深層学習実装マニュアル**：MNIST分類器をPyTorchで構築する

はじめに：深層学習実装の全体像とプログラミングの基本

1.1. このマニュアルの目的と対象者

このマニュアルは、AIの理論は理解しているものの、Pythonでの実装経験が少ない社会人初心者を対象としています。手書き数字の分類という具体的なタスク（MNIST）を通して、深層学習モデルをPyTorchで構築し、学習させる一連のプロセスを体系的に学びます。

- **深層学習：多層のニューラルネットワークを用いてデータから特徴を自動的に学習する機械学習の一分野です。**
- **MNIST：手書き数字の画像データセットで、0から9までの数字の画像とその正解ラベルが含まれます。深層学習の入門やベンチマークによく用いられます。**
- **PyTorch：Facebook（現Meta）が開発したオープンソースの機械学習ライブラリで、特に深層学習モデルの構築と訓練に強みを持つ。柔軟性が高く、Pythonでの記述が直感的であるため、研究開発から実用まで幅広く利用されています。**

学ぶことの全体像: 深層学習の実装は、以下の主要なステップで構成されます。

1. **データ準備:** モデルに学習させるためのデータを整える。
2. **モデル構築:** ニューラルネットワークの構造を定義する。
3. **学習:** モデルにデータからパターンを学ばせる。
4. **評価:** 学習したモデルの性能を測る。
5. **可視化:** 学習の過程や結果をグラフなどで確認する。

PyTorchは、これらの深層学習モデルの構築と学習を効率的に行うための強力なオープンソースライブラリです。なぜPyTorchを使うのかというと、柔軟性が高く、Pythonのコードとして直感的に記述できるため、研究開発から実用まで幅広く利用されているからです。

1.2. プログラミング設計の基本概念

深層学習の実装に限らず、プログラミングにおいて「設計」は非常に重要です。特に、コードを「クラス」や「関数」に分割する「モジュール化」は、以下の点

で大きなメリットがあります。

- **モジュール化**：プログラム全体を機能ごとに小さな部品（モジュール）に分割すること。これにより、コードの管理がしやすくなり、再利用性や可読性が向上します。
- **関数**：特定の処理をひとまとまりにしたコードのブロック。同じ処理を何度も書く手間を省き、コードの再利用性を高めます。
- **クラス**：データ（属性）と、そのデータを操作する処理（メソッド）をひとまとめにした「設計図」。現実世界のモノや概念をプログラム上で表現する際に用いられます。
- **インスタンス（オブジェクト）**：クラスという設計図に基づいて実際に作られた具体的なもの。クラスの定義に従って、それぞれが独自のデータを持つことができます。
- **オブジェクト指向プログラミング**：プログラムを「オブジェクト」という単位で組み立てていくプログラミングの考え方。クラス、継承、カプセル化、ポリモーフィズムなどの概念が中心となる。
- **継承**：既存のクラス（親クラス）の機能や特性を新しいクラス（子クラス）が引き継ぐ仕組み。共通のコードを再利用し、効率的にプログラムを拡張できます。
- **torch.utils.data.Dataset、torch.nn.Module**：PyTorchでデータセットやニューラルネットワークモデルを定義する際に、その「ひな形」となる基底クラス。これらを継承することで、PyTorchの機能と連携しやすくなります。

- **再利用性:** 一度書いたコードを別の場所や別のプロジェクトで使い回せる。
- **可読性:** コードの各部分が何をしているのかが分かりやすくなる。
- **保守性:** 問題が発生した際に、原因の特定や修正がしやすくなる。

Pythonの基礎：関数とクラス

Pythonでは、特定の処理をまとめるために「関数」を定義します。

```
# 関数の定義例
def greet(name):
    return f”こんにちは、{name}さん！”

# 関数の呼び出し例
message = greet(”田中”)
print(message) # 出力: こんにちは、田中さん！
```

さらに、データとそれに関連する処理をひとまとめにするのが「クラス」です。クラスは「設計図」のようなもので、その設計図から作られる具体的なものが「インスタンス（オブジェクト）」です。

```
# クラスの定義例
class Dog:
    def __init__(self, name, breed): # インスタンスが作られるときに呼ばれるメソッド
        self.name = name
        self.breed = breed

    def bark(self): # インスタンスの振る舞いを定義するメソッド
        return f"{self.name}がワンワンと吠える！"

# クラスからインスタンスを作成（オブジェクト化）
my_dog = Dog("ポチ", "柴犬")

# インスタンスのメソッドを呼び出す
print(my_dog.bark()) # 出力: ポチがワンワンと吠える！
```

オブジェクト指向プログラミングの初歩：継承

オブジェクト指向プログラミングの重要な概念の一つに「継承」があります。これは、既存のクラス（親クラス）の機能を引き継ぎ、新しいクラス（子クラス）を作成する仕組みです。これにより、共通の機能を何度も書く手間を省き、コードの再利用性を高めることができます。

Pythonでは、以下のように記述します。

```
class ParentClass:
    def common_method(self):
        print("これは親クラスの共通メソッドです。")

class ChildClass(ParentClass): # ParentClassを継承
    def unique_method(self):
        print("これは子クラス独自のメソッドです。")

# 子クラスのインスタンスを作成
child_obj = ChildClass()

# 親クラスから継承したメソッドを呼び出す
child_obj.common_method() # 出力: これは親クラスの共通メソッドです。

# 子クラス独自のメソッドを呼び出す
child_obj.unique_method() # 出力: これは子クラス独自のメソッドです。
```

