Eksperymenty na prostych sieciach neuronowych

Natalia Pospiech i Iga Zatylna
Specjalność: Matematyka Stosowana

Maj 2026

Spis treści

1 Wprowadzenie

Praca *Eksperymenty na prostych sieciach neuronowych* przedstawia trzy doświadczenia, których głównym założeniem jest zaprogramowanie modelu sieci neuronowej dedykowanego dla każdego z trzech postawionych problemów:

- bramki XOR dla 10-elementowych wejść
- predykcji cukrzycy
- predykcji szeregów czasowych dotyczących pogody.

Część programistyczna pracy powstała w oparciu o bibliotekę *tensorflow* i moduł *Keras* w języku Python. Praca powstała w ramach przedmiotu *Zespołowy projekt specjalnościowy* w semestrze letnim roku akademickiego 2025/26 prowadzonego na Uniwersytecie Wrocławskim.

2 Wstęp

2.1 Krótka historia sieci neuronowych

Sieci neuronowe mają szerokie zastosowanie w każdej dziedzinie życia. Systemy oparte na nich używane są powszechnie od rozpoznawania języka naturalnego po diagnostykę medyczną oraz procedury bezpieczeństwa. Pierwsze badania nad sieciami, których działanie miało być oparte na tym jak działa mózg człowieka, zostały rozpoczęte już w latach 40. XX stulecia. W 1943 r. Warren McCulloch and Walter Pitts opublikowali artykuł [1] zawierający matematyczny opis modelu systemu nerwowego jako prostej sieci zawierającej jedynie elementy logiczne. Uznawany za pierwowzór, jego koncepcja przyczyniła się do powstania oraz rozwoju algorytmów uczenia maszynowego używanych współcześnie.

Przez wiele lat trwały badania nad perceptronem Rosenblatta, który, jak twierdzili ówcześni naukowcy, miał przynieść początek sztucznej inteligencji, jaką znamy dzisiaj. Metoda Rosenblatta pozwoliła na uczenie się sieci i rozwiązała problem jego poprzedników, którzy liczbę wejść, wagi i bias ustawiali przez zgadywanie - ręcznie. Jednak pod koniec lat 60 – tych XX wieku, Minsky i Papert w swojej książce [2] udowodnili, że perceptron jednowarstwowy działa tylko dla danych separowalnych liniowo (które da się podzielić na dwie odrębne klasy za pomocą hiperpłaszczyzny) i nie rozwiąże on, na przykład problemu XOR. Ich publikacja na długie lata zatrzymała rozwój sieci neuronowych, mimo błędnej tezy co do perceptronów wielowarstwowych (autorzy nie sądzili, że bariera obliczeniowa, która wtedy wydawała się nie do pokonania, zostanie przełamana przez dzisiejsze procesory), które w późniejszych latach stały się fundamentem uczenia maszynowego.

2.2 Budowa sieci neuronowej

Rozważania na temat sieci neuronowych warto rozpocząć od przedstawienia terminologii [3] i podstaw ich architektury.

Perceptron to prosta, sztuczna sieć neuronowa przedstawiona przez Franka Rosenblatta w 1958 roku [4]. Jego podstawowa postać (perceptron prosty) składa

się z jednego sztucznego neuronu McCullocha-Pittsa, który stanowi matematyczny model neuronu [5].

W jego schemacie rozważamy wiele wejść $x_1, x_2, ..., x_n$ wraz z przyporządkowaną liczbą rzeczywistą w_i nazywaną wagą oraz jedno wyjście. Wyjście definiujemy jako wartość funkcji aktywacji dla sumy iloczynów $x_i w_i$, do której dodajemy wartość w_0, którą nazywamy bias.

Pojęcie **funkcji aktywacji** definiujemy jako funkcję, według której obliczana jest wartość wyjścia neuronów sieci neuronowej [6]. Najczęściej rozważanymi funkcjami są:

- Funkcja liniowa (neuron liniowy)

$$f(x) = ax + b$$

- **Funkcja sigmoidalna** (neuron sigmoidalny)

$$f(x) = \frac{1}{1+e^{-\beta x}}$$

- **Funkcja tangensoidalna** (neuron tangensoidalny) - oparta na tangensie hiperbolicznym

$$f(x) = \frac{1-e^{-\beta x}}{1+e^{-\beta x}}$$

- **Funkcja ReLU** - Rectified Linear Unit

$$\text{f(x)} = \max(0, x) = \begin{cases} 0, \text{dla x} < 0 \\ x, \text{dla x} \geq 0 \end{cases}$$

W szczególności funkcja sigmoidalna i ReLU są istotne w rozwiązywaniu licznych problemów z wykorzystaniem sieci neuronowych. Funkcja sigmoidalna pozwala na mapowanie wartości do przedziału [0,1], co w szczególności pozwala na klasyfikację binarną i reprezentację prawdopodobieństwa. To często czyni ją dobrym wyborem na funkcję aktywacji w warstwie wyjściowej. Funkcja ReLU wyróżnia się tym, że pozwala na szybsze obliczenia w procesie uczenia i zapobiega problemowi znikającego gradientu.

Możemy rozważać też **perceptron wielowarstwowy** (ang. Multilayer Perceptron, MLP), który obecnie jest najpopularniejszym typem sztucznych sieci neuronowych jednokierunkowych. Struktura takiej sieci składa się z warstw:

- jednej warstwy wejściowej
- kilku warstw ukrytych (najczęściej złożonej z neuronów sigmoidalnych)
- jednej warstwy wyjściowej (najczęściej złożonej z neuronów sigmoidalnych lub liniowych)

Uczenie maszynowe dzielimy na trzy kategorie:

- Uczenie nadzorowane (ang. supervised learning)
- Uczenie nienadzorowane (ang. unsupervised learning)
- Uczenie przez wzmacnianie (ang. reinforcement learning)

Uczenie nadzorowane ma na celu trenowanie modelu za pomocą oznakowanych danych uczących (ang. training data) do przewidywania informacji z przyszłości. Jego cecha "nadzorowane" odnosi się do obecności etykiet, które stanowią znane nam sygnały wejściowe i są one częścią danych wejściowych do modelu.

