POLITECHNIKA WARSZAWSKA

Wydział Elektroniki i Technik Informacyjnych

Sieci Neuronowe w Zastosowaniach Biomedycznych

# 31.Przewidywanie skuteczność immunoterapii w leczeniu brodawek za pomocą sieci uczonej z nauczycielem (MLP)

Etap 2: ”Implementacja sieci neuronowej”

Mateusz Małkowski 321358

# Spis treści

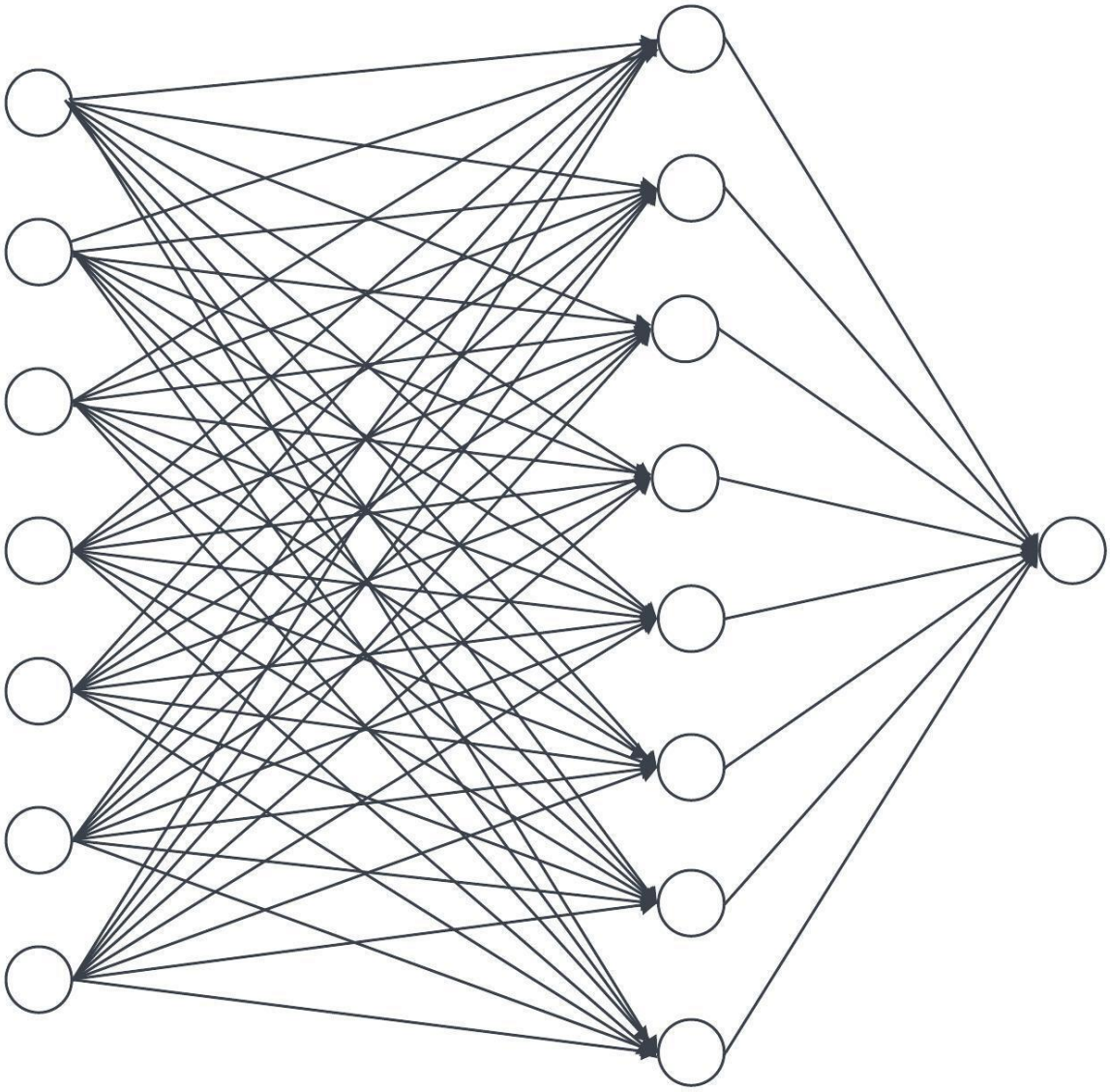
# Wstęp

Celem projektu było zaprojektowanie i zaimplementowanie sztucznej sieci neuronowej z nauczycielem do przewidywanie skuteczności immunoterapii w leczeni brodawek.

