SPRAWOZDANIE

Zajęcia: Matematyka Konkretna

Prowadzący: prof. dr hab. inż. Vasyl Martsenyuk

Laboratorium Nr 9 Data 28.11.2023 Temat: Nieliniowe sieci RNN Wariant 6	Rafał Klinowski Informatyka II stopień, stacjonarne, 2 semestr, gr. a

1. Polecenie:

Ćwiczenie polegało na stworzeniu notatnika Jupyter w języku Python do utworzenia nieliniowej rekurencyjnej sieci neuronowej (RNN) oraz przetestowania jej dla określonego działania matematycznego na liczbach binarnych o określonym rozmiarze.

Wariant zadania: 6

6. Suma dwóch liczb 24-bitowych.

2. Napisany program, uzyskane wyniki

Podczas realizacji laboratorium skorzystano z funkcji, obliczeń i parametrów zaproponowanych w instrukcji laboratoryjnej do laboratorium 9 – Nieliniowe RNN.

Pierwszym krokiem było napisanie funkcji tworzącej zbiór danych uczących lub testowych i wygenerowanie takich zbiorów.

```
# Utworzenie zestawu danych
nb_train = 2000  # Ilość danych treningowych
sequence_len = 24  # Długość sekwencji (liczba bitów)

def create_dataset(nb_samples, sequence_len):
    """Create a dataset for binary addition and
    return as input, targets."""
    max_int = 2**(sequence_len-1) # Maximum integer that can be added
    # Transform integer in binary format
    format_str = '{:0' + str(sequence_len) + 'b}'
    nb_inputs = 2  # Add 2 binary numbers
    nb_outputs = 1  # Result is 1 binary number
    # Input samples
    X = np.zeros((nb_samples, sequence_len, nb_inputs))
    # Target samples
    T = np.zeros((nb_samples, sequence_len, nb_outputs))
    # Fill up the input and target matrix
    for i in range(nb_samples):
        # Generate random numbers to add
        nb1 = np.random.randint(0, max_int)
        nb2 = np.random.randint(0, max_int)
        # Fill current input and target row.
        # Note that binary numbers are added from right to left,
        #  but our RNN reads from left to right, so reverse the sequence.
        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

# Utworzenie zbioru treningowego i testowego
X_train, T_train = create_dataset(nb_train, sequence_len)
```

Rysunek 1. Utworzenie funkcji oraz zbioru testowego. Funkcja tworzy liczby 24-bitowe.

Przetestowano dodawanie dla dwóch przykładów liczb.

```
# Funkcja do wypisywania przykładów
def printSample(x1, x2, t, y=None):
    """Print a sample in a more visual way."""
    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 not y is None:
        y = ''.join([str(int(d[0])) for d in y])
    print(f'x1:   {x1:s}   {x1_r:2d}')
    print(f'x2: + {x2:s}   {x2_r:2d}')
    print(f'      -------   --')
    print(f't:  = {t:s}   {t_r:2d}')
    if not y is None:
        print(f'y:  = {y:s}')

# Wypisanie przykładu dodawania liczb 24-bitowych
printSample(X_train[0,:,0], X_train[0,:,1], T_train[0,:,:])
```

Rysunek 2. Funkcja wypisująca dodawanie i przykład jej wywołania.

```
x1:   000011110111111011100010   4685552

x2: + 001011111100001111111110   8373236

      -------   --

t:  = 001001110100001011100011   13058788
```

Rysunek 3. Wynik dodawania dwóch liczb 24-bitowych.

