画像認識AI実装演習（7月2日AI班）

導入

前回、AI訓練のコードを実行し、理解を深めてもらいました。

今日はそれを自らの手で実装してみましょう

ただし、難しいので、

穴埋め形式で進めます。

穴埋めのために**インターネットで調べたり**、**俺に聞いたり**して進めましょう💪

前回のコード（参考）

今日の実装（この後に示す）のために、前回使った以下のコードも参考にしてください

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np
import matplotlib.pyplot as plt

# ================================================================================
# 1. データセットの定義（理論：データセット）
# torch.utils.data.Datasetを継承し、データ管理をクラスにまとめます。
# これが実践的なデータハンドリングの第一歩です。
# ================================================================================
class XorDataset(Dataset):
    """XOR問題のカスタムデータセットクラス"""
    def __init__(self):
        # データを定義
        self.X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
        self.y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)

    def __len__(self):
        # データセットのサンプル数を返す
        return len(self.y)

    def __getitem__(self, idx):
        # 指定されたインデックスの入力データと正解ラベルを返す
```

```
        return self.X[idx], self.y[idx]


# ================================================================================
# 2. モデルの定義（理論: モデル関数）
# nn.Moduleを継承する点は同じですが、より汎用的な書き方を意識します。
# ================================================================================
class XORNet(nn.Module):
    """XOR問題を解くためのニューラルネットワークモデル"""
    def __init__(self, input_size, hidden_size, output_size):
        super(XORNet, self).__init__()
        # 各層を定義
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, output_size)

        # 活性化関数を定義
        # ReLUは現代のニューラルネットワークで広く使われる標準的な活性化関数です
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    # （理論: 順伝播）
    def forward(self, x):
        """順伝播のプロセスを定義"""
        # 層 -> 活性化関数 -> 層 -> 活性化関数 という流れ
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        x = self.sigmoid(x)
        return x

# ================================================================================
# 3. 学習関数の定義
# 学習プロセス全体を一つの関数にまとめることで、コードの見通しが良くなります。
# ================================================================================
def train_model(model, dataloader, criterion, optimizer, epochs):
    """モデルを学習させるための関数"""
    print("学習を開始します...")
    # 損失の履歴を保存するためのリスト
    loss_history = []

    for epoch in range(epochs):
        epoch_loss = 0.0

        # （理論: ミニバッチ学習）
        # DataLoaderからバッチ単位でデータを取り出す
        for inputs, labels in dataloader:

            # --- 勾配のリセット ---
            optimizer.zero_grad()

            # --- 順伝播・損失計算・逆伝播・パラメータ更新 ---
            # （理論: 順伝播）
            outputs = model(inputs)
            # （理論: 損失関数）
            loss = criterion(outputs, labels)
```

```
            #（理論：誤差逆伝播）
            loss.backward()
            #（理論：パラメータ更新）
            optimizer.step()

            # このバッチの損失を加算
            epoch_loss += loss.item()

        # エポック全体の平均損失を計算し、履歴に保存
        avg_epoch_loss = epoch_loss / len(dataloader)
        loss_history.append(avg_epoch_loss)

        if (epoch + 1) % 1000 == 0:
            print(f'エポック: {epoch+1:5d}/{epochs}, 損失: {avg_epoch_loss:.4f}')

    print("学習が完了しました。")
    return loss_history

# ==============================================================================
# 4. 評価関数の定義
# 学習済みモデルの性能を評価する部分も関数化します。
# ==============================================================================
def evaluate_model(model, dataloader):
    """学習済みモデルの性能を評価する関数"""
    print("¥n学習済みモデルの評価:")
    # model.eval()でモデルを評価モードに切り替える
    model.eval()

    # 勾配計算を無効にする（評価時には不要なため）
    with torch.no_grad():
        for inputs, labels in dataloader:
            outputs = model(inputs)
            # 確率出力を0か1の予測に変換
            predicted = (outputs > 0.5).float()

            # バッチ内の全サンプルについて表示
            for i in range(len(inputs)):
                print(f"入力: {inputs[i].numpy()} -> "
                      f"正解: {int(labels[i].item())}, "
                      f"予測: {int(predicted[i].item())}")


