

Metoda gradientu prostego

Wsteczna propagacja błędu

Sprawozdanie z ćwiczeń
Matematyka Konkretna

Data wykonania:
28.06.2025

Autor:
Bartosz Bieniek 058085

1. Cel ćwiczenia

Metoda gradientu prostego. Wsteczna propagacja błędu

Zadanie 1 dotyczy odnalezienia wartości minimalnej funkcji dwóch zmiennych f oraz zmiennych x i y metodą gradientu wraz z wizualizacją w 3D odpowiednio do określonego zadania. Można skorzystać z dowolnych bibliotek Python.

Zadanie 2 dotyczy obliczenia gradientów sieci neuronowej z pomocą biblioteki numpy zadanej z pomocy architektury.

Zadania umieścić na [Github](#).

2. Przebieg ćwiczenia

Zadanie 1: Gradient prosty: znalezienie minimum funkcji

1. Import bibliotek

Zaimportowano biblioteki numpy, matplotlib oraz mpl_toolkits.mplot3d do obliczeń numerycznych i wizualizacji trójwymiarowej. Dzięki nim możliwe było przetworzenie danych i przedstawienie powierzchni funkcji celu wraz z trajektorią jej minimalizacji.

```
[1] # Zadanie 1: Znalezienie minimum funkcji dwóch zmiennych (x, y)
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
✓ 0.2s
```

Rys. 1. Import bibliotek

2. Definicja funkcji celu i jej gradientu

Zdefiniowano funkcję celu $f(x,y)=x^2+y^2$, która posiada minimum globalne w punkcie $(0, 0)$. Obliczono jej gradient analityczny jako wektor pochodnych cząstkowych: $\nabla f=[2x,2y]$, co pozwoliło na iteracyjną aktualizację współrzędnych.

```
[2] # Funkcja celu: f(x, y) = x^2 + y^2
def f(x, y):
    return x**2 + y**2

# Gradient funkcji
def grad_f(x, y):
    return np.array([2*x, 2*y])
✓ 0.0s
```

Rys. 2. Definicja funkcji celu i jej gradientu

3. Algorytm gradientu prostego

Zaimplementowano prostą metodę gradientu, rozpoczynając od punktu startowego (4, -3). W każdej iteracji obliczano gradient i wykonywano krok w przeciwnym kierunku, zmniejszając wartość funkcji. Śledzono historię punktów, aby później przedstawić trajektorię ruchu.

```
# Parametry startowe
x, y = 4.0, -3.0 # punkt początkowy
alpha = 0.1      # krok uczenia
epochs = 50      # liczba iteracji

# Historia punktów
trajectory = [(x, y)]

for i in range(epochs):
    grad = grad_f(x, y)
    x -= alpha * grad[0]
    y -= alpha * grad[1]
    trajectory.append((x, y))

trajectory = np.array(trajectory)
```

[3] ✓ 0.0s Python

Rys. 3. Algorytm gradientu prostego

4. Wizualizacja w 3D

Wygenerowano wykres 3D przedstawiający powierzchnię funkcji oraz trajektorię minimalizacji. Ścieżka spadku gradientowego przypominała ruch kuli toczącej się w dół w stronę minimum, zgodnie z kierunkiem największego spadku. Wizualizacja potwierdziła poprawność działania algorytmu.

```
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')

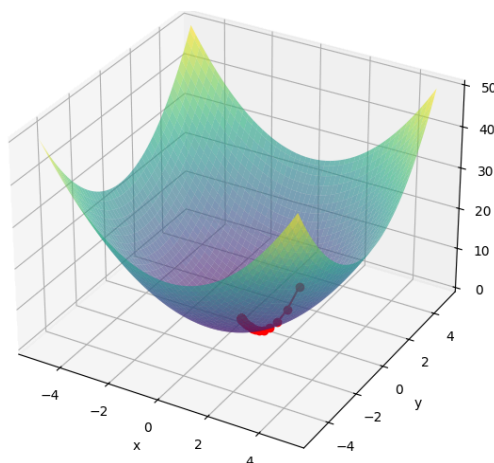
# Siatka do wykresu
X = np.linspace(-5, 5, 100)
Y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)

# Rysowanie powierzchni funkcji
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.6)

# Rysowanie ścieżki spadku gradientowego
Z_traj = f(trajectory[:, 0], trajectory[:, 1])
ax.plot(trajectory[:, 0], trajectory[:, 1], Z_traj, color='red', marker='o')

ax.set_title("Spadek gradientowy funkcji  $f(x, y) = x^2 + y^2$ ")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")
plt.show()
```

