

Podstawy sieci neuronowych

Projekt – wariant #1

Adam Roś (272569)

Mateusz Potoczny (272584)

Poniedziałek 18:55-20:25

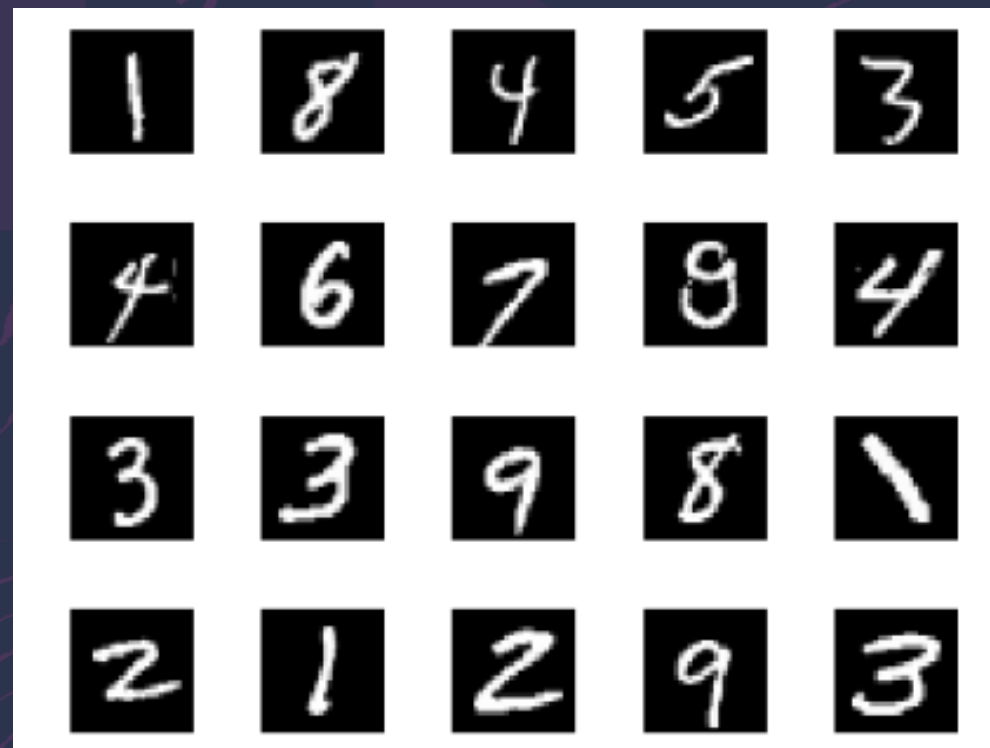


Wprowadzenie

- Cel projektu:
 - Implementacja sieci MLP w Pythonie
 - Rozwiązanie dwóch zadań:
 - Klasyfikacja obrazów (MNIST)
 - Aproksymacja funkcji dwuwymiarowej (Ackley'a)
- Założenia:
 - Implementacja na bazie macierzy i wektorów bez wykorzystania gotowych bibliotek

Klasyfikacja obrazów

- Opis:
 - Klasyfikacja obrazów binarnych (czarno-białych)
 - Dane z bazy MNIST:
 - Treningowe: 60,000 obrazów
 - Testowe: 10,000 obrazów
- Podział danych:
 - Nauka: dane treningowe
 - Test: dane niewidoczne podczas nauki



Przygotowanie danych

- Kroki przygotowania danych:
 1. Wczytanie obrazów
 2. Konwersja obrazów na macierze NumPy
 3. Normalizacja pikseli (zakres [0, 1])
 4. Kodowanie etykiet w formacie one-hot

```
def load_images(file_path):  
    with open(file_path, 'rb') as f:  
        magic_number = int.from_bytes(f.read(4), byteorder: 'big')  
        num_images = int.from_bytes(f.read(4), byteorder: 'big')  
        num_rows = int.from_bytes(f.read(4), byteorder: 'big')  
        num_cols = int.from_bytes(f.read(4), byteorder: 'big')  
  
        buffer = f.read(num_images * num_rows * num_cols)  
        data = np.frombuffer(buffer, dtype=np.uint8)  
        data = data.reshape(num_images, num_rows, num_cols)  
        return data
```

