Jan Mogielski 303899

Laboratorium 5
Sztuczne sieci neuronowe

28 maja 2023

Spis treści

1. Cel laboratorium

Celem laboratorium było napisanie kodu w języku Programowania Python i zaimplementowanie perceptronu wielowarstwowego oraz wybranego algorytmu optymalizacji gradientowej z algorytmem propagacji wstecznej.

Perceptron wielowarstwowy ma za zadanie rozpoznawać ręcznie napisane cyfry w zakresie od 0 do 9.

2. Zbiór danych

Wytrenowanie naszego modelu wymaga dużych ilości danych, w tym celu wykorzystamy zbiór danych MNIST, a dokładniej mnist_784, który zawiera 70000 przykładów ręcznie napisanych cyfr. Każda cyfra jest zapisana w postaci czarno-białego obrazka składającego się z 28x28 pikseli. Każda cyfra jest odpowiednio opisana etykietą w celu jej identyfikacji i późniejszego trenowania.

Dane dzielimy na 3 pod zestawy: trenujący, testujący i walidacyjny. Powszechną praktyką jest podział danych w stosunku 80:20 i taki też zastosowałem. Do podziału danych użyłem gotowej funkcji "train_test_split" wbudowanej w bibliotekę scikit-learn. Dodatkowo zaimplementowałem w kodzie zliczanie wystąpień danej cyfry w celu lepszej analizy wyników.

Przed podziałem danych został zastosowany StandardScaler, który pomaga znormalizować wartości intensywności pikseli, może poprawić wydajność algorytmu, promuje niezależność funkcji i zwiększa interpretowalność wyników.

3. Implementacja

Implementacja dzieli się na dwa elementy: załadowanie danych oraz zbudowanie pereceptronu wielowarstwowego.

Korzystam z podstawowych bibliotek: numpy, scikit-learn, matplotlib, pandas, time, random oraz seaborn.

Ładowanie danych to użycie zestawu danych, który jest wbudowany w bibliotekę scikit-learn. Następnie dokonujemy stosujemy StandardScaler, podział danych oraz OneHotEncoder, aby przypisać każdej cyfrze osobną etykietę.

Widzimy, że najwięcej jest cyfr o wartości "1", a najmniej o wartości "5".

```
Loading MNIST dataset...
1    7877
7    7293
3    7141
2    6990
9    6958
0    6903
6    6876
8    6825
4    6824
5    6313
Name: class, dtype: int64
Dataset size: 70000 images
Train set: 42000 images
Validation set: 14000 images
Test set: 14000 images
```

Rys. 1. Podział danych

Implementacja perceptronu odbywa się w klasie MLP (Multi Layer Perceptron). W klasie mamy funkcje, które pozwalają nam na trenowanie naszej sieci neuronowej. Funkcje w klasie to:

— forward,
— backward,

— relu,
— relu_derivative,
— softmax,
— one_hot_encoder.

Klasa MLP przyjmuje trzy wejściowe parametry:

— input_size,
— hidden_size,
— output_size.

input_size to parametr wielkości na wejściu sieci neuronowej, w naszym przypadku wielkość warstwy wejściowej to 784, ponieważ każdy obrazek składa się z 28x28 pikseli. Każdy piksel jest reprezentowany przez 1 z 784 neuronów na warstwie wejściowej.

hidden_sizes to parametr, który mówi nam o tym ile ukrytych warstw mamy w swojej sieci oraz ile neuronów ma poszczególna warstwa.

output_size parametr określający nam wynik końcowy działania trenowania naszej sieci, w naszym przypadku mamy 10 różnych cyfr, więc wartość tego parametru to 10.

Na samym początku tworzymy puste tablice dla wag i tendencyjności poszczególnych neuronów. Na początku każdy neuron jest wypełniany losowymi wartościami, które będą aktualizowane w trakcie uczenia. Warstwa wyjściowa jest na początku wypełniona zerami.

Funkcja forward ma za zadanie wykonać przejście danych od warstwy wejściowej do warstwy wyjściowej. Dla każdej warstwy ukrytej dokonujemy przejścia danych i aktualizacji wag i tendencyjności, a następnie zapisujemy wyniki i przekazujemy je na warstwę wyjściową. Kolejną funkcją jest funkcja backward, która na wejściu przyjmuje zestaw danych oraz parametr learning_rate. Dokonujemy przepuszczenia danych w drugą stronę od warstwy wyjściowej do warstwy wejściowej. W tym czasie dokonujemy aktualizacji parametru delta, którego użyjemy do obliczenia gradientów. Obliczone gradienty wykorzystujemy do aktualizacji wartości wag i tendencyjności dla poszczególnych neuronów.

