SIECI NEURONOWE

Raport z laboratoriów 1-4

Student: Van Hien Le - 257795

Narzędzia i technologie

- Python, Pandas, Numpy, Scikit-learn, PyTorch
- Jupyter Notebook

Dataset

Zbiór danych używany w tym ćwiczeniu to zbiór danych chorób serca, do którego można uzyskać dostęp pod adresem https://archive.ics.uci.edu/dataset/45/heart+disease.

Analiza eksploracyjna

1. **Import danych**

   Instalacja bibliotek i pobieranie zbioru danych:

```
!pip install ucimlrepo
!pip install scikit-learn
```

```
from ucimlrepo import fetch_ucirepo, list_available_datasets
list_available_datasets()
```

```
heart_disease = fetch_ucirepo(id=45)
X = heart_disease.data.features
y = heart_disease.data.targets
```

   Pierwsze 10 przykładów zbioru danych:

```
dataset = pd.concat([X, y], axis=1)
dataset[0:10]
```

	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

   Kształt zbioru danych:

```
dataset.shape
```

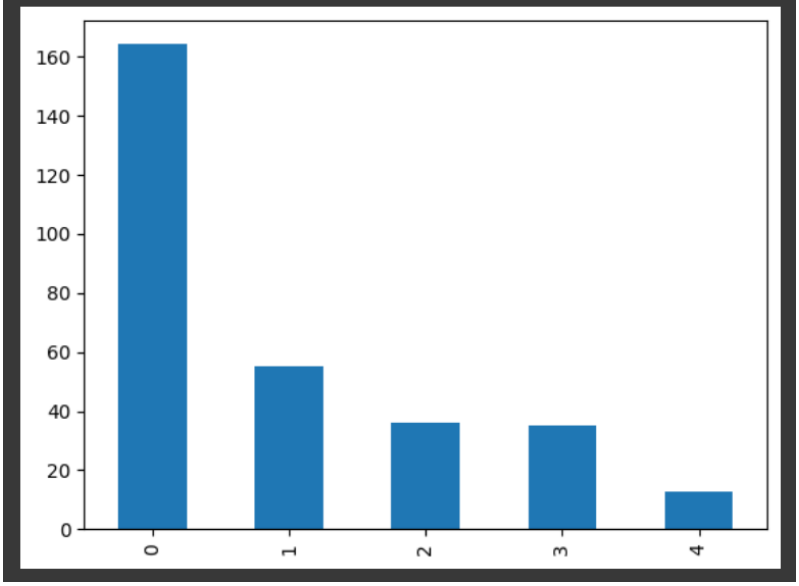
```
(303, 14)
```

2. Balans klas

- Czy zbiór jest zbalansowany pod względem liczby próbek na klasy?

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

```
<Axes: >
```



Zbiór danych nie jest zbalansowany pod względem liczby próbek na klasę. Liczba próbek klasy 0 jest znacznie wyższa niż pozostałych.

3. Cechy liczbowe

- Jakie są średnie i odchylenia cech liczbowych?

  Cechy liczbowe w zbiorze danych to "age", "trestbps", "chol", "thalach" i "oldpeak".

  Średnie i odchylenia:

```
num_features = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']
dataset[num_features].describe().loc[['mean', 'std']]
```

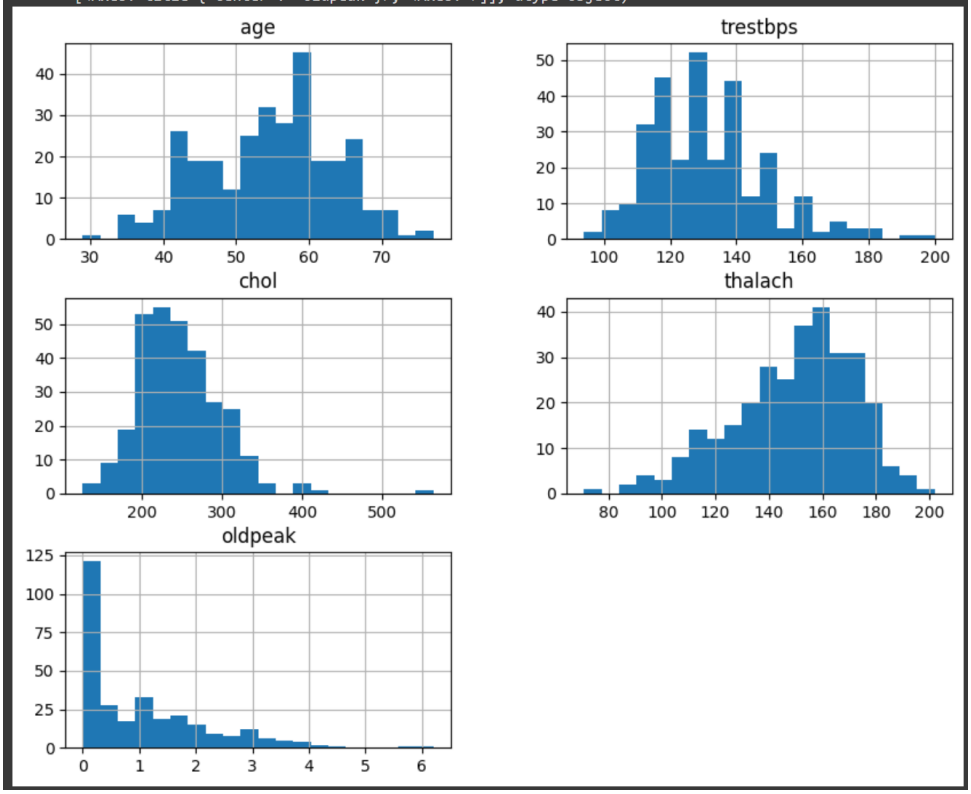
	age	trestbps	chol	thalach	oldpeak
mean	54.438944	131.689769	246.693069	149.607261	1.039604
std	9.038662	17.599748	51.776918	22.875003	1.161075

- Dla cech liczbowych: czy ich rozkład jest w przybliżeniu normalny?

  Histogram cech liczbowych:

```
dataset[num_features].hist(figsize=(10, 8), bins=20)
```

```
array([[<Axes: title={'center': 'age'}>,
        <Axes: title={'center': 'trestbps'}>],
       [<Axes: title={'center': 'chol'}>,
        <Axes: title={'center': 'thalach'}>],
       [<Axes: title={'center': 'oldpeak'}>, <Axes: >]], dtype=object)
```



Patrząc na histogramy, możemy rozpoznać, że rozkład normalny istnieje w cechach "age", "chol" i "thalach", ale wydaje się, że nie istnieje w cechach "trestbps" i "oldpeak".

