機械学習講習会

第五回：ニューラルネットワークの実装と訓練

2025/07/08

@Kobakos32

前回までの復習

1. **ニューラルネットワークの基本構造**：線形層と活性化関数の組み合わせ
2. **誤差逆伝播法**：効率的な勾配計算手法
3. **勾配降下法**：パラメータ最適化の基本手法

今回は理論を実装に落とし込み、実際にニューラルネットワークを訓練してみます。

本日のゴール

1. **バッチ学習**と**確率的勾配降下法**の理解
2. **モデルの検証・評価**と**データセット分割**を理解する
3. **深層学習の構成要素**を整理する
4. **PyTorch**を用いた実装方法を学ぶ
5. **MNIST手書き数字認識**を実装・訓練する
6. **モデルの評価と改善**テクニックを学ぶ

もくじ

1. バッチ学習と確率的勾配降下法

学習のゴールと素朴な方法

ゴール ：データセット全体での損失 $L(\theta)$ を最小化するパラメータ θ を見つける

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} L_i(\theta)$$

素朴な方法 ：データセット全体の勾配 $\nabla L(\theta)$ を正確に計算してパラメータを更新

$$\nabla L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla L_i(\theta)$$

問題点 ：

- **メモリ不足**：巨大なデータセットはメモリに収まらない
- **計算時間**：1回の更新に全データの計算が必要で非常に遅い

解決策：データの一部だけを使う

アイデア：毎回、データ全体ではなく、ランダムに選んだ一部（**ミニバッチ**）を使って勾配を「推定」する

ミニバッチ学習の進め方：

1. データからミニバッチをランダムに抽出
2. そのミニバッチで勾配を計算
3. パラメータを更新
4. 違うミニバッチで1-3を繰り返す

この手法を **確率的勾配降下法（Stochastic Gradient Descent, SGD）** と呼びます。

確率的勾配降下法（SGD）

ミニバッチSGDの更新式 ：

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{|B|} \sum_{i \in B} \frac{\partial L_i}{\partial \theta}$$

- B：ランダムに選ばれたミニバッチ
- $|B|$：**バッチサイズ**

用語の整理 ：

- **SGD**：本来は $|B| = 1$（データ1つ）を指したが、最近ではミニバッチを使う方法全般を指すことが多い
- **ミニバッチSGD**：$1 < |B| < N$ の場合。SGDと同義で使われる

なぜ「確率的」なのか？

ミニバッチから計算した勾配は、全データから計算した「真の勾配」と完全に一致はしません。
しかし、その **期待値は真の勾配と一致** します。

$$\mathbb{E}\left[\nabla L_i(\theta)\right] = \frac{1}{N}\sum_{i=1}^{N} \nabla L_i(\theta) = \nabla L(\theta)$$

- **直感的な意味** ：ミニバッチの勾配は毎回「ブレ」ますが、平均すれば真の勾配の方向を向いている
- この「ブレ」（ノイズ）が、学習の停滞（局所解）を防ぐ効果も期待できます
- 毎回ランダムにバッチを選ぶため、勾配の計算が **確率的** になります

訓練のサイクル：エポックとバッチ

エポック (Epoch) ：訓練データ全体を1周見ること

1. **シャッフル** ：1エポックの開始時に、訓練データをランダムにかき混ぜる
   - `[1, 2, ..., 100] -> [28, 12, 55, ...]`
2. **バッチ分割** ：シャッフルしたデータを、バッチサイズの固まりに分割
   - `| 28, 12 | 55, 23 | ... |`
3. **イテレーション** ：バッチを1つずつモデルに投入し、パラメータ更新を繰り返す

例 ：データ10,000個、バッチサイズ100なら

- 1エポック = 100イテレーション（= 100回のパラメータ更新）

バッチサイズの決め方

バッチサイズは重要なチューニング項目。典型的には32, 64, 128, 256など。

小さいバッチ (例: 4)

- ⭕ メモリ消費が少ない
- ⭕ 更新回数が多く、学習の進みが速いことがある
- ⭕ ノイズがパラメーターの停滞を防ぐことも
- ❌ 勾配のブレが大きく、学習が不安定になりやすい

大きいバッチ (例: 256)

- ⭕ 勾配が安定し、学習も安定
- ⭕ 計算効率が良い（並列化）
- ❌ メモリ消費が大きい
- ❌ 更新回数が少なく、収束に時間がかかることも
- ❌ 鋭い谷（汎化性能の低い解）に陥りやすいとの指摘も

2. モデルの検証・評価

モデル改善のサイクル

問題：モデルを反復的に改善するには、性能を測定する必要がある

しかし：最終的なテストデータを何度も使って評価すると...

