Artificial Intelligence Final Report Assignment 問題1 (Problem 1)

レポート解答用紙 (Report Answer Sheet)

Group Leader

学生証番号 (Student ID): 20521180

名前(Name): Trương Quốc Đạt

Group Members

学生証番号 (Student ID): 20521729

名前(Name): Nguyễn Triệu Kim Oanh

学生証番号 (Student ID): 20521714

名前(Name): Nguyễn Phan Uyển Nhi

学生証番号 (Student ID): 20521505

名前(Name): Trần Ngô Tuấn Kiệt

問題1 (Problem 1)のレポート

### Program

#import libraries

import torch

import torch.nn as nn

import torch.nn.functional as F

import torchvision

import torchvision.transforms as transforms

from torch.utils.data import DataLoader

import seaborn as sns

import numpy as np

from torch.utils.data import random\_split

device=torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

BATCH\_SIZE=64

num\_epochs=5

lr=1e-4

class\_size=10

tranform\_train = transforms.Compose([transforms.Resize((224,224)), transforms.RandomHorizontalFlip(p=0.7), transforms.ToTensor(), transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

tranform\_test = transforms.Compose([transforms.Resize((224,224)), transforms.ToTensor(), transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

#prep the train, validation and test dataset

torch.manual\_seed(2021)

train = torchvision.datasets.CIFAR10("data/", train=True, download=True, transform=tranform\_train)

val\_size = 10000

train\_size = len(train) - val\_size

train, val = random\_split(train, [train\_size, val\_size])

test = torchvision.datasets.CIFAR10("data/", train=False, download=True, transform=tranform\_test)

# train, val and test datasets to the dataloader

train\_loader = DataLoader(train, batch\_size=BATCH\_SIZE, shuffle=True)

val\_loader = DataLoader(val, batch\_size=BATCH\_SIZE, shuffle=False)

class VGG16\_NET(nn.Module):

def \_\_init\_\_(self):

super(VGG16\_NET, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(in\_channels=3, out\_channels=64, kernel\_size=3, padding=1)

self.conv2 = nn.Conv2d(in\_channels=64, out\_channels=64, kernel\_size=3, padding=1)

self.conv3 = nn.Conv2d(in\_channels=64, out\_channels=128, kernel\_size=3, padding=1)

self.conv4 = nn.Conv2d(in\_channels=128, out\_channels=128, kernel\_size=3, padding=1)

self.conv5 = nn.Conv2d(in\_channels=128, out\_channels=256, kernel\_size=3, padding=1)

self.conv6 = nn.Conv2d(in\_channels=256, out\_channels=256, kernel\_size=3, padding=1)

self.conv7 = nn.Conv2d(in\_channels=256, out\_channels=256, kernel\_size=3, padding=1)

self.conv8 = nn.Conv2d(in\_channels=256, out\_channels=512, kernel\_size=3, padding=1)

self.conv9 = nn.Conv2d(in\_channels=512, out\_channels=512, kernel\_size=3, padding=1)

self.conv10 = nn.Conv2d(in\_channels=512, out\_channels=512, kernel\_size=3, padding=1)

self.conv11 = nn.Conv2d(in\_channels=512, out\_channels=512, kernel\_size=3, padding=1)

self.conv12 = nn.Conv2d(in\_channels=512, out\_channels=512, kernel\_size=3, padding=1)

self.conv13 = nn.Conv2d(in\_channels=512, out\_channels=512, kernel\_size=3, padding=1)

self.maxpool = nn.MaxPool2d(kernel\_size=2, stride=2)

self.fc14 = nn.Linear(25088, 4096)

self.fc15 = nn.Linear(4096, 4096)

self.fc16 = nn.Linear(4096, 10)

def forward(self, x):

x = F.relu(self.conv1(x))

x = F.relu(self.conv2(x))

x = self.maxpool(x)

x = F.relu(self.conv3(x))

x = F.relu(self.conv4(x))