Uczenie przez wzmacnianie zakłada utworzenie systemu składającego się z agenta, który na podstawie interakcji ze środowiskiem poprawia skuteczność algorytmu. System nie potrzebuje "nauczyciela", który pokazuje, jaki wykonać kolejny krok, tylko określa poprawność poprzednich kroków i na tej podstawie decyduje, jaką akcję następnie wykonać. Określenie korzystności poprzedniego kroku agenta jest określane nagrodą i celem jest jej maksymalizacja.

W uczeniu nienadzorowanym najczęściej mamy do czynienia z danymi nieoznakowanymi bądź o nieznanej strukturze. Modele tego typu pozwalają odkryć strukturę danych i nie wykorzystują znanych informacji jak oczekiwana zmienna wyjściowa lub funkcja nagrody.

W uczeniu maszynowym kluczowe jest odpowiednie przygotowanie danych, często ich normalizacja i skalowanie.

Ponadto, dane dzieli się (najczęściej w stosunku 70/15/15) na trzy rozłączne zbiory [7] :

- zbiór trenigowy, który służy do nauki zależności oraz dostosowywania parametrów,

- zbiór walidacyjny, który pozwala na dobranie jak najlepszych parametrów modelu przez ich testowanie,

- zbiór testowy, który pozwala na ocenę dokładności.

Jednym z najpopularniejszych algorytmów uczenia sieci neuronowej jest algorytm propagacji wstecznej (ang. **backpropagation**). Polega on na wyznaczeniu błędów dokładnych na podstawie danych wyjściowych i odpowiedzi wzorcowych ze zbioru uczącego. Propagacja wsteczna jest wykorzystywana dla neuronów w warstwach ukrytych.

Funkcja kosztu (ang. cost function lub loss function) to miara poprawności modelu prognozującego wyniki na podstawie danych treningowych. Celem algorytmów uczenia sieci neuronowych jest minimalizowanie funkcji kosztu, co pozwala na coraz dokładniejsze prognozowanie informacji. Poniżej zaprezentowane zostały najczęściej wykorzystywane funkcje kosztu.

- **Binary Cross – Entropy** (entropia krzyżowa) to logarytmiczna funkcja kosztu, mierząca różnicę między przewidywalnymi przez model prawdopodobieństwami a właściwymi etykietami (przyjmującymi wartości 0 lub 1) [8]. Funkcja postaci logarytmu naturalnego uczy się przypisywać wartości bliskie 1 do odpowiedniej etykiety a bliskie 0 do drugiej, dzięki temu model uczy się zwiększać prawdopodobieństwo poprawnych predykcji i zmniejszać błędnych, minimalizując przy tym wartość funkcji straty. Jest standardowym wyborem w problemach klasyfikacji binarnej i przykładem probabilistycznej funkcji straty [9].

Wzór binary cross - entropy

$$BCE = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)]$$

gdzie

N – liczba obserwacji

y_i – faktyczna etykieta (0 lub 1) dla i-tej obserwacji

p_i – przewidywane prawdopodobieństwo, że i-ta obserwacja będzie etykietowana 1

Kolejne funkcje są przykładami regresyjnych funkcji straty [10].

- **Błąd średniokwadratowy** (ang. Mean Square Error, MSE)

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_{true} - y_{pred})^2$$

gdzie

N – liczba obserwacji

y_i – etykieta

p_i – przewidywana wartość

- **Uśredniony błąd bezwzględny** (ang. Mean Absolute Error, MAE)

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_{true} - y_{pred}|$$

gdzie

N – liczba obserwacji

y_i – etykieta

p_i – przewidywana wartość

- **Huber Loss** to funkcja, która łączy cechy błędu średniokwadratowego i średniego błędu bezwzględnego. Składa się z części kwadratowej (MSE), aktywowanej przy małych błędach i jednocześnie karającej za wartości odstające. Część liniowa (MAE) pozwala na mniejszą wrażliwość i naukę modelu tak, by nie dodawał zbyt dużej kary za duży błąd. [11]

$$L_\delta(y, p) = \begin{cases} \frac{1}{2}(y-p)^2, \text{ dla } |(y-p)| \leq \delta \\ \delta|y-p| - \frac{1}{2}\delta^2, \text{ dla } |(y-p)| > \delta \end{cases}$$

y – etykieta

p – przewidywana wartość

δ – parametr decydujący o zmianie zachowania funkcji - numeryczna granica między zachowaniem podobnym do MSE/MAE.

Algorytm gradientowy jest jednym z najskuteczniejszych narzędzi minimalizowania funkcji kosztu. Polega on na wyborze losowych wag i bias, by iteracyjnie znaleźć ich najlepsze wartości poprzez określanie kierunku (odwrotny gradient) zmniejszającego stratę. Przypisuje te wartości wagom i bias oraz powtarza proces do uzyskania najlepszych wyników. Wybór wielkości kroku decyduje o tym, jak duża zmiana nastąpi w wielkości parametrów. Zbyt mały krok spowoduje ogromną liczbę iteracji, co przełoży się na szybkość uczenia, zbyt duży na niższą dokładność bądź rozbieżność (błąd zamiast maleć, zaczyna rosnąć).

W algorytmie propagacji wstecznej pojawia się problem zanikających gradientów [12], który polega na dużej rozbieżności w wielkości gradientów między warstwami, napotykany podczas trenowania sieci neuronowych. W takich algorytmach wagi w sieci są zmieniane na podstawie wartości pochodnej cząstkowej funkcji straty (kosztu). Wraz z kolejnymi krokami w głąb sieci, gradienty są wielokrotnie mnożone i tym samym zmniejszane, co zwiększa różnicę gradientów między głębszymi warstwami a początkowymi. To skutkuje niestabilnością numeryczną w trenowaniu sieci bądź całkowitą porażką jego wykonania. Częstym rozwiązaniem problemu jest wykorzystanie funkcji aktywacji ReLU, bądź Batch Normalization.

