# Sprawozdanie z projektu ze Sztucznej Inteligencji
# Porównanie metod uczenia maszynowego w problemie MNIST i jego pochodnych

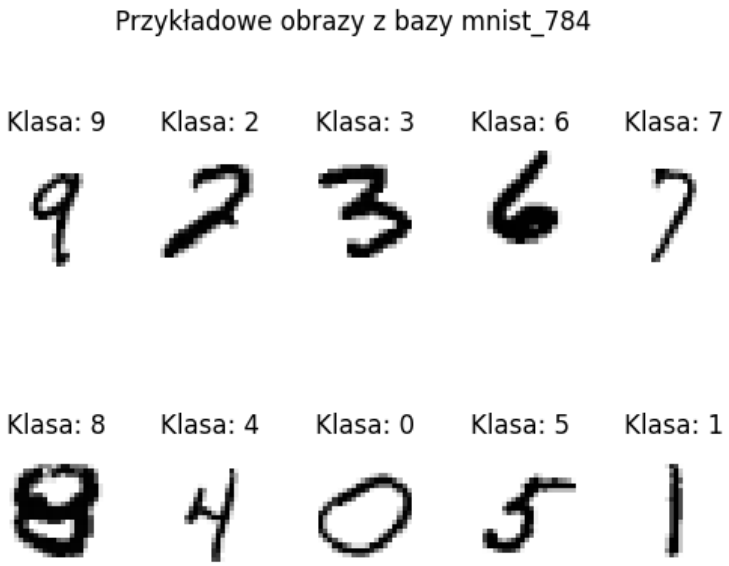
Filip Gołaś s188776
Damian Jankowski s188597
Mikołaj Storoniak s188806

1 czerwca 2023

## Spis treści

# 1 Opis problemu

Celem projektu było zbudowanie i porównanie różnych modeli uczenia maszynowego w problemie klasyfikacji obrazów z bazy MNIST. Korzystaliśmy głównie z bazy danych **MNIST-784**, która zawiera 70 tysięcy obrazów cyfr napisanych ręcznie. Każdy obraz jest w skali szarości i ma rozmiar 28x28 pikseli. Każdy piksel jest reprezentowany przez liczbę całkowitą z zakresu od 0 do 255, która określa jasność piksela. Dodatkowo każdy obraz ma przypisaną etykietę, która określa jaką cyfrę przedstawia obraz.



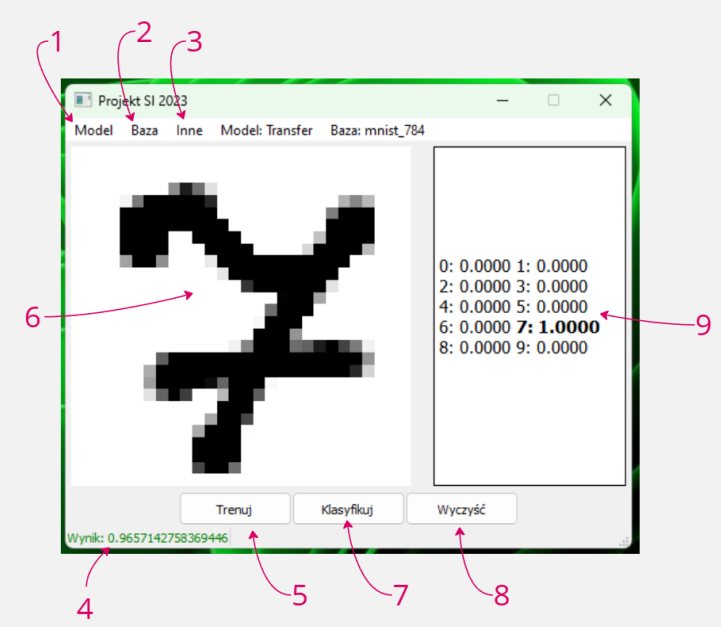

Zdecydowaliśmy się również na wykorzystanie bazy danych **Fashion-MNIST**, która podobnie jak poprzednia, zawiera 70 tysięcy obrazów o rozmiarze 28x28 pikseli, natomiast każdy obraz przedstawia wybrane ubranie lub akcesorium.

Opis klas w bazie danych Fashion-MNIST:

0. T-shirt/top
1. Trouser
2. Pullover
3. Dress
4. Coat
5. Sandal
6. Shirt
7. Sneaker
8. Bag
9. Ankle boot

# 2 Opis aplikacji

W ramach projektu stworzono aplikację służącą do testowania różnych modeli nauczania maszynowego dla problemu MNIST i jego pochodnych. Aplikacja umożliwia pobranie dowolnej bazy danych problemu klasyfikacji obrazu podobnej do MNIST ze strony OpenML. Następnie użytkownik może wybrać jeden z zaimplementowanych modeli, wytrenować go oraz przeprowadzić szereg testów. Przede wszystkim jednak aplikacja umożliwia ręczne rysowanie obrazu do klasyfikacji, lub wczytanie i modyfikację już istniejącego w bazie obrazu.




1. Model - zawiera opcje utworzenia nowego modelu spośród zaimplementowanych, zapisanie już wytrenowanego modelu oraz wczytanie modelu z pliku.
2. Baza - zawiera opcje pobrania bazy danych z serwisu OpenML i wczytania pobranej bazy z pliku.
3. Inne - zawiera opcje:
   - Wczytania losowego obrazu wybranej klasy z załadowanej bazy
   - Walidacji modelu na zbiorze testowym załadowanej bazy
   - Kroswalidacji modelu dla wybranej ilości podziałów
   - Generowanie macierzy konfuzji
   - Opcję generowania wykresów z przebiegu procesu trenowania, wykresów z wynikami walidacji oraz wykresów z macierzy konfuzji
   - Opcję włączenia przetwarzania obrazów tak by były kompatybilne z bazą MNIST-784. W przypadku modyfikowania obrazów wczytanych z bazy lub korzystania z bazy innej niż MNIST-784 należy odznaczyć tę opcję.

4. Pasek stanu - wyświetla informacje o stanie aplikacji oraz komunikaty, w tym informacje o błędach i niektóre wyniki walidacji
5. Trenuj - przeprowadza proces uczenia modelu na załadowanej bazie danych. Gdy jest to możliwe generuje i pokazuje wykresy z przebiegu procesu trenowania
6. Kanwa - na niej można dokonywać zmian obrazu, który dostanie model do klasyfikacji
7. Klasyfikuj - wciśnięcie tego przycisku rozpocznie proces walidacji obrazu widocznego na kanwie. Obraz będzie poddany przetwarzaniu, jeżeli ta opcja z menu Inne jest wybrana.
8. Wyczyść - czyści kanwę
9. Wyniki klasyfikacji - wyświetla wyniki klasyfikacji obrazu do poszczególnych klas.

Proces przetwarzania obrazów z kanwy jest niezwykle ważny w przypadku odręcznego rysowania cyfr dla modelu uczonego na bazie MNIST-784. Obrazy obecne w tej bazie muszą spełniać pewne założenia, dlatego model nie jest w stanie rozpoznawać cyfr, które drastycznie różnią się od tych obecnych w bazie. W celu przetwarzania obrazów z kanwy stosuje się:

- Skalowanie - cyfra w bazie mnist musi mieć rozmiar $20 \times 20$ pikseli mimo, że obrazy w bazie mają rozmiar $28 \times 28$ pikseli. W tym celu do rozmiaru $20 \times 20$ pikseli skalowany jest prostokąt zawierający niezerowe piksele obrazu a następnie reszta uzupełniana jest zerami.
- Środkowanie - obraz przesuwany jest tak, by jego środek masy znajdował się w środku obrazu $28 \times 28$ pikseli

# 3 Opis porównanych modeli

## 3.1 Drzewa

### 3.1.1 Drzewo decyzyjne

Drzewo decyzyjne jest modelem predykcyjnym wykorzystywanym w dziedzinie uczenia maszynowego i analizy danych. Jest to struktura drzewiasta, w której każdy węzeł reprezentuje test na jednej z cech, gałęzie reprezentują możliwe wyniki tego testu, a liście reprezentują etykiety lub wartości predykcyjne. Drzewo decyzyjne może być wykorzystane zarówno do klasyfikacji, jak i do regresji.

Podczas konstrukcji drzewa decyzyjnego, algorytm dokonuje podziału zbioru danych na podzbiory na podstawie wybranych cech. Celem jest jak najlepsze rozdzielenie danych, aby w każdym podzbiorze dominowała jedna klasa lub aby zminimalizować błąd predykcji dla zmiennych ciągłych w przypadku regresji.

Drzewa decyzyjne posiadają wiele zalet, takich jak prostota interpretacji, zdolność do obsługi zarówno danych kategorycznych, jak i numerycznych, oraz efektywność obliczeniowa w przypadku dużych zbiorów danych. Jednakże, mogą być podatne na przetrenowanie, co oznacza, że mogą zbyt dobrze dopasować się do danych treningowych i słabo generalizować na nowe dane.

Ważną czynnością podczas tworzenia drzewa decyzyjnego jest wybór tzw. hiperparametrów. Są to parametry, które nie są uczone przez model, a jedynie wpływają na jego działanie.

W przypadku drzew decyzyjnych najważniejszymi hiperparametrami są:

- **max_depth**: Określa maksymalną głębokość drzewa decyzyjnego. Głębokość drzewa to liczba poziomów w drzewie, które składają się z węzłów decyzyjnych i liści. Im większa wartość max_depth, tym bardziej skomplikowane drzewo może zostać utworzone, co może prowadzić do bardziej dopasowanego modelu. Jednak zbyt duża wartość max_depth może prowadzić do przeuczenia (overfittingu) modelu.
- **max_features**: Określa maksymalną liczbę cech, które należy wziąć pod uwagę przy każdym podziale węzła. Do wyboru są trzy opcje:
  - **None**: max_features = n_features
  - **sqrt**: max_features = $\sqrt{n\_features}$
  - **log2**: max_features = $\log_2 n\_features$

- **min_samples_split**: Określa minimalną liczbę próbek wymaganą do podziału węzła decyzyjnego. Jeśli liczba próbek w węźle jest mniejsza niż min_samples_split, to węzeł nie będzie podlegał dalszemu podziałowi, co prowadzi do utworzenia liścia. Niskie wartości min_samples_split mogą prowadzić do przeuczenia modelu, podczas gdy wysokie wartości mogą prowadzić do niedouczenia (underfittingu).
- **criterion**: Określa funkcję używaną do pomiaru jakości podziału. Istnieją dwie do wyboru:
  - **gini**: Współczynnik Giniego to miara nieczystości węzła, wyrażona jako suma prawdopodobieństw kwadratu prawdopodobieństwa każdej klasy.

$$G = 1 - \sum_{i=1}^{J} p_i^2 \tag{1}$$

    gdzie:
    * $J$ - liczba klas
    * $p_i$ - prawdopodobieństwo wystąpienia klasy $i$

    Im niższa wartość współczynnika Giniego, tym lepszy podział.
  - **entropy**: Entropia wyrażona równaniem:

$$E = -\sum_{i=1}^{J} p_i \log_2 p_i \tag{2}$$

    Podobnie jak w przypadku współczynnika Giniego, im niższa wartość entropii, tym lepszy podział.
- **splitter**: Określa strategię wyboru podziału węzła. Do wylosowania jest jedna z dwóch opcji:
  - **best**: Wybiera najlepszy podział.
  - **random**: Wybiera najlepszy losowy podział.

