

Sieci Neuronowe

Sprawozdanie z ćwiczenia 1-4

Technologia

Python 3.11, Jupyter Notebook.

Realizacja ćwiczeń

1. Analiza eksploracyjna zbioru danych

Cel ćwiczenia

Celem ćwiczenia jest wprowadzenie/przypomnienie narzędzi i zapoznanie się z danymi z których będziemy korzystać w dalszej części kursu do ewaluacji sieci neuronowych jako metody uczenia maszynowego.

Wybór zbioru danych

Zbiór danych pochodzi ze źródła: [Heart Disease - UCI Machine Learning Repository](https://archive.ics.uci.edu/ml/datasets/Heart+Disease)

Można było pobrać cały zbiór, który zawierał też dane z innych regionów, ale tak jak w informacji użyto przetworzonego zbioru „cleveland”. Co ciekawe przy porównaniu rzekomych tych samych zbiorów widać pewne różnice. Zatem końcowy postawiono na użycie kodu z „Import in python”.



```
Install the ucimlrepo package

pip install ucimlrepo

Import the dataset into your code

from ucimlrepo import fetch_ucirepo

# fetch dataset
heart_disease = fetch_ucirepo(id=45)

# data (as pandas dataframes)
X = heart_disease.data.features
y = heart_disease.data.targets

# metadata
print(heart_disease.metadata)

# variable information
print(heart_disease.variables)

View the full documentation
```

Jeżeli połączymy dane w pełen zbiór danych, możemy wpisać kilka pierwszych wierszy.

```
df = pd.concat([X, y], axis=1)
df.head(20)
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	num
0	63	1	1	145	233	1	2	150	0	2.3	3	0.0	6.0	0
1	67	1	4	160	286	0	2	108	1	1.5	2	3.0	3.0	2
2	67	1	4	120	229	0	2	129	1	2.6	2	2.0	7.0	1
3	37	1	3	130	250	0	0	187	0	3.5	3	0.0	3.0	0
4	41	0	2	130	204	0	2	172	0	1.4	1	0.0	3.0	0
5	56	1	2	120	236	0	0	178	0	0.8	1	0.0	3.0	0
6	62	0	4	140	268	0	2	160	0	3.6	3	2.0	3.0	3
7	57	0	4	120	354	0	0	163	1	0.6	1	0.0	3.0	0
8	63	1	4	130	254	0	2	147	0	1.4	2	1.0	7.0	2
9	53	1	4	140	203	1	2	155	1	3.1	3	0.0	7.0	1
10	57	1	4	140	192	0	0	148	0	0.4	2	0.0	6.0	0
11	56	0	2	140	294	0	2	153	0	1.3	2	0.0	3.0	0
12	56	1	3	130	256	1	2	142	1	0.6	2	1.0	6.0	2
13	44	1	2	120	263	0	0	173	0	0.0	1	0.0	7.0	0
14	52	1	3	172	199	1	0	162	0	0.5	1	0.0	7.0	0
15	57	1	3	150	168	0	0	174	0	1.6	1	0.0	3.0	0
16	48	1	2	110	229	0	0	168	0	1.0	3	0.0	7.0	1
17	54	1	4	140	239	0	0	160	0	1.2	1	0.0	3.0	0
18	48	0	3	130	275	0	0	139	0	0.2	1	0.0	3.0	0
19	49	1	2	130	266	0	0	171	0	0.6	1	0.0	3.0	0

Możemy sprawdzić typy danych.

```
df.info()

[25] ✓ 0.0s

... <class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
#   Column      Non-Null Count  Dtype
---  -
0    age         303 non-null    int64
1    sex         303 non-null    int64
2    cp          303 non-null    int64
3    trestbps    303 non-null    int64
4    chol        303 non-null    int64
5    fbs         303 non-null    int64
6    restecg     303 non-null    int64
7    thalach     303 non-null    int64
8    exang       303 non-null    int64
9    oldpeak     303 non-null    float64
10   slope       303 non-null    int64
11   ca          299 non-null    float64
12   thal        301 non-null    float64
13   num         303 non-null    int64
dtypes: float64(3), int64(11)
memory usage: 33.3 KB
```

Mimo, że niektóre kolumny są kategoriyczne to są zapisane w postaci liczbowej. Zbiór danych posiada 303 próbek i 14 kolumn w tym kolumna 'num' jest kolumną klasy próbki.

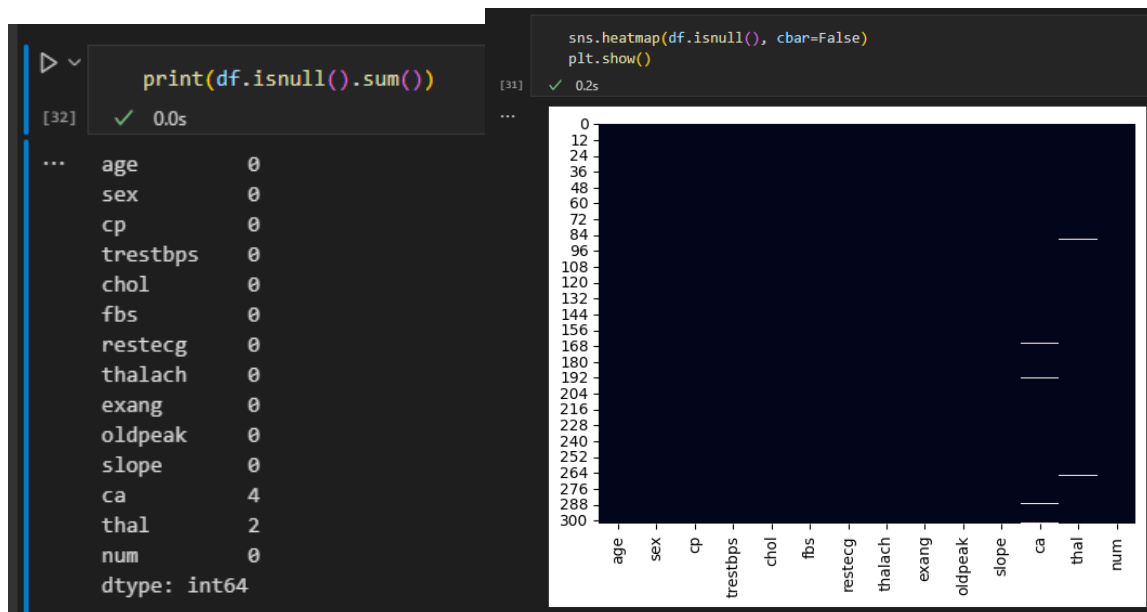
Możemy uzyskać podstawowe statystyki zbioru.

```
df.describe()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	num
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	299.000000	301.000000	303.000000
mean	54.438944	0.679868	3.158416	131.689769	246.693069	0.148515	0.990099	149.607261	0.326733	1.039604	1.600660	0.672241	4.734219	0.937294
std	9.038662	0.467299	0.960126	17.599748	51.776918	0.356198	0.994971	22.875003	0.469794	1.161075	0.616226	0.937438	1.939706	1.228536
min	29.000000	0.000000	1.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	1.000000	0.000000	3.000000	0.000000
25%	48.000000	0.000000	3.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	3.000000	0.000000
50%	56.000000	1.000000	3.000000	130.000000	241.000000	0.000000	1.000000	153.000000	0.000000	0.800000	2.000000	0.000000	3.000000	0.000000
75%	61.000000	1.000000	4.000000	140.000000	275.000000	0.000000	2.000000	166.000000	1.000000	1.600000	2.000000	1.000000	7.000000	2.000000
max	77.000000	1.000000	4.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	3.000000	3.000000	7.000000	4.000000

Teraz możemy zauważyć, że niektórych danych brakuje. Count(thal) = 301 oraz Count(ca) = 299

Czy występują cechy brakujące i jaką strategię możemy zastosować, żeby je zastąpić?



W przypadku cechy liczbowej 'ca' możemy wstawić średnią w miejsce braków. W przypadku kategorycznej cechy 'thal' możemy wstawić modalną. Byłoby to błędem w przypadku 'ca', ponieważ, mimo że to są liczby to są w przedziale liczb całkowitych 0-3 zatem wstawimy medianę.

```
median = df['ca'].median()
df['ca'].fillna(median, inplace=True)
mode_category = df['thal'].mode()[0]
df['thal'].fillna(mode_category, inplace=True)
```

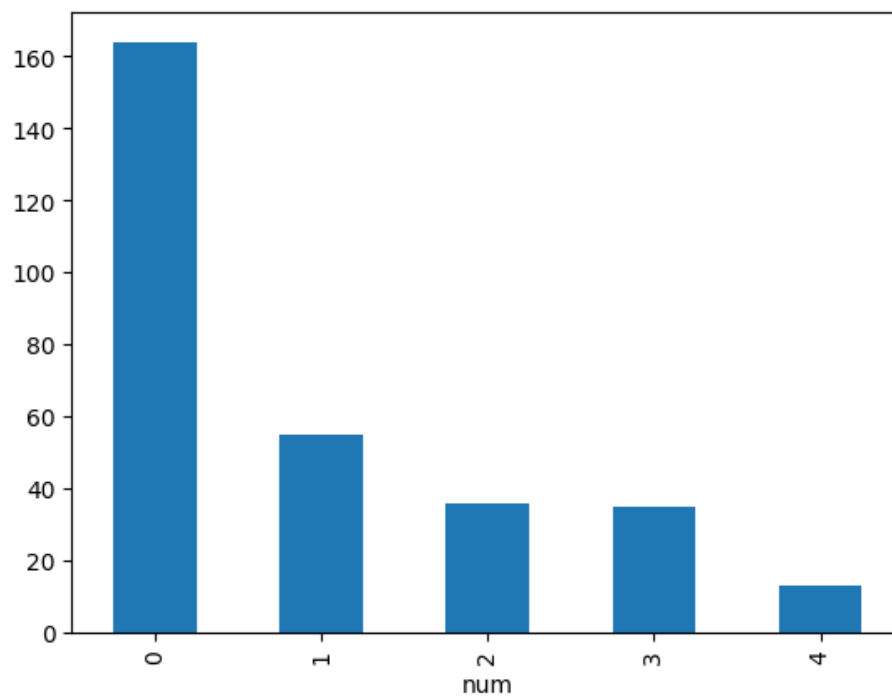
```
print(df.isnull().sum())
```

[62] ✓ 0.0s

...	age	0
	sex	0
	cp	0
	trestbps	0
	chol	0
	fbs	0
	restecg	0
	thalach	0
	exang	0
	oldpeak	0
	slope	0
	ca	0
	thal	0
	num	0
	dtype:	int64

Czy zbiór jest zbalansowany pod względem liczby próbek na klasy?

```
df['num'].value_counts().plot.bar()
```



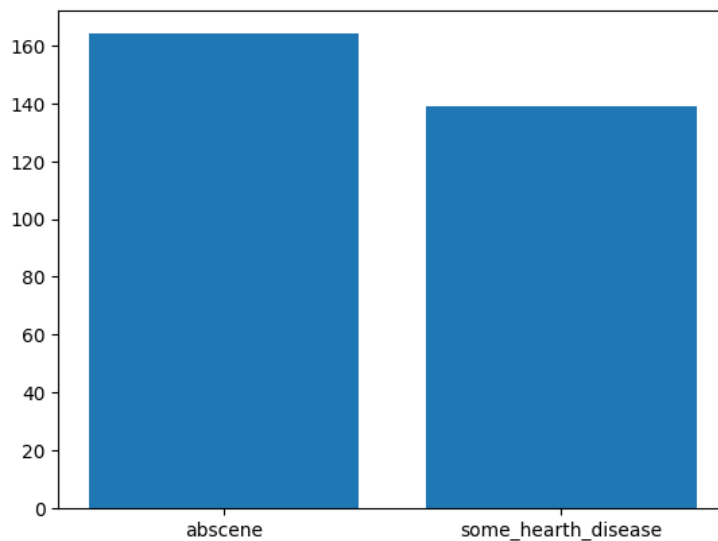
0 – brak choroby, [1,4] – choroba

Z informacji dotyczących zbioru wynika, że zbiór miał koncentrować się na wykrywaniu braku lub obecności jakiejś choroby. Dlatego też możemy wyświetlić wykres:

```
abscene = df['num'].value_counts()[0]
some_hearth_disease = df['num'].value_counts()[1:].sum()
print('abscene: ', abscene)
print('some_hearth_disease: ', some_hearth_disease)
plt.bar(['abscene', 'some_hearth_disease'], [abscene, some_hearth_disease])
```

abscene: 164

some_hearth_disease: 139



Dopiero teraz można zaakceptować zbalansowanie pod względem liczby próbek na klasy, aczkolwiek to czy będziemy rozróżniać kategorie choroby trzeba gruntownie ustalić.

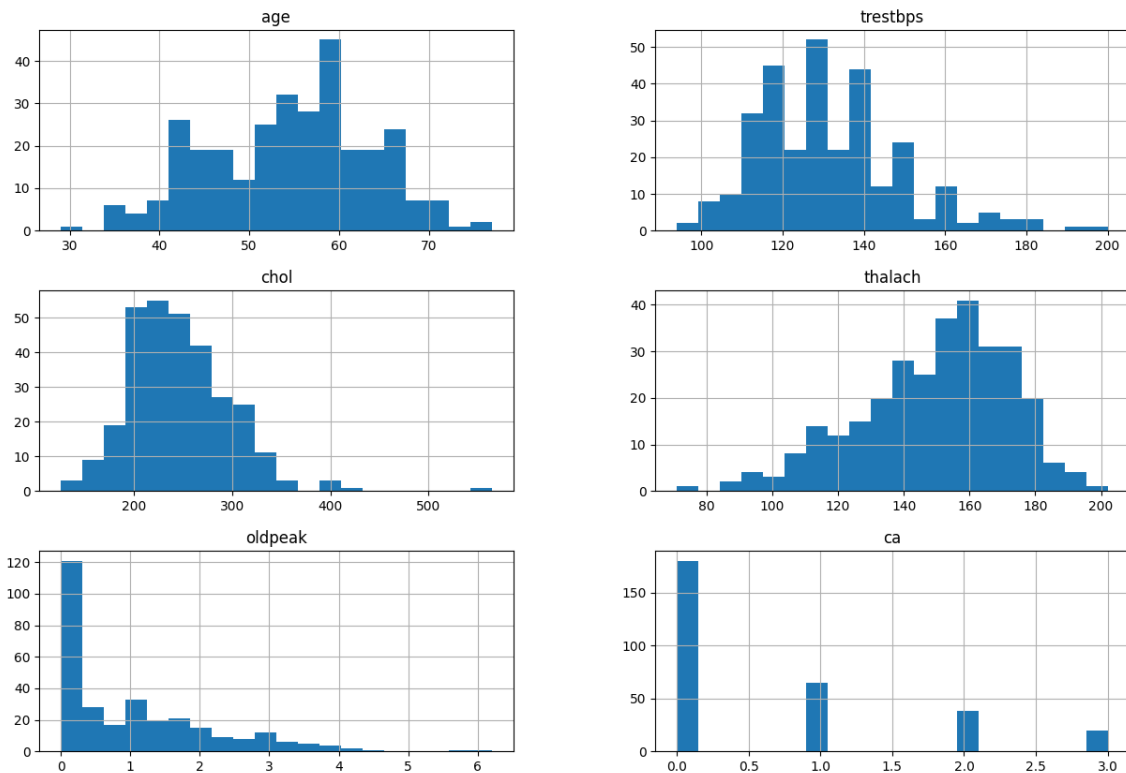
Średnie i odchylenia cech liczbowych.

```
numeric_features = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak', 'ca']
df[numeric_features].describe().loc[['mean', 'std']]
```

	age	trestbps	chol	thalach	oldpeak	ca
mean	54.438944	131.689769	246.693069	149.607261	1.039604	0.663366
std	9.038662	17.599748	51.776918	22.875003	1.161075	0.934375

Rozkłady cech liczbowych.

