googlenet

# example code

|  |
| --- |
| from google.colab import drive  drive.mount('googlenet')  # 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 the data path  path2data = '/content/googlenet/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())  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  train\_transformation = transforms.Compose([                          transforms.ToTensor(),                          transforms.Resize(224),                          transforms.Normalize([train\_meanR, train\_meanG, train\_meanB],[train\_stdR, train\_stdG, train\_stdB]),                          transforms.RandomHorizontalFlip(),  ])  val\_transformation = transforms.Compose([                          transforms.ToTensor(),                          transforms.Resize(224),                          transforms.Normalize([train\_meanR, train\_meanG, train\_meanB],[train\_stdR, train\_stdG, train\_stdB]),  ])  # apply transforamtion  train\_ds.transform = train\_transformation  val\_ds.transform = val\_transformation  # create DataLoader  train\_dl = DataLoader(train\_ds, batch\_size=32, shuffle=True)  val\_dl = DataLoader(val\_ds, batch\_size=32, shuffle=True)  # display sample images  def show(img, y=None, color=True):      npimg = img.numpy()      npimg\_tr = np.transpose(npimg, (1, 2, 0))      plt.imshow(npimg\_tr)      if y is not None:          plt.title('labels: ' + str(y))  np.random.seed(0)  torch.manual\_seed(0)  grid\_size=4  rnd\_inds=np.random.randint(0,len(train\_ds),grid\_size)  print("image indices:",rnd\_inds)  x\_grid=[train\_ds[i][0] for i in rnd\_inds]  y\_grid=[train\_ds[i][1] for i in rnd\_inds]  x\_grid=utils.make\_grid(x\_grid, nrow=4, padding=2)  print(x\_grid.shape)  # call helper function  plt.figure(figsize=(10,10))  show(x\_grid,y\_grid)  class GoogLeNet(nn.Module):      def \_\_init\_\_(self,aux\_logits=True, num\_classes=10, init\_weights=True):          super(GoogLeNet, self).\_\_init\_\_()          assert aux\_logits == True or aux\_logits == False          self.aux\_logits = aux\_logits          # conv\_block takes in\_channels, out\_channels, kernel\_size, stride, padding          # Inception block takes out1x1, red\_3x3, out\_3x3, red\_5x5, out\_5x5, out\_1x1pool          self.conv1 = conv\_block(3, 64, kernel\_size=7, stride=2, padding=3)          self.maxpool1 = nn.MaxPool2d(3, 2, 1)          self.conv2 = conv\_block(64, 192, kernel\_size=3, stride=1, padding=1)          self.maxpool2 = nn.MaxPool2d(3, 2, 1)          self.inception3a = Inception\_block(192, 64, 96, 128, 16, 32, 32)          self.inception3b = Inception\_block(256, 128, 128, 192, 32, 96, 64)          self.maxpool3 = nn.MaxPool2d(3, 2, 1)          self.inception4a = Inception\_block(480, 192, 96, 208, 16, 48, 64)          # auxiliary classifier          self.inception4b = Inception\_block(512, 160, 112, 224, 24, 64, 64)          self.inception4c = Inception\_block(512, 128, 128, 256, 24, 64, 64)          self.inception4d = Inception\_block(512, 112, 144, 288, 32, 64, 64)          # auxiliary classifier          self.inception4e = Inception\_block(528, 256, 160, 320, 32, 128, 128)          self.maxpool4 = nn.MaxPool2d(3, 2, 1)          self.inception5a = Inception\_block(832, 256, 160, 320, 32, 128, 128)          self.inception5b = Inception\_block(832, 384, 192, 384, 48, 128, 128)          self.avgpool = nn.AvgPool2d(7, 1)          self.dropout = nn.Dropout(p=0.4)          self.fc1 = nn.Linear(1024, num\_classes)          if self.aux\_logits:              self.aux1 = InceptionAux(512, num\_classes)              self.aux2 = InceptionAux(528, num\_classes)          else:              self.aux1 = self.aux2 = None          # weight initialization          if init\_weights:              self.