1　機械学習で学ぶ Python の基礎

Python の入門書は多くありますが, 数値計算やリストの扱いで飽きてしまう人が多く, 読み切れる人は少ないでしょう. そこで本稿では機械学習を題材にして, Python の基礎を学ぶことを目的とします. 具体的には, 機械学習の基本的な概念やアルゴリズムを学びながら, Python の文法やデータ構造を理解していきます. これにより, 最速で Python の基礎を習得し, 機械学習の実装に取り組むことができるようになります.

1.1　早速実践機械学習

機械学習とは、コンピュータがデータから学習し、予測や分類を行う技術です。

1.2　Module, Package, Library

Python では、機械学習のための多くのライブラリが用意されています。代表的なものには、NumPy、Pandas、Scikit-learn、TensorFlow、PyTorch などがあります。import numpy as np という表現は、NumPy というライブラリを np という名前でインポートすることを意味します。これにより、NumPy の機能を np という短い名前で使用できるようになります。すなわち, import module 名 as あだ名 という形で import するのですね. それではここで, module を実際に作ってみましょう. 例えば

module の基本的な構成がわかると from module 名 import 関数名 as あだ名という形も用意に理解できるようになります. これはつまり, from filename import function name as あだ名 が可能ということを意味します.

1.3　機械学習の種類

機械学習には、教師あり学習、教師なし学習、強化学習などの種類があります。

1.4　機械学習の応用

機械学習は、画像認識、自然言語処理、音声認識など、様々な分野で応用されています。

2　機械学習のアルゴリズム

機械学習のアルゴリズムには、回帰分析、決定木、ニューラルネットワークなどがあります。

2.1　回帰分析

回帰分析は、数値データの予測に使用される手法です

2.2　決定木

決定木は、データを分類するためのツリー構造のモデルです。

2.3　ニューラルネットワーク, CNN

ニューラルネットワークは、人間の脳の構造を模したモデルで、深層学習に使用されます。

付録 A　Git によるバージョン管理とプロジェクト構成

A.1　Git の基本概念

Git は分散型バージョン管理システムで、ソースコードの変更履歴を追跡・管理するためのツールです。機械学習プロジェクトでは、コードの変更履歴を管理し、実験結果を再現可能にするために重要な役割を果たします。

A.2　プロジェクト構造とモジュール化

大規模な機械学習プロジェクトでは、以下のような構造でコードを整理することが推奨されます：

```
MyProject/
├── src/                    # ソースコード
│   ├── models/             # モデル定義
│   ├── data/               # データ処理
│   ├── train/              # 学習スクリプト
│   ├── analysis/           # 分析ツール
│   └── visualization/      # 可視化
├── exefile/                # 実行ファイル
├── models/                 # 保存されたモデル
├── results/                # 実験結果
└── data/                   # データセット
```

A.3　Python のインポートシステム

異なるディレクトリのモジュールをインポートする際は、パスの設定が重要です：

```python
import sys
import os

# プロジェクトルートをパスに追加
sys.path.append(os.path.join(os.path.dirname(__file__), '..', 'src'))

# モジュールのインポート
from train.mn_cnn_train import train_model
from analysis.mn_cnn_lossf import analyze_loss_landscape
from visualization.mn_cnn_plots import plot_training_results
```

A.4　Git の基本操作

A.4.1　全ファイルの追加とコミット

プロジェクトの全ファイルを Git に追加する基本的な手順：

```
# 現在の状態を確認
git status

# 全ファイルを追加
git add .
# または
git add -A

# コミット
git commit -m "MNIST CNN プロジェクト初期実装"

# リモートリポジトリにプッシュ
git push
```

A.4.2　.gitignore ファイルの活用

GitHub にアップロードしたくないファイル（大きなデータセットやモデルファイル）は、.gitignore ファイルで除外します：

```
# Python関連
__pycache__/
*.pyc
*.pyo
.Python

# 機械学習関連
models/*.pt
models/*.pth
data/
results/
*.pkl

# IDE設定
.vscode/
.idea/

# OS関連
.DS_Store
Thumbs.db
```

A.5　改行コードの問題

異なる OS 間でプロジェクトを共有する際、改行コードの違いによる警告が表示されることがあります：

```
warning: in the working copy of 'file.py',
LF will be replaced by CRLF the next time Git touches it
```

この警告は以下の方法で対処できます：

```
# 改行コード自動変換を無効化
git config core.autocrlf false

# またはグローバル設定
git config --global core.autocrlf false
```