### 3.1.2 Las losowy

Las losowy (ang. Random Forest) jest złożonym modelem predykcyjnym, który opiera się na kombinacji wielu drzew decyzyjnych. Polega na budowie wielu drzew decyzyjnych na podstawie różnych losowych podzbiorów danych treningowych, a następnie łączeniu ich wyników w celu uzyskania ostatecznej predykcji. Każde drzewo w lesie losowym jest budowane niezależnie od pozostałych, a wyniki są łączone w procesie głosowania lub uśredniania.

Las losowy ma wiele zalet, w tym wysoką dokładność predykcji, zdolność do obsługi zarówno danych kategorycznych, jak i numerycznych, oraz odporność na przetrenowanie. Dodatkowo, las losowy może dostarczać ważność cech, co oznacza, że można ocenić, które cechy mają największy wpływ na predykcje.

Las losowy znajduje zastosowanie w wielu dziedzinach, takich jak klasyfikacja obrazów, przetwarzanie języka naturalnego, analiza danych medycznych itp. Jest to popularny model ze względu na swoją elastyczność i dobrą wydajność nawet w przypadku dużych zbiorów danych.

Podobnie jak w przypadku drzew decyzyjnych, w celu zainicjalizowania modelu należy podać wartości parametrów. W przypadku lasu losowego są to:

- **n_estimators**: Określa liczbę drzew decyzyjnych w lesie losowym.
- **max_depth**: Określa maksymalną głębokość drzewa decyzyjnego.
- **max_features**: Określa maksymalną liczbę cech, które należy wziąć pod uwagę przy każdym podziale węzła.
- **min_samples_split**: Określa minimalną liczbę próbek wymaganą do podziału węzła decyzyjnego.
- **min_samples_leaf**: Określa minimalną liczbę próbek wymaganą do utworzenia liścia.
- **criterion**: Określa funkcję używaną do pomiaru jakości podziału.

## 3.2 Sieci neuronowe

Sieć neuronowa jest modelem statystycznym czerpiącym inspirację z natury i działania układów nerwowych żywych organizmów. Modele te są stosowane zarówno do problemów regresji (aproksymacji) jak i klasyfikacji poprzez nauczanie nadzorowane.

Podstawową składową każdej sieci neuronowej jest neuron, który może być reprezentowany jako funkcja wielu parametrów zwracająca dyskretną wartość prawda lub fałsz. Choć jeden neuron nie jest w stanie opisać złożonych modeli, dużo większe możliwości mają sieci neuronowe zbudowane z wielu neuronów połączonych w warstwy w taki sposób, że wynik jednej warstwy służy jako parametry kolejnej.

Poprzez odpowiednie ustawienie współczynników funkcji w neuronach jesteśmy w stanie zamodelować zależności dużo bardziej złożone niż te możliwe do opisania jedną prostą funkcją wielu parametrów.

### 3.2.1 Wielowarstwowy perceptron

W praktyce w sieciach neuronowych neurony są zastąpione perceptronami, które od neuronów różnią się tym, że są funkcjami w dziedzinie rzeczywistej. Same warstwy neuronów również nie są reprezentowane jako zbiory obiektów typu neuron. Operacja przechodzenia danych wejściowych przez kolejne warstwy sieci zwana propagacją w przód może być zrealizowana w każdej warstwie poprzez proste mnożenie macierzy

$$S = X \times W \tag{3}$$

gdzie:

- $X$ jest macierzą $(n, 1)$ parametrów będących danymi wejściowymi sieci w przypadku warstwy pierwszej (wejściowej), lub wartości zwróconych przez poprzednią warstwę w sieci
- $W$ jest macierzą $(m, n)$ współczynników funkcji opisujących perceptrony, w której każdy wiersz opisuje jeden perceptron, a każda kolumna odpowiada wartości współczynnika tego perceptronu dla danego parametru z wektora wejściowego warstwy
- $S$ jest macierzą wynikową warstwy $(1, m)$ zawierającą wartości zwrócone przez funkcje opisujące perceptrony tworzące tę warstwę

Następnie należy zdecydować kiedy uznamy dany neuron za pobudzony. Do tego zadania stosuje się funkcje aktywacji, których dobór jest niezwykle ważny i może łatwo zadecydować o użyteczności sieci.

$$f(S) = Z \tag{4}$$

- $f(\cdot)$ jest wybraną funkcją aktywacji, która powinna być różniczkowalna by możliwe było użycie jej w propagacji wstecz
- $S$ jest macierzą wynikową $(1, m)$ warstwy
- $Z$ jest macierzą $(1, m)$, w której każdy element jest intensywnością pobudzenia danego neuronu

Aby korzystać z sieci neuronowych do opisywania skomplikowanych modeli statystycznych nie możemy ręcznie decydować o wartościach parametrów w warstwach. Byłoby to niezwykle ciężkie o ile nie niemożliwe w sposób analityczny. Zamiast tego sieci neuronowe poddaje się trenowaniu.

Proces trenowania sieci neuronowej nazywany jest propagacją wstecz. Gdy przeprowadzimy proces propagacji w przód dla sieci z wartościami współczynników o wartościach losowych o dowolnym rozkładzie jesteśmy w stanie poprawić je stosując wybraną różniczkowalną funkcję straty i znając wartości pożądane. Dzieje się to w najprostrzym przypadku poprzez metodę spadku po gradiencie, którą można opisać macierzowo dla całej warstwy neuronów jako:

$$W_1 = W_0 - \eta \times \frac{\partial E}{\partial Z} \tag{5}$$

gdzie:

- $W_0$ jest macierzą $(m, n)$ współczynników perceptronów warstwy, a $W_1$ jest jej nową postacią
- $\eta$ jest współczynnikiem $learning\_rate$ kontrolującym tempo uczenia

- $\frac{\partial E}{\partial Z}$ jest pochodną z sygnału błędu po macierzy wynikowej $S$ danej warstwy. Sygnał błędu w przypadku ostatniej warstwy (wyjściowej) jest gradientem funkcji błędu macierzy wynikowej, w przypadku reszty warstw jest on sygnałem błedu pochodzącym z poprzednio aktualizowanej w procesie propagacji wstecz warstwy wyznacznym poprzez:

$$E_1 = -\frac{\partial E_0}{\partial S} \tag{6}$$

gdzie:

  - $E_0$ jest sygnałem błędu zwracanym przez daną warstwę
  - $\frac{\partial E}{\partial S}$ jest pochodną cząstkową z sygnału błedu, który otrzymała ta warstwa od warstwy poprzedniej w procesie propagacji wstecz, lub w przypadku warstwy wyjściowej gradientem funkcji straty dla macierzy wyjściowej tej warstwy

### 3.2.2 Sieć konwolucyjna

Sieć konwolucyjna jest odmianą sieci neuronowej, w której stosuje się warstwy konwolucyjne. W takich warstwach każdy neuron zamiast nakładać na zbiór parametrów funkcję, nakłada na nie filtr, którego wagi są współczynnikami neuronu. Filtr może być w postaci wektora, małej macierzy kwadratowej, lub kilku macierzy w zależności od tego czy interpretujemy dane wejściowe jako wektor, powierzchnię czy przestrzeń punktów. Odpowiednio wytrenowana sieć konwolucyjna jest w stanie skutecznie wykrywać i wzmacniać poprzez nakładanie filtrów istotne cechy danych, które następnie mogą posłużyć jako wejście dla warstwy zwykłych perceptronów, które dzięki temu wyszczególnieniu kluczowych cech dużo skuteczniej poradzą sobie w rozwiązaniu zadania.

Poza warstwami konwolucyjnymi w sieciach konwolucyjnych stosuje się często Max Pooling. Nie jest to warstwa neuronowa, lecz procedura przetwarzająca obrazy utworzone przez warstwy konwolucyjne zmniejszająca rozmiar tworzonych przez sieć obrazów poprzez wstawianie w miejsce każdego okna $n \times n$ wartość maksymalną z tego okna.

Kolejne udoskonalenie modelu polegało na dodaniu procedury Dropout jako jednej z warstw sieci. Dropout ustawia z pewnym prawdopodobieństwem pojedyncze wartości mu przekazane na 0 w ten sposób wykluczając pewne obserwacje sieci z rezultatu i zmniejszając szansę na przeuczenie sieci, co skutkowałoby doskonałym klasyfikowaniem danych, na których model był uczony, lecz nie radzeniem sobie zupełnie z nowymi nie widzianymi przez niego danymi.

### 3.2.3 Sieć grafowa

Zastosowanie sieci neuronowej do analizy grafów wymaga przekształcenia tych grafów do postaci "zrozumiałej" przez sieć neuronową. W tym celu wykorzystana została biblioteka Spektral, stanowiąca rozszerzenie biblioteki Keras.

Spektral pozwala na zastosowania obiektu Loadera, który otrzymawszy listę grafów, przekształca ją w serię porcji danych zwanych batchami, na których można trenować model. To sprawia, że dane wejściowe stają się w pełni kompatybilne ze wszystkimi mechanizmami dostarczanymi przez bibliotekę Keras. Przekształcanie listy grafów w batche polega na dopełnieniu ich macierzy sąsiedztwa zerami, tak, aby wszystie miały identyczny rozmiar. Następnie grafy te łączy się w trójwymiarowe tensory o wymiarach $rozmiarBatcha \times maxLiczbaWierzcholkow \times liczbaCechGrafu$. Rozmiar batcha jest ustalony z góry, na poziomie kodu.

Tak, jak w przypadku pozostałych modeli w projekcie, sieć neuronowa została zbudowana przy użyciu gotowych warstw dostępnych w bibliotece Keras. Tym, co odróżnia model grafowy od pozostałych jest zastosowanie warstwy GCNConv, dostosowanej do działania na danych w postaci grafów. Warstwa ta działa analogicznie do warstwy konwolucyjnej w "tradycyjnych" neuronowych sieciach konwolucyjnych - to znaczy sprawia, że każdy punkt danych zostaje niejako wzbogacony o informacje na temat kontekstu w którym wystąpił - czyli sąsiadujących punktów danych. W tym przypadku "punktami danych" są wierzchołki grafu, a cały proces odbywa się na zasadzie przekazywania wiadomości między sąsiadującymi wierzchołkami.