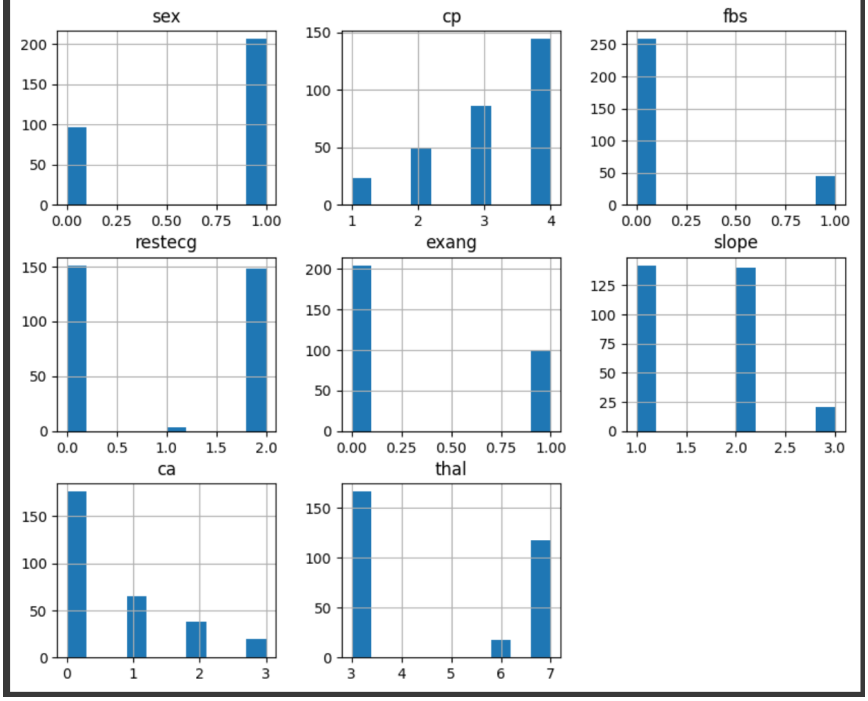
4. Cechy kategoryczne

- Dla cech kategorycznych: czy rozkład jest w przybliżeniu równomierny?
  Cechy kategoryczne w zbiorze danych to "sex", "cp", "fbs", "restecg", "exang", "slope", "ca" i "thal".

  Histogram cech kategorycznych:

```
cat_features = ['sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'ca', 'thal']
dataset[cat_features].hist(figsize=(10, 8))
```

```
array([[<Axes: title={'center': 'sex'}>, <Axes: title={'center': 'cp'}>,
        <Axes: title={'center': 'fbs'}>],
       [<Axes: title={'center': 'restecg'}>,
        <Axes: title={'center': 'exang'}>,
        <Axes: title={'center': 'slope'}>],
       [<Axes: title={'center': 'ca'}>, <Axes: title={'center': 'thal'}>,
        <Axes: >]], dtype=object)
```



Patrząc na histogramy, możemy zauważyć, że cechy kategoryczne tego zbioru danych nie mają równomiernego rozkładu.

5. Brakujące dane

- Czy w zbiorze danych brakuje jakichś danych?

```
dataset.isnull().sum()
```

```
age         0
sex         0
cp          0
trestbps    0
chol        0
fbs         0
restecg     0
thalach     0
exang       0
oldpeak     0
slope       0
ca          4
thal        2
num         0
dtype: int64
```

W zbiorze danych brakuje 6 wartości, 4 w kolumnie "ca" i 2 w kolumnie "thal".

- Jaką strategię można zastosować, aby zastąpić brakujące dane?

  Dla cechy numerycznej "ca" możemy zastąpić brakujące dane wartościami mediany. Dla cechy kategorycznej 'thal' możemy zastąpić brakujące wartości najczęściej występującą wartością tej cechy:

```
ca_median = dataset['ca'].median()
ca_median
```

```
0.0
```

```
thal_mode = dataset['thal'].mode()[0]
thal_mode
```

```
3.0
```

```
dataset['ca'].fillna(ca_median, inplace=True)
dataset['thal'].fillna(thal_mode, inplace=True)
```

Kod przekształcający zbiór danych w macierz cech liczbowych (przykłady x cechy)

W przypadku cech kategorycznych używam ‘one hot encoder’ do kodowania wartości:

```
mapped_column_names = {'cp_1': 'cp_typical_angina', 'cp_2': 'cp_atypical_angina', 'cp_3':
'cp_non_anginal_pain', 'cp_4': 'cp_asymptomatic',
                       'restecg_0': 'restecg_normal', 'restecg_1': 'restecg_wave_abnormality',
'restecg_2': 'restecg_definite_lvh',
                       'thal_3': 'thal_normal', 'thal_6': 'thal_fixed', 'thal_7':
'thal_reversable',
                       'slope_1': 'slope_upsloping', 'slope_2': 'slope_flat', 'slope_3':
'slope_downsloping'}
```

```
column_names = ['cp', 'restecg', 'thal', 'slope']
dataset[column_names] = dataset[column_names].astype('int64')
dataset_encoded = pd.get_dummies(dataset, columns=column_names)
dataset_encoded.rename(columns=mapped_column_names, inplace=True)
```

Kształt zbioru danych po przekształceniu to (303, 23)

Regresja logistyczna

1. **Przygotowanie i wstępne przetwarzanie danych**
   Ponieważ celem jest utworzenie klasyfikatora binarnego, przedefiniowujemy klasy w następujący sposób:
   - 0: brak chorób serca
   - 1: ma chorobę serca

```
X = dataset_encoded.drop('num', axis=1)
y = dataset_encoded['num']

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

```
0    164
1    139
Name: num, dtype: int64
```

W kroku wstępnego przetwarzania danych używamy standardowego skalera ze scikit-learn do skalowania cech numerycznych:

```
from sklearn import preprocessing
scaler = preprocessing.StandardScaler().fit(X[num_features])
X[num_features] = scaler.transform(X[num_features])
```

2. Implementacja regresji logistycznej

```
import numpy as np

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

class LogisticRegression():
  def __init__(self, learning_rate=1e-3, max_iterations=500,
batch_size=5, convergence_threshold=1e-5, random_state=0):
    self.learning_rate = learning_rate
    self.max_iterations = max_iterations
    self.batch_size = batch_size
    self.convergence_threshold = convergence_threshold
    self.random_state = random_state
    self.cost_list = []
    self.theta = None

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

  def fit(self, X, y):
    self.theta = np.zeros(X.shape[1] + 1)
    X_bias = np.c_[np.ones(X.shape[0]), X]
    np.random.seed(self.random_state)
    for epoch in range(self.max_iterations):
      loss_batch = []
      for i in range(0, len(y), self.batch_size):
        X_batch = X_bias[i:i + self.batch_size]
        y_batch = y[i:i + self.batch_size]
```

