Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Podstawy Sieci Neuronowych

Projekt – wariant # 1

Adam Roś 272569

Mateusz Potoczny 272584

Poniedziałek 18:55 – 20:25

Spis treści

1. Wprowadzenie

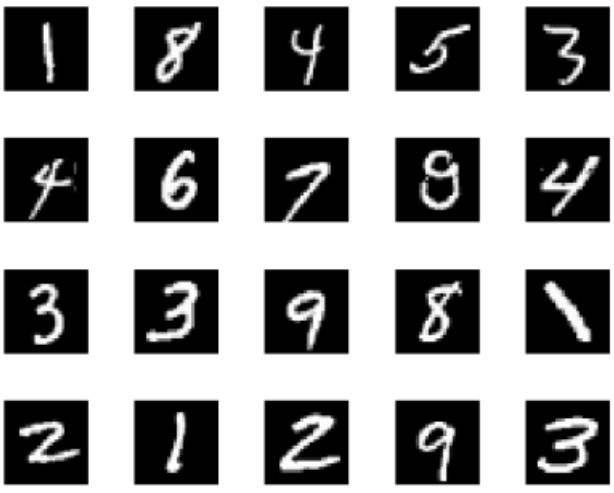
Celem wariantu # 1 było zaprojektowanie i implementacja sztucznej sieci neuronowej MLP (Multi-Layered Perceptron, wielowarstwowa) do realizacji dwóch wyznaczonych zadań:

- Zadanie 1 – Klasyfikacja obrazów
- Zadanie 2 – Aproksymacja funkcji dwuwymiarowej

Implementacja powinna bazować na macierzach i wektorach bez wykorzystania gotowych bibliotek do tworzenia sieci neuronowych, takich jak TensorFlow. Wybranym przez nas językiem programowania jest język Python.

2. Zadanie 1 – Klasyfikacja obrazów

Celem zadania pierwszego było zaprojektowanie i implementacja sztucznej sieci neuronowej typu MLP do rozwiązania prostego problemu klasyfikacji obrazów dla dwóch zestawów danych składających się z obrazów binarnych (czarno – białych). Obrazy zostały pozyskane z bazy cyfr MNIST stworzonej specjalnie do trenowania sieci neuronowych. Składa się ona ze zbioru obrazów o wymiarach 28×28 przedstawiających skany odręcznie pisanych cyfr z przedziału od 0 do 9. Baza składa się z 4 plików z których dwa zawierają zbiory obrazów a dwa legendy do nich. Jest to konieczne by program mógł zidentyfikować poprawność dopasowania i kontynuować uczenie sieci. Dodatkowo baza dzieli się na bazę obrazów przeznaczonych do nauki i bazę przeznaczoną do testowania już wyuczonej sieci. Pierwsza baza zawiera 60 000 elementów i to na jej podstawie program w kolejnych epokach uczy się rozpoznawać cyfry. Drugi zbiór jest mniejszy gdyż zawiera 10 000 dodatkowych elementów do których program nie miał wcześniej dostępu. Dzięki temu możemy określić rzeczywistą sprawność sieci która w przypadku uczenia na tym samym zbiorze będzie wraz z kolejnymi epokami zbiegała do 100%.



Rysunek 1 Przykładowe elementy z bazy MNIST

Dla tej części projektu stworzyliśmy dwa programy różniące się zastosowanymi rozwiązaniami co ostatecznie wpływa na wydajność całej sieci.