深層学習のPyTorch実装では、データセットやモデルの定義にこの「クラス」と「継承」の概念が頻繁に登場します。特に、PyTorchが提供する `torch.utils.data.Dataset` や `torch.nn.Module` といった基底クラスを継承して、独自のデータセットやモデルを構築することになります。

ステップ1：データセットの準備とPythonクラスの活用

2.1. データセットの役割とPyTorchでの表現

深層学習において「データセット」は、モデルが学習するための入力データ（画像、テキストなど）と、それに対応する正解ラベル（手書き数字の「5」など）の集合を指します。PyTorchでは、このデータセットを効率的に管理するために`torch.utils.data.Dataset`という抽象クラスが提供されています。

`Dataset`クラスを継承することで、以下の2つの重要なメソッドを実装するだけで、PyTorchがデータを扱うための標準的なインターフェースを提供できます。

- **`データセット`：モデルが学習するために使う、入力データとそれに対応する正解ラベルの集まりです。**
- **`抽象クラス`：それ自体はインスタンス化されず、他のクラスに継承されることを前提としたクラスです。共通のインターフェース（メソッドの定義）を強制することで、コードの一貫性を保ちます。**
- **`__len__(self)`、`__getitem__(self, idx)`：Pythonの特殊メソッド（マジックメソッド）で、それぞれ`len()`関数や`[]`（インデックスアクセス）演算子に対応します。PyTorchの`Dataset`クラスを継承する際には、これらのメソッドを実装する必要があります。**

- `__len__(self)`: データセットに含まれるサンプルの総数を返します。
- `__getitem__(self, idx)`: 指定されたインデックス`idx`に対応する1つのデータサンプル（入力と正解ラベルのペア）を返します。

2.2. MNISTデータセットの読み込み

MNISTは手書き数字の画像データセットで、深層学習の入門によく使われます。PyTorchの`torchvision`ライブラリを使えば、簡単にダウンロードして利用できます。

`torchvision.datasets.MNIST`は、MNISTデータセットを扱うための便利なクラスです。

- **`torchvision`：PyTorchの公式ライブラリの一つで、画像処理やコンピュータビジョン関連のデータセット、モデル、変換（Transformations）機能を提供します。**
- **`torchvision.datasets.MNIST`：`torchvision`が提供する、MNISTデータセットを簡単に扱えるようにしたクラスです。**
- **`transform`：データセットからデータを読み込む際に、そのデータに対して適用する前処理のパイプラインです。画像のサイズ変更、正規化、テンソルへの変換などを行います。**

- `root`: データセットをダウンロードして保存するディレクトリを指定します。
- `train`: `True`にすると訓練データセットを、`False`にするとテストデータセットをロードします。
- `download`: `True`にすると、指定された`root`ディレクトリにデータセットが存在しない場合に自動的にダウンロードします。
- `transform`: データの前処理（変換）を指定します。

2.3. `NumberDataset`クラスの実装詳細

`MNIST訓練実装答え.py`では、`NumberDataset`というカスタムデータセットクラスが定義されています。これは`torch.utils.data.Dataset`を継承しています。

- **`NumberDataset`：このマニュアルで定義する、MNISTデータセットをPyTorchで扱いやすいようにカスタマイズしたクラスです。**
- **`__init__`：Pythonのクラスでインスタンスが生成される際に自動的に呼び出される特殊メソッドです。初期設定や必要なデータの読み込みなどを行います。**
- **`self.dataset`：`NumberDataset`クラスのインスタンスが持つ属性（変数）で、実際にMNISTデータセットのデータを保持しています。**
- **`transforms.ToTensor()`：画像をPyTorchのテンソル形式に変換し、同時にピクセル値を0から1の範囲に正規化する前処理です。**
- **`PIL形式の画像`：Pythonで画像を扱うための標準的なライブラリであるPillow（PILの後継）で扱われる画像形式です。**
- **`テンソル`：PyTorchやTensorFlowなどの深層学習フレームワークでデータを扱うための基本的なデータ構造です。多次元配列のようなもので、数値計算に最適化されています。**
- **`ピクセル値`：デジタル画像を構成する最小単位であるピクセルが持つ色の情報です。通常、0から255の整数値で表現されます。**
- **`正規化`：データの値を特定の範囲（例: 0から1、または平均0、標準偏差1）に変換することです。これにより、モデルの学習が安定しやすくなります。**
- **`image.view(-1)`：テンソルの形状を変更するメソッドです。`-1`を指定すると、その次元のサイズが自動的に計算されます。この場合、28x28の画像を1次元の784要素のベクトルに変換しています。**
- **`1次元ベクトル`：要素が一列に並んだデータ構造です。画像データを全結合層に入力する際によく用いられます。**
- **`全結合層（Linear層）`：ニューラルネットワークの基本的な層の一つで、入力のすべてのニューロンが出力のすべてのニューロンと結合している層です。線形変換を行います。**
- **`カプセル化`：データとそのデータを操作するメソッドを一つの単位（クラス）にまとめること。外部から直接データにアクセスするのを防ぎ、コードの保守性や安全性を高めます。**