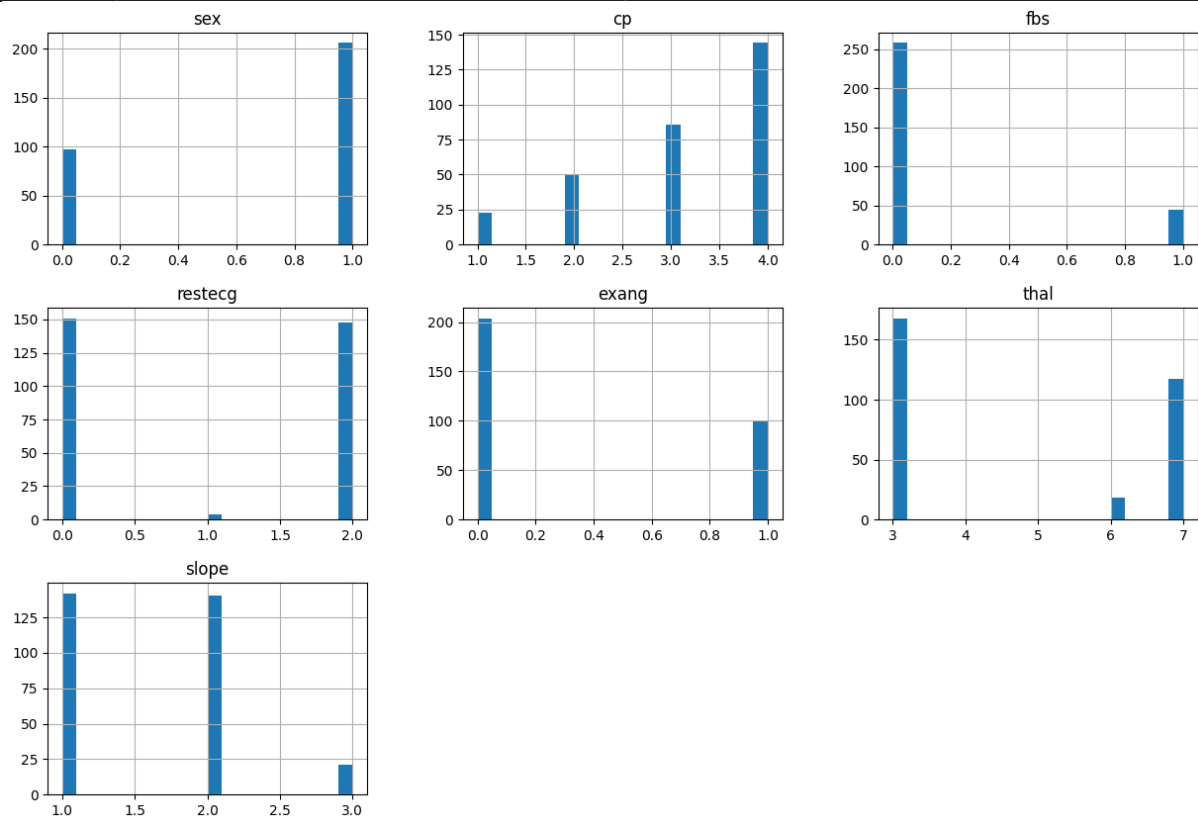
```
df[numeric_features].hist(bins=20, figsize=(15, 10))
```



Jedynymi cechami, które mogły być o rozkładzie normalnym to 'chol', 'thalach' i 'age' reszta nie przypomina rozkładów normalnych.

Rozkłady cech kategorycznych:

```
categories_features = ['sex', 'cp', 'fbs', 'restecg', 'exang', 'thal', 'slope']
df[categories_features].hist(bins=20, figsize=(15, 10))
```



Tutaj trzeba zauważyć, że dla każdej cechy istnieje jedna klasa, której licznosc jest bardzo mala, co sprawia, że rozkłady nie są idealnie równomierne.

Kod przekształcający dane do macierzy cech liczbowych (przykłady × cechy)

```
def one_hot_encode(df, column, column_names):
    dummies = pd.get_dummies(df[column], prefix=column)
    column_names = [column + '_' + str(name) for name in column_names]
    dummies.columns = column_names
    dummies = dummies.astype('int64')
    df = pd.concat([df, dummies], axis=1)
    df.drop(column, axis=1, inplace=True)
    return df

df = one_hot_encode(df, 'cp', ['typical_angina', 'atypical_angina', 'non-anginal_pain', 'asymptomatic'])
df = one_hot_encode(df, 'thal', ['normal', 'ST-T_wave_abnormality', 'left_ventricular_hypertrophy'])
df = one_hot_encode(df, 'slope', ['upsloping', 'flat', 'downsloping'])
df = one_hot_encode(df, 'restecg', ['normal', 'fixed_defect', 'reversable_defect'])
```



```
df.info()
```

[163] ✓ 0.0s

```
... <class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 303 entries, 0 to 302
```

```
Data columns (total 23 columns):
```

#	Column	Non-Null Count	Dtype
0	age	303 non-null	int64
1	sex	303 non-null	int64
2	trestbps	303 non-null	int64
3	chol	303 non-null	int64
4	fbs	303 non-null	int64
5	thalach	303 non-null	int64
6	exang	303 non-null	int64
7	oldpeak	303 non-null	float64
8	ca	303 non-null	float64
9	num	303 non-null	int64
10	cp_typical_angina	303 non-null	int64
11	cp_atypical_angina	303 non-null	int64
12	cp_non-anginal_pain	303 non-null	int64
13	cp_asymptomatic	303 non-null	int64
14	thal_normal	303 non-null	int64
15	thal_ST-T_wave_abnormality	303 non-null	int64
16	thal_left_ventricular_hypertrophy	303 non-null	int64
17	slope_upsloping	303 non-null	int64
18	slope_flat	303 non-null	int64
19	slope_downsloping	303 non-null	int64

```
...
```

```
21 restecg_fixed_defect 303 non-null int64
```

```
22 restecg_reversable_defect 303 non-null int64
```

```
dtypes: float64(2), int64(21)
```

```
memory usage: 54.6 KB
```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [set](#)

2. Uczenie maszynowe – regresja logistyczna

Cel ćwiczenia

W drugim ćwiczeniu zajmiemy się uczeniem maszynowym. Wykorzystamy regresję logistyczną – metodę statystyczną którą można uznać za formę najprostszej (jednowarstwowej) sieci neuronowej. Model będziemy wykorzystywać do klasyfikacji, czyli chcemy, aby aproksymował prawdopodobieństwo przynależności próbki opisanej wektorem x do właściwej klasy. Do tego celu możemy wykorzystać zbiór przykładów uczących w postaci par wejście-klasa.

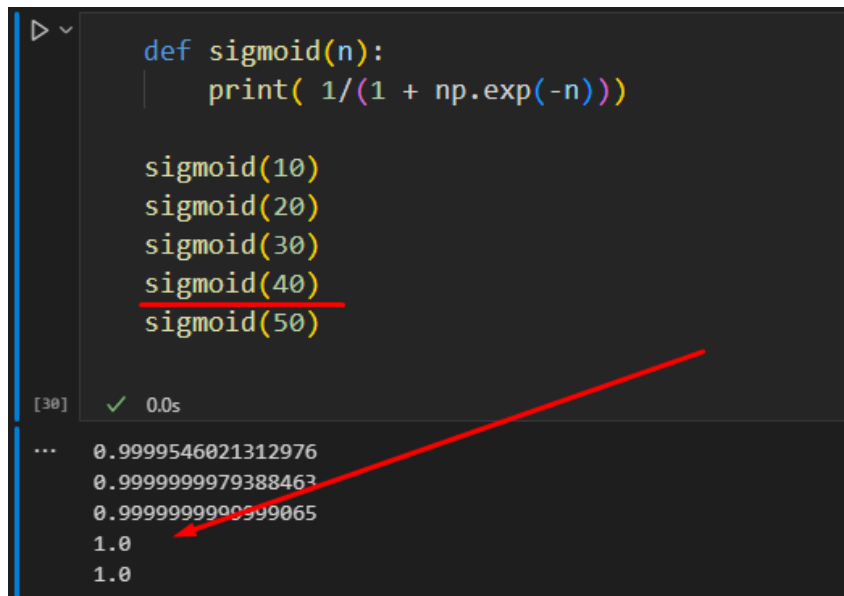
Przeprocesowanie danych

Kolumny kategoryczne zostały przeprocesowane na dodatkowe kolumny reprezentujące przynależność do danej kategorii w postaci binarnej przy pomocy metody „One-hot-encoding”, która została zrealizowana w poprzednim ćwiczeniu.

Przeprowadzimy standaryzację cech numerycznych:

```
# Standaryzacja cech numerycznych
from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
X[numeric_features] = scaler.fit_transform(X[numeric_features])
```

Później podczas używania funkcji `sigmoid()` okaże się, że dla dużych wartości n `sigmoid` szybko zbiega do wartości 1.0, co jest tylko możliwe z faktu, że `float64` nie wystarcza aby przedstawić liczbę tak bliską 1.0 a jeszcze nie równą 1.0.



```
def sigmoid(n):
    print( 1/(1 + np.exp(-n)))

sigmoid(10)
sigmoid(20)
sigmoid(30)
sigmoid(40)
sigmoid(50)
```

[30] ✓ 0.0s

```
... 0.9999546021312976
     0.999999979388463
     0.9999999999999065
     1.0
     1.0
```

Z założenia zbioru danych, wynikiem każdej próbki jest przynależność do klasy wystąpienia choroby serca. Sprawdzamy, czy jakkolwiek choroba występuje lub nie występuje. Kolejnym krokiem będzie przekształcenie różnych typów choroby serca na informację, że jakkolwiek choroba serca występuje.

```

y = y.replace([1, 2, 3, 4], 1)
y.value_counts()

[31] ✓ 0.0s

... num
0    164
1    139
Name: count, dtype: int64

```

Implementacja klasyfikatora regresji logistycznej po paczce przykładów w iteracji

$$\sigma(n) = \frac{1}{1 + e^{-n}}$$

```

def sigmoid(n):
    return 1 / (1 + np.exp(-n))

```

$$p(x) = \sigma(Wx + b)$$

```

# (n, 22) @ (22,) -> (n,)
predictions = sigmoid(X_batch @ self.weights + self.bias)

```

$$L = -y \ln p(x) - (1 - y) \ln(1 - p(x))$$

```

def loss(self, y, y_pred):
    return -(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))

```

$$\frac{\partial L}{\partial w_i} = -(y - p(x))x_i$$

```

# (n,22)T = (22,n) dot (n ,) -> (22,)
dw = (1/self.batch_size) * np.dot(X_batch.T, (predictions - y_batch))
db = (1/self.batch_size) * np.sum(predictions - y_batch)

```

$$w'_i = w_i - \alpha \frac{\partial L}{\partial w_i}$$

```

self.weights = self.weights - self.lr * dw
self.bias = self.bias - self.lr * db

```

Zbieżność modelu można zdefiniować przez wystarczająco małą zmianę funkcji kosztu w danej iteracji i pewną maksymalną liczbę iteracji

Cały kod przedstawia się następująco:

```

import numpy as np

```



```
def sigmoid(n):
    return 1 / (1 + np.exp(-n))

class MiniBatchLogisticRegression:

    def __init__(self, lr=0.001, epochs=1000, batch_size=5, seed=42, epsilon=0.0001):
        self.lr = lr # współczynnik uczenia
        self.epochs = epochs # liczba epok
        self.weights = None # wagi
        self.bias = None # bias
        self.batch_size = batch_size # rozmiar paczki
        self.seed = seed
        self.epsilon = epsilon
        self.cost_list = [] # lista kosztów
        self.mean_cost_list = [] # lista średnich kosztów
        self.epoch_list = [] # lista epok (opcjonalne)

    def loss(self, y, y_pred):
        return -(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.ones(n_features) # inicjacja wag
        self.bias = 0

        # Jeśli batch_size jest większy niż liczba próbek, to ustawiamy go na liczbę próbek ->
        # zwyczajnie uczymy na całym zbiorze GD
        if self.batch_size > n_samples:
            self.batch_size = n_samples

        #inicjacja list na koszt i epoki
        self.cost_list = []
        self.mean_cost_list = []
        self.epoch_list = []
        cost = 0
        #ustawiamy seed
        np.random.seed(self.seed)
        #dla każdej iteracji/epoki
        for i in range(self.epochs):
            #tasowanie indeksów ale deterministycznie bo ustawiliśmy seed
            random_order = np.random.permutation(n_samples)
            #tasujemy X i y
            X_shuffled = X[random_order]
            y_shuffled = y[random_order]
            # lista kosztów w paczkach dla każdej epoki
            cost_list_in_batch = []

            #dla każdej paczki
            for j in range(0, n_samples, self.batch_size):

                X_batch = X_shuffled[j:j + self.batch_size]
                y_batch = y_shuffled[j:j + self.batch_size]

                # (n, 22) @ (22,) -> (n,)
                predictions = sigmoid(X_batch @ self.weights + self.bias)
                # (n,22)T = (22,n) dot (n ,) -> (22,)
                dw = (1/self.batch_size) * np.dot(X_batch.T, (predictions - y_batch))
                db = (1/self.batch_size) * np.sum(predictions - y_batch)

                self.weights = self.weights - self.lr * dw
                self.bias = self.bias - self.lr * db

                cost = np.mean(self.loss(y_batch, predictions))
                cost_list_in_batch.append(cost)

            #wystarczająco mała zmiana funkcji kosztu w iteracji
            if(i > 0 and abs(cost - self.cost_list[-1]) < self.epsilon):
                break
            self.cost_list.append(cost)
            iteration_cost = np.mean(cost_list_in_batch)
            self.mean_cost_list.append(iteration_cost)
            self.epoch_list.append(i)
```

```
print('cost', cost, 'cost_list[-1]', self.cost_list[-1], 'i', i, 'abs(cost - cost_list[-1])',  
abs(cost - self.cost_list[-1]))  
  
def predict(self, X):  
    linear_pred = np.dot(X, self.weights) + self.bias  
    y_pred = sigmoid(linear_pred)  
    class_pred = [0 if y <= 0.5 else 1 for y in y_pred]  
    return class_pred
```