Funkcja relu zwraca nam maksymalną wartość na warstwie wyjściowej. Pochodna funkcji relu zwraca nam wartość 0 dla wartości mniejszych niż zero, a wartość 1 dla wartości dodatnich.

4. Trenowanie sieci neuronowej

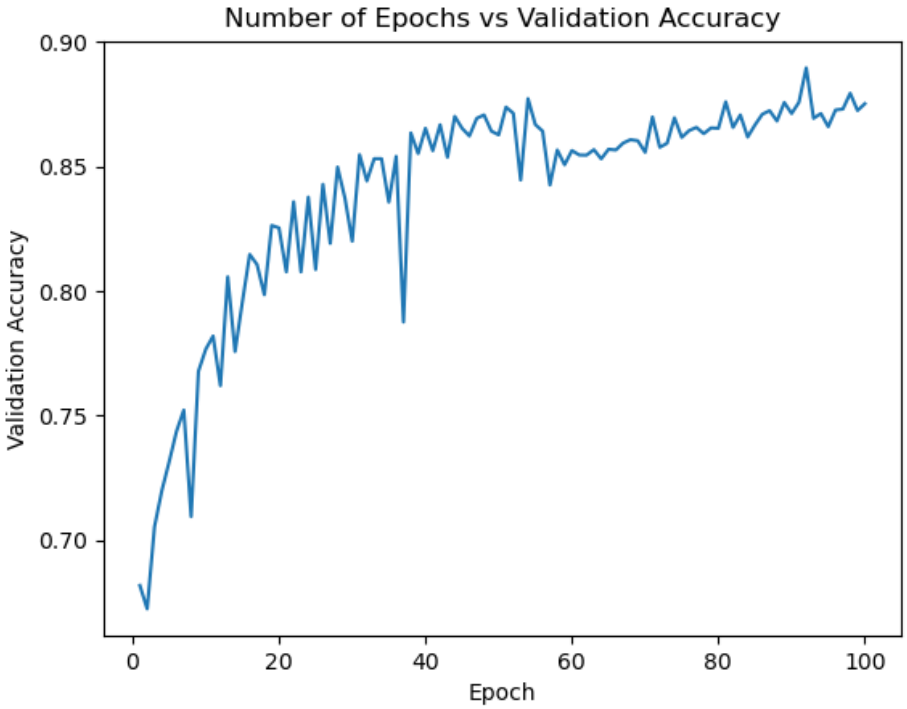
Podczas trenowania naszej sieci musimy zapewnić kilka istotnych parametrów:
— hidden_sizes - określamy ilość ukrytych warstw i ilość neuronów,
— learning_rate - szybkość uczenia się sieci neuronowej, im więcej warstw ukrytych tym mniejszy powinien być,
— outpu_size - dynamicznie ustanawiany przez ilość unikalnych elementów w zbiorze danych,
— num_epochs - liczba epoków w trakcie uczenia sieci neuronowej, im więcej tym większą dokładność powinniśmy uzyskać,
— batch_sizes - ilość próbek, po których dokonujemy aktualizacji wag i współczynnika tendencyjności.

Trenowanie odbywa się na wcześniej podzielonych danych z wykorzystaniem różnych konfiguracji sieci neuronowej. Dzięki wysokiej mocy obliczeniowej byłem w stanie przeprowadzić kilka różnych trenowań w ciągu kilku godzin. Jednak ilość możliwych kombinacji przerasta moje możliwości na sprawdzenie wszystkich, najlepszym rozwiązaniem byłoby wykorzystanie wbudowanej funkcji GridSearchCV albo RandomSerachCV, które sprawdziłyby wiele różnych możliwości. Jednak implementacja tej funkcji w moim kodzie jest bardzo pracochłonna i zrezygnowałem z takiego podejścia. Wyniki przedstawię w sekcji Wnioski.