Budowa sieci neuronowej

- Elementy sieci:
 - Wagi i biasy inicjalizowane losowo
 - Struktura:
 - Warstwa wejściowa: 784 neurony (28×28)
 - Ukryta: 128 neuronów
 - Wyjściowa: 10 neuronów (cyfry 0-9)
- Inicjalizacja wag:
 - Losowanie z rozkładu normalnego
 - Małe wartości wag redukują dominację neuronów

```
def __init__(self, input_size, hidden_size, output_size, learning_rate):  
    self.weights_input_hidden = np.random.randn(input_size, hidden_size) * 0.01  
    self.bias_hidden = np.zeros((1, hidden_size))  
  
    self.weights_hidden_output = np.random.randn(hidden_size, output_size) * 0.01  
    self.bias_output = np.zeros((1, output_size))  
  
    self.learning_rate = learning_rate
```

Funkcje aktywacji

- Funkcje używane w sieci:
 - ReLU (Rectified Linear Unit):
 - Dla $x > 0$: przepuszcza sygnał
 - Dla $x \leq 0$: blokuje sygnał
 - Sigmoid:
 - Przekształca wynik na zakres $[0, 1]$
 - Używana w warstwie wyjściowej

```
@staticmethod
def relu(x):
    return np.maximum(*args: 0, x)
```

1 usage

```
@staticmethod
def relu_derivative(x):
    return (x > 0).astype(float)
```

```
@staticmethod
def sigmoid(x):
    return 1 / (1 + np.exp(-np.clip(x, -500, a_max: 500)))
```

1 usage

```
@staticmethod
def sigmoid_derivative(x):
    return x * (1 - x)
```

Propagacja w przód

- Działanie:
 1. Dane wejściowe → Warstwa ukryta (ReLU)
 2. Warstwa ukryta → Warstwa wyjściowa (Sigmoid)
 3. Wynik: prawdopodobieństwa klasyfikacji
- Zalety:
 - Warstwa ReLU: unika problemu znikających gradientów
 - Warstwa Sigmoid: interpretuje wyniki jako prawdopodobieństwa.

```
def forward(self, x):  
    self.input = x  
    self.hidden_input = np.dot(self.input, self.weights_input_hidden) + self.bias_hidden  
    self.hidden_output = self.relu(self.hidden_input)  
    self.final_input = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output  
    self.final_output = self.sigmoid(self.final_input)  
    return self.final_output
```

Propagacja wsteczna

- Cel:
 - Minimalizacja błędu poprzez aktualizację wag
- Kroki:
 1. Obliczanie błędów wyjściowych (sigmoid)
 2. Propagacja błędów wstecz do warstwy ukrytej (ReLU)
 3. Aktualizacja wag i biasów algorytmem spadku gradientu

```
def backward(self, y_true):  
    output_error = (self.final_output - y_true) * self.sigmoid_derivative(self.final_output)  
  
    hidden_error = np.dot(output_error, self.weights_hidden_output.T) * self.relu_derivative(self.hidden_input)  
    self.weights_hidden_output -= self.learning_rate * np.dot(self.hidden_output.T, output_error)  
    self.bias_output -= self.learning_rate * np.sum(output_error, axis=0, keepdims=True)  
  
    self.weights_input_hidden -= self.learning_rate * np.dot(self.input.T, hidden_error)  
    self.bias_hidden -= self.learning_rate * np.sum(hidden_error, axis=0, keepdims=True)
```


