  def createFolder(directory):      try:          if not os.path.exists(directory):              os.makedirs(directory)      except OSerror:          print('Error')  createFolder('./models')  # train model  model, loss\_hist, metric\_hist = train\_val(model, params\_train)  # Train-Validation Progress  num\_epochs=params\_train["num\_epochs"]  # plot loss progress  plt.title("Train-Val Loss")  plt.plot(range(1,num\_epochs+1),loss\_hist["train"],label="train")  plt.plot(range(1,num\_epochs+1),loss\_hist["val"],label="val")  plt.ylabel("Loss")  plt.xlabel("Training Epochs")  plt.legend()  plt.show()  # plot accuracy progress  plt.title("Train-Val Accuracy")  plt.plot(range(1,num\_epochs+1),metric\_hist["train"],label="train")  plt.plot(range(1,num\_epochs+1),metric\_hist["val"],label="val")  plt.ylabel("Accuracy")  plt.xlabel("Training Epochs")  plt.legend()  plt.show() |

# testing result

|  |
| --- |
| Epoch 0/9, current lr=0.001  train loss: 4.938084, dev loss: 2.294983, accuracy: 15.04, time: 155.2663 s  ----------  Epoch 1/9, current lr=0.001  train loss: 2.236087, dev loss: 2.047815, accuracy: 20.23, time: 310.0073 s  ----------  Epoch 2/9, current lr=0.001  train loss: 2.111822, dev loss: 1.977423, accuracy: 20.03, time: 465.4137 s  ----------  Epoch 3/9, current lr=0.001  train loss: 2.080703, dev loss: 2.067385, accuracy: 22.25, time: 620.9159 s  ----------  Epoch 4/9, current lr=0.001  train loss: 2.107729, dev loss: 1.936401, accuracy: 19.60, time: 777.3536 s  ----------  Epoch 5/9, current lr=0.001  train loss: 2.032415, dev loss: 1.893231, accuracy: 22.73, time: 933.1465 s  ----------  Epoch 6/9, current lr=0.001  train loss: 2.008678, dev loss: 1.955711, accuracy: 21.38, time: 1089.1959 s  ----------  Epoch 7/9, current lr=0.001  train loss: 1.923636, dev loss: 1.818512, accuracy: 22.86, time: 1246.1021 s  ----------  Epoch 8/9, current lr=0.001  train loss: 1.883151, dev loss: 1.816732, accuracy: 24.93, time: 1402.7985 s ---------- Epoch 9/9, current lr=0.001 train loss: 1.845764, dev loss: 1.760043, accuracy: 27.46, time: 1559.4186 s ---------- |