A.6　ローカル Git と GitHub の使い分け

- **ローカル Git**: 全ての変更履歴、大きなファイル、個人的な実験結果を管理
- **GitHub**: ソースコードのみを公開、データセットやモデルファイルは除外

この使い分けにより、コードの共有と個人データの保護を両立できます。

A.7　環境管理の将来展望

A.7.1　Docker による環境コンテナ化

長期的な開発では、Docker による環境のコンテナ化が重要になります：

```
# Dockerfile例
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "exefile/mn_cnn_main.py"]
```

A.7.2　段階的な環境管理戦略

1. **学習期**: Git + .gitignore による基本管理
2. **中級期**: Docker 導入による環境統一
3. **上級期**: Kubernetes + CI/CD による本格運用

この段階的アプローチにより、学習コストを最小化しながら本格的な機械学習システムへの発展が可能になります。

付録 B　Advanced PyTorch: torch.func による関数型プログラミング

B.1　torch.func とは

PyTorch 2.0 以降で導入された `torch.func` は、モデルのパラメータを直接変更することなく、関数的にモデルを操作できる革新的な機能です。

B.2　従来の方法の問題点

従来のパラメータ変更は危険な副作用を持ちます：

```
# 危険: モデルのパラメータを直接変更
for i, param in enumerate(model.parameters()):
    param.data = trained_params[i] + t * random_vector[i]

outputs = model(input)  # 変更されたパラメータで実行

# パラメータを手動で復元する必要がある
for i, param in enumerate(model.parameters()):
    param.data = trained_params[i]
```

B.3　torch.func による安全な実装

`functional_call` を使用すると、元のモデルを変更せずに異なるパラメータでモデルを実行できます：

```
import torch.func

def compute_loss_with_perturbation(model, t, random_vector,
                                   test_loader, criterion):
    # 1. 摂動パラメータを辞書形式で準備
    perturbed_params = {}
    for (name, param), rand_vec in zip(model.named_parameters(),
                                       random_vector):
        perturbed_params[name] = param + t * rand_vec

    # 2. 元のモデルは変更せず、一時的にパラメータを適用
    model.eval()
    total_loss = 0.0
    with torch.no_grad():
        for images, labels in test_loader:
            # functional_call: 非破壊的なモデル実行
            outputs = torch.func.functional_call(
                model, perturbed_params, images)
            loss = criterion(outputs, labels)
            total_loss += loss.item()
```

```
    return total_loss / len(test_loader)
```

B.4 実用例: 損失ランドスケープ解析

機械学習の最適化過程を視覚化する損失ランドスケープ解析での使用例：

```
def analyze_loss_landscape(model, test_loader, criterion):
    # ランダムな方向ベクトルを生成
    random_vector = [torch.randn_like(param)
                     for param in model.parameters()]

    # t ∈ [-0.05, 0.05] の範囲で損失を計算
    t_range = torch.linspace(-0.05, 0.05, 50)
    loss_values = []

    for t in t_range:
        loss = compute_loss_with_perturbation(
            model, t, random_vector, test_loader, criterion)
        loss_values.append(loss)

    return t_range, loss_values

# 使用例
model = trained_cnn_model
t_vals, losses = analyze_loss_landscape(model, test_loader, criterion)

# 結果のプロット
import matplotlib.pyplot as plt
plt.plot(t_vals.numpy(), losses)
plt.xlabel('Parameter Perturbation (t)')
plt.ylabel('Loss')
plt.title('Loss Landscape around Optimum')
plt.show()
```

B.5 torch.func の利点

1. **安全性**: 元のモデルパラメータが変更されない
2. **並列処理**: 複数のパラメータセットを同時に試験可能
3. **関数型**: 副作用のない純粋関数として動作
4. **デバッグ性**: パラメータの復元忘れによるバグを防止

この手法により、安全で効率的なモデル解析が可能になり、機械学習研究における新しいアプローチが開けます。

付録 C　高級 Python 機能の解説

C.1　zip 関数の活用

zip 関数は複数のイテラブルを同時に処理する強力な機能です：

```
# 基本的なzip
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]

for name, age in zip(names, ages):
    print(f"{name}は{age}歳です")
# 出力: Aliceは25歳です, Bobは30歳です, Charlieは35歳です

# torch.funcでの応用例
for (name, param), rand_vec in zip(model.named_parameters(), random_vector):
    perturbed_params[name] = param + t * rand_vec
```