W naszej pracy zostały przeprowadzone trzy eksperymenty. Każdy z nich ma inną charakterystykę, co pokazuje różnorodne możliwości zastosowań teorii sieci neuronowych.

3 Bramka XOR z 10-cioma zmiennymi

3.1 Opis eksperymentu

Pierwszym eksperymentem, który zostanie przeprowadzony, będzie dotyczył bramki XOR.

Problem bramki XOR jest trudny dla sieci neuronowych, ponieważ jest on silnie liniowo nieseparowalny. Zauważmy, że jeśli rozważamy ciąg 10 wartości 0-1, bramka XOR da nam inną wartość już przy pojedynczej zmianie jednego bitu, a więc przy najmniejszej możliwej modyfikacji. To sprawia, że sieci neuronowej trudno na zasadzie prawdopodobienstwa przewidzieć wartość dla danego wejścia i tym bardziej nauczyć się zasady matematycznej stojącej za wynikami.

3.2 Konstrukcja modelu i opis architektury

Pierwsza konstrukcja sieci odbyła się przy pomocy tutorialu [13], w którym przedstawiono problem dwuwejściowej bramki XOR.

Do konstrukcji sieci neuronowej wykorzystana została biblioteka *tensorflow keras*, która jest standardowym narzędziem do uczenia maszynowego. Program powstał w języku Python.

W sieci neuronowej wykorzystane zostały funkcje aktywacji ReLU oraz funkcję sigmoidalną. To połączenie świetnie sprawdza się w problemach binarnych oraz o nieliniowej naturze, dlatego zostały wybrane do eksperymentu bramki XOR, którego wynikiem ma być prawda lub fałsz. W warstwach ukrytych na funkcję aktywacji została wybrana funkcja ReLU, która często jest standardowym wyborem w problemach wykorzystujących perceptron wielowarstwowy. W przeciwieństwie do innych funkcji aktywacji potrafi ona zapobiegać problemowi znikającego gradientu, przez co umożliwia efektywniejszą naukę sieci neuronowej [14]. Funkcja sigmoidalna wykorzystana w ostatniej warstwie stanowi ostateczny szlif, który sprowadza wartości do przedziału (0,1) i pozwala na ostateczne przyporządkowanie binarne. Pośród warstw ukrytych znajduje się również warstwa Dropout, która losowo wyłącza 20 procent neuronów, by zapobiec przeuczeniu (tzw. overfittingowi), oraz BatchNormalization, czyli warstwa normalizująca dane podczas treningu, co pozwala na zachowanie stabilności i szybkości uczenia. Celem uniknięcia przeuczenia dodano również EarlyStopping, który sprawdza spadek błędu na zbiorze testowym i jeśli przestanie spadać, wraca do najskuteczniejszej wersji sieci.

Liczba neuronów w warstwach jest ustalona tak, by ułatwić sieci naukę problemu na wysokim poziomie złożoności z zachowaniem jak najlepszej wydajności algorytmu i dynamiki treningu, a jednocześnie uniknięcia overfittingu.

Przy kompilacji modelu został wykorzystany optymalizator Adam z logarytmiczną funkcją kosztu – binary cross entropy.

Optymalizator Adam (ang. Adaptive Moment Estimation) to algorytm optymalizacyjny wykorzystywany w szkoleniu sieci neuronowych. Jego głównym celem jest minimalizacja funkcji kosztu, poprzez dobór odpowiednich wag. Łączy dwie metody uczenia RMSProp i Momentum, dzięki czemu jest bardziej efektywny od

innych algorytmów. Adam wykorzystuje momentum jako swój pierwszy moment, by przyspieszyć proces spadku gradientu poprzez średnią wykładniczą. Pozwala to na szybszą zbieżność dzięki redukcji oscylacji. Drugi moment (ang. Root Mean Square Propagation) służy do skalowania gradientów. Śledzenie średnich kwadratów gradientów pomaga w wyborze wielkości kroku.

3.3 Część programistyczna

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Dropout,
    BatchNormalization
from tensorflow.keras.optimizers import Adam
import tqdm

# 1. Generujemy X - wszystkie 1024 kombinacje 10-bitowe
num_inputs = 10
X = np.array([[(i >> j) & 1 for j in
    reversed(range(num_inputs))] for i in
    range(2**num_inputs)])

# 2. Generujemy y - wynik XOR (1 dla nieparzystej liczby
    jedynek), odpowiada działaniu mod 2
y = np.array([[np.sum(row) % 2] for row in X])

# 3. Tworzenie zbioru treningowego - na nim model sieć
    będzie się uczyć oraz
#     im wynikom dla bramki XOR, a także wyodrębnienie
    pozostałej części zbioru jako zbioru walidacyjnego i
    testowego

indices = np.arange(len(X))
train_ind = np.random.choice(indices, size=900,
    replace=False)

test_mask = np.ones(len(X), dtype=bool)
test_mask[train_ind] = False

X_train = X[train_ind]
y_train = y[train_ind]

X_test = X[test_mask]
y_test = y[test_mask]

X_val = X_test[:int(0.7*len(X_test))]
y_val = y_test[:int(0.7*len(y_test))]
```

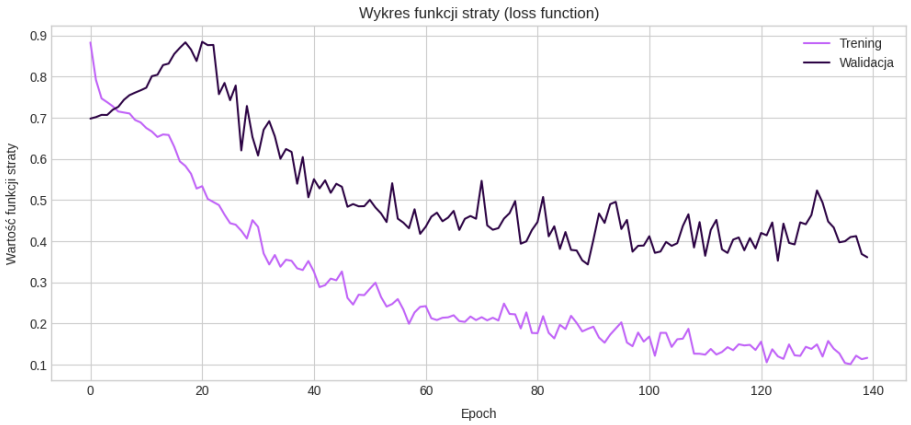
```
X_test = X_test[int(0.7*len(X_test)):]
y_test = y_test[int(0.7*len(y_test)):]

print(f"Rozmiar treningowy: {len(X_train)}")
print(f"Rozmiar testowy: {len(X_test)}")