x = self.maxpool(x)

x = F.relu(self.conv5(x))

x = F.relu(self.conv6(x))

x = F.relu(self.conv7(x))

x = self.maxpool(x)

x = F.relu(self.conv8(x))

x = F.relu(self.conv9(x))

x = F.relu(self.conv10(x))

x = self.maxpool(x)

x = F.relu(self.conv11(x))

x = F.relu(self.conv12(x))

x = F.relu(self.conv13(x))

x = self.maxpool(x)

x = x.reshape(x.shape[0], -1)

x = F.relu(self.fc14(x))

x = F.dropout(x, 0.5) #dropout was included to combat overfitting

x = F.relu(self.fc15(x))

x = F.dropout(x, 0.5)

x = self.fc16(x)

return x

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = VGG16\_NET()

model = model.to(device=device)

load\_model = True

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr= lr)

for epoch in range(num\_epochs): #I decided to train the model for 50 epochs

loss\_var = 0

for idx, (images, labels) in enumerate(train\_loader):

images = images.to(device=device)

labels = labels.to(device=device)

## Forward Pass

optimizer.zero\_grad()

scores = model(images)

loss = criterion(scores,labels)

loss.backward()

optimizer.step()

loss\_var += loss.item()

if idx%64==0:

print(f'Epoch [{epoch+1}/{num\_epochs}] || Step [{idx+1}/{len(train\_loader)}] || Loss:{loss\_var/len(train\_loader)}')

print(f"Loss at epoch {epoch+1} || {loss\_var/len(train\_loader)}")

with torch.no\_grad():

correct = 0

samples = 0

for idx, (images, labels) in enumerate(val\_loader):

images = images.to(device=device)

labels = labels.to(device=device)

outputs = model(images)

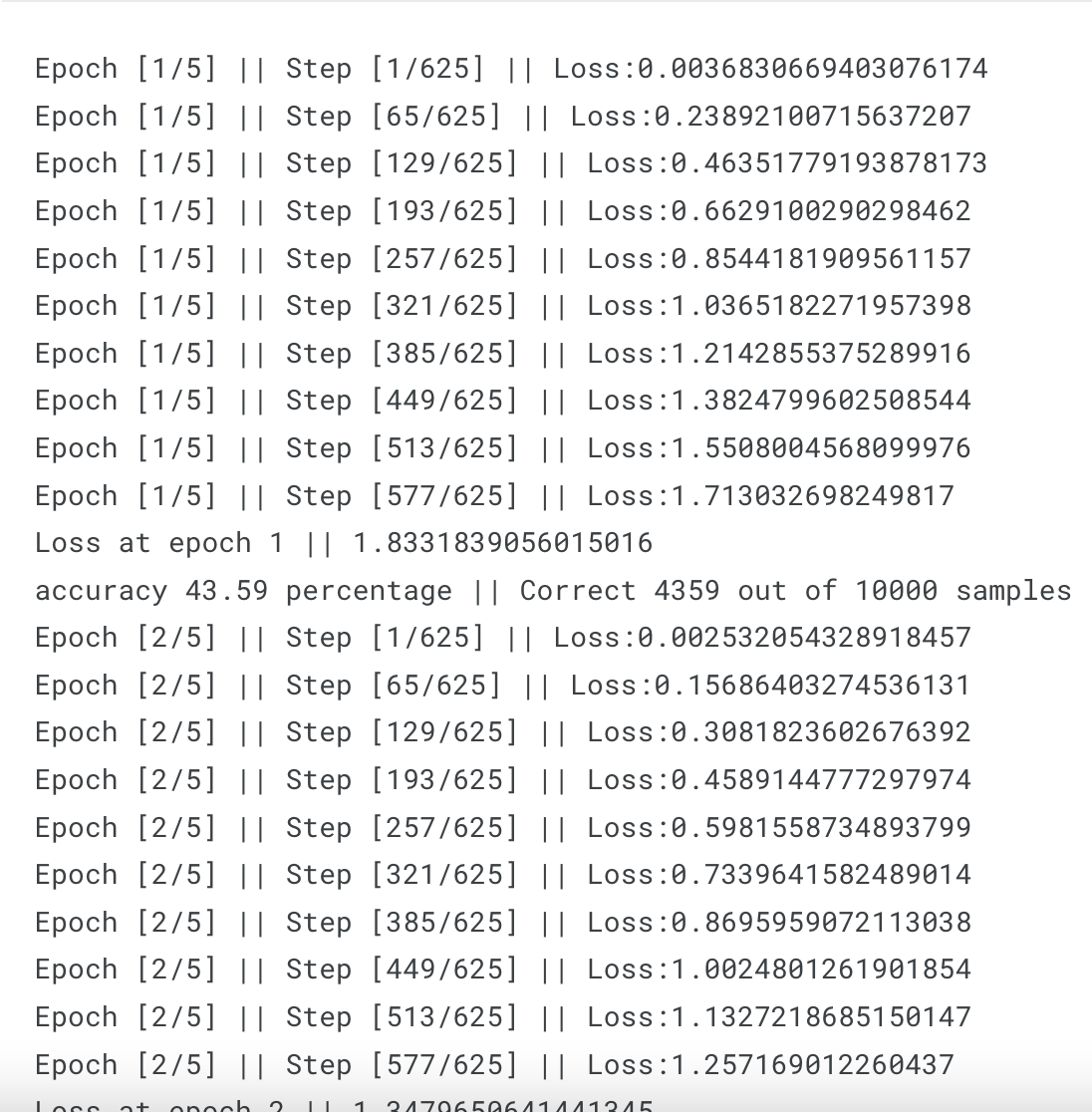
\_, preds = outputs.max(1)

