

Lab 6 - Rekurencyjne Sieci Neuronowe

1. Wstęp

Rekurencyjne sieci neuronowe, (RNN, od ang. *Recurrent Neural Networks*), to rodzina sieci neuronowych przeznaczonych do przetwarzania danych sekwencyjnych, takich jak szeregi czasowe, język naturalny, dźwięk czy dane finansowe.

Sieć konwolucyjna (Lab4 i Lab5) jest siecią neuronową wyspecjalizowaną do przetwarzania siatki wartości X , takiej jak obraz. Wymagają one wejść o stałym rozmiarze oraz nie uwzględniają sekwencyjności ani wzajemnych powiązań pomiędzy elementami zbioru.

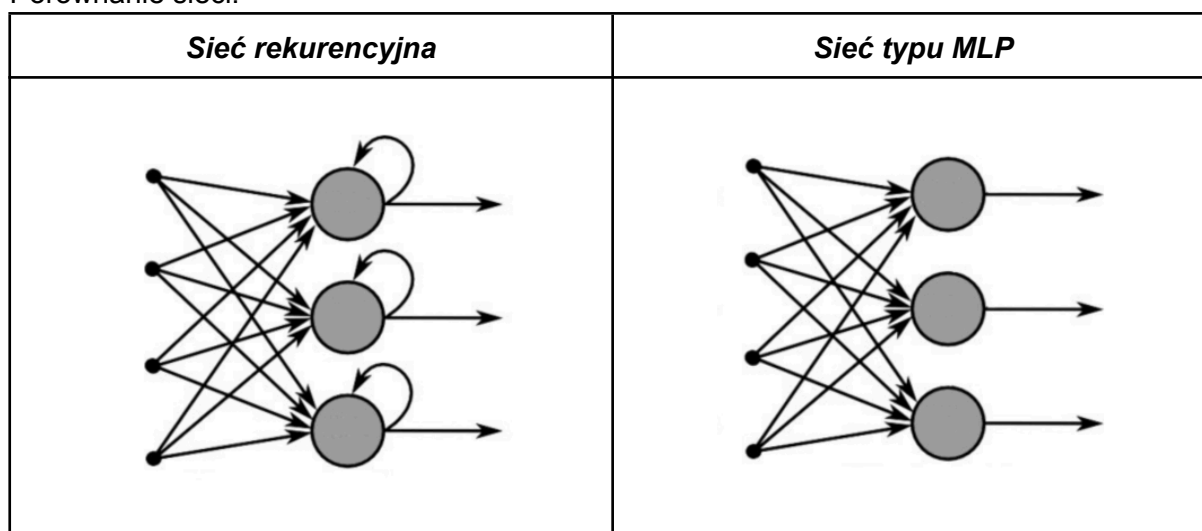
Z kolei rekurencyjne sieci neuronowe są zaprojektowane do przetwarzania sekwencji $x(1), x(2), \dots, x(t)$, umożliwiając analizę danych, w których istotna jest zarówno ich kolejność, jak i zależności pomiędzy kolejnymi elementami.

2. Sieci rekurencyjne

Sieci rekurencyjne przechowują kontekst poprzednich kroków za pomocą tzw. **stanu ukrytego** a (czasem oznaczane w literaturze jako h).

RNN wprowadzają **pętle rekurencyjne**, które umożliwiają przepływ informacji z poprzedniego stanu ukrytego do stanu ukrytego w bieżącym kroku czasowym. Każdy stan ukryty a_t jest obliczany na podstawie wejścia x_t oraz poprzedniego stanu ukrytego a_{t-1} , zwykle z pomocą funkcji aktywacyjnej f .

Porównanie sieci:

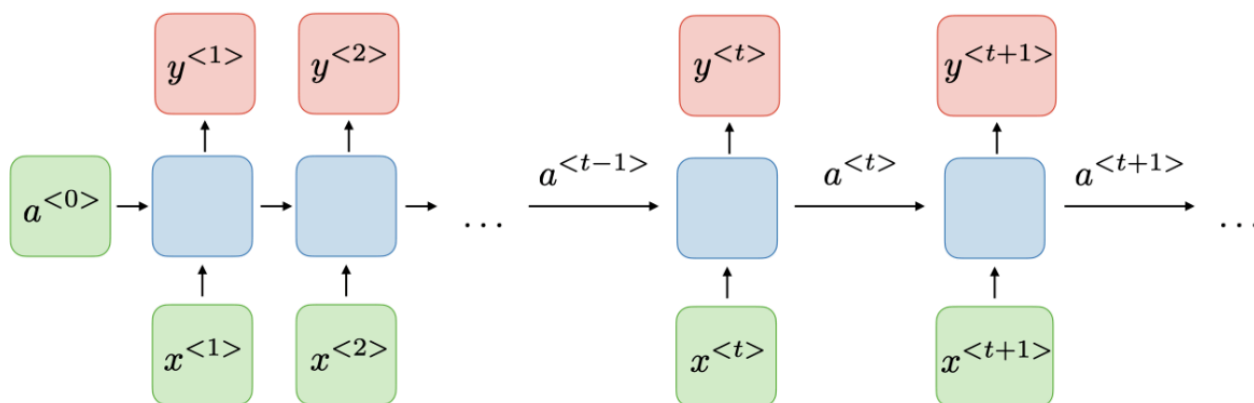


Kluczowe cechy sieci rekurencyjnych:

Pamięć wewnętrzna	Zapamiętywanie wcześniejszych danych wejściowych i wykorzystanie tego kontekstu podczas przetwarzania nowych informacji.
Przetwarzanie danych sekwencyjnych	Radzenie sobie z danymi sekwencyjnymi, w których kolejność elementów ma znaczenie.
Zrozumienie kontekstu	Analizowanie bieżących danych wejściowych w odniesieniu do tego, co występowało wcześniej.
Dynamiczne przetwarzanie	Aktualizowanie na bieżąco pamięci wewnętrznej podczas przetwarzania nowych danych.

3. Typowa Architektura RNN

Typowy RNN działa podobnie jak MLP - przetwarza dane "od lewej do prawej" w czasie. Każdy krok czasowy uwzględnia zarówno bieżące dane wejściowe x_t , jak i stan ukryty z poprzedniego kroku a_{t-1} .



Dla każdego kroku czasowego t , stan ukryty obliczany jest jako:

$$a_t = g_1(W_{aa} a_{t-1} + W_{ax} x_t + b_a)$$

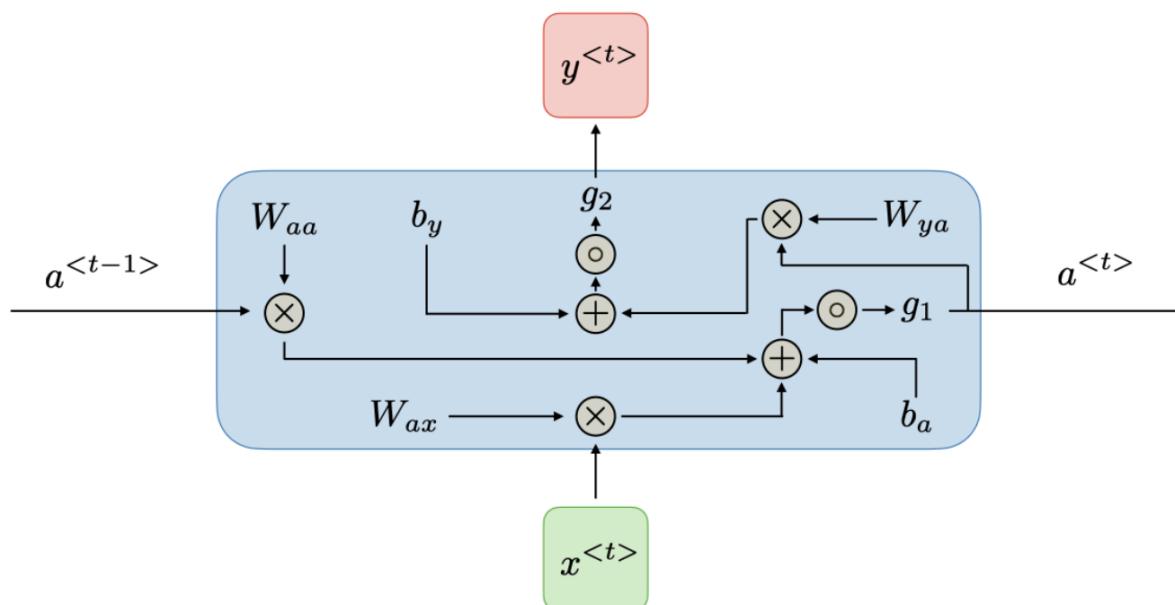
Wyjście y_t obliczane jest jako:

$$y_t = g_2(W_{ya} a_t + b_y)$$

gdzie

- W_{aa}, W_{ax} : macierze wag dla stanu ukrytego i danych wejściowych,
- b_a, b_y : przesunięcia (bias),
- g_1, g_2 : funkcje aktywacyjne; dla g_1 najczęściej używana jest funkcja \tanh .