Weryfikacja powinna uwzględnić podział na dane uczące i testowe

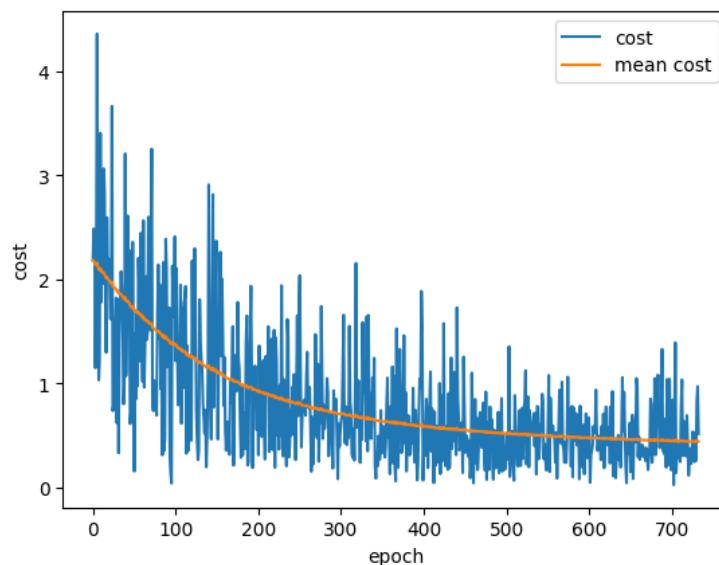
```
from sklearn.model_selection import train_test_split  
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

Uczenie się modelu powinno być weryfikowalne metryką (np. accuracy, fscore, precision – można korzystać z bibliotek)

```
clf = MiniBatchLogisticRegression(lr=0.0005, epochs=3000, batch_size=10, seed=42,  
epsilon=0.001)
```

```
clf.fit(X_train, y_train)  
y_pred = clf.predict(X_test)
```

cost 0.5155389870377539 cost_list[-1] 0.5148583909592925 i 733 abs(cost - cost_list[-1]) 0.0006805960784613818



Wykres funkcji kosztu w każdej iteracji, gdzie mean cost, to uśredniony koszt z wszystkich paczek w iteracji, a cost to koszt z ostatniej paczki w iteracji.

```
from sklearn.metrics import accuracy_score  
from sklearn.metrics import precision_score  
from sklearn.metrics import recall_score  
from sklearn.metrics import f1_score  
  
print("Accuracy: ", accuracy_score(y_test, y_pred))  
print("Precision: ", precision_score(y_test, y_pred))  
print("Recall: ", recall_score(y_test, y_pred))  
print("F1: ", f1_score(y_test, y_pred))
```

Accuracy: 0.75

Precision: 0.725

Recall: 0.7837837837837838

F1: 0.7532467532467533

Po lepszym dopasowaniu hiperparametrów możemy uzyskać następujące wyniki metryk:

```
clf = MiniBatchLogisticRegression(lr=0.001, epochs=1000, batch_size=7,  
seed=27, epsilon=0.0001)
```

Accuracy: 0.8688524590163934

Precision: 0.8529411764705882

Recall: 0.90625

F1: 0.8787878787878787

3. Sieć neuronowa

Cel ćwiczenia

W trzecim ćwiczeniu zajmiemy się zbudowaniem modelu wielowarstwowego sieci neuronowej z dowolną funkcją aktywacji i funkcją kroszu taką, jak dla regresji logistycznej. Ćwiczenie obejmuje algorytm propagacji wstecznej.

Przeprocesowanych danych

```
heart_disease = fetch_ucirepo(id=45)  
  
X = heart_disease.data.features  
Y = heart_disease.data.targets  
  
Y = Y['num'].replace([1, 2, 3, 4], 1)  
  
X['num'] = Y  
  
median = X['ca'].median()  
X['ca'].fillna(median, inplace=True)  
mode_category = X['thal'].mode()[0]  
X['thal'].fillna(mode_category, inplace=True)
```

Podobnie jak w ćwiczeniu drugim po wgraniu zbioru danych, przetwarzamy je. Zamieniamy wieloklasowość na postać binarną (choroba/brak). Brakujące dane wypełniamy modą i medianą.

```
def one_hot_encode(df, column, column_names):  
    dummies = pd.get_dummies(df[column], prefix=column)  
    column_names = [column + '_' + str(name) for name in column_names]  
    dummies.columns = column_names  
    dummies = dummies.astype('int64')  
    df = pd.concat([df, dummies], axis=1)  
    df.drop(column, axis=1, inplace=True)  
    return df  
  
X = one_hot_encode(X, 'cp', ['typical_angina', 'atypical_angina', 'non-anginal_pain',  
    'asymptomatic'])  
X = one_hot_encode(X, 'thal', ['normal', 'ST-T_wave_abnormality', 'left_ventricular_hypertrophy'])  
X = one_hot_encode(X, 'slope', ['upsloping', 'flat', 'downsloping'])  
X = one_hot_encode(X, 'restecg', ['normal', 'fixed_defect', 'reversible_defect'])
```

Wykonujemy „one hot encoding” dla cech kategorycznych.

```
X_norm = (X - X.min()) / (X.max() - X.min())  
X_norm.describe()
```

	age	sex	trestbps	chol	fb	thalach	exang	oldpeak	ca	cp_typical_angina	...
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	...
mean	0.529978	0.679868	0.355564	0.275555	0.148515	0.600055	0.326733	0.167678	0.221122	0.075908	...
std	0.188305	0.467299	0.166035	0.118212	0.356198	0.174618	0.469794	0.187270	0.311458	0.265288	...
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...
25%	0.395833	0.000000	0.245283	0.194064	0.000000	0.477099	0.000000	0.000000	0.000000	0.000000	...
50%	0.562500	1.000000	0.339623	0.262557	0.000000	0.625954	0.000000	0.129032	0.000000	0.000000	...
75%	0.666667	1.000000	0.433962	0.340183	0.000000	0.725191	1.000000	0.258065	0.333333	0.000000	...
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	...

Normalizujemy dane od zakresu [0,1] ponieważ w modelu użyjemy regresji logistycznej, która dla danych nieprzeskalowanych może szybko osiągać wartości 0 i 1, co może powodować problemy zbieżności.

Implementacja sieci neuronowej

```
def sigmoid(X):
    return np.exp(-np.logaddexp(0, -X))

def sigmoid_derivative(X):
    return sigmoid(X) * (1 - sigmoid(X))

def cross_entropy(y, y_pred):
    return -np.sum(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred), axis=1)

def cross_entropy_derivative(y, y_pred):
    return -np.sum(y / y_pred - (1 - y) / (1 - y_pred), axis=1)

class Neuron():
    def __init__(self, num_of_weights, activation_function, activation_derivative, stand_dev=1):
        self.weights = np.random.normal(scale=stand_dev, size=num_of_weights)
        self.bias = np.random.normal(scale=stand_dev)
        self.X = None
        self.derivative = None
        self.activation_function = activation_function
        self.activation_derivative = activation_derivative

    def forward(self, inputs):
        self.X = inputs
        return self.activation_function(np.dot(inputs, self.weights) + self.bias)

    def backward(self, error, weights_next_layer=None):
        if weights_next_layer is not None:
            error = error.T @ weights_next_layer
        self.derivative = error * self.activation_derivative(np.dot(self.X, self.weights) + self.bias)
        return self.derivative

    def update(self, learning_rate):
        self.weights -= learning_rate * np.dot(self.X.T, self.derivative)
        self.bias -= learning_rate * np.sum(self.derivative)
        self.X = None
        self.derivative = None

class NeuralNetwork():
    def __init__(self, num_of_inputs, hidden_layers, num_of_outputs, activation_function = sigmoid,
activation_derivative = sigmoid_derivative, loss_function = cross_entropy, loss_derivative =
cross_entropy_derivative, stand_dev=1.0):
        self.layers = []
        self.num_of_inputs = num_of_inputs
        self.hidden_layers = hidden_layers
        self.num_of_outputs = num_of_outputs
        self.num_of_hidden_layers = len(hidden_layers)
        self.activation_function = activation_function
        self.activation_derivative = activation_derivative
        self.loss_function = loss_function
        self.loss_derivative = loss_derivative
```



```
self.stand_dev = stand_dev

self.layers.append([Neuron(num_of_inputs, activation_function, activation_derivative,
self.stand_dev) for _ in range(hidden_layers[0])])
for i in range(1, self.num_of_hidden_layers):
    self.layers.append([Neuron(hidden_layers[i-1], activation_function, activation_derivative,
self.stand_dev) for _ in range(hidden_layers[i])])
if self.num_of_outputs == 1:
    self.layers.append([Neuron(hidden_layers[-1], activation_function, activation_derivative,
self.stand_dev)])

def forward(self, X):
    for layer in self.layers:
        X = np.array([neuron.forward(X) for neuron in layer]).T
    return X

def backward(self, output_error):
    output_layer = self.layers[-1]
    output_error = np.array([neuron.backward(output_error) for neuron in output_layer])
    weights_next_layer = np.array([neuron.weights for neuron in output_layer]).T

    for layer in reversed(self.layers[:-1]):
        output_error = np.array([neuron.backward(output_error, weights_next_layer[index]) for
index, neuron in enumerate(layer)])
        weights_next_layer = np.array([neuron.weights for neuron in layer]).T

    return output_error

def update(self, learning_rate):
    for layer in self.layers:
        for neuron in layer:
            neuron.update(learning_rate)

def fit(self, X, y, X_test, Y_test, epochs=1000, batch_size=30, learning_rate=0.001):
    n_samples = X.shape[0]
    if batch_size > n_samples:
        batch_size = n_samples
    train_cost_list = []

    test_cost_list = []

    for epoch in range(epochs):
        random_order = np.random.permutation(n_samples)
        X_shuffled = X.values[random_order]
        y_shuffled = y.values[random_order]

        for batch_index in range(0, n_samples, batch_size):
            X_batch = X_shuffled[batch_index:batch_index + batch_size]
            y_batch = y_shuffled[batch_index:batch_index + batch_size].reshape(-1, 1)
            predictions = self.forward(X_batch)
            output_error = self.loss_derivative(y_batch, predictions)
            self.backward(output_error)
            self.update(learning_rate)

        train_cost = np.mean(self.loss_function(y_batch, predictions))

        test_predictions = self.predict(X_test)
        test_cost = np.mean(self.loss_function(Y_test.values.reshape(-1, 1), test_predictions))

        train_cost_list.append(train_cost)

        test_cost_list.append(test_cost)

        if epoch % (epochs // 8) == 0:
            print(f'Epoch: {epoch}, Train Loss: {train_cost}, Test Loss: {test_cost}')

    print(f'Epoch: {epoch}, Train Loss: {train_cost}, Test Loss: {test_cost}')
    return self, train_cost_list, test_cost_list

def predict(self, X):
    return self.forward(X)
```

Na samym początku zdefiniowałem potrzebną funkcję aktywacji oraz jej pochodną.

- **sigmoid(x)**: stabilnie numeryczna funkcja sigmoidalna
- **sigmoid_derivative(x)**: pochodna funkcji sigmoidalnej

Oraz funkcję liczenia kosztu:

- **cross_entropy(y, y_pred)**: funkcja kosztu
- **cross_entropy_derivative(y, y_pred)**: pochodna funkcji kosztu

Klasa **Neuron** reprezentuje pojedynczy neuron w sieci neuronowej. Przyjmuje następujące parametry podczas inicjalizacji:

- **num_of_weights**: Liczba wag (parametrów) neuronu.
- **activation_function**: Funkcja aktywacji neuronu. Przekazana przez parametr. Domyślnie funkcja sigmoidalna.
- **activation_derivative**: Pochodna funkcji aktywacji neuronu. Przekazana przez parametr. Domyślnie pochodna funkcji sigmoidalnej.
- **stand_dev**: Odchylenie standardowe używane do inicjalizacji wag neuronu (domyślnie 1).

Metody klasy **Neuron** obejmują:

- **forward(self, inputs)**: wykonuje krok przód, zapisuje wejście (macierz) do propagacji wstecznej, oblicza wyjście neuronu na podstawie wejścia i funkcji aktywacji.
- **backward(self, error, weights_next_layer=None)**: wykonuje krok wstecz, oblicza pochodną błędu względem wejścia neuronu. Przyjmuje macierz gradientów z warstwy następnej. Dla warstwy wyjściowej przyjmuje błąd wyjściowy. Zwraca macierz gradientów dla warstwy poprzedniej. Jeżeli przyjmuje wagi warstwy następnej to obliczany jest sumę iloczynu wag i gradientów z warstwy następnej, potem zwracamy macierz pochodnych funkcji kosztu.
- **update(self, learning_rate)**: aktualizuje wagi oraz bias neuronu na podstawie gradientów funkcji kosztu dla każdej z wag i współczynnika uczenia. Przechodzimy po wszystkich wagach i aktualizujemy je.

Klasa **NeuralNetwork** reprezentuje sieć neuronową. Przyjmuje następujące parametry podczas inicjalizacji:

- **num_of_inputs**: Liczba wejść sieci.
- **hidden_layers**: Lista zawierająca liczbę neuronów w każdej z warstw ukrytych.
- **num_of_outputs**: Liczba wyjść sieci.
- **activation_function**: Funkcja aktywacji używana w neuronach sieci.
- **activation_derivative**: Pochodna funkcji aktywacji.
- **loss_function**: Funkcja kosztu sieci.
- **loss_derivative**: Pochodna funkcji kosztu sieci.
- **stand_dev**: Odchylenie standardowe przy inicjacji wag neuronów w warstwach sieci.

Metody klasy **NeuralNetwork** obejmują:

- **forward(self, inputs)**: Wykonuje krok przód, obliczając wyjście neuronu na podstawie wejścia i funkcji aktywacji. Przyjmuje na wejściu macierz wejść sieci czyli danych, wyniki

przekazuje jako macierz wyjść, które są wejściem dla kolejnej warstwy, tak do momentu aż dojdziemy do ostatniej warstwy, która zwraca wynik.

- **backward(self, error, weights_next_layer=None):** Wykonuje krok wstecz, obliczając pochodną błędu względem wejścia neuronu. Jeśli - dostępne są wagi warstwy następnej, można je przekazać, aby obliczyć błąd wsteczny w oparciu o te wagi. W przypadku ostatniej warstwy wyjściowej nie mamy wag następnej warstwy, więc przekazujemy None. Wtedy obliczamy błąd wsteczny na podstawie błędu wyjściowego, który jest przekazany jako parametr.
- **update(self, learning_rate):** Aktualizuje wagi oraz bias neuronu na podstawie pochodnej błędu i współczynnika uczenia. Czynność wykonuje się przez wszystkie neurony w warstwie. Przechodzimy po wszystkich warstwach. Przyjmuje jako parametr współczynnik uczenia.
- **fit(self, X, y, epochs=1000, batch_size=30, learning_rate=0.001):** funkcja przeprowadzająca procedure nauki modelu. Przyjmuje:
 - **X** : dane wejściowe
 - **Y** : dane wyjściowe
 - **epochs** : liczba epok, domyślnie 1000
 - **batch_size** : rozmiar batcha, domyślnie 20
 - **learning_rate** : współczynnik uczenia, domyślnie 0.001
- Metoda w ramach każdej epoki oraz batcha wykonuje **forward**, obliczenie kosztu, **backward**, oraz **update**.
- **predict(self, X):** na wejściu przyjmuj macierz wejść do sieci, zwraca wynik wywnioskowany przez sieć

Badanie zachowania modelu od jego ustawień

Zaaranżowanie ustawień:

```
seed = 42
np.random.seed(seed)
X_train, X_test, Y_train, Y_test = train_test_split(X_norm, Y, test_size=0.2, random_state=42)
X_train_n, X_test_n, Y_train_n, Y_test_n = train_test_split(X, Y, test_size=0.2, random_state=42)

# domyślna sieć
default_hidden_layers = [8, 4]

#1. Różnej wymiarowości warstwy ukrytej
dim_hidd_1 = [4, 8]
dim_hidd_2 = [20, 10]
dim_hidd_3 = [50, 25]
#2. Różnej wartości współczynnika uczenia
learning_rate_1 = 0.0001
learning_rate_2 = 0.001
#3. Różnej wartości parametru standaryzacji
stand_dev_1 = 0.5
stand_dev_2 = 2
#4. danych znormalizowanych i nieznormalizowanych
unnorm = X_train_n, Y_train_n
#5. Różnej liczby warstw ukrytych
hidden_layers_size_1 = [8]
hidden_layers_size_2 = [8,8,8]
hidden_layers_size_3 = [8,8,8,8]
```



Wyniki modeli:

Default

Accuracy: 0.8524590163934426
Precision: 0.8484848484848485
Recall: 0.875
F1: 0.8615384615384615

hidden_1 [4, 8]

Accuracy: 0.8688524590163934
Precision: 0.8529411764705882
Recall: 0.90625
F1: 0.8787878787878787

hidden_2 [20, 10]

Accuracy: 0.8688524590163934
Precision: 0.9
Recall: 0.84375
F1: 0.870967741935484

hidden_3 [50, 25]

Accuracy: 0.8688524590163934
Precision: 0.8529411764705882
Recall: 0.90625
F1: 0.8787878787878787

learning_rate_1 0.0001

Accuracy: 0.7377049180327869
Precision: 0.8636363636363636
Recall: 0.59375
F1: 0.7037037037037037

learning_rate_2 0.01

Accuracy: 0.7704918032786885
Precision: 0.7647058823529411
Recall: 0.8125
F1: 0.7878787878787878

stand_dev_1 = 0.5

Accuracy: 0.8852459016393442
Precision: 0.8787878787878788
Recall: 0.90625
F1: 0.8923076923076922

stand_dev_2 = 0.1

Accuracy: 0.47540983606557374
Precision: 0.0
Recall: 0.0
F1: 0.0

unnorm

Accuracy: 0.6885245901639344
Precision: 0.696969696969697
Recall: 0.71875
F1: 0.7076923076923077

hidden_layers_size_1 [8]

Accuracy: 0.8688524590163934
Precision: 0.9
Recall: 0.84375
F1: 0.870967741935484

hidden_layers_size_2 [8, 8, 8]

Accuracy: 0.9508196721311475
Precision: 0.9393939393939394
Recall: 0.96875
F1: 0.9538461538461539

hidden_layers_size_3 [8, 8, 8, 8]

Accuracy: 0.8852459016393442
Precision: 0.8787878787878788
Recall: 0.90625
F1: 0.8923076923076922

Uczenie modeli:



Wnioski

Pierwszym wnioskiem, który bardzo rzuca się w oczy jest niezdolność modelu do nauki w przypadku danych nieznormalizowanych. Pobudzenie neuronów było zbyt duże, co jest spowodowane problemem zbieżności.

Kolejnym może być, że w przypadku inicjalizacji wag, mniejsze odchylenie standardowe powoduje mniejsze zmiany wyników, co prowadzi do wydłużenia procesu uczenia, a gdy odchylenie

standardowe jest bardzo małe, skutkuje to brakiem możliwości dalszej nauki, przez mnożenie przez liczby bliskie zeru.

Wysoki współczynnik uczenia sprawił, że model szybko dopasował się do danych treningowych, ale za słabo „uogólnia” predykcję. Zbyt niski, wydłuża proces uczenia się, zatem w 1000 epokach nie osiągnął dobrych wyników, oraz nie widać trendu uczenia się.

Zmiana wymiarowości dwóch warstw ukrytych poprzez odwrócenie kolejności warstw na [4,8] sprawiło, wydłużenie procesu uczenia się, błąd uczenia się nie spada tak szybko, oraz końcowo uzyskano lepszy wynik od domyślnych warstw ukrytych. Zwiększenie liczby neuronów w warstwach spowodowało coraz szybszy spadek kosztu modelu na danych co jest widoczne w początkowych etapach uczenia się.

Natomiast zwiększanie liczby warstw bardziej stabilizuje wartości funkcji kosztu, dodatkowo wydłuża proces uczenia się modelu.

Na 1000 epok najlepiej wyuczony został model **Hidden layers size 2 [8, 8, 8]** gdzie reszta parametrów była domyślna: `wsp. Uczenia = 0.001`, `stand_dev = 1.0`

```
hidden_layers_size_2 [8, 8, 8]
Accuracy: 0.9508196721311475
Precision: 0.9393939393939394
Recall: 0.96875
F1: 0.9538461538461539
-----
```

4. Otworzenie architektury sieci w PyTorch oraz porównanie optimzerów

Cel ćwiczenia

W ćwiczeniu 4 odtworzymy zaimplementowaną już architekturę sieci w pełni połączonej korzystając z gotowego rozwiązania do budowania sieci neuronowych. Przeprowadzimy badanie jak zachowuje się model w zależności od użytych optymalizatorów, rozmiaru paczek oraz współczynnika uczenia się modelu.

Rekreacja modelu z wykorzystaniem pytorch

Architektura sieci:

```
class FullyConnectedNetwork(nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size):
        torch.manual_seed(seed)
        super(FullyConnectedNetwork, self).__init__()
        self.hidden_layers = nn.ModuleList()
        prev_size = input_size
        for hidden_size in hidden_sizes:
            layer = nn.Linear(prev_size, hidden_size)
            torch.nn.init.normal_(layer.weight)
            self.hidden_layers.append(layer)
            prev_size = hidden_size
        self.output = nn.Linear(prev_size, output_size)
        torch.nn.init.normal_(self.output.weight)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        for hidden_layer in self.hidden_layers:
            x = hidden_layer(x)
            x = self.sigmoid(x)
        x = self.output(x)
        return x
```

Klasa **FullyConnectedNetwork** opowiada za zdefiniowanie własnej architektury sieci neuronowej dziedzicząc po abstrakcyjnej klasie **nn.Module**.

Konstruktor **__init__** przyjmuje następujące argumenty:

- **input_size** (rozmiar wejścia),
- **hidden_sizes** (lista rozmiarów ukrytych warstw)
- **output_size** (rozmiar wyjścia).

Wewnątrz tej metody, tworzymy listę **self.hidden_layers**, która zawiera wszystkie ukryte warstwy sieci. Każda warstwa jest inicjalizowana za pomocą **nn.Linear**, co oznacza, że jest to warstwa liniowa. Wagi dla każdej warstwy są inicjalizowane za pomocą rozkładu normalnego. Domyślnymi wartościami tensora z rozkładem normalnym jest średnia o wartości 0.0 i odchylenie standardowe równe 1.0.

forward: Ta metoda definiuje, jak sieć przetwarza dane wejściowe. Dla każdej ukrytej warstwy, dane wejściowe są przekształcane za pomocą tej warstwy, a następnie przekształcane za pomocą funkcji aktywacji sigmoidalnej. Na końcu, dane są przekształcane za pomocą warstwy wyjściowej, co daje końcowe wyjście sieci.

Pętla uczenia:

```
def train_model(model, criterion, optimizer, X_train, Y_train, X_test, Y_test, epochs=1000,
batch_size=30):
    torch.manual_seed(seed)
    np.random.seed(seed)
    losses = []
    test_cost_list = []

    X_train_tensor = torch.from_numpy(X_train.values).float()
    Y_train_tensor = torch.from_numpy(Y_train.values).float()

    n_samples = X_train_tensor.shape[0]
    if batch_size > n_samples:
        batch_size = n_samples

    for epoch in range(epochs):
        random_order = torch.randperm(n_samples)
        X_shuffled = X_train_tensor[random_order]
        y_shuffled = Y_train_tensor[random_order]

        for batch_index in range(0, n_samples, batch_size):
            X_batch = X_shuffled[batch_index:batch_index + batch_size]
            y_batch = y_shuffled[batch_index:batch_index + batch_size].reshape(-1, 1)

            optimizer.zero_grad()
            y_pred = model.forward(X_batch)

            loss = criterion(y_pred, y_batch)
            loss.backward()
            optimizer.step()

        losses.append(loss.item())
        with torch.no_grad():
            test_outputs = model(X_test)
            test_loss = criterion(test_outputs.squeeze(), Y_test)
            test_cost_list.append(test_loss.item())