- そのテストデータについてのみうまくいくモデルかもしれない
- たまたまテストデータでうまくいっただけかも
- 未知データへの汎化性能が過大評価される

機械学習の目的は未知データに対してうまく予測ができること！（汎化）

解決策：データを適切に分割して、改善サイクルを回す

試験のアナロジー

模試・過去問演習：

- 何度も受けて弱点を発見
- 勉強法を調整・改善
- 本番前の実力確認

本試（本番試験）：

- 一度だけの真剣勝負
- 真の実力を測定
- 結果で合否が決まる

検証セット（Validation Set）：

- 何度も使って性能確認
- ハイパーパラメータ調整
- 訓練中の監視

テストセット（Test Set）：

- 一度だけの最終評価
- 真の汎化性能を測定
- 最終的な性能報告

データセット分割

訓練セット（Training Set）：

- モデルのパラメータ学習に使用

検証セット（Validation Set）：

- モデルの改善をするときの指標として使用
- 訓練中のモデル性能監視

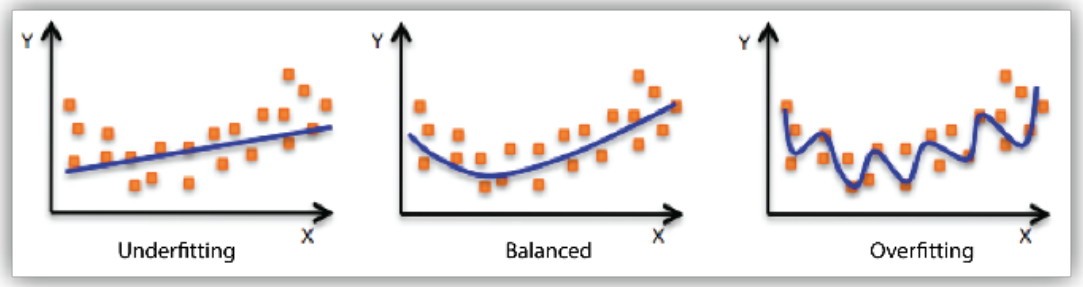
テストセット（Test Set）：

- 最終的な性能評価に使用
- 基本的に**一度だけ使用**

過学習（Overfitting）

過学習とは：

- 訓練データに過度に適合
- 新しいデータに対する性能が低下
- 「暗記」してしまった状態



左が学習がうまくいってない状態
真ん中がちょうどよいところ
右が過学習

検出方法：

- 訓練誤差は減少
- 検証誤差は増加
- 両者の差が拡大

検証セットの重要性：未知のデータに対応できるか確認できる

実践的な改善サイクル

1. **訓練セットでモデルを学習**
2. **検証セットで性能を評価**
3. **結果に基づいてモデル・訓練の仕方を調整**
4. **1-3を繰り返し**
5. **最後にテストセットで最終評価**

重要：テストセットは「模試」ではなく「本試」！

3. PyTorchによる実装

PyTorchとは

PyTorch ：Meta（旧Facebook）が開発した深層学習フレームワーク

主要な機能：

- **自動微分（Autograd）**：誤差逆伝播を自動計算
- **GPUを使った高速化**：大規模計算の高速化
- **いろいろな層が実装済み**：記述が省略できる
- **柔軟なモデル構築**：Pythonのクラスとして直感的にモデルを定義可能

