Raport
Wprowadzenie do Sztucznej Inteligencji – Ćwiczenie 5.

Sztuczne sieci neuronowe

Autorzy:
Aleksandra Jamróz, nr albumu: 310 708
Bartosz Latosek, nr albumu: 310 790

Treść zadania

W ramach piątych ćwiczeń należy zaproponować architekturę, zaimplementować, wytrenować i przeprowadzić walidację sieci neuronowej do klasyfikacji ręcznie pisanych cyfr. Zbiór danych MNIST do użycia: http://yann.lecun.com/exdb/mnist/. Zbiór ten dostępny jest też w ramach sklearn - https://scikit-learn.org/0.19/datasets/mldata.html.

Importowanie danych

Dane MNIST przygotowujemy za pomocą bibliotek joblib, sklearn i keras. Tworzymy folder tmp, w którym przechowujemy pobrane dane. Dzielimy je na listę x odpowiadającą za piksele obrazków oraz listę y odpowiadającą za ich opisy. Następnie rozdzielamy je odpowiednich proporcjach na dane trenujące i testujące, które później wykorzystujemy w sieci. Zazwyczaj wykorzystywaliśmy standardowy podział 6:1, ale zaimplementowana przez nas funkcja pozwala na uzyskanie dowolnych proporcji.

GŁÓWNA KLASA – SIEĆ NEURONOWA

Sieć neuronowa powstaje z warstw. Każda z nich zawiera konkretną liczbę neuronów. Jako parametr w konstruktorze klasy należy podać listę zawierającą liczby neuronów warstw ukrytych. Ta lista definiuje tym samym liczbę tworzonych takich warstw. Warstwa wyjściowa jest tworzona domyślnie. Zawiera ona 10 neuronów, ponieważ nasza sieć ma funkcjonować jako klasyfikator rozpoznający cyfry, których jest 10. Wyjście każdego z neuronów odpowiada za wynik dla innej cyfry. Domyślnie ustawiamy również liczbę wejść neuronów pierwszej warstwy ukrytej. Wynosi ona 784, ponieważ każdy obrazek zawiera 784 piksele, a one będą stanowiły wejścia pierwszej warstwy.

Inicjalizacja wag i stałych wejść neuronów

Macierz wag sieci generujemy za pomocą biblioteki random. Każda z wartości losowana jest z przedziału (-1,1). Macierz stałych wejść neuronów jest natomiast wypełniona początkowo zerami. Wielkość macierzy i listy uzależnione są od liczby neuronów w sieci. Dla każdego

neuronu generujemy listę wag o długości odpowiadającej liczbie jego wejść oraz jedną wartość stałą.

Funkcje pomocnicze

Zaimplementowaliśmy na początku funkcje potrzebne później do prawidłowego działania algorytmu. Jest to funkcja sigmoidalna, funkcja obliczająca pochodną sigmoidy oraz funkcja kosztu.

Propagacja i funkcja aktywacji

Propagacja w sieci przeprowadzana jest jednorazowo dla jednego obrazka, czyli jednej listy pikseli. Mnożymy macierze zawierające piksele oraz wagi i dodajemy do tego stałe wejścia neuronów. Otrzymane wartości poddajemy funkcji aktywacji. Do tego celu wykorzystaliśmy sigmoidę. Lista złożona z wyników obliczeń jest efektem działania propagacji. Ma ona długość równą liczbie neuronów powiększoną o 1. Pierwszym elementem pozostaje propagowany obrazek.

Propagacja wsteczna

Stanowi podstawę do optymalizacji wag i wejść stałych neuronów sieci. Jako parametry przyjmuje listę wyznaczającą spodziewaną liczbę, efekt propagacji i współczynnik uczenia. Pobiera obrazek z pierwszej pozycji listy propagacji. Następnie lista jest odwracana, a ostatni element (obrazek) usuwany, aby łatwo iterować po liście od końca. Inicjalizujemy dwie listy wypełnione zerami dla gradientów: dla wag i wejść stałych neuronów. Te listy również odwracamy. Potrzebną listą jest również lista różnic między wyjściami ostatniej warstwy sieci, a wartością oczekiwaną. Wyjścia ostatniej warstwy pobieramy z pierwszej pozycji odwróconej listy propagacji, oczekiwane wartości są podane jako parametr w postaci listy zer z jedną jedynką na pozycji odpowiadającej oczekiwanej cyfrze. Iterując po liście propagacji uaktualniamy listy gradientów poprzez odjęcie macierzy wymnożonych przez współczynnik nauki. Ponownie odwrócone listy stanowią wyjście funkcji propagacji wstecznej.
Zaimplementowaliśmy dwie metody rozwijające sieć: metodę zwykłego spadku gradientu oraz stochastycznego spadku gradientu.

