

AI実装の基礎を学ぶための実践課題

序文 この課題は、AIの学習原理を理解すると同時に、PyTorchを使った実装における基本的な選択肢（データの与え方、モデルの部品、学習方法など）について学ぶことを目的とします。

AI理論復習のためのコード

こちらは今までに学習したAI学習の流れを復讐するためのコードです。こちらの出力をよく確認して、今までに学習した理論と、実際のデータでの重みの変化に着目して理解を試みてください

```
import torch
import torch.nn as nn
import torch.optim as optim

# 結果を固定するため、乱数シードを設定します
torch.manual_seed(0)

# =====
# 1. 準備（データセット、モデル、損失関数、オプティマイザ）
# ※この部分は前回のコードと同じです
# =====

# --- データセット ---
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)

# --- モデル定義 ---
class XORNet(nn.Module):
    def __init__(self):
        super(XORNet, self).__init__()
        self.hidden_layer = nn.Linear(2, 16)
        self.output_layer = nn.Linear(16, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        hidden_output = self.sigmoid(self.hidden_layer(x))
        output = self.sigmoid(self.output_layer(hidden_output))
        return output

# --- モデル、損失関数、オプティマイザのインスタンス作成 ---
model = XORNet()
criterion = nn.BCELoss()
lr = 0.1 # 学習率を後で使うために変数に入れておく
optimizer = optim.SGD(model.parameters(), lr=lr)
```

```
# =====
# 2. 学習プロセスのステップ・バイ・ステップ実行
# ここからが本題です。学習の最初の1ステップを詳細に追跡します。
# =====

print("===== 学習の1ステップを可視化します =====")

# --- 特定のパラメータに注目する ---
# 例として、隠れ層の最初の重みとバイアスに注目します
# model.parameters()の中身はジェネレータなのでリストに変換
params = list(model.parameters())
hidden_weights = params[0]
hidden_biases = params[1]

print(f"【初期状態】 隠れ層の重み(一部): {hidden_weights[0, 0].item():.4f}")
print("-" * 50)

# --- ステップ1: 順伝播 (Forward Pass) ---
# 入力データXをモデルに通し、予測値 y_pred を得ます。
y_pred = model(X)
print("【ステップ1: 順伝播】")
print(f"正解ラベル y:{n{y.T}") # .T で転置して見やすくする
print(f"モデルの予測値 y_pred:{n{y_pred.T.detach().numpy()}) # .detach()で勾配計算グラフから切り離し、numpyに変換
print("-" * 50)

# --- ステップ2: 損失計算 (Calculate Loss) ---
# 予測値 y_pred と正解ラベル y のズレを損失関数で計算します。
loss = criterion(y_pred, y)
print("【ステップ2: 損失計算】")
print(f"計算された損失 (Loss): {loss.item():.4f}")
print("-" * 50)

# --- ステップ3: 逆伝播 (Backpropagation) ---
# 損失を基に、各パラメータの勾配を計算します。
# まず、前回の勾配が残らないようにリセットします。
optimizer.zero_grad()
# 損失から逆方向に勾配を計算します。
loss.backward()

# 注目しているパラメータの勾配を確認します
hidden_weights_grad = hidden_weights.grad
print("【ステップ3: 逆伝播 (勾配計算)】")
print("各パラメータについて、「損失を小さくするには、この値をどちらにどれだけ動かせば良いか」を示す「勾配」が計算されました。")
print(f"隠れ層の重み(一部)に対する勾配: {hidden_weights_grad[0, 0].item():.4f}")
print("-" * 50)

# --- ステップ4: パラメータ更新 (Update Parameters) ---
# オプティマイザが勾配を基にパラメータを更新します。
```

```
# SGDの更新式: 新しい重み = 古い重み - 学習率 * 勾配

# 更新前の値を保存しておきます
weight_before_update = hidden_weights[0, 0].item()

# オプティマイザにパラメータを更新させます
optimizer.step()

# 更新後の値を取得します
weight_after_update = hidden_weights[0, 0].item()

print("【ステップ4: パラメータ更新】")
print("SGDのルールに従って、重みが更新されました。")
print(f"更新前の重み: {weight_before_update:.4f}")
print(f" - (学習率 {lr} * 勾配 {hidden_weights_grad[0, 0].item():.4f})")
manual_calculation = weight_before_update - (lr * hidden_weights_grad[0, 0].item())
print(f" = 手計算による更新後の重み: {manual_calculation:.4f}")
print(f"実際に更新された重み: {weight_after_update:.4f}")
print("-" * 50)

print("=====  
可視化終了 =====")
print("ニューラルネットワークの学習では、この4ステップを何千・何万回と繰り返すことで、")
print("損失が徐々に小さくなるように、すべてのパラメータが少しずつ調整されていきます。")
```