`NumberDataset` クラスのコード（一部抜粋）：

```
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import Dataset

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: # `===` はPythonでは無効な比較演算子です。`is` または `==` を使用します。
            transform = transforms.Compose([
                transforms.ToTensor(),
                # 必要なら正規化も追加
            ])
        self.dataset = torchvision.datasets.MNIST(
            root=_____, train=_____, download=_____, transform=transform # 穴埋め部分
        )

    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(_____) # 穴埋め部分
        return image, label
```

`__init__` メソッド

このメソッドは、`NumberDataset` のインスタンスが作成されるときに最初に実行されます。ここでは、`torchvision.datasets.MNIST` を使って実際のMNISTデータセットをロードし、`self.dataset` に保持しています。

`transform` 引数がない場合は、デフォルトで `transforms.ToTensor()` が適用されます。これは、PIL形式の画像をPyTorchのテンソルに変換し、ピクセル値を0から1の範囲に正規化する重要な前処理です。

`__len__` メソッド

このメソッドは、データセットの全サンプル数を返します。`len(self.dataset)` とすることで、内部で保持しているMNISTデータセットのサイズをそのまま返しています。

`__getitem__` メソッド

このメソッドは、データセットから特定のインデックス `idx` のデータサンプルを取り出す際に呼び出されます。

`image, label = self.dataset[idx]` で、元のMNISTデータセットから画像とラベルを取得します。 `image = image.view(-1)` は、28x28ピクセルの画像を1次元のベクトル（784要素）に変換しています。これは、後で使う全結合層（Linear層）が1次元の入力を期待するためです。

プログラミング設計のポイント： `NumberDataset` クラスは、データセットの「設計図」として機能します。 `__init__` で初期設定を行い、 `__len__` と `__getitem__` というPyTorchの `Dataset` が要求する「インターフェース」を実装することで、このクラスのインスタンスがPyTorchのデータローダーと連携できるようになります。これにより、データ管理のロジックがカプセル化され、コードの見通しが良くなります。

2.4. データの前処理（Transformations）

深層学習モデルにデータを入力する前には、適切な形式に変換したり、モデルが学習しやすいように加工したりする「前処理（Transformations）」が必要です。 `torchvision.transforms` は、画像データに対する様々な前処理機能を提供します。

- **`transforms.ToTensor()`：**
  - PIL（Python Imaging Library）形式の画像やNumPyの `ndarray` をPyTorchの `Tensor` に変換します。
  - 同時に、画像のピクセル値を自動的に `[0, 255]` の範囲から `[0.0, 1.0]` の範囲に正規化します。これは、ニューラルネットワークの入力として適切なスケールです。
  - 画像の次元の並びも `(H, W, C)`（高さ，幅，チャンネル）から `(C, H, W)`（チャンネル，高さ，幅）に変更します。PyTorchの畳み込み層などがこの形式を期待するためです。
- **正規化の概念（ `transforms.Normalize` ）：**
  - `transforms.Normalize(mean, std)` を使うと、さらにデータセット全体の平均と標準偏差に基づいてデータを正規化できます。これにより、データの分布がモデルにとってより扱いやすくなり、学習の安定性や収束速度が向上することがあります。
  - MNISTの場合、ピクセル値はすでに0-1に正規化されていますが、さらに平均0、標準偏差1に近づけるような正規化を行うこともあります。

2.5. `DataLoader` によるバッチ処理

深層学習では、一度に全データを使って学習するのではなく、データを小さな塊（「ミニバッチ」）に分割して、少しずつ学習を進めるのが一般的です。このミニバッチ学習を効率的に行うために、PyTorchでは `torch.utils.data.DataLoader` が提供されています。

`DataLoader`は、`Dataset`からデータを取り出し、指定された`batch_size`ごとにまとめてモデルに供給する役割を担います。

- `batch_size`: 一度にモデルに入力するサンプル数。
- `shuffle`: `True`に設定すると、各エポックの開始時にデータセットの順序をシャッフルします。これにより、モデルがデータの特定の順序に依存して学習してしまうのを防ぎ、汎化性能の向上に役立ちます。

2.6. 訓練データと検証データの分割

モデルの学習がうまくいっているか、そして未知のデータに対しても正しく予測できるか（「汎化性能」）を評価するために、データセットを「訓練データ」と「検証データ」に分割することが非常に重要です。

- **`訓練データ`：モデルが学習するために直接使用するデータです。モデルは訓練データからパターンを学び、パラメータを調整します。**
- **`検証データ`：学習中にモデルの性能を評価するために使用するデータです。モデルのパラメータ更新には直接使われず、過学習の兆候を早期に発見するために役立ちます。**
- **`過学習`：モデルが訓練データに過剰に適合しすぎてしまい、未知のデータ（検証データやテストデータ）に対しては性能が著しく低下する現象です。汎化性能が低い状態を指します。**
- **`torch.utils.data.random_split`：PyTorchのデータセットをランダムに複数のサブセットに分割するための関数です。**
- **`torch.Generator().manual_seed(42)`：PyTorchの乱数生成器のシード（乱数の初期値）を固定する設定です。これにより、乱数に依存する処理（例: データの分割）が毎回同じ結果を生成し、実験の再現性が確保されます。**
- **`再現性`：同じ条件で実験を繰り返した場合に、常に同じ結果が得られることです。深層学習の実験では、シード固定によって再現性を確保することが重要です。**

