SPRAWOZDANIE

Zajęcia: Matematyka Konkretna

Prowadzący: prof. dr hab. Vasyl Martsenyuk

Laboratorium Nr 7 Data 15.06.2025 Temat: Nieliniowe sieci RNN w oparciu o tensory Wariant 6	Imię Nazwisko Hubert Mentel Informatyka II stopień, niestacjonarne, 2 semestr, gr.1a

1. Cel:

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

2. Zadanie:

Opracować rekurencyjną sieć neuronową która implementuje operacje na dwóch liczbach binarnych zgodnie z wariantem zadania:
Wariant 6: Suma dwóch liczb 24-bitowych

Pliki dostępne są pod linkiem:
https://github.com/HubiPX/NOD/tree/master/MK/Zadanie%207

3. Opis programu opracowanego (kody źródłowe, zrzuty ekranu)

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

# Parametry
BIT_LENGTH = 24
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)

# 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)


# 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)

# 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
```

```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)

# 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}")

# 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**22, 2**22)
```

```
Epoch 1/10, Loss: 82.1046
Epoch 2/10, Loss: 2.3206
Epoch 3/10, Loss: 0.5581
Epoch 4/10, Loss: 0.2782
Epoch 5/10, Loss: 0.1702
Epoch 6/10, Loss: 0.1158
Epoch 7/10, Loss: 0.0841
Epoch 8/10, Loss: 0.0638
Epoch 9/10, Loss: 0.0500
Epoch 10/10, Loss: 0.0402

Testowanie modelu:
123456 + 654321 = 777777 (model), 777777 (true)
1000000 + 2000000 = 3000000 (model), 3000000 (true)
0 + 0 = 0 (model), 0 (true)
4194304 + 4194304 = 8388608 (model), 8388608 (true)
```

4. Wnioski

Wykorzystanie rekurencyjnej sieci neuronowej (RNN) do operacji sumowania dwóch 24-bitowych liczb binarnych pozwoliło na skuteczne nauczenie modelu wykonywania tej arytmetycznej funkcji krok po kroku. Sieć, przetwarzając bity sekwencyjnie, mogła modelować zależności i przeniesienia charakterystyczne dla dodawania binarnego, co potwierdziły wyniki testowe — model poprawnie sumował liczby, często nawet w bliskim przybliżeniu lub dokładnie, co świadczy o zdolności RNN do uchwycenia mechanizmu przenoszenia bitów.

Jednocześnie zauważono, że skuteczność modelu silnie zależy od odpowiedniej konfiguracji architektury (np. liczby warstw, liczby neuronów), funkcji strat i parametrów treningowych. Pomimo wysokiej dokładności, model może mieć trudności z poprawnym sumowaniem w skrajnych przypadkach przeniesienia na najstarszy bit, co wskazuje na wyzwania związane z długim zasięgiem zależności sekwencyjnych w RNN. Jednak ogólnie, podejście to pokazuje, że sieci rekurencyjne mogą efektywnie zastąpić klasyczne algorytmy arytmetyczne dla zadań binarnej arytmetyki, co otwiera ciekawą perspektywę dla zastosowań w uczeniu maszynowym i przetwarzaniu sygnałów binarnych.