Dane wykorzystane do stworzenia sieci zostały opisane i przygotowane w poprzednim etapie projektu.

# Implementacja sieci neuronowej

Zgodnie z założeniami z poprzedniego etapu stworzono sieć z jedną warstwą ukrytą i po jednej warstwie wyjścia i wejścia.



*Rysunek 1:Schemat sieci neuronowej*

# Warstwa wejściowa

Składa się z 7 neuronów reprezentujących nasze dane wejściowe, czyli: Sex, Age, Time elapsed before treatment, Number of warts, types of wart, Surface area of the warts i induration diameter

of initial test. Wszystkie te dane zostały wcześniej przygotowane i znormalizowane do przedziału [0,1], więc nie trzeba ich modyfikować w tej warstwie.

## Warstwa ukryta

Podczas tworzenia warstwy ukrytej inicjalizujemy jej wartości początkowe sposobem Kaiminga He, czyli ustawiamy biasy na 0 i wagi jako liczbę losową z rozkładem prawdopodobieństwa Gaussa ze średnią 0 i odchyleniem standardowym $\sqrt{\frac{2}{n_l}}$

$$w_1 \sim \mathcal{N}(0, 2/n_l) \tag{1}$$

Metoda Kaiminga He jest jednym z podejść do zapewnienia odpowiednich wartości początkowych wag, szczególnie w przypadku funkcji aktywacji ReLU, która jest często stosowana ze względu na swoją skuteczność i prostotę obliczeniową. Powinna pomagać z problemem zanikających lub eksplodujących gradientów.

Do stworzenia sieci neuronowej jest potrzebna funkcja aktywacji, ponieważ bez niej wagi rosły by liniowo.

Używamy funkcji aktywacji ReLU (Rectified Linear Unit). Jest to funkcja nieliniowa opisana wzorem:

$$\begin{cases} 0 \; dla \; x < 0 \\ x \; dla \; x \geq 0 \end{cases} \tag{2}$$

Gdy dane wejściowe nie są większe od zera funkcja ReLU nie wprowadza żadnych zmian (jedynie przepuszcza pozytywne wartości), co prowadzi do skutecznej eliminacji problemu zanikającego gradientu.

Jest ona prosta obliczeniowo, co przyspiesza proces uczenia się sieci neuronowej i dla większości problemów działa bardzo dobrze.

## Warstwa wyjściowa

W warstwie wyjściowej stosujemy inicjalizacje Xawiera. Metoda ta zakłada inicjalizowanie wag w warstwach sieci neuronowej zgodnie z rozkładem normalnym o średniej zero i wariancji:

$$Var(w) = \frac{2}{n_{in}+n_{out}} \tag{3}$$

gdzie $n_{in}$ to liczba neuronów w warstwie poprzedniej, a $n_{out}$ to liczba neuronów w warstwie obecnej. Następnie wagi są losowane z tego rozkładu.

Istotą tej metody jest odpowiednie skalowanie wag początkowych w taki sposób, aby sygnał wejściowy do warstwy były mniej więcej tej samej wielkości, co sygnał wyjściowy. Ma to na celu zapobieżenie problemowi zanikającego lub eksplodującego gradientu podczas propagacji wstecznej w procesie uczenia się.

Funkcją aktywacji warstwy wyjściowej jest sigmoid, który opisany jest wzorem:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4}$$

Funkcja sigmoidalna przekształca dowolną wartość rzeczywistą na zakres od 0 do 1, co pozwala nam na określenie prawdopodobieństwa

Większość danych wejściowych była dla pacjentów dla których leczenie było sukcesem, dlatego sieć ma więcej wartości bliższych 1 niż 0. By sobie z tym poradzić korzystamy nie dzielimy

danych od prawdopodobieństwa 50%, tylko sami decydujemy jakie ustawiamy odcięcie przy inicjalizacji sieci.

## Funkcja straty

Jako funkcje straty używamy  binarnej entropii krzyżowe (ang. binary cross entropy). Mierzy różnicę między rzeczywistymi etykietami klas a prognozowanymi przez model prawdopodobieństwami należenia do klasy pozytywnej i opisana jest wzorem:

$$Loss = -(y(log(p) + (1 - y)(log(1 - p)) \quad (5)$$

Gdzie, $y$ to rzeczywista etykieta klasy (0 lub 1), a $p$ to prognozowane prawdopodobieństwo przynależności do klasy pozytywnej przez model.

