This a notebook for the Astar training 2025. You are guided to train a simple classifier on the cifar-10 dataset.

Please try to read through this notebook, and fill in the blank according to the instructions given after `TODO` *.*

You may download the notebook or excute directly on kaggle.

submit .pdf converted from this notebook after completion

-- Fan Zhenyi

This notebook requires the following packages

excute the following command to install the required packages

```
!pip install torch torchvision
!pip install matplotlib
!pip install numpy
!pip install tqdm
```

```
Requirement already satisfied: torch in
/opt/conda/lib/python3.10/site-packages (2.4.0+cpu)
Requirement already satisfied: torchvision in
/opt/conda/lib/python3.10/site-packages (0.19.0+cpu)
Requirement already satisfied: filelock in
/opt/conda/lib/python3.10/site-packages (from torch) (3.15.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/opt/conda/lib/python3.10/site-packages (from torch) (4.12.2)
Requirement already satisfied: sympy in
/opt/conda/lib/python3.10/site-packages (from torch) (1.12)
Requirement already satisfied: networkx in
/opt/conda/lib/python3.10/site-packages (from torch) (3.3)
Requirement already satisfied: jinja2 in
/opt/conda/lib/python3.10/site-packages (from torch) (3.1.4)
Requirement already satisfied: fsspec in
/opt/conda/lib/python3.10/site-packages (from torch) (2024.6.1)
Requirement already satisfied: numpy in
/opt/conda/lib/python3.10/site-packages (from torchvision) (1.26.4)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/opt/conda/lib/python3.10/site-packages (from torchvision) (9.5.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/opt/conda/lib/python3.10/site-packages (from jinja2->torch) (2.1.5)
Requirement already satisfied: mpmath>=0.19 in
/opt/conda/lib/python3.10/site-packages (from sympy->torch) (1.3.0)
Requirement already satisfied: matplotlib in
/opt/conda/lib/python3.10/site-packages (3.7.5)
Requirement already satisfied: contourpy>=1.0.1 in
/opt/conda/lib/python3.10/site-packages (from matplotlib) (1.2.1)
Requirement already satisfied: cycler>=0.10 in
/opt/conda/lib/python3.10/site-packages (from matplotlib) (0.12.1)
```

```
Requirement already satisfied: fonttools>=4.22.0 in
/opt/conda/lib/python3.10/site-packages (from matplotlib) (4.53.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/opt/conda/lib/python3.10/site-packages (from matplotlib) (1.4.5)
Requirement already satisfied: numpy<2,>=1.20 in
/opt/conda/lib/python3.10/site-packages (from matplotlib) (1.26.4)
Requirement already satisfied: packaging>=20.0 in
/opt/conda/lib/python3.10/site-packages (from matplotlib) (21.3)
Requirement already satisfied: pillow>=6.2.0 in
/opt/conda/lib/python3.10/site-packages (from matplotlib) (9.5.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/opt/conda/lib/python3.10/site-packages (from matplotlib) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in
/opt/conda/lib/python3.10/site-packages (from matplotlib)
(2.9.0.post0)
Requirement already satisfied: six>=1.5 in
/opt/conda/lib/python3.10/site-packages (from python-dateutil>=2.7-
>matplotlib) (1.16.0)
Requirement already satisfied: numpy in
/opt/conda/lib/python3.10/site-packages (1.26.4)
Requirement already satisfied: tqdm in /opt/conda/lib/python3.10/site-
packages (4.66.4)
```

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import datasets, transforms


import matplotlib.pyplot as plt

import numpy as np
import pickle

import os

from tqdm import tqdm
```

The dataloader is already implemented for you. You can use it to load the cifar-10 dataset.

```python
class CIFAR10(Dataset):
    #__init__ is a special method in python classes, it is called when
an object is created
    def __init__(self, root, train=True, transform=None):
        #root path to the dataset, you may ignore this
        self.root = root

        #this is a boolean value to indicate whether we are loading
the training or test set
```

```python
        self.train = train

        #this is the transformation that will be applied to the images

        #it's none by default, you are required to pass a transform
later
        self.transform = transform

        #checking if the dataset exists, if not download it
        if not self._check_exists():
            print("Downloading cifar10 dataset")
            self.download()

        #load the data
        #We are going to load the data in memory
        #Store the images and labels in the self.data and self.targets
variables
        if self.train:
            self.data, self.targets = self._load_training_data()
        else:
            self.data, self.targets = self._load_test_data()