- **`訓練データ:`** モデルが学習するために使用するデータ。
- **`検証データ:`** 学習中にモデルの性能を評価するために使用するデータ。このデータはモデルのパラメータ更新には直接使用されません。これにより、モデルが訓練データに過剰に適合しすぎる「過学習」を防ぐことができます。

`MNIST訓練実装答え.py`では、`torch.utils.data.random_split`を使って訓練データセットをさらに訓練用と検証用に分割しています。

- `torch.Generator().manual_seed(42)`: 乱数生成器のシードを固定することで、データの分割が毎回同じ結果になり、実験の「再現性」を確保できます。

ステップ2：ニューラルネットワークモデルの構築とPyTorchのモジュール

3.1. モデルの役割とPyTorchでの表現

深層学習における「モデル」とは、入力データを受け取り、何らかの変換（計算）を行って出力（予測結果）を生成する関数や構造のことです。ニューラルネットワークは、このモデルの一種です。

PyTorchでは、ニューラルネットワークモデルを構築するために`torch.nn.Module`という基底クラスが提供されています。このクラスを継承して独自のモデルクラスを定義することが、PyTorchにおけるモデル構築の基本的な方法です。

- **モデル**：入力データを受け取り、何らかの計算を行って予測結果を生成する、深層学習における中心的な構成要素です。
- **ニューラルネットワーク**：人間の脳の神経回路を模倣した計算モデルです。複数の層（ニューロンの集まり）が結合されており、データから複雑なパターンを学習できます。
- **torch.nn.Module**：PyTorchでニューラルネットワークモデルを構築する際の基底クラスです。このクラスを継承することで、モデルの層やパラメータの管理、GPUへの移動、勾配計算などが容易になります。
- **層の管理**：モデル内の各層（例: 線形層、畳み込み層）をPyTorchが自動的に認識し、そのパラメータ（重みやバイアス）を効率的に管理する機能です。
- **順伝播**：入力データがニューラルネットワークの各層を順番に通過し、最終的な出力（予測結果）が計算されるプロセスです。
- **forwardメソッド**：`nn.Module`を継承したクラスで必ず実装するメソッドです。モデルにデータが入力されたときに、そのデータがどのように各層を伝播していくか（順伝播の計算グラフ）を定義します。
- **GPU対応**：モデルの計算をCPUではなく、より高速なGPU（Graphics Processing Unit）で行うことができる機能です。大規模な深層学習モデルの訓練時間を大幅に短縮できます。
- **model.to('cuda')**：PyTorchのモデルをGPU（CUDAデバイス）に移動させるためのコードです。これにより、モデルの計算がGPU上で行われるようになります。
- **パラメータの自動追跡**：モデルの学習可能なパラメータ（重みとバイアス）に対して、勾配計算に必要な情報をPyTorchが自動的に記録・追跡する機能です。これにより、誤差逆伝播が容易になります。
- **勾配**：損失関数の値を最小化するために、各パラメータをどの方向にどれだけ変化させるべきかを示す値です。微分によって計算されます。

`nn.Module`を継承する主な理由は以下の通りです。

- **層の管理:** モデル内の各層（例: 線形層、畳み込み層）を自動的に認識し、パラメータを管理してくれます。

- **順伝播の定義:** `forward` メソッドを実装することで、入力データがモデル内をどのように伝播していくかを定義します。
- **GPU対応:** `model.to('cuda')` のように記述するだけで、モデル全体をGPUに簡単に移動できます。
- **パラメータの自動追跡:** 勾配計算に必要なパラメータを自動的に追跡し、最適化プロセスを容易にします。

3.2. `MyNet` クラスの構造

`MNIST訓練実装答え.py` では、`MyNet` というニューラルネットワークモデルが定義されています。これは `nn.Module` を継承しています。

- **`MyNet`**：このマニュアルで定義する、手書き数字分類のためのシンプルなニューラルネットワークモデルのクラスです。
- **`super(MyNet, self).__init__()`**：Pythonで子クラスの `__init__` メソッド内で、親クラスの `__init__` メソッドを呼び出すための標準的な記述です。これにより、親クラスの初期化処理が実行され、`nn.Module` の持つ機能（パラメータ管理など）が `MyNet` に引き継がれます。
- **`nn.Linear(in_features, out_features)`**：PyTorchで全結合層（線形層）を定義するためのクラスです。`in_features` は入力の次元数、`out_features` は出力の次元数を指定します。
- **重みとバイアス**：ニューラルネットワークの各層における学習可能なパラメータです。重みは入力の特徴の重要度を調整し、バイアスは出力のオフセットを調整します。
- **行列積と加算**：線形層で行われる基本的な数学的演算です。入力ベクトルに重み行列を掛け（行列積）、バイアスベクトルを加算することで、次の層への出力が計算されます。
- **`input_size`、`hidden_size`、`output_size`**：ニューラルネットワークの構造を定義する際の重要なハイパーパラメータです。`input_size` は入力層のニューロン数、`hidden_size` は隠れ層のニューロン数、`output_size` は出力層のニューロン数を指します。
- **活性化関数**：ニューラルネットワークの各ニューロンの出力に非線形性をもたらす関数です。これにより、モデルはより複雑なパターンを学習できるようになります。
- **`nn.ReLU()` (Rectified Linear Unit)**：代表的な活性化関数の一つで、入力が0以下なら0、0より大きければその値をそのまま出力するシンプルな関数です。計算が高速で、勾配消失問題の緩和に役立ちます。
- **非線形性**：入力と出力の関係が直線的ではない性質です。活性化関数によってニューラルネットワークに非線形性が導入されることで、線形モデルでは表現できない複雑な関係を学習できるようになります。
- **`nn.Softmax(dim=-1)`**：主に多クラス分類問題の出力層で用いられる活性化関数です。入力された複数の値を、合計が1になるような確率分布に変換します。`dim=-1` は、最後の次元に沿ってSoftmaxを適用することを意味します。
- **分類問題の出力層**：入力データがどのカテゴリ（クラス）に属するかを予測する問題において、最終的な予測結果を出力するニューラルネットワークの層です。通常、各クラスに属する確率を出力しま