Dobrze nadaje się do problemów klasyfikacji binarnej i poprzez uwzględnienie prawdopodobieństw przynależności do klasy, funkcja binarnej entropii krzyżowej pozwala na ocenę pewności predykcji modelu. Wartość funkcji straty będzie większa, gdy model będzie bardziej pewny swoich przewidywań i popełni błąd. Jest też łatwa obliczeniowo co przyśpiesza proces optymalizacji modelu.

Dla dobrze douczonego modelu jej wartości dla danych trenujących i testowych powinien być podobny i mały.

## Metoda propagacji wstecznej

Metoda propagacji wstecznej (ang. Backpropagation) jest fundamentalnym algorytmem używanym do uczenia się w sieciach neuronowych. Służy do obliczania gradientów funkcji straty względem wag sieci neuronowej, co umożliwia aktualizację wag w kierunku minimalizacji funkcji straty.

Jej kroki to:

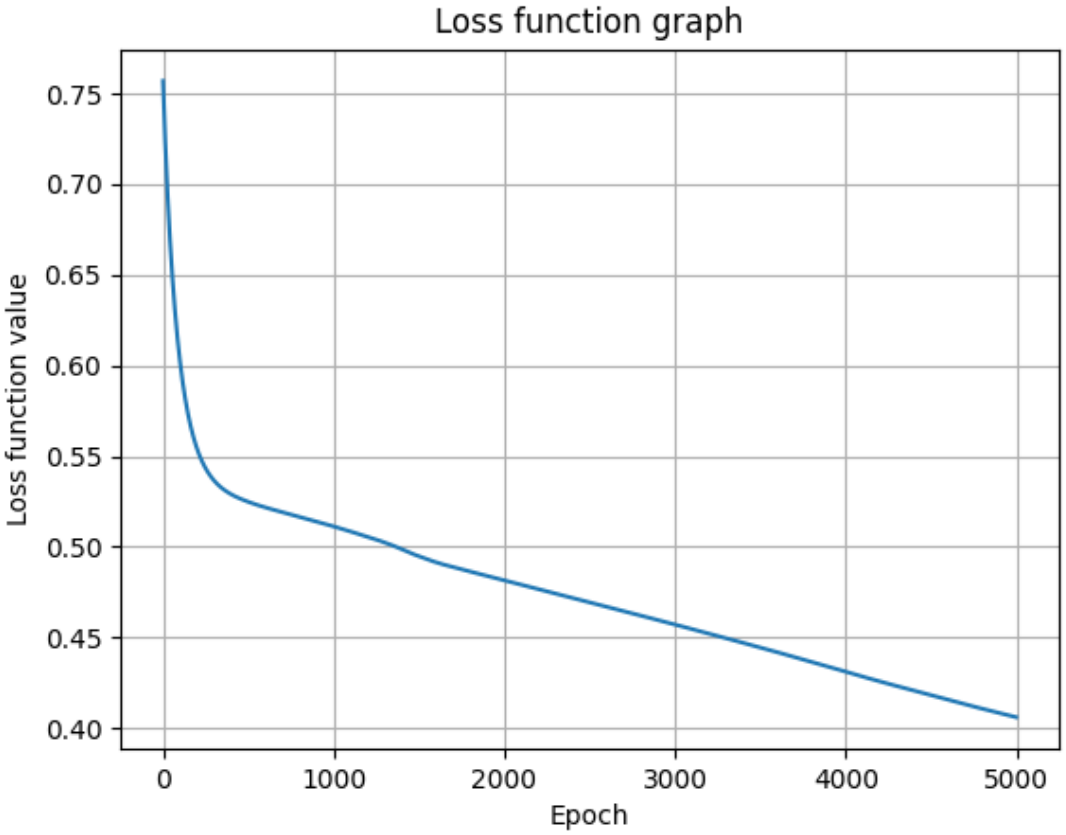
1. Przekazanie danych wejściowych: Na początku, dane wejściowe są przekazywane przez sieć neuronową, a każda warstwa oblicza swoje wyjście na podstawie aktualnych wag i funkcji aktywacji.
2. Obliczenie funkcji straty: Następnie, obliczana jest funkcja straty, która mierzy różnicę między rzeczywistymi etykietami a prognozowanymi przez sieć neuronową wartościami. Funkcja ta jest używana jako miara błędu sieci.
3. Propagacja wsteczna: W tej fazie, algorytm propagacji wstecznej oblicza gradient funkcji straty względem każdej wagi w sieci neuronowej, zaczynając od warstwy wyjściowej i przechodząc wstecz przez sieć.
4. Aktualizacja wag: Na podstawie obliczonych gradientów, algorytm stosuje regułę aktualizacji wag (np. metodę gradientu prostego) w celu dostosowania wartości wag w kierunku minimalizacji funkcji straty.
5. Powtarzanie procesu: Proces propagacji wstecznej i aktualizacji wag jest powtarzany aż do momentu, gdy sieć neuronowa osiągnie akceptowalny poziom dokładności lub zbiegnie do minimum funkcji straty.