    #this method returns the length of the dataset
    def __len__(self):
        return len(self.data)

    #this method returns a sample from the dataset at the given index
    #this is very important because it allows us to iterate over the
dataset
    def __getitem__(self, index):
        img, target = self.data[index], int(self.targets[index])

        img = torch.from_numpy(img).float().permute(2,0,1) / 255.0

        if self.transform:
            img = self.transform(img)

        return img, target


    def _check_exists(self):
        return os.path.exists(os.path.join(self.root, "cifar-10-
batches-py"))

    def download(self):
        if self._check_exists():
            print("Dataset already exists !!!")
            return
        return datasets.CIFAR10(self.root, train=self.train,
download=True)
```

```
    #this function looks complicated but it's just reading the data
from the files
    def _load_batch(self, file_path):
        with open(file_path, 'rb') as f:
            data = pickle.load(f, encoding='bytes')
        return data[b'data'], data[b'labels']

    def _load_training_data(self):
        data = []
        targets = []
        for i in range(1, 6):
            file_path = os.path.join(self.root, "cifar-10-batches-py",
f"data_batch_{i}")
            batch_data, batch_labels = self._load_batch(file_path)
            data.append(batch_data)
            targets.extend(batch_labels)

        data = np.vstack(data).reshape(-1, 3, 32, 32).transpose(0, 2,
3, 1)
        return data, np.array(targets)

    def _load_test_data(self):
        file_path = os.path.join(self.root, "cifar-10-batches-py",
"test_batch")
        data, labels = self._load_batch(file_path)
        return data.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1),
np.array(labels)


#We have defined our datasetclass, now we are going to instantiate it
#The instantiation will download the dataset and load it in memory, it
takes about 1-2 mins
train_dataset = CIFAR10(root="data", train=True)
test_dataset = CIFAR10(root="data", train=False)

#simple demonstration __getitem__ method
img, target = train_dataset[0] #get the first image in the dataset
plt.figure(figsize=(1,1))
plt.imshow(img.permute(1,2,0))
#TODO: what does permute do?, and why do we need it here?
#answer: rearranges the dimensions of the original img tensor into
(channels = 1, height = 2, width = 0)
#
#end of you answer
plt.title("Original Image")
print(img.shape, "label: ",target)
#the image is a torch tensor (3, 28, 28) and the target is the label
of the image
```

```
#TODO: define reasonable transformations for the images, you can use
the transforms module from torchvision
transform = transforms.Compose([
    transforms.ToPILImage(),

    #TODO: Implement the transformations here
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2,
saturation=0.2, hue=0.2),

    transforms.RandomRotation(15),

    #end of your implementation;

    transforms.ToTensor()
])

train_dataset.transform = transform

img, target = train_dataset[0] #get the first image in the dataset
plt.figure(figsize=(1,1))
plt.imshow(img.permute(1,2,0))
plt.title("Transformed Image")
print(img.shape, "label: ",target)

#apart from test set, we are going to use the training set to create a
validation set
#we are going to split the training set into two parts
train_size = int(0.9 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_dataset, val_dataset =
torch.utils.data.random_split(train_dataset, [train_size, val_size])

#we are going to use the DataLoader class to create an iterator for
our dataset
#this iterator will be used to iterate over the dataset in batches
#tentatively we are going to use a batch size of 32
#TODO: change different batch sizes and see how it affects the
training process
train_loader = DataLoader(train_dataset, batch_size=256, shuffle=True,
num_workers= 4, pin_memory=True)
val_loader = DataLoader(val_dataset, batch_size=256,
shuffle=False,num_workers= 4, pin_memory = True)
test_loader = DataLoader(test_dataset, batch_size=256,
shuffle=False,num_workers= 4, pin_memory = True)
```

This defines the model architecture. You can use the model as it is or modify it as per your requirements.