# 4. Budowa wielowarstwowego modelu zbudowanego

model = Sequential()
model.add(Input(shape=(10,)))
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.2))
model.add(Dense(256, kernel_initializer='he_uniform',
    activation='relu'))
model.add(Dense(128, kernel_initializer='he_uniform',
    activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.2))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, kernel_initializer='glorot_uniform',
    activation='sigmoid'))

callback =
    tf.keras.callbacks.EarlyStopping(monitor='val_loss',
    patience=50, restore_best_weights=True)

# 5. Kompilacja modelu

model.compile(optimizer=Adam(learning_rate=0.001,
    clipnorm=1.0), loss='binary_crossentropy',
    metrics=['accuracy'])

# 6. Trenowanie modelu na zestawie treningowym

model.fit(X_train, y_train, epochs=1500, verbose=0,
    batch_size=32,
                    validation_data=(X_val, y_val),
    callbacks=callback)

# 7. Ewaluacja modelu na zestawie testowym, sprawdzenie
    jego accuracy

_, accuracy = model.evaluate(X_test, y_test)
print(f"Accuracy: {accuracy * 100:.2f}%")

# 8. Predykcje

pred_rest = model.predict(X_test)
```

```
predictions = np.round(pred_rest).astype(int)

print("Predictions:")
for i in range(len(X_test)):
    print(f"Input: {X_test[i]} => Predicted Output:
    {predictions[i]}, Actual Output: {y_test[i]}")
```

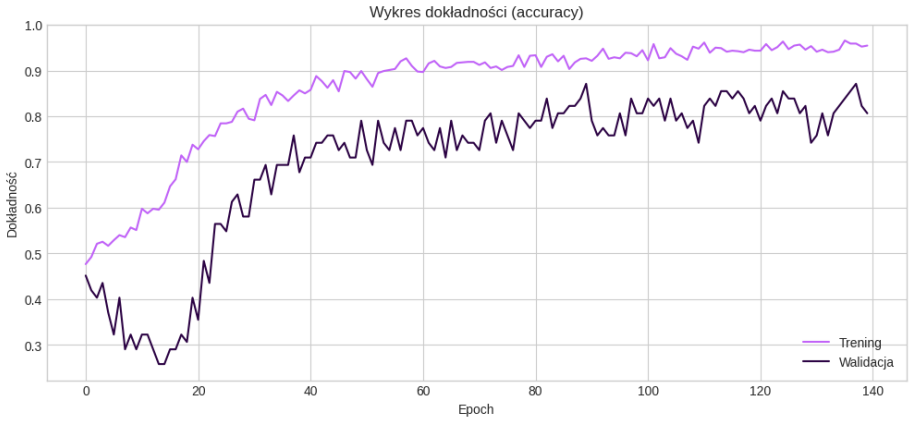
3.4 Rezultaty

Model był trenowany na zbiorze treningowym stworzonym z losowo wybranych 900 kombinacjach 10-elementowych o elementach 0 i 1. Pozostałe podzielono na zbiór walidacyjny i testowy (70 procent do walidacji i 30 procent do testu). Osiągane w różnych próbach rezultaty skutkowały dokładnością między 70 a 90 procent, co stanowi bardzo dobry wynik dla problemu o tej złożoności i skrajnej nieliniowości. Konstrukcja modelu pozwala na przeprowadzenie eksperymentu w około minutę.

Wykres funkcji kosztu prezentuje względnie poprawne tendencje (jej wartości spadają zarówno przy teście, jak i treningu), ale zauważalne są silne oscylacje. Przyczyną tego zachowania jest fakt, że problem bramki XOR jest liniowo nieseparowalny i prosta sieć neuronowa nie potrafi dobrze przewidywać wartości. Jest na skraju między przeuczeniem i uczeniem się wartości na pamięć, a zgadywaniem, co nie pozwala na osiągnięcie maksymalnej dokładności (zjawisko prezentuje wykres dokładności, ang. accuracy).



Rysunek 1: Wykres funkcji straty



Rysunek 2: Wykres dokładności (accuracy)

4 Model predykcyjny do klasyfikacji ryzyka cukrzycy

4.1 Opis eksperymentu

Model predykcyjny został oparty na danych medycznych *The Pima Indian Diabetes Dataset* [15], który jest zbiorem danych pochodzącym z National Institute of Diabetes and Digestive and Kidney Diseases. Zawiera on dane 768 kobiet zamieszkujących obszar w pobliżu miasta Phoenix (Arizona, USA), z czego 258 choruje na cukrzycę a 500 jest zdrowych.

4.2 Przygotowanie danych

Występujące w zbiorze braki danych zostały uzupełnione medianą (wyznaczoną niezależnie dla grupy osób chorych i zdrowych). Został dodany szum losowy, do każdej uzupełnionej danej, by zapobiec wyłapaniu przez model sztucznych zależności między osobami z identycznymi wynikami. Zbiór zawiera dane w różnych skalach, zapobiegając faworyzowaniu zmiennych o wyższych wartościach, zostały one poddane standaryzacji za pomocą metody StandardScaler. Podział na zbiory treningowy, walidacyjny i testowy został przeprowadzony losowo (70% część treningowa, 15% walidacyjna, 15% testowa).

4.3 Budowa modelu

Architektura sieci neuronowej została oparta na 8 cechach klinicznych pacjentek (wieku, BMI, glukozy, ciśnienia krwi, liczby ciąż, historii rodzinnej, insuliny i grubości skóry), z 4 warstwami uczącymi i 2 operacjami pomocniczymi. Standardem dla danych medycznych jest wybór funkcji aktywacji ReLU, która dla wartości ujemnych zwraca zero, a wartości dodatnie przepuszcza bez zmian oraz niemającej górnego limitu – doskonale radzi sobie z interpretacją skrajnie wysokich wyników laboratoryjnych, które są kluczowe w diagnozie chorób. W celu stabilizacji i optymalizacji procesu uczenia się zostały dodane operacje Dropout (która losowo wyrzuca dany procent neuronów, by zapobiec tzw. overfittingowi) oraz BatchNormalization zapewniający, że każda kolejna warstwa dostanie dane o podobnej strukturze, co zapobiega traceniu czasu na dopasowywanie danych. Dodatkowo zastosowano mechanizm EarlyStopping. Monitoruje on błąd na zbiorze testowym i cofa model do momentu, w którym był on najskuteczniejszy.

4.4 Część programistyczna

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout,
    BatchNormalization
```

```
from tensorflow.keras.optimizers import Adam
import tqdm
import pandas as pd
# sam losowo wybiera dane do trenowania i sprawdzania
from sklearn.model_selection import train_test_split
# dla standardowego skalowania danych
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt

# 1. Wczytanie danych medycznych
df = pd.read_csv("https://raw.githubusercontent.com/
npradaschnor/Pima-Indians-Diabetes-Dataset/refs/heads/master
/diabetes.csv")
columns = ['Age', 'BMI', 'Glucose', 'BloodPressure',
    'Outcome', 'Pregnancies','DiabetesPedigreeFunction',
    'Insulin', 'SkinThickness']
df = df[columns]

# 2. Przygotowanie danych
columns_with_zeros = ['BMI', 'Glucose', 'BloodPressure',
    'Insulin', 'SkinThickness']

for column in columns_with_zeros:
    df[column] = df[column].replace(0, np.nan)

# mamy dane 768 osób, uzupełniamy brakujące dane medianą
    dla grup (osobno dla zdrowych i chorych), dodając do
    każdej danej szum losowy
# clip zamienia wartosci na dodatnie
# std pozwala na dodanie szumu niewiększego od 5% rozrzutu
    dla konkretnych danych

for col in columns_with_zeros:
    medians = df.groupby('Outcome')[col].transform('median')
    noise = np.random.normal(0, df[col].std() * 0.05,
    size=len(df))
    df[col] = df[col].fillna(medians + noise).clip(lower=0)

X = df.drop('Outcome', axis=1).values
y = df['Outcome'].values

# 3. Podział na zbiór treningowy, walidacyjny i testowy
    (70% trening, 15% walidacyjny, 15% test)
X_train, X_temp, y_train, y_temp = train_test_split(X, y,
    test_size=0.3, random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp,
    y_temp, test_size=0.5, random_state=42, stratify=y_temp)
```