ZWYKŁY SPADEK GRADIENTU

Spadek gradientu - trenowanie

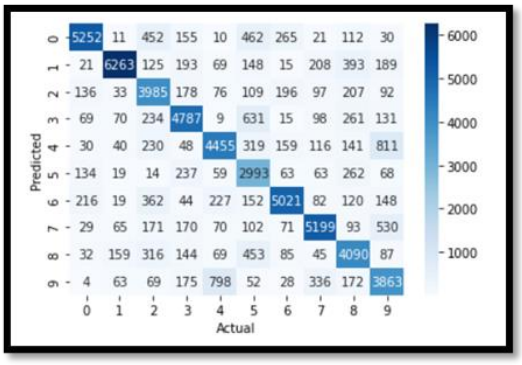
Do rzeczywistego uaktualniania wag i wejść stałych neuronów wykorzystaliśmy metodę prostego spadku gradientu. Na podstawie podanych danych treningowych, współczynnika uczenia oraz liczby epok „uczy” sieć, czyli wykonuje optymalizację parametrów sieci. Po nauczeniu sieć testuje się na zbiorach testowych. Listy przewidywanych wartości oraz wartości precyzji obliczane w międzyczasie dla różnych epok wraz z listą oczekiwanych wartości są zwracane przez funkcję.

Macierze pomyłek

Na podstawie listy przewidzianych cyfr i listy oczekiwanych cyfr generujemy macierz pomyłek. Wykorzystaliśmy do tego bibliotekę pandas, seaborn oraz matplotlib. Pierwsza stworzona przez nas funkcja tworzyła tabelę pandas i wypisywała ją w konsoli. Aby poprawić czytelność macierzy przerobiliśmy tą funkcję, żeby tworzyła macierz wyświetlaną jako wykres.

	0	1	2	3	4	5	6	7	8	9
0	845	0	14	5	0	19	10	1	1	2
1	0	1118	113	64	8	27	13	41	85	7
2	24	2	689	33	1	2	17	18	4	4
3	62	1	85	637	0	206	10	5	264	23
4	5	2	3	15	697	27	8	16	59	170
5	2	0	0	53	0	233	0	2	5	2
6	39	1	90	47	86	98	853	4	73	12
7	2	0	7	16	0	15	0	734	3	147
8	1	8	22	110	2	257	47	2	422	4
9	0	3	9	30	188	8	0	205	58	638

Rysunek 1. Pierwotna macierz pomyłek



Rysunek 2. Macierz pomyłek z wykorzystaniem matplotlib

Analiza zwykłego spadku gradientu

Podzieliliśmy dane MNIST standardowo na zestaw trenujący zawierający 60 000 elementów oraz zestaw testujący zawierający 10 000 elementów. Analizę działania sieci uczącej się za pomocą metody zwykłego spadku gradientu przeprowadzimliśmy ze względu na trzy podstawowe parametry: liczbę warstw ukrytych, współczynnik *learnig_rate* oraz liczbę epok.

I. Liczba warstw ukrytych

hidden layers	learning rate	neurons in hidden layers	epochs	accuracy
3	0.3	100	2	0.89
3	0.3	100	4	0.89
3	0.3	100	5	0.89
3	0.3	100	1	0.88
3	0.3	100	3	0.88
3	0.3	16	5	0.82
3	0.3	16	2	0.77
3	0.3	16	3	0.77
3	0.3	16	4	0.77
3	0.3	10	4	0.75
3	0.3	10	5	0.74
3	0.3	10	2	0.69
3	0.3	10	3	0.69
3	0.3	16	1	0.69
3	0.3	10	1	0.68

Tabela 1. Wynik działania dla 3 ukrytych warstw

hidden layers	learning rate	neurons in hidden layers	epochs	accuracy
1	0.3	100	3	0.95
1	0.3	100	4	0.95
1	0.3	100	5	0.95
1	0.3	100	2	0.94
1	0.3	16	5	0.91
1	0.3	100	1	0.91
1	0.3	16	3	0.9
1	0.3	10	3	0.89
1	0.3	10	5	0.89
1	0.3	16	2	0.89
1	0.3	16	4	0.89
1	0.3	10	4	0.88
1	0.3	10	2	0.87
1	0.3	16	1	0.87
1	0.3	10	1	0.86