2.1 Przygotowanie danych

Przygotowanie danych rozpoczyna się od funkcji $load\_images$ która wczytuje obrazy zapisane w binarnym formacie plików MNIST (idx3-ubyte). Funkcja otwiera plik wskazany ścieżką w trybie binarnym, a następnie odczytuje metadane i dane pikselowe obrazów. Na tej podstawie określana jest struktura danych. Następnie funkcja odczytuje pozostałą część pliku jako surowy blok danych o rozmiarze równym liczbie obrazów pomnożonej przez wymiary obrazu. Dane te są ładowane do tablicy NumPy za pomocą metody $np.frombuffer$, która konwertuje bajty na liczby typu uint8. Ostatecznie tablica jest przekształcana w trójwymiarową macierz o wymiarach (num_images, num_rows, num_cols), gdzie każdy obraz jest reprezentowany jako macierz pikseli. Funkcja zwraca tak przygotowaną tablicę NumPy, która stanowi wejście dla dalszego przetwarzania i trenowania modelu.

```python
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
```

Rysunek 2 Funkcja wczytująca obrazy z plików MNIST

W podobny sposób odbywa się wczytywanie pliku z etykietami. Funkcja realizująca to zadanie zwraca tablicę NumPy z etykietami od 0 do 9.

```python
def load_labels(file_path):
    with open(file_path, 'rb') as f:
        magic_number = int.from_bytes(f.read(4), byteorder: 'big')
        num_labels = int.from_bytes(f.read(4), byteorder: 'big')

        buffer = f.read(num_labels)
        labels = np.frombuffer(buffer, dtype=np.uint8)
        return labels
```

Rysunek 3 Funkcja wczytująca etykiety z plików MNIST

Funkcja $prepare\_data$ służy do przygotowania danych wejściowych i etykiet dla procesu trenowania oraz testowania modelu sieci neuronowej. Załadowane dane obrazów mają postać trójwymiarowej macierzy NumPy o wymiarach $(liczba\_obrazów, 28, 28)$, gdzie każdy obraz reprezentowany jest jako macierz pikseli. Dane te są następnie przekształcane: obrazy zostają „spłaszczone" do wektorów o długości 784 pikseli (28×28), aby mogły być przetwarzane przez sieć neuronową, a ich wartości są dzielone przez 255, co normalizuje piksele do zakresu [0, 1]. Etykiety, które początkowo mają postać pojedynczych liczb całkowitych, są konwertowane do formatu one-hot za pomocą funkcji NumPy $np.eye(10)$, co tworzy wektor o długości 10 z wartością 1 w pozycji odpowiadającej danej klasie (np. cyfra „3" to $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$). Funkcja umożliwia również dostosowanie wielkości zbioru elementów na których zostanie przeprowadzone uczenie i testowanie. Dzięki temu możemy wpływać na szybkość trenowania i jego skuteczność.

```python
def prepare_data(train_images_path, train_labels_path, test_images_path, test_labels_path):
    x_train = load_images(train_images_path)
    y_train = load_labels(train_labels_path)
    x_test = load_images(test_images_path)
    y_test = load_labels(test_labels_path)

    x_train = x_train.reshape(-1, 28*28) / 255.0
    x_test = x_test.reshape(-1, 28*28) / 255.0

    y_train_onehot = np.eye(10)[y_train]
    y_test_onehot = np.eye(10)[y_test]

    x_train = x_train[:60000]
    y_train_onehot = y_train_onehot[:60000]
    x_test = x_test[:1000]
    y_test = y_test[:1000]
    y_test_onehot = y_test_onehot[:1000]

    return x_train, y_train_onehot, x_test, y_test, y_test_onehot
```

Rysunek 4 Funkcja przygotowująca dane pobrane z plików bazy MNIST

2.2 Budowa sieci neuronowej

Budowa sieci rozpoczyna się od konstruktora który odpowiada za inicjację parametrów sieci neuronowej takich jak wagi, biasy oraz współczynnik uczenia. Wagi między warstwą wejściową a ukrytą ($weights\_input\_hidden$) oraz między ukrytą a wyjściową ($weights\_hidden\_output$) są inicjalizowane losowo przy użyciu rozkładu normalnego z wartością średnią 0 i odchyleniem standardowym 0.01. Taka inicjalizacja zapewnia, że początkowe wagi mają małe wartości, co pomaga uniknąć problemów związanych z dominacją jednego neuronu w początkowych etapach trenowania. Przesunięcia (biasy) są inicjalizowane jako wektory zerowe, ponieważ początkowe wartości biasów nie wpływają na propagację wsteczną. Wartości wejściowe konstruktora w tym współczynnik uczenia są inicjalizowane ręcznie.

```
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
```

Rysunek 5 Konstruktor klasy NeuralNetwork

2.3 Funkcje aktywacji

W tej implementacji zastosowano dwie funkcje aktywacji: $ReLu$ oraz $sigmoid$, wraz z ich pochodnymi, które są kluczowe w procesie propagacji wstecznej.