W praktyce jest to zrealizowane przy pomocy następujących operacji macierzowych:

$$X' = \hat{D}^{-\frac{1}{2}} \hat{A} \hat{D}^{-\frac{1}{2}} XW + b \tag{7}$$

gdzie:

- $X$ - dane wejściowe
- $X'$ - dane wyjściowe
- $W$ - wektor wag modelu
- $b$ - bias
- $\hat{A}, \hat{D}$ - macierze sąsiedztwa z pętlami (A + I) i stopni wierzchołków grafu

#### 3.2.4 Konwolucyjna sieć transferowa

Z racji, że różne zadania klasyfikacji obrazów nie różnią się od siebie tak drastycznie jak mogłoby się wydawać, popularnym sposobem na tworzenie bardzo dokładnych klasyfikatorów jest korzystanie z sieci transferowych. Są to sieci konwolucyjne korzystające z wytrenowanej już wcześniej na innych problemach klasyfikacji warstw konwolucyjnych. Proces nauczania takiej sieci neuronowej jest o wiele prostszy, gdyż wytrenowania wymaga jedynie mała ilość nowych warstw, które zinterpretują wyniki juz gotowych i wytrenowanych w klasyfikacji obrazów warstw konwolucyjnych. W pierwszych etapach nauczania należy pominąć już wytrenowane warstwy konwolucyjne aby usprawnić ten proces. Gdy model jest już dostatecznie dobrze wytrenowany można odblokować trenowanie warstw konwolucyjnych, by przeprowadzić tzw. fine tuning, który pozwoli wyspecjalizować warstwy konwolucyjne w detekcji cech charakterystycznych dla danego problemu.

### 3.3 KNN - K Nearest Neighbours

KNN jest modelem bezparametrycznego uczenia nadzorowanego. Jest to niezwykle prosty model, który nie wymaga procesu uczenia, jednak przypłaca to bardzo kosztowną predykcją. Algorytm K Nearest Neighbours polega na obliczeniu odległości wektora danych wejściowych długości n traktowanego jako punkt w przestrzeni n wymiarowej i porównaniu go z każdym z pośród wektorów danych wejściowych dostarczonych modelowi w ramach danych treningowych, które to model zapamiętał. Punkty z pośród danych treningowych wraz z ich etykietami następnie są sortowane rosnąco wedle ich odległości od danej wejściowej, której klasyfikację przeprowadza model. Następnie dopierane jest K pierwszych z pośród punktów, których to odległość od klasyfikowanego punktu jest najniższa i zliczane są wystąpienia różnych typów etykiet pośród wybranych K punktów. Ta etykieta, która pośród nich powtarza się najczęściej jest odpowiedzią modelu na zadanie klasyfikacji.

## 4 Opis realizacji zadania

### 4.1 Drzewo decyzyjne

Wykorzystując bibliotekę scikit-learn zaimplementowano model drzewa decyzyjnego. W celu znalezienia najlepszych parametrów modelu losowo wybierano wartości z pewnego przedziału i sprawdzano, dla których wartości model osiąga najlepsze wyniki.

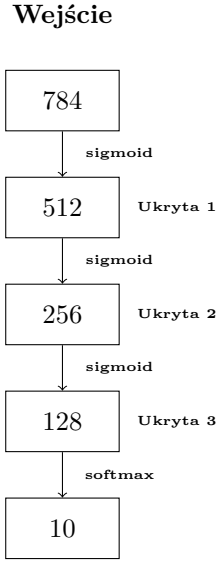
### 4.2 Las losowy

Podobnie jak w przypadku drzewa decyzyjnego, wykorzystując bibliotekę scikit-learn zaimplementowano model, losując wartości parametrów początkowych.

### 4.3 Własna implementacja Wielowarstwowego perceptronu

Z wykorzystaniem biblioteki numpy zaimplementowano model sieci neuronowej typu wielowarstwowy perceptron zgodny z powyższą teorią do bazy MNIST-784.

Architektura warstw stworzonej sieci wyglądała następująco:




Za funkcję straty przyjęto funkcję Categorical Crossentropy, learning rate wynosił 0.01. W trakcie trenowania nie korzystano żadnych mechanizmów nieomówionych w części teoretycznej w tym: batchingu czy przetwarzania danych wejściowych poza skalowaniem do zakresu $[0, 1]$.

## 4.4 Wielowarstwowy perceptron Tensorflow Keras

Z wykorzystaniem pakietu tensorflow.keras utworzono sieć neuronową o architekturze analogicznej do tej zaimplementowanej ręcznie. Zastosowano kilka zmian w postaci:

- Do propagacji wstecz zamiast stochastycznego schodzenia po gradiencie użyto algorytmu Adam będącego jego udoskonaleniem uwzględniającym momenty gradientu.
- Zastosowano mechanizm batchingu, $batch\_size = 128$

Struktura sieci

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               401920

dense_1 (Dense)              (None, 256)               131328

dense_2 (Dense)              (None, 128)               32896

dense_3 (Dense)              (None, 10)                1290

=================================================================
Total params: 567,434
Trainable params: 567,434
Non-trainable params: 0
_________________________________________________________________
```

## 4.5 Sieć konwolucyjna Tensorflow Keras

### 4.5.1 Wersja podstawowa

Sieć konwolucyjną zaimplementowano przy użyciu pakietu tensorflow keras. Wszystkie hiperparametry pozostały identyczne z wielowarstwowym perceptronem zaimplementowanym z użyciem tych samych bibliotek. Struktura sieci wyglądała następująco:

Struktura sieci

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320

max_pooling2d (MaxPooling2D)  (None, 14, 14, 32)       0

conv2d_1 (Conv2D)            (None, 14, 14, 64)        18496

max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)         0
2D)

flatten (Flatten)            (None, 3136)              0

dropout (Dropout)            (None, 3136)              0

dense (Dense)                (None, 10)                31370

=================================================================
Total params: 50,186
Trainable params: 50,186
Non-trainable params: 0
_________________________________________________________________
```

### 4.5.2 Wersja rozszerzona

Dodatkowo dodano więcej warstw konwolucyjnych oraz warstw gęstych.

Struktura sieci

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d (Conv2D)              (None, 24, 24, 32)        832

 conv2d_1 (Conv2D)            (None, 20, 20, 32)        25600

 batch_normalization (BatchN  (None, 20, 20, 32)       128
 ormalization)

 activation (Activation)      (None, 20, 20, 32)        0

 max_pooling2d (MaxPooling2D  (None, 10, 10, 32)       0
 )

 dropout (Dropout)            (None, 10, 10, 32)        0

 conv2d_2 (Conv2D)            (None, 8, 8, 64)          18496

 conv2d_3 (Conv2D)            (None, 6, 6, 64)          36864

 batch_normalization_1 (Batc  (None, 6, 6, 64)         256
 hNormalization)

 activation_1 (Activation)    (None, 6, 6, 64)          0

 max_pooling2d_1 (MaxPooling  (None, 3, 3, 64)         0
 2D)

 dropout_1 (Dropout)          (None, 3, 3, 64)          0

 flatten (Flatten)            (None, 576)               0

 dense (Dense)                (None, 256)               147456

 batch_normalization_2 (Batc  (None, 256)              1024
 hNormalization)

 activation_2 (Activation)    (None, 256)               0
```

```
 dense_1 (Dense)             (None, 128)               32768

 batch_normalization_3 (Batc  (None, 128)              512
 hNormalization)

 activation_3 (Activation)   (None, 128)               0

 dense_2 (Dense)             (None, 84)                10752

 batch_normalization_4 (Batc  (None, 84)               336
 hNormalization)

 activation_4 (Activation)   (None, 84)                0

 dropout_2 (Dropout)         (None, 84)                0

 dense_3 (Dense)             (None, 10)                850

=================================================================
Total params: 275,874
Trainable params: 274,746
Non-trainable params: 1,128
_________________________________________________________________
```

Natomiast w celu poprawy wydajności trenowania dodano następujące callbacki:

- EarlyStopping - zatrzymujący trenowanie w przypadku braku poprawy dokładności
- ReduceLROnPlateau - zmniejszający współczynnik uczenia wraz z trenowaniem

Dodatkowo w celu poprawy generalizacji modelu wprowadzono data augmentation w postaci obracania obrazów z bazy MNIST-784 o $\pm 10°$.

## 4.6 Sieć transferowa Tensorflow Keras

Sieć konwolucyjną transferową zbudowano z użyciem biblioteki tensorflow keras oraz modelu transferowego MobileNet klasyfikującego obrazy w przestrzeni RGB o rozmiarze przynajmniej $32 \times 32$px.

Z powodu rozbieżności rozmiarów obrazów, które wymaga sieć MobileNet i obrazów należących do baz danych MNIST i podobnych konieczny był preprocessing danych wejściowych. Do zastosowanych metod należały:

- skalowanie obrazów $28 \times 28$px do rozmiaru $32 \times 32$px
- zklonowanie skali szarości na kanały RGB

Dodatkowo by uzyskać jak najlepsze rezultaty na danych nie tylko należących do bazy danych, ale stworzyć model, który spełni zadanie klasyfikacji możliwie tak dobrze jak człowiek zastosowano generowanie nowych danych treningowych w oparciu o dane z bazy z wykorzystaniem:

- skalowania zawartości obrazu
- obracania obrazów
- przesuwania obrazów w osiach X i Y

### 4.6.1 Przed fine-tuningiem

Struktura sieci

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenet_1.00_224 (Functio  (None, 1, 1, 1024)       3228864
 nal)

 flatten (Flatten)           (None, 1024)              0
```

```
 dropout (Dropout)           (None, 1024)              0

 dense (Dense)               (None, 512)               524800

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 3,758,794
Trainable params: 529,930
Non-trainable params: 3,228,864
_________________________________________________________________
```

#### 4.6.2 Fine-tuning

Struktura sieci

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenet_1.00_224 (Functio  (None, 1, 1, 1024)       3228864
 nal)