Główne zalety metody propagacji wstecznej to jej skuteczność w uczeniu się z danych, zdolność do uczenia się na dużej skali i możliwość stosowania w różnych architekturach sieci neuronowych.

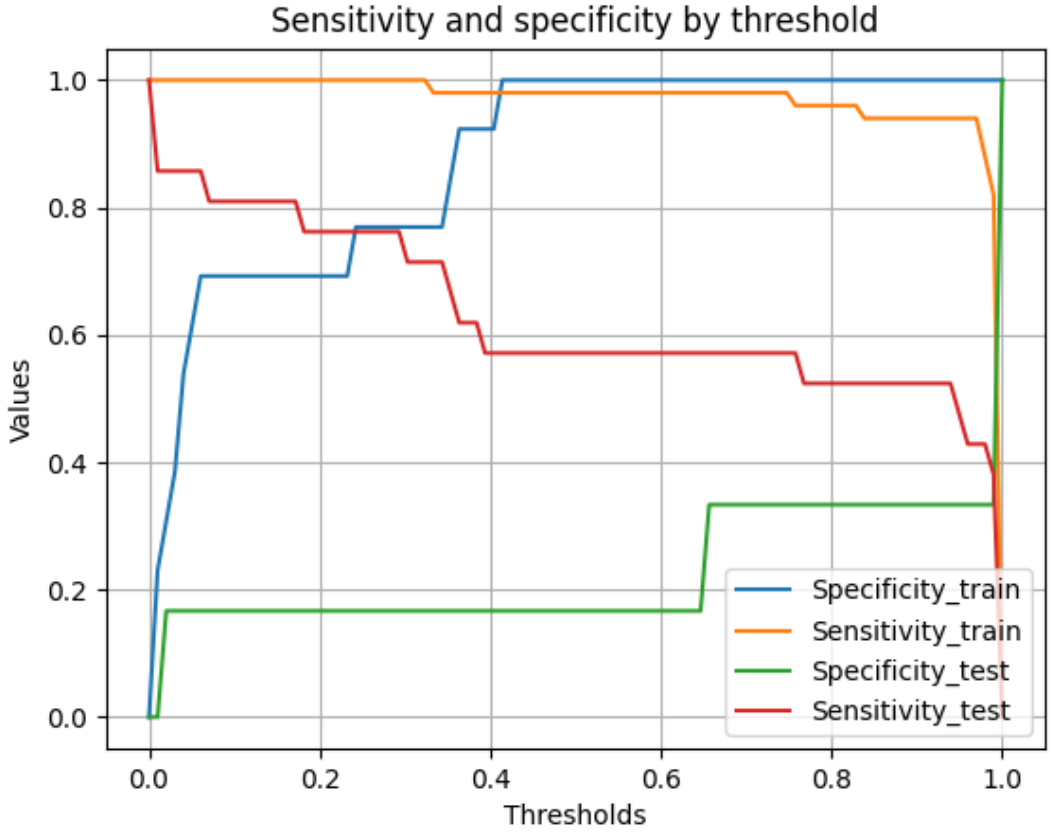
# Wyniki

*Tabela 1: Wyniki dla 1000 epok i współczynnika uczenia 0.1*

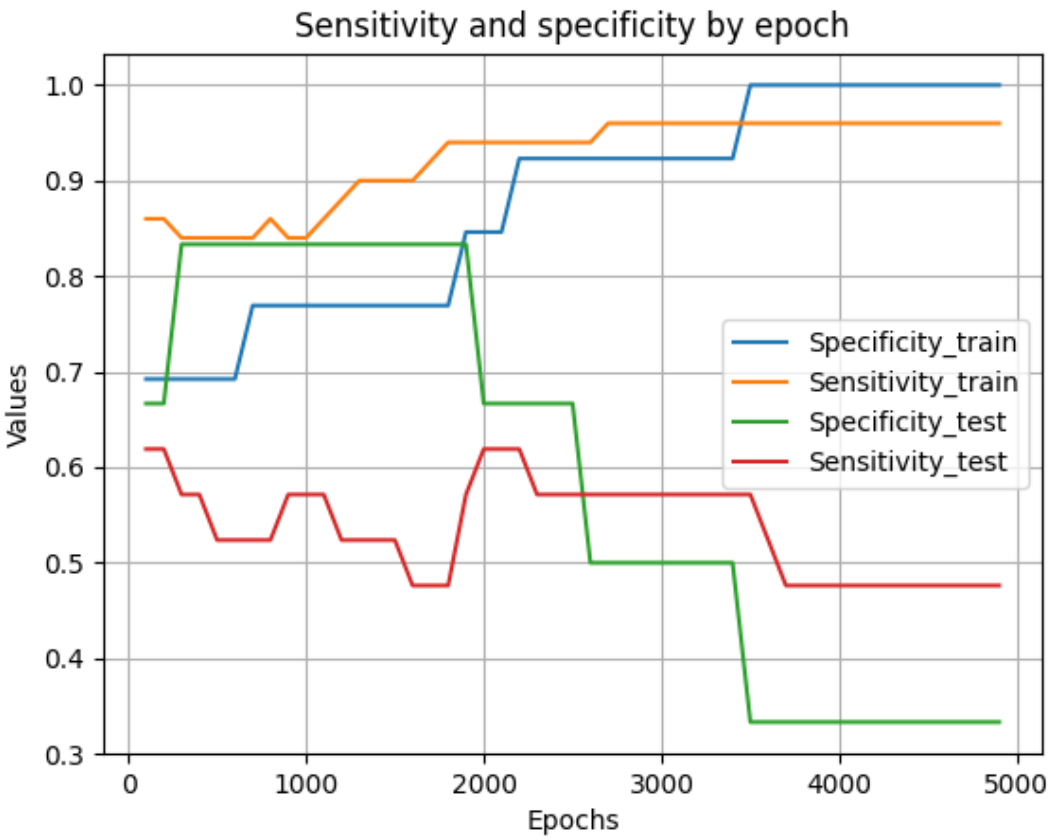
| Dla 1000 epok i współczynnika uczenia 0.1 | | |
|---|---|---|
| | Dane trenujące | Dane testowe |
| Czułość | 0,92 | 0,57 |
| Specyficzność | 0,76 | 0,83 |
| funkcja straty | 0,313 | 0,528 |




*Rysunek 2:Wartość funkcji straty do ilości epok*



*Rysunek 3:Czułość i specyficzność do wartości odcięcia*




*Rysunek 4:Czułość i specyficzność do ilości epok przez które rozwijał się algorytm*

# Wnioski

W typ projekcie stworzono sieć z jedna ukrytą warstwą, ale kod został przygotowany w taki sposób, by można było przeanalizować wynik w innej architekturze sieci neuronowej w kolejnym etapie.

Wyniki z prostej sieci jednowarstwowej były mało zadowalające. Z tabeli 1 można zauważyć, że czułość dla danych testowych była mało zadowalająca.  Wartość specyficzności była zdecydowanie lepsza. Jest to spowodowane mniejszą ilością wartości negatywnych w zbiorze danych. Dla danych testowych i trenujących wartość funkcji straty była całkiem mała, ale jednak większa dla trenujących, co pokazuje nam lekkie przetrenowanie sieci.

Mimo to, dla 1000 iteracji i współczynnika uczenia 0.1 osiągaliśmy najlepsze wyniki.

Z rysunku 1 można wywnioskować, że najważniejsze dla algorytmu jest pierwsze 500 epok podczas których wartość funkcji straty ostro malała. Dla kolejnych epok jest spadek nie był tak gwałtowny.

Na rysunku 3 widać, że czułości i specyficzności przy odcięciu 0,7 były bardziej miarodajne niż te dla odcięcia 0,5.

Z rysunku 4 widać, że po 2000 epok sieć była przetrenowana i specyficzność i czułość dla danych ostro spadała.

Duży wpływ na wyniki miał sposób inicjalizacji wag i biasów. W kolejnym etapie projektu trzeba będzie sprawdzić jak sieć będzie się zachowywała dla innych metod.

## Listing programów

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as
plt
import contextlib
import io

def
load_data(filename_X_train:str,
filename_X_test:str,
filename_Y_train:str,
filename_Y_test:str):
    """
    Load data from CSV files.

    Args:
        filename_X_train (str):
File path for training data
features.
        filename_X_test (str):
File path for test data features.
        filename_Y_train (str):
File path for training data
labels.
        filename_Y_test (str):
File path for test data labels.