Powyższe równania można przedstawić jako:



W RNN mamy do czynienia z rozciągnięciem sieci w czasie, ponieważ ta sama sieć jest używana wielokrotnie w różnych krokach czasowych t . Algorytm obliczania gradientów w RNN nazywany jest **propagacją wsteczną przez czas** (ang. *Backpropagation Through Time* – *BPTT*).

Gradient błędu propaguje się wstecznie w czasie przez wszystkie kroki czasowe. Obliczane są pochodne funkcji straty względem wag sieci W_{aa} , W_{ax} , W_{ya} oraz przesunięć b_a , b_y i następnie używane są do aktualizacji wag (np. Z użyciem SGD), podobnie jak w MLP.

Gradient błędu względem stanu ukrytego musi być sumowany dla każdego kroku czasowego, ponieważ wpływ jednego wejścia x_t rozciąga się na przyszłe kroki:

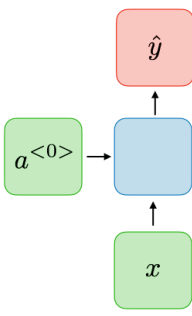
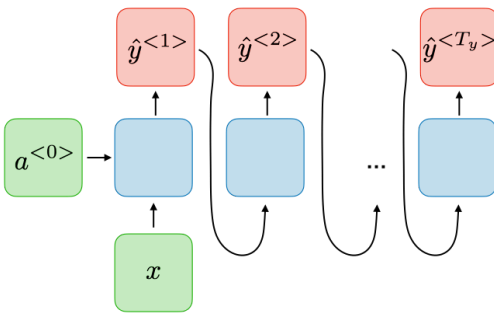
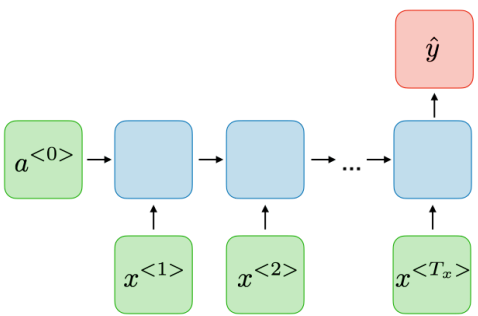
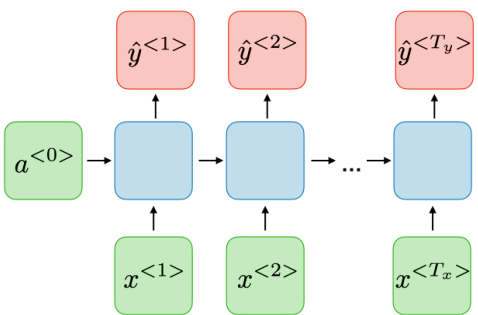
$$\frac{\partial L}{\partial a_t} = \frac{\partial L_t}{\partial a_t} + \frac{\partial L_{t+1}}{\partial a_t} + \dots$$

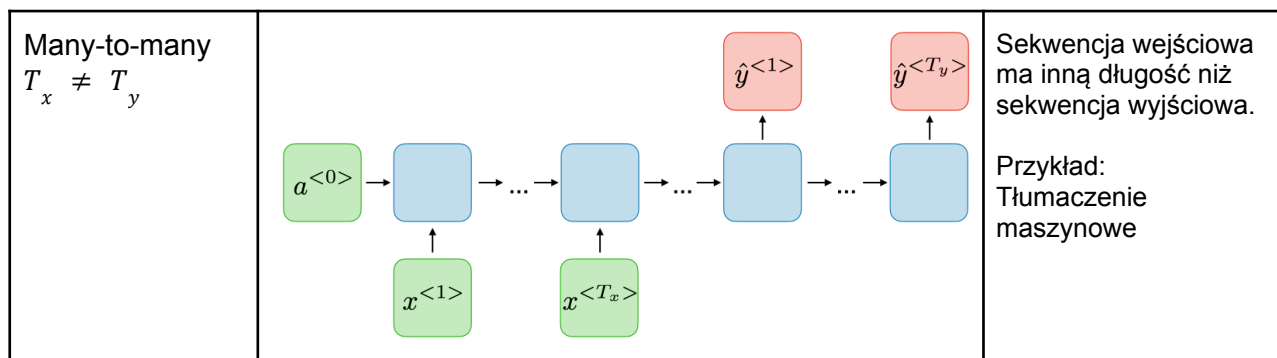
Długie sekwencje powodują, że gradienty szybko stają się bardzo małe (zanikanie gradientów) lub bardzo duże (eksplozja gradientów).

