SIECI NEURONOWE – ćwiczenie 2

W drugim ćwiczeniu zajmiemy się uczeniem maszynowym. Wykorzystamy regresję logistyczną – metodę statystyczną która 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. Możemy zdefiniować wyjście jako:

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

Gdzie x to wektor wejściowy, W i b to parametry modelu natomiast funkcja sigma to:

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

Jest to wygodna funkcja dająca wyniki przedziale [0,1] dla każdego rzeczywistego n, przy okazji rosnąca i różniczkowalna.

Aby wyuczyć model w jakikolwiek sposób, potrzebujemy zdefiniowanego zadania optymalizacji, a do tego potrzebujemy funkcji kosztu którą będziemy minimalizować.

Korzystamy z entropii krzyżowej, dla przykładu (x,y) w klasyfikacji binarnej:

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

(y będzie zawsze zerem lub jedynką, czyli tylko jeden ze składników będzie niezerowy. Wartość sumujemy po wszystkich przykładach w zbiorze danych, suma będzie pominięta w wzorach.)

Zaletą takiego kosztu jest fakt, że jej pochodna po wagach modelu jest bardzo prosta:

$\partial L / \partial wi = -(y - p(x))xi$

Optymalizacja modelu będzie polegała na wykonywaniu niewielkich kroków w kierunku wyznaczonym przez gradient – pochodne cząstkowe po wszystkich wagach – w pętli uczenia, aż model osiągnie zbieżność. Innymi słowy, aktualizujemy wagi według:

$wi' = wi - \alpha * \partial L / \partial wi$

Gdzie alfa to pewien współczynnik uczenia, hiperparametr który musimy ustawić z góry.

Zadaniem na zajęcia 2 jest implementacja działającego na zbiorze heart disease modelu klasyfikacji, przy czym:

- Zbieżność modelu można zdefiniować przez wystarczająco małą zmianę funkcji kosztu w danej iteracji i pewną maksymalną liczbę iteracji
- Model może uczyć się wyliczając sumaryczną pochodną z funkcji kosztu po całym zbiorze, po jednym przykładzie lub po paczce przykładów w iteracji. Dla sieci neuronowej w następnym zadaniu będzie już wymagany tryb paczkowania, więc warto przećwiczyć operacje na całych macierzach a nie tylko pojedynczych wektorach danych
- Uczenie się modelu powinno być weryfikowalne metryką (np. accuracy, fscore, precision – można korzystać z bibliotek)

- Weryfikacja powinna uwzględnić podział na dane uczące i testowe

Ćwiczenie oceniane jest w skali 0-10 pkt.

Ładowanie Danych

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

column_names = ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs',
'restecg', 'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal',
'num']
data = pd.read_csv('../heart-disease/processed.cleveland.data',
header=None, names=column_names, encoding='latin1')
```

Normalizacja cech

```
data = data.replace('?', np.nan)
data = data.apply(pd.to_numeric, errors='coerce')

print("Liczba braków przed dropem:")
print(data.isna().sum())
data = data.dropna()
print("\nLiczba braków po dropie:")
print(data.isna().sum())

data['num'] = (data['num'] > 0).astype(int)
```

```
Liczba braków przed dropem:
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

Liczba braków po dropie:
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
```

Podział na zbiór testowy i uczący

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

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

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

scaler = StandardScaler()
X_train = pd.DataFrame(scaler.fit_transform(X_train),
columns=X.columns)
X_test = pd.DataFrame(scaler.transform(X_test), columns=X.columns)
```

Inicjacja wag i funkcji

```
epsilon = 1e-15

# Inicjacja listy wag, W, b
n_features = X_train.shape[1]
W = np.zeros(n_features)
b = 0.0

# Definicja funkcji sigmoidalnej
def sigmoid(z):
    z = np.clip(z, -500, 500)
    return 1 / (1 + np.exp(-z))

# Definicja funkcji kosztu
def compute_loss(X, y, W, b):
    m = len(y)
    p = sigmoid(np.dot(X, W) + b)
    cost = - (y * np.log(p + epsilon) + (1 - y) * np.log(1 - p +
```

```
epsilon))
    return np.mean(cost)

# Definicja pochodnej i obliczenia na macierzy
def compute_derivative(X, Y, Y_hat):
    m = X.shape[0]
    dW = np.dot(X.T, (Y_hat - Y)) / m
    db = np.sum(Y_hat - Y) / m
    return dW, db
```