す。

- **確率分布**：ある事象が起こる可能性を数値で示したものです。分類問題では、入力データが各クラスに属する確率の集まりとして表現されます。
- **インスタンス変数**：クラスのインスタンス（オブジェクト）ごとに異なる値を保持できる変数です。`self.変数名`の形式で定義され、そのインスタンスの生存期間中、値を保持します。

`MyNet`クラスのコード（一部抜粋）：

`MyNet`クラスのコード（一部抜粋）：

```
import torch.nn as nn

class MyNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyNet, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, hidden_size)
        self.layer4 = nn.Linear(hidden_size, output_size)

        # 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 = nn.ReLU()
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        x = self.relu(x)
        x = self.layer3(x)
        x = self.relu(x)
        x = self.layer4(x)
        x = self.softmax(x)
        return x
```

`__init__`メソッド

このメソッドは、`MyNet`のインスタンスが作成されるときに実行され、モデルの「部品」となる各層や活性化関数を定義します。

- `super(MyNet, self).__init__()`: これは、継承元の`nn.Module`クラスの初期化メソッドを呼び出すための定型句です。必ず記述する必要があります。
- **`nn.Linear(in_features, out_features)`（線形層/全結合層）：**
  - 入力`in_features`の数と出力`out_features`の数を指定します。

- この層は、入力データに対して線形変換（重みとバイアスによる行列積と加算）を行います。
- `MyNet`では、`input_size`（784）から`hidden_size`（128）への変換、隠れ層間の変換、そして最後の隠れ層から`output_size`（10クラス）への変換を行っています。

- **活性化関数:**
  - **`nn.ReLU()` (Rectified Linear Unit):** `max(0, x)`というシンプルな関数で、ニューラルネットワークに非線形性をもたらします。これにより、モデルはより複雑なパターンを学習できるようになります。
  - **`nn.Softmax(dim=-1)`:** 主に分類問題の出力層で使われます。入力された値を、合計が1になるような確率分布に変換します。`dim=-1`は、最後の次元（この場合、10クラスの出力）に沿ってSoftmaxを適用することを意味します。

プログラミング設計のポイント: `__init__`メソッドでは、モデルの「骨格」となる各層や活性化関数をインスタンス変数として定義します。これにより、これらの部品がモデルの内部状態として保持され、`forward`メソッドで利用できるようになります。これは、モデルの構造を明確にし、再利用可能な部品として管理するためのオブジェクト指向的なアプローチです。

実装課題: `MyNet`クラスの`__init__`メソッドで、各線形層と活性化関数を正しく定義してください。

```
class MyNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyNet, self).__init__()
        self.layer1 = nn.Linear(______, ______) # 入力層から隠れ層
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, hidden_size)
        self.layer4 = nn.Linear(______, ______) # 隠れ層から出力層

        self.relu = nn.______() # ReLU活性化関数
        self.softmax = nn.______(dim=-1) # Softmax活性化関数

    # ... (forwardメソッドは省略)
```

解答:

```
class MyNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyNet, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, hidden_size)
        self.layer4 = nn.Linear(hidden_size, output_size)

        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=-1)
```

```
    # ... (forwardメソッドは省略)
```

3.3. `forward` メソッドの実装

`forward` メソッドは、モデルにデータが入力されたときに、そのデータが各層をどのように伝播していくか（順伝播の計算グラフ）を定義します。

- **順伝播の計算グラフ：入力データがモデルの各層を通過し、どのような計算が順に行われて最終的な出力が得られるかを示す一連の処理の流れです。`forward` メソッドで定義されます。**

`MyNet` の `forward` メソッドでは、入力 `x` が `layer1` を通過し、`relu` で活性化され、次に `layer2`、`relu`、`layer3`、`relu`、`layer4` を順に通過し、最後に `softmax` で処理されて出力が返されます。

実装課題: `MyNet` クラスの `forward` メソッドは、入力 `x` が各層と活性化関数をどのように通過するかを定義します。以下の穴埋めコードを完成させてください。

```
class MyNet(nn.Module):
    # ... (__init__メソッドは省略)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        x = self.relu(x)
        # ここにlayer3とreluの処理を追加
        x = self._____(x) # layer3の適用
        x = self._____(x) # reluの適用
        x = self.layer4(x)
        x = self.softmax(x)
        return x
```

解答:

```
class MyNet(nn.Module):
    # ... (__init__メソッドは省略)

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