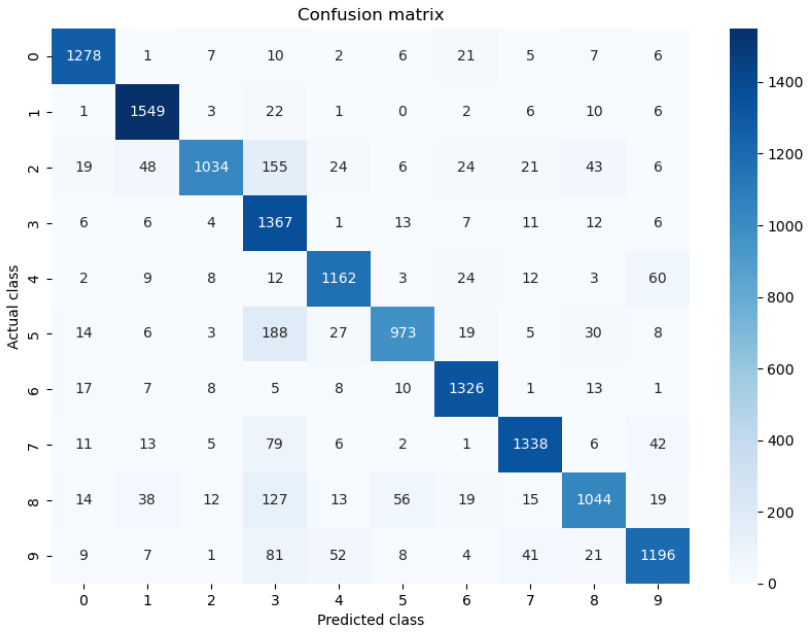
5. Wnioski

Sprawdziłem kilka różnych wariantów działania sieci neuronowej dla różnych ilości warstw ukrytych oraz ilości neuronów.

5.1. test_size = 0.2, val_size = 0.25, hidden_sizes = 32, learning_rate = 0.01, num_epochs = 100, batch_size = 32



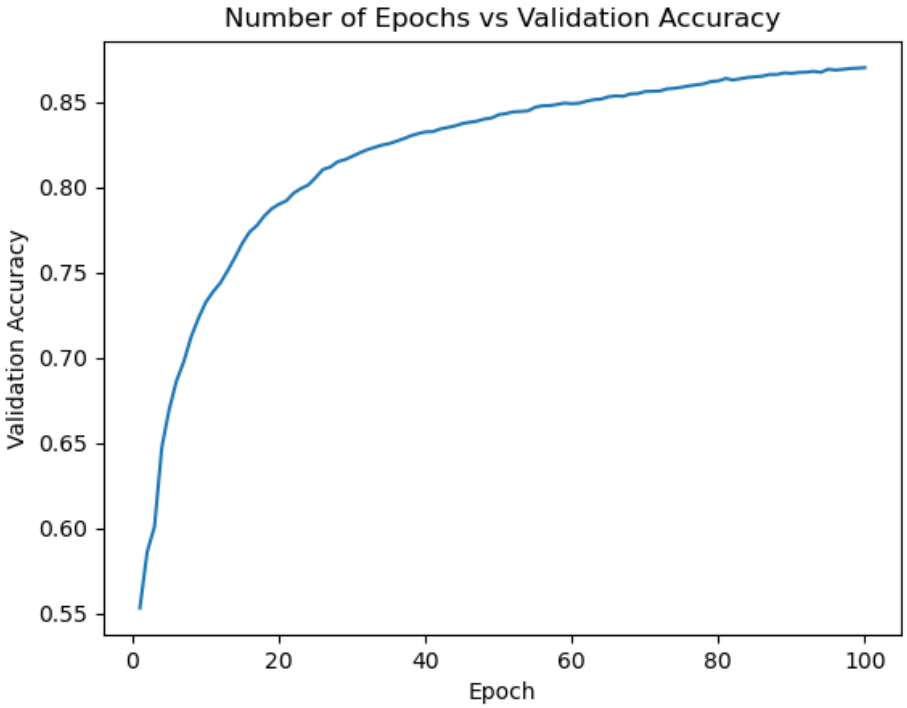
Rys. 2. Zależność dokładności od ilości epochs