Funkcja $relu(x)$ (Rectified Linear Unit) zastosowana w warstwie ukrytej, dla każdej wartości wejściowej x zwraca wartość maksymalną pomiędzy 0 a x. Oznacza to że dla wartości ujemnych wynik wynosi 0 natomiast dla wartości dodatnich sygnał jest przepuszczany bez zmian. Dzięki temu możliwa jest redukcja problemu znikających gradientów. Pochodna funkcji $ReLu$, zwraca wartość 1 dla $x > 0$ i 0 dla pozostałych wartości x. Jest ona niezbędna podczas aktualizacji wag w procesie uczenia sieci, ponieważ określa wpływ danego neuronu na wynik końcowy.

```
@staticmethod
def relu(x):
    return np.maximum( *args: 0, x)
1 usage
@staticmethod
def relu_derivative(x):
    return (x > 0).astype(float)
```

Rysunek 6 Funkcja aktywacji ReLu

Funkcja $sigmoid(x)$ przekształca dowolną wartość na wejściową na wynik mieszczący się w przedziale [0, 1]. W tym przypadku argument jest ograniczony do przedziału $[-500, 500]$, co zapobiega przepełnieniu podczas obliczeń wykładniczych. Pochodna funkcji sigmoidalnej, jest wyrażona jako $x * (1 - x)$, gdzie x to wartość sigmoidy. Odgrywa ona kluczową rolę w propagacji wstecznej, umożliwiając obliczanie zmian wag w zależności od różnicy między wartościami przewidywalnymi a rzeczywistymi.

```
@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)
```

Rysunek 7 Funkcja sigmoidalna

2.4 Funkcja propagacji w przód

Funkcja propagacji w przód to kluczowy element działania modelu, w którym surowe dane są przekształcane na wyniki klasyfikacji lub regresji poprzez przechodzenie przez wszystkie warstwy sieci. W poniższej implementacji funkcja przypisuje dane wejściowe do atrybutu $self.input$. Następnie obliczane jest wejście warstwy ukrytej jako iloczyn macierzowy danych wejściowych i wag warstwy wejściowej, do którego dodawane są biasy. Kolejnym stepem jest obliczenie wejścia warstwy wyjściowej , które powstaje przez przemnożenie wyjść warstwy ukrytej przez wagi warstwy wyjściowej oraz dodanie biasów. Wynik ten jest przekształcany przez funkcję aktywacji $sigmoid$ dzięki czemu otrzymujemy wynik w postaci $[0, 1]$.

```python
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
```