 flatten (Flatten)           (None, 1024)              0

 dropout (Dropout)           (None, 1024)              0

 dense (Dense)               (None, 512)               524800

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 3,758,794
Trainable params: 3,736,906
Non-trainable params: 21,888
_________________________________________________________________
```

### 4.7 Sieć grafowa

Sieci grafowe były trenowane jedynie na zbiorze MNIST zawierającym cyfry, ze względu na fakt, że przekształcanie obrazów na grafy jest bardzo czasochłonne - przetworzenie jednego zbioru trwało kilkanaście minut.

- Każdy obraz był najpierw dzielony na superpiksele - to znaczy klastry pikseli o zbliżonej jasności. Do przypisania pikseli do klastrów wykorzystano metodę k-means.
- Następnie na podstawie listy superpikseli tworzony był graf w postaci macierzy sąsiedztwa. Każdy superpiksel stanowił jeden wierzchołek grafu. Stykające się superpiksele były połączone krawędziami.
- Każdy wierzchołek grafu (superpiksel) został opatrzony cechą - uśrednioną jasnością składających się na niego pikseli (w zakresie 0 - 255). Każda krawędź posiada wagę, będącą odległością między środkami ciężkości łączonych przez nią superpikseli.

Próby zostały przeprowadzone na kilku zbiorach grafów. Każdy zestaw powstał na bazie identycznego zestawu danych źródłowych (obrazków):

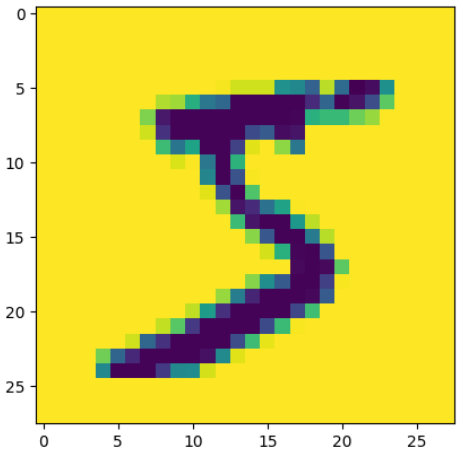
(**A**) Grafy z max. 25 wierzchołkami

(**B**) Grafy z max. 50 wierzchołkami

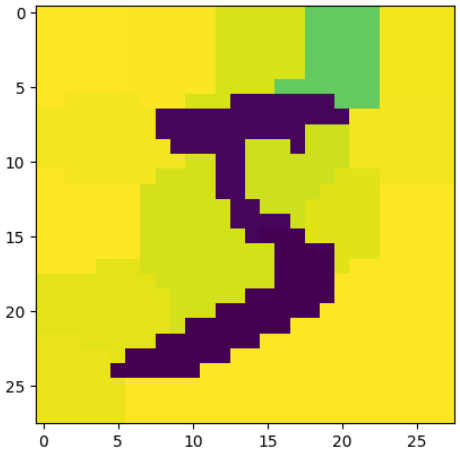
(**C**) Grafy z max. 25 wierzchołkami, bez wag na krawędziach (wszystkie wagi równe 1)

Maksymalna liczba wierzchołków była regulowana poprzez zmianę parametru algorytmu k-means - tzn. początkowej liczby klastrów do których dopasowywane są piksele.

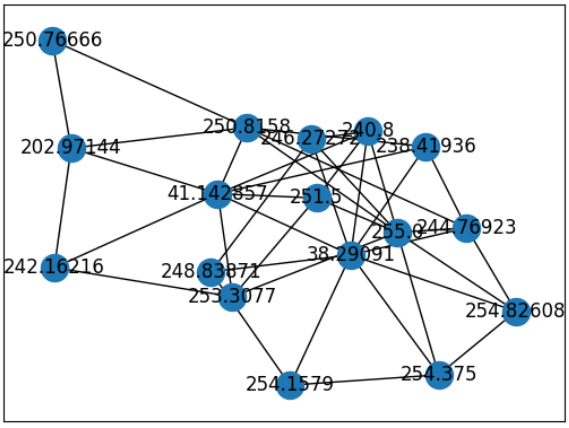
Obrazki po przetworzeniu wyglądały w sposób następujący:




Rysunek 1: Obraz nieprzetworzony




Rysunek 2: Obraz z podziałem na superpiksele




Rysunek 3: Graf powstały na bazie podzielonego obrazu.

Powyższe obrazy to wizualizacje macierzy zawierajacych jasności pikseli. Oryginalne obrazy są czarno-białe.

Sieć grafowa została zbudowana w sposób następujący:

Struktura sieci

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gcn_conv (GCNConv)           multiple                  100

gcn_conv_1 (GCNConv)         multiple                  2550

global_sum_pool (GlobalSumP  multiple                  0
ool)

dense (Dense)                multiple                  26112

dense_1 (Dense)              multiple                  5130

=================================================================
Total params: 33,892
Trainable params: 33,892
Non-trainable params: 0
_________________________________________________________________
```

### 4.8 K Nearest Neighbours

Model KNN został zaimplementowany z użyciem biblioteki numpy. Za parametr K wybrano wielokrotność liczności zbioru etykiet, 20.

## 5 Wyniki

### 5.1 Drzewo decyzyjne

Przykładowo dla następujących parametrów dla bazy MNIST-784:

- max_depth: 19
- max_features: None
- min_samples_split: 5
- min_samples_leaf: 4

- criterion: entropy
- splitter: random

model osiągał skuteczność na danych testowych wynoszącą 0.873.

## 5.2 Las losowy

Dla następujących parametrów dla bazy MNIST-784:

- n_estimators: 16
- max_depth: 18
- max_features: sqrt
- min_samples_split: 18
- min_samples_leaf: 1
- criterion: entropy

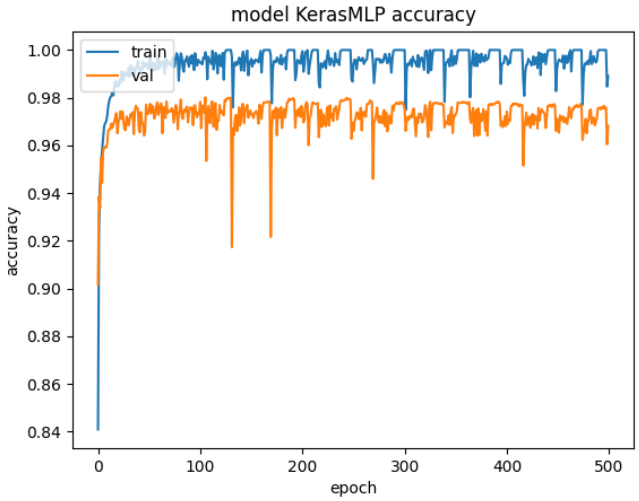
model osiągał skuteczność na danych testowych wynoszącą 0.951.

## 5.3 Własna implementacja Wielowarstwowego perceptronu

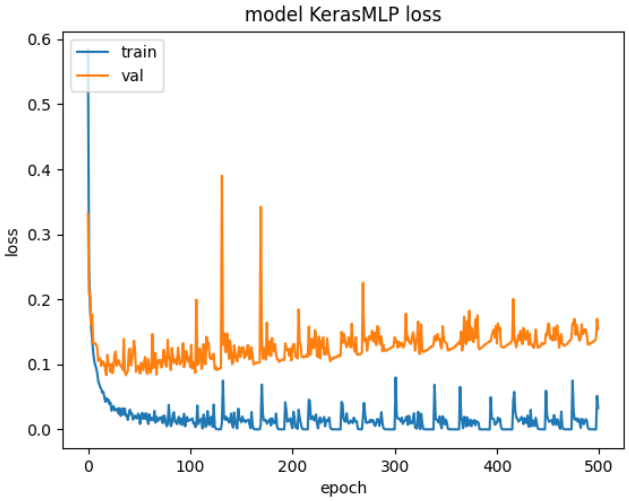
## 5.4 Wielowarstwowy perceptron Tensorflow Keras

Sieć wyuczono dla dwóch zbiorów danych: MNIST-784 i Fashion-MNIST. W obydwu przypadkach model był trenowany przez 500 epoch'ów. Proces i rezultaty widoczne są na poniższych wykresach:

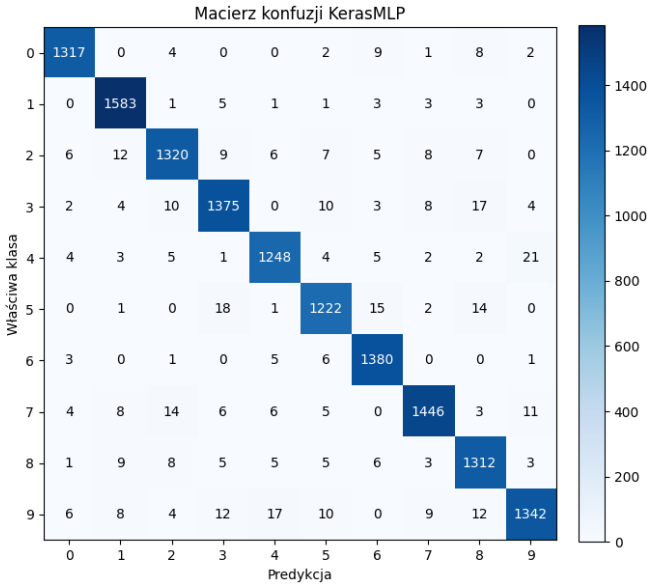
### 5.4.1 MNIST-784




Rysunek 4: Wykres dokładności modelu Keras MLP dla bazy MNIST-784 w zależności od epoki



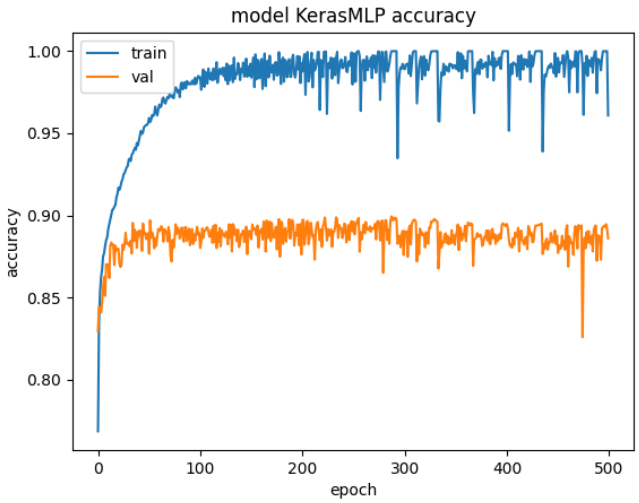

Rysunek 5: Wykres wartości funkcji straty modelu Keras MLP dla bazy MNIST-784 w zależności od epoki



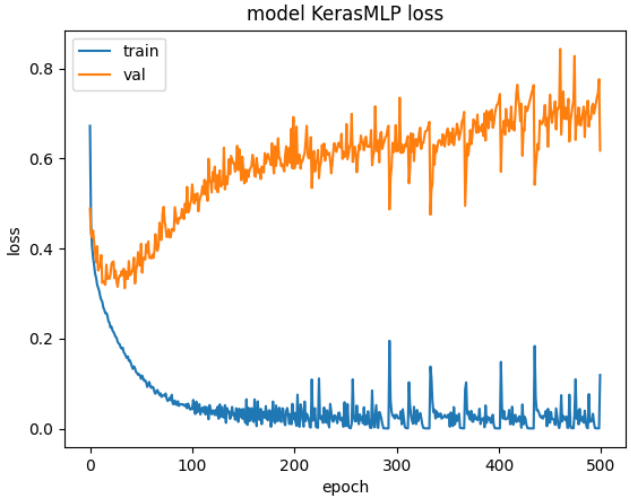

Rysunek 6: Macierz konfuzji modelu Keras MLP dla bazy MNIST-784

Dla bazy MNIST-784 model Keras MLP zdołał uzyskać accuracy w wysokości 0.975

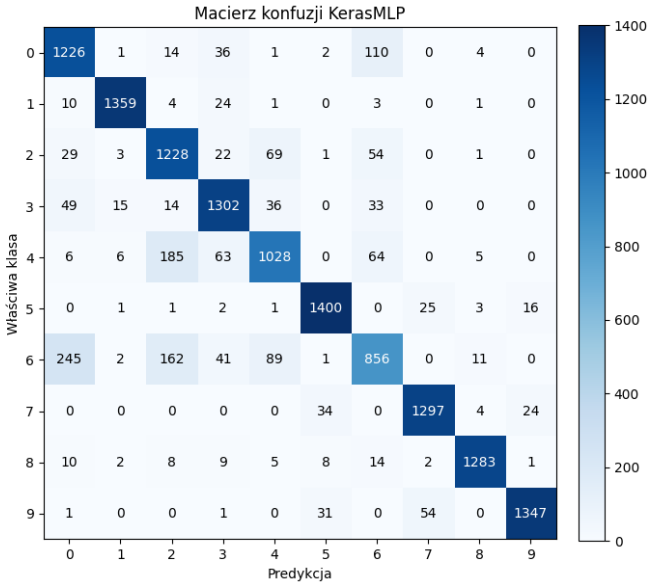
### 5.4.2 Fashion-MNIST




Rysunek 7: Wykres dokładności modelu Keras MLP dla bazy Fashion-MNIST w zależności od epoki