# ==============================================================================
# 5. メイン実行ブロック
# ここで上記で定義した要素を組み合わせて実行します。
# ==============================================================================
if __name__ == "__main__":

    # --- ハイパーパラメータの設定 ---
    INPUT_SIZE = 2
    HIDDEN_SIZE =   16# 隠れ層のノード数を少し増やしてみる
    OUTPUT_SIZE = 1
    LEARNING_RATE = 0.01
    BATCH_SIZE = 4   # データが4つしかないので全件バッチと同じ
    EPOCHS = 10000
```

```
    # --- 1. データ準備 ---
    dataset = XorDataset()
    #（理論：ミニバッチ学習）
    # DataLoaderは、バッチ処理やデータのシャッフルを自動化してくれる便利なツール
    dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

    # --- 2. モデル・損失関数・オプティマイザの準備 ---
    model = XORNet(INPUT_SIZE, HIDDEN_SIZE, OUTPUT_SIZE)
    criterion = nn.BCELoss() #（理論：損失関数）
    # AdamはSGDを改良した、より収束が速い傾向にある人気のオプティマイザ
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

    # --- 3. 学習の実行 ---
    loss_history = train_model(model, dataloader, criterion, optimizer, EPOCHS)

    # --- 4. 評価の実行 ---
    evaluate_model(model, dataloader)

    # --- 5. 結果の可視化 ---
    plt.figure(figsize=(10, 5))
    plt.plot(loss_history)
    plt.title("Loss History")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.grid(True)
    plt.show()
```

今日の実装

今日は画像を判定できるAIを創りましょう❗

以下の穴埋めを完成させ、実行しましょう❗

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import os
import torchvision
import torchvision.transforms as transforms

# ================================================================================
# 1. データセットの定義（理論：データセット）
# torch.utils.data.Datasetを継承し、データ管理をクラスにまとめます。
# ================================================================================
class NumberDataset(Dataset):
```

```
    def __init__(self, train=True, transform=None):
        # torchvisionのMNISTデータセットをダウンロードして利用
        if transform is None:
            transform = transforms.Compose([
                transforms.ToTensor(),
                # 必要なら正規化も追加
            ])
        self.dataset = torchvision.datasets.MNIST(
            root='./data', train=train, download=True, transform=transform
        )

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):
        image, label = self.dataset[idx]
        # 画像を1次元ベクトルに変換
        image = image.view(-1)
        return image, label


# ================================================================================
# 2. モデルの定義（理論: モデル関数）
# nn.Moduleを継承する点は同じですが、より汎用的な書き方を意識します。
# ================================================================================
class MyNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyNet, self).__init__()
        self.layer1 = _____
        self.layer2 = _____
        # 必要に応じて層の数を増やす

        # 活性化関数を定義(ここではreluとsoftmax、ネットで使い方を調べてもよい 「Pytorch Relu」「Pytorch Softmax」)
        self.relu = _____
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = _____
        ...


        x = self.softmax(x)
        return x

# ================================================================================
# 3. 学習関数の定義
# 学習プロセス全体を一つの関数にまとめることで、コードの見通しが良くなります。
# ================================================================================
def train_model(model, test_loader, validation_loader, criterion, optimizer, epochs):
    """モデルを学習させるための関数"""
    print("学習を開始します...")
    # 損失の履歴を保存するためのリスト
    train_loss_history = []
```

```
    val_loss_history = []

    for epoch in range(epochs):
        epoch_loss = 0.0

        # (理論: ミニバッチ学習)
        # DataLoaderからバッチ単位でデータを取り出す
        for inputs, labels in test_loader:

            # --- 勾配のリセット ---
            optimizer._____()

            # --- 順伝播・損失計算・逆伝播・パラメータ更新 ---
            # (理論: 順伝播)
            outputs = _____(_____)
            # (理論: 損失関数)
            loss = _____(outputs, labels)
            # (理論: 誤差逆伝播)
            loss._____()
            # (理論: パラメータ更新)
            optimizer._____()

            # このバッチの損失を加算
            epoch_loss += loss.item()

        # エポック全体の平均損失を計算し、履歴に保存
        avg_epoch_loss = epoch_loss / len(test_loader)