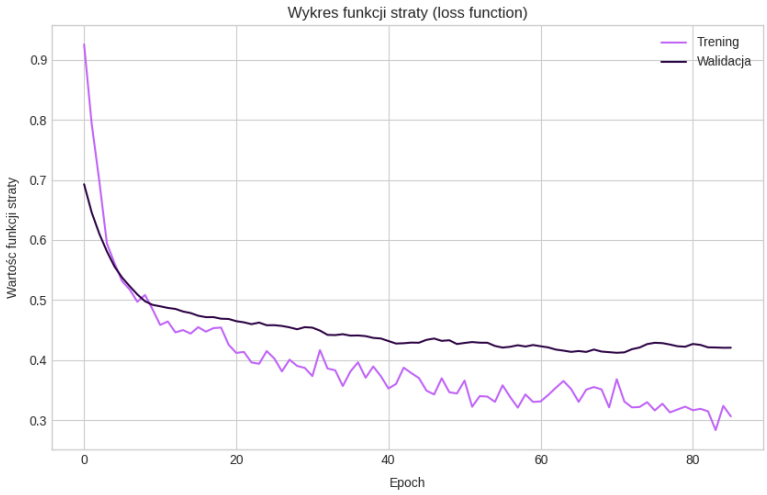
```
# skalowanie danych, by nie przypisać im niepotrzebnie
    dużych wag
skala = StandardScaler()

X_train_scaled = skala.fit_transform(X_train)
X_test_scaled = skala.transform(X_test)
X_val_scaled = skala.transform(X_val)


print(f"Rozmiar treningowy: {len(X_train)}")
print(f"Rozmiar testowy: {len(X_test)}")
print(f"Rozmiar walidacyjny: {len(X_val)}")


# 4. Budowa modelu zbudowanego z 3 warstw
# trzeba zmienić liczbę wejść, by model nie uczył się na
    pamięć danych (overfitting)
# tylko znalazł jakieś tendencje
# mamy 8 zmiennych do nauki
# zostawiamy tylko funkcje aktywacji 'relu' - standard dla
    danych medycznych

model = Sequential()

model.add(Dense(32, input_dim=8, activation='relu'))
# stabilizacja wag
model.add(BatchNormalization())

# losowo wyrzuca 20% neuronów podczas nauki, by zapobiec
    overfittingowi
model.add(Dropout(0.2))

model.add(Dense(16, activation='relu'))

# generuje wyniki z zakresu [0,1], które interpretujemy
    jako prawdopodobieństwo wystąpienia cukrzycy
model.add(Dense(1, activation='sigmoid'))

# 5. Kompilacja modelu

# obniżenie learning rate dla większej precyzji uczenia
model.compile(optimizer=Adam(learning_rate = 0.0005),
    loss='binary_crossentropy', metrics=['accuracy'])

# wcześniejsze zatrzymanie zapobiega przeuczeniu
# jeżeli przez 15 cykli nauki błąd na zbiorze testowym nie
    spadnie
# cofamy się do momentu, w którym sieć była
    najskuteczniejsza
```

```
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=15,
    restore_best_weights=True
)

# 6. Trenowanie modelu na zestawie treningowym

model.fit(X_train_scaled, y_train,
          epochs=500,
          batch_size=16,
          verbose = 0,
          validation_data = (X_val_scaled, y_val),
          callbacks=[early_stop]
          )

# 7. Ewaluacja modelu na zestawie testowym, sprawdzenie
    jego dokładności
_, accuracy = model.evaluate(X_test_scaled, y_test)
print(f"Accuracy: {accuracy * 100:.2f}%")

# 8. Predykcje - model próbuje przewidzieć kolejne 15%

pred_rest = model.predict(X_test_scaled)
predictions = np.round(pred_rest).astype(int)

print("Predictions:")
for i in range(len(X_test)):
  print(f"Prawdopodobieństwo cukrzycy: {pred_rest[i][0] *
    100:.2f}%")
```