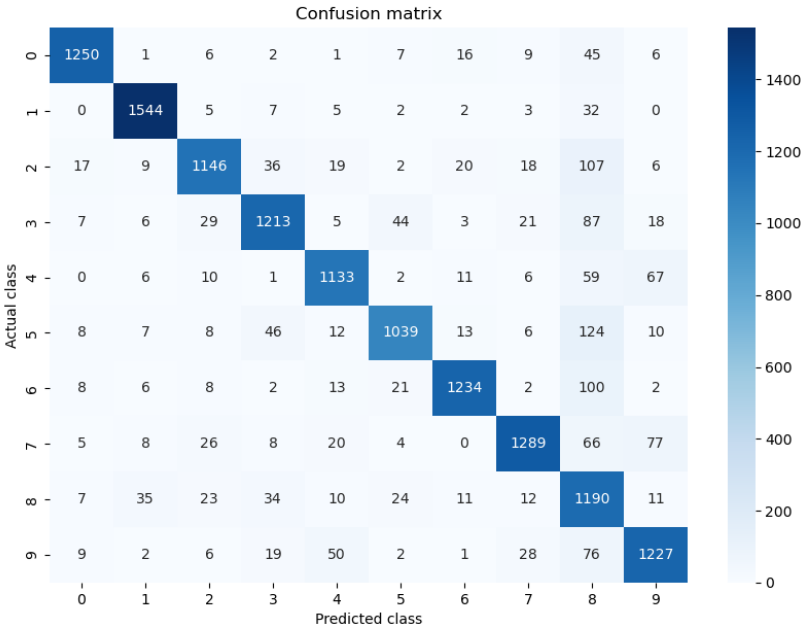
Rys. 3. Confusion matrix dla jednej ukrytej warstwy z 32 neuronami

Test Accuracy: 0.7354
Execution Time: 335.09 seconds

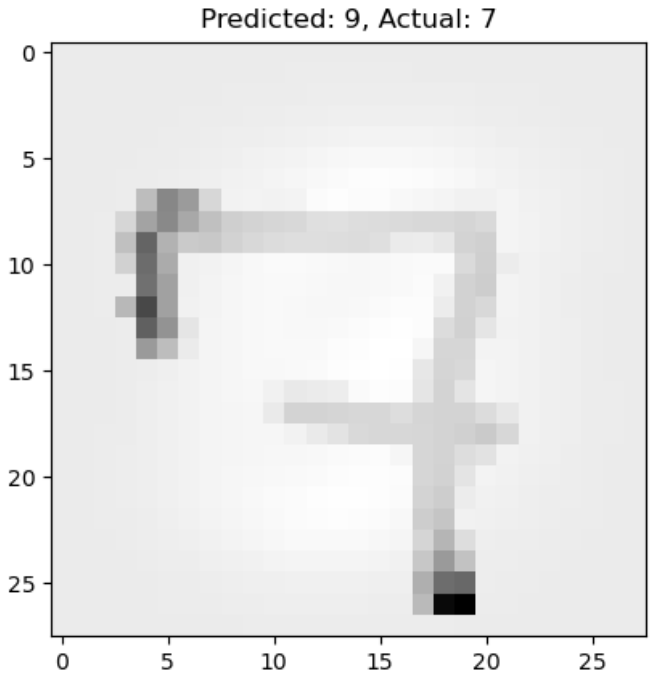
5.2. test_size = 0.2, val_size = 0.25, hidden_sizes = [64, 64], learning_rate = 0.01, num_epochs = 100, batch_size = 32



Rys. 4. Zależność dokładności od ilości epochs



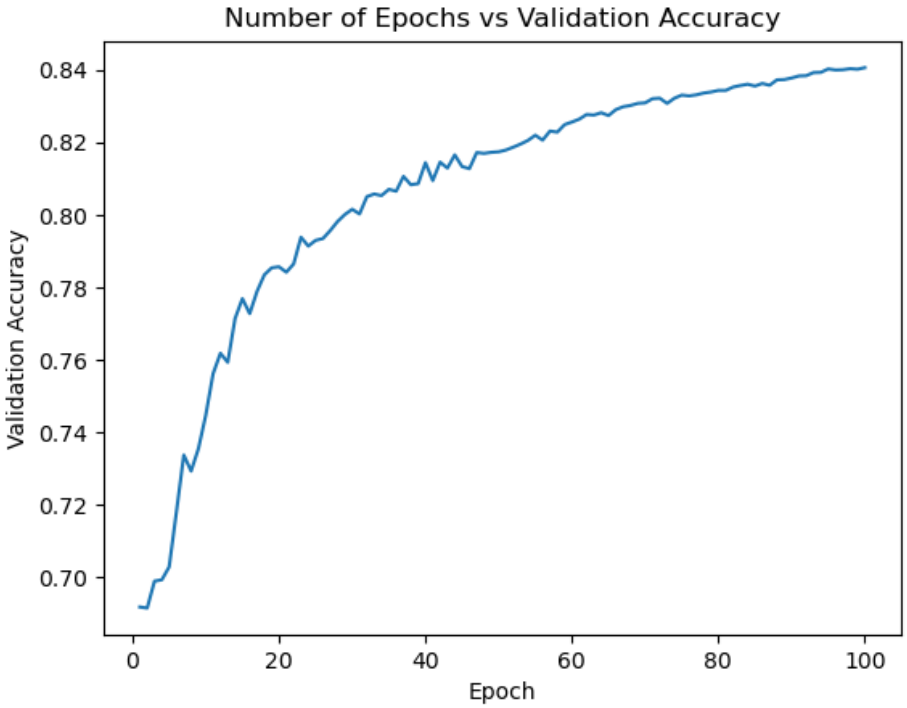
Rys. 5. Confusion matrix dla dwóch ukrytych warstw z 64 neuronami