Rysunek 8 Funkcja propagacji w przód

2.5 Funkcja propagacji wstecznej

Funkcja propagacji wstecznej w sieci neuronowej służy do obliczania gradientów błędu i aktualizacji wag oraz biasów. Przyjmuje jako argument rzeczywiste wartości wyjściowe, i wykorzystuje różnicę między nimi a wartościami przewidywanymi do oszacowania błędu na warstwie wyjściowej. Najpierw obliczany jest błąd wyjściowy jako iloczyn różnicy między przewidywaniami a prawdziwymi etykietami oraz pochodnej funkcji aktywacji $sigmoid$ zastosowanej na wyjściu. Następnie błąd ten jest propagowany wstecz przez warstwę ukrytą, co wyznacza iloczyn macierzowy błędu wyjściowego i transponowanych wag warstwy wyjściowej oraz pochodnej funkcji aktywacji $ReLU$ zastosowanej na wejściu do warstwy ukrytej. Na podstawie tych błędów wagi i biasy w obu warstwach są aktualizowane zgodnie z regułą spadku gradientu: od obecnych wartości wag i biasów odejmowany jest iloczyn współczynnika uczenia oraz gradientów obliczonych na podstawie błędów i wyjść odpowiednich warstw. Wagi między warstwą ukrytą a wyjściową są aktualizowane przy użyciu iloczynu wyjść warstwy ukrytej i błędu wyjściowego, a biasy warstwy wyjściowej zmieniane są proporcjonalnie do sumy błędów wyjściowych. Analogicznie wagi między warstwą wejściową a ukrytą są modyfikowane w oparciu o iloczyn danych wejściowych i błędów ukrytych, a biasy warstwy ukrytej aktualizowane są na podstawie sumy błędów ukrytych.

```python
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)
```

Rysunek 9 Funkcja propagacji wstecznej

2.6 Trenowanie sieci

Proces uczenia sieci neuronowej odbywa się poprzez iteracyjne aktualizowanie jej parametrów na podstawie danych treningowych. Rozpoczyna się od inicjacji zmiennej $best\_loss$ (w naszym przypadku na nieskończoność) oraz licznika epok który zapobiega przeuczeniu sieci gdy kolejne epoki nie przynoszą już poprawy. Liczba próbek w danych treningowych jest obliczana na podstawie ich wymiaru. W każdej epoce dane są mieszane przy użyciu permutacji indeksów.

Zastosowana została metoda mini-batchy która dzieli dane treningowe na małe porcje (omawiane mini-batche) i wykorzystuje je do aktualizacji wag modelu w każdej iteracji. Dla każdego mini-batcha wykonywana jest propagacja w przód w celu obliczenia predykcji modelu na danej porcji danych a następnie propagacja wsteczna gdzie obliczane są gradienty funkcji kosztu względem wag, uwzględniając jedynie próbki z danego mini-batcha. Na podstawie tych gradientów odbywa się aktualizacja wag. Proces ten jest powtarzany dla wszystkich mini-batchy w epoce. Następnie dane są ponownie mieszane. Metoda ta poprawia efektywność obliczeń i prowadzi do lepszego wykorzystania zasobów obliczeniowych.

```
for start_idx in range(0, n_samples, batch_size):
    end_idx = start_idx + batch_size
    x_batch = x_train[start_idx:end_idx]
    y_batch = y_train[start_idx:end_idx]
    self.forward(x_batch)
    self.backward(y_batch)
```

Rysunek 10 Trenowanie mini-batchami

Po zakończeniu epoki funkcja oblicza stratę dla całego zbioru treningowego za pomocą średniego błędu kwadratowego MSE oraz dokładność klasyfikacji, porównując przewidywane klasy z rzeczywistymi etykietami. Wyniki te, wraz z czasem trwania epoki, są wyświetlane w konsoli, co pozwala na monitorowanie postępów treningu. Jeśli w bieżącej epoce strata była mniejsza niż dotychczas najlepsza, jest ona zapisywana, a licznik epok resetowany. W przeciwnym przypadku licznik jest zwiększany, a gdy osiągnie z góry ustaloną wartość (w naszym przypadku 20), trening zostaje przerwany

```python
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
```

Rysunek 11 Funkcja trenująca

2.7 Wynik pracy programu

Poniżej znajduje się efekt pracy programu dla 20 epok. Można zauważyć że dokładność trenowania jest wyższa niż dokładność na danych testowych co jest jak najbardziej zrozumiałe gdyż do tego zboru sieć nie miała dostępu podczas nauki. Dodatkowo program losuję jedną cyfrę ze zbioru testowego i wyświetla ją a następnie próbuje określić jej wartość. Ponieważ według programu dokładność na danych testowych wynosi około 98% jeszcze nie zdarzyła mu się pomyłka.

```
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
```