        if (epoch + 1) % 100 == 0:
            print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}')
    return losses, test_cost_list
```

Tutaj warto zadbać o deterministyczne pseudolosowanie sekwencji paczek w epokach pomiędzy modelami oraz transformację danych na tensory. Zdecydowano się na użycie pętli, która była użyta w poprzednich ćwiczeniach zamiast korzystania z elementów biblioteki torch jak RandomSampler,

czy DataLoader, aby pseudolosowa kolejność próbek była identyczna co w autorskiej implementacji sieci z ćwiczenia trzeciego.

Porównanie optymalizatorów

SGD (Stochastic Gradient Descent): SGD jest jednym z najprostszych, ale skutecznych algorytmów optymalizacji używanych w uczeniu maszynowym. SGD jest wybierany ze względu na jego prostotę i szybkość, szczególnie dla problemów z dużą ilością danych. Jednak może mieć problemy z osiągnięciem optymalnego rozwiązania w niektórych przypadkach, na przykład, gdy powierzchnia błędu jest bardzo nierówna.

Adam (Adaptive Moment Estimation): Adam jest bardziej zaawansowanym algorytmem optymalizacji, który adaptuje wielkość kroków dla każdego z parametrów. To oznacza, że dla różnych parametrów może używać różnych szybkości uczenia się. Adam jest często wybierany, ponieważ zazwyczaj działa dobrze na praktycznie wszystkich rodzajach problemów.

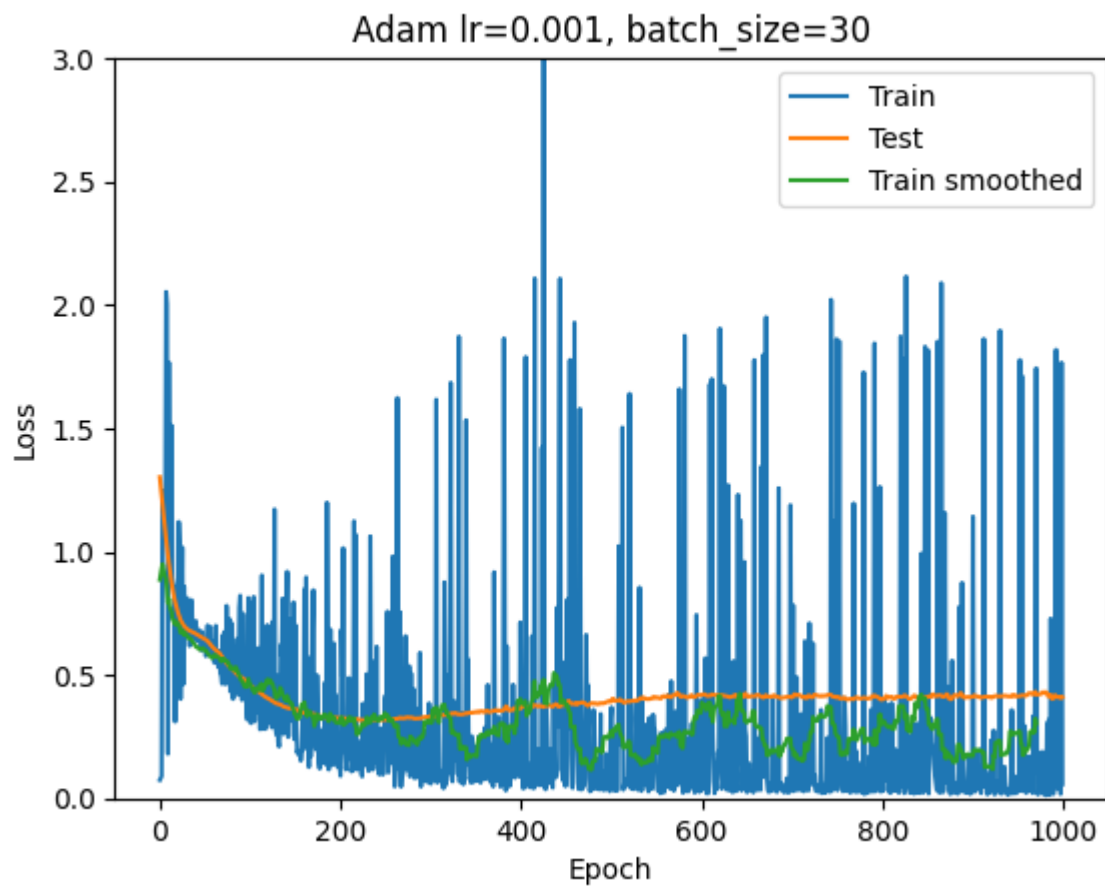
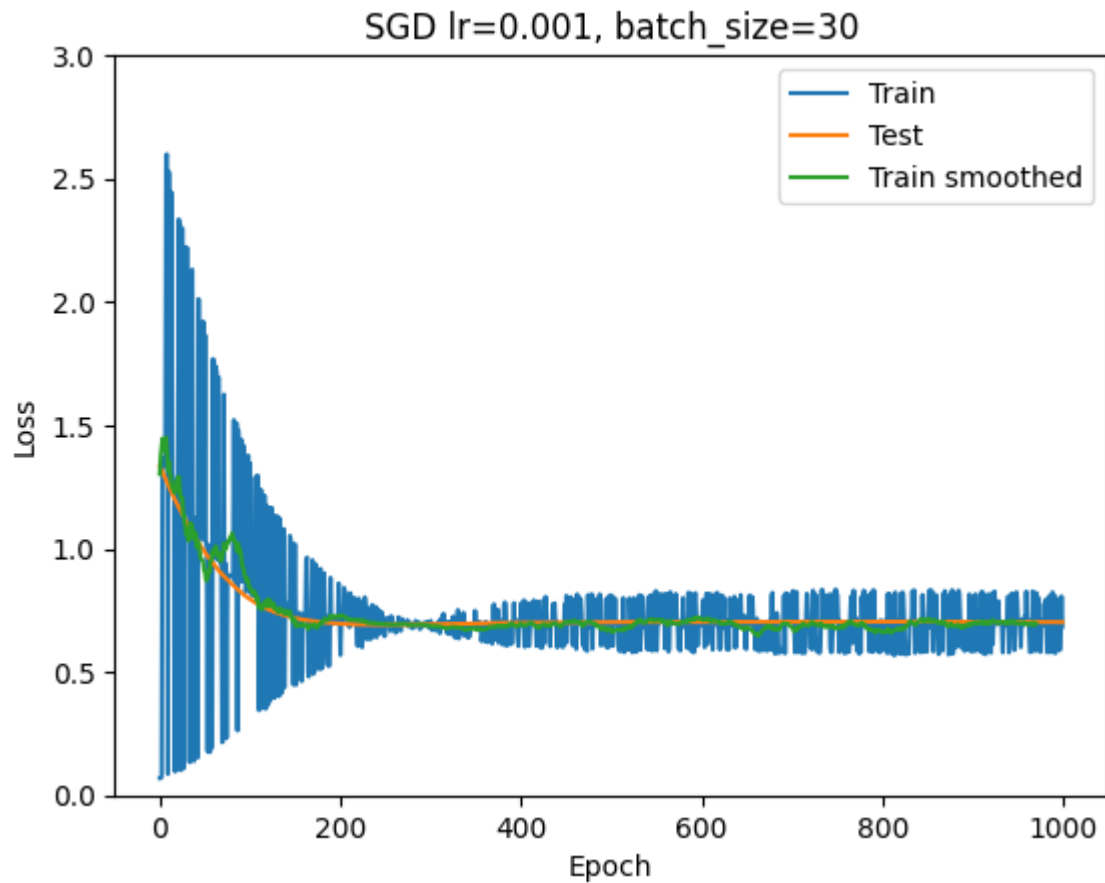
Adagrad (Adaptive Gradient Algorithm): Adagrad jest innym algorytmem optymalizacji, który adaptuje szybkość uczenia się dla każdego z parametrów, co jest szczególnie przydatne dla problemów z rzadkimi danymi. Adagrad jest wybierany, gdy chcemy różne szybkości uczenia się dla różnych cech. Jednak Adagrad ma tendencję do zbyt szybkiego zmniejszania szybkości uczenia się, co może prowadzić do zbyt wczesnego zatrzymania procesu optymalizacji.

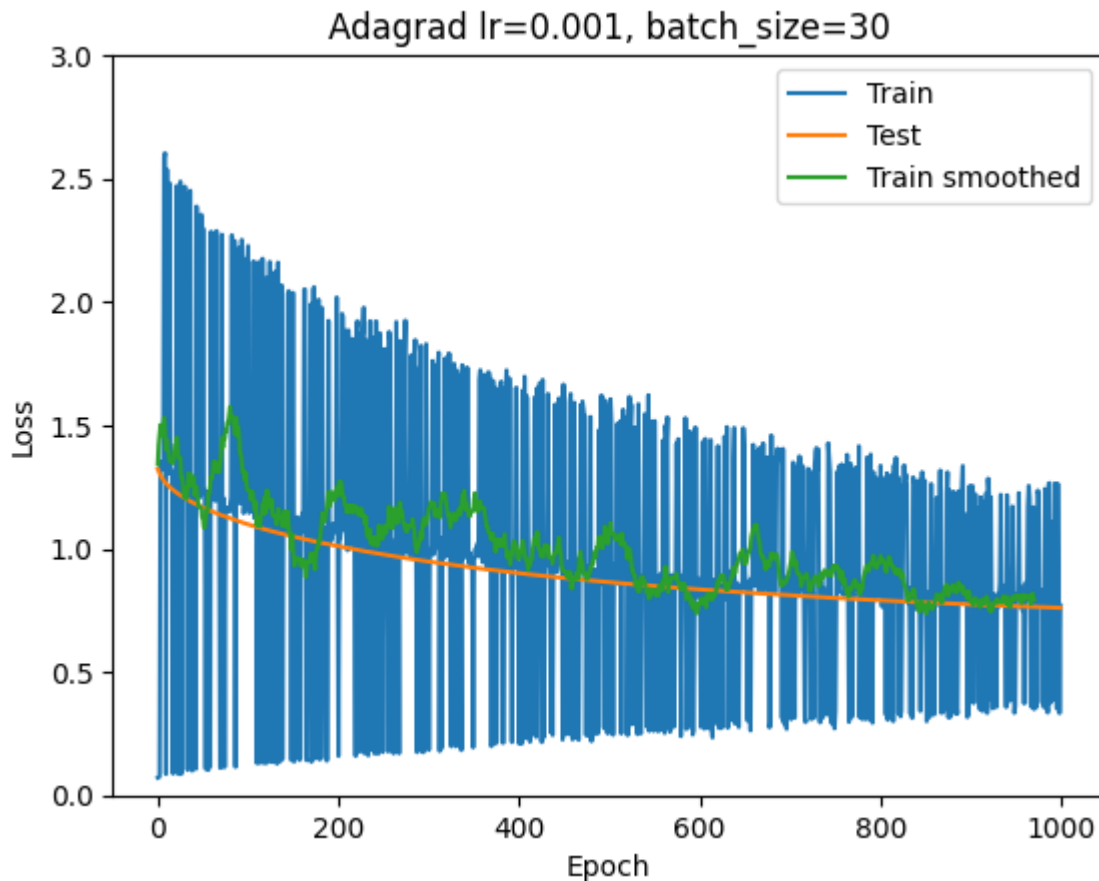
```
print('SGD')
model_sgd = FullyConnectedNetwork(input_size=22, hidden_sizes=hidden_layers, output_size=1)
sgd = torch.optim.SGD(model_sgd.parameters(), lr=lr)
lossesSGD = train_model(model_sgd, criterion, sgd, X_train, Y_train, X_test, Y_test, epochs)
plot_loss(lossesSGD)
evaluate_model(model_sgd, X_test, Y_test)
print('-----')

print('Adam')
model_adam = FullyConnectedNetwork(input_size=22, hidden_sizes=hidden_layers, output_size=1)
adam = torch.optim.Adam(model_adam.parameters(), lr=lr)
lossesAdam = train_model(model_adam, criterion, adam, X_train, Y_train, X_test, Y_test, epochs)
plot_loss(lossesAdam)
evaluate_model(model_adam, X_test, Y_test)
print('-----')