\_initialize\_weights()      def forward(self, x):          x = self.conv1(x)          x = self.maxpool1(x)          x = self.conv2(x)          x = self.maxpool2(x)          x = self.inception3a(x)          x = self.inception3b(x)          x = self.maxpool3(x)          x = self.inception4a(x)          if self.aux\_logits and self.training:              aux1 = self.aux1(x)          x = self.inception4b(x)          x = self.inception4c(x)          x = self.inception4d(x)          if self.aux\_logits and self.training:              aux2 = self.aux2(x)          x = self.inception4e(x)          x = self.maxpool4(x)          x = self.inception5a(x)          x = self.inception5b(x)          x = self.avgpool(x)          x = x.view(x.shape[0], -1)          x = self.dropout(x)          x = self.fc1(x)          if self.aux\_logits and self.training:              return x, aux1, aux2          else:              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)  class conv\_block(nn.Module):      def \_\_init\_\_(self, in\_channels, out\_channels, \*\*kwargs):          super(conv\_block, self).\_\_init\_\_()          self.conv\_layer = nn.Sequential(              nn.Conv2d(in\_channels, out\_channels, \*\*kwargs),              nn.BatchNorm2d(out\_channels),              nn.ReLU(),          )        def forward(self, x):          return self.conv\_layer(x)  class Inception\_block(nn.Module):      def \_\_init\_\_(self, in\_channels, out\_1x1, red\_3x3, out\_3x3, red\_5x5, out\_5x5, out\_1x1pool):          super(Inception\_block, self).\_\_init\_\_()          self.branch1 = conv\_block(in\_channels, out\_1x1, kernel\_size=1)          self.branch2 = nn.Sequential(              conv\_block(in\_channels, red\_3x3, kernel\_size=1),              conv\_block(red\_3x3, out\_3x3, kernel\_size=3, padding=1),          )          self.branch3 = nn.Sequential(              conv\_block(in\_channels, red\_5x5, kernel\_size=1),              conv\_block(red\_5x5, out\_5x5, kernel\_size=5, padding=2),          )          self.branch4 = nn.Sequential(              nn.MaxPool2d(kernel\_size=3, stride=1, padding=1),              conv\_block(in\_channels, out\_1x1pool, kernel\_size=1)          )      def forward(self, x):          # 0차원은 batch이므로 1차원인 filter 수를 기준으로 각 branch의 출력값을 묶어줍니다.          x = torch.cat([self.branch1(x), self.branch2(x), self.branch3(x), self.branch4(x)], 1)          return x  # auxiliary classifier의 loss는 0.3이 곱해지고, 최종 loss에 추가합니다. 정규화 효과가 있습니다.  class InceptionAux(nn.Module):      def \_\_init\_\_(self, in\_channels, num\_classes):          super(InceptionAux, self).\_\_init\_\_()          self.conv = nn.Sequential(              nn.AvgPool2d(kernel\_size=5, stride=3),              conv\_block(in\_channels, 128, kernel\_size=1),          )          self.fc = nn.Sequential(              nn.Linear(2048, 1024),              nn.ReLU(),              nn.Dropout(),              nn.Linear(1024, num\_classes),          )      def forward(self,x):          x = self.conv(x)          x = x.view(x.shape[0], -1)          x = self.fc(x)          return x  device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  print(device)  model = GoogLeNet(aux\_logits=True, num\_classes=10, init\_weights=True).to(device)  print(model)  x = torch.randn(3, 3, 224, 224).to(device)  output = model(x)  print(output)  summary(model, input\_size=(3,224,224), device=device.type)  loss\_func = nn.CrossEntropyLoss(reduction='sum')  opt = optim.Adam(model.parameters(), lr=0.001)  from torch.optim.lr\_scheduler import StepLR  lr\_scheduler = StepLR(opt, step\_size=30, gamma=0.1)  def get\_lr(opt):      for param\_group in opt.param\_groups:          return param\_group['lr']    def