```
        preds = sigmoid(X_batch.dot(self.theta))
        gradient = X_batch.T.dot(preds - y_batch) / len(y_batch)
        self.theta -= self.learning_rate * gradient
        loss_batch.append(np.mean(self.loss(y_batch, preds)))

      cost = np.mean(loss_batch)
      if (len(self.cost_list) > 0 and abs(cost - self.cost_list[-1]) <
self.convergence_threshold):
        break
      self.cost_list.append(cost)

  def predict(self, X):
    X_bias = np.c_[np.ones(X.shape[0]), X]
    preds = sigmoid(X_bias.dot(self.theta))
    y_preds = [1 if pred > 0.5 else 0 for pred in preds]
    return y_preds
```

+ Używana funkcja aktywacji to funkcja sigmoidalna. Hiperparametry modelu to “learning_rate”, “max_iterations”, “batch_size”, “convergence_threshold”, i “random_state”.

+ Funkcja strat jest funkcją entropii krzyżowej.

+ Model uczy się według rozmiaru paczki, który jest przekazywany jako hiperparametr.

+ Proces uczenia się zostaje zakończony, gdy osiągnie maksymalną liczbę iteracji lub osiągnie próg zbieżności.

3. Weryfikacja modelu

Podział danych na dane treningowe i dane testowe:

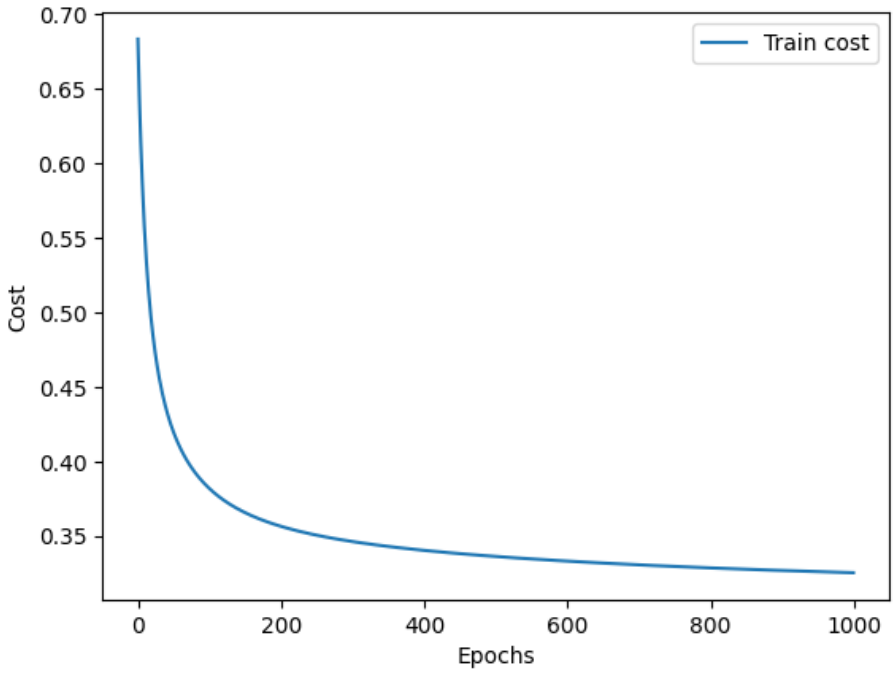
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Dopasowanie modelu i metryki:

```
model = LogisticRegression(learning_rate=2e-3, max_iterations=1000, batch_size=5, convergence_threshold=1e-5, random_state=42)
model.fit(X_train, y_train)

y_preds = model.predict(X_test)

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("Recall: %.2f" % recall_score(y_test, y_preds))
print("Precision: %.2f" % precision_score(y_test, y_preds))
print("F1: %.2f" % f1_score(y_test, y_preds))
print("Accuracy: %.2f" % accuracy_score(y_test, y_preds))
```

```
Recall: 0.91
Precision: 0.85
F1: 0.88
Accuracy: 0.87
```

Wykres kosztu trenowania w każdej iteracji:



Sieć neuronowa

1. **Architektura modelu**
   Siecią neuronową w tym zadaniu jest wielowarstwowy perceptron (MLP) służący do binarnej klasyfikacji chorób serca. Kluczowe komponenty modelu:
   - Model składa się z warstwy wejściowej, po której następuje wiele warstw ukrytych i warstwa wyjściowa.
   - Używana funkcja aktywacji to funkcja sigmoidalna.
   - Wagi i biasy są randomizowane w określonym zakresie odchylenia standardowego.
   - Model wykorzystuje entropię krzyżową jako funkcję kosztu i aktualizuje swoje wagi i biasy przez mini-batch gradient descent.
2. **Implementacja modelu**

```
class MLP():
  def __init__(self, hidden_layers, batch_size=50, learning_rate=1e-3,
stand_dev=1.0, max_iter=500, random_state=42):
    self.hidden_layers = hidden_layers
    self.num_layers = len(hidden_layers) + 1
    self.batch_size = batch_size
    self.learning_rate = learning_rate
    self.stand_dev = stand_dev
    self.max_iter = max_iter
    self.random_state = random_state
    self.train_cost_list = []
    self.test_cost_list = []
```

```
  def wandb_init(self, input_size, output_size):
    self.input_size = input_size
    self.output_size = output_size
    np.random.seed(self.random_state)
    self.weights = [np.random.normal(0, self.stand_dev, (input_size,
self.hidden_layers[0]))]
    self.bias = [np.random.normal(0, self.stand_dev, ((1, self.hidden_layers[0])))]

    for i in range(self.num_layers - 2):
      self.weights.append(np.random.normal(0, self.stand_dev,
(self.hidden_layers[i], self.hidden_layers[i + 1])))
      self.bias.append(np.random.normal(0, self.stand_dev, ((1,
self.hidden_layers[i + 1]))))

    self.weights.append(np.random.normal(0, self.stand_dev, (self.hidden_layers[-
1], output_size)))
    self.bias.append(np.random.normal(0, self.stand_dev, ((1, output_size))))

  def sigmoid(self, x):
    x = np.array(x, dtype=np.float128)
    return 1 / (1 + np.exp(-x))

  def sigmoid_derivative(self, y):
    return y * (1 - y)

  def loss(self, y, pred):
    return -np.sum(y * np.log(pred) + (1 - y) * np.log(1 - pred), axis=1)

  def feedforward(self, X):
    layer_outputs = [X]
    for i in range(self.num_layers):
      z = np.dot(layer_outputs[-1], self.weights[i]) + self.bias[i]
      a = self.sigmoid(z)
      layer_outputs.append(a)
    return layer_outputs

  def backward(self, y, layer_outputs):
    deltas = [None] * self.num_layers

    error = y - layer_outputs[-1]

    deltas[-1] = error * self.sigmoid_derivative(layer_outputs[-1])

    for i in range(self.num_layers - 2, -1, -1):
        deltas[i] = deltas[i + 1].dot(self.weights[i + 1].T) *
self.sigmoid_derivative(layer_outputs[i + 1])
```

