Implementacja i analiza pętli uczących w bibliotece PyTorch

Sprawozdanie Techniczne

7 lutego 2026

Streszczenie

Niniejsze sprawozdanie przedstawia proces implementacji pętli treningowych dla głębokich sieci neuronowych z wykorzystaniem biblioteki PyTorch. Omówiono kluczowe hiperparametry wpływające na proces uczenia, zaprezentowano listingi kodu dla podstawowej oraz zaawansowanej pętli walidacyjnej, a także przedstawiono metodykę wizualizacji przebiegu treningu (funkcja straty i dokładność).

Spis treści

1 Wstęp

Proces uczenia sieci neuronowej jest iteracyjną optymalizacją wag modelu w celu minimalizacji funkcji kosztu. W bibliotece PyTorch, w przeciwieństwie do wysokopoziomowych bibliotek jak Keras, pętla ucząca (ang. *training loop*) musi zostać zdefiniowana jawnie. Daje to programiście pełną kontrolę nad przepływem gradientów, momentem aktualizacji wag oraz logowaniem metryk.

2 Kluczowe hiperparametry uczenia

Efektywność pętli uczącej zależy od właściwego doboru hiperparametrów. Poniżej przedstawiono najważniejsze z nich:

2.1 Współczynnik uczenia (Learning Rate - α)

Jeden z najważniejszych parametrów, określający jak duże kroki podejmuje optymalizator w kierunku minimum funkcji straty.

- **Zbyt mały:** Sieć uczy się bardzo wolno, ryzyko utknięcia w minimach lokalnych.
- **Zbyt duży:** Proces uczenia może być niestabilny, funkcja straty może oscylować lub dywergować.

2.2 Liczba epok (Epochs)

Epoka oznacza jednokrotne przejście całego zbioru treningowego przez sieć (ang. *forward pass*) i aktualizację wag (ang. *backward pass*). Liczba epok determinuje, jak długo sieć będzie się uczyć.

2.3 Rozmiar wsadu (Batch Size)

Określa liczbę próbek treningowych przetwarzanych jednocześnie przed jedną aktualizacją wag.

- **Batch Gradient Descent:** Cały zbiór danych na raz (stabilne, ale wymaga dużo pamięci).
- **Mini-batch Gradient Descent:** Np. 32, 64, 128 próbek (kompromis między stabilnością a szybkością).
- **Stochastic Gradient Descent (SGD):** Jedna próbka na raz (duży szum, szybkie iteracje).

3 Implementacja pętli uczących

3.1 Przygotowanie środowiska

Przed uruchomieniem pętli, zakładamy istnienie obiektu modelu (`model`), ładowarki danych (`dataloader`), funkcji straty (`criterion`) oraz optymalizatora (`optimizer`).

3.2 Listing 1: Podstawowa pętla ucząca

Poniższy kod przedstawia minimalistyczną wersję pętli uczącej, wykonującą jedynie propagację wsteczną na zbiorze treningowym.

```python
import torch

def train_simple(model, dataloader, criterion, optimizer, epochs=10):
    model.train() # Przelaczenie modelu w tryb treningowy

    for epoch in range(epochs):
        running_loss = 0.0

        for inputs, labels in dataloader:
            # Przeniesienie danych na GPU jesli dostepne
            device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
            inputs, labels = inputs.to(device), labels.to(device)
            model.to(device)

            # 1. Zerowanie gradientow
            optimizer.zero_grad()

            # 2. Forward pass (predykcja)
            outputs = model(inputs)

            # 3. Obliczenie straty
            loss = criterion(outputs, labels)

            # 4. Backward pass (propagacja wsteczna)
            loss.backward()

            # 5. Aktualizacja wag
            optimizer.step()

            running_loss += loss.item()

        print(f"Epoka [{epoch+1}/{epochs}], Strata: {running_loss/len(dataloader):.4f}")
```

Listing 1: Podstawowa pętla treningowa

3.3 Listing 2: Zaawansowana pętla z walidacją

W praktycznych zastosowaniach konieczne jest monitorowanie metryk na zbiorze walidacyjnym, aby zapobiec przeuczeniu (overfitting). Funkcja zwraca historię wyników, co jest kluczowe dla późniejszej wizualizacji.

```python
def train_validate(model, train_loader, val_loader, criterion, optimizer, epochs=10):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    # Slowniki do przechowywania historii
    history = {'train_loss': [], 'val_loss': [], 'val_acc': []}

    for epoch in range(epochs):
        # --- FAZA TRENINGU ---
        model.train()
        train_loss = 0.0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
```

```
            train_loss += loss.item()

        avg_train_loss = train_loss / len(train_loader)
        history['train_loss'].append(avg_train_loss)

        # --- FAZA WALIDACJI ---
        model.eval() # Wylaczenie Dropout i BatchNorm
        val_loss = 0.0
        correct = 0
        total = 0

        with torch.no_grad(): # Wylaczenie obliczania gradientow
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        avg_val_loss = val_loss / len(val_loader)
        val_accuracy = 100 * correct / total

        history['val_loss'].append(avg_val_loss)
        history['val_acc'].append(val_accuracy)

        print(f"Epoka {epoch+1}: Train Loss: {avg_train_loss:.4f}, "
              f"Val Loss: {avg_val_loss:.4f}, Val Acc: {val_accuracy:.2f}%")

    return history
```

Listing 2: Pętla z walidacją i zbieraniem metryk

4 Wizualizacja procesu uczenia

Analiza wykresów funkcji straty (Loss) i dokładności (Accuracy) pozwala ocenić, czy model uczy się poprawnie, czy występuje overfitting (strata walidacyjna rośnie, gdy treningowa maleje) lub underfitting.

Poniższy kod wykorzystuje bibliotekę `matplotlib` oraz słownik `history` zwrócony przez funkcję z Listingu 2.

```
import matplotlib.pyplot as plt

def plot_training_results(history):
    epochs = range(1, len(history['train_loss']) + 1)

    plt.figure(figsize=(12, 5))

    # Wykres 1: Funkcja Straty (Loss)
    plt.subplot(1, 2, 1)
    plt.plot(epochs, history['train_loss'], 'b-', label='Strata Treningowa')
    plt.plot(epochs, history['val_loss'], 'r--', label='Strata Walidacyjna')
    plt.title('Przebieg Funkcji Straty')
    plt.xlabel('Epoki')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    # Wykres 2: Dok adno    (Accuracy)
```

```
    plt.subplot(1, 2, 2)
    plt.plot(epochs, history['val_acc'], 'g-', label='Dok adno    Walidacyjna')
    plt.title('Dok adno    Modelu')
    plt.xlabel('Epoki')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.savefig('wykresy_treningu.png') # Zapis do pliku
    plt.show()
```

Listing 3: Generowanie wykresów uczenia

5 Podsumowanie

W sprawozdaniu przedstawiono kluczowe aspekty implementacji pętli uczących w PyTorch. Poprawne zarządzanie stanami modelu (`.train()` vs `.eval()`), kontrola gradientów (`.zero_grad()`) oraz monitorowanie metryk na zbiorze walidacyjnym są niezbędne do stworzenia efektywnego i zgeneralizowanego modelu głębokiego uczenia.