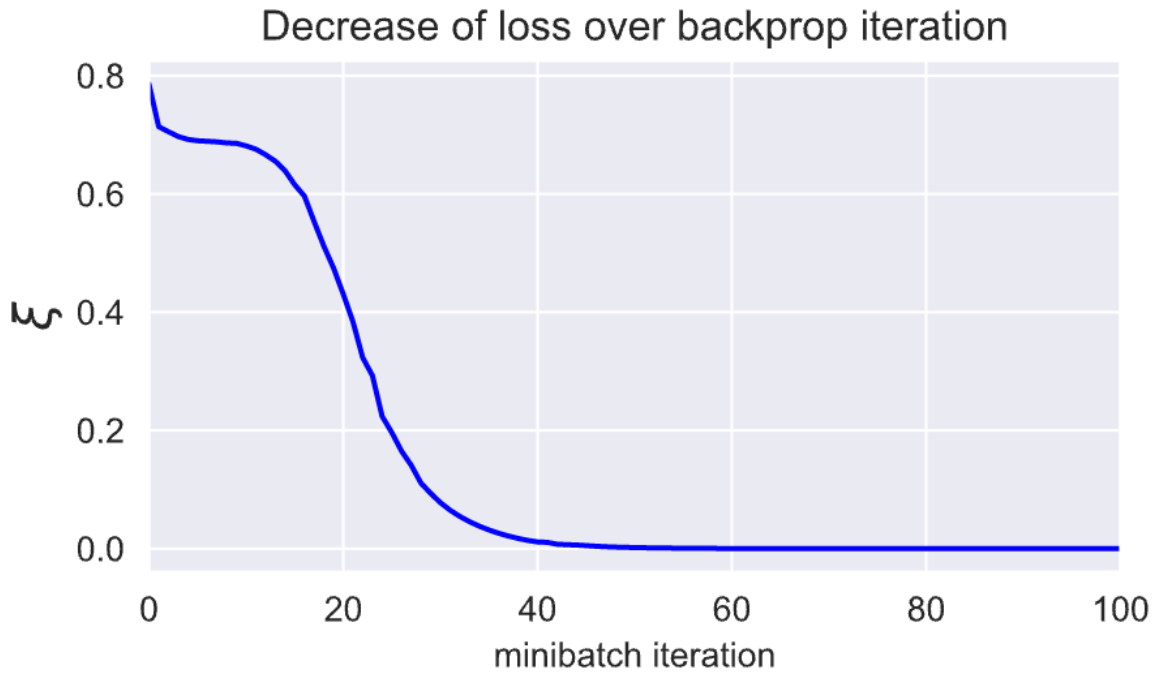
Następnie zaimplementowano architekturę sieci RNN. Implementacja jest dość długa, całość można znaleźć w pliku zawierającym Jupyter Notebook.

```
# Ustalenie hiperparametrów
lmbd = 0.5  # Rmsprop lambda
learning_rate = 0.05  # Learning rate
momentum_term = 0.80  # Momentum term
eps = 1e-6  # Numerical stability term to prevent division by zero
mb_size = 100  # Size of the minibatches (number of samples)

# Utworzenie końcowej sieci RNN
nb_of_states = 3  # Number of states in the recurrent layer
RNN = RnnBinaryAdder(2, 1, nb_of_states, sequence_len)
# Set the initial parameters
# Number of parameters in the network
nbParameters =  sum(1 for _ in RNN.get_params_iter())
# Rmsprop moving average
maSquare = [0.0 for _ in range(nbParameters)]
Vs = [0.0 for _ in range(nbParameters)]  # Momentum

# Create a list of minibatch losses to be plotted
ls_of_loss = [
    RNN.loss(RNN.getOutput(X_train[0:100,:,:]), T_train[0:100,:,:])]
# Iterate over some iterations
for i in range(5):
    # Iterate over all the minibatches
    for mb in range(nb_train // mb_size):
        X_mb = X_train[mb:mb+mb_size,:,:]  # Input minibatch
        T_mb = T_train[mb:mb+mb_size,:,:]  # Target minibatch
        V_tmp = [v * momentum_term for v in Vs]
        # Update each parameters according to previous gradient
        for pIdx, P in enumerate(RNN.get_params_iter()):
            P += V_tmp[pIdx]
        # Get gradients after following old velocity
        # Get the parameter gradients
        backprop_grads = RNN.getParamGrads(X_mb, T_mb)
        # Update each parameter seperately
        for pIdx, P in enumerate(RNN.get_params_iter()):
            # Update the Rmsprop moving averages
            maSquare[pIdx] = lmbd * maSquare[pIdx] + (
                    1-lmbd) * backprop_grads[pIdx]**2
            # Calculate the Rmsprop normalised gradient
            pGradNorm = ((
                                    learning_rate * backprop_grads[pIdx]) /
np.sqrt(
                maSquare[pIdx]) + eps)
            # Update the momentum
            Vs[pIdx] = V_tmp[pIdx] - pGradNorm
            P -= pGradNorm   # Update the parameter
        # Add loss to list to plot
        ls_of_loss.append(RNN.loss(RNN.getOutput(X_mb), T_mb))
```

Rysunek 4. Utworzenie końcowej sieci oraz przeprowadzenie analizy funkcji straty.



Rysunek 5. Funkcja straty w zależności od numeru iteracji.

Na koniec przetestowano sieć dla wygenerowanych danych testowych.

```
# Utworzenie zbioru testowego dla gotowej sieci
nb_test = 5
Xtest, Ttest = create_dataset(nb_test, sequence_len)
# Push test data through network
Y = RNN.getBinaryOutput(Xtest)
Yf = RNN.getOutput(Xtest)

# Print out all test examples
for i in range(Xtest.shape[0]):
    printSample(Xtest[i,:,0], Xtest[i,:,1], Ttest[i,:,:], Y[i,:,:])
    print('')
```

Rysunek 6. Wygenerowanie zbioru testowego i przetestowanie sieci dla jego danych.

x1: 101010110101111011011100 3898069

x2: + 101100110010001001000110 6440141

------- --

t: = 010001011111110110111001 10338210

y: = 010001011111110110111001

Rysunek 7. Przykład uzyskanego wyniku – wartość oczekiwana jest równa wartości uzyskanej z sieci.

Wnioski:

- Uzyskano sieć z funkcją straty końcowo równą 0, co oznacza, że sieć nauczyła się doskonale dodawać liczby 24-bitowe
- Implementacja RNN w taki sposób jest dość długa i wymaga sporo pracy, jednak tym samym daje dobre wyniki, dobrze przystosowuje się do różnych wartości celu oraz jest łatwo rozszerzalna i można ją w prosty sposób modyfikować

Repozytorium zawierające uzyskane wyniki wraz z niezbędnymi plikami: https://github.com/Stukeley/MatematykaKonkretna/tree/master/Lab9