Więcej na temat propagacji wstecznej w RNN można znaleźć [tutaj](#).

4. Typy struktur sieci rekurencyjnych

Istnieje kilka typów struktur sieci rekurencyjnych (RNN), które różnią się sposobem mapowania wejść na wyjścia w kontekście przetwarzania sekwencji:

typ	schemat	Opis i przykład
One-to-one $T_x = T_y = 1$		<p>Każde pojedyncze wejście x jest mapowane na pojedyncze wyjście y (brak sekwencji).</p> <p>Przykład: Klasyczna klasyfikacja</p>
One-to-many $T_x = 1, T_y > 1$		<p>Pojedyncze wejście x generuje sekwencję wyjść.</p> <p>Przykład: Generowanie muzyki Generowanie obrazów</p>
Many-to-one $T_x > 1, T_y = 1$		<p>Cała sekwencja wejściowa x_1, x_2, \dots, x_t jest przetwarzana na pojedyncze wyjście y.</p> <p>Przykład: Analiza sentymentu Rozpoznawanie mowy</p>
Many-to-many $T_x = T_y$ (równoległe)		<p>Mapowanie jest równoległe: wyjście w kroku t zależy od wejścia x_t i poprzedniego stanu ukrytego.</p> <p>Przykład: Tagowanie sekwencji</p>



5. Zanikający i eksplodujący gradient

Zanikający i eksplodujący gradient to problemy, które często występują podczas trenowania rekurencyjnych sieci neuronowych za pomocą BPTT. Problemy te wynikają z natury obliczeń w sieciach rekurencyjnych, zwłaszcza gdy sekwencje danych są długie.

	Przyczyna	Skutek
Zanikający gradient	<p>Zanikający gradient wynika z funkcji aktywacji oraz mnożenia macierzy wag podczas obliczeń. Jeśli sieć używa funkcji aktywacyjnej, która ma pochodne mniejsze niż 1 (np. sigmoidalna lub tanh), to podczas mnożenia kolejnych pochodnych, wartości te maleją.</p> <p>Gdy pochodna $\frac{\partial a_t}{\partial a_{t-1}} < 1$, to przy mnożeniu wielu takich pochodnych ich wartość zmierza do zera.</p> <p>Jeśli gradienty są bardzo małe (bliskie zeru), to podczas kolejnych kroków ich wartość <u>maleje wykładniczo</u>.</p>	<p>1. Sieć zapomina informacje z początku sekwencji, co uniemożliwia skuteczne uczenie długoterminowych zależności.</p> <p>2. Parametry bliżej początku sekwencji uczą się bardzo wolno lub w ogóle się nie uczą.</p>
Eksplodujący gradient	<p>Eksplodujący gradient pojawia się, gdy wartości wag są duże, a ich pochodne mają wartość znacznie większą niż 1. W miarę propagacji wstecznej mnożenie tych wartości prowadzi do bardzo dużych gradientów.</p> <p>Gdy pochodna $\frac{\partial a_t}{\partial a_{t-1}} \gg 1$, to wartości gradientów <u>rosną wykładniczo</u> w miarę propagacji wstecznej.</p>	<p>1. Wagi sieci mogą przyjmować bardzo duże wartości, co prowadzi do niestabilnego uczenia.</p> <p>2. Model może nie zbiegać do optymalnych wartości.</p>

Rozwiązania:

- **Gradient Clipping** - Ograniczanie wartości gradientu do pewnego maksymalnego progu.
- **Użycie “bramek”** - kontrola przepływu informacji (używane przez LSTM i GRU).
- **Regularizacja/normalizacja wag** - Dodanie kar za zbyt duże wagi, np. z użyciem L2 czy batch normalization.