print('Adagrad')
model_adagrad = FullyConnectedNetwork(input_size=22, hidden_sizes=hidden_layers, output_size=1)
adagrad = torch.optim.Adagrad(model_adagrad.parameters(), lr=lr)
lossesSGD2 = train_model(model_adagrad, criterion, adagrad, X_train, Y_train, X_test, Y_test, epochs)
plot_loss(lossesSGD2)
evaluate_model(model_adagrad, X_test, Y_test)
print('-----')
```

SGD	Adam	Adagrad
Epoch 100/1000, Loss: 0.8277	Epoch 100/1000, Loss: 0.3436	Epoch 100/1000, Loss: 1.1607
Epoch 200/1000, Loss: 0.7124	Epoch 200/1000, Loss: 0.1392	Epoch 200/1000, Loss: 1.0817
Epoch 300/1000, Loss: 0.7000	Epoch 300/1000, Loss: 0.3364	Epoch 300/1000, Loss: 0.9723
Epoch 400/1000, Loss: 0.6907	Epoch 400/1000, Loss: 0.7133	Epoch 400/1000, Loss: 0.9150
Epoch 500/1000, Loss: 0.6975	Epoch 500/1000, Loss: 0.3227	Epoch 500/1000, Loss: 0.8617
Epoch 600/1000, Loss: 0.6909	Epoch 600/1000, Loss: 0.0324	Epoch 600/1000, Loss: 0.8773
Epoch 700/1000, Loss: 0.7083	Epoch 700/1000, Loss: 0.1500	Epoch 700/1000, Loss: 0.8447
Epoch 800/1000, Loss: 0.6966	Epoch 800/1000, Loss: 0.0341	Epoch 800/1000, Loss: 0.8505
Epoch 900/1000, Loss: 0.5915	Epoch 900/1000, Loss: 0.0568	Epoch 900/1000, Loss: 1.2554
Epoch 1000/1000, Loss: 0.6835	Epoch 1000/1000, Loss: 0.0567	Epoch 1000/1000, Loss: 0.7706
Accuracy: 0.4754	Accuracy: 0.8361	Accuracy: 0.5246
F1 Score: 0.0000	F1 Score: 0.8485	F1 Score: 0.6882
Precision: 0.0000	Precision: 0.8235	Precision: 0.5246
Recall: 0.0000	Recall: 0.8750	Recall: 1.0000





Wnioski:

SGD: Ten optymalizator osiągnął najniższą dokładność (0.4754) i F1 Score (0.0000). To sugeruje, że SGD miał trudności z optymalizacją modelu dla tego zestawu danych. Może to wynikać z faktu, że SGD nie dostosowuje szybkości uczenia się dla różnych parametrów.

Adam: Ten optymalizator osiągnął najwyższą dokładność (0.8361) i F1 Score (0.8485). To sugeruje, że Adam był w stanie skutecznie optymalizować model dla tego zestawu danych. Adam jest znany z tego, że zazwyczaj działa dobrze na praktycznie wszystkich rodzajach problemów.

Adagrad: Ten optymalizator osiągnął średnią dokładność (0.5246) i F1 Score (0.6882). Adagrad ma tendencję do zbyt szybkiego zmniejszania szybkości uczenia się, co może prowadzić do zbyt wczesnego zatrzymania procesu optymalizacji. Może to tłumaczyć, dlaczego Adagrad nie osiągnął tak dobrych wyników jak Adam.

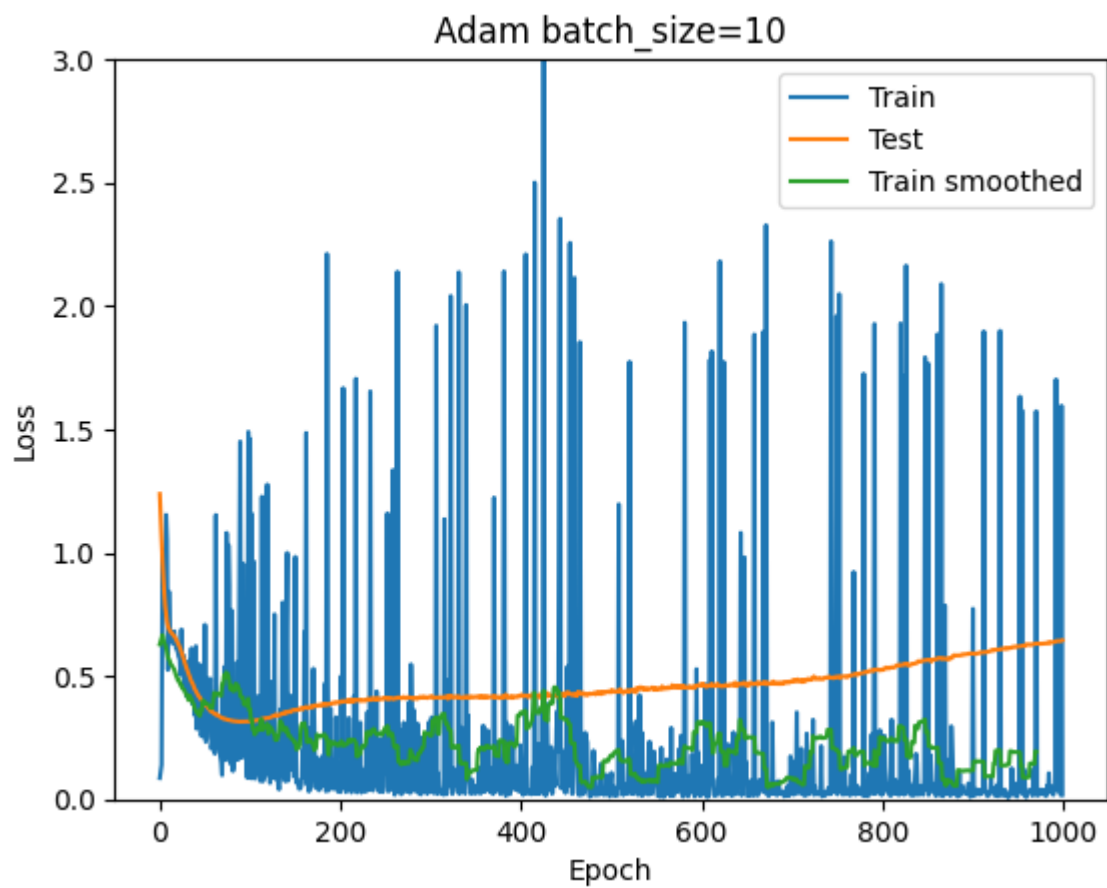
Porównanie w zależności od rozmiaru paczek

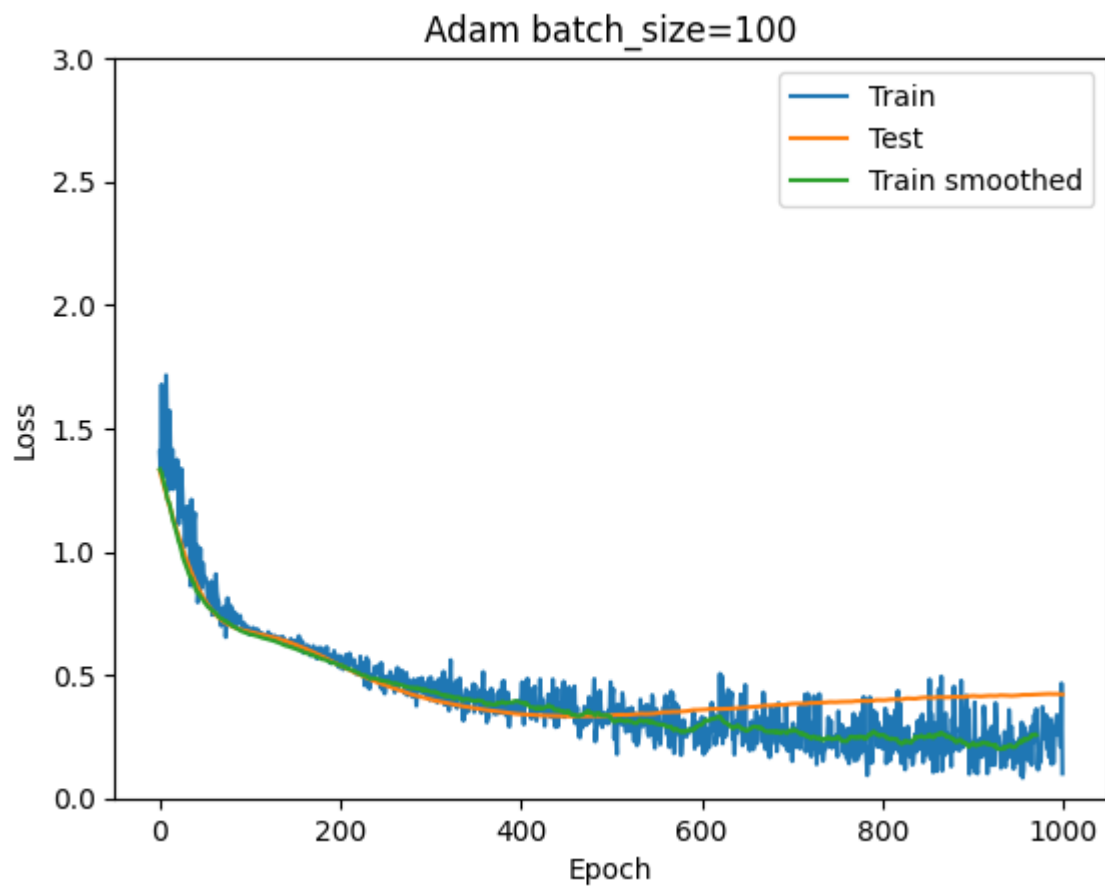
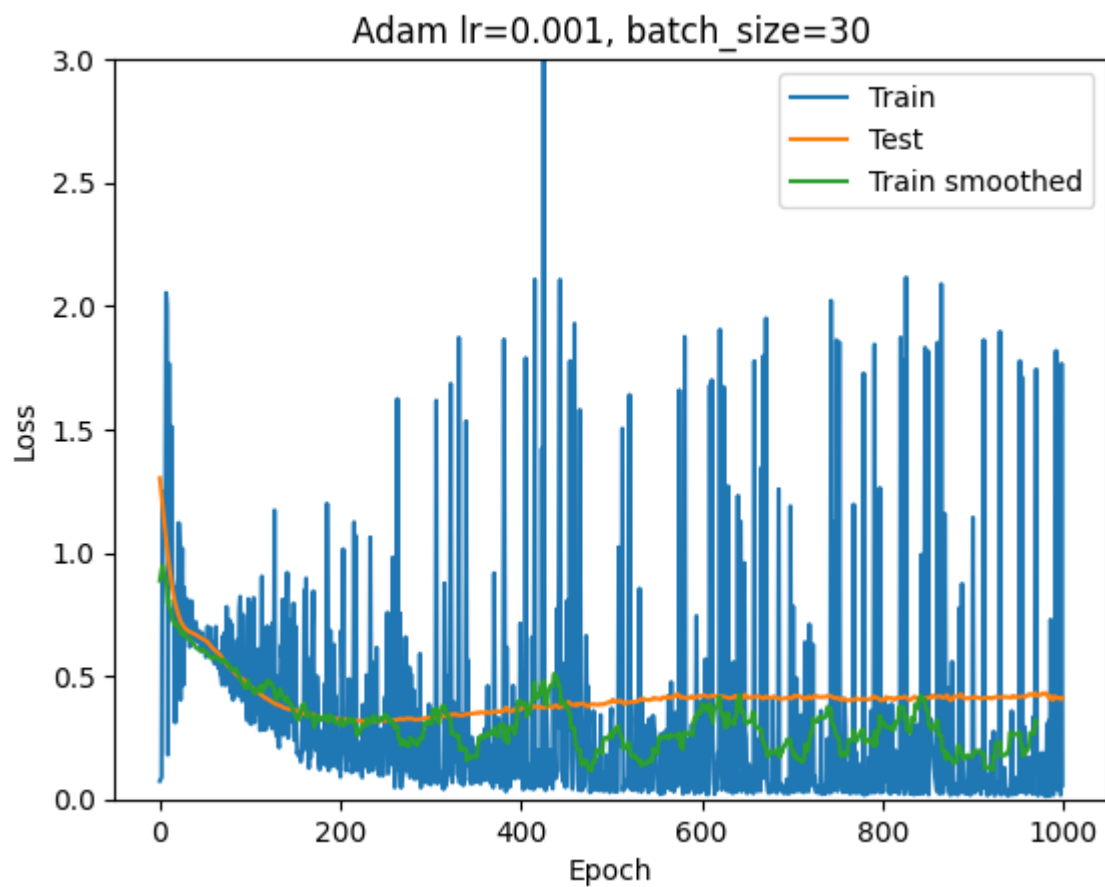
Do porównania wpływu rozmiaru paczek wybrano najlepszy model z poprzedniego badania.

ADAM: batch_size 10	Adam	ADAM: batch_size 100
Epoch 100/1000, Loss: 0.0905	Epoch 100/1000, Loss: 0.3436	Epoch 100/1000, Loss: 0.6648
Epoch 200/1000, Loss: 0.0332	Epoch 200/1000, Loss: 0.1392	Epoch 200/1000, Loss: 0.5303
Epoch 300/1000, Loss: 0.0822	Epoch 300/1000, Loss: 0.3364	Epoch 300/1000, Loss: 0.3983
Epoch 400/1000, Loss: 0.2200	Epoch 400/1000, Loss: 0.7133	Epoch 400/1000, Loss: 0.3578
Epoch 500/1000, Loss: 0.0790	Epoch 500/1000, Loss: 0.3227	Epoch 500/1000, Loss: 0.4712
Epoch 600/1000, Loss: 0.0155	Epoch 600/1000, Loss: 0.0324	Epoch 600/1000, Loss: 0.2900
Epoch 700/1000, Loss: 0.0466	Epoch 700/1000, Loss: 0.1500	Epoch 700/1000, Loss: 0.2769
Epoch 800/1000, Loss: 0.0169	Epoch 800/1000, Loss: 0.0341	Epoch 800/1000, Loss: 0.2368



Epoch 900/1000, Loss: 0.0285	Epoch 900/1000, Loss: 0.0568	Epoch 900/1000, Loss: 0.1299
Epoch 1000/1000, Loss: 0.0180	Epoch 1000/1000, Loss: 0.0567	Epoch 1000/1000, Loss: 0.1006
Accuracy: 0.7869	Accuracy: 0.8361	Accuracy: 0.8197
F1 Score: 0.8000	F1 Score: 0.8485	F1 Score: 0.8308
Precision: 0.7879	Precision: 0.8235	Precision: 0.8182