深層学習の実装：4つの構成要素

これから、PyTorchを使って深層学習の4つの主要な部品を組み立てていきます。

1. **データセット**：モデルに与えるデータを用意する
2. **モデル**：ニューラルネットワークの構造を定義する
3. **損失関数**：モデルの予測の「悪さ」を測る
4. **オプティマイザ**：モデルのパラメータを更新する

前準備：Colab Notebook の設定

Google Colaboratory は、ブラウザ上でPythonコードを実行できる無料の環境です。
特に、深層学習で不可欠な **GPUを無料で利用できる** 点が大きなメリットです。

GPUを有効にする手順：

1. 配布したノートブックを開く
2. Drive にコピーを保存を選択（編集できるようになります）
3. Colabノートブックの上部メニューから **「ランタイム」** をクリックします。
4. ドロップダウンメニューから **「ランタイムのタイプを変更」** を選択します。
5. 表示されたポップアップウィンドウで、「ハードウェア アクセラレータ」のドロップダウンから **「GPU」** を選択し、「保存」をクリックします。

GPUが利用可能か確認するコード：

```
import torch
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')
# 期待される出力: Using device: cuda
```

`cuda` と表示されれば成功です。

1. データセットの準備

役割 ：訓練データを管理し、モデルにミニバッチとして供給します。

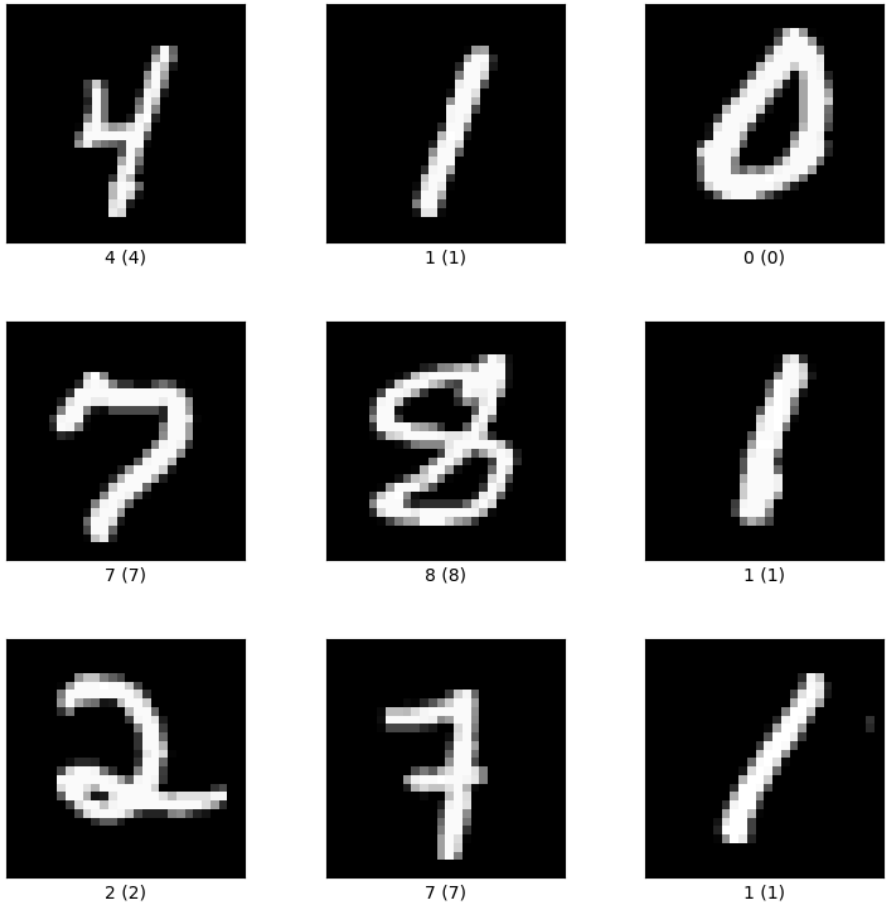
- **Dataset** ：データとラベルをひとまとめに持つオブジェクト。PyTorchでは `torch.utils.data.Dataset` を使います。
- **DataLoader** ： `Dataset` からデータを効率的に引き出し、シャッフルしたりバッチにまとめたりする便利なコンポーネントです。