```
        x = self.layer2(x)
        x = self.relu(x)
        x = self.layer3(x)
        x = self.relu(x)
        x = self.layer4(x)
        x = self.softmax(x)
        return x
```

ステップ3：モデルの学習プロセスと最適化の仕組み

4.1. 学習プロセスの全体像

深層学習における「学習」とは、モデルが与えられたデータからパターンを抽出し、未知のデータに対して正確な予測ができるように、モデル内部のパラメータ（重みとバイアス）を調整するプロセスです。

`MNIST訓練実装答え.py` では、 `train_model` という関数に学習プロセス全体がまとめられています。このように学習のロジックを関数としてカプセル化することで、コードの見通しが良くなり、再利用性も高まります。

- **学習**：モデルが与えられたデータからパターンを抽出し、未知のデータに対して正確な予測ができるように、モデル内部のパラメータ（重みとバイアス）を調整するプロセスです。
- **パラメータ（重みとバイアス）**：ニューラルネットワークの各層における学習可能な数値です。これらの値を調整することで、モデルはデータから特徴を学び、予測能力を向上させます。
- **train_model**：このマニュアルで定義する、モデルの学習プロセス全体をまとめた関数です。
- **カプセル化**：データとそのデータを操作するメソッドを一つの単位（クラスや関数）にまとめることです。これにより、コードの見通しが良くなり、再利用性や保守性が向上します。
- **順伝播（Forward Pass）**：入力データがモデルを通過し、予測結果が生成されるプロセスです。
- **損失計算（Loss Calculation）**：モデルの予測結果と正解ラベルとの間の誤差を数値化するプロセスです。
- **誤差**：モデルの予測と実際の正解との間のずれや違いです。この誤差を小さくすることが学習の目標となります。
- **損失関数**：モデルの予測結果と正解ラベルの誤差を数値化する関数です。この値が小さいほど、モデルの予測は正確であると判断できます。
- **誤差逆伝播（Backward Pass）**：損失関数の誤差を基に、モデルの各パラメータ（重みとバイアス）が、その誤差にどれだけ寄与したかを計算するプロセスです。
- **勾配の計算**：誤差逆伝播によって、各パラメータに対する損失関数の傾き（勾配）を求めることです。この勾配が、パラメータを更新する方向と大きさを決定します。

- **パラメータ更新（Parameter Update）：計算された勾配に基づいて、モデルのパラメータ（重みとバイアス）を微調整するプロセスです。**
- **最適化手法：損失関数の値を最小化するために、モデルのパラメータをどのように更新するかを決定するアルゴリズムです。**

学習プロセスは、通常、以下のサイクルを繰り返します。

1. **順伝播（Forward Pass）:** 入力データをモデルに通し、予測結果を得る。
2. **損失計算（Loss Calculation）:** 予測結果と正解ラベルを比較し、その「誤差」を数値化する（損失関数）。
3. **誤差逆伝播（Backward Pass）:** 損失をモデルの各パラメータにどのように分配するかを計算する（勾配の計算）。
4. **パラメータ更新（Parameter Update）:** 計算された勾配に基づいて、モデルのパラメータを微調整する（最適化手法）。

4.2. 損失関数（Criterion）の選択

「損失関数（Loss Function）」または「基準（Criterion）」は、モデルの予測結果がどれだけ正解から離れているかを示す指標です。この値が小さいほど、モデルの予測は正確であると判断できます。

- `nn.CrossEntropyLoss()`:
  - 主に多クラス分類問題で使われる損失関数です。
  - モデルの出力（Softmaxを適用する前の「ロジット」と呼ばれる値、またはSoftmax適用後の確率分布）と、正解ラベル（クラスのインデックス）を受け取ります。
  - 内部でSoftmaxと負の対数尤度損失（Negative Log Likelihood Loss）を組み合わせて計算するため、モデルの出力層で`nn.Softmax`を明示的に適用する必要がない場合もありますが、今回のコードでは明示的に適用しています。

4.3. 最適化手法（Optimizer）の選択

「最適化手法（Optimizer）」は、損失関数の値を最小化するために、モデルのパラメータをどのように更新するかを決定するアルゴリズムです。

- `torch.optim.Adam()`:
  - 現在、最も広く使われている最適化手法の一つです。
  - 学習率を自動的に調整する機能（適応的学習率）を持っており、効率的に学習を進めることができます。
  - `model.parameters()`: モデル内の学習可能なすべてのパラメータ（重みとバイアス）を最適化器に渡します。
  - `lr`（学習率）: パラメータを更新する際の「歩幅」の大きさを決定します。この値が大きすぎると学習が不安定になり、小さすぎると学習に時間がかかります。

4.4. 学習ループの核心：ミニバッチ学習

`train_model`関数内の`for inputs, labels in test_loader:`ループが、ミニバッチ学習の核心です。`DataLoader`からバッチ単位でデータを取り出し、以下のステップを繰り返します。

勾配のリセット

```
optimizer.zero_grad()
```

- **原理原則:** ニューラルネットワークの学習では、各バッチの処理ごとに勾配を計算し、パラメータを更新します。PyTorchでは、勾配はデフォルトで累積されるため、新しいバッチの処理を開始する前に、前回の勾配をゼロにリセットする必要があります。これを忘れると、誤った勾配でパラメータが更新されてしまいます。

順伝播

```
outputs = model(inputs)
```

- **原理原則:** 入力データ`inputs`を定義した`model`（`MyNet`のインスタンス）に渡し、モデルの予測結果`outputs`を得るプロセスです。これは`MyNet`クラスで定義した`forward`メソッドが実行されることに相当します。

損失計算