metric\_batch(output, target):      pred = output.argmax(dim=1, keepdim=True)      corrects = pred.eq(target.view\_as(pred)).sum().item()      return corrects    def loss\_batch(loss\_func, outputs, target, opt=None):      if np.shape(outputs)[0] == 3:          output, aux1, aux2 = outputs          output\_loss = loss\_func(output, target)          aux1\_loss = loss\_func(aux1, target)          aux2\_loss = loss\_func(aux2, target)          loss = output\_loss + 0.3\*(aux1\_loss + aux2\_loss)          metric\_b = metric\_batch(output,target)      else:          loss = loss\_func(outputs, target)          metric\_b = metric\_batch(outputs, 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:          xb = xb.to(device)          yb = yb.to(device)          output= model(xb)          loss\_b, metric\_b = loss\_batch(loss\_func, output, yb, opt)          running\_loss += loss\_b          if metric\_b is not None:              running\_metric += metric\_b            if sanity\_check is True:              break      loss = running\_loss / len\_data      metric = running\_metric / len\_data      return loss, metric    def train\_val(model, params):      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"]      loss\_history = {'train': [], 'val': []}      metric\_history = {'train': [], 'val': []}      best\_model\_wts = copy.deepcopy(model.state\_dict())      best\_loss = float('inf')        start\_time = time.time()      for epoch in range(num\_epochs):          current\_lr = get\_lr(opt)          print('Epoch {}/{}, current lr={}'.format(epoch, num\_epochs - 1, current\_lr))            model.train()          train\_loss, train\_metric = loss\_epoch(model, loss\_func, train\_dl, sanity\_check, opt)          loss\_history['train'].append(train\_loss)          metric\_history['train'].append(train\_metric)          model.eval()          with torch.no\_grad():              val\_loss, val\_metric = loss\_epoch(model, loss\_func, val\_dl, sanity\_check)          if val\_loss < best\_loss:              best\_loss = val\_loss              best\_model\_wts = copy.deepcopy(model.state\_dict())              torch.save(model.state\_dict(), path2weights)              print('Copied best model weights!')          loss\_history['val'].append(val\_loss)          metric\_history['val'].append(val\_metric)          lr\_scheduler.step()          print('train loss: %.6f, val loss: %.6f, accuracy: %.2f, time: %.4f min' %(train\_loss, val\_loss, 100\*val\_metric, (time.time()-start\_time)/60))          print('-'\*10)      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')  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  Copied best model weights!  train loss: 2.747296, val loss: 1.658408, accuracy: 35.24, time: 1.0218 min  ----------  Epoch 1/9, current lr=0.001  Copied best model weights!  train loss: 2.311925, val loss: 1.643239, accuracy: 38.51, time: 1.9798 min  ----------  Epoch 2/9, current lr=0.001  Copied best model weights!  train loss: 2.013739, val loss: 1.510814, accuracy: 43.64, time: 2.9302 min  ----------  Epoch 3/9, current lr=0.001  Copied best model weights!  train loss: 1.818441, val loss: 1.254463, accuracy: 55.20, time: 3.9611 min  ----------  Epoch 4/9, current lr=0.001  train loss: 1.641959, val loss: 1.308688, accuracy: 54.29, time: 4.9076 min  ----------  Epoch 5/9, current lr=0.001  Copied best model weights!  train loss: 1.485846, val loss: 1.024666, accuracy: 62.64, time: 5.8595 min  ----------  Epoch 6/9, current lr=0.001  Copied best model weights!  train loss: 1.178133, val loss: 1.002145, accuracy: 65.66, time: 8.8277 min ---------- Epoch 9/9, current lr=0.001 train loss: 1.031729, val loss: 1.117213, accuracy: 63.80, time: 9.8314 min ----------    Z |