1. データセットの準備

MNISTデータセット

手書き文字認識のデータセット

28*28ピクセルの白黒画像から数字を分類する



1. データセットの準備

ふつうは `torch.utils.data.Dataset` を継承して、データの読み込み方を自分で定義します。

`Dataset` クラスに用意すべきこと：

- `__len__` メソッド：データセットの長さを定義します
- `__getitem__` メソッド：`0` からデータセットの長さ未満の数字を受け取ってサンプルを返す

しかし、MNISTのような有名なデータセットは `torchvision` に便利なクラスが用意されているため、今回はそれを使います。

MNIST データセットの定義

```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
# 1. 画像データをどう変換するかのルールを定義
transform = transforms.Compose([
    transforms.ToTensor(), # データをテンソルに変換
    transforms.Normalize((0.5,), (0.5,)) # データを[-1, 1]の範囲に正規化
])
# 2. MNISTデータセットを読み込み、変換ルールを適用
train_dataset = datasets.MNIST(
    root='./data',          # データ保存先
    train=True,             # 訓練データを使用
    download=True,          # なければダウンロード
    transform=transform     # 上で定義した変換を適用
)
# 3. データをバッチにまとめて供給するDataLoaderを作成
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

これで、訓練ループの中でミニバッチを簡単に取り出せるようになりました。

データセットの確認

```
print(len(train_dataset))
print(len(train_dataset[0]))
print(train_dataset[0])
```

出力：

```
60000
2
(tensor([[[-1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
           -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
...

           -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
           -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000]]]),
 5)
```

タプルの最初の要素が画像データ（1,28,28）、2番目の要素がラベル（数字）です。

2. モデルの定義（コンセプト）

役割 ：入力データを受け取り、予測を出力する計算グラフ（ニューラルネットワーク）を定義します。

実装 ：PyTorchでは、`torch.nn.Module` を継承したクラスとしてモデルを定義するのが標準的です。

- **`__init__(self)`**
  - モデルで使う層（全結合層、畳み込み層など）を部品として初期化・準備する場所。
- **`forward(self, x)`**
  - データ `x` が `__init__` で用意した部品（層）をどのように流れていくか、その計算の流れ（順伝播）を定義する場所。

逆伝播は自動でやってくれる！！

2. モデルの定義（実装）

`__init__` で層を定義し、`forward` でそれらを繋ぎ合わせます。

```
import torch.nn as nn
import torch.nn.functional as F

class MNISTClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(28 * 28, 128)  # 784 -> 128
        self.layer2 = nn.Linear(128, 64)       # 128 -> 64
        self.layer3 = nn.Linear(64, 10)        # 64 -> 10 (クラス数)
    def forward(self, x):
        x = x.view(-1, 28 * 28)# 1. 入力画像を1次元のベクトルに平坦化
        x = F.relu(self.layer1(x))# 2. layer1に通し、活性化関数ReLUを適用
        x = F.relu(self.layer2(x))# 3. layer2に通し、活性化関数ReLUを適用
        x = self.layer3(x)# 4. 出力層に通す（活性化関数は不要）
        return x
```

2. モデルの確認

```
model = MNISTClassifier().to(device)
print(model)
print(model(torch.randn(4, 1, 28, 28).to(device)).shape)
```

出力：

```
MNISTClassifier(
  (layer1): Linear(in_features=784, out_features=128, bias=True)
  (layer2): Linear(in_features=128, out_features=64, bias=True)
  (layer3): Linear(in_features=64, out_features=10, bias=True)
)
torch.Size([4, 10])
```

3. 損失関数

役割 ：モデルの出力と正解ラベルを比較し、その「隔たり」を数値（損失）として計算します。

実装 ：今回は多クラス分類なので、一般的に使われる **交差エントロピー損失** を利用します。

```
import torch.nn as nn

# 損失関数のインスタンスを作成
criterion = nn.CrossEntropyLoss()
```

`nn.CrossEntropyLoss` は、内部でソフトマックス関数の計算と損失の計算を両方行ってくれるため、モデルの出力層に活性化関数を適用する必要はありません。

3. 損失関数の確認