```
loss = criterion(outputs, labels)
```

- **原理原則:** モデルの予測結果`outputs`と、実際の正解ラベル`labels`を比較し、その間の「誤差」を`criterion`（損失関数）を使って数値化します。この`loss`の値が、モデルがどれだけ間違っているかを示します。

誤差逆伝播

```
loss.backward()
```

- **原理原則:** 計算された`loss`（誤差）を基に、モデルの各パラメータ（重みとバイアス）が、その誤差にどれだけ寄与したかを計算します。このプロセスは「誤差逆伝播法（Backpropagation）」と呼ばれ、連鎖律の原理を用いて、出力層から入力層に向かって勾配を効率的に計算します。このステップで、各パラメータの`grad`属性に勾配が格納されます。

パラメータ更新

```
optimizer.step()
```

- **原理原則:** `loss.backward()` で計算された勾配（`grad` 属性に格納されている値）と、選択した最適化手法（`Adam`）のアルゴリズムに基づいて、モデルのパラメータ（`model.parameters()`）を実際に更新します。これにより、モデルは次のバッチでより正確な予測ができるように調整されます。

実装課題: `train_model` 関数内で、各エポックの終わりに訓練データでの平均損失を計算し、`train_loss_history` に追加する部分を完成させてください。

```
def train_model(model, test_loader, validation_loader, criterion, optimizer, epochs):
    # ... (前略)
    train_loss_history = []
    # ... (中略)
    for epoch in range(epochs):
        epoch_loss = 0.0
        for inputs, labels in test_loader:
            # ... (勾配のリセット、順伝播、損失計算、逆伝播、パラメータ更新)
            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(______) # 穴埋め部分
        train_loss_history.append(__________)      # 穴埋め部分
        # ... (後略)
```

解答:

```
def train_model(model, test_loader, validation_loader, criterion, optimizer, epochs):
    # ... (前略)
    train_loss_history = []
    # ... (中略)
    for epoch in range(epochs):
        epoch_loss = 0.0
        for inputs, labels in test_loader:
            # ... (勾配のリセット、順伝播、損失計算、逆伝播、パラメータ更新)
            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)
        # ... (後略)
```

4.5. 検証データでの損失計算

学習中に訓練データだけでなく、検証データでもモデルの性能を評価することは非常に重要です。これにより、モデルが訓練データに過学習していないか（つまり、未知のデータに対しても汎化できているか）を確認できます。

- `model.eval()`: モデルを評価モードに切り替えます。これにより、ドロップアウト層やバッチ正規化層など、訓練時と評価時で挙動が変わる層が適切に動作するようになります。
- `with torch.no_grad():`: このブロック内では、PyTorchは勾配の計算を行いません。評価時にはパラメータを更新する必要がないため、勾配計算を無効にすることでメモリ使用量を削減し、計算を高速化できます。

ステップ4：モデルの評価と結果の解釈

5.1. 評価プロセスの全体像

モデルの学習が完了したら、そのモデルがどれだけ正確に予測できるかを最終的に評価する必要があります。この評価には、学習には一切使われなかった「テストデータ」を使用するのが一般的です。

`MNIST訓練実装答え.py`では、`evaluate_model`関数がモデルの評価を担当しています。

5.2. 正解率の計算

分類問題における最も一般的な評価指標の一つが「正解率（Accuracy）」です。
これは、モデルが正しく予測したサンプルの割合を示します。

- `model.eval()`と`with torch.no_grad():`は、学習時と同様に評価時にも適用されます。これは、モデルを評価モードにし、勾配計算を無効にするためです。
- `outputs = model(inputs)`: テストデータ（`inputs`）をモデルに入力し、予測結果（各クラスの確率分布）を得ます。
- `_, predicted = torch.max(outputs, 1)`:
  - `torch.max()`関数は、テンソルの最大値とそのインデックスを返します。
  - `outputs`は各クラスの確率分布なので、`dim=1`（クラスの次元）に沿って最大値のインデックス（最も確率が高いクラス）を取得します。これがモデルの予測したクラスになります。
  - `_`は、最大値自体は不要なので破棄していることを意味します。
- `total += labels.size(0)`: バッチ内のサンプル数を合計します。
- `correct += (predicted == labels).sum().item()`: 予測されたクラスが正解ラベルと一致する数を数え、合計します。
- `100 * correct / total`: 正解率をパーセンテージで計算します。

5.3. 予測結果の可視化

数値としての正解率だけでなく、実際にモデルがどのように予測しているかを視覚的に確認することは、モデルの挙動を理解する上で非常に役立ちます。

`visualize_predictions`関数は、テストデータセットからいくつかのサンプルを取り出し、元の画像、正解ラベル、そしてモデルの予測結果を並べて表示します。

- `model.eval()`と`with torch.no_grad():`はここでも適用されます。
- `image, label = dataset[i]`: データセットから画像とラベルを取得します。
- `output = model(image.unsqueeze(0))`: モデルはバッチ入力を期待するため、1つの画像でも`unsqueeze(0)`でバッチ次元を追加します。
- `pred = output.argmax(dim=1).item()`: モデルの出力から最も確率の高いクラス（予測）を取得します。
- `image2d = image.view(28, 28)`: 1次元に変換されていた画像を元の28x28の2次元に戻します。
- `matplotlib.pyplot`（`plt`）: Pythonでグラフや画像をプロットするための標準的なライブラリです。
  - `plt.figure()`: 新しい図を作成します。
  - `plt.subplot()`: 複数のプロットを1つの図に配置します。
  - `plt.imshow(image2d, cmap="gray")`: 画像を表示します。`cmap="gray"`は画像をグレースケールで表示することを意味します。
  - `plt.title()`: 各サブプロットのタイトルを設定します。
  - `plt.axis("off")`: 軸の表示をオフにします。
  - `plt.suptitle()`: 図全体のタイトルを設定します。
  - `plt.show()`: 図を表示します。

5.4. 損失履歴のプロット

学習の進捗を視覚的に確認するために、訓練損失と検証損失の履歴をプロットすることは非常に重要です。

`MNIST訓練実装答え.py`のメイン実行ブロックの最後で、`plt.plot()`を使って訓練損失と検証損失の履歴をグラフ化しています。

- **グラフから読み取れること:**
  - **損失の減少:** 訓練損失と検証損失がともに減少していれば、モデルが順調に学習していることを示します。
  - **過学習の兆候:** 訓練損失は減少し続けるが、検証損失が途中で増加に転じる場合、モデルが訓練データに過剰に適合し、未知のデータに対する汎化性能が低下している（過学習）可能性があります。
  - **学習の停滞:** 損失がほとんど変化しなくなった場合、学習が停滞している可能性があります。

ステップ5*：メイン実行ブロックとハイパーパラメータの調整

6.1. 全体の統合と実行フロー

`if __name__ == "__main__":` ブロックは、Pythonスクリプトが直接実行されたときにのみ実行されるコードを記述するための標準的な慣習です。このブロック内で、これまでに定義したすべてのコンポーネント（データセット、モデル、損失関数、最適化手法、学習関数、評価関数、可視化関数）が統合され、深層学習のパイプライン全体が実行されます。

実行フローの概要:

1. **ハイパーパラメータの設定:** モデルの挙動を制御する各種パラメータを定義します。
2. **データ準備:** `NumberDataset` と `DataLoader` を使って、訓練、検証、テスト用のデータを準備します。
3. **モデル、損失関数、最適化手法のインスタンス化:** 定義した `MyNet` クラスからモデルのインスタンスを作成し、損失関数と最適化手法を設定します。
4. **訓練前の予測可視化:** 学習前のモデルがどのように予測するかを確認します（通常はランダムな予測）。
5. **学習の実行:** `train_model` 関数を呼び出し、モデルを訓練します。
6. **訓練後の予測可視化:** 学習後のモデルがどのように予測するかを確認します。
7. **評価の実行:** `evaluate_model` 関数を呼び出し、テストデータでモデルの最終的な性能を評価します。
8. **結果の可視化:** 訓練損失と検証損失の履歴をプロットし、学習の進捗を確認します。

6.2. ハイパーパラメータの設定と影響

「ハイパーパラメータ」とは、モデルの学習プロセスを制御するために、学習開始前に手動で設定する必要がある値のことです。これらはモデルの性能に大きな影響を与えます。

- `INPUT_SIZE = 28 * 28`:
  - **意味:** モデルへの入力データの次元数。MNIST画像は28x28ピクセルなので、1次元に平坦化すると784次元になります。
  - **影響:** モデルの入力層のサイズを決定します。データと一致しないとエラーになります。
- `HIDDEN_SIZE = 128`:
  - **意味:** 隠れ層のニューロン数。モデルの「表現力」を決定します。
  - **影響:** 大きすぎると過学習しやすくなり、小さすぎると学習能力が不足する可能性があります。適切な値を見つけるには試行錯誤が必要です。
- `OUTPUT_SIZE = 10`:
  - **意味:** モデルの出力層の次元数。MNISTは0から9までの10クラス分類なので、10となります。

- **影響:** 分類問題の場合、分類するクラス数と一致させる必要があります。

- `LEARNING_RATE = 0.001`:
  - **意味:** 最適化手法がパラメータを更新する際の「歩幅」の大きさ。
  - **影響:** 大きすぎると損失が発散したり、最適解を飛び越えたりする可能性があります。小さすぎると学習が非常に遅くなります。最も調整が難しいハイパーパラメータの一つです。
- `BATCH_SIZE = 64`:
  - **意味:** 一度にモデルに入力するデータサンプルの数（ミニバッチのサイズ）。
  - **影響:** 大きすぎるとメモリ消費が増え、学習の汎化性能が低下する可能性があります。小さすぎると学習が不安定になることがあります。
- `EPOCHS = 5`:
  - **意味:** データセット全体を何回繰り返して学習させるか。
  - **影響:** 少なすぎるとモデルが十分に学習できません。多すぎると過学習を引き起こす可能性があります。

6.3. 再現性のためのシード設定

深層学習の学習プロセスには、パラメータの初期化やデータのシャッフルなど、
多くのランダムな要素が含まれます。そのため、何も設定しないと、同じコード
を実行しても毎回異なる結果になる可能性があります。

- `torch.Generator().manual_seed(42)`: PyTorchの乱数生成器のシードを固定します。
- **重要性:** シードを固定することで、乱数に依存する処理（例: データの分割、モデルの初期化）が毎回同じ結果を生成するようになります。これにより、実験の「再現性」が確保され、コードの変更やハイパーパラメータの調整がモデルの性能に与える影響を正確に評価できるようになります。

これで、`AI班/活動/7_2/MNIST訓練実装答え.py`を実装するための知識をゼロから体系的に取得するためのマニュアルが完成しました。