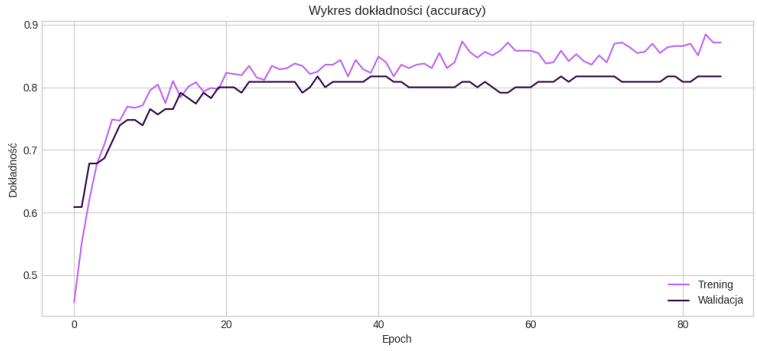
4.5 Rezultaty

Dokładność modelu oscyluje między 80% – 90%. Jest to zadowalający wynik dla skomplikowanych danych medycznych oraz próby relatywnie niewielkiego rozmiaru. Dzięki zastosowanej konstrukcji modelu i operacji pomocniczych nauka modelu zajmuje ok. 20 sekund.

Malejąca funkcja straty (na rysunku 1) wskazuje na to, że uczy on się poprawnie, unikając tzw. overfittingu (uczenia się na pamięć), a rosnąca dokładność (rysunek 2) potwierdza, że model skutecznie wychwytuje zależności pomiędzy danymi i z wysokim prawdopodobieństwem jest w stanie postawić poprawną diagnozę.



Rysunek 3: Wykres funkcji straty



Rysunek 4: Wykres dokładności (accuracy)

5 Model prognozy pogody

5.1 Opis eksperymentu

Prognozowanie pogody stanowi duże wyzwanie przez liczbę elementów składowych, które ją tworzą. Chociaż początkowo próbowano estymować parametry pogodowe metodami liniowymi, szybko okazało się, że ich trafność nie jest najlepsza w związku z nieliniową naturą połączeń między różnymi czynnikami ją tworzącymi.

Eksperyment polega na próbie zaprognozowania pogody na podstawie danych pogodowych i atmosferycznych pobranych w stacji pogodowej Instytutu Maxa Plancka [16]. Zestaw danych zawiera 20 różnych parametrów pogodowych zbieranych w 10-minutowych odstępach od 1.01.2020 do 1.01.2021 roku. W eksperymencie skupimy się na 3 z nich: ciśnieniu atmosferycznym (p, mierzonej w mbar), temperaturze powietrza (T, mierzonej w $^{\circ}C$) oraz względnej wilgotności (rh, wyrażanej w procentach, które mówią o ilości wilgoci w powietrzu w odniesieniu do maksimum wilgoci możliwej do utrzymania w danej temperaturze).

5.2 Przygotowanie danych

Obszerność i złożoność danych jest bardzo duża, stąd celem tego eksperymentu jest predykcja trzech wybranych parametrów. Ze względu na ich różną strukturę i zakres wartości, dane zostały poddane skalowaniu za pomocą narzędzia StandardScaler. Bierzemy pod uwagę dane o dłuższym interwale czasowym - co trzeci pomiar, czyli co pół godziny. Dane dzielimy na zbiory treningowy, walidacyjny i testowy. Ponadto do predykcji tworzymy za pomocą funkcji okna czasowe. Mają one na celu stworzenie ramek danych z pomiarami z 6 dni i odpowiadającym im wartościom z kolejnego 1,5 dnia, by model miał dane o strukturze 'dane-etykieta' do nauki.

5.3 Architektura

W części programistycznej przedstawiony jest model sieci neuronowej stworzonej w oparciu o warstwę GRUs (ang. Gated Recurrent Units), które są wariantem rekurencyjnych sieci neuronowych (RNN), często wykorzystywanych w problemach szeregów czasowych. Cechują się wysoką efektywnością w przetwarzaniu danych sekwencyjnych, co koresponduje z problemem przedstawionym w eksperymencie - prognozowania na podstawie szeregu czasowego. Jest to warstwa o podobnym, chociaż uproszczonym (przez mniejszą liczbę parametrów) do LSTM (ang. Long Short-Term Memory), które są skuteczne w zapobieganiu problemowi znikającego gradientu i umożliwiają na tworzenie zależności w większej perspektywie. [17] Warstwa składa się z dwóch bramek - reset i update. Pierwsza z nich determinuje zależność przeszłej informacji do ustalenia nowego stanu i dla wartości bliskiej zero, zapomina poprzedni stan ukryty. Druga ustala ilość informacji niezbędnej do przekazania kolejnemu stanowi ukrytemu do wyliczonego kandydata na

nowy stan ukryty - dla wartości bliskiej 1 duża jej część jest przekazywana, w przeciwnym przypadku wyliczony kandydat pozostaje niezmieniony.