6. LSTM

LSTM (ang. *Long Short-Term Memory*) to specjalny rodzaj RNN, zaprojektowany w celu rozwiązania problemu zanikającego gradientu. LSTM może efektywnie zapamiętywać długoterminowe zależności, dzięki komórkom pamięci oraz mechanizmom bramkowania, które decydują o tym, które informacje należy zapamiętać, a które odrzucić.

LSTM posiada trzy główne bramki:

1. **Bramka zapominania** (ang. *Forget Gate*) - Decyduje, które informacje z komórki pamięci powinny zostać usunięte.
2. **Bramka wejściowa** (ang. *Input Gate*) - Decyduje, które nowe informacje dodać do komórki pamięci.
3. **Bramka wyjściowa** (ang. *Output Gate*) - Decyduje, jaka część informacji z komórki pamięci powinna zostać przekazana jako wynik w bieżącym kroku czasowym.

Przy każdym kroku czasowym t , LSTM wykonuje następujące operacje:

1. Obliczenie bramki zapominania:

$$F_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

2. Obliczenie bramki wejściowej:

$$I_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

3. Obliczenie nowej propozycji aktualizacji komórki pamięci \bar{C}_t :

$$\bar{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

4. Aktualizacja komórki pamięci C_t :

$$C_t = F_t \odot C_{t-1} + I_t \odot \bar{C}_t$$

\odot oznacza iloczyn Hadamarda, czyli mnożenie element po elemencie

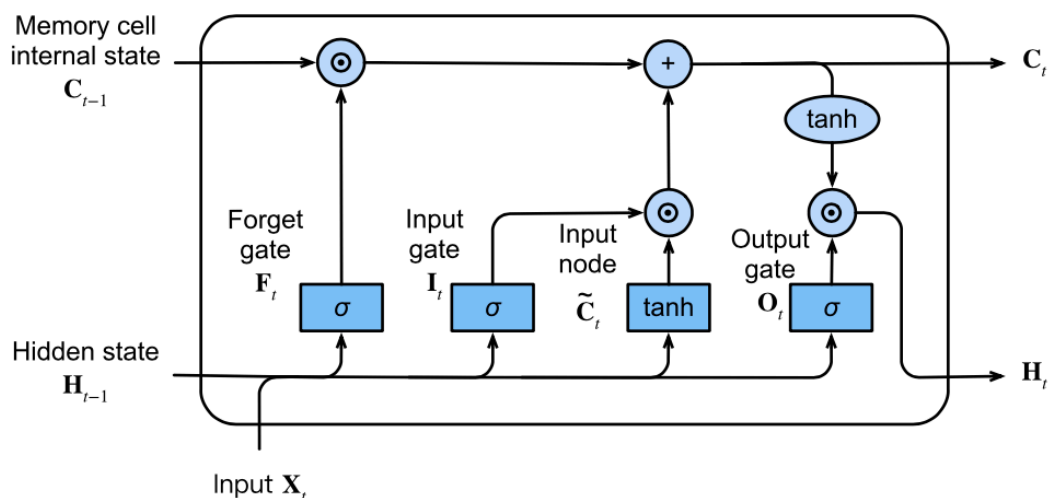
5. Obliczenie bramki wyjściowej:

$$O_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

6. Obliczenie nowego stanu ukrytego:

$$H_t = O_t \odot \tanh(C_t)$$

Schematycznie można to przedstawić jako:



7. GRU

GRU (ang. *Gated Recurrent Unit*) to uproszczona wersja LSTM. GRU została zaprojektowana, aby rozwiązywać problem zanikającego gradientu i jednocześnie zmniejszyć złożoność obliczeniową, która występuje w LSTM.

GRU wykorzystuje dwie bramki zamiast trzech (resetująca i aktualizująca). GRU nie posiada oddzielnego stanu komórki C_t , jak w LSTM. Zamiast tego używa tylko stanu ukrytego h_t , co zmniejsza liczbę parametrów i upraszcza obliczenia. GRU może mieć gorszą wydajność w złożonych zadaniach z długimi sekwencjami.

Więcej na temat GRU można znaleźć [tutaj](#).

