Deep learining

Classification

Exmaple

```
import torch
from torch import nn
from torch import optim
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
from tqdm import tqdm  # visual bar
import matplotlib.pyplot as plt

class Mymodel(nn.Module):
    def __init__(self, in_channels, classnumber):
        super(Mymodel, self).__init__()
        self.mymodel = nn.Sequential(
            nn.Conv2d(
                in_channels=in_channels,
                out_channels=32,
                kernel_size=3,
                # stride=1,
                padding=1,
            ),
            nn.BatchNorm2d(32),  # 正则化
            nn.ReLU(),
            nn.MaxPool2d(2,2),  # 28/2=14
            nn.Conv2d(
                in_channels=32,
                out_channels=32,
                kernel_size=3,
                # stride=1,
                # padding=0,
            ),
            # 14-2=12
            nn.BatchNorm2d(32),  # 正则化
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 12/2=6
            nn.Conv2d(
                in_channels=32,
                out_channels=32,
                kernel_size=3,
                # stride=1,
                # padding=0
            ),
            # 6-2=4
            nn.BatchNorm2d(32),
            nn.ReLU(),
```

```
            nn.MaxPool2d(2, 2),  # 4/2=2
        )

        self.filter = nn.Sequential(
            nn.Linear(32*2*2, 64),
            nn.ReLU(),
            nn.Linear(64, classnumber)
        )
    def forward(self,inputs):
        inputs = self.mymodel(inputs)
        inputs = inputs.view(-1, 32*2*2)
        inputs = self.classifier(inputs)
        return inputs


if __name__ == '__main__':
    train_dataset = MNIST(root="D:\File\CODE\python\dataset")
    train_dataloader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=
True)

    test_dataset = MNIST(root= "D:\File\CODE\python\dataset", train=False,
transform=ToTensor())
    test_dataloader = DataLoader(dataset=test_dataset, batch_size=10)

    device = "cuda"
    model = Mymodel(in_channels=1, classnumber=10).to(device)

    epochs = 100
    optimizer = optim.Adam(model.parameters()) # lr=1e-3 by default
    loss_fn = nn.CrossEntropyLoss()  # 交叉熵

    for i in range(epochs):
        model.train()
        train_data = tqdm(train_dataloader)

        mean_loss, acc = 0.0, 0.0  # 损失、精确度
        data_num = 0

        for x,y in train_data:  # x is input, y is label or target
            x = x.to(device)  # Move input and target to GPU if available
            y = y.to(device)

            predicts = model(x)  # 先预测
            # predicts: This is the output of the model for the current batch of
inputs x.
            # [batch_size, n_classes], where n_classes is the number of classes in
the classification problem.
            loss = loss_fn(predicts, y)  # 再算损失
            optimizer.zero_grad() # 优化器梯度清零
            loss.backward()  # 反向传播
            optimizer.step()  # 更新模型参数

            acc += torch.sum(torch.argmax(predicts,dim=1) == y).item()
            # torch.argmax(predicts, dim=1): This function finds the indices
```

```
(i.e., the classes)
            # of the maximum values along dimension 1 (the class dimension) of
predicts.
            # This effectively performs a prediction by selecting the class with
the highest score for each input in the batch.
            # The output shape matches the batch size, containing the predicted
class for each input.

            # torch.argmax(predicts, dim=1) == y: This compares the predicted
classes to the true labels y.
            # If a prediction matches the true label, the comparison returns True;
otherwise, it returns False.
            # this comparison effectively counts the correct predictions by
returning a tensor of 1s and 0s.

            # torch.sum(...): This function sums up the values in the tensor of 1s
(correct predictions) and 0s (incorrect predictions),
            # giving the total number of correct predictions in the batch.

            # item(): This method converts a PyTorch scalar tensor (a tensor with
a single value) to a Python number.
            # It's used here to extract the number of correct predictions as a
Python integer or float, which can then be accumulated in the acc variable.

            # acc += ...: This accumulates the number of correct predictions over
all batches processed in the epoch.
            # By adding up the number of correct predictions after processing each
batch, you keep a running total of how many samples have been correctly classified
so far during the epoch.
            mean_loss /= loss.item() * x.size(0)
            data_num += x.size(0)
            train_data.set_description(f"Training..Epoach:{i+1}/{epochs},Loss:
{loss.item(* x.size(0)):.4f}")

        mean_loss /=data_num
        acc /= data_num
        print(f"Training acc:{acc:.4f},Loss:{mean_loss:.4f}")

        model.eval()  # Set the model to evaluation mode评估模式
        test_data = tqdm(test_dataloader)  # Wrap dataloader with tqdm for a
progress bar
        mean_loss, acc = 0.0, 0.0
        data_num = 0
        with torch.no_grad():  # Disable gradient calculation没有梯度计算和更新
            for x, y in test_data:
                x = x.to(device)
                y = y.to(device)
                predicts = model(x)
                loss = loss_fn(predicts, y)
                # 不需要优化和更新
                acc += torch.sum(torch.argmax(predicts, dim=1) == y).item()
                mean_loss += loss.item() * x.size(0)
                data_num += x.size(0)
                test_data.set_description(
```

```
                    f"Evaluation ... Epoch: {i + 1}/{epochs}, Loss: {loss.item() *
x.size(0):.4f}")
            mean_loss /= data_num
            acc /= data_num
            print(f"\nTraining Acc: {acc:.4f}, Loss: {mean_loss:.4f}")

        torch.save(model.state_dict(), "./model.pth")
```