Pozostałe elementy mają podobne zastosowanie jak w przypadku poprzednich eksperymentów.

5.4 Część programistyczna

```
import numpy as np
import tensorflow as tf
import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Input,
    Flatten, Reshape, Dropout, GlobalAveragePooling1D,
    RepeatVector, TimeDistributed, GRU
from tensorflow.keras.losses import Huber
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler,
    MinMaxScaler
import tqdm
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

#---data processing---

df = pd.read_csv('cleaned_weather.csv', index_col='date')

df_shrink = df.iloc[::3, ].copy()
df_shrink.info()

data = df_shrink[['p','T','rh']]

skala = StandardScaler()

# podział danych na zestaw treningowy i testowy

split = int(len(data)*0.7)
split_val = int(len(data)*0.85)

train_data = data.iloc[:split]
val_data = data.iloc[split:split_val]
test_data = data.iloc[split_val:]

scaled_train_vals = skala.fit_transform(train_data.values)
scaled_train_data = pd.DataFrame(scaled_train_vals,
    columns=train_data.columns, index=train_data.index)
```

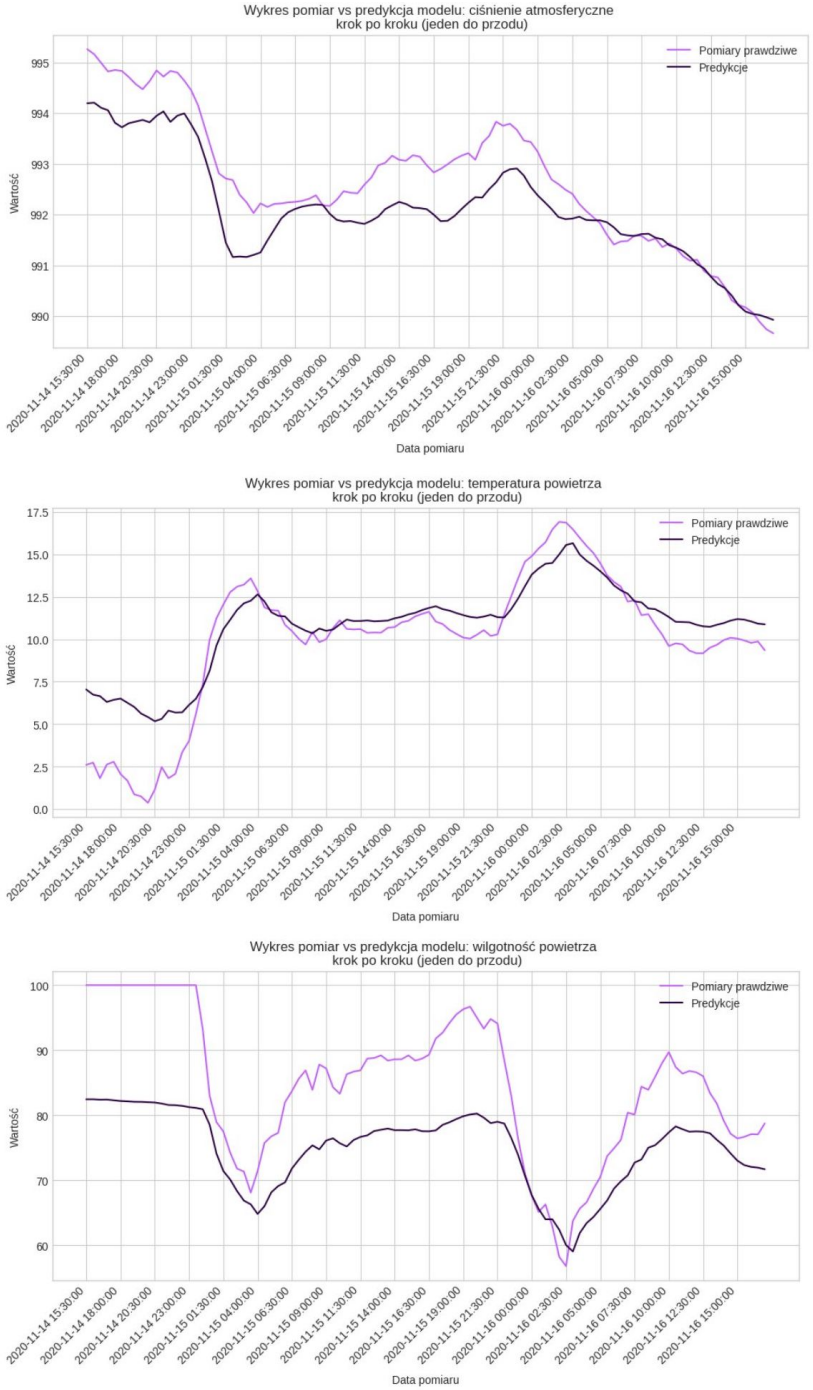
```
scaled_test_vals = skala.transform(test_data.values)
scaled_test_data = pd.DataFrame(scaled_test_vals,
    columns=test_data.columns, index=test_data.index)

scaled_val_vals = skala.transform(val_data.values)
scaled_val_data = pd.DataFrame(scaled_val_vals,
    columns=val_data.columns, index=val_data.index)

# funkcja tworząca okna czasowe

def create_window(data, window_size, pred_size):

    x, y = [], []

    for i in range(len(data)-window_size-pred_size-1):

        x.append(data[i : i + window_size])
        y.append(data[i + window_size : i + window_size +
    pred_size])

    return np.array(x), np.array(y)

window_size = 288
pred_size = 72

x_train, y_train = create_window(scaled_train_vals,
    window_size, pred_size)
x_test, y_test = create_window(scaled_test_vals,
    window_size, pred_size)
x_val, y_val = create_window(scaled_val_vals, window_size,
    pred_size)

print("X_train shape: ", x_train.shape)
print("Y_train shape: ", y_train.shape)
print("X_test shape: ", x_test.shape)
print("Y_test shape: ", y_test.shape)

model = Sequential([
  Input(shape=(x_train.shape[1], x_train.shape[2])),
  GRU(units=128, return_sequences=False),
  Dropout(0.2),
  RepeatVector(pred_size),
  GRU(units=128, return_sequences=True),
  Dropout(0.2),
  TimeDistributed(Dense(x_train.shape[2],
    activation='linear'))
])

callback =
    tf.keras.callbacks.EarlyStopping(monitor='val_loss',
```

```
    patience=2, restore_best_weights=True)

model.compile(optimizer=Adam(learning_rate=0.001),
    loss=Huber(), metrics=['mae'])

model.summary()

model.fit(x_train, y_train, epochs=10, batch_size=128,
    validation_data=(x_val, y_val), callbacks=callback)

predictions = model.predict(x_test)

n, m, p = predictions.shape
predictions_2d = predictions.reshape(-1, p)
pred_transformed = skala.inverse_transform(predictions_2d)
predictions = pred_transformed.reshape(n, m, p)

y_test_2d = y_test.reshape(-1, p)
y_test_transformed = skala.inverse_transform(y_test_2d)
y_test = y_test_transformed.reshape(n, m, p)
```