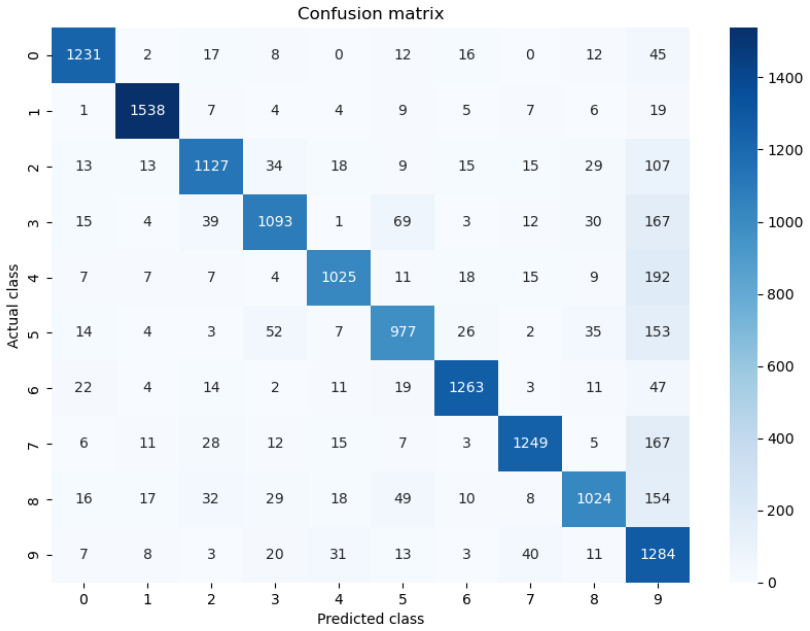
Rys. 6. Błędnie przewidziana liczba

Test Accuracy: 0.7862
Execution Time: 449.48 seconds

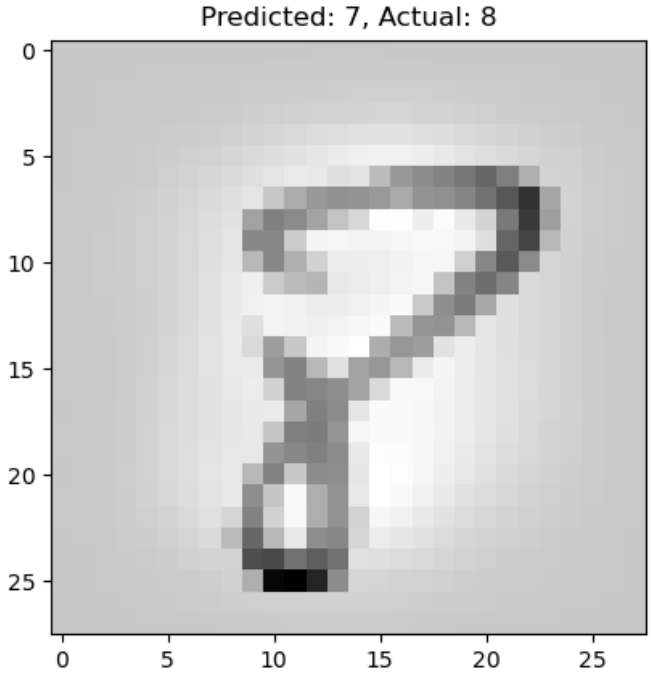
5.3. test_size = 0.2, val_size = 0.25, hidden_sizes = [128, 128], learning_rate = 0.01, num_epochs = 100, batch_size = 32