        train_loss_history.append(avg_epoch_loss)


        # --- 検証データでの損失計算 ---
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for val_inputs, val_labels in validation_loader:
                val_outputs = model(val_inputs)
                v_loss = criterion(val_outputs, val_labels)
                val_loss += v_loss.item()
        avg_val_loss = val_loss / len(validation_loader)
        val_loss_history.append(avg_val_loss)


        if (epoch + 1) % 1000 == 0:
            print(f'エポック: {epoch+1:5d}/{epochs}, 損失: {avg_epoch_loss:.4f}, 検証損失: {avg_val_loss:.4f}')

    print("学習が完了しました。")
    return train_loss_history, val_loss_history



# ================================================================================
# 4. 評価関数の定義
# 学習済みモデルの性能を評価する部分も関数化します。
# ================================================================================
def evaluate_model(model, dataloader):
```

```
    print("¥n学習済みモデルの評価:")
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in dataloader:
            outputs = model(inputs)
            # 出力から最大値のインデックス（予測クラス）を取得
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            # バッチ内の全サンプルについて表示（最初のバッチのみ表示例）
            for i in range(len(inputs)):
                print(f"正解: {int(labels[i].item())}, 予測: {int(predicted[i].item())}")
            break  # 全部表示すると多すぎるので最初のバッチのみ
    print(f"正解率: {100 * correct / total:.2f}%")

# ==============================================================================
# 5. 可視化用関数の定義
# ==============================================================================
def visualize_predictions(model, dataset, title, num_samples=10):
    model.eval()
    plt.figure(figsize=(10, 2))
    with torch.no_grad():
        for i in range(num_samples):
            image, label = dataset[i]
            output = model(image.unsqueeze(0))
            pred = output.argmax(dim=1).item()
            image2d = image.view(28, 28)
            plt.subplot(1, num_samples, i+1)
            plt.imshow(image2d, cmap="gray")
            plt.title(f"T:{label}¥nP:{pred}")
            plt.axis("off")
    plt.suptitle(title)
    plt.show()

# ==============================================================================
# 6. メイン実行ブロック
# ここで上記で定義した要素を組み合わせて実行します。
# ==============================================================================
if __name__ == "__main__":
    # --- ハイパーパラメータの設定 ---
    INPUT_SIZE = 28 * 28  # MNIST画像は28x28
    HIDDEN_SIZE = 128
    OUTPUT_SIZE = 10      # 10クラス分類
    LEARNING_RATE = 0.001
    BATCH_SIZE = 64
    EPOCHS = 5

    VALIDATION_RATIO = 0.1  # 訓練データの10%を検証用に使用

    # --- 1. データ準備 ---
    full_train_dataset = NumberDataset(train=True)
    test_dataset = NumberDataset(train=False)
```

```
# 訓練・検証データに分割
n_total = len(full_train_dataset)
n_val = int(n_total * VALIDATION_RATIO)
n_train = n_total - n_val
train_dataset, validation_dataset = torch.utils.data.random_split(
    full_train_dataset, [n_train, n_val],
    generator=torch.Generator().manual_seed(42)  # 再現性のため
)

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
validation_loader = DataLoader(validation_dataset, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)



model = ______(INPUT_SIZE, HIDDEN_SIZE, OUTPUT_SIZE)
criterion = nn.CrossEntropyLoss()
optimizer = optim._____(model.parameters(), lr=_____)  # TODO: Adamという最適化手法を用いる

# --- 訓練前の予測可視化 ---
print("【訓練前の予測】")
visualize_predictions(model, test_dataset, "Before Training (T:正解, P:予測)")

# --- 3. 学習の実行 ---
train_loss_history, val_loss_history = train_model(_____, train_loader, val_loader, _____, _____, EPOCHS)

# --- 訓練後の予測可視化 ---
print("【訓練後の予測】")
visualize_predictions(model, test_dataset, "After Training (T:正解, P:予測)")

# --- 4. 評価の実行 ---
evaluate_model(model, test_loader)

# --- 5. 結果の可視化 ---
plt.figure(figsize=(10, 5))
plt.plot(train_loss_history, label="Train Loss")
plt.plot(val_loss_history, label="Validation Loss")
plt.title("Loss History")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.grid(True)
plt.legend()
plt.show()
```