```
    for i in range(self.num_layers - 1, -1, -1):
      self.weights[i] += self.learning_rate * np.dot(layer_outputs[i].T, deltas[i])
      self.bias[i] += self.learning_rate * np.sum(deltas[i])

  def fit(self, X, y, X_test, y_test):
    self.wandb_init(X.shape[1], 1)
    if self.batch_size == None:
      self.batch_size = X.shape[1]
    for epoch in range(self.max_iter):
      X, y = shuffle(X, y, random_state=self.random_state)

      train_loss_batch = []
      for i in range(0, len(y), self.batch_size):
        X_batch = X[i:i + self.batch_size]
        y_batch = y[i:i + self.batch_size]

        layer_outputs = self.feedforward(X_batch)
        self.backward(y_batch, layer_outputs)

        train_loss_batch.append(np.mean(self.loss(y_batch, layer_outputs[-1])))

      self.train_cost_list.append(np.mean(train_loss_batch))
      self.test_cost_list.append(np.mean(self.loss(y_test,
self.feedforward(X_test)[-1])))

  def predict(self, X):
    outputs = self.feedforward(X)[-1]
    y_preds = [1 if pred > 0.5 else 0 for pred in outputs]
    return y_preds

  def get_metrics(self, X, y):
    preds = self.predict(X)
    print("Recall: %.2f" % recall_score(y, preds))
    print("Precision: %.2f" % precision_score(y, preds))
    print("F1: %.2f" % f1_score(y, preds))
    print("Accuracy: %.2f" % accuracy_score(y, preds))
```

3. **Konfiguracje eksperymentów**
   a. **Wstępne przetwarzanie danych**

   Przed podziałem zbioru danych na zbiór uczący i testowy normalizujemy dane liczbowe przy użyciu skalowania min-max:

```
X_normalized = X.copy()
X_normalized[num_features] = (X[num_features] - X[num_features].min()) / (X[num_features].max() - X[num_features].min())
```

   Podział zbioru danych na zbiór uczący i zbiór testowy:

```
X_train, X_test, y_train, y_test = train_test_split(X_normalized, y, test_size=0.2, random_state=42)
X_train_non_norm, X_test_non_norm, y_train_non_norm, y_test_non_norm = train_test_split(X, y, test_size=0.2, random_state=42)
```

   b. **Konfiguracje hiperparametrów**

   Domyślna konfiguracja:

```
default_conf = {
  "max_iter": 1000,
  "learning_rate": 10e-4,
  "stand_dev": 1.0,
  "batch_size": 50,
  "random_state": 42,
  "hidden_layers": [15, 15]
}
```

Konfiguracje do testowania różnych wymiarów warstw ukrytych:

```
# Hidden layer dimensions configs
hld_config1 = {
  **default_conf,
  "hidden_layers": [8, 15]
}
hld_config2 = {
  **default_conf,
  "hidden_layers": [2, 2]
}
hld_config3 = {
  **default_conf,
  "hidden_layers": [30, 30]
}
```

Konfiguracje do testowania różnych szybkości uczenia się:

```
#Learning rates configs
lr_config1 = {
  **default_conf,
  "learning_rate": 0.1
}
lr_config2 = {
  **default_conf,
  "learning_rate": 10e-4
}
lr_config3 = {
  **default_conf,
  "learning_rate": 10e-5
}
```

Konfiguracje do testowania różnych odchyleń standardowych podczas inicjalizacji wag:

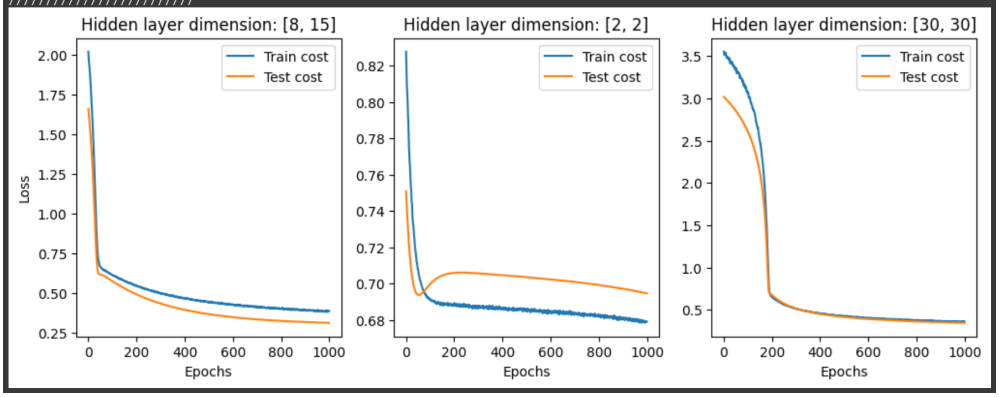
```
#Standard deviations configs
sd_config1 = {
    **default_conf,
    "stand_dev": 0.1
}
sd_config2 = {
    **default_conf,
    "stand_dev": 1.0
}
sd_config3 = {
    **default_conf,
    "stand_dev": 10.0
}
```

Konfiguracje do testowania różnej liczby ukrytych warstw:

```python
#Number of layers configs
nol_config1 = {
    **default_conf,
    "hidden_layers": [15]
}
nol_config2 = {
    **default_conf,
    "hidden_layers": [15, 15]
}
nol_config3 = {
    **default_conf,
    "hidden_layers": [15, 15, 15, 15]
}
```

4. **Wyniki**
   a. **Wymiary warstw ukrytych**

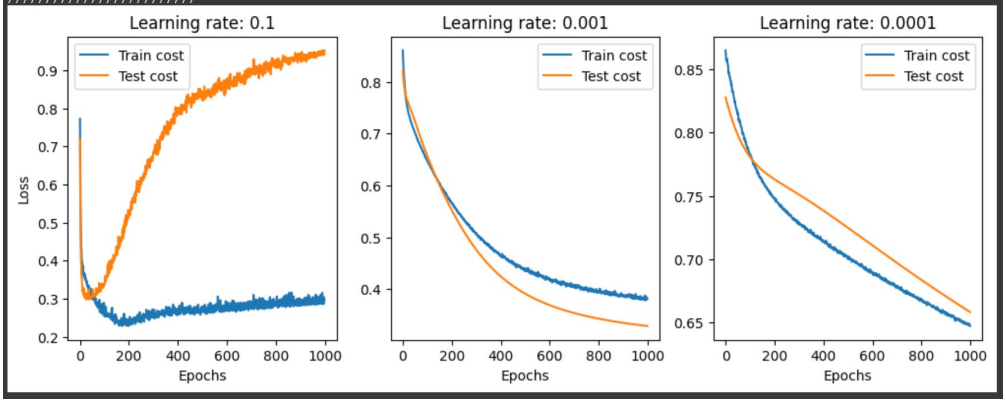
```
Hidden layer dimension: [8, 15]
Recall: 0.94
Precision: 0.91
F1: 0.92
Accuracy: 0.92
//////////////////////
Hidden layer dimension: [2, 2]
Recall: 0.00
Precision: 0.00
F1: 0.00
Accuracy: 0.48
//////////////////////
Hidden layer dimension: [30, 30]
Recall: 0.81
Precision: 0.93
F1: 0.87
Accuracy: 0.87
//////////////////////
```



Model z ukrytymi warstwami [2, 2] ma dość wysokie koszty trenowania i testowania i nie może uczyć się wzorca danych (wyniki Recall, Precision i F1 wynoszą 0,00). Dzieje się tak dlatego, że architektura modelu jest zbyt prosta.
Pozostałe modele dość dobrze demonstrują zdolność do uczenia się wzorca danych.

   b. **Wartości współczynników uczenia się**

```
Learning rate: 0.1
Recall: 0.78
Precision: 0.83
F1: 0.81
Accuracy: 0.80
//////////////////////
Learning rate: 0.001
Recall: 0.91
Precision: 0.91
F1: 0.91
Accuracy: 0.90
//////////////////////
Learning rate: 0.0001
Recall: 0.50
Precision: 0.73
F1: 0.59
Accuracy: 0.64
//////////////////////
```



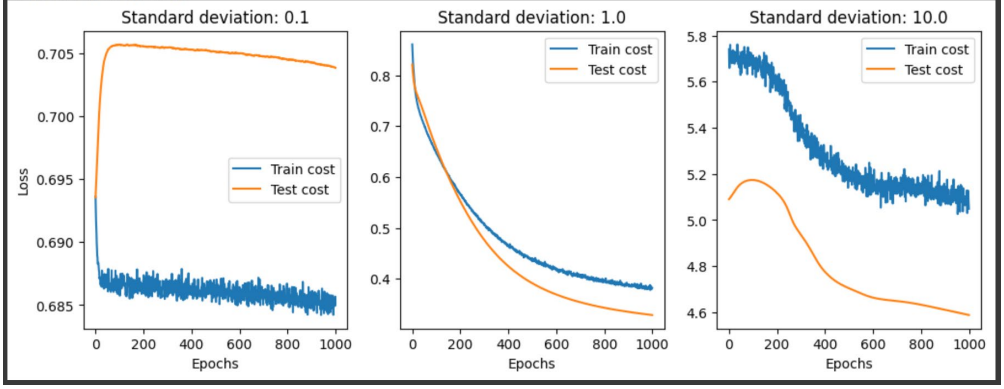
Model ze współczynnikiem uczenia się 0,1 wykazuje nadmierne dopasowanie i oscylacje, koszt trenowania i koszt testowania spadają zbyt szybko i zaczynają ponownie rosnąć, model nie jest zbieżny.

Z drugiej strony, model ze współczynnikiem uczenia się 0,0001 zajmuje dużo czasu, aby się zbiegać, wymaga wielu epok, aby osiągnąć akceptowalną wydajność.

Model ze współczynnikiem uczenia się 0,001 jest stabilny i może dobrze uogólniać na niewidoczne dane.

c. Inicjalizacja wag z odchyleniem standardowym

```
Standard deviation: 0.1
Recall: 0.00
Precision: 0.00
F1: 0.00
Accuracy: 0.48
//////////////////////
Standard deviation: 1.0
Recall: 0.91
Precision: 0.91
F1: 0.91
Accuracy: 0.90
//////////////////////
Standard deviation: 10.0
Recall: 0.88
Precision: 0.56
F1: 0.68
Accuracy: 0.57
//////////////////////
```

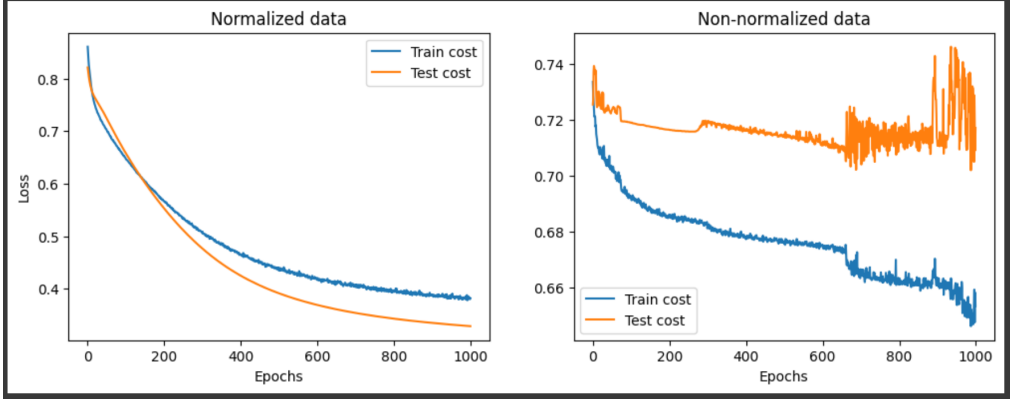


Model z odchyleniem standardowym wynoszącym 10,0 może wymagać wielu epok do zbieżności, ponieważ koszt trenowania i koszt testowania są nadal bardzo wysokie w

pierwszych 1000 epok. Model w środku może się efektywnie zbiegać i wymaga mniejszej liczby epok, aby osiągnąć dobrą wydajność. Model po lewej stronie nie może się zbiegać, prawdopodobnie z powodu eksplozji gradientów podczas propagacji wstecznej, co spowodowało niestabilne aktualizacje wag.

d. Normalizacja danych

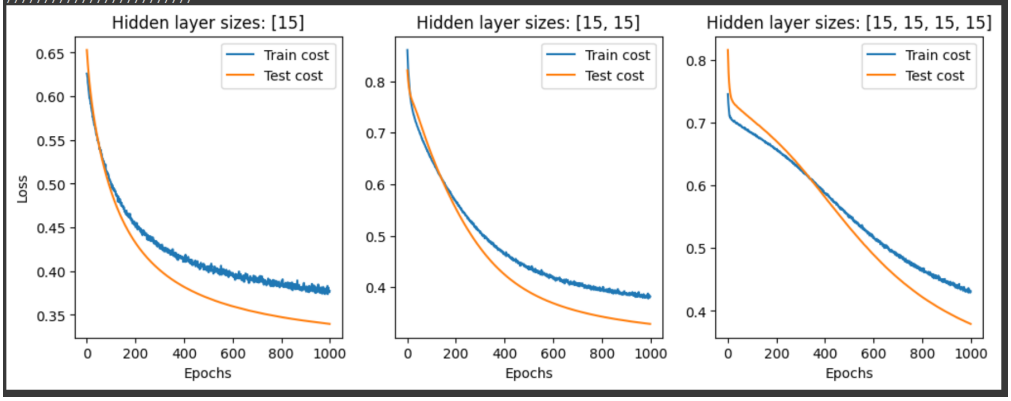
```
Normalized data
Recall: 0.91
Precision: 0.91
F1: 0.91
Accuracy: 0.90
//////////////////////
Non-normalized data
Recall: 0.69
Precision: 0.55
F1: 0.61
Accuracy: 0.54
```



Patrząc na dwa wykresy, wiemy, że model nie może uczyć się na nieznormalizowanych danych, ponieważ model bardzo oscyluje, zwłaszcza gdy zwiększa się liczba epok. W przypadku danych znormalizowanych model może uczyć się całkiem dobrze przy bardzo małych oscylacjach.

e. Liczba ukrytych warstw