Trenowanie sieci

- Metoda mini-batch:
 - Dane dzielone na małe porcje
 - Każdy batch aktualizuje wagi niezależnie
- Zalety:
 - Szybsze trenowanie
 - Stabilniejsze wyniki
- Monitorowanie:
 - Strata (MSE)
 - Dokładność klasyfikacji.

```
def train(self, x_train, y_train, epochs, batch_size=64, patience=20):
    best_loss = float('inf')
    no_improve_count = 0
    n_samples = x_train.shape[0]
    for epoch in range(epochs):
        indices = np.arange(n_samples)
        np.random.shuffle(indices)
        x_train = x_train[indices]
        y_train = y_train[indices]
        start_time = time.time()
        for start_idx in range(0, n_samples, batch_size):...
        train_output = self.forward(x_train)
        loss = self.mse(y_train, train_output)
        predictions_train = self.predict(x_train)
        y_train_labels = np.argmax(y_train, axis=1)
        train_accuracy = np.mean(predictions_train == y_train_labels) * 100
        duration = time.time() - start_time
        print(f"Epoka {epoch+1}/{epochs}, Strata: {loss:.6f}, Dokładność tren.: {train_a
        if loss < best_loss:
            best_loss = loss
            no_improve_count = 0
        else:
            no_improve_count += 1
            if no_improve_count >= patience:
                print(f"Zatrzymanie w epokach {epoch+1} z powodu braku poprawy.")
                break
```

Wyniki pracy programu

- Dokładność:
 - Trening: 100%
 - Test: 98%
- Losowanie cyfry z bazy testowej:
 - Losowa cyfra z MNIST
 - Poprawna klasyfikacja


```
Epoka 1/20, Strata: 0.013722, Dokładność tren.: 92.16%, Czas: 2.90s
Epoka 2/20, Strata: 0.010128, Dokładność tren.: 94.20%, Czas: 2.77s
Epoka 3/20, Strata: 0.007927, Dokładność tren.: 95.64%, Czas: 2.62s
Epoka 4/20, Strata: 0.007026, Dokładność tren.: 96.22%, Czas: 2.66s
Epoka 5/20, Strata: 0.005954, Dokładność tren.: 96.86%, Czas: 2.64s
Epoka 6/20, Strata: 0.005186, Dokładność tren.: 97.34%, Czas: 2.83s
Epoka 7/20, Strata: 0.004708, Dokładność tren.: 97.61%, Czas: 2.82s
Epoka 8/20, Strata: 0.004214, Dokładność tren.: 97.90%, Czas: 2.60s
Epoka 9/20, Strata: 0.003844, Dokładność tren.: 98.10%, Czas: 2.56s
Epoka 10/20, Strata: 0.003540, Dokładność tren.: 98.28%, Czas: 2.48s
Epoka 11/20, Strata: 0.003302, Dokładność tren.: 98.40%, Czas: 2.42s
Epoka 12/20, Strata: 0.003305, Dokładność tren.: 98.40%, Czas: 2.29s
Epoka 13/20, Strata: 0.002889, Dokładność tren.: 98.58%, Czas: 2.28s
Epoka 14/20, Strata: 0.002656, Dokładność tren.: 98.71%, Czas: 2.37s
Epoka 15/20, Strata: 0.002618, Dokładność tren.: 98.75%, Czas: 2.35s
Epoka 16/20, Strata: 0.002396, Dokładność tren.: 98.86%, Czas: 2.33s
Epoka 17/20, Strata: 0.002321, Dokładność tren.: 98.89%, Czas: 2.33s
Epoka 18/20, Strata: 0.002260, Dokładność tren.: 98.91%, Czas: 2.37s
Epoka 19/20, Strata: 0.002044, Dokładność tren.: 99.01%, Czas: 2.58s
Epoka 20/20, Strata: 0.001963, Dokładność tren.: 99.04%, Czas: 2.47s
Dokładność na danych testowych: 98.10%
Przewidziana cyfra: 4
```