    Returns:
        tuple: Tuple containing
X_train, X_test, Y_train, Y_test.
    """
    X_train =
pd.read_csv(filename_X_train)
    X_test =
pd.read_csv(filename_X_test)
    Y_train =
pd.read_csv(filename_Y_train)
    Y_test =
pd.read_csv(filename_Y_test)
    X_train = np.array(X_train).T
    X_test = np.array(X_test).T
    Y_train = np.array(Y_train).T
    Y_test = np.array(Y_test).T
    return X_train, X_test,
Y_train, Y_test

def relu(Z: np.array):
    """ReLU activation
function."""
    return np.maximum(Z, 0)

def sigmoid(Z: np.array):
    """Sigmoid activation
function."""
    return 1/(1 + np.exp(-Z))

def relu_deriv(Z: np.array):
    """Derivative of ReLU
activation function."""
    return Z > 0

class Hidden_Layer:
    """Hidden layer of the neural
network."""
    def __init__(self, n_inputs:
int, n_neurons: int,
learning_rate: float, bias_offset
= 0):
        """
        Initialize hidden layer
parameters.

        Args:
            n_inputs (int): Number
of input features.
            n_neurons (int):
Number of neurons in the layer.
            learning_rate (float):
Learning rate for training.
            bias_offset (float,
optional): Offset for bias
initialization. Defaults to 0.5.
        """
        self.weights =
np.random.randn(n_neurons,
n_inputs) * np.sqrt(2 / n_inputs)
        self.bias =
np.zeros((n_neurons, 1)) -
bias_offset
        self.learning_rate =
learning_rate
        self.activation_function =
relu

self.activation_function_derivativ
e = relu_deriv

    def forward_propagation(self,
inputs: np.array):
        """
```

```
        Perform forward
propagation through the layer.

        Args:
            inputs (np.array):
Input data.

        """
        self.Z =
np.dot(self.weights, inputs) +
self.bias
        self.output =
self.activation_function(self.Z)

    def backward_propagation(self,
X: np.ndarray, W_next: np.ndarray,
dZ_next: np.ndarray):
        """
        Perform backward
propagation through the layer.

        Args:
            X (np.ndarray): Input
data.
            W_next (np.ndarray):
Weights of the next layer.
            dZ_next (np.ndarray):
Gradient from the next layer.

        Returns:
            tuple: Updated weights
and gradient for the previous
layer.
        """

        m = X.shape[1]
        W = self.weights

        dZ = np.dot(W_next.T,
dZ_next) *
self.activation_function_derivativ
e(self.Z)
        dW = np.dot(dZ, X.T)/m
        db = np.sum(dZ, axis=1,
keepdims=True)/m

        self.weights -=
self.learning_rate * dW
        self.bias -=
self.learning_rate * db
        return W, dZ


class Output_layer(Hidden_Layer):
    """Output layer of the neural
network."""
    def __init__(self, n_inputs:
int, n_neurons: int,
learning_rate: float, bias_offset
= 0):
        """Initialize output layer
parameters.

        Args:
            n_inputs (int): Number
of input features.
            n_neurons (int):
Number of neurons in the layer.
            learning_rate (float):
Learning rate for training.
            bias_offset (float,
optional): Offset for bias
initialization. Defaults to 0.
        """
        super().__init__(n_inputs,
n_neurons, learning_rate,
bias_offset)
        self.activation_function =
sigmoid
        limit = limit = np.sqrt(2
/ (n_inputs + n_neurons))
        self.weights =
np.random.uniform(-limit, limit,
(n_neurons, n_inputs))

    def backward_propagation(self,
Y: np.ndarray, X: np.ndarray):
        """Perform backward
propagation through the output
layer.

        Args:
            Y (np.ndarray): True
labels.
            X (np.ndarray): Input
data.

        Returns:
            tuple: Tuple
containing dW and dZ.
        """
        m = Y.shape[1]
        W = self.weights

        dZ = self.output - Y
        dW = np.dot(dZ, X.T) / m
        db = np.sum(dZ, axis=1,
keepdims=True) / m

        self.weights -=
self.learning_rate * dW
        self.bias -=
self.learning_rate * db

        return W, dZ


class Neural_Network:
    """Neural Network class."""
    def __init__(self,
hidden_layers: list, output_layer:
Output_layer, threshold = 0.7):
```

```
        """
        Initialize the neural
network.

        Args:
            hidden_layers (list):
List of hidden layers.
            output_layer
(Output_layer): Output layer of
the network.
            threshold (float,
optional): Threshold for binary
classification. Defaults to 0.7.
        """
        self.hidden_layers =
hidden_layers
        self.output_layer =
output_layer
        self.threshold = threshold

    def cross_entropy(self,
Y_true: np.array, Y_prediction:
np.array):
        """
        Calculate the binary
cross-entropy loss.