Inicjacja funkcji bezpośrednio dot. regresji logistycznej

```
# Aktualizacja wag
def update_parameters(W, b, dW, db, learning_rate):
    W -= learning_rate * dW
    b -= learning_rate * db
    return W, b

# Funkcja predykcji
def predict(X, W, b):
    z = np.dot(X, W) + b
    return sigmoid(z)

# Funkcja obliczania gradientów
def compute_gradients(X, Y, Y_hat):
    m = X.shape[0]
    dW = np.dot(X.T, (Y_hat - Y)) / m
    db = np.sum(Y_hat - Y) / m
    return dW, db

# Funkcja trenowania modelu
def train(X, Y, epochs=1000, learning_rate=0.01, tol=1e-6):
    n_features = X.shape[1]
    W = np.zeros(n_features)
    b = 0.0
    prev_loss = float('inf')

    for i in range(epochs):
        Y_hat = predict(X, W, b)
        loss = compute_loss(X, Y, W, b)
        dW, db = compute_gradients(X, Y, Y_hat)
        W, b = update_parameters(W, b, dW, db, learning_rate)

        if i % 50 == 0:
            print(f'Epoch {i}, Loss: {loss:.4f}')

        if abs(prev_loss - loss) < tol:
            print(f'Zbieżność osiągnięta po {i} epokach (delta L <
{tol})')
```

```
            break
        prev_loss = loss

    return W, b
```

Ocena modelu

```
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score

W, b = train(X_train, y_train, epochs=2000, learning_rate=0.005,
tol=1e-5)

y_pred = predict(X_test, W, b) > 0.5
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-score:", f1_score(y_test, y_pred))
```

```
Epoch 0, Loss: 0.6931
Epoch 50, Loss: 0.6206
Epoch 100, Loss: 0.5694
Epoch 150, Loss: 0.5323
Epoch 200, Loss: 0.5048
Epoch 250, Loss: 0.4839
Epoch 300, Loss: 0.4675
Epoch 350, Loss: 0.4545
Epoch 400, Loss: 0.4439
Epoch 450, Loss: 0.4352
Epoch 500, Loss: 0.4279
Epoch 550, Loss: 0.4217
Epoch 600, Loss: 0.4164
Epoch 650, Loss: 0.4119
Epoch 700, Loss: 0.4079
Epoch 750, Loss: 0.4044
Epoch 800, Loss: 0.4013
Epoch 850, Loss: 0.3985
Epoch 900, Loss: 0.3961
Epoch 950, Loss: 0.3939
Epoch 1000, Loss: 0.3918
Epoch 1050, Loss: 0.3900
Epoch 1100, Loss: 0.3884
Epoch 1150, Loss: 0.3868
Epoch 1200, Loss: 0.3854
Epoch 1250, Loss: 0.3841
Epoch 1300, Loss: 0.3829
Epoch 1350, Loss: 0.3818
Epoch 1400, Loss: 0.3808
Epoch 1450, Loss: 0.3798
Epoch 1500, Loss: 0.3789
```

```
Epoch 1550, Loss: 0.3781
Epoch 1600, Loss: 0.3773
Epoch 1650, Loss: 0.3766
Epoch 1700, Loss: 0.3759
Epoch 1750, Loss: 0.3752
Epoch 1800, Loss: 0.3746
Epoch 1850, Loss: 0.3740
Epoch 1900, Loss: 0.3734
Epoch 1950, Loss: 0.3729
Zbieżność osiągnięta po 1992 epokach (delta L < 1e-05)
Accuracy: 0.9166666666666666
Precision: 0.88
Recall: 0.9166666666666666
F1-score: 0.8979591836734694
```