演習に使うコード

以下はPytorchというライブラリを用いた実際の理想的なAI訓練コードです。こちらを実行し、またある程度の流れを今まで学習したAI理論と結び付けて考えてみましょう

```
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

# =====
# 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()
```

ステップ1: AIの動作を「観察」する

まず、コードの主要な部品が何をしているのかを、`print` 文を使って確認します。

課題1: AIが学習する「問題集」の確認

- **目的**: AIが学習に用いるデータ（入力と正解のペア）の内容を理解する。
- **手順**: `if __name__ == "__main__":` ブロック内にある `dataset = XorDataset()` の直後に、以下の4行を追加して実行してください。

```
print("--- データセットの内容 ---")
for i in range(len(dataset)):
    input_data, correct_answer = dataset[i]
    print(f"問題{i+1}: 入力 {input_data.numpy()} => 正解 {correct_answer.numpy()}")
print("-" * 25)
```

- **考察** : AIが何を学習しようとしているのかを説明してください。

課題2 : AIの「脳の設計図」の確認

- **目的** : ニューラルネットワークモデルがどのような層で構成されているかを確認する。
- **手順** : `if __name__ == "__main__":` ブロック内にある `model = XORNet(...)` の直後に、`print(model)` という1行を追加して実行してください。
- **考察** : `XORNet` クラスの `__init__` メソッド内のコード (`self.layer1` など) と、`print` で表示された設計図が、どのように対応しているか説明してください。

ステップ2 : AIの「学習方法」を実験する

ここでは、AIにどうやって勉強させるか、その「方法」に関する設定を変更して結果を比較します。

課題3 : 学習の勢い（学習率）の実験

- **目的** : 学習率の大きさが、学習の進み方に与える影響を体験する。
- **手順** : `LEARNING_RATE` の値を `0.1` (大きい値) と `0.0001` (小さい値) にそれぞれ変更して実行し、結果を比較します。
- **考察** : 学習率が大きい場合と小さい場合で、損失の減り方はどう違いましたか。それぞれの長所と短所を記述してください。

課題4 : 学習の戦略（オプティマイザ）の実験

- **目的** : 学習のパラメータを更新するアルゴリズム（オプティマイザ）による違いを体験する。
- **手順** : `optimizer = optim.Adam(...)` の行の先頭に `#` を付けて無効化し、代わりに以下の行を有効にしてください。

```
# optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)
```

- **考察** : `Adam` と `SGD`、2つのオプティマイザで学習の進み方に違いはありましたか。どちらが速く損失を下げるのができたかを報告してください。

課題5 : 一度に解く問題数（バッチサイズ）の実験

- **目的**：データをまとめて処理する「バッチ処理」の意味を理解する。
- **手順**：`BATCH_SIZE` の値を `1` に変更して実行してください。これは「1問ずつ問題を解いては、すぐに答え合わせと復習をする」学習方法に相当します。
- **考察**：バッチサイズを `4` から `1` に変更したとき、損失グラフの形はどのように変化しましたか（例：滑らか、ギザギザなど）。なぜデータをまとめて処理する（バッチサイズを1より大きくする）方が、学習が安定しやすいのか、理由を推測してください。