```
Hidden layer sizes: [15]
Recall: 0.84
Precision: 0.90
F1: 0.87
Accuracy: 0.87
//////////////////////
Hidden layer sizes: [15, 15]
Recall: 0.91
Precision: 0.91
F1: 0.91
Accuracy: 0.90
//////////////////////
Hidden layer sizes: [15, 15, 15, 15]
Recall: 0.94
Precision: 0.91
F1: 0.92
Accuracy: 0.92
//////////////////////
```



W rezultacie model z największą liczbą ukrytych wartsw [15, 15, 15, 15] ma najlepszą wydajność z wynikiem Recall wynoszącym 0,94. Wygląda na to, że wraz ze wzrostem

liczby warstw wydajność modelu ma tendencję do poprawy, zwłaszcza pod względem metryk Recall i Accuracy.

Odtworzenie architektury sieci w PyTorch z optimizerami

1. **Architektura sieci**

```
class MLP(nn.Module):
  def __init__(self, input_size, hidden_sizes, output_size):
    torch.manual_seed(42)
    super(MLP, self).__init__()
    self.input_layer = nn.Linear(input_size, hidden_sizes[0])

    self.hidden_layers = nn.ModuleList()
    for i in range(len(hidden_sizes) - 1):
      self.hidden_layers.append(nn.Linear(hidden_sizes[i],
hidden_sizes[i + 1]))

    self.output_layer = nn.Linear(hidden_sizes[-1], output_size)

    self.sigmoid = nn.Sigmoid()

  def forward(self, X):
    X = self.sigmoid(self.input_layer(X))

    for hidden_layer in self.hidden_layers:
      X = self.sigmoid(hidden_layer(X))

    X = self.sigmoid(self.output_layer(X))
    return X
```

- Zdefiniowana sieć neuronowa odziedziczona z modułu nn.Module zawiera 2 następujące metody:
+ **init**: Aby zainicjować parametry modelu
+ **forward**: Aby wysłać sygnał do przodu i obliczyć predykcje

2. **Pętla treningowa**

```
def train_one_epoch(model, optimizer, loss_fn, data_loader):
  train_loss = 0.0
  for (index, batch) in enumerate(data_loader):
    X, y = batch
    optimizer.zero_grad()
    outputs = model.forward(X)
    loss = loss_fn(outputs, y)
    loss.backward()
```

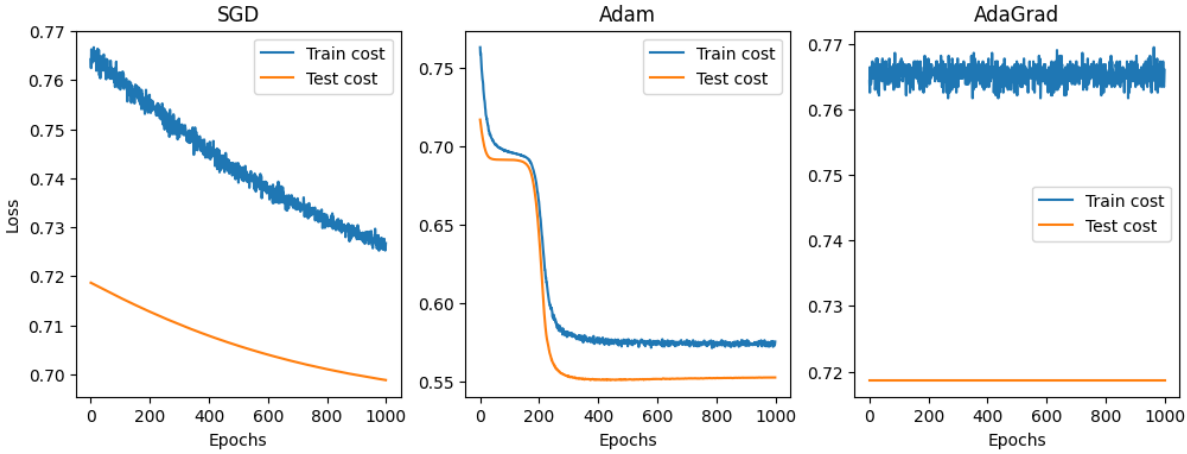
```
    optimizer.step()
    train_loss += loss.item()
  avg_loss = train_loss / len(data_loader)
  return avg_loss

def train(model, optimizer, loss_fn, num_epochs, batch_size, X_train,
y_train, X_test, y_test):
  train_loss_list = []
  test_loss_list = []
  train_data_loader =
DataLoader(TensorDataset(torch.tensor(X_train.values).float(),
torch.tensor(y_train).float()), batch_size=batch_size, shuffle=True)
  test_data_loader =
DataLoader(TensorDataset(torch.tensor(X_train.values).float(),
torch.tensor(y_train).float()), batch_size=batch_size, shuffle=True)

  for i in range(num_epochs):
    loss = train_one_epoch(model, optimizer, loss_fn,
train_data_loader)
    train_loss_list.append(loss)

    with torch.no_grad():
      outputs = model(torch.tensor(X_test.values).float())
      test_loss = loss_fn(outputs.squeeze(),
torch.tensor(y_test).float().squeeze())
      test_loss_list.append(test_loss)