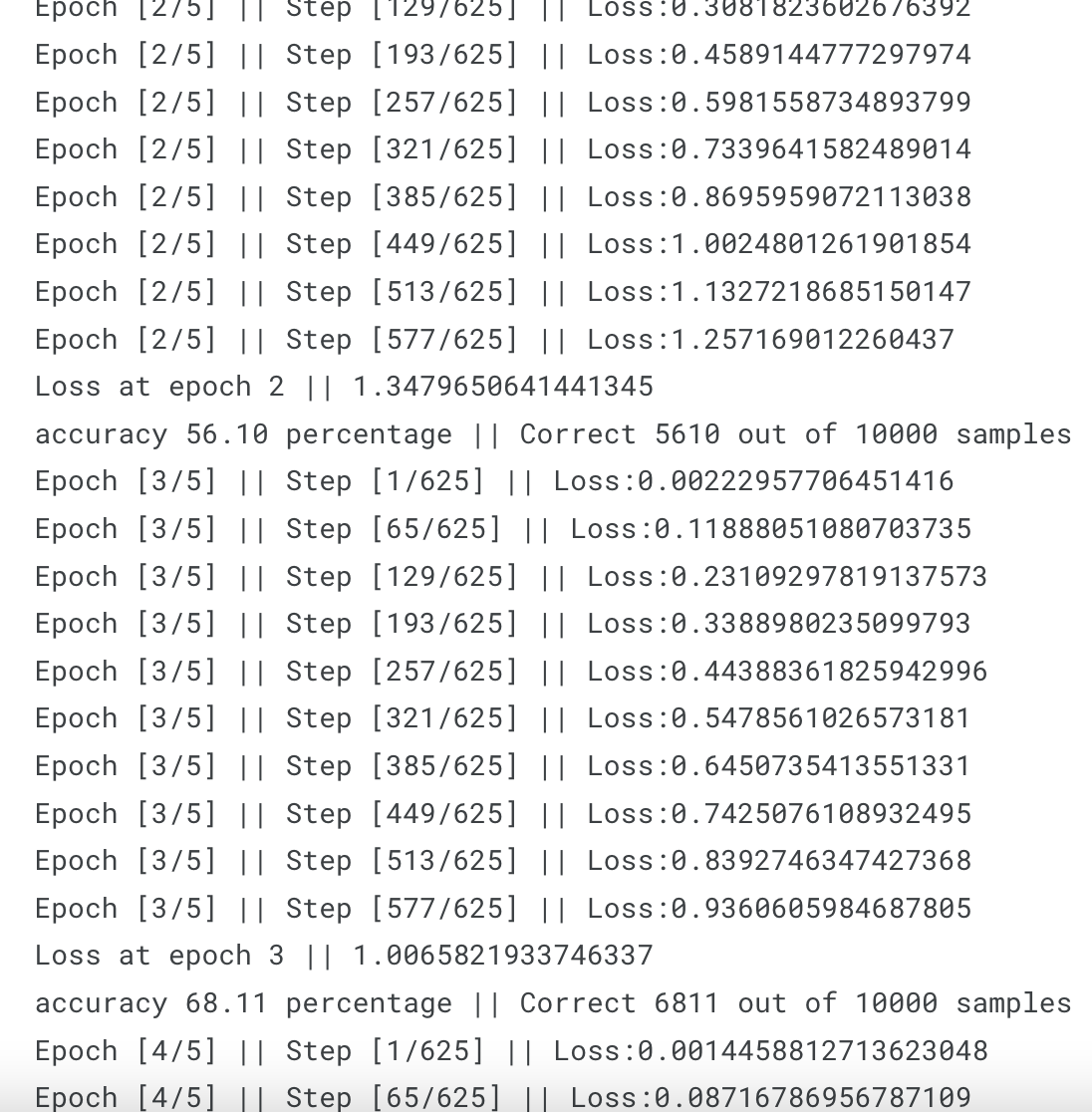
correct += (preds == labels).sum()

