ニューラルネットワークの学習

本章の目的は、PyTorchを使ったニューラルネットワークを学習させるための基本的な手順について理解することです。

PyTorchではデータの読み込みに`DataLoader`モジュールを使うことが一般的です。`DataLoader`はバッチ処理やデータのシャッフル機能を提供しているので、データの前処理などを簡潔に書くことができます。ここでは、`DataLoader`を利用せず、データセットはNumpy形式で自作します。`DataLoader`を利用する方法は次の章で説明します。

学習プログラム

まず学習プログラムの全体構成を示します。ディープラーニングの学習プログラムはこの形式が基本形になります。

```
（データの準備処理）

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        ネットワークの各レイヤの定義
    def forward(self, …):
        順伝搬処理の定義

model = MyModel()
optimizer = 最適化アルゴリズム
criterion = 誤差関数

for epoch in range(エポック数):
    output = model(学習データ)
    loss = criterion(output, 学習データに対応するラベル)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), '学習したモデルを保存するパス')
```

最初のブロックはデータの準備です。複数の学習データと、識別の場合は学習データと同じ数のラベルデータを、回帰の場合は目標ベクトルデータを準備します。

次のブロックは学習するネットワークを定義する部分です。ネットワークは`nn.Module`を検証したクラスで記述し、クラス名の`MyModel`は任意の名前で問題ありません。このクラスは必ず`__init__`と順伝搬処理を記述する`foward`の関数をオーバーライドする必要があります。他に関数を追加することも可能です。

次のブロックは、モデルと最適化アルゴリズムを定義する部分です。

次のブロックは学習の部分です。foward関数からモデルの出力値を求め、損失関数に入力して損失値を得ます。次に勾配を初期化して、計算した損失値から購買を求め、最後に指定した最適化アルゴリズムの方式で

パラメータを更新します。

最後のブロックは学習結果のモデルを保存する部分です。ここでテストを行ったりもします。

学習の全体像を把握したところで、実際に学習プログラムを作っていきます。

ライブラリの読み込み

以下のライブラリをインポートしておきます。

```
import numpy as np
from sklearn import datasets
import torch
import torch.nn as nn
import torch.optim as optim
```

学習データの準備

ここでは、scikit-learnのirisでデータセットを用います。scikit-learnをインストールし、以下のようなデータセットを取得する関数をつくります。scikit-learnのデータセットはNumpy形式になっているので、`torch.from_numpy()`を使ってTensor形式に変換します。

```
def load_dataset():
    '''irisデータセットを読み込んで、学習・検証データに分割して返す'''
    # irisデータセットを読み込む
    iris = datasets.load_iris()
    x = iris.data.astype(np.float32)
    y = iris.target.astype(np.int64)

    # 偶数番目を学習用データセットとして取得
    train_x = torch.from_numpy(x[::2])
    train_y = torch.from_numpy(y[::2])

    # 奇数番目を検証用データセットとして取得
    test_x = torch.from_numpy(x[1::2])
    test_y = torch.from_numpy(y[1::2])

    return train_x, train_y, test_x, test_y

train_x, train_y, test_x, test_y = load_dataset()
```

ネットワークの定義

irisデータセットはあやめの4つの数値（花びらの長さ、幅、がく片の長さ、幅）から3種類のあやめの種類（setosa、versicolor、virginica）のどれかを推定する問題のデータセットです。ネットワークの構造もその問題に合わせ、入力層が4ユニット、出力層が3ユニットとなるネットワークをつくります。活性化関数はシグモイド関数とします。中間層はいくつの層でもいいですが、ここでは6つの層を置きます。それぞれの層は

全結合されているものとします。ニューラルネットワークではバイアスユニットがありますが、PyTorchではネットワークの定義に記述する必要はなく、自動的に作られます。

このネットワークをプログラムで記述します。

```
class Net(nn.Module):
    '''irisの4データからirisの3種類のいずれかに分類するネットワーク'''
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(4, 6)
        self.fc2 = nn.Linear(6, 3)

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = self.fc2(x)
        return x
```

モデルの生成・最適化アルゴリズムと損失関数の設定

まず、先程作成したネットワークのインスタンスをつくります。

```
model = Net()
```

次に最適化アルゴリズムを設定します。PyTorchは数種類の最適化アルゴリズムを提供しており、そこから1つ選択します。どの最適化アルゴリズムがあるかは、リファレンスを参照してください。

torch.optim - Pytorch

ここでは、Adamを使います。lrは学習率を示します。どのパラメータがあるかは最適化アルゴリズムによるので、都度リファレンスを調べてください。また、どのような値を設定すればよいかも、何度か学習して試行錯誤するしかありません。ここでは`0.1`と設定しました。

```
optimizer = optim.Adam(model.parameters(), lr=0.1)
```

次に損失関数を設定します。今回のような識別問題では、一般的にクロスエントロピー`nn.CrossEntropyLoss()`を使います。回帰問題の場合には、二乗誤差`nn.MSELoss()`を使います。

```
criterion = nn.CrossEntropyLoss()
```

学習

次に学習を行います。これは、勾配降下法の手順そのものです。

```
for epoch in range(2000):
    output = model(train_x)
    loss = criterion(output, train_y)
    print(epoch, loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

学習したモデルの出力

学習したモデルは、`torch.save()`で保存することができます。ここでは`my_iris_model`というファイル名で保存しました。

```
torch.save(model.state_dict(), 'my_iris_model')
```

検証プログラム

学習したモデルに検証データを入力することで、検証データに対してモデルがどのラベルに判定したかを検証することができます。

学習したモデルの読み込み

保存したモデルは`torch.load()`で読み込むことができます。

```
model = Net()
model.load_state_dict(torch.load('my_iris_model'))
```

検証

検証は検証データをモデルに入力して順伝搬処理を行わせるだけです。ただし、学習と違って検証では微分値を求める必要がないので、`model.eval()`と`torch.no_grad()`を呼び出します。実際の検証プログラムは以下です。

```
model.eval()
with torch.no_grad():
    output = model(test_x)
    ans = torch.argmax(output, 1)
    print(((test_y == ans).sum().float() / len(ans)).item())
```

全体を通したプログラムは以下を参照してください。

iris.py

train.py

test.py

ミニバッチ

上記の学習では、1回のパラメータ更新ですべてのデータを使いました。つまりバッチ学習です。今回は1回のパラメータ更新でランダムに選択した25個のデータを使う手法も確認します。これはミニバッチ学習という手法になります。

DataLoaderを使わない場合、np.random.permutation()を使ってランダムな順番のインデックス情報を生成し、バッチサイズで学習データをスライスすることで実装します。以下に学習プログラムのうちミニバッチのために修正したブロックのみ示します。

```
# バッチサイズの設定
data_size = len(train_x)
batch_size = 30

# 学習
for epoch in range(2000):
    idx = np.random.permutation(data_size)
    for batch in range(0, data_size, batch_size):
        train_x_batch = train_x[idx[batch:batch+batch_size]]
        train_y_batch = train_y[idx[batch:batch+batch_size]]
        output = model(train_x_batch)
        loss = criterion(output, train_y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

GPUの利用

PyTorchでGPUを使うためには、CUDAをインストールする必要があります。PyTorchでGPUを使う方法は、データやモデルを.to('cuda:0')によりGPUに移動させるだけです。'cuda:0'はマシンに搭載されている1枚目のGPUを指します。GPUが搭載されいればGPUを、搭載されていなければCPUを使うといった使いわけがしたい場合は、以下のようにします。

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
train_x, train_y = train_x.to(device), train_y.to(device)

model = Net().to(device)
```

目次へ戻る