Rys. 7. Zależność dokładności od ilości epochs



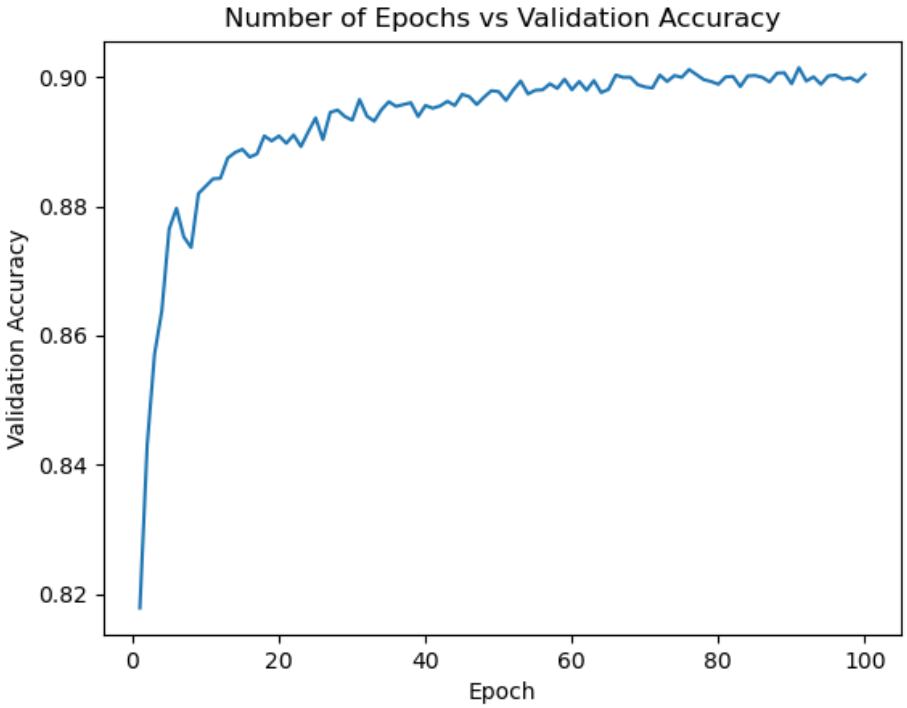
Rys. 8. Confusion matrix dla dwóch ukrytych warstw z 128 neuronami