Rysunek 12 Efekt pracy programu

3. Ulepszona wersja programu

W celu przetestowania różnych metod implementacji tej sieci do poprzedniego programu wprowadziliśmy szereg zmian mających na celu poprawę jego wydajności. Do tych zmian możemy zaliczyć:

- Inicjalizację wag metodą $Xavier'a$
- Zmianę funkcji aktywacji w warstwie wyjściowej na $softmax$
- Entropię krzyżową jako funkcję kosztu
- Zmiana sposobu obliczania błędu wyjściowego w propagacji wstecznej

3.1 Inicjalizacja wag metodą $Xavier'a$

Zastosowana w nowym programie metoda inicjalizacji wag, opiera się na losowaniu wartości z równomiernego rozkładu w przedziale $[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}]$, gdzie $n_{in}$to liczba neuronów w poprzedniej warstwie, a n_{out} to liczba neuronów w bieżącej warstwie. Taka inicjalizacja pozwala na lepszą równowagę w sygnałach propagowanych przez sieć, co zmniejsza ryzyko problemów zanikających lub eksplodujących gradientów. W

poprzednim programie wagi były losowane z normalnego rozkładu z małym odchyleniem standardowym (0.01), co mogło prowadzić do wolniejszego zbiegania.

```
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
```

Rysunek 13 Konstruktor inicjujący wagi metodą $Xavier'a$

3.2 Funkcja aktywacji i funkcja kosztu

Funkcja aktywacji $softmax$ w warstwie wyjściowej przekształca wyniki na prawdopodobieństwa sumujące się do 1. Entropia krzyżowa jako funkcja kosztu lepiej odzwierciedla różnice między rozkładami prawdopodobieństw generowanymi przez model a rzeczywistymi etykietami.

```
def softmax(x):
    x_shifted = x - np.max(x, axis=1, keepdims=True)
    exps = np.exp(x_shifted)
    return exps / np.sum(exps, axis=1, keepdims=True)
1 usage
@staticmethod
def cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
```

Rysunek 14 Funkcja aktywacji i funkcja kosztu

3.3 Propagacja wsteczna

Zmiana funkcji aktywacji wymusiła pewne zmiany w funkcji propagacji wstecznej dotyczące błędów. W nowym programie błąd w warstwie wyjściowej jest prostą różnicą między przewidywaniami a rzeczywistymi etykietami. Eliminacja jawnego obliczania pochodnej softmaxa upraszcza gradienty, podczas gdy w poprzednim programie funkcja sigmoidalna i średni błąd kwadratowy wymagają bardziej złożonych operacji, co i zmniejsza efektywność obliczeniową.

```python
def backward(self, y_true):
    output_error = self.final_output - y_true

    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)
```

Rysunek 15 Funkcja propagacji wstecz

3.4 Wynik pracy programu

Na poniższym obrazku widoczny jest efekt pracy ulepszonego programu również dla 20 epok. Sieć była trenowana na pełnym zbiorze 60 000 elementów natomiast do testów wykorzystano losowe 1000 elementów z pośród elementów testowych. Jak widać czas pracy programu znacznie się skrócił a sieć już na początku osiąga znacznie większą dokładność. Dokładność trenowania która poprzedniej implementacji zajęła 20 epok, tutaj została osiągnięta już po 5. Co więcej już na wysokości 15 epoki sieć sięgnęła po dokładność na poziomie 100%. Jak widzimy po wyniku z prób na danych testowych, nie przełożyło się to na poprawienie dokładności która jest nieco mniejsza niż w poprzedniej sieci. Prawdopodobnie jest to wynik przeuczenia sieci która wyspecjalizowała się w przykładach z puli treningowej (ta teza znalazła potwierdzenie po zmniejszeniu liczby epok do 15 kiedy to dokładność osiągnęła 98.30%).

```
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
```

Rysunek 16 Efekt pracy programu

4. Zadanie 2 – Aproksymacja funkcji dwuwymiarowej

Celem zadania było zaimplementowanie prostej sztucznej sieci neuronowej typu MLP (Multi-Layer Perceptron) w celu aproksymacji wartości funkcji dwuwymiarowej $Ackley'a$ na przedziale . Funkcja ta jest często wykorzystywana jako testowa funkcja w optymalizacji ze względu na swoją nieliniowość i liczne minima lokalne.