Rysunek 8: Wykres wartości funkcji straty modelu Keras MLP dla bazy Fashion-MNIST w zależności od epoki



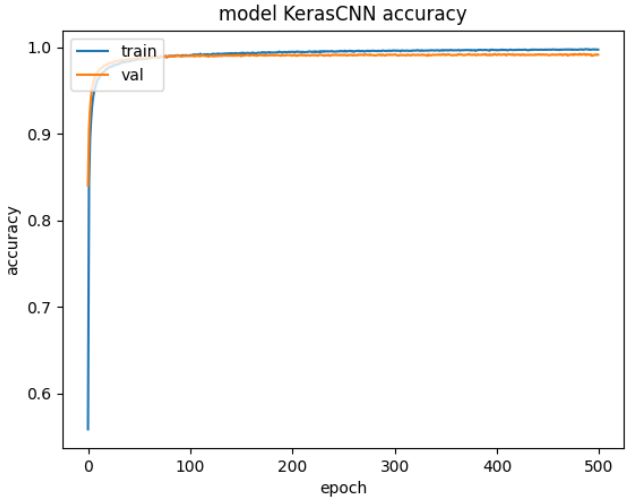

Rysunek 9: Macierz konfuzji modelu Keras MLP dla bazy Fashion-MNIST

Dla bazy Fashion-MNIST model Keras MLP zdołał uzyskać accuracy w wysokości 0.88

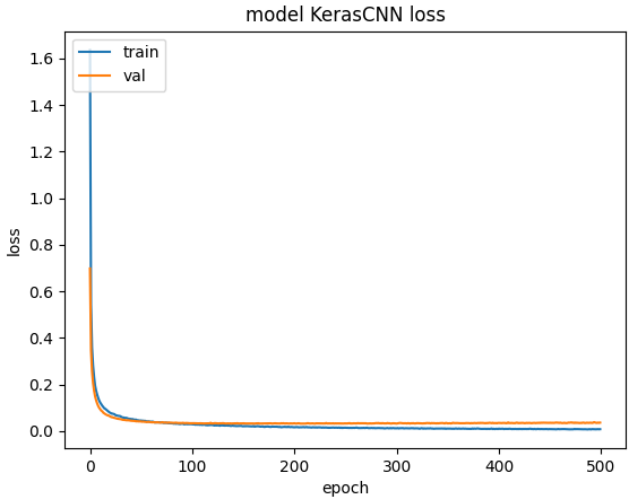
## 5.5 Sieć konwolucyjna Tensorflow Keras, wersja podstawowa

Sieć wyuczono dla dwóch zbiorów danych: MNIST-784 i Fashion-MNIST. W obydwu przypadkach model był trenowany przez 500 epoch'ów. Proces i rezultaty widoczne są na poniższych wykresach:

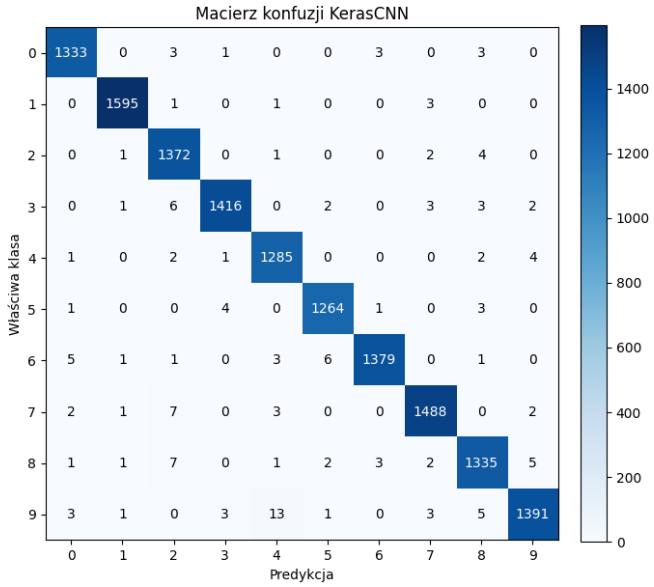
### 5.5.1 MNIST-784




Rysunek 10: Wykres dokładności modelu Keras CNN dla bazy MNIST-784 w zależności od epoki



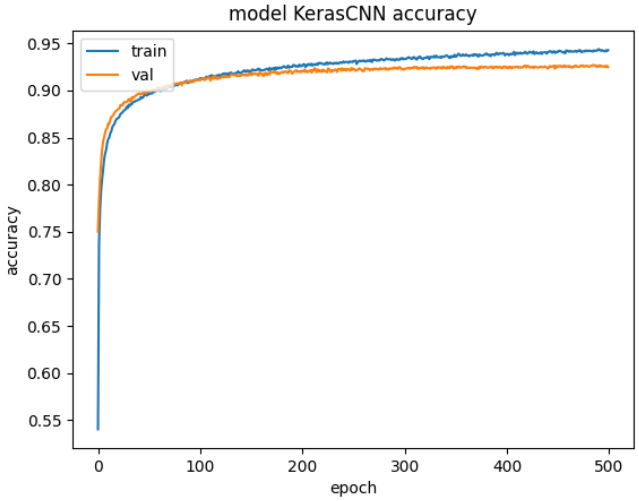

Rysunek 11: Wykres wartości funkcji straty modelu Keras CNN dla bazy MNIST-784 w zależności od epoki



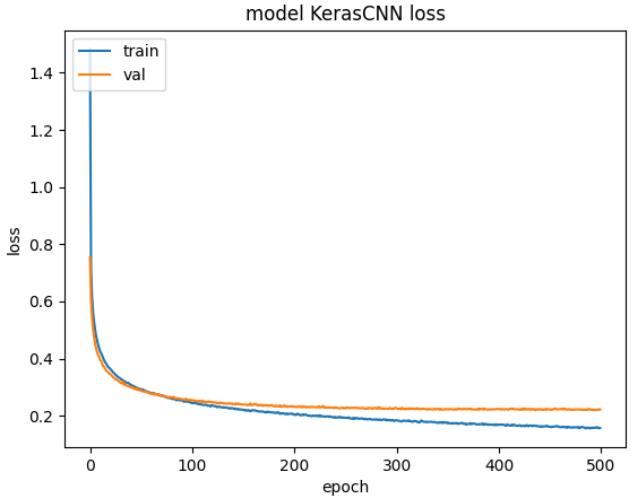

Rysunek 12: Macierz konfuzji modelu Keras CNN dla bazy MNIST-784

Dla bazy MNIST-784 model Keras CNN zdołał uzyskać accuracy w wysokości 0.9899

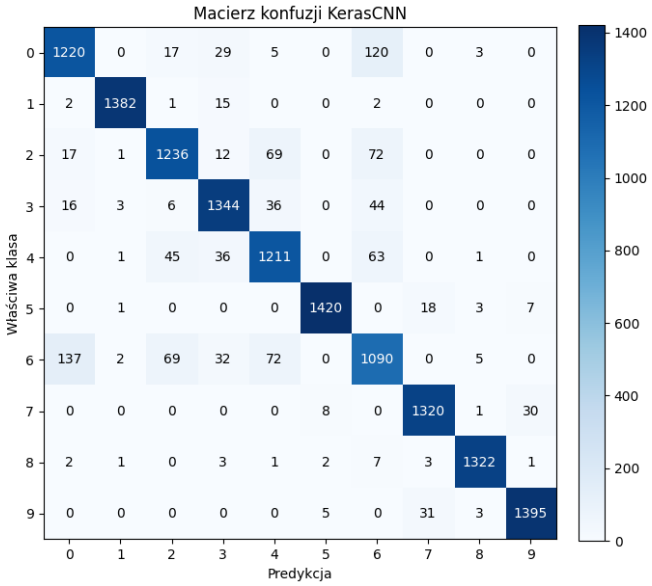
### 5.5.2 Fashion-MNIST




Rysunek 13: Wykres dokładności modelu Keras CNN dla bazy Fashion-MNIST w zależności od epoki




Rysunek 14: Wykres wartości funkcji straty modelu Keras CNN dla bazy Fashion-MNIST w zależności od epoki



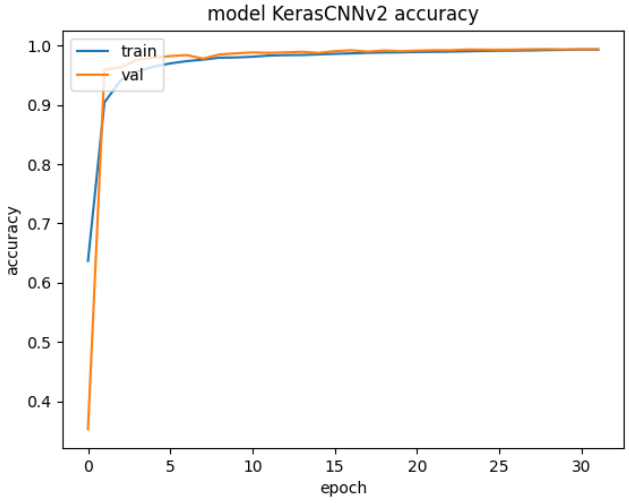

Rysunek 15: Macierz konfuzji modelu Keras CNN dla bazy Fashion-MNIST

Dla bazy Fashion-MNIST model Keras CNN zdołał uzyskać accuracy w wysokości 0.924

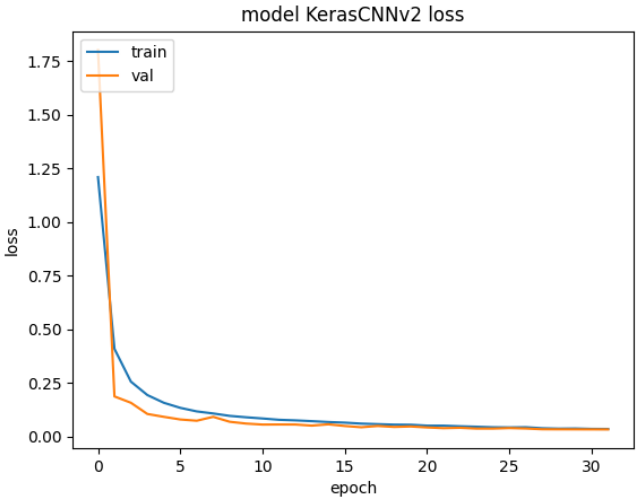
## 5.6 Sieć konwolucyjna Tensorflow Keras, wersja rozszerzona