```
# モデルの出力をランダムに生成
output = torch.randn(4, 10).to(device)  # バッチサイズ4、クラス数10
# 正解ラベルをランダムに生成
target = torch.randint(0, 10, (4,)).to(device)  # バッチサイズ4、クラス数10
print("Output:", output)
print("Target:", target)
# 損失を計算
loss = criterion(output, target)
```

出力例：

```
Output: tensor([[ 0.1012, -1.2391,  0.4854, -0.1074, -1.4884,  0.6156, -0.6801, -0.7512,
          0.4496,  0.5602],
        [-0.8033, -0.8500, -0.6183,  1.7212,  0.4777, -0.0310, -1.0744,  0.8239,
          1.5189, -0.2671]])
Target: tensor([2, 2])
tensor(2.6644)
```

4. オプティマイザ

役割 ：計算された損失（と勾配）にもとづいて、モデルのパラメータを少しずつ更新し、性能を改善します。

実装 ：いろいろな種類がありますが、まずはSGDを試しましょう。

```
import torch.optim as optim

# モデルの全パラメータを更新対象として、SGDオプティマイザを作成
# lrは学習率（learning rate）
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

これで訓練の準備が整いました！

基本的な訓練ループ

```
def train(model, device, train_loader, optimizer, criterion, epoch):
    model.train()  # 訓練モードに設定
    total_loss = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        # 勾配の初期化
        optimizer.zero_grad()

        # 順伝播
        output = model(data)
        loss = criterion(output, target)

        # 逆伝播
        loss.backward()

        # パラメータ更新
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(train_loader)
```

モデル評価の実装

```
def evaluate(model, device, test_loader, criterion):
    model.eval()  # 評価モードに設定
    test_loss = 0
    correct = 0

    with torch.no_grad():  # 勾配計算を無効化
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item()

            # 予測値の計算
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader)
    accuracy = 100. * correct / len(test_loader.dataset)

    return test_loss, accuracy
```

完全な訓練スクリプト

```
# デバイスの設定（CPUがGPUか）
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# モデル、損失関数、オプティマイザの初期化
model = MNISTClassifier().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)

# 訓練ループ
for epoch in range(1, 11):
    train_loss = train(model, device, train_loader, optimizer, criterion, epoch)
    test_loss, accuracy = evaluate(model, device, test_loader, criterion)

    print(f'Epoch {epoch}: Train Loss: {train_loss:.4f}, '
          f'Test Loss: {test_loss:.4f}, Accuracy: {accuracy:.2f}%')
```

重要な設定

model.train() vs model.eval()：

- `model.train()`：訓練モード（訓練時のみ有効になる設定を有効に）
- `model.eval()`：評価モード

torch.no_grad()：

- 勾配計算を無効化
- 逆伝播をしないことでメモリ使用量を削減
- 評価時に使用

optimizer.zero_grad()：

- 勾配の初期化
- PyTorchでは勾配が累積されるため必須

実際の実行結果例

```
Epoch 1: Train Loss: 0.3913, Test Loss: 0.2326, Accuracy: 92.98%
Epoch 2: Train Loss: 0.1888, Test Loss: 0.1635, Accuracy: 94.74%
...
Epoch 9: Train Loss: 0.0591, Test Loss: 0.0780, Accuracy: 97.83%
Epoch 10: Train Loss: 0.0561, Test Loss: 0.0782, Accuracy: 97.66%
```

期待される結果：

- 訓練損失は徐々に減少
- テスト精度は97-98%程度

4. モデルの評価と改善

学習曲線の可視化

```
import matplotlib.pyplot as plt

def plot_learning_curves(train_losses, test_losses, accuracies):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

    # 損失の推移
    ax1.plot(train_losses, label='Train Loss')
    ax1.plot(test_losses, label='Test Loss')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.legend()
    ax1.set_title('Loss Curves')

    # 精度の推移
    ax2.plot(accuracies, label='Test Accuracy')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy (%)')
    ax2.legend()
    ax2.set_title('Accuracy Curve')

    plt.tight_layout()
    plt.show()
```