        Args:
            Y_true (np.array):
True labels.
            Y_prediction
(np.array): Predicted
probabilities.

        Returns:
            float: Cross-entropy
loss.
        """
        Y_prediction =
np.clip(Y_prediction, 1e-15, 1 -
1e-15)
        return -np.mean(Y_true *
np.log(Y_prediction) + (1 -
Y_true) * np.log(1 -
Y_prediction))

    def
calculate_sensitivity_specificity(
self, Y_true: np.array,
y_prediction: np.array):
        """
        Calculate sensitivity and
specificity.

        Args:
            Y_true (np.array):
True labels.
            y_prediction
(np.array): Predicted labels.

        Returns:
            tuple: Tuple
containing sensitivity and
specificity.
        """
        TP = np.sum((Y_true == 1)
& (y_prediction == 1))
        TN = np.sum((Y_true == 0)
& (y_prediction == 0))
        FP = np.sum((Y_true == 0)
& (y_prediction == 1))
        FN = np.sum((Y_true == 1)
& (y_prediction == 0))

        sensitivity = TP / (TP +
FN) if (TP + FN) > 0 else 0
        specificity = TN / (TN +
FP) if (TN + FP) > 0 else 0

        return sensitivity,
specificity
    def train(self,X: np.array, Y:
np.array, epochs: int):
        """
        Train the neural network.

        Args:
            X (np.array): Input
data.
            Y (np.array): True
labels.
            epochs (int): Number
of training epochs.
        """
        input = list()
        input.append(X)
        self.loss_history = list()

        for epoch in
range(epochs):
            for i in
range(len(self.hidden_layers)):

self.hidden_layers[i].forward_prop
agation(input[i])

input.append(self.hidden_layers[i]
.output)

self.output_layer.forward_propagat
ion(input[-1])

            loss =
Neural_Network.cross_entropy(self,
Y, self.output_layer.output)

self.loss_history.append(loss)

            W, dZ =
self.output_layer.backward_propaga
tion(Y, input[-1])
            for i in
```

```
reversed(range(len(self.hidden_layers))):
                W, dZ = self.hidden_layers[i].backward_propagation(input[i], W, dZ)

            if epoch % 100 == 0:
                print('Epoch ', epoch, ', Loss: ', loss)

    def predict(self, X: np.array):
        """
        Make predictions using the trained model.

        Args:
            X (np.array): Input data.

        Returns:
            np.array: Predicted labels.
        """
        input = list()
        input.append(X)

        for i in range(len(self.hidden_layers)):
            self.hidden_layers[i].forward_propagation(input[i])
            input.append(self.hidden_layers[i].output)
        self.output_layer.forward_propagation(input[-1])

        return self.output_layer.output

    def plot_loss(self):
        """Plot the loss function over epochs."""
        plt.plot(self.loss_history, linestyle='-')
        plt.title('Loss function graph')
        plt.xlabel('Epoch')
        plt.ylabel('Loss function value')
        plt.grid(True)


def test_network(X_train: np.array, X_test: np.array, Y_train: np.array, Y_test: np.array):
    """
    Test the neural network on training and test data.

    Args:
        X_train (np.array): Training data features.
        X_test (np.array): Test data features.
        Y_train (np.array): Training data labels.
        Y_test (np.array): Test data labels.
    """
    learning_rate = 0.01
    epochs = 5000

    layer1 = Hidden_Layer(7, 8, learning_rate, bias_offset=0)
    output_layer = Output_layer(8, 1, learning_rate)

    network = Neural_Network([layer1], output_layer, threshold=0.7)

    network.train(X_train, Y_train, epochs)

    predictions_train = network.predict(X_train)
    predictions_test = network.predict(X_test)

    loss_train = network.cross_entropy(Y_train, predictions_train)
    loss_test = network.cross_entropy(Y_test, predictions_test)

    predictions_train = (predictions_train[0,:]> network.threshold).astype(int)
    predictions_test = (predictions_test[0,:]> network.threshold).astype(int)

    sensitivity_train, specificity_train = network.calculate_sensitivity_specificity(Y_train, predictions_train)
    sensitivity_test, specificity_test = network.calculate_sensitivity_specificity(Y_test, predictions_test)

    print('\nTrain data:')
    print('Predictions: ', predictions_train)
    print('True values: ',
```

```
Y_train.astype(int)[0, :])
    print("Sensitivity: ",
sensitivity_train)
    print("Specificity: ",
specificity_train)
    print("loss function value: ",
loss_train)
    print('\nTest data:')
    print('Predictions: ',
predictions_test)
    print('True values: ',
Y_test.astype(int)[0, :])
    print("Sensitivity: ",
sensitivity_test)
    print("Specificity: ",
specificity_test)
    print("loss function value: ",
loss_test)
    network.plot_loss()

plot_specificity_sensitivity(X_tra
in, X_test, Y_train, Y_test,
network)
def
plot_specificity_sensitivity(X_tra
in: np.array, X_test: np.array,
Y_train: np.array, Y_test:
np.array, network:
Neural_Network):
    """
    Plot sensitivity and
specificity by epoch and
threshold.

