SPRAWOZDANIE

Zajęcia: Matematyka Konkretna

Prowadzący: prof. dr hab. Vasyl Martsenyuk

Laboratorium Nr 6 Data 14.06.2025 Temat: „Liniowe RNN” Wariant 10	Anna Więzik Informatyka II stopień, niestacjonarne, 2 semestr, gr.1a TTO

1. Polecenie:

Link do repozytorium: https://github.com/AnaShiro/MK_2025

Rekurencyjne sieci neuronowe (RNN) to architektury neuronowe zaprojektowane do przetwarzania sekwencji danych. Charakteryzują się tym, że potrafią przechowywać informacje o wcześniejszych elementach sekwencji w postaci stanów ukrytych, co czyni je szczególnie przydatnymi w zadaniach, gdzie ważny jest kontekst czasowy. W omawianym zadaniu celem było opracowanie nieliniowej sieci RNN, która realizuje zadanie binarnego dodawania dwóch liczb 6-bitowych (rozszerzonych do 7 bitów), przy czym dane wejściowe, stany pośrednie i wyjścia są reprezentowane jako tensory trzeciego rzędu.

Architektura sieci składa się z kilku kluczowych komponentów: warstwy liniowej przetwarzającej dane wejściowe, warstwy rekurencyjnej aktualizującej stan sieci w kolejnych krokach czasowych, oraz warstwy klasyfikacyjnej z funkcją logistyczną obliczającej prawdopodobieństwo wygenerowania bitu wyjściowego. Przetwarzanie tensorowe umożliwia jednoczesne operacje na wielu próbkach i krokach czasowych, co znacznie zwiększa efektywność obliczeń. W celu zapewnienia poprawności implementacji przeprowadzono kontrolę gradientów, a do optymalizacji parametrów wykorzystano algorytm RMSProp z momentem Niestierowa, który pozwala skutecznie trenować sieci głębokie mimo złożoności powierzchni błędu.

2. Opis programu opracowanego

```python
import numpy as np

np.random.seed(1)

Qodo Gen: Options | Test this function
def printSample(x1, x2, t, y=None):
    x1 = ''.join([str(int(d)) for d in x1])
    x1_r = int(''.join(reversed(x1)), 2)
    x2 = ''.join([str(int(d)) for d in x2])
    x2_r = int(''.join(reversed(x2)), 2)
    t = ''.join([str(int(d[0])) for d in t])
    t_r = int(''.join(reversed(t)), 2)
    if y is not None:
        y = ''.join([str(int(d[0])) for d in y])
    print(f'x1: {x1:s}   {x1_r:4d}')
    print(f'x2: + {x2:s}   {x2_r:4d}')
    print(f'      -------   ----')
    print(f't:  = {t:s}   {t_r:4d}')
    if y is not None:
        print(f'y:  = {y:s}')

Qodo Gen: Options | Test this function
def create_sum_dataset(nb_samples, sequence_len):
    max_int = 2**(sequence_len-1)
    format_str = '{:0' + str(sequence_len) + 'b}'
    X = np.zeros((nb_samples, sequence_len, 2))
    T = np.zeros((nb_samples, sequence_len, 1))
    for i in range(nb_samples):
        nb1 = np.random.randint(0, max_int)
        nb2 = np.random.randint(0, max_int)
        X[i,:,0] = list(reversed([int(b) for b in format_str.format(nb1)]))
        X[i,:,1] = list(reversed([int(b) for b in format_str.format(nb2)]))
        T[i,:,0] = list(reversed([int(b) for b in format_str.format(nb1 + nb2)]))
    return X, T
```

```python
Qodo Gen: Options | Test this function
def create_sub_dataset(nb_samples, sequence_len):
    max_int = 2**(sequence_len-1)
    format_str = '{:0' + str(sequence_len) + 'b}'
    X = np.zeros((nb_samples, sequence_len, 2))
    T = np.zeros((nb_samples, sequence_len, 1))
    for i in range(nb_samples):
        nb1 = np.random.randint(0, max_int)
        nb2 = np.random.randint(0, max_int)
        nb1, nb2 = max(nb1, nb2), min(nb1, nb2)
        X[i,:,0] = list(reversed([int(b) for b in format_str.format(nb1)]))
        X[i,:,1] = list(reversed([int(b) for b in format_str.format(nb2)]))
        T[i,:,0] = list(reversed([int(b) for b in format_str.format(nb1 - nb2)]))
    return X, T
```

✓ 0.2s

```python
sequence_len = 15
nb_train = 2000
X_train, T_train = create_sub_dataset(nb_train, sequence_len)

printSample(X_train[0,:,0], X_train[0,:,1], T_train[0,:,:])
```

✓ 0.0s

```
x1: 101001000010110   13349
x2: + 110101110000000    235
      -------   ----
t:  = 010111001100110   13114
```

3. Wnioski

Zadanie pokazało, że rekurencyjne sieci neuronowe mogą skutecznie rozwiązywać zadania binarnego dodawania, o ile odpowiednio zaimplementowane są mechanizmy propagacji w czasie oraz funkcje aktywacji. Wykorzystanie tensorów trzeciego rzędu do reprezentacji wejść, stanów i wyjść umożliwiło jednoczesne i wydajne przetwarzanie wielu próbek. Kontrola gradientów potwierdziła poprawność zaimplementowanej wstecznej propagacji, a zastosowanie RMSProp z pędem Niestierowa przyczyniło się do stabilnej i skutecznej optymalizacji parametrów sieci.

Pomimo prostoty zadania (dodawanie binarne), jego realizacja wymagała złożonego przetwarzania sekwencyjnego oraz uwzględnienia wpływu wcześniejszych kroków czasowych na aktualny stan. Pokazuje to potencjał RNN do zastosowań w bardziej złożonych problemach szeregów czasowych czy przetwarzania języka naturalnego. Przeprowadzone eksperymenty potwierdziły, że odpowiednia konstrukcja architektury, inicjalizacja wag oraz dobór hiperparametrów mają kluczowy wpływ na sukces treningu i ogólną jakość działania modelu.