Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Informatyczne Systemy Automatyki

Podstawy sieci neuronowych

Sprawozdanie z projektu wariant #1

Wykonał: Jakub Dębosz, Kacper Krystaszek
Nr albumu: 264197, 264235

21 stycznia 2024

Spis treści

1 Klasyfikacja

1.1 Cel zadania

Pierwsze zadanie polegało na implementacji sieci typu MLP do rozwiązywania prostego problemu klasyfikacji obrazów. Jako obrazy do klasyfikacji obrano obrazy ręcznie narysowanych cyfr od 1 do 9 ze zbioru danych MNIST.

1.2 Realizacja

W celu realizacji zadania zaprojektowano sieć 3 warstwową - warstwa wejścia, warstwa ukryta i wyjście. Tworzyły ją macierze o wymiarach:

- 1x784 - odpowiada każdemu pikselowi obrazu wejściowego,
- 784x128 - warstwa ukryta,
- 128x10 - wyjście odpowiadające każdej cyfrze.

Dane z MNIST są podzielone na dwie kategorie - dane do treningu modelu oraz dane do testowania modelu. Z danych do treningu (60000 próbek) wydzielono 10 tys. losowych danych jako dane do walidacji modelu w trakcie procesu uczenia.

```
X = fetch("http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz")[0x10:].reshape((-1, 28, 28))
Y = fetch("http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz")[8:]

rand = np.arange(60000)
np.random.shuffle(rand)

train_no = rand[:50000]
val_no = np.setdiff1d(rand, train_no)

X_train, X_val = X[train_no, :, :], X[val_no, :, :]
Y_train, Y_val = Y[train_no], Y[val_no]
```

Następnie utworzono model - stworzono macierze typu $float32$ o w/w wymiarach oraz zdefiniowano podstawowe funkcje:

- Funkcję sigmoidalną unipolarną oraz jej pochodną w punkcie - funkcja aktywacyjna modelu:

$$y(x) = \frac{1}{1-e^{-\beta x}}$$

- Funkcję softmax oraz jej pochodną w punkcie - funkcja normalizująca wektor wejściowy na rozkład prawdopodobieństw

$$\sigma(z)_i = \frac{e^{z_i}}{\Sigma_{j=1}^{K} e^{z_j}}$$

Następnie zdefiniowano funkcję forward pass i backpropagation - czyli przejścia danych wejściowych przez model i skorygowanie poszczególnych warstw o błędy, co w efekcie daje uczenie się modelu. W tym celu ze zbioru Y, w którym przechowywano poprawne przypisania cyfr do obrazów ze zbioru X, utworzono zbiór macierzy przypominających macierz wyjściową zawierającą 1 przy indeksie odpowiadającym prawdziwej cyfrze. Następnie wykonano przejście danych przez warstwy poprzez przemnożenie danych wejściowych przez wektory L1, funckję aktywacyjną, L2 i na końcu znormalizowanie wyników funkcją softmax:

```
    #L1 -> L2
    x_l1 = x.dot(l1)
    x_sigmoid = sigmoid(x_l1)

    #L2 -> L3
    x_l2 = x_sigmoid.dot(l2)
    out = softmax(x_l2)
```

Błąd obliczono jako różnicę wektora wyjściowego i wcześniej utworzonego wektora wynikowego, dzielonego przez ilość neuronów i pochodną dla L2 funkcji softmax, a dla L1 funkcji sigmoidalnej. Otrzymano dzięki temu "kierunek" błędów. W celu realizacji procesu uczenia utworzono zmienne $update\_l1$ i $update\_l2$, które posłużą później do aktualizacji modelu. Po przygotowaniu wszystkich funkcji utworzono proces nauczania, zdefiniowano podstawowe parametry:

- epoki = 10000
- prędkość nauczania = 0.01
- batch = 128 (ilość danych brana co iterację)