Tabela 2. Wynik działania dla 1 ukrytej warstwy

hidden layers	learning rate	neurons in hidden layers	epochs	accuracy
0	0.3	100	2	0.86
0	0.3	100	3	0.86
0	0.3	100	4	0.85
0	0.3	10	1	0.85
0	0.3	16	4	0.85
0	0.3	100	5	0.84
0	0.3	10	2	0.84
0	0.3	10	5	0.84
0	0.3	16	1	0.84
0	0.3	16	3	0.84
0	0.3	10	3	0.83
0	0.3	10	4	0.83
0	0.3	16	2	0.83
0	0.3	16	5	0.83
0	0.3	100	1	0.82

Tabela 3. Wynik działania dla 0 ukrytych warstw

Na podstawie otrzymanych wyników jesteśmy w stanie zaobserwować, że najlepsze wyniki dostajemy przy jednej warstwie ukrytej. Może to być spowodowane tym, że brak warstw ukrytych niesie ze sobą gorsze wyuczenie sieci, przez to, że dane wejściowe są od razu mapowane na wyjście. Trzy warstwy ukryte z dużą liczbą neuronów działają jeszcze gorzej, prawdopodobnie dlatego, że taka sieć wymaga znacznie dłuższego czasu nauczania, aby poprawnie dostosować wszystkie wagi każdej z warstw.

II. *Learning rate*

hidden layers	learning rate	neurons in hidden layers	epochs	accuracy
3	0.01	100	5	0.96
3	0.01	100	3	0.95
3	0.01	100	4	0.95
3	0.01	100	2	0.94
3	0.01	16	5	0.93
3	0.01	16	3	0.92
3	0.01	16	4	0.92
3	0.01	100	1	0.92
3	0.01	10	5	0.91
3	0.01	10	4	0.9
3	0.01	16	2	0.9
3	0.01	10	3	0.89
3	0.01	10	2	0.87
3	0.01	16	1	0.87
3	0.01	10	1	0.82

Tabela 4. Wynik działania dla learning_rate = 0.01

hidden layers	learning rate	neurons in hidden layers	epochs	accuracy
3	0.1	100	5	0.96
3	0.1	100	4	0.96
3	0.1	100	3	0.96
3	0.1	100	2	0.95
3	0.1	100	1	0.93
3	0.1	16	5	0.92
3	0.1	16	4	0.91
3	0.1	16	3	0.91
3	0.1	16	2	0.91
3	0.1	10	5	0.89
3	0.1	10	3	0.88
3	0.1	10	2	0.87
3	0.1	10	1	0.87
3	0.1	16	1	0.87
3	0.1	10	4	0.86

Tabela 5. Wynik działania dla learning_rate = 0.1

hidden layers	learning rate	neurons in hidden layers	epochs	accuracy
3	0.3	100	2	0.89
3	0.3	100	4	0.89
3	0.3	100	5	0.89
3	0.3	100	1	0.88
3	0.3	100	3	0.88
3	0.3	16	5	0.82
3	0.3	16	2	0.77
3	0.3	16	3	0.77
3	0.3	16	4	0.77
3	0.3	10	4	0.75
3	0.3	10	5	0.74
3	0.3	10	2	0.69
3	0.3	10	3	0.69
3	0.3	16	1	0.69
3	0.3	10	1	0.68

Tabela 6. Wynik działania dla learning_rate = 0.3

Najbardziej zauważalne różnice w skuteczności dla neuronowej były zauważalne przy 3 warstwach ukrytych, w związku z czym na ich podstawie przeprowadziliśmy analizę. Wartość *learning_rate* nie może być zbyt mała (wtedy uczenie następuje zbyt wolno) ani zbyt duża

(wtedy uczenie następuje zbyt szybko). Najoptymalniejsze wyniki przyniosła wartość znajdująca się pomiędzy wartością zbyt duża i zbyt małą.