Recall: 0.8125	Recall: 0.8750	Recall: 0.8438





Wnioski:

Batch size = 10: Model z tym rozmiarem batcha osiągnął najniższą dokładność (0.7869) i F1 Score (0.8000) w porównaniu do innych rozmiarów batcha. Może to wynikać z faktu, że mniejszy rozmiar batcha może prowadzić do większej wariancji w aktualizacjach parametrów. Na wykresie przebiegu uczenia się można zauważyć wspomnianą wariancję, dobre szybkie wyuczenie parametrów, ale widać też zjawisko „overfittingu” gdzie model słabo uogólnia dane testowe.

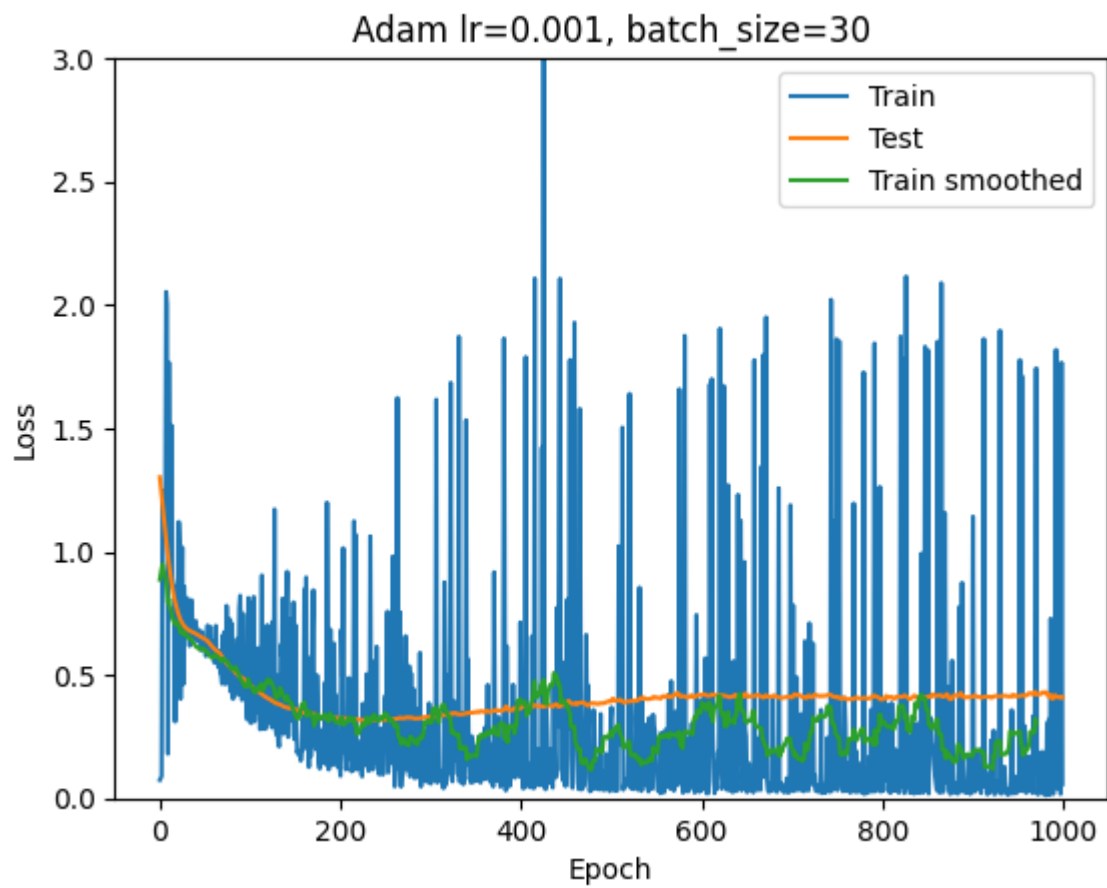
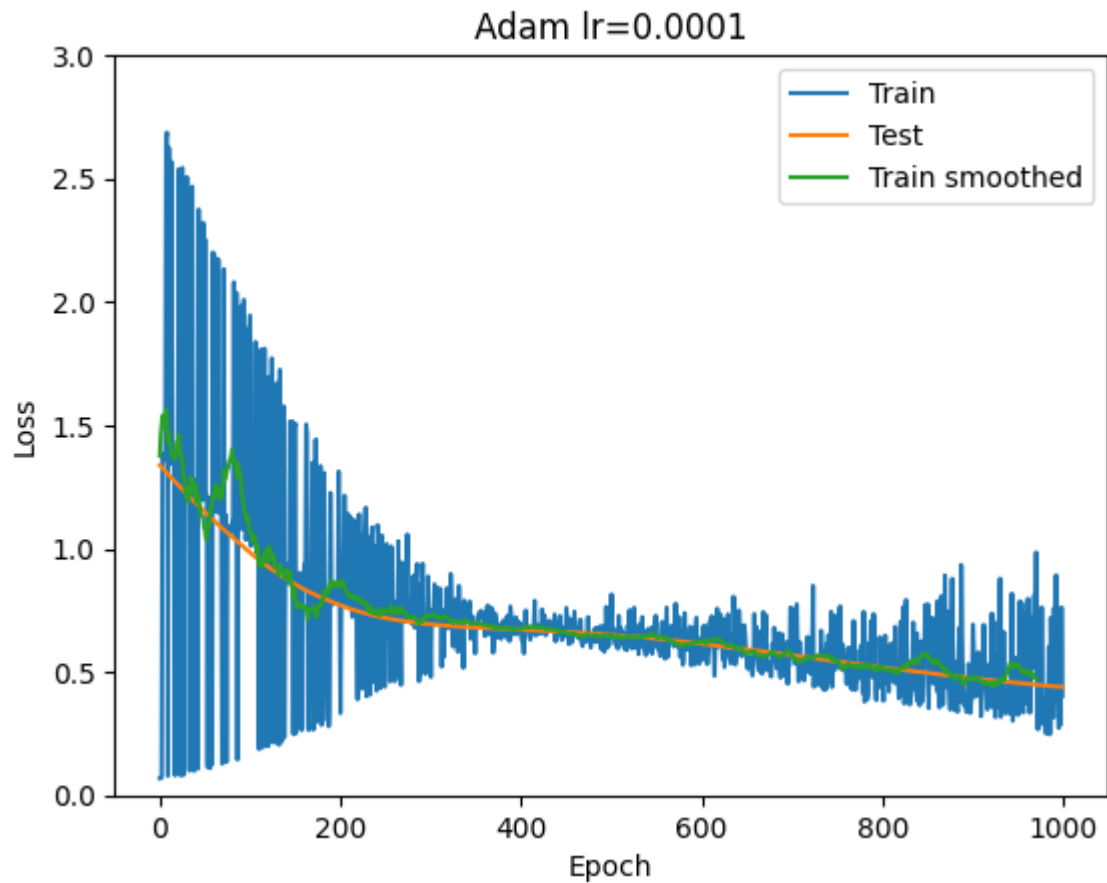
Batch size = 30: (Domyślna wartość batcha) Model z tym rozmiarem batcha osiągnął najwyższą dokładność (0.8361) i F1 Score (0.8485). Przebieg uczenia na danych testowych jest bardziej stabilny i można zauważyć minimalny „overfitting”. Model lepiej uogólnia predykcję danych testowych.

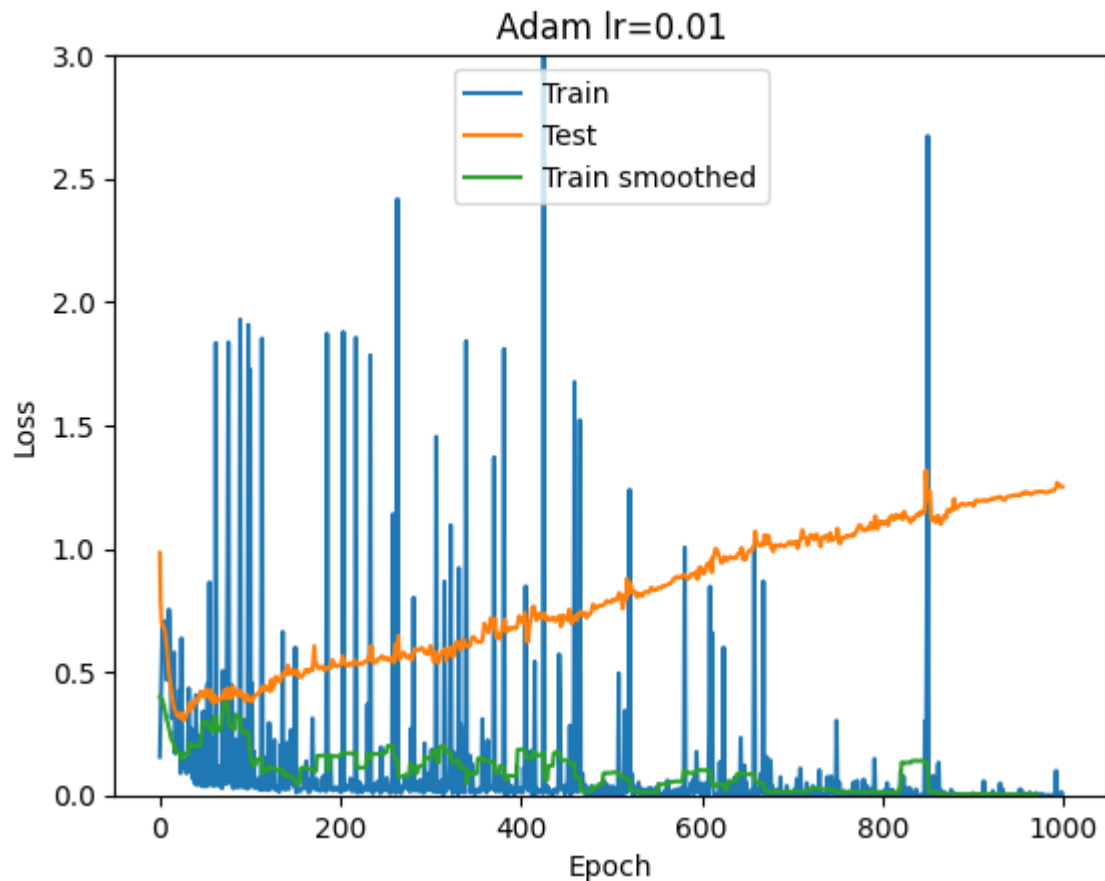
Batch size = 100: Model z tym rozmiarem batcha osiągnął średnią dokładność (0.8197) i F1 Score (0.8308). Większy rozmiar batcha może prowadzić do stabilniejszych aktualizacji parametrów, ale może również prowadzić do utknięcia w gorszym minimum lokalnym. Można zauważyć „wolniejszy” proces uczenia się modelu.

Porównanie w zależności od współczynnika uczenia dla optymalizatorów

Adam:

ADAM 0.0001	Adam	ADAM 0.01
Epoch 100/1000, Loss: 1.0396	Epoch 100/1000, Loss: 0.3436	Epoch 100/1000, Loss: 0.0303
Epoch 200/1000, Loss: 0.7915	Epoch 200/1000, Loss: 0.1392	Epoch 200/1000, Loss: 0.0178
Epoch 300/1000, Loss: 0.6831	Epoch 300/1000, Loss: 0.3364	Epoch 300/1000, Loss: 0.0205
Epoch 400/1000, Loss: 0.6972	Epoch 400/1000, Loss: 0.7133	Epoch 400/1000, Loss: 0.0285
Epoch 500/1000, Loss: 0.7115	Epoch 500/1000, Loss: 0.3227	Epoch 500/1000, Loss: 0.0101
Epoch 600/1000, Loss: 0.5642	Epoch 600/1000, Loss: 0.0324	Epoch 600/1000, Loss: 0.0013
Epoch 700/1000, Loss: 0.7378	Epoch 700/1000, Loss: 0.1500	Epoch 700/1000, Loss: 0.0039
Epoch 800/1000, Loss: 0.4899	Epoch 800/1000, Loss: 0.0341	Epoch 800/1000, Loss: 0.0008
Epoch 900/1000, Loss: 0.4327	Epoch 900/1000, Loss: 0.0568	Epoch 900/1000, Loss: 0.0026
Epoch 1000/1000, Loss: 0.4024	Epoch 1000/1000, Loss: 0.0567	Epoch 1000/1000, Loss: 0.0005
Accuracy: 0.9016	Accuracy: 0.8361	Accuracy: 0.7705
F1 Score: 0.9062	F1 Score: 0.8485	F1 Score: 0.7941
Precision: 0.9062	Precision: 0.8235	Precision: 0.7500
Recall: 0.9062	Recall: 0.8750	Recall: 0.8438





Wnioski:

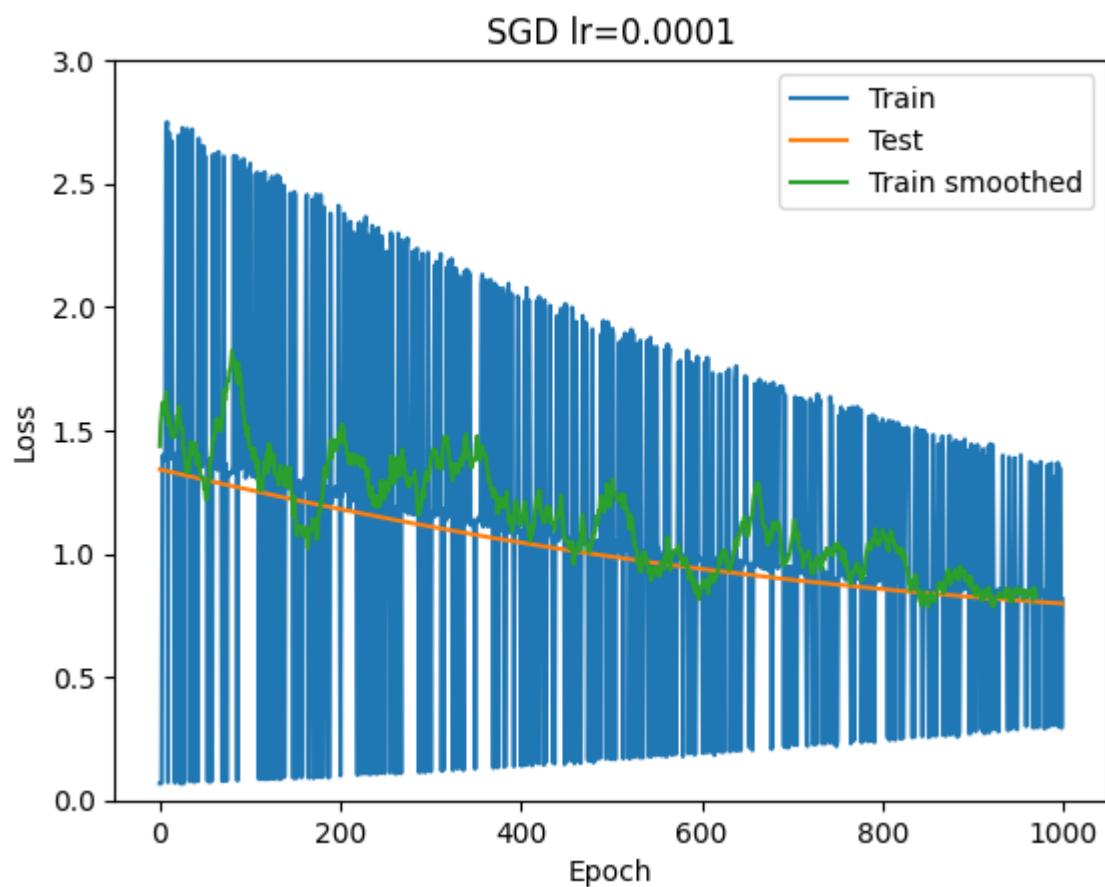
0.0001: Model z tym współczynnikiem uczenia osiągnął najwyższą dokładność (0.9016) i F1 Score (0.9062). Może to wynikać z faktu, że mniejszy współczynnik uczenia pozwala modelowi dokładniej eksplorować przestrzeń parametrów, co może prowadzić do lepszych wyników, ale kosztem dłuższego czasu treningu.

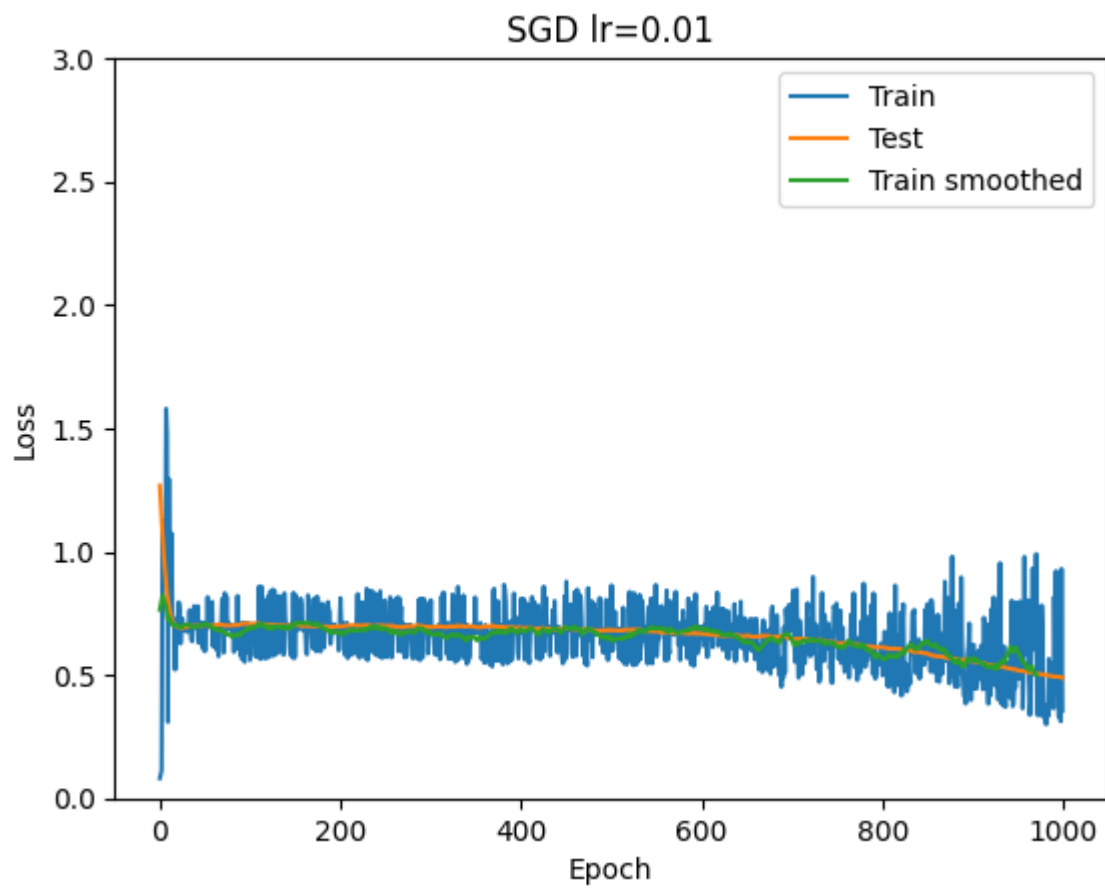
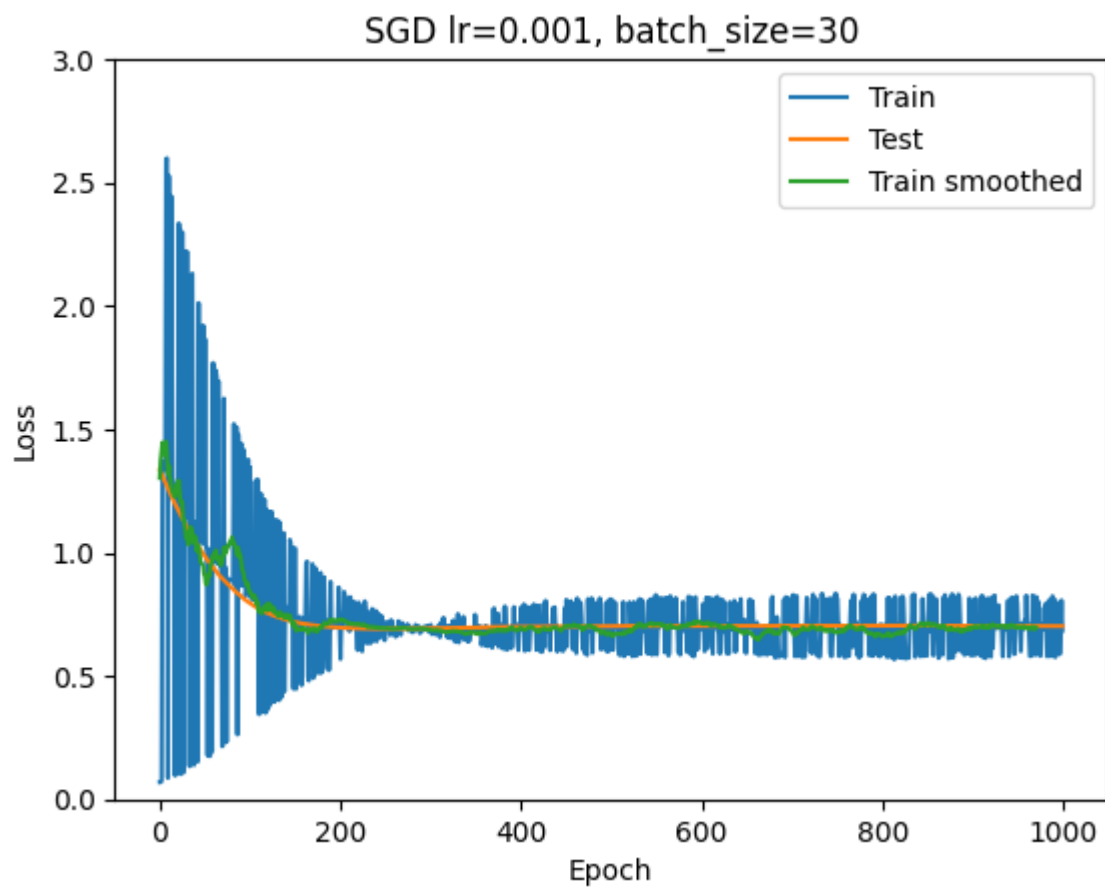
0.001: Model z tym współczynnikiem uczenia osiągnął średnią dokładność (0.8361) i F1 Score (0.8485). Jest to domyślna wartość współczynnika uczenia dla optymalizatora Adam. Domyślna wartość może być za duża dla tego optymalizatora.

0.01: Model z tym współczynnikiem uczenia osiągnął najniższą dokładność (0.7705) i F1 Score (0.7941). Większy współczynnik uczenia może prowadzić do szybkiego przeuczenia się modelu, co widać dobrze na wykresie.

SGD:

SGD 0.0001	SGD	SGD 0.01
Epoch 100/1000, Loss: 1.3276	Epoch 100/1000, Loss: 0.8277	Epoch 100/1000, Loss: 0.7071
Epoch 200/1000, Loss: 1.2663	Epoch 200/1000, Loss: 0.7124	Epoch 200/1000, Loss: 0.6975
Epoch 300/1000, Loss: 1.1568	Epoch 300/1000, Loss: 0.7000	Epoch 300/1000, Loss: 0.7125
Epoch 400/1000, Loss: 1.0820	Epoch 400/1000, Loss: 0.6907	Epoch 400/1000, Loss: 0.6990