Sieć wyuczono dla dwóch zbiorów danych: MNIST-784 i Fashion-MNIST. Dzięki zastosowaniu callbacków proces trenowania w przypadku bazy MNIST-784 trwał 32 epochy a w przypadku Fashion-MNIST 47 epochów. Proces i rezultaty widoczne są na poniższych wykresach:

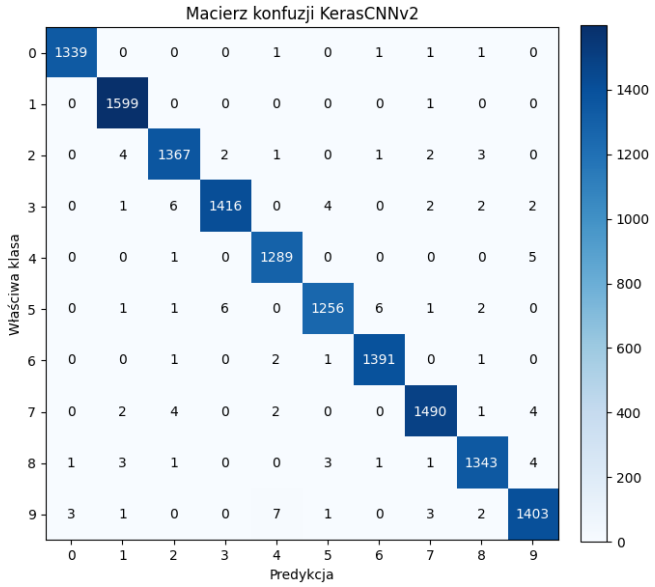
### 5.6.1 MNIST-784




Rysunek 16: Wykres dokładności modelu Keras CNN dla bazy MNIST-784 w zależności od epoki



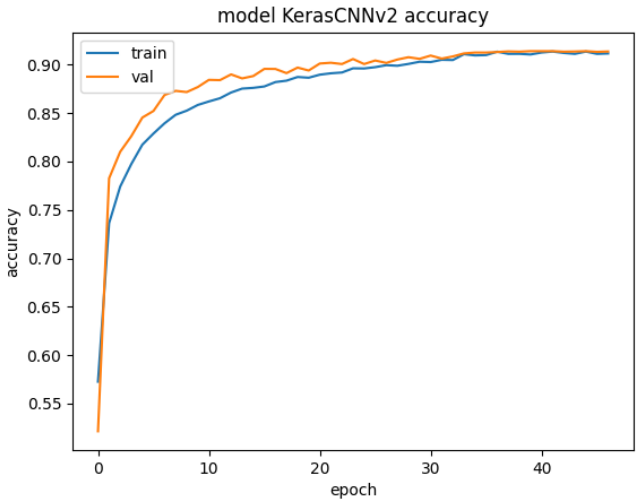

Rysunek 17: Wykres wartości funkcji straty modelu Keras CNNv2 dla bazy MNIST-784 w zależności od epoki



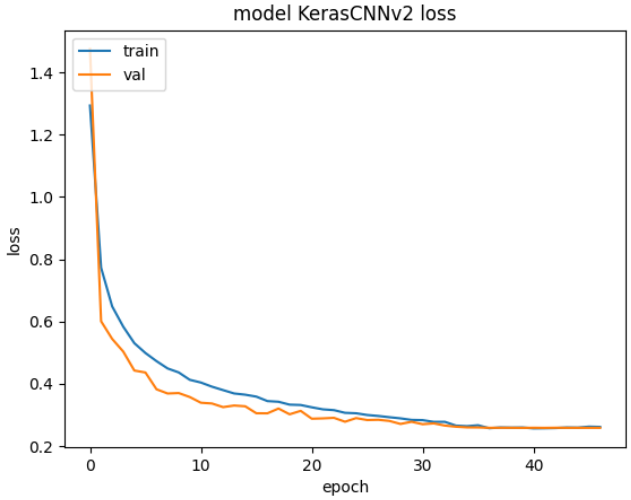

Rysunek 18: Macierz konfuzji modelu Keras CNNv2 dla bazy MNIST-784

Dla bazy MNIST-784 model Keras CNNv2 zdołał uzyskać accuracy w wysokości 0.9924

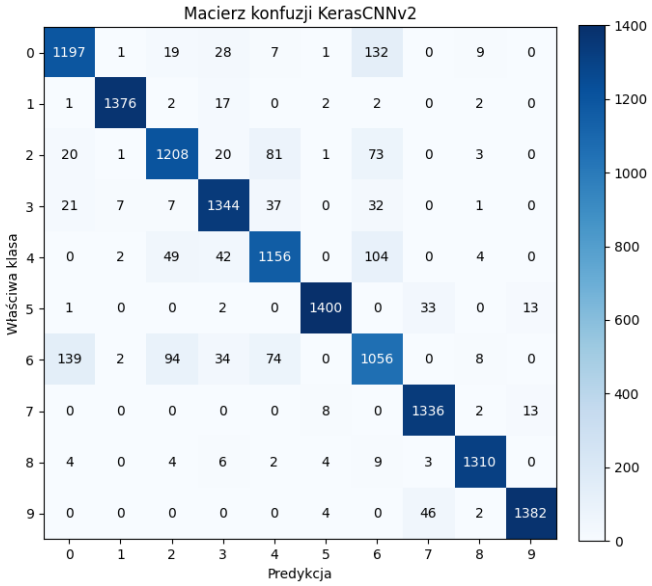
### 5.6.2 Fashion-MNIST




Rysunek 19: Wykres dokładności modelu Keras CNN dla bazy Fashion-MNIST w zależności od epoki




Rysunek 20: Wykres wartości funkcji straty modelu Keras CNNv2 dla bazy Fashion-MNIST w zależności od epoki



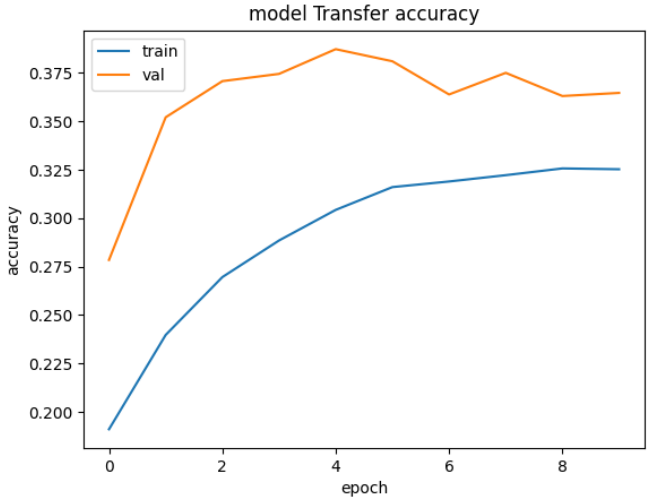

Rysunek 21: Macierz konfuzji modelu Keras CNNv2 dla bazy Fashion-MNIST

Dla bazy Fashion-MNIST model Keras CNNv2 zdołał uzyskać accuracy w wysokości 0.912

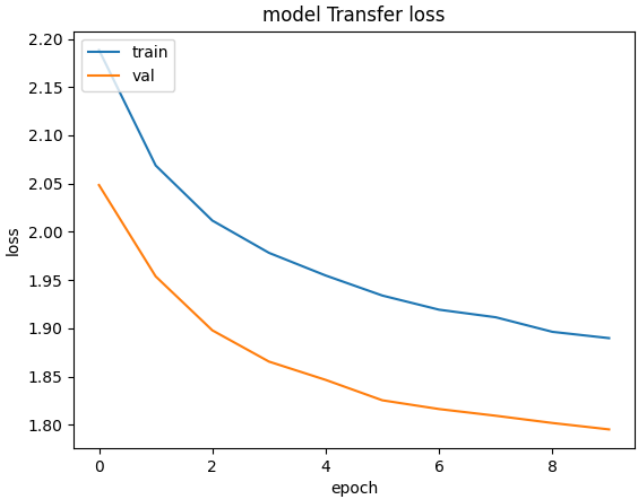
## 5.7 Sieć transferowa Tensorflow Keras

Sieć wyuczono dla dwóch zbiorów danych: MNIST-784 i Fashion-MNIST. W obydwu przypadkach model trenowany był najpierw przez 10 epochów, następnie przez 100 epochów z fine tuningiem. Proces i rezultaty widoczne są na poniższych wykresach:

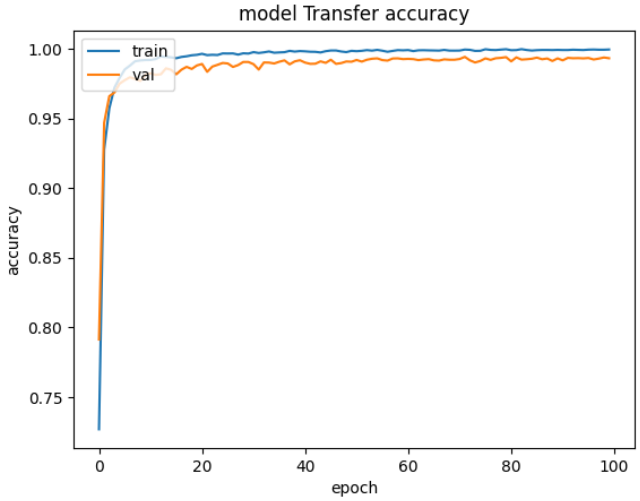
### 5.7.1 MNIST-784



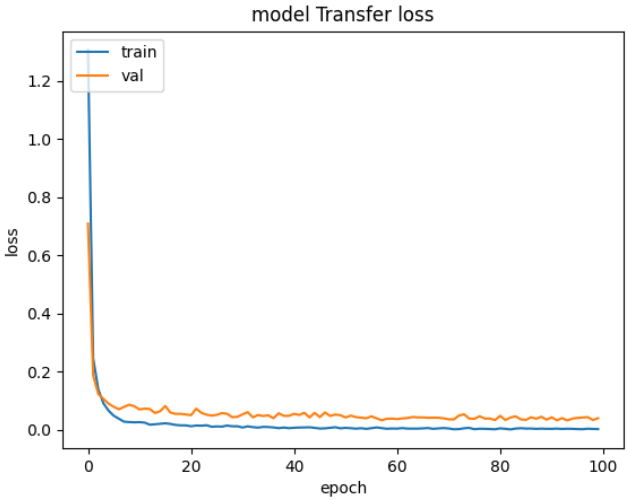

Rysunek 22: Wykres dokładności modelu Transferowego dla bazy MNIST-784 w zależności od epoki w trenowaniu wstępnym