Aproksymacja polegała na:

1. Wygenerowaniu losowego zbioru punktów uczących w zadanym zakresie.
2. Stworzeniu sieci neuronowej o jednej warstwie ukrytej.
3. Przeprowadzeniu procesu trenowania z wykorzystaniem metody mini-batch.
4. Przeanalizowaniu jakości aproksymacji na regularnej siatce punktów testowych.
5. Zwizualizowaniu wyników aproksymacji w porównaniu z rzeczywistymi wartościami funkcji $Ackley'a$.

4.1 Generowanie zbioru treningowego

Losowe punkty wejściowe zostały wygenerowane w zadanym zakresie, a ich odpowiadające wartości funkcji $Ackley'a$ zostały obliczone. W celu zapewnienia losowości, użyto ziarna inicjalizującego generator liczb pseudolosowych, co gwarantuje powtarzalność wyników w różnych uruchomieniach. Punkty wejściowe były generowane w równomiernym rozkładzie na przedziale dla każdej zmiennej. Następnie, dla każdego punktu obliczano odpowiadającą wartość funkcji $Ackley'a$, która została zapisana jako element wektora wynikowego.

```python
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
```

Rysunek 17 Generowanie zbioru treningowego

4.2 Definicja sieci neuronowej

Aby zrealizować zadanie aproksymacji, stworzono sieć neuronową składającą się z dwóch warstw. Pierwsza warstwa, ukryta, zawierała 10 neuronów, co pozwoliło modelowi na uchwycenie nieliniowych zależności w danych. Warstwa wyjściowa przetwarzała dane w jednowymiarowy wynik, będący aproksymacją funkcji $Ackley'a$. Wagi warstw zostały zainicjalizowane małymi losowymi wartościami, co zmniejszało ryzyko eksplozji gradientów podczas trenowania. Biasy początkowo ustawiono na zero, aby zapewnić równowagę na starcie procesu uczenia.

```python
class SimpleMLP:
    def __init__(self, input_dim, hidden_dim, output_dim, lr=0.01, seed=42):
        np.random.seed(seed)
        self.lr = lr

        self.w1 = np.random.randn(input_dim, hidden_dim) * 0.1
        self.b1 = np.zeros((1, hidden_dim))

        self.w2 = np.random.randn(hidden_dim, output_dim) * 0.1
        self.b2 = np.zeros((1, output_dim))
```

Rysunek 18 Inicjalizacja sieci

4.3 Proces propagacji w przód

Przetwarzanie danych w sieci neuronowej polegało na zastosowaniu dwóch etapów transformacji. Najpierw dane były przemnażane przez wagi pierwszej warstwy i dodawano do nich biasy, a następnie wynik przechodził przez nieliniową funkcję aktywacji, która eliminowała wartości ujemne. W drugim etapie dane były przekształcane w warstwie wyjściowej, co prowadziło do uzyskania końcowego wyniku aproksymacji.

```python
def forward(self, X):
    self.Z1 = X @ self.w1 + self.b1
    self.A1 = self.relu(self.Z1)
    self.Z2 = self.A1 @ self.w2 + self.b2
    y_hat = self.Z2
    return y_hat

def relu(self, x):
    return np.maximum(0, x)
```