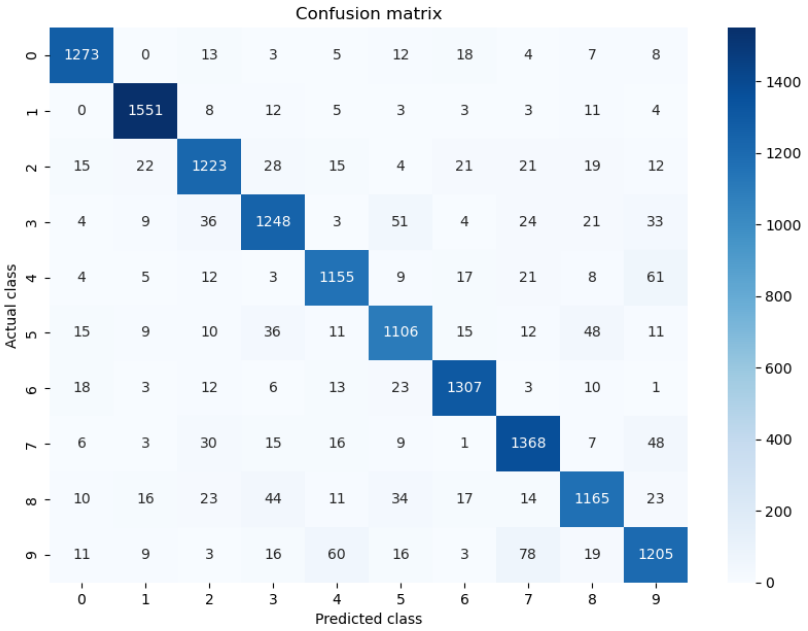
Rys. 9. Błędnie przewidziana liczba

Test Accuracy: 0.8436
Execution Time: 561.21 seconds

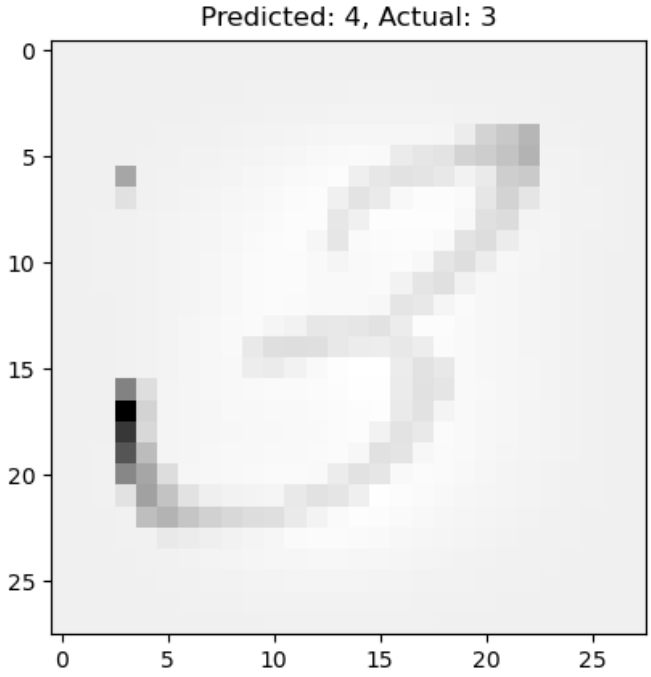
5.4. test_size = 0.2, val_size = 0.25, hidden_sizes = [128, 128], learning_rate = 0.001, num_epochs = 100, batch_size = 32



Rys. 10. Zależność dokładności od ilości epochs



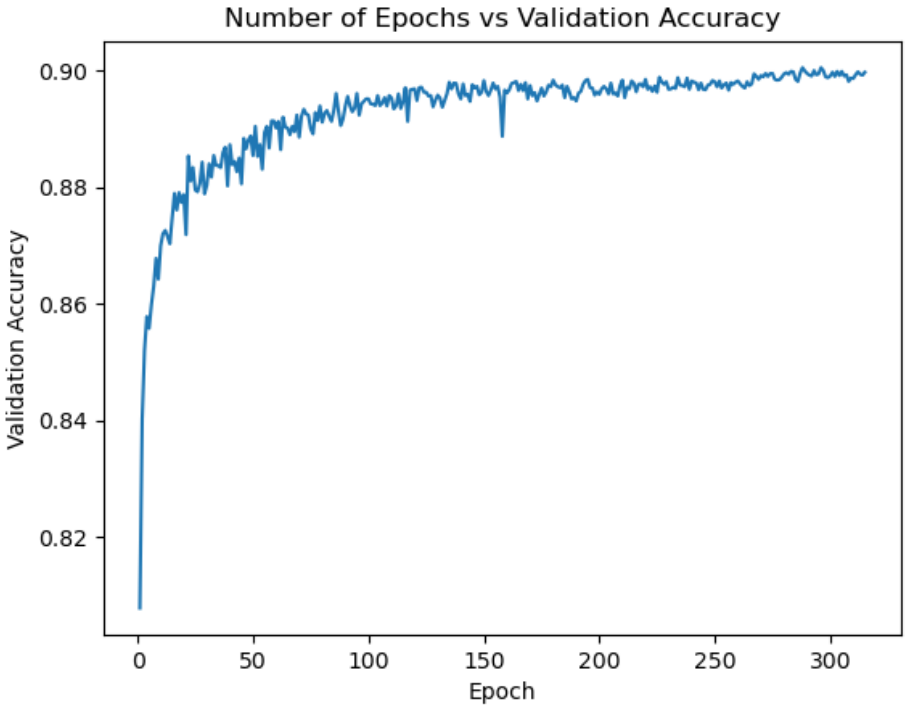
Rys. 11. Confusion matrix dla dwóch ukrytych warstw z 128 neuronami