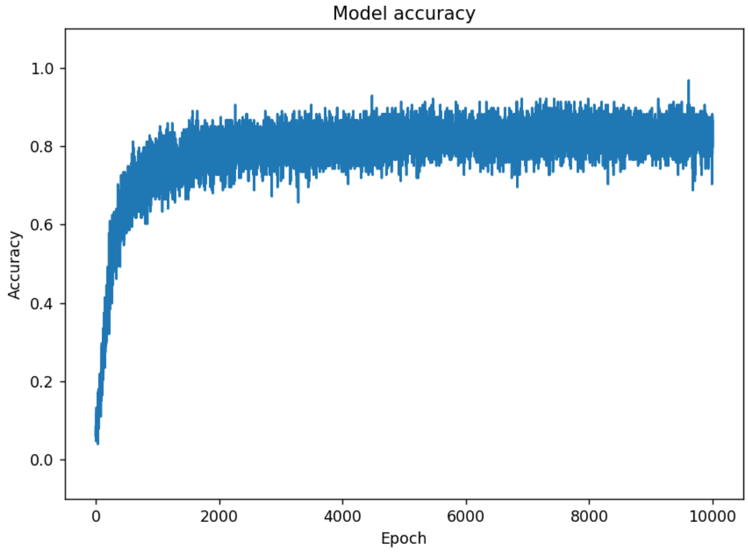
Proces nauczania przebiegał następująco:

- Wybrano losową próbkę danych ze zbioru, przygotowano je jako wektory pasujące do wejścia modelu,
- przyporządkowano próbce poprawne dane wyjściowe ze zbioru Y,
- zaaplikowano funkcję przejścia i backpropagation,
- następnie wyjście porównano z poprawnymi danymi, obliczono MSE,
- zaktualizowano model o poprawki modelu z obecnej iteracji,
- co 20 iteracji ze zbioru walidacyjnego pobierano próbki i sprawdzano przebieg nauczania.

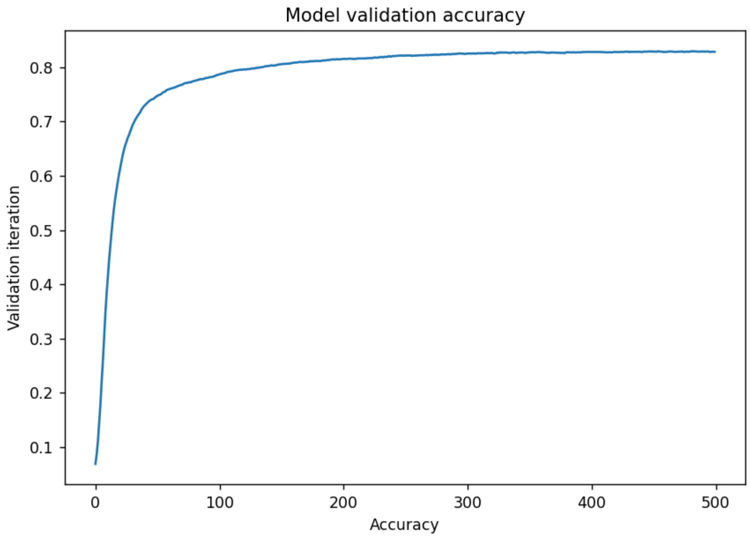
1.3 Wyniki i testowanie

W ten sposób otrzymano model uczący się rozpoznawać ręcznie pisane cyfry. Dla dobranych parametrów, funkcji i implementacji, otrzymano następujące wyniki:

- Celność modelu, sprawdzana w trakcie nauczania na danych do trenowania, oraz walidacja na losowych próbkach:

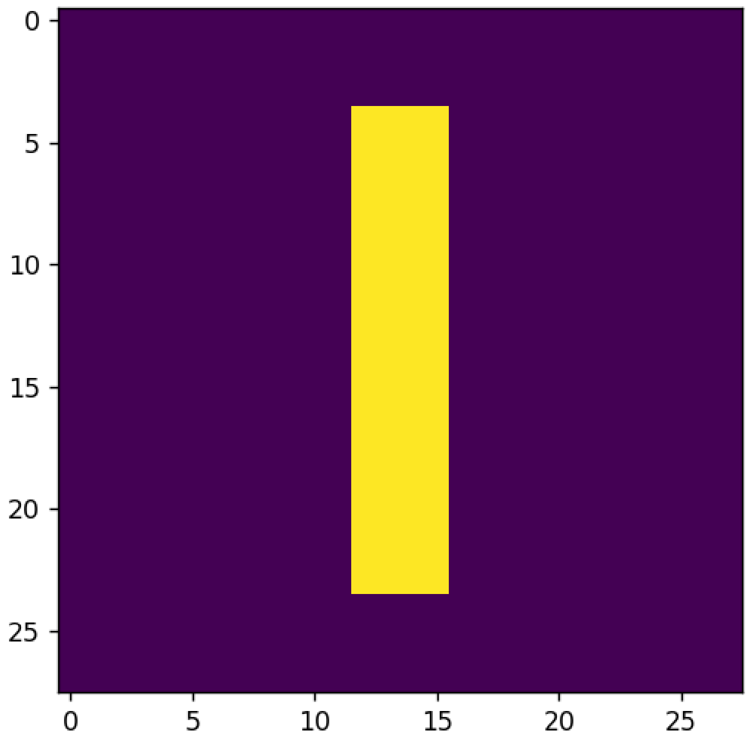


Rysunek 1: Celność modelu w trakcie procesu trenowania



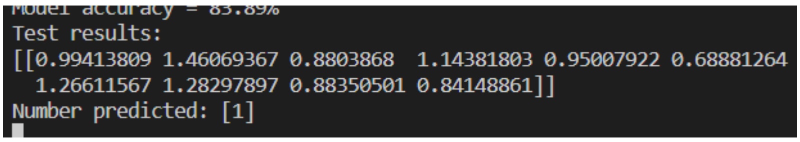
Rysunek 2: Walidacja modelu na losowych próbkach

Za pomocą zbioru do testowania sprawdzono ostateczną celność modelu testując go na całym zbiorze. Otrzymano dzięki temu model o poprawności wynoszącej $\approx$ 83.70%. Dodatkowo, przeprowadzono testy "ręczne". Pierwszym testem było utworzenie prostej macierzy 7x7 na której zaimitowano cyfrę. Przeskalowano ją tak, by nadawała się jako wejście do modelu i sprawdzono, jaką będzie cyfrą:



Rysunek 3: Imitacja cyfry 1 jako macierz

Otrzymano w wyniku:

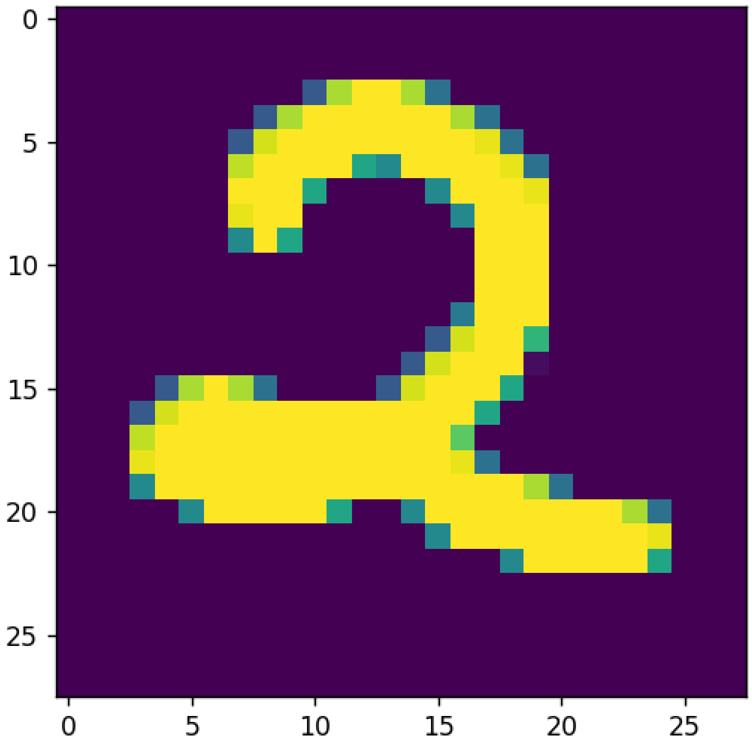


```
Model accuracy = 83.89%
Test results:
[[0.99413809 1.46069367 0.8803868  1.14381803 0.95007922 0.68881264
  1.26611567 1.28297897 0.88350501 0.84148861]]
Number predicted: [1]
```

Rysunek 4: Wynik testu

Przedstawiona macierz pokazuje wartości przewidywane przez model, gdzie wartość najwyższa, to wartość najbardziej prawdopodobna wg. modelu. W tym przypadku poprawnie wskazana jako wartość 1.