    Args:
        X_train (np.array):
Training data features.
        X_test (np.array): Test
data features.
        Y_train (np.array):
Training data labels.
        Y_test (np.array): Test
data labels.
        network (Neural_Network):
Trained neural network.
    """
    specificity_history_train =
list()
    sensitivity_history_train =
list()
    specificity_history_test =
list()
    sensitivity_history_test =
list()
    specificity_history_train_th =
list()
    sensitivity_history_train_th =
list()
    specificity_history_test_th =
list()
    sensitivity_history_test_th =
list()

    for epoch in range(100, 5000,
100):
        f = io.StringIO()
        # Używamy
contextlib.redirect_stdout do
przekierowania stdout
        with
contextlib.redirect_stdout(f):
            network.train(X_train,
Y_train, epoch)

            predictions_train =
network.predict(X_train)
            predictions_test =
network.predict(X_test)
            predictions_train =
(predictions_train[0, :] >
network.threshold).astype(int)
            predictions_test =
(predictions_test[0, :] >
network.threshold).astype(int)

            sensitivity_train,
specificity_train =
network.calculate_sensitivity_spec
ificity(Y_train,
predictions_train)

sensitivity_history_train.append(s
ensitivity_train)

specificity_history_train.append(s
pecificity_train)

            sensitivity_test,
specificity_test =
network.calculate_sensitivity_spec
ificity(Y_test, predictions_test)

specificity_history_test.append(sp
ecificity_test)

sensitivity_history_test.append(se
nsitivity_test)

    epochs = range(100, 5000, 100)
    plt.figure()
    plt.plot(epochs,
specificity_history_train,
label='Specificity_train')
    plt.plot(epochs,
sensitivity_history_train,
label='Sensitivity_train')
    plt.plot(epochs,
specificity_history_test,
label='Specificity_test')
    plt.plot(epochs,
sensitivity_history_test,
label='Sensitivity_test')
```

```
    plt.xlabel('Epochs')
    plt.ylabel('Values')
    plt.grid(True)
    plt.title('Sensitivity and
specificity by epoch')
    plt.legend()

    thresholds = np.linspace(0, 1,
100)
    for threshold in thresholds:
        network.threshold =
threshold
        f = io.StringIO()
        # Używamy
contextlib.redirect_stdout do
przekierowania stdout
        with
contextlib.redirect_stdout(f):

            network.train(X_train,
Y_train, 1000)
            predictions_train =
network.predict(X_train)
            predictions_test =
network.predict(X_test)
            predictions_train =
(predictions_train[0, :] >
network.threshold).astype(int)
            predictions_test =
(predictions_test[0, :] >
network.threshold).astype(int)

            sensitivity_train,
specificity_train =
network.calculate_sensitivity_spec
ificity(Y_train,
predictions_train)

sensitivity_history_train_th.appen
d(sensitivity_train)

specificity_history_train_th.appen
d(specificity_train)

            sensitivity_test,
specificity_test =
network.calculate_sensitivity_spec
ificity(Y_test, predictions_test)

specificity_history_test_th.append
(specificity_test)

sensitivity_history_test_th.append
(sensitivity_test)

    plt.figure()
    plt.plot(thresholds,
specificity_history_train_th,
label='Specificity_train')
    plt.plot(thresholds,
sensitivity_history_train_th,
label='Sensitivity_train')
    plt.plot(thresholds,
specificity_history_test_th,
label='Specificity_test')
    plt.plot(thresholds,
sensitivity_history_test_th,
label='Sensitivity_test')
    plt.xlabel('Thresholds')
    plt.ylabel('Values')
    plt.grid(True)
    plt.title('Sensitivity and
specificity by threshold')
    plt.legend()
    plt.show()



if __name__ == '__main__':
    X_train, X_test, Y_train,
Y_test = load_data('X_train.csv',
'X_test.csv', 'Y_train.csv',
'Y_test.csv')

    test_network(X_train, X_test,
Y_train, Y_test)
```