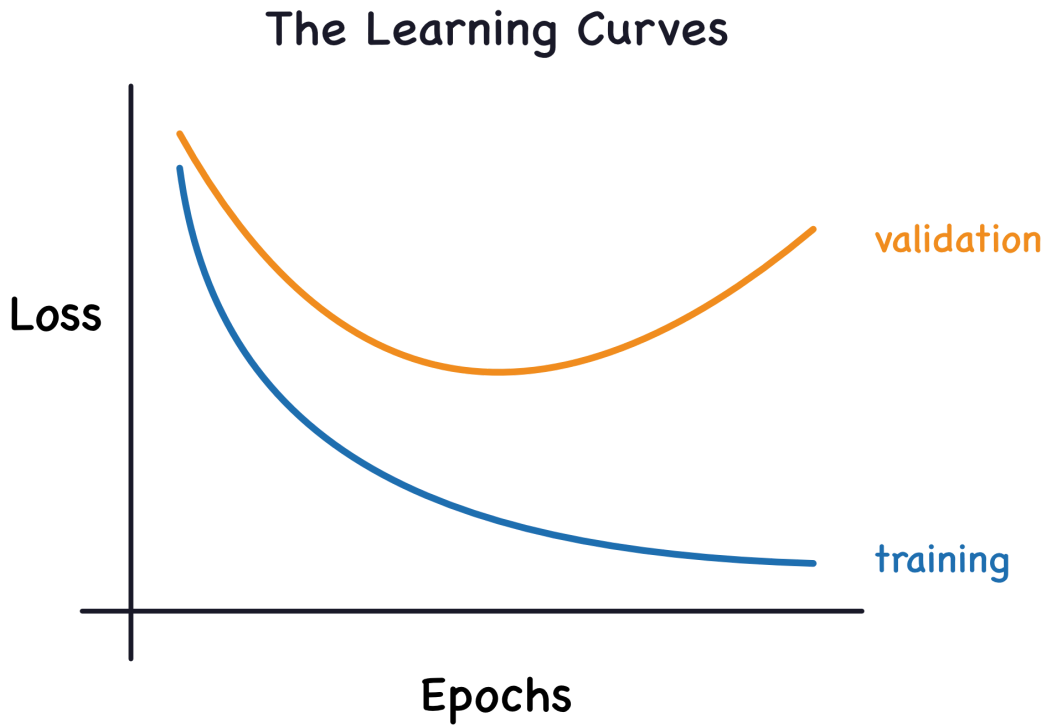
過学習の検出

過学習の兆候：

- 訓練誤差は減少するが検証誤差が増加
- 訓練精度と検証精度の差が拡大

対策：

- ドロップアウトの追加
- 正則化の強化
- データ拡張
- 早期停止（Early Stopping）



https://www.kaggle.com/code/ryanholbrook/overfitting-and-underfitting

モデルの保存と読み込み

```
# モデルの保存
torch.save(model.state_dict(), 'mnist_model.pth')

# モデルの読み込み
model = MNISTClassifier()
model.load_state_dict(torch.load('mnist_model.pth'))
model.eval()
```

実際の予測例

```
def predict_single_image(model, image):
    model.eval()
    with torch.no_grad():
        image = image.unsqueeze(0)  # バッチ次元を追加
        output = model(image)
        prediction = output.argmax(dim=1)
        confidence = F.softmax(output, dim=1).max()
        return prediction.item(), confidence.item()

# 使用例
prediction, confidence = predict_single_image(model, test_image)
print(f'予測: {prediction}, 信頼度: {confidence:.2f}')
```

ハイパーパラメータの調整

主要なハイパーパラメータ：

- 学習率（Learning Rate）
- バッチサイズ（Batch Size）
- 隠れ層のサイズ・数
- ドロップアウト率

5. その他のテクニック・ポイント

デバッグのコツ

小さなデータセットで検証：

```
# 小さなサブセットで動作確認
small_dataset = torch.utils.data.Subset(train_dataset, range(100))
small_loader = DataLoader(small_dataset, batch_size=32)
```