Epoch 500/1000, Loss: 1.0190	Epoch 500/1000, Loss: 0.6975	Epoch 500/1000, Loss: 0.7068
Epoch 600/1000, Loss: 0.9755	Epoch 600/1000, Loss: 0.6909	Epoch 600/1000, Loss: 0.6510
Epoch 700/1000, Loss: 0.9264	Epoch 700/1000, Loss: 0.7083	Epoch 700/1000, Loss: 0.7606
Epoch 800/1000, Loss: 0.8975	Epoch 800/1000, Loss: 0.6966	Epoch 800/1000, Loss: 0.6093
Epoch 900/1000, Loss: 1.4354	Epoch 900/1000, Loss: 0.5915	Epoch 900/1000, Loss: 0.5256
Epoch 1000/1000, Loss: 0.8162	Epoch 1000/1000, Loss: 0.6835	Epoch 1000/1000, Loss: 0.3543
Accuracy: 0.5246	Accuracy: 0.4754	Accuracy: 0.8033
F1 Score: 0.6882	F1 Score: 0.0000	F1 Score: 0.8000
Precision: 0.5246	Precision: 0.0000	Precision: 0.8571
Recall: 1.0000	Recall: 0.0000	Recall: 0.7500





Wnioski:

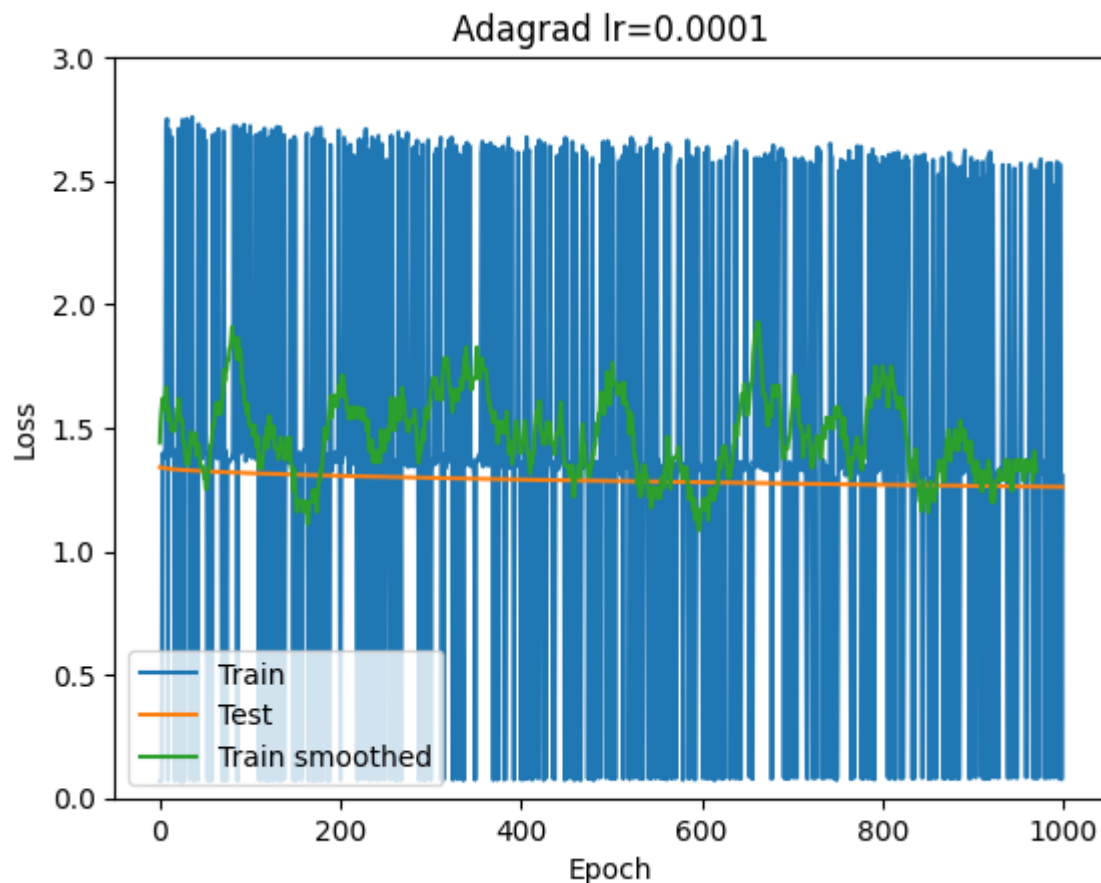
0.0001: Model osiąga niską dokładność (accuracy) 0.5246, ale pełne pokrycie (recall) 1.0000. To sugeruje, że model klasyfikuje wszystkie próbki jako pozytywne, co jest nieodpowiednie. Po wykresie widać, że współczynnik uczenia jest zbyt niski.

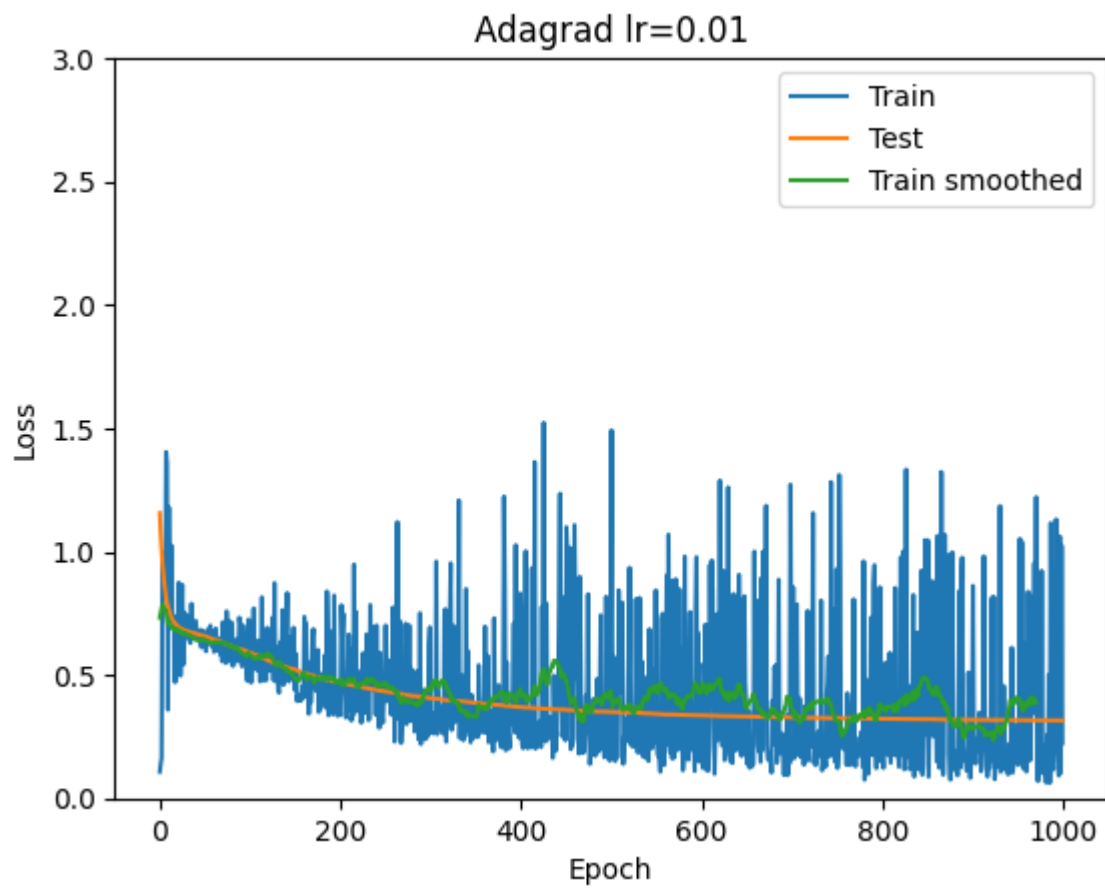
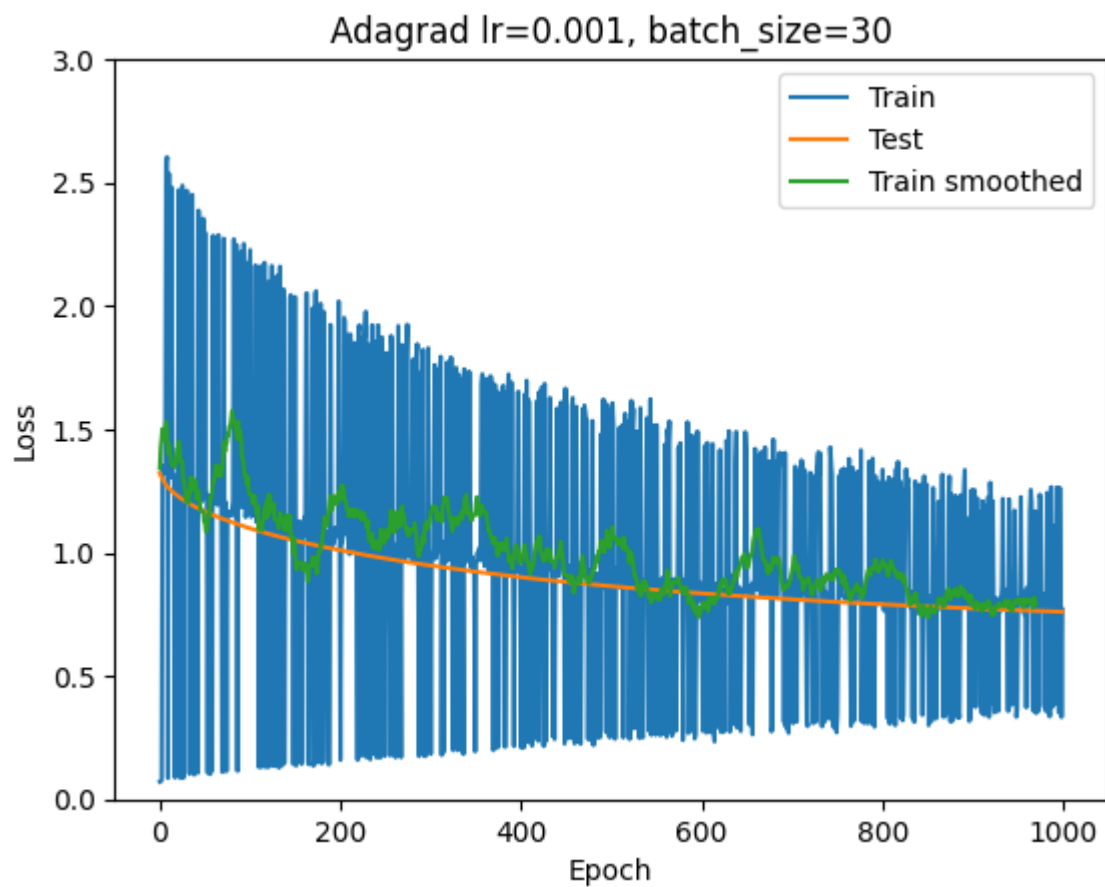
0.001: Model uczy się szybciej ale nadal nie jest w stanie poprawnie klasyfikować żadnej próbki.

0.01: Model osiąga znacznie lepsze wyniki, z dokładnością 0.8033 i F1 Score 0.8000. To sugeruje, że wyższy współczynnik uczenia pozwala modelowi skuteczniej się uczyć.

Adagrad:

Adagrad 0.0001	Adagrad	Adagrad 0.01
Epoch 100/1000, Loss: 1.3900	Epoch 100/1000, Loss: 1.1607	Epoch 100/1000, Loss: 0.5234
Epoch 200/1000, Loss: 1.4017	Epoch 200/1000, Loss: 1.0817	Epoch 200/1000, Loss: 0.3784
Epoch 300/1000, Loss: 1.3520	Epoch 300/1000, Loss: 0.9723	Epoch 300/1000, Loss: 0.6937
Epoch 400/1000, Loss: 1.3407	Epoch 400/1000, Loss: 0.9150	Epoch 400/1000, Loss: 0.8310
Epoch 500/1000, Loss: 1.3268	Epoch 500/1000, Loss: 0.8617	Epoch 500/1000, Loss: 0.9039
Epoch 600/1000, Loss: 1.3477	Epoch 600/1000, Loss: 0.8773	Epoch 600/1000, Loss: 0.1660
Epoch 700/1000, Loss: 1.3251	Epoch 700/1000, Loss: 0.8447	Epoch 700/1000, Loss: 0.5246
Epoch 800/1000, Loss: 1.3543	Epoch 800/1000, Loss: 0.8505	Epoch 800/1000, Loss: 0.1999
Epoch 900/1000, Loss: 2.5575	Epoch 900/1000, Loss: 1.2554	Epoch 900/1000, Loss: 0.2047
Epoch 1000/1000, Loss: 1.3064	Epoch 1000/1000, Loss: 0.7706	Epoch 1000/1000, Loss: 0.2229
Accuracy: 0.5246	Accuracy: 0.5246	Accuracy: 0.9016
F1 Score: 0.6882	F1 Score: 0.6882	F1 Score: 0.9032
Precision: 0.5246	Precision: 0.5246	Precision: 0.9333
Recall: 1.0000	Recall: 1.0000	Recall: 0.8750





Wnioski:

0.0001: Model osiąga niską dokładność (accuracy) 0.5246, ale pełne pokrycie (recall) 1.0000. To sugeruje, że model klasyfikuje wszystkie próbki jako pozytywne, co jest nieodpowiednie. Po wykresie widać, że współczynnik uczenia jest zbyt niski. W porównaniu do SGD tempo uczenia jest jeszcze mniejsze oraz widać bardziej stałą wariancję kosztu dla danych treningowych.

0.001: Model uczy się szybciej ale nadal nie jest w stanie poprawnie klasyfikować żadnej próbki. W porównaniu do SGD to tempo uczenia wygląda podobnie co do SGD przy $lr = 0.001$

0.01: Model osiąga znacznie lepsze wyniki, z dokładnością 0.9016 i F1 wynoszącym 0.9032. To sugeruje, że wyższy współczynnik uczenia pozwala modelowi skuteczniej się uczyć. Są to lepsze wyniki niż dla SGD. Widać, że szybko osiąga mniejsze wartości kosztu dla danych treningowych jak i testowych.

Wnioski:

Jak widać po dokonanych eksperymentach dla różnych optymalizatorów wartości hiperparametrów takich jak rozmiar paczki czy współczynnik uczenia mają różną wagę i w celu znalezienia najlepszych wartości, należy przeprowadzić strojenie każdego z optymalizatorów tak aby wyłonić, najlepszy optymalizator dla danego problemu.