```
class LinearModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.name = "Linear"
        self.num_inputs = 3*32*32
        hidden_size = 512
        num_classes = 10
        super().__init__()
        self.linear = nn.Sequential(
            nn.Linear(self.num_inputs, hidden_size),    # batch_size x
784 -> batch_size x 512
            nn.ReLU(), #activation function             # batch_size x
512 -> batch_size x 512
            nn.Linear(hidden_size, num_classes)         # batch_size x
512 -> batch_size x 10
        ) #nn.Sequential is a container for other layers, it applies
the layers in sequence

    #forward is the method that defines the forward pass of the
network
    #not rigurously: model.forward(x) = model(x)
    def forward(self, x):
        x = x.view(-1, self.num_inputs) # flatten the image from
3x32x32 to 3072
        x = self.linear(x)
        return x

class CNNModel(nn.Module):
  def __init__(self):
    super().__init__()
    self.name = "CNN"
    self.conv = nn.Sequential(
      # Layer 1: Convolution
      nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3,
stride=1, padding=1),  # Input: 3x32x32, Output: 32x32x32
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),  # Input: 32x32x32,
Output: 16x16x32
      nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
stride=1, padding=1),  # Input: 16x16x32, Output: 64x16x16
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),  # Input: 64x16x16,
Output: 8x8x64
      nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,
stride=1, padding=1),  # Input: 8x8x64, Output: 128x8x8
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),  # Input: 128x8x8,
Output: 4x4x128
    )
    def forward(self, x):
        x = self.conv(x)
        x = x.view(-1, 128*4*4)
        x = self.fc(x)
        return x

    def getFeature(self, x):
        x = self.conv(x)
        feat = x.view(-1, 128*4*4) #this is the 128*4*4 feature
        return feat


class ModelX(nn.Module):
  def __init__(self):
    super().__init__()
    self.name = "Your_model"

    self.conv = nn.Sequential(
        nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3,
stride=1, padding=1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),

        nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,
stride=1, padding=1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2)

    )

    self.fc = nn.Sequential(
        nn.Linear(32 * 7 * 7, 64),
        nn.ReLU(),
        nn.Linear(64, 10)
    )

  def forward(self, x):
    x = self.conv(x)
    x = x.view(-1, 32 * 7 * 7)
    x = self.fc(x)

    return x
```

This is the training loop. You can modify the training loop as per your requirements.

The training takes some time. You can do other tasks while the training is in progress.

you may use the gpu on kaggle or colab to speed up the training process.
https://www.kaggle.com/discussions/general/97939

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("using device:", device)
#instantiating the model
#TODO: change the model to the CNNModel and Your model, and evaluate
the performance.
model = CNNModel()
# model = ModelX()
model = model.to(device)
print(f"training {model.name} model")

#defining the loss function
criterion = nn.CrossEntropyLoss()

#defining the optimizer
#TODO: try to modify the optimizer and see how it affects the training
process
optimizer = optim.Adam(model.parameters(), lr=0.01)

best_accuracy = 0
#early stopping
early_stopping = 5
early_stopping_counter = 0


#TODO: adjust the number of epochs and see how it affects the training
process
epochs = 40
patience = 5
best_accuracy = 0.0
early_stopping_count = 0
for epoch in range(epochs):
    #training
    for images, labels in tqdm(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        #forward pass
        outputs = model(images)
        #calculate the loss
        loss = criterion(outputs, labels)
        #zero the gradients
        optimizer.zero_grad() #ensure that the gradients are zero
        #backward pass
        loss.backward()
        #optimize
        optimizer.step()
    #validation
    total = 0
    correct = 0
    for images, labels in val_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    accuracy = correct / total

    #TODO: implement early stopping
    #what is early stopping?
https://en.wikipedia.org/wiki/Early_stopping

    #end of early stopping

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        early_stopping_count = 0  # Reset counter on improvement
        print(f"Epoch: {epoch}, Loss: {loss.item()}, Accuracy:
{accuracy} ***")
        # Save the best model (optional, uncomment if desired)
        # torch.save(model.state_dict(),
f"best_model_{model.name}.pth")
    else:
        early_stopping_count += 1
        print(f"Epoch: {epoch}, Loss: {loss.item()}, Accuracy:
{accuracy}")

    # Check for early stopping condition
    if early_stopping_count >= patience:
        print(f"Early stopping triggered after {patience} epochs with
no improvement.")
        break

# Save the best model (alternative approach)
if best_accuracy > 0.0:
    torch.save(model.state_dict(), f"best_model_{model.name}.pth")




#load the best model
model.load_state_dict(torch.load(f"best_model_{model.name}.pth",
weights_only=False))

#testing
total = 0
correct = 0
for images, labels in test_loader:
    images = images.to(device)
    labels = labels.to(device)
    outputs = model(images)
    _, predicted = torch.max(outputs, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()
accuracy = correct / total
print(f"Test Accuracy: {accuracy}")
```