確認のために毎回大きなデータセットで回していては時間がかかる

モデルの出力確認：

```
# モデルの出力形状を確認
with torch.no_grad():
    sample_output = model(torch.randn(1, 1, 28, 28))
    print(f'Output shape: {sample_output.shape}')
```

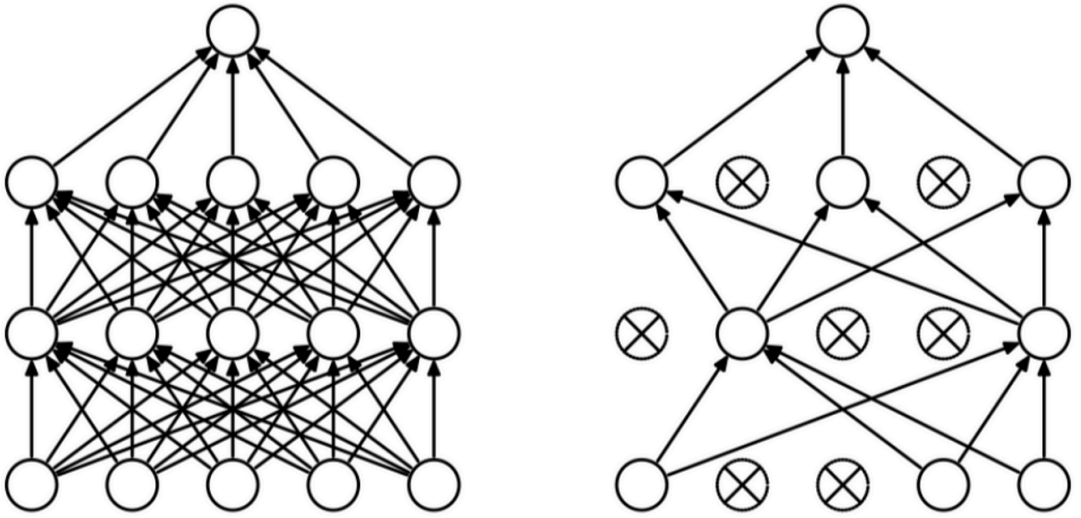
出力・入力の形があっているかは非常に大事、ほとんどのエラーは形が原因（ソース：私）

過学習防止テクニック

ドロップアウト層（Dropout）：訓練時に一定確率でニューロンをランダムに「無効化」

直感的な理解：

- 人間の学習に例えると「部分的な情報だけで判断する練習」
- 特定のニューロンに依存しすぎることを防ぐ



(a) Standard Neural Net　(b) After applying dropout.

なぜ効果的か：

- **過学習の防止**：特定のニューロンへの過度な依存を防ぐ
- **ロバストネス向上**：一部のニューロンが欠けても機能する

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

実装

```
class MyModel(torch.nn.Module):
    def __init__(self, ...):
        ...
        self.dropout = nn.Dropout(p=0.2)  # 20%の確率でニューロンを無効化
    def forward(self, x):
        ...
        x = F.relu(self.layer1(x))
        x = self.dropout(x)
        ...
```