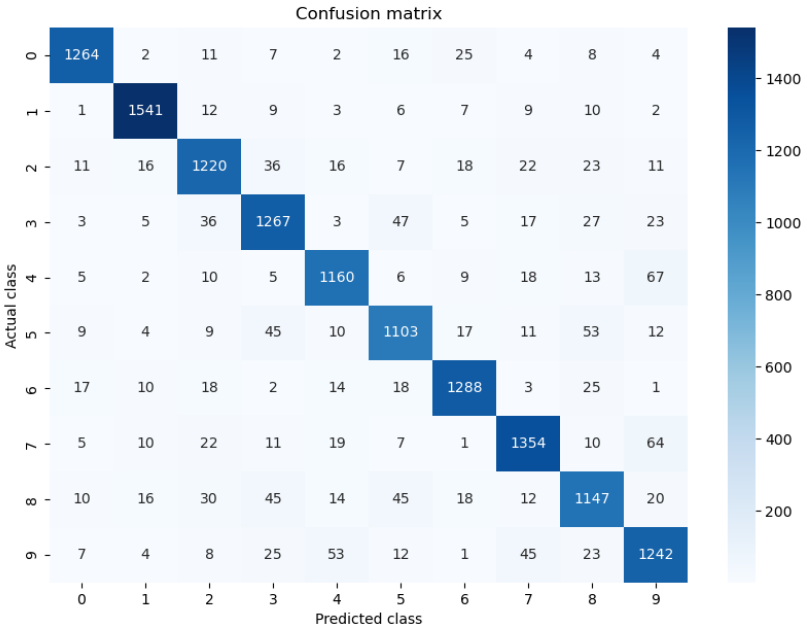
Rys. 12. Błędnie przewidziana liczba

Test Accuracy: 0.9001
Execution Time: 610.11 seconds

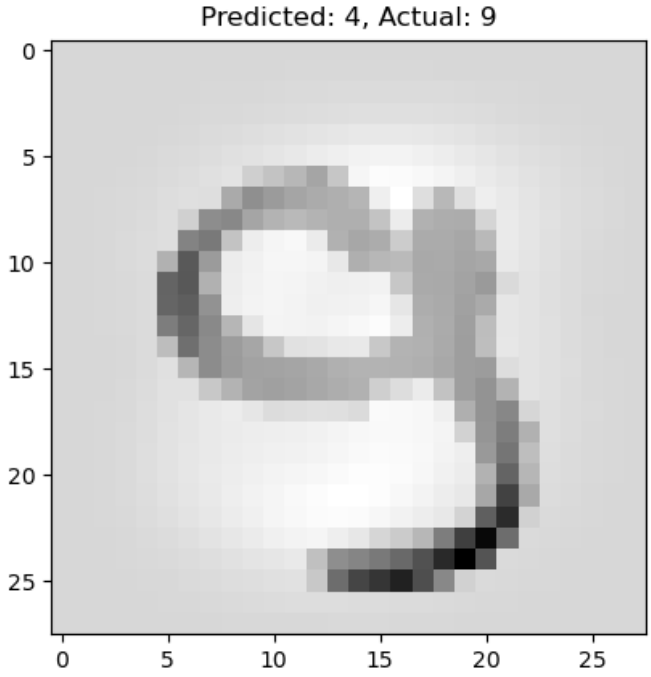
5.5. test_size = 0.2, val_size = 0.25, hidden_sizes = [128, 128], learning_rate = 0.001, num_epochs = 315, batch_size = 32



Rys. 13. Zależność dokładności od ilości epochs



Rys. 14. Confusion matrix dla dwóch ukrytych warstw z 128 neuronami



Rys. 15. Błędnie przewidziana liczba

Test Accuracy: 0.9191
Execution Time: 734.45 seconds

Jak widać wyniki jasno prezentują, że większa sieć neuronowa pozytywnie wpływa na dokładność rozpoznawania ręcznie napisanych cyfr. Dodatkowo zwiększanie ilości neuronów również pozytywnie wpływa na dokładność modelu. Jednak większe sieci neuronowe wymagają mniejszego współczynnika uczenia oraz zajmują więcej czasu jeśli chodzi o ich wytrenowanie. Większość wykresów pokazuje logiczną tendencję, gdzie większa ilość epochs pozytywnie wpływa na dokładność modelu, jednak dla niskiej ilości neuronów proces trenowania jest nierówny. Wszystkie Confusion Matrix wyglądają dobrze, ponieważ występuje niewiele pomyłek wśród liczb i ich predykcji.

Duże sieci neuronowe dokonują niewielu pomyłek, ale dla wszystkich liczb. Małe sieci neuronowe dokonują większej ilości pomyłek ale dla kilku liczb. Prawdopodobnie stworzenie większej sieci neuronowej z większą ilością neuronów doprowadziłoby do mniejszej ilości pomyłek i lepszej dokładności samej sieci, jednak jej wytrenowanie zajęłoby znacząco więcej czasu, a zysk mógłby być niewielki.

Podsumowując udało się stworzyć i wyszkolić sieć neuronową, która w poprawny sposób rozpoznanie ręczne pismo.