5.5 Rezultaty

Problem wieloskładnikowej prognozy pogody stanowi duże wyzwanie. Różnorodność parametrów utrudnia sieci neuronowej poprawne predykcje, co widać na załączonych wykresach. Przedstawiają one prognozy jednokrokowe - przewidywanie tylko kolejnego kroku na podstawie "czasowego okna danych"(porównanie punktowe) oraz zamodelowaną prognozę na 1.5 dnia do przodu dla pewnego przedziału testowego dla każdego z trzech rozważanych parametrów. Najlepsze rezultaty osiąga dla ciśnienia atmosferycznego oraz temperatury powietrza, trochę gorzej modeluje wilgotność powietrza. Można wnioskować, że jest to wynik tego, że pierwsze dwa składniki mają mniej drastyczne zmiany i są prostsze do estymacji. Wyniki RMSE pozwalają stwierdzić, że relatywnie prosta sieć neuronowa potrafi w pewnym stopniu miarodajnie przewidywać przyszłe wartości parametrów.

- RMSE dla ciśnienia atmosferycznego: 5.5947 (w mbar)
- RMSE dla temperatury: 3.2839 (w $^{\circ}C$)
- RMSE dla wilgotności powietrza: 14.2938 (w %).



Rysunek 5: Wykres porównawczy wartości punktowych

Rysunek 6: Wykres porównawczy wartości punktowych

6 Podsumowanie eksperymentów

Eksperymenty na prostych sieciach neuronowych wykazały różne problemy ze zbiorami danych czy architekturą modeli. Problem z nieliniowością funkcji xor okazał się sporym wyzwaniem dla sieci, a różnorodność parametrów, zarówno w modelu predykcyjnym pogody jak i cukrzycy utrudniał poprawne zrozumienie zależności. Proste modele są w stanie nauczyć się wielu rzeczy z zadowalającą dokładnością, a dzięki funkcjom optymalizującym ich kompilacja nie zajmuje wiele czasu. Trzeba jednak wiedzieć, że mają one swoje ograniczenia, a skomplikowane problemy wymagają bardziej zaawansowanych struktur.

Bibliografia

[1] *A Logical Calculus of the Ideas Immanent in Nervous Activity*. URL: https://en.wikipedia.org/wiki/A_Logical_Calculus_of_the_Ideas_Immanent_in_Nervous_Activity.

[2] Minsky i Papert. *Perceptrons*. URL: https://rodsmith.nz/wp-content/uploads/Minsky-and-Papert-Perceptrons.pdf.

[3] Maciej Szaleniec i Ryszard Tadeusiewicz. *LEKSYKON SIECI NEURONOWYCH [Lexicon on Neural Networks]*. URL: https://www.researchgate.net/publication/294578763_LEKSYKON_SIECI_NEURONOWYCH_Lexicon_on_Neural_Networks#pf34.

[4] *Wikipedia : Perceptron*. URL: https://pl.wikipedia.org/wiki/Perceptron.

[5] *Wikipedia: Neuron McCullocha-Pittsa*. URL: https://pl.wikipedia.org/wiki/Neuron_McCullocha-Pittsa.

[6] *Wikipedia: Funkcja aktywacji*. URL: https://pl.wikipedia.org/wiki/Funkcja_aktywacji.

[7] *Wikipedia: Podział zbiorów*. URL: https://pl.wikipedia.org/wiki/Zbi%C3%B3r_ucz%C4%85cy,_walidacyjny_i_testowy.

[8] *Binary Cross Entropy/Log Loss for Binary Classification*. URL: https://www.geeksforgeeks.org/deep-learning/binary-cross-entropy-log-loss-for-binary-classification/.

[9] *Probabilistic losses - dokumentacja Keras 3 API*. URL: https://keras.io/api/losses/probabilistic_losses/.

[10] *Regression losses - dokumentacja Keras 3 API*. URL: https://keras.io/api/losses/regression_losses/.

[11] *Loss Functions in Machine Learning Explained*. URL: https://www.datacamp.com/tutorial/loss-function-in-machine-learning.

[12] *Wikipedia: Vanishing gradient problem*. URL: https://en.wikipedia.org/wiki/Vanishing_gradient_problem.

[13] *How neural networks solve the xor problem*. URL: https://www.geeksforgeeks.org/artificial-intelligence/how-neural-networks-solve-the-xor-problem/.

[14] *Wikipedia: ReLU*. URL: https://pl.wikipedia.org/wiki/ReLU.

[15] J. W. Smith i in. *Pima Indians Diabetes Database*. 1988. URL: https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database.

[16] Max Planck Institute. *Wikipedia: Vanishing gradient problem*. URL: https://www.kaggle.com/code/fahmidachowdhury/lstm-time-series.

[17] *Gated Recurrent Units (GRUs)*. URL: https://apxml.com/courses/deep-learning-fundamentals-keras/chapter-5-recurrent-neural-networks-rnns/gated-recurrent-units-gru.