Ulepszona wersja programu

- Zmiany:
 - Inicjalizacja wag metodą Xavier'a
 - Softmax jako funkcja aktywacji
 - Entropia krzyżowa jako funkcja kosztu
- Efekt:
 - Szybsze trenowanie
 - Dokładność po zmniejszeniu liczby epok: 98.3%

```
def __init__(self, input_size, hidden_size, output_size, learning_rate):  
    limit_in = np.sqrt(6.0 / (input_size + hidden_size))  
    self.weights_input_hidden = np.random.uniform(-limit_in, limit_in, size: (input_size, hidden_size))  
    self.bias_hidden = np.zeros((1, hidden_size))  
    limit_out = np.sqrt(6.0 / (hidden_size + output_size))  
    self.weights_hidden_output = np.random.uniform(-limit_out, limit_out, size: (hidden_size, output_size))  
    self.bias_output = np.zeros((1, output_size))  
    self.learning_rate = learning_rate
```



Wynik ulepszanego programu

- Porównanie:
 - Poprawiona efektywność czasowa
 - Przeuczenie sieci przy 20 epokach
- Wniosek:
 - Redukcja liczby epok poprawia wyniki

```
Epoka 1/20, Strata: 0.109122, Dokładność tren.: 96.75%, Czas: 1.99s
Epoka 2/20, Strata: 0.082179, Dokładność tren.: 97.30%, Czas: 1.96s
Epoka 3/20, Strata: 0.048776, Dokładność tren.: 98.47%, Czas: 1.97s
Epoka 4/20, Strata: 0.042453, Dokładność tren.: 98.65%, Czas: 1.98s
Epoka 5/20, Strata: 0.031599, Dokładność tren.: 99.00%, Czas: 2.01s
Epoka 6/20, Strata: 0.030541, Dokładność tren.: 99.03%, Czas: 1.99s
Epoka 7/20, Strata: 0.023269, Dokładność tren.: 99.27%, Czas: 2.00s
Epoka 8/20, Strata: 0.017350, Dokładność tren.: 99.48%, Czas: 1.97s
Epoka 9/20, Strata: 0.012435, Dokładność tren.: 99.66%, Czas: 2.00s
Epoka 10/20, Strata: 0.020393, Dokładność tren.: 99.39%, Czas: 2.01s
Epoka 11/20, Strata: 0.009729, Dokładność tren.: 99.70%, Czas: 1.95s
Epoka 12/20, Strata: 0.006126, Dokładność tren.: 99.86%, Czas: 1.97s
Epoka 13/20, Strata: 0.004513, Dokładność tren.: 99.92%, Czas: 1.92s
Epoka 14/20, Strata: 0.003098, Dokładność tren.: 99.96%, Czas: 1.97s
Epoka 15/20, Strata: 0.001749, Dokładność tren.: 100.00%, Czas: 2.01s
Epoka 16/20, Strata: 0.001396, Dokładność tren.: 99.99%, Czas: 1.96s
Epoka 17/20, Strata: 0.001043, Dokładność tren.: 100.00%, Czas: 1.96s
Epoka 18/20, Strata: 0.000966, Dokładność tren.: 100.00%, Czas: 1.97s
Epoka 19/20, Strata: 0.000813, Dokładność tren.: 100.00%, Czas: 2.02s
Epoka 20/20, Strata: 0.000730, Dokładność tren.: 100.00%, Czas: 1.96s
Dokładność na danych testowych: 97.80%
Przewidziana cyfra: 5
```