  return train_loss_list, test_loss_list
```

- Funkcja **train_one_epoch**:
+ Trenuje model przez jedną epokę (przebieg przez cały zbiór danych treningowych).
+ Oblicza i zwraca średnią stratę treningową dla tej epoki.
+ Wykorzystuje optymalizator, funkcję straty i DataLoader do iteracyjnego treningu modelu na batchach danych.
- Funkcja **train**:
+ Przeprowadza trening modelu przez określoną liczbę epok.
+ Oblicza i zwraca listy strat treningowych i testowych dla każdej epoki.
+ Wykorzystuje funkcję **train_one_epoch** do treningu modelu w każdej epoce.
+ Przyjmuje parametry: model, optymalizator, funkcję straty, liczbę epok, rozmiar batcha danych oraz dane treningowe i testowe.

3. **Konfiguracja i procedura eksperymentu**
- Wybrane 3 optimizery: SGD, Adam, AdaGrad
- Domyślne parametry modelów:
+ hidden_sizes: [15, 15, 15, 15]
+ num_epochs: 1000
- Dla każdego optimizera, testowane będą różne kombinacji konfiguracji parametrów, w tym:
+ batch_size: 10, 50 i 200
+ learning_rate: 0.01, 0.001, 0.0001

4. Wyniki eksperymentu

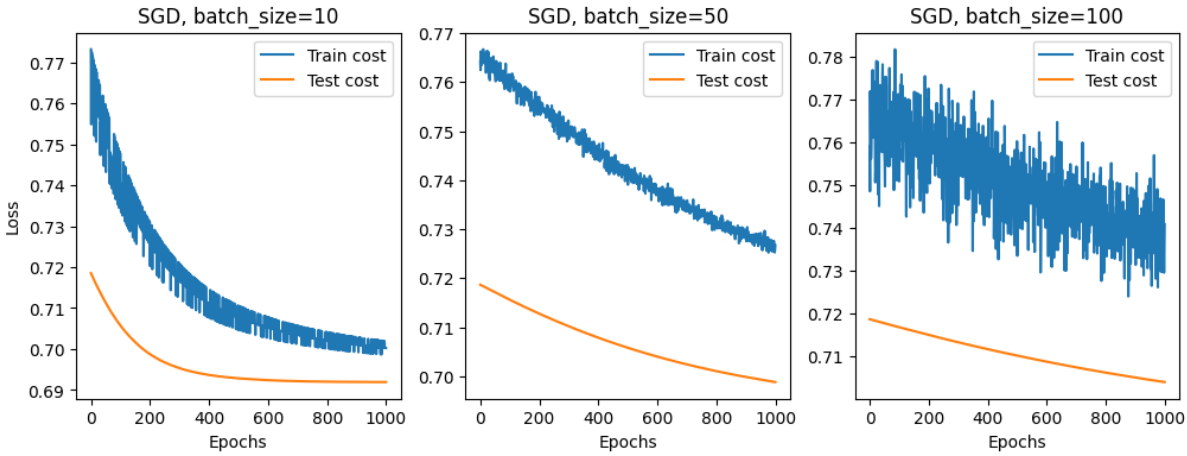
- Eksperyment 3 optimizerów z domyślnymi parametrami:
    + hidden_sizes: [15, 15, 15, 15]
    + num_epochs: 1000
    + batch_size: 50
    + learning_rate: 0.001

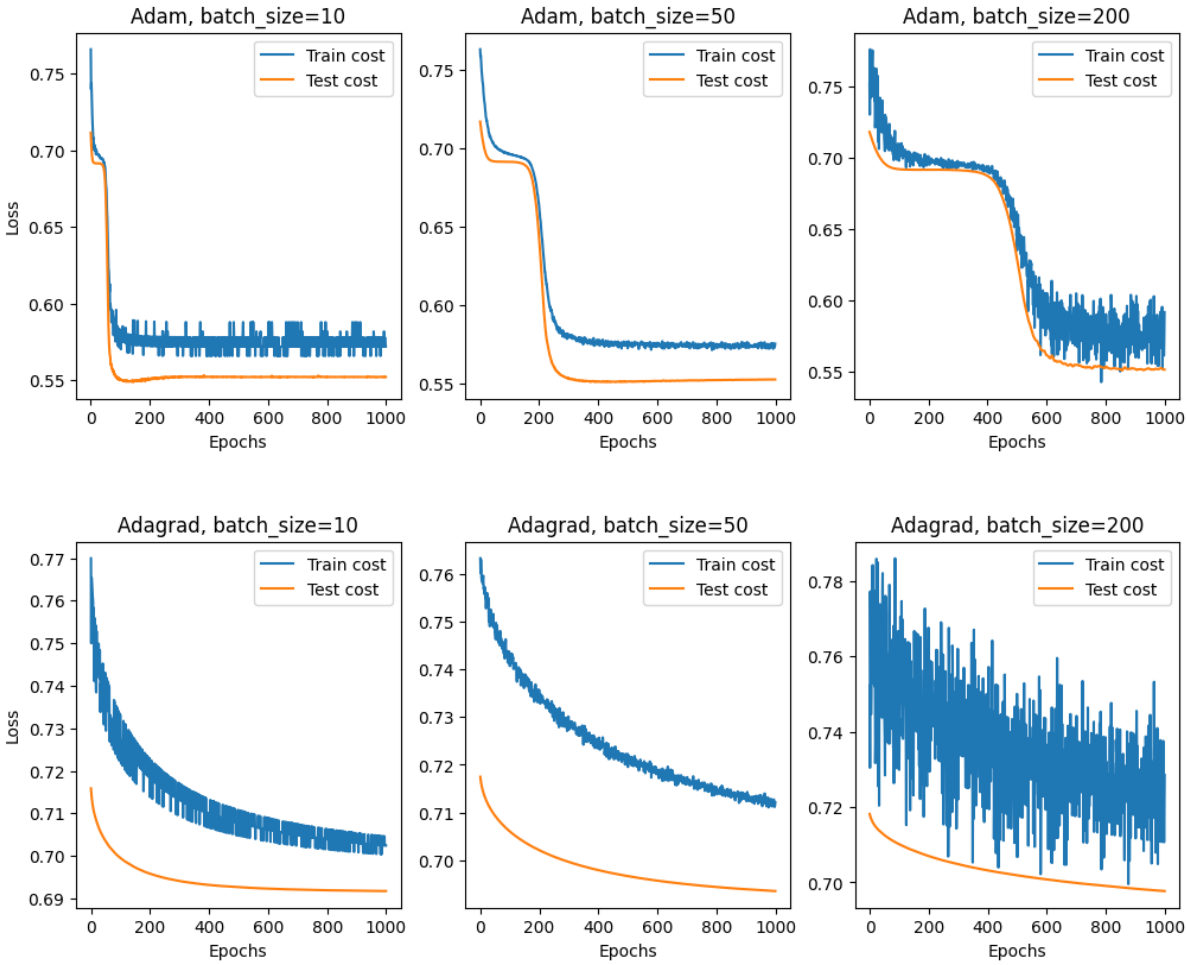


Wstępne analizy:

+ Optimizer Adam wydaje się najskuteczniejszy w tym eksperymencie. Szybko zmniejsza stratę i utrzymuje niewielką lukę pomiędzy stratami treningowymi i testowymi, co sugeruje dobrą generalizację.
+ SGD wykazuje uczenie się, ale może być podatne na nadmierne dopasowanie lub może skorzystać na harmonogramie tempa uczenia się lub większej regularyzacji.
+ AdaGrad wydaje się najmniej skuteczny w tym eksperymencie. Jego wydajność można potencjalnie poprawić, dostosowując szybkość uczenia się lub inne hiperparametry.