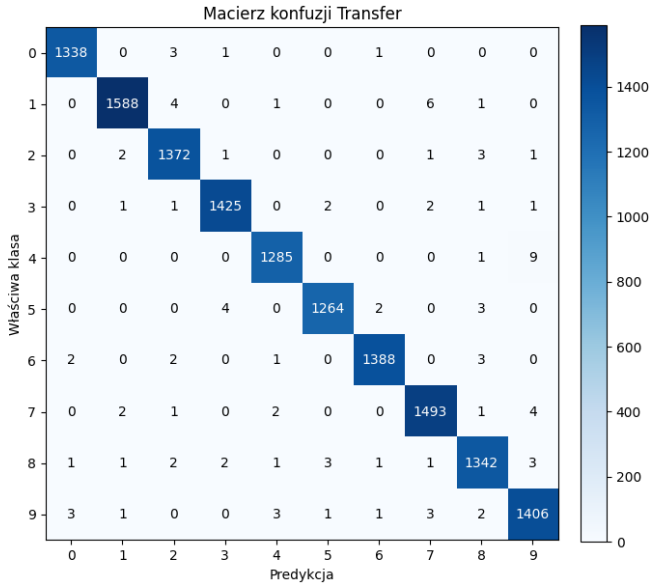

Rysunek 23: Wykres wartości funkcji straty modelu Transferowego dla bazy MNIST-784 w zależności od epoki w trenowaniu wstępnym




Rysunek 24: Wykres dokładności modelu Transferowego dla bazy MNIST-784 w zależności od epoki w fine tuningu



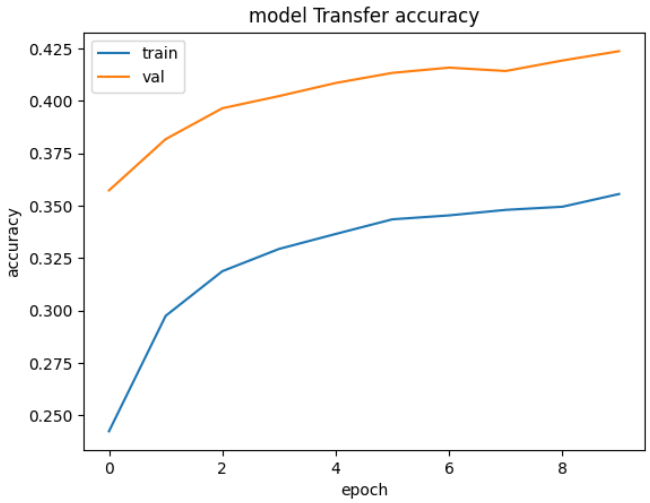

Rysunek 25: Wykres wartości funkcji straty modelu Transferowego dla bazy MNIST-784 w zależności od epoki w fine tuningu



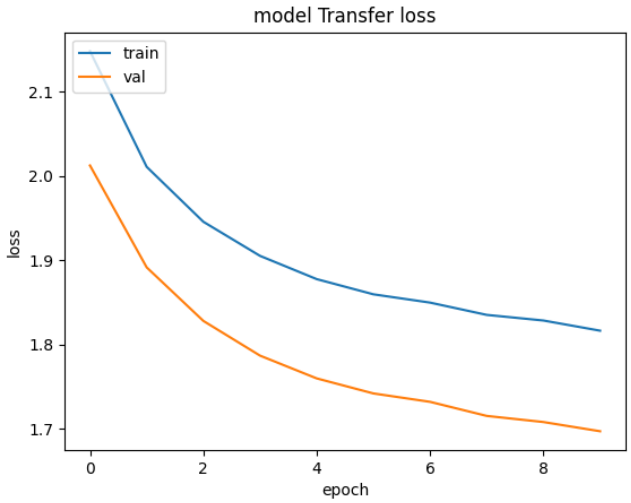

Rysunek 26: Macierz konfuzji modelu Transferowego dla bazy MNIST-784

Dla bazy MNIST-784 model Keras CNN zdołał uzyskać accuracy w wysokości 0.9929

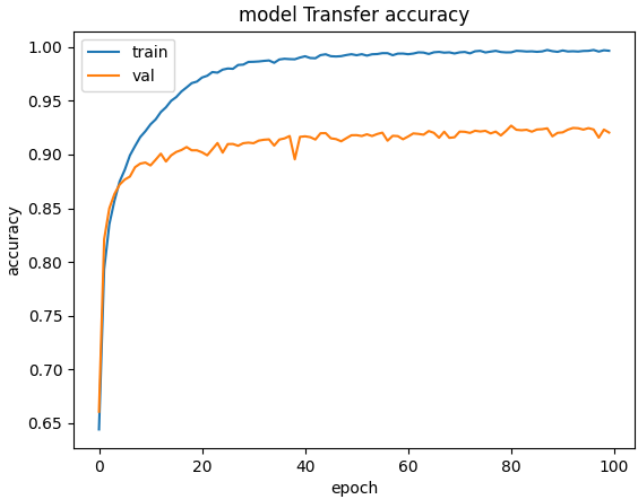
### 5.7.2 Fashion-MNIST



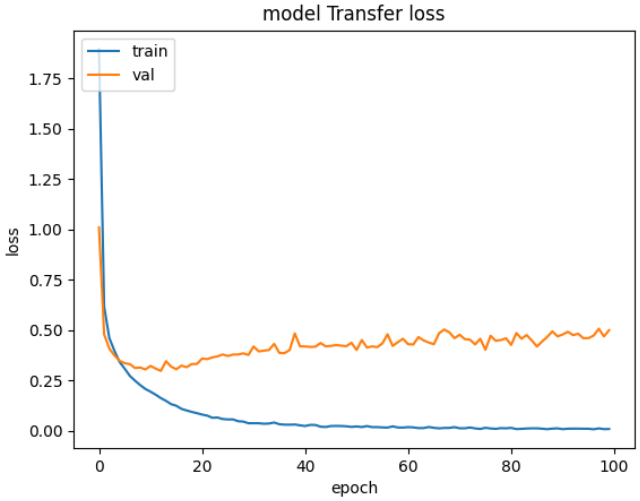

Rysunek 27: Wykres dokładności modelu Transferowego dla bazy Fashion-MNIST w zależności od epoki w trenowaniu wstępnym



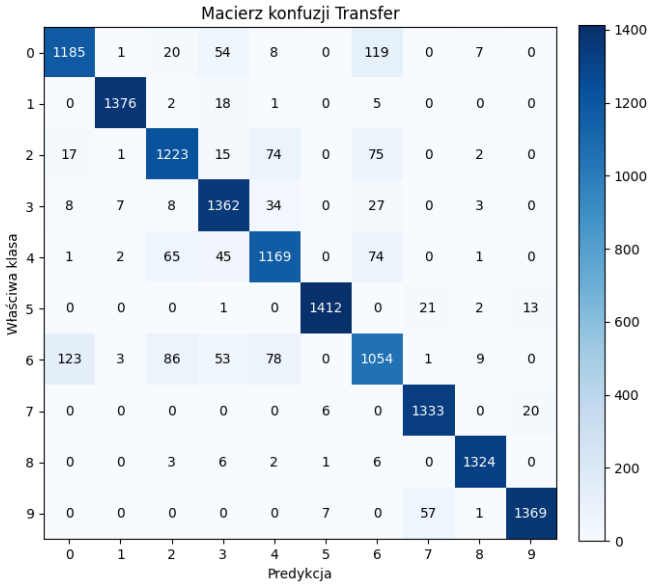

Rysunek 28: Wykres wartości funkcji straty modelu Transferowego dla bazy Fashion-MNIST w zależności od epoki w trenowaniu wstępnym




Rysunek 29: Wykres dokładności modelu Transferowego dla bazy Fashion-MNIST w zależności od epoki w fine tuningu




Rysunek 30: Wykres wartości funkcji straty modelu Transferowego dla bazy Fashion-MNIST w zależności od epoki w fine tuningu




Rysunek 31: Macierz konfuzji modelu Transferowego dla bazy Fashion-MNIST

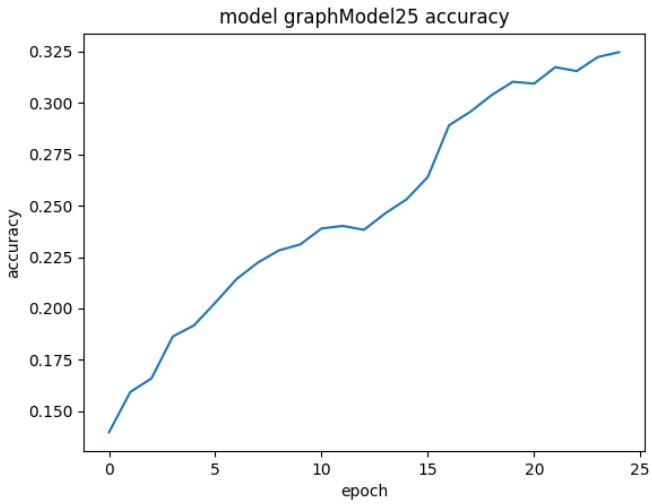
Dla bazy Fashion-MNIST model Keras CNN zdołał uzyskać accuracy w wysokości 0.9148

## 5.8 Sieć grafowa

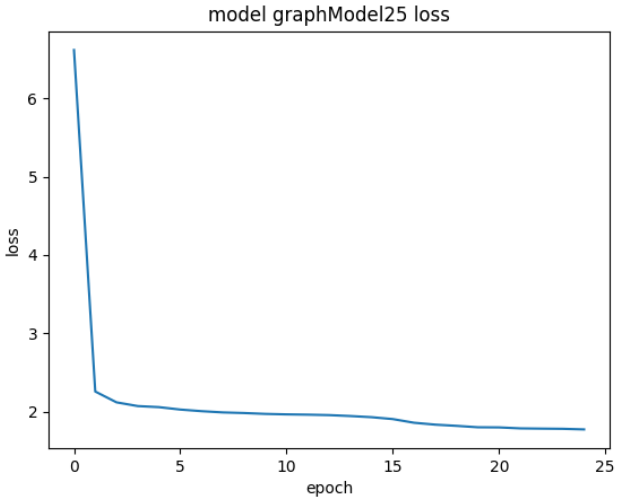
### 5.8.1 Zbiór A

Wyniki dla zbioru A:

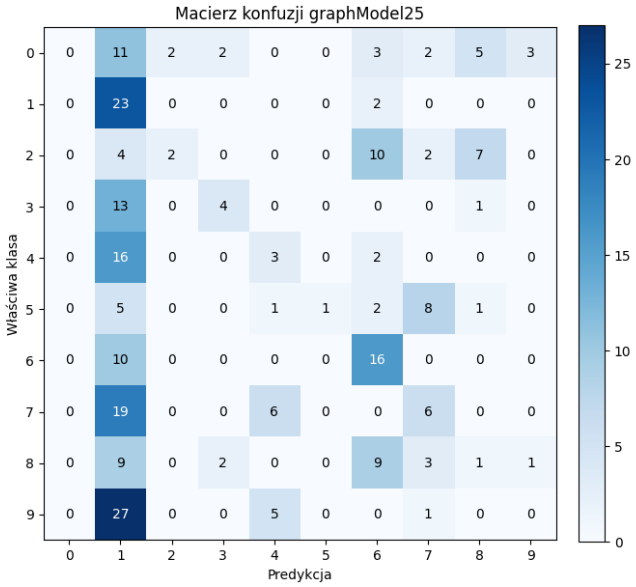
- Loss: 1.9428
- Accuracy: 0.2689




Rysunek 32: Wykres celności dla modelu grafowego trenowanego na zbiorze A w zależności od epoki