Drugim testem było utworzenie prostej bitmapy 28x28 i narysowanie na niej cyfry. Plik z cyfrą wczytano do programu, przygotowano go jako macierz 28x28, którą można zastosować na modelu:



Rysunek 5: Przedstawienie narysowanej ręcznie cyfry 2 jako bitmapa

Otrzymano w wyniku:

```
Number predicted: [1]
Test results:
[[1.5486794 1.7297937 2.1596498 1.6049433 1.5992514 0.7464386 1.8849802
  1.5067074 1.3681781 1.1498228]]
Number predicted: [2]
```

Rysunek 6: Wynik testu

2 Aproksymacja

2.1 Cel zadania

Drugie zadanie polegało na implementacji sieci typu MLP do aproksymowania wartości funkcji jednej zmiennej. Do treningu użyto podanego w treści zadania zbioru 11 par punktów.

2.2 Realizacja

W celu realizacji zadania użyto mechanizmu generowania sieci z różną liczbą warstw ukrytych oraz różną liczbą neuronów (wszystkie warstwy ukryte miały taką samą liczbę neuronów). Warstwy wejścia i wyjścia posiadały 1 jednostkę.

Aby stworzyć zbiór punktów na podstawie, którego sieć odtworzy funkcję, wykonano próbki co 6 minut co pozwoliło stworzyć ciąg

$$x = 0.0, 0.1, ..., 10.0 \tag{1}$$

Następnie dane treningowe zostały znormalizowane do wartości pomiędzy 0 i 1.

```python
def create_func(x_values, y_values):
    result_x = []
    result_y = []

    for index, x in enumerate(x_values):
        if index + 1 != len(x_values):
            result_x.append(np.linspace(x-1, x, 11)[:-1])
        else:
            result_x.append(np.linspace(x-1, x, 11))

    for index, y in enumerate(y_values[1:], start=1):
        if index + 1 != len(y_values):
            result_y.append(np.linspace(y_values[index-1], y, 11)
    [:-1])
        else:
            result_y.append(np.linspace(y_values[index-1], y, 11))

    return np.concatenate(result_x).reshape(-1, 1), np.concatenate(
    result_y).reshape(-1, 1)

# Test data
x_train = np.array([float(x) for x in range(11)]).reshape(-1, 1)
y_train = np.array([1.0, 1.32, 1.6, 1.54, 1.41, 1.01, 0.6, 0.42,
    0.2, 0.51, 0.8]).reshape(-1, 1)

x, y = create_func(x_train[1:], y_train)

# Normalize the data between 0 and 1
y_max = y_train.max()
y_train /= y_max
```

Znormalizowanie wartości jest wymagane, aby użyć sigmoidalnej bipolarnej - funkcji aktywacyjnej modelu.

$$f_b(u) = tanh(\beta u) \tag{2}$$

Następnie stworzono klasę *MultilayerPerceptron* wraz z metodami potrzebnymi do nauczania sieci oraz metodami pomocniczymi do obsługi sieci. Metodą wykorzystaną do nauczania sieci jest *backpropagation error*.

```
class MultilayerPerceptron:
    def __init__(self, input_size, hidden_sizes, output_size, learning_rate=0.01):
        self.input_size = input_size
        self.hidden_sizes = hidden_sizes
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.num_layers = len(hidden_sizes) + 1
        self.activations = None
        self.layer_outputs = None

        # Initialize weights and biases for the network
        self.weights = [np.random.randn(input_size, hidden_sizes[0])]
        self.weights.extend([np.random.randn(hidden_sizes[i], hidden_sizes[i+1]) for i in range(len(hidden_sizes) - 1)])
        self.weights.append(np.random.randn(hidden_sizes[-1], output_size))

        self.biases = [np.zeros((1, size)) for size in hidden_sizes]
        self.biases.append(np.zeros((1, output_size)))
```

Następnie rozpoczęto proces uczenia o podanych poniżej parametrach:

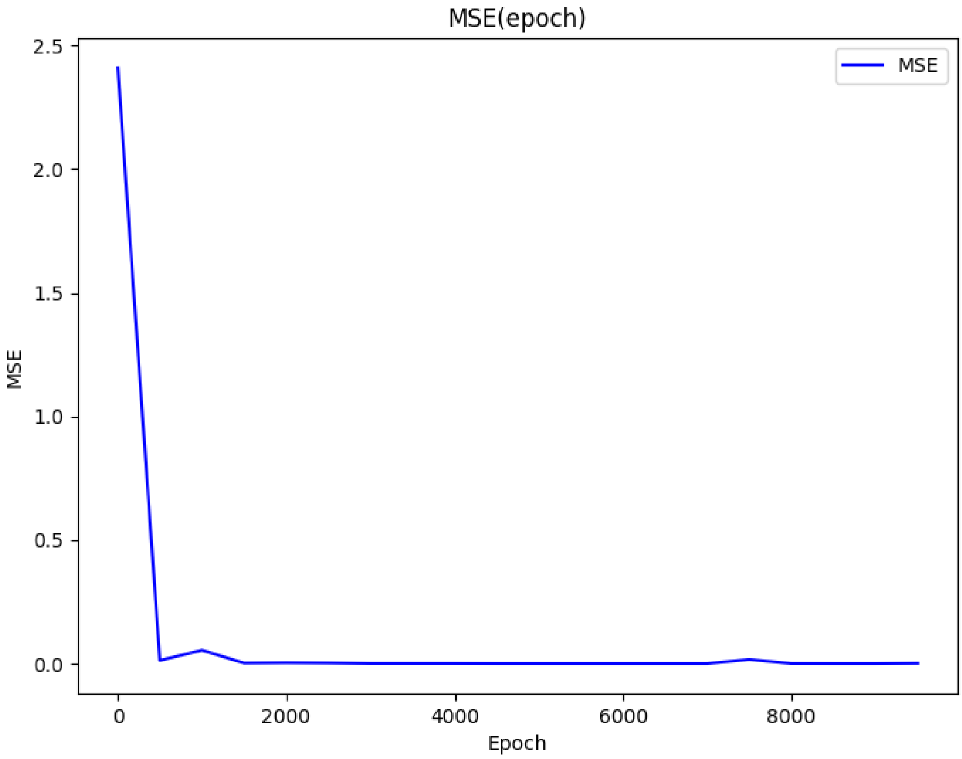
- liczba epok - 10000
- prędkość nauczania - 0.001

Proces nauczania przebiegał następująco:

- podano modelowi dane do nauczania
- wykonano metodę *forward pass* i *backpropagation*
- porównano wyjście z danymi testowymi i obliczono MSE

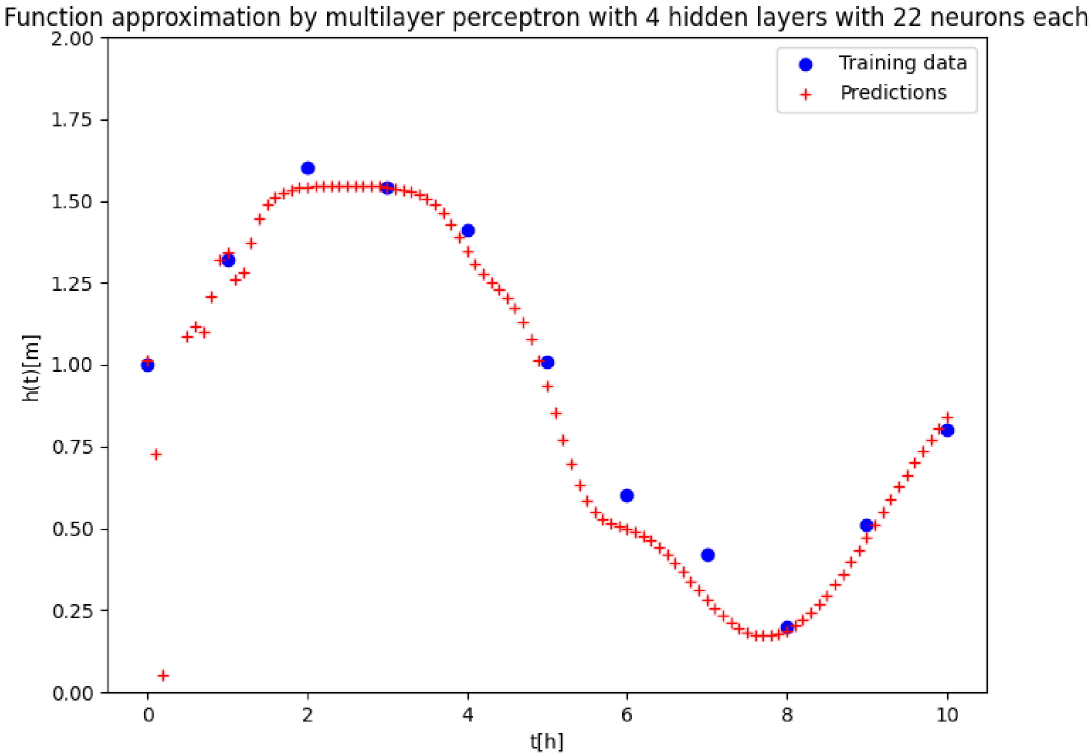
2.3 Wyniki i testowanie

W wyniku nauczania otrzymano model, który potrafi zaproksymować wartości dla funkcji o kształcie podobnym do tego, na którym przebiegło nauczanie. Poniżej znajduje się wykres błędu średniokwadratowego w zależności od epoki.

