Uniwersytet Bielsko-Bialski

Nieliniowe sieci RNN w oparciu o tensory

Matematyka Konkretna

Prowadzący: prof. dr hab. Vasyl Martsenyuk

Autor:

Bartosz Bieniek 058085

1. Cel ćwiczenia

Celem jest nabycie podstawowej znajomości użycia propagacji wstecznej w czasie dla nieliniowych sieci RNN - podstawowe poje¸cia oraz zagadnienia. Zadania umieścić na Github.

2. Przebieg ćwiczenia

1. Import bibliotek

W pierwszym kroku zaimportowano najważniejsze biblioteki do języka Python, pozwalające na wykonanie ćwiczenia:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
```

Rys. 1. Import bibliotek

2. Ustawienie parametrów oraz funkcji konwerujących

Zdefiniowano podstawowe parametry długości bitów dla operacji binarnych: BIT_LENGTH oraz SUM_BIT_LENGTH, uwzględniając maksymalną możliwą długość sumy dwóch liczb binarnych. Następnie zaimplementowano funkcję int_to_bin_array, która przekształcała liczbę całkowitą na wektor bitów w kolejności od najmniej znaczącego bitu (LSB). Opracowano również funkcję bin_array_to_int, która dokonywała konwersji wektora bitów na odpowiadającą mu liczbę całkowitą.

```python
# Parametry
BIT_LENGTH = 28
SUM_BIT_LENGTH = BIT_LENGTH + 1  # max suma dwóch 24-bitowych liczb to 25 bitów

# Funkcja konwertująca liczbę całkowitą na wektor bitów (LSB first)
def int_to_bin_array(x, length=BIT_LENGTH):
    return np.array([int(b) for b in np.binary_repr(x, width=length)][::-1])

# Funkcja konwertująca wektor bitów na liczbę całkowitą
def bin_array_to_int(arr):
    return int("".join(str(b) for b in arr[::-1]), 2)
```

Rys. 2. Zdefiniowanie parametrów początkowych i funkcji konwerujących

3. Implementacja funkcji generującej dane treningowe

Zaimplementowano funkcję generate_data, która generowała dane treningowe do zadania sumowania liczb binarnych. Dla każdej próbki losowano dwie liczby całkowite o długości BIT_LENGTH, po czym konwertowano je na reprezentacje binarne. Obliczano również ich sumę i przekształcano ją do formatu binarnego o długości SUM_BIT_LENGTH. Wektory wejściowe rozszerzano o dodatkowy bit zerowy, aby wyrównać długość do 25 bitów. Dane wejściowe tworzono jako sekwencję par bitów (dla każdej pozycji bitowej obu liczb), a dane wyjściowe jako odpowiadającą im sekwencję bitów sumy. Całość zwracano w postaci tablic NumPy.

```python
# Generujemy dane treningowe
def generate_data(num_samples):
    X = []
    Y = []
    for _ in range(num_samples):
        a = np.random.randint(0, 2**BIT_LENGTH)
        b = np.random.randint(0, 2**BIT_LENGTH)
        a_bin = int_to_bin_array(a)
        b_bin = int_to_bin_array(b)
        s_bin = int_to_bin_array(a + b, length=SUM_BIT_LENGTH)  # suma 25-bit

        # Dodajemy krok czasowy z zerami do wejścia, żeby mieć długość 25
        a_bin_extended = np.append(a_bin, 0)
        b_bin_extended = np.append(b_bin, 0)

        X.append(np.vstack([a_bin_extended, b_bin_extended]).T)  # shape (25, 2)
        Y.append(s_bin)  # shape (25,)
    return np.array(X), np.array(Y)
```

Rys. 3. Implementacja funkcji generującej dane

4. Implementacja modelu sieci neuronowej o nazwie BinaryAdderRNN

Zaimplementowano model sieci neuronowej typu RNN o nazwie BinaryAdderRNN, przeznaczony do sekwencyjnego sumowania liczb binarnych. W konstruktorze zdefiniowano warstwę rekurencyjną nn.RNN przyjmującą na wejściu dwuwymiarowy wektor bitowy oraz warstwę w pełni połączoną nn.Linear, przekształcającą wyjście ukryte do pojedynczego bitu. Na wyjściu zastosowano funkcję aktywacji sigmoid, aby uzyskać wartości w przedziale [0, 1], odpowiadające prawdopodobieństwu dla każdego bitu sumy. W metodzie forward przetwarzano dane wejściowe sekwencyjnie, a wynik końcowy redukowano do postaci dwuwymiarowej macierzy (batch, sequence length).

```python
# Model RNN
class BinaryAdderRNN(nn.Module):
    def __init__(self, input_size=2, hidden_size=16, output_size=1):
        super(BinaryAdderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # x shape: (batch, seq_len, input_size)
        out, _ = self.rnn(x)
        out = self.fc(out)
        out = self.sigmoid(out)
        return out.squeeze(-1)  # shape (batch, seq_len)
```

Rys. 4. Implementacja funkcji generującej dane

5. Implementacja funkcji przekształcającej dane wejściowe i wyjściowe z formatu NumPy

Przygotowano funkcję prepare_tensor_data, która przekształcała dane wejściowe i wyjściowe z formatu NumPy do tensora PyTorch. Dane typu float konwertowano do odpowiedniego formatu liczbowego wymaganego przez model RNN. Dzięki temu dane mogły zostać bezpośrednio wykorzystane w procesie treningu sieci neuronowej.