III. Liczba epok

hidden layers	learning rate	neurons in hidden layers	epochs	accuracy
3	0.3	100	5	0.89
3	0.3	16	5	0.82
3	0.3	10	5	0.74
3	0.3	100	4	0.89
3	0.3	16	4	0.77
3	0.3	10	4	0.75
3	0.3	100	3	0.88
3	0.3	16	3	0.77
3	0.3	10	3	0.69
3	0.3	100	2	0.89
3	0.3	16	2	0.77
3	0.3	10	2	0.69
3	0.3	100	1	0.88
3	0.3	16	1	0.69
3	0.3	10	1	0.68

Tabela 7. Wynik działania dla 3 warstw ukrytych, learning_rate = 0.3

hidden layers	learning rate	neurons in hidden layers	epochs	accuracy
2	0.3	100	5	0.94
2	0.3	10	5	0.87
2	0.3	16	5	0.87
2	0.3	100	4	0.94
2	0.3	10	4	0.87
2	0.3	16	4	0.87
2	0.3	100	3	0.93
2	0.3	16	3	0.84
2	0.3	10	3	0.82
2	0.3	100	2	0.92
2	0.3	16	2	0.87
2	0.3	10	2	0.85
2	0.3	100	1	0.91
2	0.3	10	1	0.85
2	0.3	16	1	0.82

Tabela 8. Wynik działania dla 2 warstw ukrytych, learning_rate = 0.3

hidden layers	learning rate	neurons in hidden layers	epochs	accuracy
1	0.1	100	5	0.96
1	0.1	16	5	0.91
1	0.1	10	5	0.89
1	0.1	100	4	0.96
1	0.1	16	4	0.91
1	0.1	10	4	0.9
1	0.1	100	3	0.96
1	0.1	16	3	0.91
1	0.1	10	3	0.89
1	0.1	100	2	0.96
1	0.1	16	2	0.9
1	0.1	10	2	0.89
1	0.1	100	1	0.94
1	0.1	16	1	0.9
1	0.1	10	1	0.88

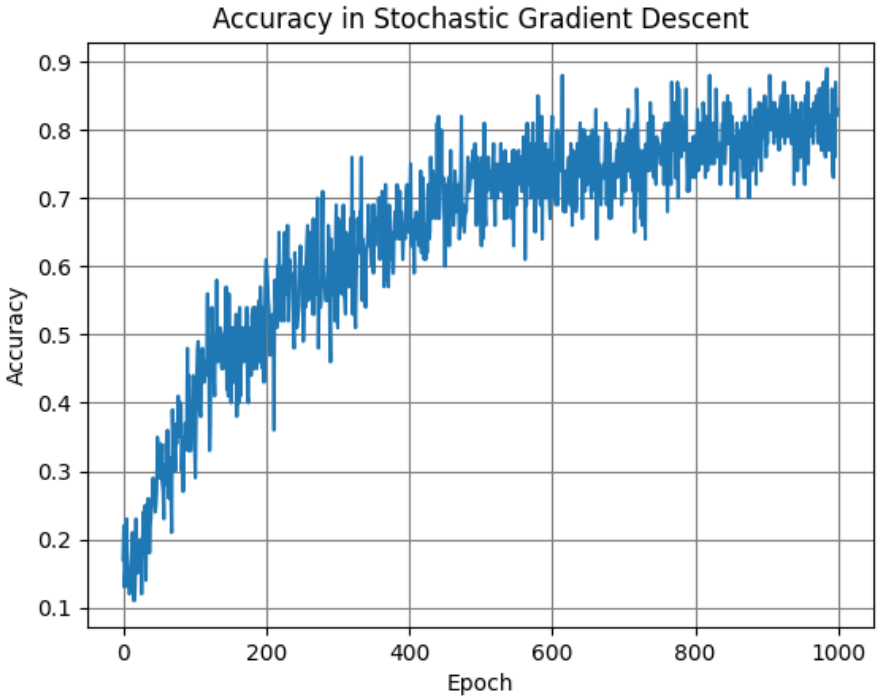
Tabela 9. Wynik działania dla 1 warstwy ukrytej, learning_rate = 0.1

Przeprowadzone testy potwierdziły nasze przypuszczenia, że skuteczność sieci rośnie wraz z liczbą epok. Ciekawą obserwacją jest skuteczność sieci dla tej samej liczby epok, ale różnej liczby neuronów w warstwie ukrytej. Na przykładzie tabeli nr 7 można zauważyć, że sieć o 100 neuronach po 1 epoce osiągnęła lepsze wyniki niż sieć o 16 neuronach po 5 epokach. W tabeli nr 8 mamy do czynienia z interesującą sytuacją. Sieć po 3 epokach radzi sobie gorzej niż po dwóch i po czterech. Może to być spowodowane niedopasowaniem parametrów sieci i niepoprawnym trenowaniem. Tabela nr 9 przedstawia natomiast działanie sieci dla dobrych parametrów. Skuteczność rośnie wraz z liczbą epok oraz liczbą neuronów w warstwach ukrytych.