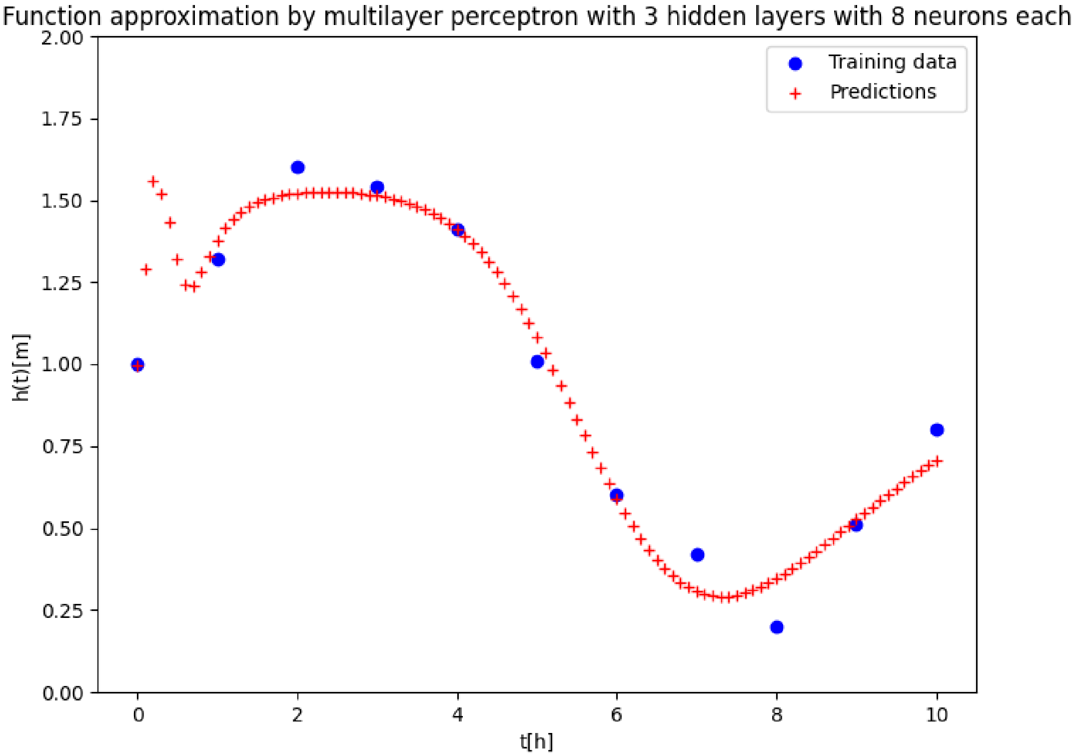


Rysunek 7: Błąd średniokwadratowy w zależności od epoki

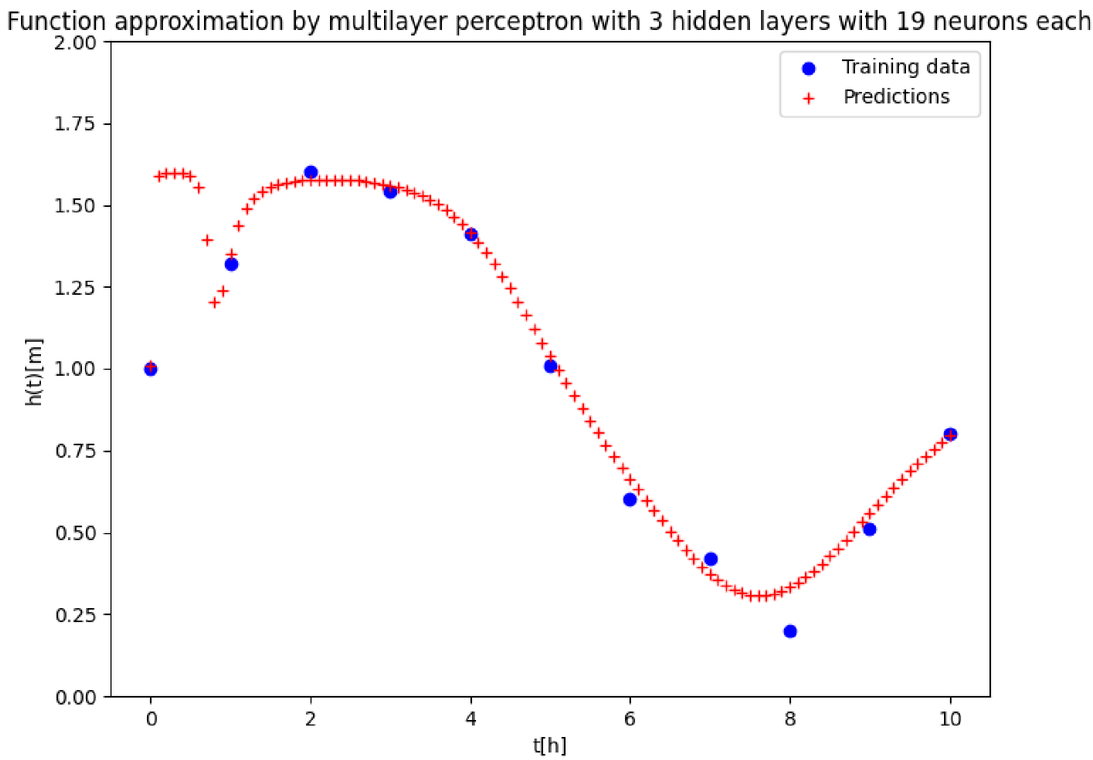
Po całym procesie nauczania model jest w stanie podać wartość funkcji z błędem ≈ 0.0033. Poniżej znajdują