SPRAWOZDANIE

Zajęcia: Matematyka Konkretna

Prowadzący: prof. dr hab. Vasyl Martsenyuk

| | |
|---|---|
| Laboratorium Nr 6<br>Data 14.06.2025<br>Temat: „Nieliniowe sieci RNN w oparciu o Tensory”<br>Wariant 10 | Szymon Nycz<br>Informatyka<br>II stopień, niestacjonarne,<br>2 semestr, gr.1a TTO |

1. Polecenie:

Link do repozytorium: https://github.com/Maciek332/Semestr_3_Nycz/tree/main/MK

Rekurencyjne sieci neuronowe (RNN) to typ architektury zaprojektowany z myślą o przetwarzaniu danych sekwencyjnych. Ich wyróżniającą cechą jest zdolność do zapamiętywania informacji z wcześniejszych kroków w postaci stanów ukrytych, co czyni je szczególnie skutecznymi w zadaniach, gdzie liczy się kontekst czasowy. W analizowanym przypadku celem było stworzenie nieliniowej sieci RNN, rozwiązującej problem binarnego dodawania dwóch 6-bitowych liczb (rozszerzonych do 7 bitów), przy czym dane wejściowe, stany pośrednie i wyjścia modelowane były jako tensory trzeciego rzędu.

Struktura sieci obejmuje trzy zasadnicze elementy: warstwę liniową odpowiedzialną za wstępne przetwarzanie danych, warstwę rekurencyjną obsługującą aktualizacje stanów w czasie, oraz warstwę wyjściową z funkcją logistyczną, odpowiadającą za generowanie prawdopodobieństwa wystąpienia danego bitu. Użycie tensorów pozwala na jednoczesne przetwarzanie wielu przykładów oraz kroków czasowych, co znacznie podnosi efektywność obliczeniową. W celu weryfikacji poprawności działania przeprowadzono analizę gradientów, a do optymalizacji parametrów modelu zastosowano algorytm RMSProp z momentem Niestierowa, umożliwiający skuteczne szkolenie sieci mimo skomplikowanej topologii przestrzeni błędu.

2. Opis programu opracowanego

```python
import itertools
import numpy as np
import matplotlib.pyplot as plt


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
```

```python
sequence_len = 12
nb_train = 2000
X_train, T_train = create_sum_dataset(nb_train, sequence_len)
printSample(X_train[0,:,0], X_train[0,:,1], T_train[0,:,:])
```

```
x1: 101001000010   1061
x2: + 110101110000    235
      -------   ----
t:  = 000010001010   1296
```

## 3. Wnioski

Badanie wykazało, że rekurencyjne sieci neuronowe są w stanie efektywnie rozwiązywać zadania dodawania binarnego, pod warunkiem właściwego zaimplementowania mechanizmów propagacji czasowej i funkcji aktywacji. Dzięki użyciu tensorów trzeciego rzędu do odwzorowania danych wejściowych, stanów wewnętrznych i wyjściowych, możliwe było jednoczesne i efektywne przetwarzanie wielu przykładów. Weryfikacja gradientów potwierdziła poprawność algorytmu wstecznej propagacji, a użycie optymalizatora RMSProp z pędem Niestierowa wpłynęło na stabilność i skuteczność procesu uczenia.

Chociaż problem dodawania binarnego jest z pozoru prosty, jego rozwiązanie wymagało zaawansowanego przetwarzania sekwencyjnego oraz uwzględnienia wpływu wcześniejszych stanów na bieżące decyzje sieci. Wyniki wskazują na potencjał RNN do pracy z bardziej złożonymi zadaniami, takimi jak analiza szeregów czasowych czy przetwarzanie języka naturalnego. Przeprowadzone testy potwierdziły, że sukces modelu zależy w dużej mierze od odpowiednio zaprojektowanej architektury, właściwej inicjalizacji wag oraz starannego doboru hiperparametrów.