課題6：間違いの測り方（損失関数）の実験

- **目的**：問題の種類（今回は分類問題）に適した損失関数を選ぶ重要性を理解する。
- **手順**：`criterion = nn.BCELoss()` の行をコメントアウトし、代わりに `criterion = nn.MSELoss()` という行を追加して実行してください。`MSELoss` は主に回帰問題（数値を予測する問題）で使われます。
- **考察**：損失関数を `MSELoss` に変更した結果、AIはうまく学習できましたか。分類問題で「間違いの大きさ」を測るのに、`BCELoss` が適している理由を考察してください。

ステップ3：AIの「設計図」を実験する

ここでは、AIの脳の構造、つまり「設計図」そのものを変更して、その影響を見ます。

課題7：脳の複雑さ（隠れ層のサイズ）の実験

- **目的**：隠れ層のニューロンの数が、問題を解く能力にどう影響するかを体験する。
- **手順**：`HIDDEN_SIZE` の値を `2` に変更して実行し、元の `16` の場合と学習結果を比較してください。
- **考察**：隠れ層のニューロン数を減らすと、学習結果にどのような影響がありましたか。

課題8：脳の部品（活性化関数）の交換実験

- **目的**：モデルの部品である活性化関数は、交換可能であり、種類によって特性が違っていることを理解する。
- **手順**：`XORNet` クラスの `__init__` メソッド内にある `self.relu = nn.ReLU()` を `self.relu = nn.Sigmoid()` に変更して実行してください。（`self.sigmoid` はすでにあるので、`self.relu` の行を修正するだけでOKです）
- **考察**：活性化関数を `ReLU` から `Sigmoid` に変えたとき、学習の進み方に違いはありましたか。

課題9：nn.Sequentialによるシンプルなモデル定義

- **目的**：PyTorchの `nn.Sequential` を使うことで、モデルの定義がどれだけ簡潔になるかを体験する。
- **手順**：`XORNet` クラスの定義を `nn.Sequential` を使った形に書き換えて実行してください。具体的には、以下のように変更します。

- **考察** : `nn.Sequential` を使うことで、モデルの定義がどのように簡潔になったか、またその利点と欠点について考察してください。

同じXOR問題を解くネットワークを `nn.Sequential` で実装した例を以下に示します。

```
# nn.Sequentialを使ったモデル定義例
model = nn.Sequential(
    nn.Linear(2, 16),
    nn.ReLU(),
    nn.Linear(16, 1),
    nn.Sigmoid()
)
```

この方法では、**forward** メソッドを自分で書かずに、層を順番に並べるだけでモデルを構築できます。

ただし、複雑な処理や分岐が必要な場合は、従来通り **nn.Module** を継承して **forward** を定義する方法が推奨されます。

ステップ4 : AIの「才能の秘密」を証明する

最後に、これまでの実験を踏まえ、AIをAIたらしめる最も重要な原理を証明します。

課題9 : 【最重要】AIから「ひらめき」を奪う実験

- **目的** : ニューラルネットワークが複雑な問題を解ける根源である「非線形活性化関数」の不可欠性を証明する。
- **手順** : `XORNet` クラスの `forward` メソッド内にある `x = self.relu(x)` の行を、`#` を使ってコメントアウト（無効化）し、実行してください。
- **考察** :
 1. **実験結果の報告** : `relu` を無効にした結果、AIはXOR問題を学習できましたか。最終的な損失の値はどうなりましたか。
 2. **結論** : この実験結果から、活性化関数はAIにとってどのような役割を果たしていると結論付けられますか。なぜこれがないと、XOR問題のような単純な直線では分けられない問題を解けないのか、その理由を説明してください。