się wykresy funkcji oraz wartości przewidziane przez model o różnej liczbie warstw ukrytych oraz neuronów.



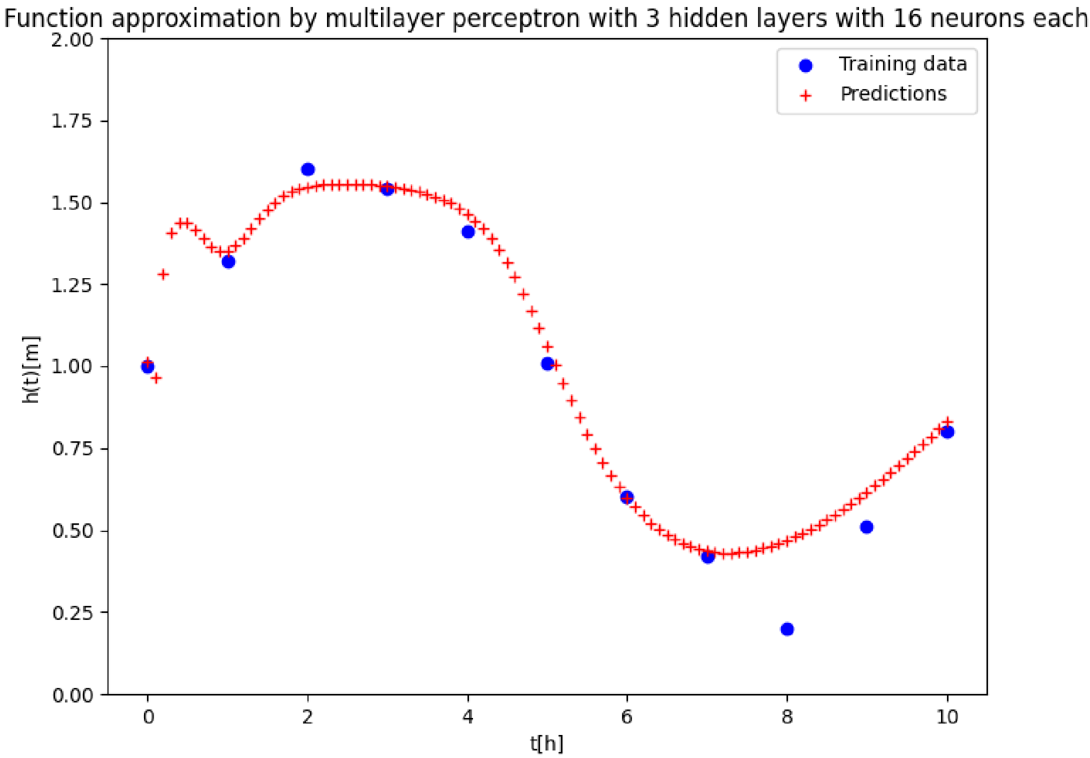
Rysunek 8: Wartości zaproksymowane przez model z 4 warstwami ukrytymi i 22 neuronami w każdej