Stochastyczny spadek gradientu

Stochastyczny spadek gradientu różni się od zwykłego sposobem trenowania danych. W tym przypadku jako parametry wywołania podawane są cały zbiór danych uczących, liczba epok oraz liczba danych w pojedynczym batchu. Po wywołaniu funkcji, zbiór danych jest mieszany a następnie wybierane jest z niego **n** danych służących jako zbiór uczący w danej iteracji. Zmiana wartości wag i biasów następuje dopiero po przepropagowaniu wstecznym wszystkich danych z pojedynczego batcha. W teorii takie rozwiązanie ma usprawnić uczenie sieci i zwiększyć jej skuteczność.

Przykład działania:



Rysunek 3.Przykładowy wykres skuteczności

Powyższy przykład ilustruje skuteczność wykrywania liczb w kolejnych iteracjach dla parametrów:

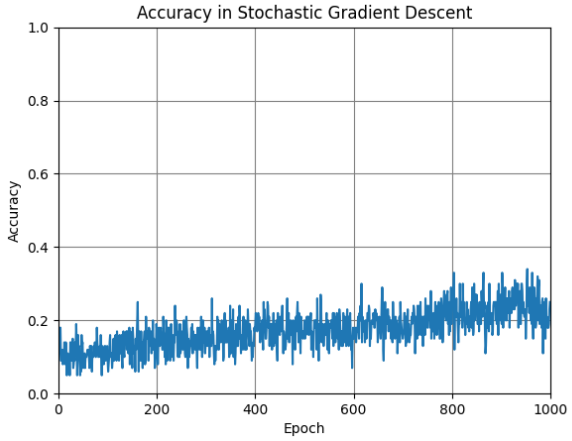
- Jedna warstwa ukryta z 10 neuronami
- Rozmiar batcha: 20
- Liczba epok: 1000

Na powyższym przykładzie widzimy, że sieć osiąga dobrą skuteczność już po 250 iteracji.

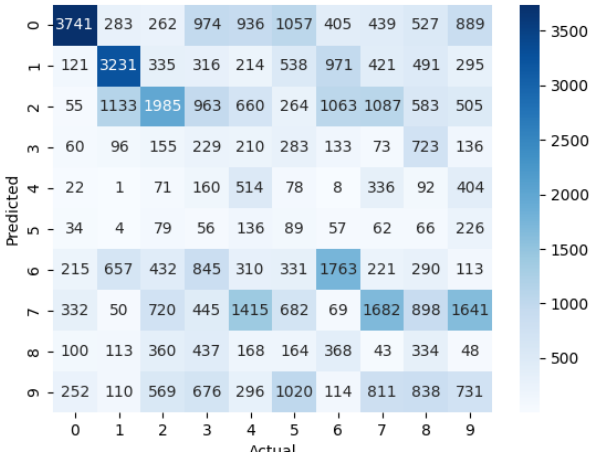
Porównanie skuteczności w zależności od rozmiaru batcha:

Testy zostaną wykonane przy ustalonej ***liczbie warstw ukrytych:*** *[10],* ***epoki*** *= 1000 i* ***learning_rate*** *= 0.01.* Skuteczność będzie następnie wyznaczana na podstawie danych testowych niezależnych od danych uczących.

I. N = 1



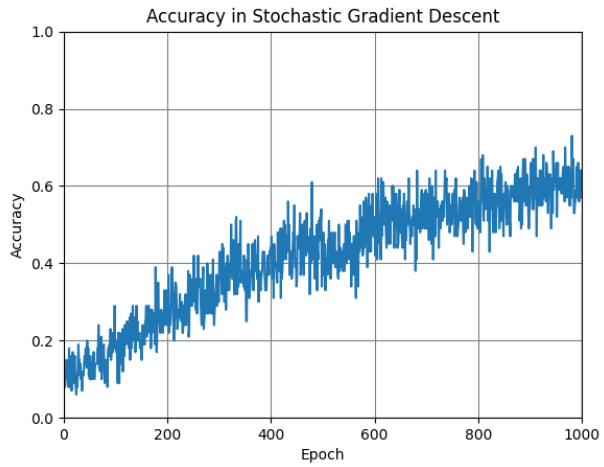
Rysunek 4. Skuteczność dla ***n*** *= 1*