実際には訓練時にのみ無効化が行われ、推論時にはすべてのニューロンを使う（`model.train()`, `model.eval()` で切り替え）

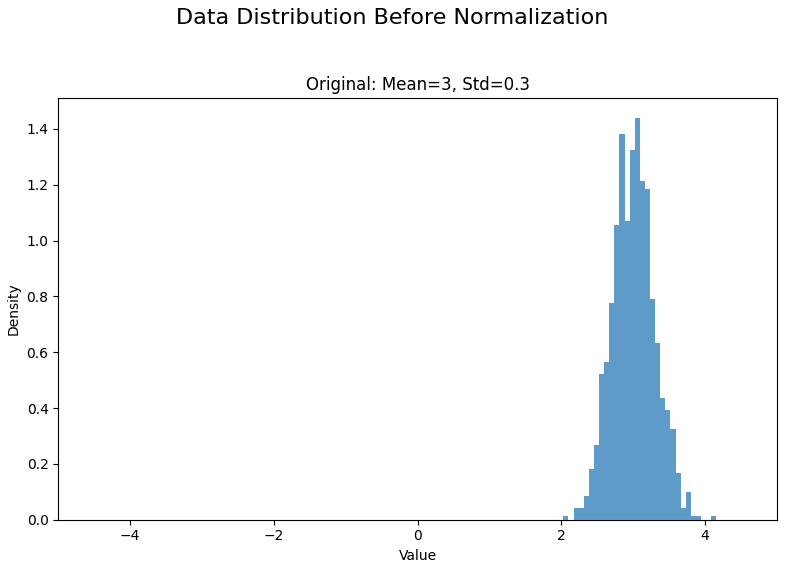
バッチ正規化（BatchNorm）：各層の入力を平均0、分散1に正規化

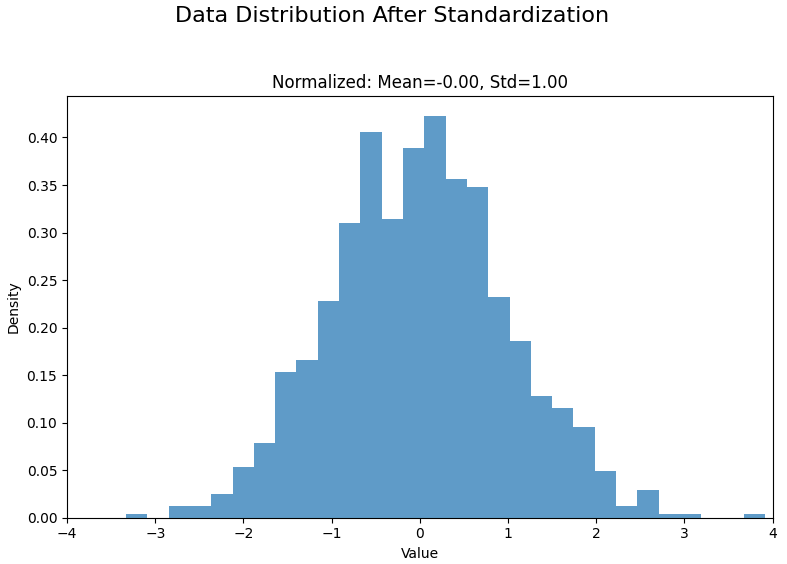
問題の背景：

- 深いネットワークは入力に何回も線形変換 + 活性化を適用
- 変換を重ねて極端な値が出てしまうと学習がしにくい

なぜ効果的か：

- **学習の安定化**：各層の入力分布を一定に保つ
- **高速学習**：より大きな学習率を使用可能





実装

```
class MyModel(torch.nn.Module):
    def __init__(self, ...):
        ...
        self.bn = nn.BatchNorm1d()
    def forward(self, x):
        ...
        x = F.relu(self.layer1(x))
        x = self.bn(x)
        ...
```

6. まとめ

本日のまとめ

1. **バッチ学習**：効率的な訓練のためのデータ分割手法
2. **深層学習の構成要素**：データセット、モデル、損失関数、オプティマイザ、評価関数
3. **PyTorch実装**：モジュール化されたニューラルネットワークの実装
4. **訓練ループ**：順伝播、逆伝播、パラメータ更新の流れ
5. **評価と改善**：学習曲線の可視化、過学習の検出、ハイパーパラメータ調整

実践的な学習のコツ

段階的な学習：

1. 小さなデータセットで動作確認
2. 基本的なモデルから始める
3. 徐々に複雑なモデルに発展
4. 性能向上のテクニックを追加

継続的な改善：

- 異なるアーキテクチャの試行
- ハイパーパラメータの最適化
- いろいろな学習テクニックを試す

次回予告

第六回：部内コンペキックオフ

部内コンペティションのキックオフを行います！
スターターノートブックを解説して、ファーストサブをするとこまでやります。

チームメートと連絡をとっておこう

スムーズに情報・コード共有ができるように早めにしておきましょう。