Rysunek 33: Wykres wartości funkcji straty dla modelu grafowego trenowanego na zbiorze A w zależności od epoki

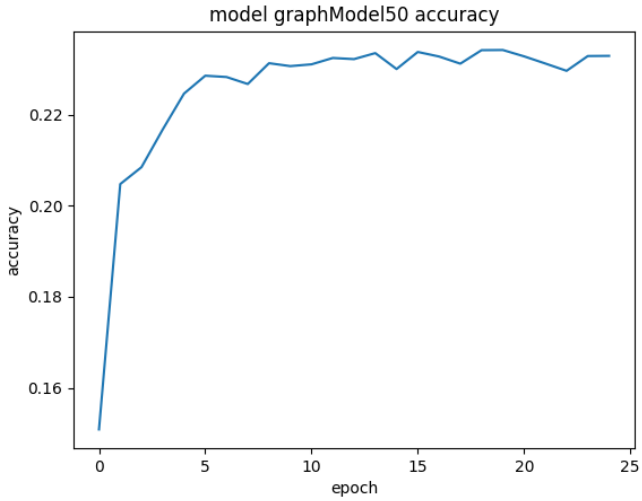



Rysunek 34: Macierz konfuzji dla modelu grafowego trenowanego na zbiorze A w zależności od epoki

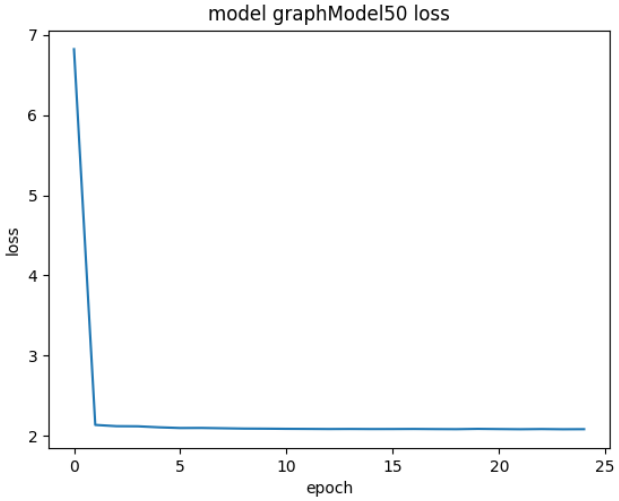
### 5.8.2 Zbiór B

Wyniki dla zbioru B:

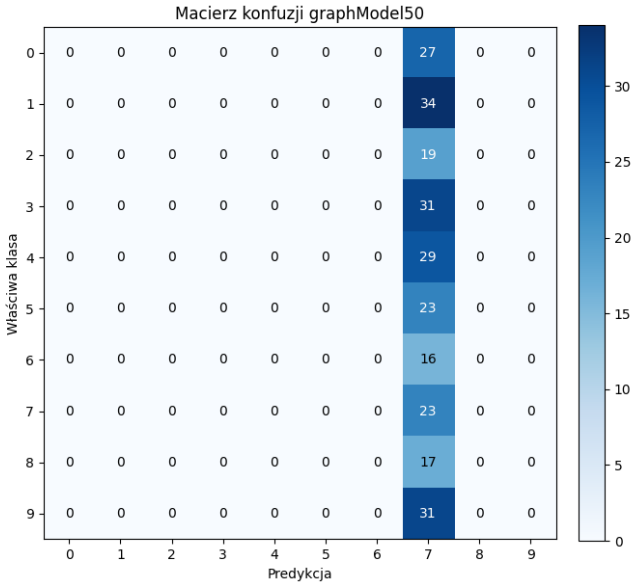
- Loss: 2.7640
- Accuracy: 0.1205




Rysunek 35: Wykres celności dla modelu grafowego trenowanego na zbiorze B w zależności od epoki




Rysunek 36: Wykres wartości funkcji straty dla modelu grafowego trenowanego na zbiorze B w zależności od epoki

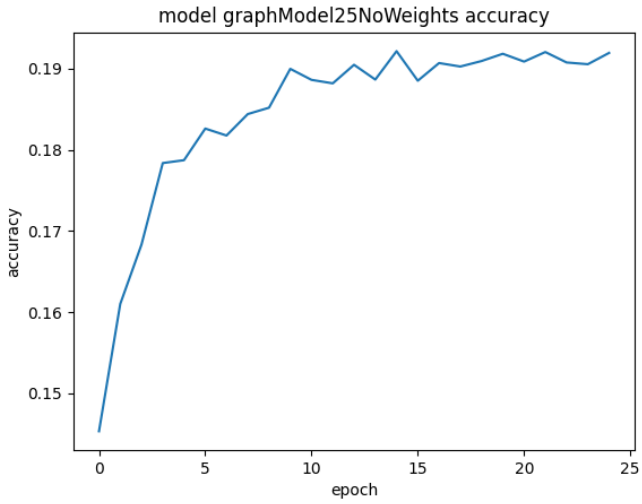



Rysunek 37: Macierz konfuzji dla modelu grafowego trenowanego na zbiorze B w zależności od epoki

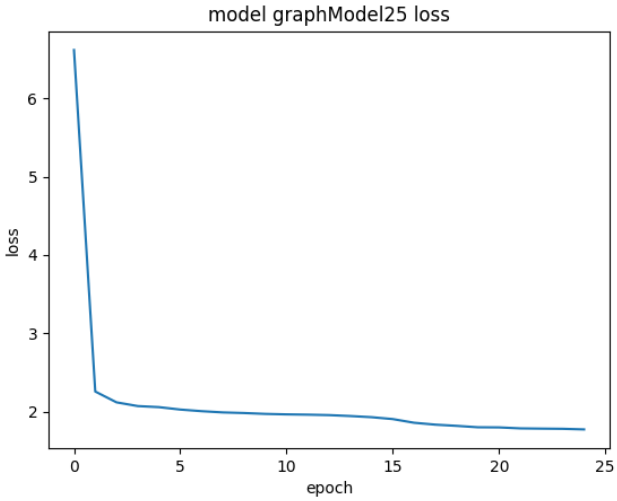
### 5.8.3 Zbiór C

Wyniki dla zbioru C:

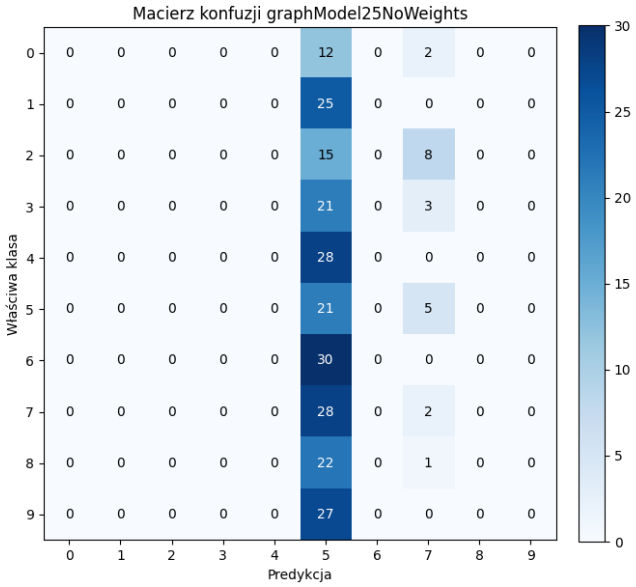
- Loss: 3.5503
- Accuracy: 0.0924




Rysunek 38: Wykres celności dla modelu grafowego trenowanego na zbiorze C w zależności od epoki




Rysunek 39: Wykres wartości funkcji straty dla modelu grafowego trenowanego na zbiorze C w zależności od epoki




Rysunek 40: Macierz konfuzji dla modelu grafowego trenowanego na zbiorze C w zależności od epoki

Sieć w żadnym z przypadków nie uzyskała zbyt imponujących wyników: celność w najlepszym wypadku wynosiła około 25%.

Można zauważyć, że zwiększanie ilości danych nie musi prowadzić do poprawienia wyników: w przypadku bardziej szczegółowych grafów o max. 50 wierzchołkach, wyniki były dużo słabsze niż w prostszym przypadku, gdzie grafy nie miały więcej niż 25 wierzchołków.

O ile redukcja liczby wierzchołków ułatwiała sieci zadanie, to już usunięcie informacji o wagach krawędzi sprawiło, że zaczęła ona uzyskiwać fatalne wyniki.

W oczy rzuca się także fakt, że badany model grafowy ma wyraźną tendencję do faworyzowania jednej konkretnej klasy i przypisywania do niej większości grafów - to zjawisko jest wyraźnie widoczne nawet w najlepszym spośród badanych przypadków.

### 5.9 K Nearest Neighbours

Z powodu długiego czasu predykcji modelu KNN testy wykonano na 10% zbioru testowego, co odpowiada 1400 danych testowych. W tej sytuacji model KNN cechował się accuracy na poziomie 0.96

Z powodu swojej konstrukcji model KNN sprawował się niemal perfekcyjnie dla zbioru danych testowych, które przypominały w znacznym stopniu dane treningowe. Niestety dla ręcznych rysunków nienależących do bazy MNIST-784 model działał dużo gorzej od sieci konwolucyjnej, która nie tylko porównuje punkt z danymi treningowymi, lecz wynajduje schematy i cechy obrazów, które decydują o przynależności do danej klasy. Należy również wspomnieć, że z powodu nieporównywalnie większej ilości obliczeń koniecznych do wykonania przez model KNN w porównaniu do dowolnego innego modelu, podczas gdy proces weryfikacji na 14000 danych testowych sieci neuronowych trwał zaledwie kilka sekund, weryfikacja modelu KNN zajęła ponad 10 minut dla 1400 danych.

### 5.10 Porównanie miar celności modeli

| **Nazwa modelu** | **MNIST-784** | **Fashion-MNIST** |
|---|---|---|
| Drzewo decyzyjne | 0.87 | 0.71 |
| Las losowy | 0.97 | 0.84 |
| Własny wielowarstwowy perceptron (MyNetwork) | 0.97 | 0.84 |
| Wielowarstwowy perceptron (KerasMLP) | 0.96749 | 0.88042 |
| Sieć konwolucyjna (KerasCNN) | 0.98985 | 0.92428 |
| Sieć konwolucyjna rozszerzona (KerasCNNv2) | 0.99235 | 0.91178 |
| Sieć transferowa (Transfer) | 0.99293 | 0.91478 |
| Sieć grafowa (Graph) | 0.2689 | – |
| K Nearest Neighbours (KNN) | 0.96 | 0.96 |

Tabela 1: Porównanie celności modeli

## 6 Dyskusja

## Źródła

[1] Wikipedia, *Sieć neuronowa*
`https://pl.wikipedia.org/wiki/Sie%C4%87_neuronowa`