Rysunek 19 Proces propagacji w przód

4.4 Proces propagacji wstecznej

Podczas procesu propagacji wstecznej obliczane były gradienty błędu dla każdej warstwy sieci. Rozpoczynano od obliczenia błędu na wyjściu, a następnie propagowano go wstecz przez kolejne warstwy. W warstwie ukrytej uwzględniano wpływ funkcji aktywacji, co zapewniało poprawne obliczanie gradientów. Aktualizacja wag i biasów odbywała się zgodnie z regułami algorytmu spadku gradientu, co umożliwiało stopniowe zmniejszanie błędu predykcji.

```python
def backward(self, X, y, y_hat):
    m = X.shape[0]
    dZ2 = (y_hat - y) / m
    dW2 = self.A1.T @ dZ2
    dB2 = np.sum(dZ2, axis=0, keepdims=True)
    dA1 = dZ2 @ self.w2.T
    dZ1 = dA1 * self.relu_deriv(self.Z1)
    dW1 = X.T @ dZ1
    dB1 = np.sum(dZ1, axis=0, keepdims=True)

    self.w2 -= self.lr * dW2
    self.b2 -= self.lr * dB2
    self.w1 -= self.lr * dW1
    self.b1 -= self.lr * dB1

def relu_deriv(self, x):
    return (x > 0).astype(float)
```

Rysunek 20 Propagacja wsteczna

4.5 Trenowanie sieci

Trenowanie sieci odbywało się iteracyjnie, przez 2000 epok, z użyciem metody mini-batch. Dane były losowo mieszane przed każdą epoką, aby zapobiec nadmiernemu dopasowaniu do ich kolejności. W każdej iteracji wykonywano propagację w przód i wstecz, a następnie aktualizowano parametry sieci. Co 200 epok monitorowano postęp treningu, obliczając średni błąd kwadratowy na zbiorze treningowym.

```python
def train(self, X, y, epochs=1000, batch_size=32, print_loss=True):
    n_samples = X.shape[0]

    for e in range(epochs):
        permutation = np.random.permutation(n_samples)

        X_shuffled = X[permutation]
        y_shuffled = y[permutation]

        for start_idx in range(0, n_samples, batch_size):
            end_idx = start_idx + batch_size
            X_batch = X_shuffled[start_idx:end_idx]
            y_batch = y_shuffled[start_idx:end_idx]

            y_hat_batch = self.forward(X_batch)

            self.backward(X_batch, y_batch, y_hat_batch)

        y_hat_full = self.forward(X)
        loss = np.mean((y_hat_full - y) ** 2)

        if print_loss and e % 200 == 0:
            print(f"[Epoka {e}] MSE (na całym zbiorze): {loss:.6f}")
```

Rysunek 21 Proces trenowania

4.6 Wynik pracy programu

W trakcie trenowania sieci neuronowej zaobserwowano znaczne zmniejszenie błędu średniokwadratowego (MSE).

Po zakończeniu trenowania, sieć została przetestowana na regularnej siatce punktów w przedziale z krokiem 0.02. Średni błąd kwadratowy (MSE) dla danych testowych wyniósł **0.348913**, co świadczy o skutecznej aproksymacji funkcji $Ackley'a$.

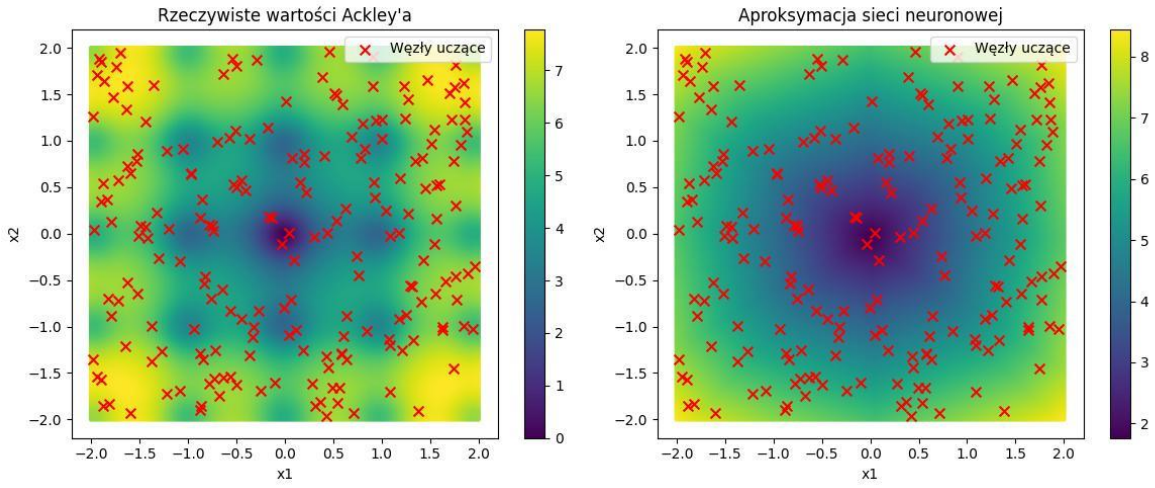
```
[Epoka 0] MSE (na całym zbiorze): 28.604015
[Epoka 200] MSE (na całym zbiorze): 0.456074
[Epoka 400] MSE (na całym zbiorze): 0.375060
[Epoka 600] MSE (na całym zbiorze): 0.351170
[Epoka 800] MSE (na całym zbiorze): 0.341386
[Epoka 1000] MSE (na całym zbiorze): 0.338875
[Epoka 1200] MSE (na całym 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
```