C.1.1　zip の詳細動作

zip は以下のように動作します：

```
# zipの内部動作イメージ
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
zipped = zip(list1, list2)

# zipは次のタプルを順次生成
# (1, 'a'), (2, 'b'), (3, 'c')

# リストに変換して確認
print(list(zip(list1, list2)))
# [(1, 'a'), (2, 'b'), (3, 'c')]
```

C.2　辞書の作成と操作

C.2.1　辞書内包表記（Dictionary Comprehension）

波括弧{}を使った効率的な辞書作成：

```
# 基本的な辞書内包表記
squares = {x: x**2 for x in range(5)}
print(squares)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# 条件付き辞書内包表記
even_squares = {x: x**2 for x in range(10) if x % 2 == 0}
print(even_squares)  # {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

```
# torch.funcでの実際の使用例
trained_params = {name: param.data.clone()
                  for name, param in model.named_parameters()}
```

C.2.2 通常の辞書作成との比較

```
# 従来の方法（冗長）
trained_params = {}
for name, param in model.named_parameters():
    trained_params[name] = param.data.clone()

# 辞書内包表記（簡潔）
trained_params = {name: param.data.clone()
                  for name, param in model.named_parameters()}
```

C.3 ジェネレータ式とリスト内包表記

C.3.1 リスト内包表記

角括弧 [] を使った効率的なリスト作成：

```
# 基本的なリスト内包表記
squares = [x**2 for x in range(5)]
print(squares)  # [0, 1, 4, 9, 16]

# torch.funcでの使用例
random_vector = [torch.randn_like(param.data)
                 for param in model.parameters()]

# 条件付きリスト内包表記
large_params = [param for param in model.parameters()
                if param.numel() > 1000]
```

C.3.2 従来の方法との比較

```
# 従来の方法
random_vector = []
for param in model.parameters():
    random_vector.append(torch.randn_like(param.data))

# リスト内包表記（Pythonic）
random_vector = [torch.randn_like(param.data)
                 for param in model.parameters()]
```

C.4 enumerate 関数の活用

インデックスと値を同時に取得する便利な関数：

```
# 基本的なenumerate
items = ['apple', 'banana', 'cherry']
for i, item in enumerate(items):
    print(f"{i}: {item}")
# 出力: 0: apple, 1: banana, 2: cherry

# 機械学習での使用例（従来コード）
for i, param in enumerate(model.parameters()):
    param.data = trained_params[i] + t * random_vector[i]
```

C.5 実践的な組み合わせ例

C.5.1 複雑なデータ処理の例

これらの機能を組み合わせた実践例：

```
# データの前処理例
data = [('Alice', 85), ('Bob', 92), ('Charlie', 78)]

# 辞書内包表記 + 条件付きフィルタリング
high_scorers = {name: score for name, score in data if score >= 80}
print(high_scorers)  # {'Alice': 85, 'Bob': 92}

# リスト内包表記 + enumerate + zip
names = ['Alice', 'Bob', 'Charlie']
scores = [85, 92, 78]
indexed_data = [(i, name, score)
                for i, (name, score) in enumerate(zip(names, scores))]
print(indexed_data)
# [(0, 'Alice', 85), (1, 'Bob', 92), (2, 'Charlie', 78)]
```

C.5.2 torch.func コードの詳細解説

実際のコードを行ごとに解説：

```
# 1. 空の辞書を作成
perturbed_params = {}

# 2. zip関数で2つのイテラブルを同時処理
#    model.named_parameters(): (名前, パラメータ)のタプルを生成
#    random_vector: ランダムベクトルのリスト
for (name, param), rand_vec in zip(model.named_parameters(), random_vector):
    # 3. 辞書に新しいキー・値ペアを追加
    #    キー: パラメータの名前（文字列）
    #    値: 元のパラメータ + 摂動
    perturbed_params[name] = param + t * rand_vec

# 結果: {'conv1.weight': tensor(...), 'conv1.bias': tensor(...), ...}
```

C.6 Python らしいコードの書き方

これらの機能を使うことで、より **Pythonic** なコードが書けます：

- **可読性**: コードの意図が明確
- **簡潔性**: 少ない行数で同じ処理を実現
- **効率性**: C 言語レベルで最適化された内部処理
- **保守性**: バグが入りにくい構造

これらの高級機能をマスターすることで、より効率的で美しい機械学習コードが書けるようになります。