Aproksymacja funkcji

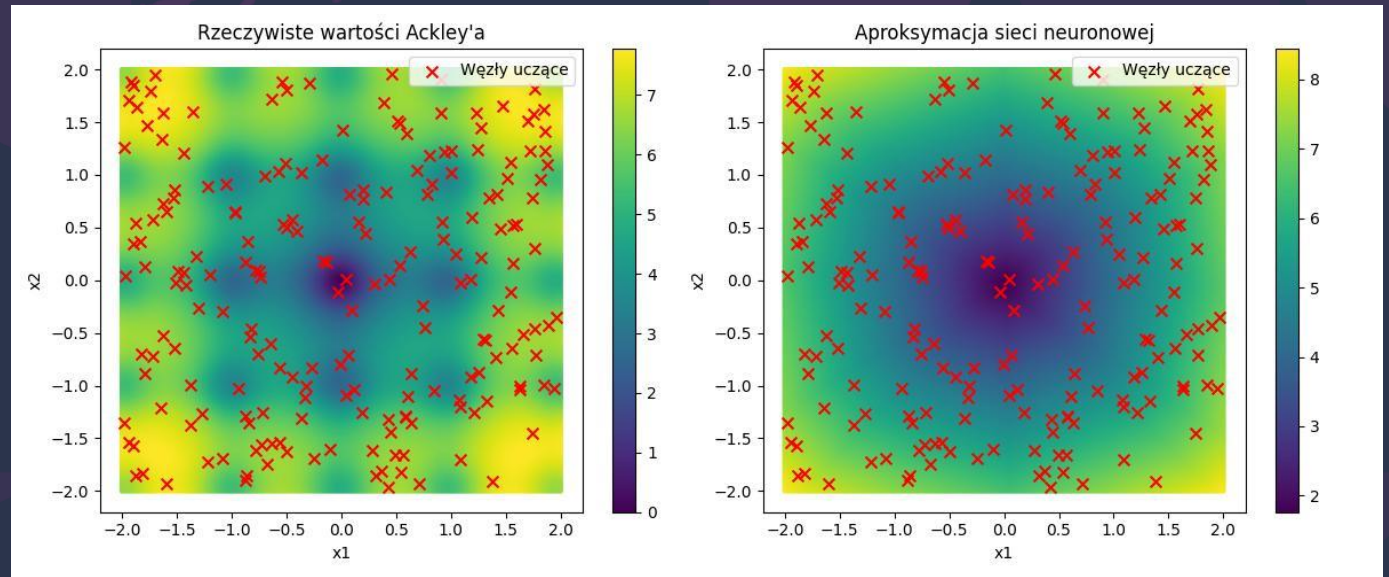
- Cel:
 - Aproksymacja funkcji Ackley'a
- Struktura sieci:
 - Warstwa ukryta: 10 neuronów
 - Wyjściowa: 1 neuron (wynik funkcji)

```
def generate_training_data(n_samples=200, low=-2.0, high=2.0, seed=42):  
    np.random.seed(seed)  
    X = np.random.uniform(low, high, size=(n_samples, 2))  
    y = []  
    for i in range(n_samples):  
        y_val = ackley_function(X[i, 0], X[i, 1])  
        y.append(y_val)  
    y = np.array(y).reshape(-1, 1)  
    return X, y
```

Wyniki aproksymacji

- MSE na danych testowych: 0.3489
- Wizualizacja:
 - Porównanie wyników rzeczywistych i aproksymowanych.

```
[Epoka 0] MSE (na całym zbiorze): 28.604015  
[Epoka 200] MSE (na całym zbiorze): 0.456074  
[Epoka 400] MSE (na całym zbiorze): 0.375060  
[Epoka 600] MSE (na całym zbiorze): 0.351170  
[Epoka 800] MSE (na całym zbiorze): 0.341386  
[Epoka 1000] MSE (na całym zbiorze): 0.338875  
[Epoka 1200] MSE (na całym zbiorze): 0.337834  
[Epoka 1400] MSE (na całym zbiorze): 0.337379  
[Epoka 1600] MSE (na całym zbiorze): 0.337017  
[Epoka 1800] MSE (na całym zbiorze): 0.320487  
Średni błąd kwadratowy (MSE) na siatce testowej: 0.348913
```



Wnioski

- Skuteczność modelu:
 - Klasyfikacja obrazów z MNIST: 98% dokładności na danych testowych
 - Aproksymacja funkcji Ackley'a: MSE na danych testowych wyniosło 0.3489
- Wnioski praktyczne:
 - Użycie jednej warstwy ukrytej okazało się wystarczające
 - Metoda mini-batch przyspieszyła trenowanie i zapewniła stabilność
 - Inicjalizacja wag metodą Xavier'a poprawiła efektywność