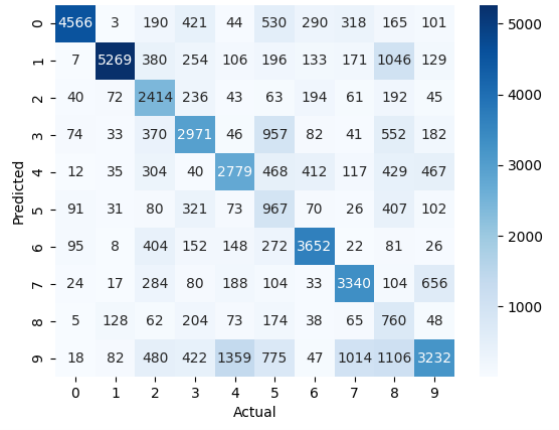
Rysunek 5. Macierz pomyłek dla ***n*** *= 1*

Jak widzimy, rozmiar batcha = 1 nie jest skutecznym rozwiązaniem. Takie trenowanie sprowadza się w głównej mierze do zwykłego spadku gradientu, dla jednej epoki i 1000 danych testowych z czego niektóre mogą się powtarzać.

II. N = 5



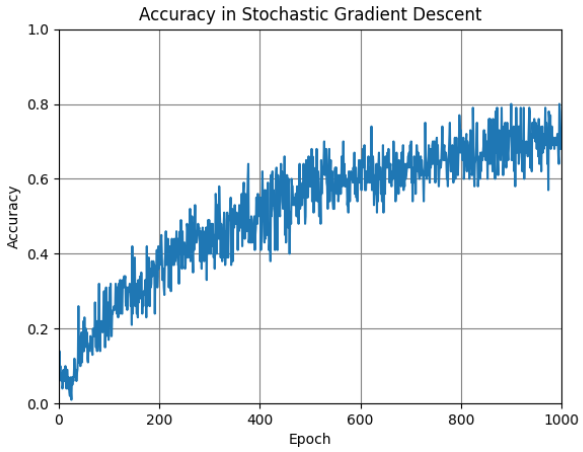
Rysunek 6. Skuteczność dla ***n*** *= 5*



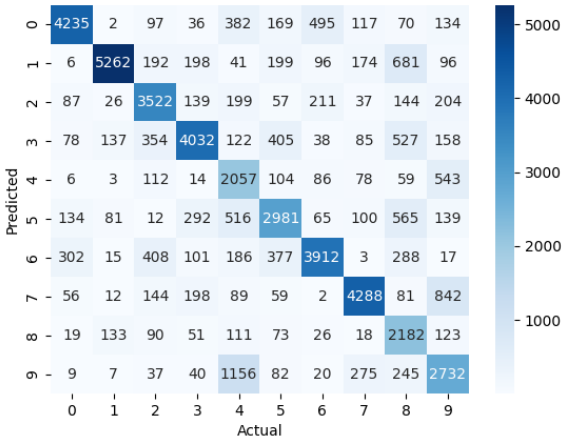
Rysunek 7. Macierz pomyłek dla ***n*** *= 5*

Dla **n** = 5 widać już znacznie lepsze wyniki niż dla poprzednio ustalonego rozmiaru. Nie są one zadowalające, ale widać znaczną poprawę co do jakości przewidywania.