8. Przykład

Korzystając z danych Narodowego Banku Polskiego ([Archiwum kursów średnich](#)) opracowano model LSTM do przewidywania kursu USD -> PLN na następny dzień na podstawie ostatnich 60 dni. Wykorzystano dane 2021->2024.

- Ładowanie i przygotowanie danych

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dropout, Dense

value_df_raw = pd.DataFrame()
for year in range(1, 5):
    value_temp = pd.read_csv(
        f"archiwum_tab_a_202{year}.csv",
        encoding='ISO-8859-1',
        sep=';'
    )[['data', '1USD']]

    value_df_raw = pd.concat([value_df_raw, value_temp], ignore_index=True)

value_df_raw = value_df_raw.fillna("NaN")
value_df = value_df_raw[value_df_raw['data'].str.isdigit()]
value_df['usd'] = value_df['1USD'].str.replace(',', '.').astype(float)
value_df = value_df[['data', 'usd']]

value_df['data'] = pd.to_datetime(value_df['data'], format='%Y%m%d',
errors='coerce')
value_df['data_str'] = value_df['data'].dt.strftime('%Y-%m-%d')
```

- Normalizacja danych

Przed użyciem danych w modelu, należy je znormalizować, aby miały zakres od 0 do 1 (MinMaxScaler)

```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(value_df['usd'].values.reshape(-1, 1))
```


- Przygotowanie sekwencji

Przygotowano dane wejściowe (X) oraz wyjściowe (y) do modelu LSTM. Każde wejście (X) stanowi sekwencję 60 dniowych kursów USD, a odpowiedź (y) to kurs USD na kolejny dzień. Dla każdej daty utworzono sekwencję wejściową i odpowiadający jej wynik.

```
look_back = 60
X, y = [], []

for i in range(lookback, len(scaled_data)):
    X.append(scaled_data[i - look_back:i, 0])
    y.append(scaled_data[i, 0])

X, y = np.array(X), np.array(y)
X = np.reshape(X, (X.shape[0], X.shape[1], 1))
```

- Podział na dane treningowe i testowe

Dane zostały podzielone na zbiór treningowy (80%) i testowy (20%).

```
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
```

- Budowanie modelu LSTM

Stworzono model składający się z trzech warstw LSTM, z których każda posiada 50 jednostek. Dodatkowo zastosowano warstwy Dropout w celu zapobiegania przeuczeniu. Na końcu dodano warstwę Dense, która przewiduje kurs.

Jednostki odnoszą się do liczby "komórek" w każdej warstwie LSTM, które przechowują i przetwarzają informacje w czasie

```
model = Sequential([
    LSTM(units=50, return_sequences=True, input_shape=(X.shape[1], 1)),
    Dropout(0.2),
    LSTM(units=50, return_sequences=True),
    Dropout(0.2),
    LSTM(units=50),
    Dropout(0.2),
    Dense(units=1)
])

model.compile(optimizer='adam', loss='mean_squared_error')
```

- Trenowanie modelu

```
history = model.fit(X_train, y_train, epochs=20, batch_size=32)
```

- Przewidywanie wartości na zbiorze testowym

```
predicted_values = model.predict(X_test)