- Eksperyment różnych rozmiarów batcha dla każdego optimizera:





Wnioski:

+ Szybkość zbieżności: Adam zbiega najszybciej, Adagrad nieco wolniej, a SGD najwolniej.

+ Stabilność: Adam jest najbardziej stabilny przy wszystkich rozmiarach batcha, podczas gdy SGD i Adagrad wykazują większą zmienność przy małym rozmiarze batcha.

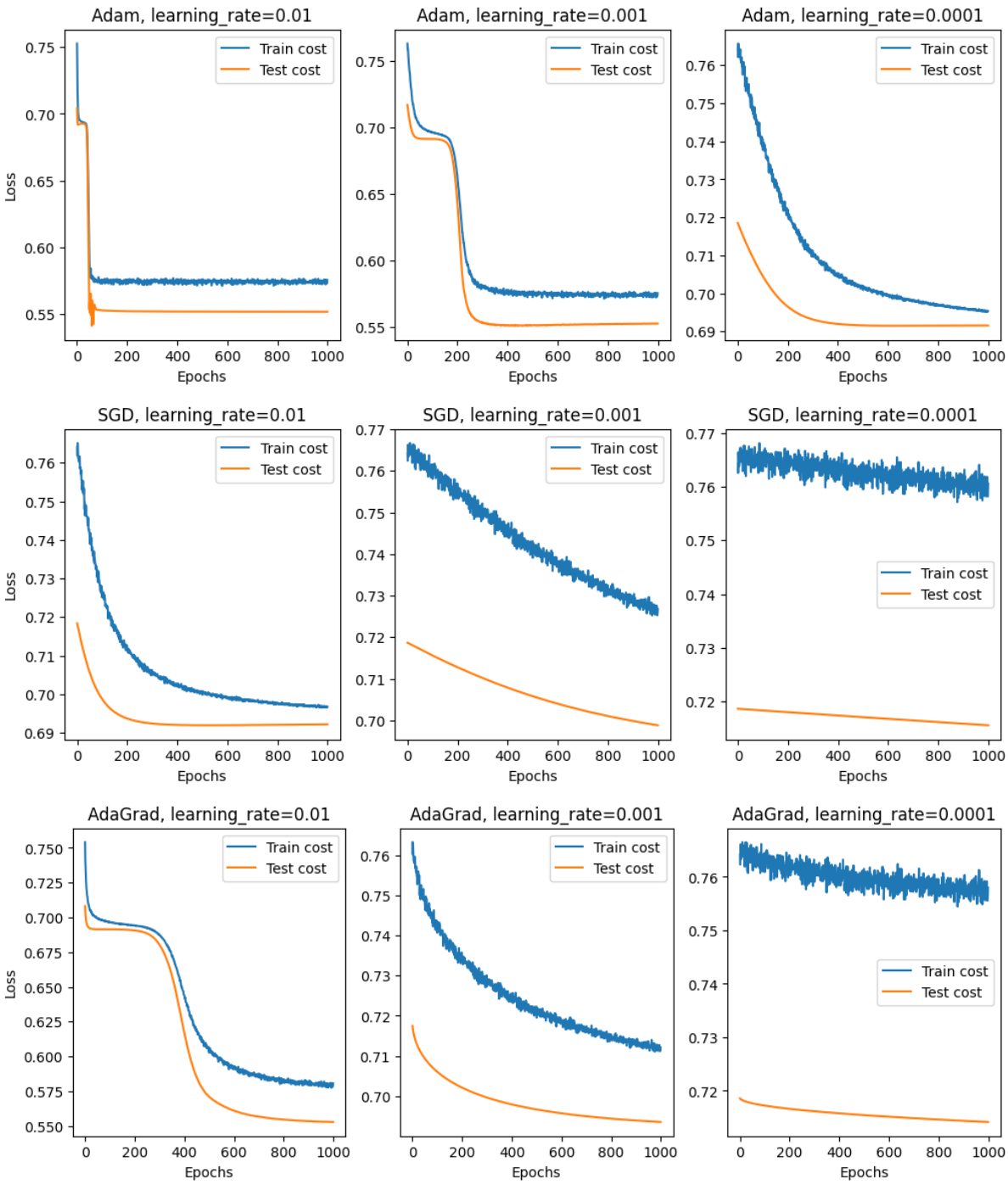
+ Przetrenowanie: Adam i Adagrad mogą wykazywać przetrenowanie przy małym rozmiarze batcha, podczas gdy przy większych rozmiarach batcha oba wykazują lepsze uogólnienie.

+ Optymalny rozmiar batcha: Dla SGD jest to 50, dla Adama większe batche są lepsze, a Adagrad poprawia swoje działanie przy większych batcha.

+ Końcowe wartości strat: Adam osiąga najniższe straty, sugerując, że jest najskuteczniejszym optimizerem dla tego zadania.

+ Szum w treningu: Małe batche powodują większy szum dla SGD i Adagrad, podczas gdy Adam jest odporny na rozmiar batcha.

- Eksperyment różnych współczynników learning_rate batcha dla każdego optimizera:



Wnioski:

+ Adam - szybko zbiega do niskiej straty przy różnych współczynnikach uczenia, z najwyższym współczynnikiem uczenia (0.01) osiągając szybką konwergencję.

+ SGD - przy wyższym współczynniku uczenia (0.01) może dochodzić do przeuczenia, a przy niższych współczynnikach uczenia konwerguje wolniej; przy najniższym współczynniku uczenia (0.0001) postęp jest minimalny.

+ AdaGrad - dobrze radzi sobie z wyższym współczynnikiem uczenia (0.01), ale podobnie jak SGD, przy niższym współczynniku uczenia konwergencja jest wolniejsza.

+ Adam wydaje się być najbardziej odporny na wybór współczynnika uczenia i osiąga konsekwentnie niższe straty w porównaniu do SGD i AdaGrad.