[4] ✓ 0.3s



Rys. 4. Wizualizacja w 3D

Zadanie 2: Wsteczna propagacja błędu

1. Import bibliotek

Zaimportowano bibliotekę numpy, która umożliwiła wektorowe przetwarzanie danych oraz implementację operacji macierzowych. Środowisko zostało przygotowane do ręcznej symulacji sieci neuronowej oraz jej uczenia przez backpropagation.

```
# Zadanie 2: Wsteczna propagacja błędu w sieci neuronowej
import numpy as np
```

[5] ✓ 0.0s

Rys. 5. Import bibliotek

2. Architektura sieci neuronowej i funkcje aktywacji

Zdefiniowano prostą sieć neuronową z dwiema warstwami: warstwą ukrytą (2 neurony) i wyjściem binarnym (1 neuron). Jako funkcję aktywacji użyto funkcji sigmoidalnej, której pochodna została również zdefiniowana do celów obliczeń gradientów.

```
# Dane wejściowe i wyjściowe (XOR-like)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Inicjalizacja wag
np.random.seed(0)
W1 = np.random.randn(2, 2) # wejście -> warstwa ukryta
b1 = np.zeros((1, 2))
W2 = np.random.randn(2, 1) # warstwa ukryta -> wyjście
b2 = np.zeros((1, 1))

# Funkcja aktywacji: sigmoid + pochodna
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

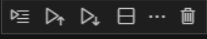
def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)
```

[6] ✓ 0.0s Python

Rys. 6. Architektura sieci neuronowej i funkcje aktywacji

3. Forward pass i backward pass

Przeprowadzono pełną propagację danych przez sieć oraz obliczono błąd między wartościami wyjściowymi a oczekiwanymi. Następnie wykonano propagację wsteczną błędu, obliczając gradienty względem wszystkich wag i biasów. Uzyskane wartości potwierdziły zdolność sieci do dalszej optymalizacji przez aktualizację parametrów.

```
> 
# Propagacja w przód
z1 = X @ W1 + b1
a1 = sigmoid(z1)
z2 = a1 @ W2 + b2
a2 = sigmoid(z2)

# Błąd
loss = y - a2

# Wsteczna propagacja
dz2 = loss * sigmoid_derivative(z2)
dW2 = a1.T @ dz2
db2 = np.sum(dz2, axis=0, keepdims=True)

dz1 = dz2 @ W2.T * sigmoid_derivative(z1)
dW1 = X.T @ dz1
db1 = np.sum(dz1, axis=0, keepdims=True)

print("Gradients:")
print("dW1 =", dW1)
print("db1 =", db1)
print("dW2 =", dW2)
print("db2 =", db2)

[7] ✓ 0.0s Python
```

```
... Gradients:
dW1 = [[-0.00341382 -0.00335098]
 [ 0.01802458  0.00122409]]
db1 = [[-0.03750792  0.0243876 ]]
dW2 = [[-0.10012408]
 [-0.0965125 ]]
db2 = [[-0.14912546]]
```

Rys. 7. Forward pass i backward pass

3. Wnioski

W drugim zadaniu przeprowadzono ręczne obliczenie gradientów sieci neuronowej metodą wstecznej propagacji błędu. Na podstawie propagacji w przód i reguły łańcuchowej uzyskano dokładne wartości pochodnych względem wag i biasów, co umożliwia dalszą aktualizację parametrów sieci w kierunku minimalizacji funkcji kosztu.

Oba zadania potwierdziły kluczową rolę gradientu w procesie optymalizacji – zarówno w analizie funkcji matematycznych, jak i w uczeniu modeli neuronowych.