Rysunek 22 Efekt pracy programu

Wyniki zostały zilustrowane na dwóch wykresach:

1. **Rzeczywiste wartości funkcji $Ackley'a$** - przedstawiające rzeczywistą topologię funkcji w badanym zakresie.
2. **Wyniki aproksymacji sieci neuronowej** - ukazujące, jak dokładnie model odwzorował funkcję na podstawie wyuczonych danych.

Na obu wykresach zaznaczono węzły uczące (czerwone punkty), co pozwala ocenić, w jaki sposób sieć uogólnia dane poza próbkami treningowymi. Wizualizacja pokazuje, że model skutecznie odwzorował główne cechy funkcji, w tym obszary o minimalnych wartościach, choć w pewnych regionach widoczne są drobne odchylenia.



Rysunek 23 Wyniki rzeczywiste i aproksymacja

5. Wnioski

1. **Skuteczność aproksymacji**
   Zaimplementowana sieć neuronowa typu MLP skutecznie aproksymowała wartości funkcji $Ackley'a$. Niski średni błąd kwadratowy (MSE) na zbiorze testowym, wynoszący 0.348913, świadczy o zdolności modelu do uogólniania danych.
2. **Rola architektury sieci**
   Zastosowanie jednej warstwy ukrytej z 10 neuronami okazało się wystarczające do uchwycenia głównych cech nieliniowej funkcji $Ackley'a$. Dodanie większej liczby neuronów mogłoby poprawić dokładność, ale mogłoby również zwiększyć ryzyko przeuczenia.
3. **Zastosowanie metody mini-batch**
   Wykorzystanie metody mini-batch pozwoliło na stabilniejsze i bardziej efektywne trenowanie sieci, co wpłynęło na szybszą zbieżność i mniejsze zapotrzebowanie na zasoby obliczeniowe.
4. **Rola wizualizacji**
   Wizualizacja wyników aproksymacji w porównaniu z rzeczywistymi wartościami funkcji potwierdziła skuteczność modelu, ale także wskazała na drobne odchylenia w pewnych obszarach przestrzeni wejściowej. Mogą one wynikać z ograniczeń sieci lub z nierównomiernej gęstości danych treningowych.
5. **Znaczenie doboru hiperparametrów**
   Poprawne ustawienie parametrów takich jak liczba epok, wielkość batchy oraz współczynnik uczenia okazało się kluczowe dla efektywności modelu. Właściwy dobór tych parametrów pozwolił uniknąć przeuczenia i zapewnił dobrą jakość wyników.

6. Źródła

- http://krzysztof.halawa.staff.iiar.pwr.wroc.pl/SieciNeuronowe2.pdf
- https://keras.io/api/layers/activations/
- https://365datascience.com/tutorials/machine-learning-tutorials/what-is-xavier-initialization/