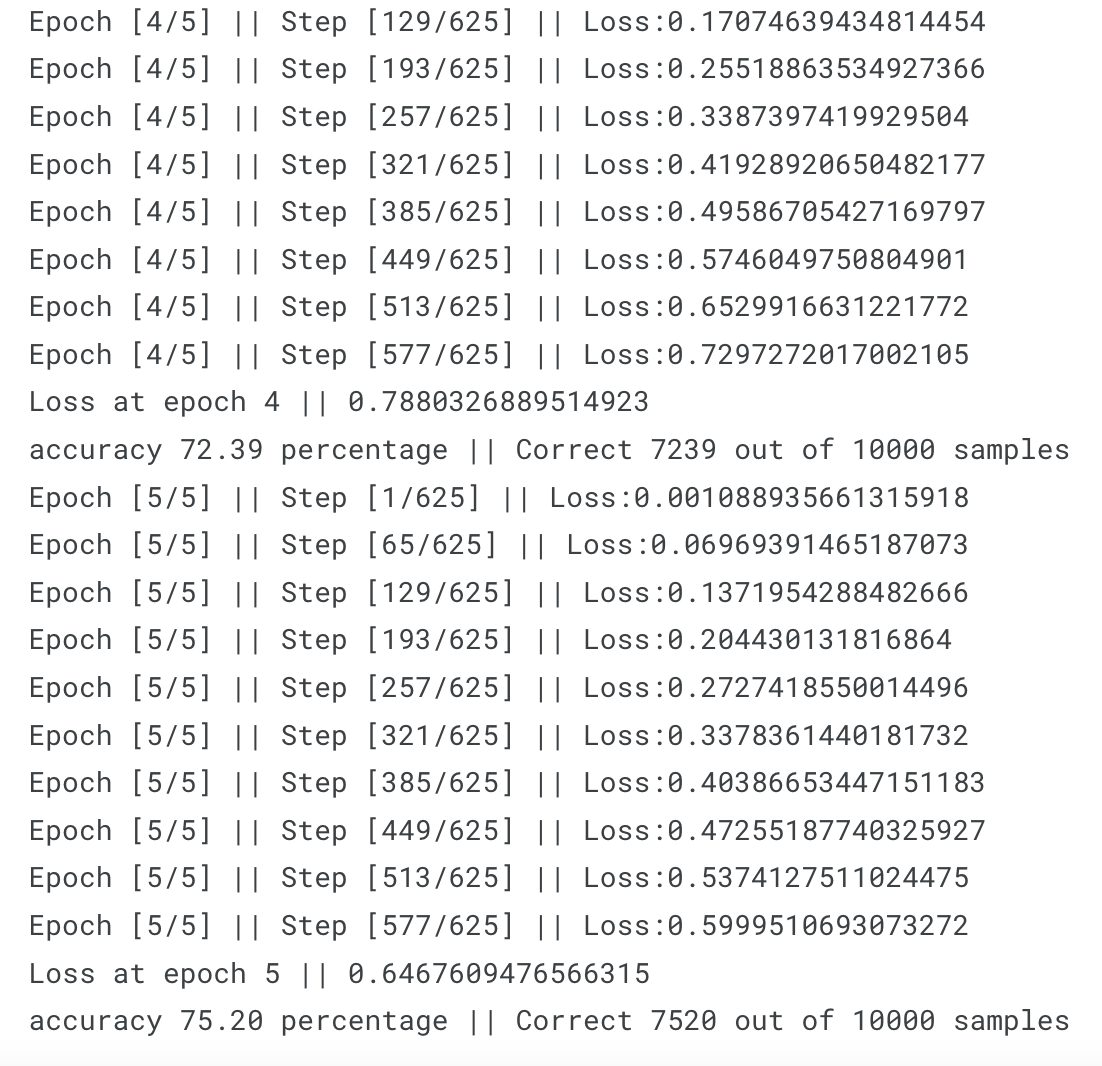
samples += preds.size(0)