III. **N = 10**



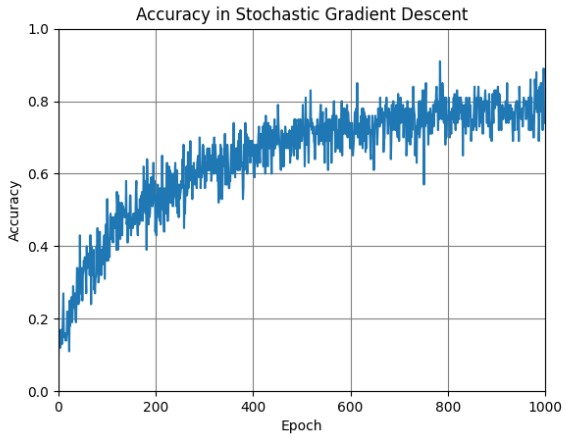
*Rysunek 8. Skuteczność dla **n** = 10*



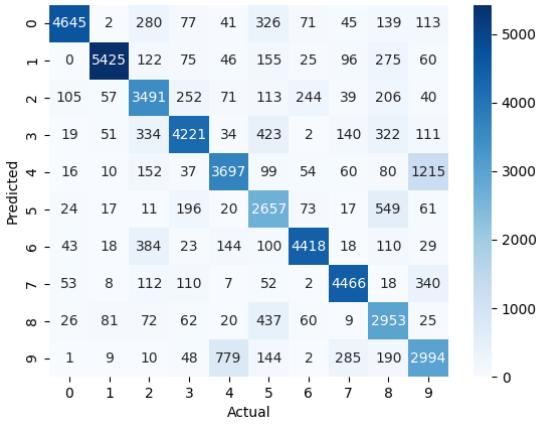
*Rysunek 9. Macierz pomyłek dla **n** = 10*

Dla **n = 10** można zaobserwować dalszą poprawę działania sieci neuronowej.

IV. **N = 20**



*Rysunek 10. Skuteczność dla **n** = 20*

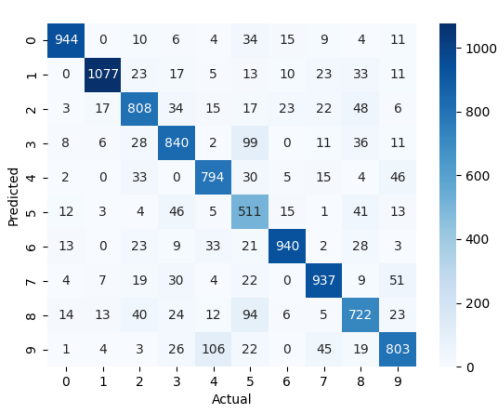


*Rysunek 11. Macierz pomyłek dla **n** = 20*

Przy **n = 20** skuteczność rzędu 80% osiągana jest już po 400 epoce. Widać tutaj, że sieć uczy się szybciej niż w przypadku, gdyby trenowana była ona za pomocą zwyczajnego spadku gradientu. Przy odpowiednio dobranej ilości iteracji i współczynnika uczenia się, można byłoby jeszcze zwiększyć skuteczność sieci.

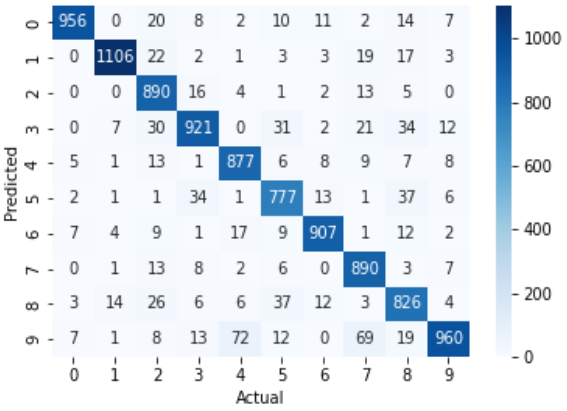
Porównanie metod SGD i GD

Porównanie zostało przeprowadzone na podstawie dwóch najlepszych wyników uzyskanych za pomocą metod SGD i GD. Testowaniu poddane zostało 10 000 danych testowych. Obie metody działały z podobną skutecznością. Widzimy, że pomyłki występowały najczęściej w tych samych miejscach.



Rysunek 12. Macierz błędów dla SGD

SGD poprawne: 8376 / 10000



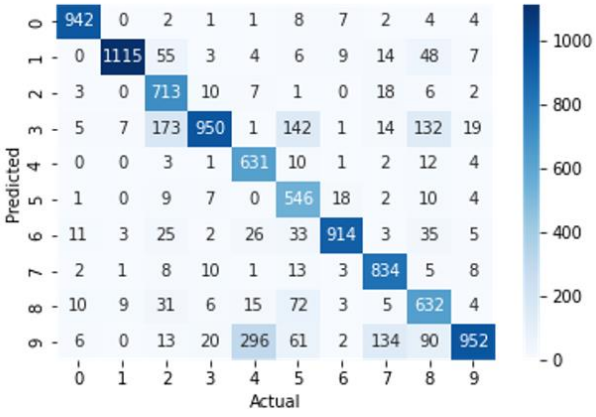
Rysunek 13. Macierz błędów dla GD

GD poprawne: 9110 / 10000

Na podstawie powyższych obserwacji można stwierdzić, że obydwie metody dają porównywalne wyniki, jednak metoda zwyczajnego spadku gradientu jest minimalnie lepsza.