```python
# Przygotowanie danych do PyTorch
def prepare_tensor_data(X, Y):
    X_t = torch.tensor(X).float()
    Y_t = torch.tensor(Y).float()
    return X_t, Y_t
```

Rys. 5. Implementacja funkcji przekształcającej dane IO do formatu PyTorch

6. Ustalenie hiperparametrów

Ustalono podstawowe hiperparametry treningu: liczbę próbek treningowych (num_samples), rozmiar partii (batch_size) oraz liczbę epok (epochs). Następnie wygenerowano dane treningowe za pomocą wcześniej zdefiniowanej funkcji generate_data i przekształcono je do formatu tensorowego za pomocą prepare_tensor_data. Zainicjalizowano model BinaryAdderRNN, funkcję straty BCELoss odpowiednią dla problemów binarnej klasyfikacji oraz optymalizator Adam z zadanym współczynnikiem uczenia (lr=0.01).

```python
# Hyperparametry
num_samples = 10000
batch_size = 64
epochs = 10

# Generowanie danych
X, Y = generate_data(num_samples)
X_t, Y_t = prepare_tensor_data(X, Y)

# Model, loss, optimizer
model = BinaryAdderRNN()
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Rys. 6. Implementacja funkcji generującej dane

7. Trening

Przeprowadzono proces treningu modelu w zadanej liczbie epok. Dla każdej epoki losowo permutowano indeksy próbek w celu zapewnienia losowego doboru danych do partii treningowych. Następnie, iterując po danych z zadanym rozmiarem partii, wykonywano kroki treningowe: zerowano gradienty, wybierano dane wsadowe, obliczano predykcje modelu, wyliczano stratę przy użyciu funkcji BCELoss, propagowano wstecznie błędy oraz aktualizowano wagi modelu przy użyciu optymalizatora Adam. Dla każdej epoki akumulowano wartość straty, którą wypisywano po zakończeniu przetwarzania wszystkich partii.

```python
# Trening
for epoch in range(epochs):
    permutation = torch.randperm(X_t.size()[0])
    epoch_loss = 0

    for i in range(0, X_t.size()[0], batch_size):
        optimizer.zero_grad()

        indices = permutation[i:i+batch_size]
        batch_x, batch_y = X_t[indices], Y_t[indices]

        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch_loss:.4f}")
```

Rys. 7. Implementacja funkcji treningu

```
Epoch 1/10, Loss: 89.8401
Epoch 2/10, Loss: 3.9460
Epoch 3/10, Loss: 0.5755
Epoch 4/10, Loss: 0.2469
Epoch 5/10, Loss: 0.1356
Epoch 6/10, Loss: 0.0856
Epoch 7/10, Loss: 0.0589
Epoch 8/10, Loss: 0.0429
Epoch 9/10, Loss: 0.0326
Epoch 10/10, Loss: 0.0255
```

Rys. 8. Efekt wywołania funkcji treningu

8. Testowanie modelu

Zaimplementowano funkcję test_model, służącą do testowania wytrenowanego modelu na wybranych przykładach. Dwie liczby całkowite konwertowano do postaci binarnej, a następnie tworzono z nich sekwencję wejściową w formacie wymaganym przez model. Przekształconą sekwencję wprowadzano do modelu w trybie bez śledzenia gradientów (no_grad), a wynik predykcji zaokrąglano do najbliższej wartości całkowitej (0 lub 1), reprezentującej bity sumy. Uzyskany wynik konwertowano z powrotem na liczbę dziesiętną i porównywano z rzeczywistą sumą, wyświetlając oba wyniki w konsoli. Model przetestowano na czterech przykładowych parach liczb, w tym również na wartościach brzegowych.

```python
# Testowanie modelu na kilku przykładach
def test_model(model, a, b):
    a_bin = int_to_bin_array(a)
    b_bin = int_to_bin_array(b)
    x = np.vstack([a_bin, b_bin]).T
    x_t = torch.tensor(x).unsqueeze(0).float()  # batch 1
    with torch.no_grad():
        output = model(x_t).round().numpy().astype(int).flatten()
    sum_pred = bin_array_to_int(output)
    print(f"{a} + {b} = {sum_pred} (model), {a + b} (true)")

print("\nTestowanie modelu:")
test_model(model, 123456, 654321)
test_model(model, 1000000, 2000000)
test_model(model, 0, 0)
test_model(model, 2**27, 2**26)
```

Rys. 9. Implementacja funkcji testowania modelu

3. Wnioski

Skuteczność procesu uczenia: Zaobserwowano wyraźny i systematyczny spadek wartości funkcji straty w kolejnych epokach treningu, co świadczyło o skutecznym uczeniu się modelu. Już po drugiej epoce nastąpił gwałtowny spadek błędu, a dalsze epoki stopniowo poprawiały dokładność predykcji.

Zdolność modelu do generalizacji: Model prawidłowo przewidział wyniki dla różnych przypadków testowych, w tym dla dużych liczb oraz wartości brzegowych. Otrzymane wyniki sumy całkowicie pokrywały się z wartościami rzeczywistymi, co potwierdzało zdolność modelu do uogólniania wiedzy nabytej podczas treningu na nowe, nieznane dane.