print(f"accuracy {float(correct) / float(samples) \* 100:.2f} percentage || Correct {correct} out of {samples} samples")

### Execution Results







### Explanation

In the improved program, several changes and improvements were made to increase accuracy. Here are the key changes:

1. Data Preprocessing: The image size was resized to (224, 224) for both training and testing data using `transforms.Resize`. Additionally, data augmentation techniques like random horizontal flip were applied to the training dataset using `transforms.RandomHorizontalFlip` to increase the diversity of training samples.

2. Data Normalization: The image data was normalized using mean and standard deviation values of [0.485, 0.456, 0.406] and [0.229, 0.224, 0.225], respectively. This normalization helps in stabilizing the training process.

3. Data Loading and Splitting: The `random\_split` function from PyTorch was used to split the original training dataset into a new training dataset and a validation dataset. This allows for monitoring the model's performance on unseen data during training.

4. Model Architecture: The model architecture was changed to a VGG16-based network. The VGG16\_NET class was defined, which consists of several convolutional layers followed by fully connected layers. This architecture has shown good performance in image classification tasks.

5. Dropout Regularization: Dropout layers were included after the fully connected layers to combat overfitting. Dropout randomly sets a fraction of input units to zero during training, which helps in reducing over-reliance on specific features.

6. Training Loop: The training loop was modified to iterate for a fixed number of epochs (5 in this case) instead of a single epoch. The loss was calculated and printed after each epoch to monitor the training progress.

7. Evaluation: After each epoch, the model's accuracy was evaluated on the validation dataset. The number of correct predictions and the total number of samples were counted to calculate the accuracy percentage.

8. Device Handling: The program includes device handling to check if a CUDA-enabled GPU is available and sets the device accordingly. This allows for training and inference on a GPU if available, enhancing performance.

9. Optimizer and Loss Function: The Adam optimizer was used with a learning rate of 1e-4 to optimize the model parameters. The cross-entropy loss function (`nn.CrossEntropyLoss()`) was utilized, which is commonly used for multi-class classification tasks.

By incorporating these changes, the improved program provides better accuracy and performance compared to the previous version.