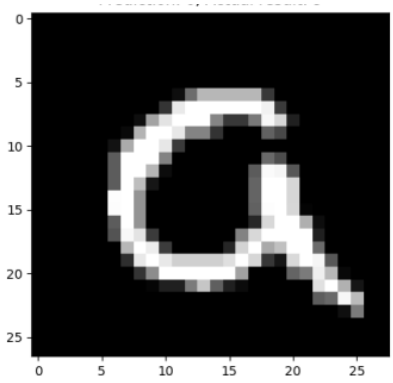
Uwagi

Na podstawie poniższej macierzy błędów możemy zaobserwować, że sieć najczęściej błędnie klasyfikowała 4 jako 9, 8 jako 3, 5 jako 3 i 2 jako 3. Jest to spowodowane podobną pisownią i kształtem podanych cyfr a w związku z rozmiarem zbioru uczącego, zdarzają się przypadki cyfr ciężko rozróżnialnych gołym okiem. Najlepiej ze wszystkich cyfr rozróżnialne były 1, a najgorzej 5

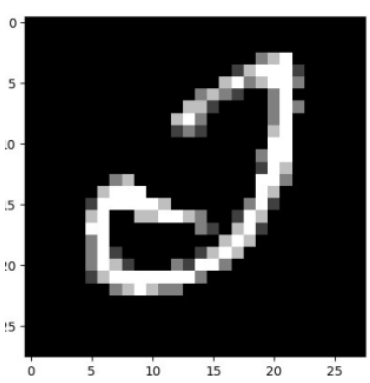


Rysunek 14. Macierz błędów dla przeciętnego GD

Przykłady błędnie zaklasyfikowanych cyfr dla sieci z 94% skutecznością:



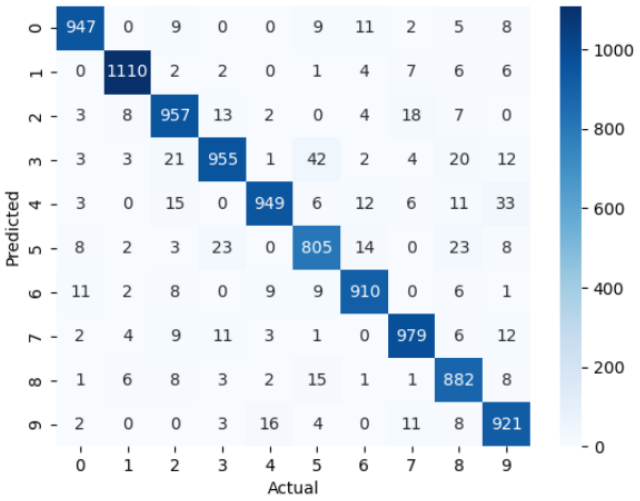
Rysunek 15. Zaklasyfikowane jako 0, w rzeczywistości 9



Rysunek 16. Zaklasyfikowane jako 6, w rzeczywistości 2

Porównanie z siecią keras

Aby sprawdzić jak dobrze poradziliśmy sobie z implementacją sieci, napisaliśmy dodatkowy kod korzystając z biblioteki keras. Przeprowadziliśmy kilka testów używając jednakowych parametrów jak podczas testowania naszej sieci. Jak się okazało, skuteczność klasyfikacji była w większości przypadków bardzo zbliżona.



Rysunek 17. Macierz wygenerowana przez sieć keras - 0.9415% skuteczności

Podział ról

Postanowiliśmy zaimplementować dwie sieci neuronowe równolegle i niezależnie od siebie. Ostateczna sieć jest wynikiem połączenia dwóch sieci, z których wybraliśmy lepsze elementy i schematy. Pierwotne sieci różniły się one m.in. funkcją aktywacji: sigmoidalną i ReLU oraz metodami wykorzystywanymi w propagacji wstecznej: jedna opierała się na metodzie stochastycznego spadku, druga na prostym spadku gradientu. Bartek połączył i poprawił otrzymane kody oraz zaimplementował funkcję wyświetlającą obrazki na ekranie. Następnie

przerobił kod tak, aby móc włączać go z linii komendy. Ola zaimplementowała funkcje generujące macierze pomyłek oraz rozpoczęła przygotowanie sprawozdania. Stworzyła również dodatkową sieć opartą na bibliotece keras. Ze względu na długi czas oczekiwania na trenowanie sieci, robiliśmy to na dwóch komputerach i wymieniliśmy się wynikami. Otrzymane wyniki i obserwacje wspólnie zamieściliśmy w sprawozdaniu.