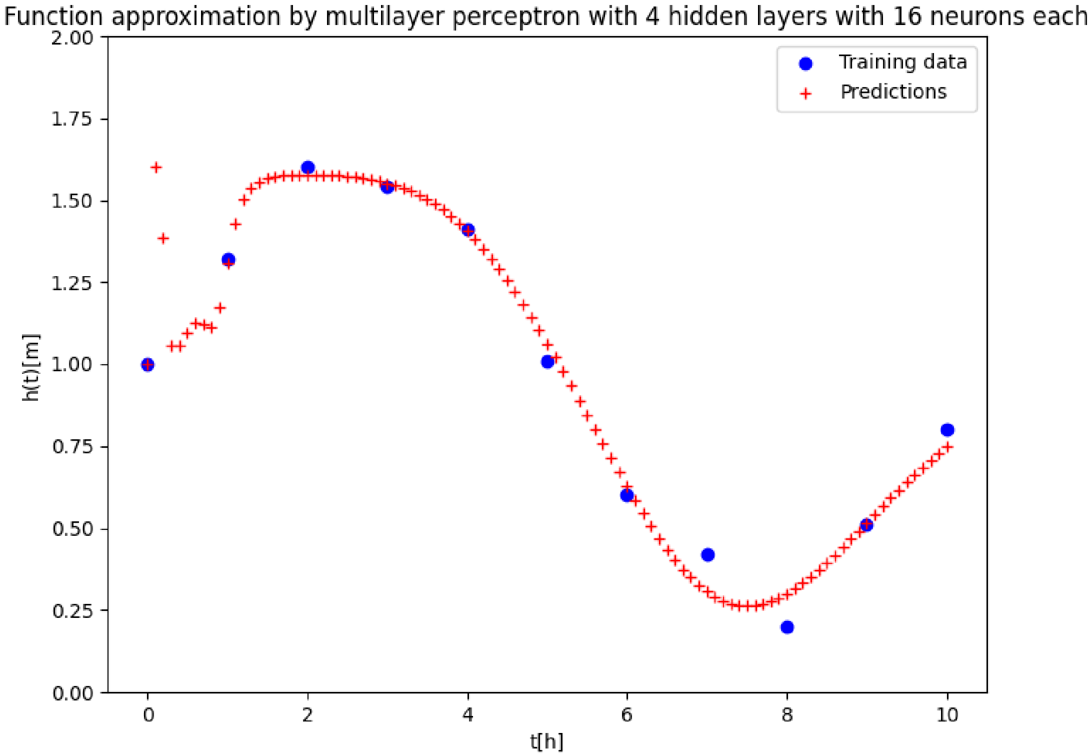
Rysunek 9: Wartości zaproksymowane przez model z 3 warstwami ukrytymi i 8 neuronami w każdej



Rysunek 10: Wartości zaproksymowane przez model z 3 warstwami ukrytymi i 19 neuronami w każdej



Rysunek 11: Wartości zaproksymowane przez model z 3 warstwami ukrytymi i 16 neuronami w każdej



Rysunek 12: Wartości zaproksymowane przez model z 4 warstwami ukrytymi i 16 neuronami w każdej

Analizując powyższe wykresy można zauważyć, że model z 4 warstwami ukrytmi i 16 neuronami najbardziej zbliżył się do danych testowych. Można także wywnioskować, że to liczba warstw najbardziej odpowiada za kształt odwzorowania funkcji, a liczba neuronów temu jak duży jest błąd.

Literatura

[1] https://pl.wikipedia.org/wiki/Funkcja_aktywacji

[2] https://en.wikipedia.org/wiki/Softmax_function

[3] https://towardsdatascience.com/neural-networks-forward-pass-and-backpropagation-be3b75a1cfcc