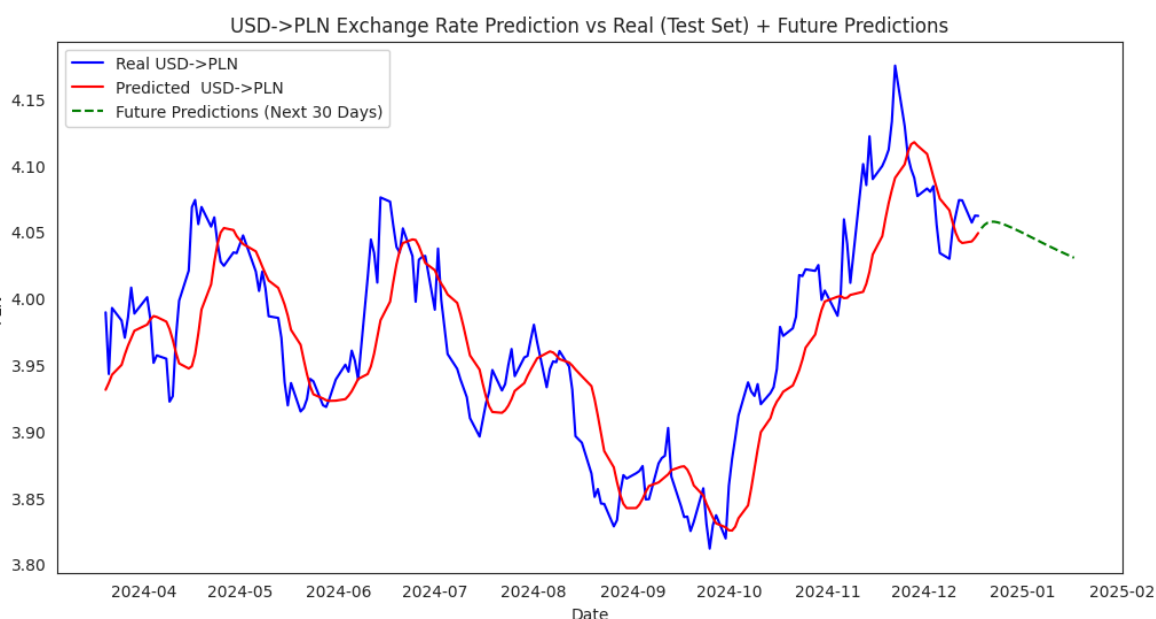
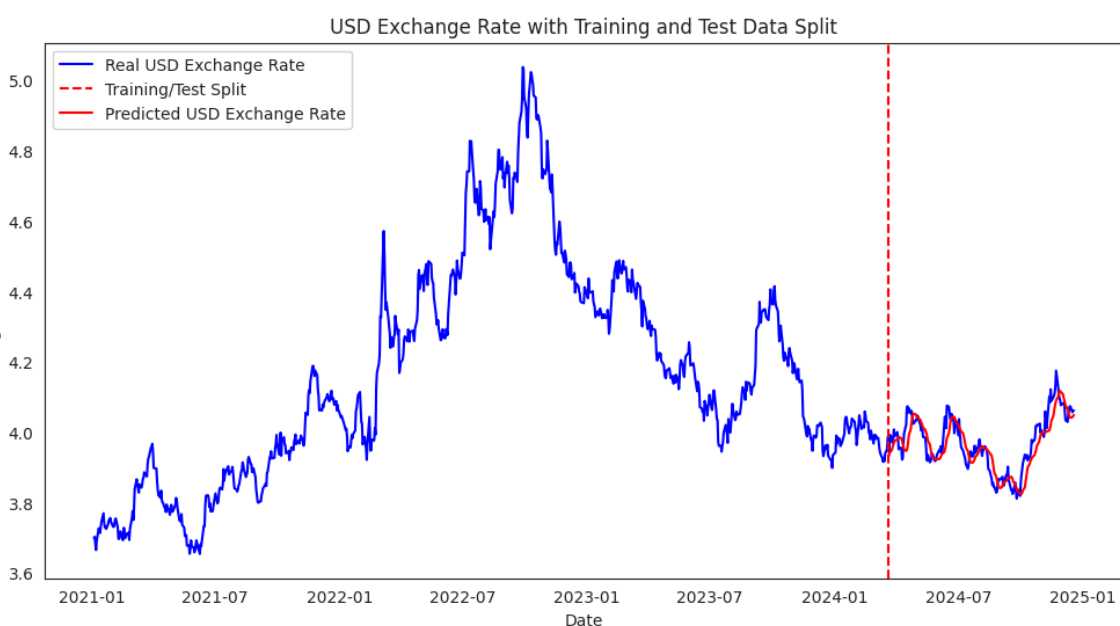
predicted_values = scaler.inverse_transform(predicted_values)
real_values = scaler.inverse_transform(y_test.reshape(-1, 1))
```

- Generowanie prognoz na przyszłość

Prognozy na przyszłość (na 30 dni) zostały wygenerowane na podstawie ostatnich 60 dni.

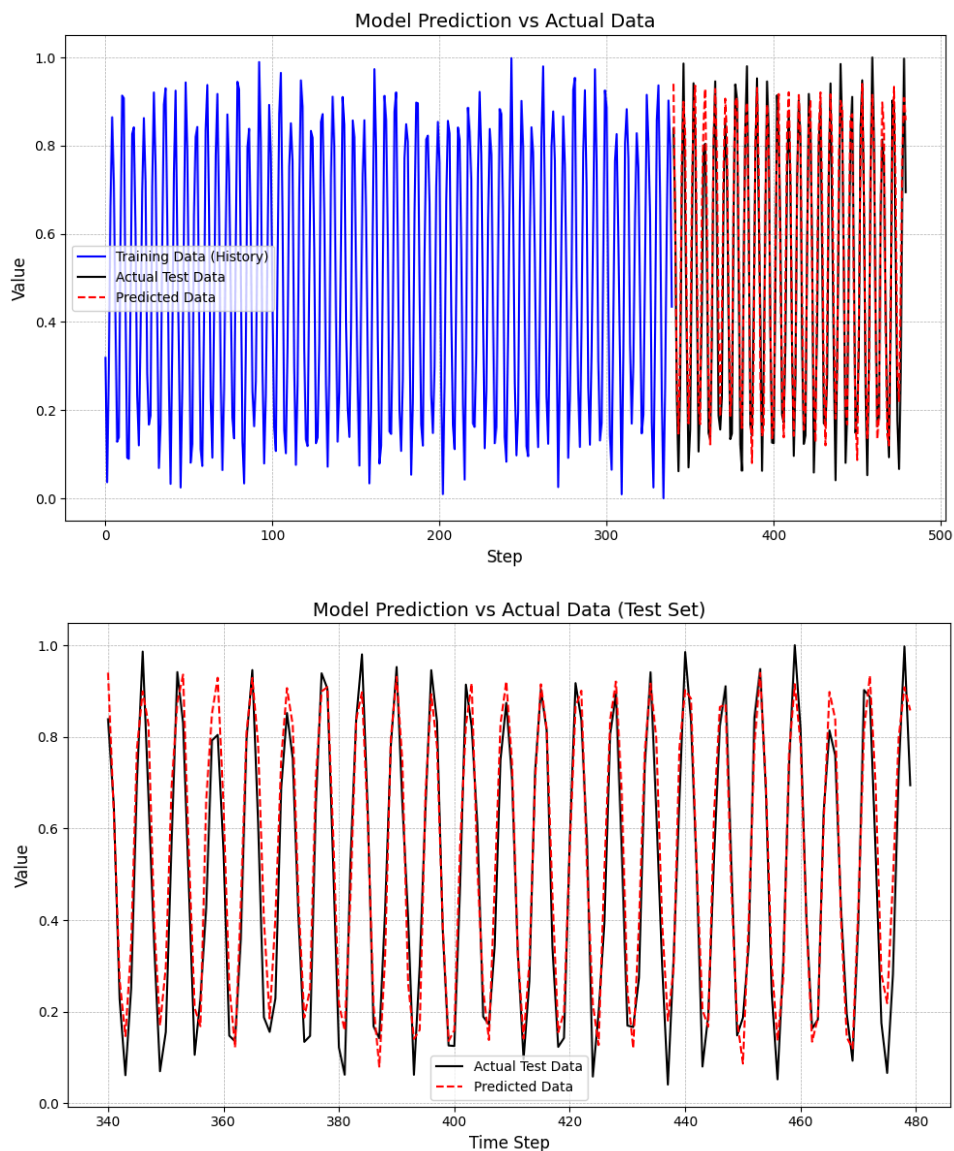
```
history = model.fit(X_train, y_train, epochs=20, batch_size=32)
```

- Wyniki



9. Zadanie

Opracuj model oparty na LSTM, który będzie w stanie przewidzieć sekwencje o sinusoidalnym charakterze z dodanym szumem losowym. Wygeneruj szereg czasowy, przygotuj sekwencje danych, podziel je na zbiory testowe i treningowe, zbuduj oraz wytrenuj model. Oczekiwany wynik:



Protips:

- Do wygenerowania szeregu czasowego można skorzystać z `np.sin` i dodać losowy szum, np. przy pomocy `np.random.rand` (`dataset = np.sin(np.arange(500)) + np.random.rand(500)*0.4`)
- Jako model można skorzystać z prostego modelu, np. `model = keras.Sequential([keras.layers.LSTM(8, input_shape=(X_train.shape[1], X_train.shape[2])), keras.layers.Dense(y_train.shape[1])])`

10. Zadanie

Na podstawie danych Głównego Inspektoratu Środowiska ze [stacji pomiarowej Kraków, ul. Złoty Róg](#) (znajdują się na Teams) opracuj model do przewidywania stężenia pyłów zawieszonych PM-10.

Przeprowadź badania różnych architektur modeli sekwencyjnych.