Sprawozdanie z Projektu: Klasyfikacja Fashion MNIST przy użyciu Sieci Konwolucyjnej (CNN)

Maksymilian Lech (280136)
Kornel Lewandowski (280101)
Wiktor Hebdowski (275964)

22 stycznia 2026

Przedmiot: Podstawy Sieci Neuronowych

Spis treści

1 Analiza problemu i wybór architektury

1.1 Opis problemu

W ramach projektu zajęliśmy się klasyfikacją obrazów z zestawu danych Fashion MNIST. Zbiór ten zawiera 70 000 obrazów w skali szarości o rozmiarze 28×28 pikseli, przedstawiających 10 różnych kategorii odzieży. Naszym celem było stworzenie modelu zdolnego do poprawnego rozpoznawania tych wzorców wizualnych.

1.2 Wybór architektury sieci

Do rozwiązania tego zadania wybraliśmy **sieć konwolucyjną (CNN)**. Decyzję tę oparliśmy na następujących przesłankach:

- Warstwy konwolucyjne efektywnie wyłapują detale ubrań, co jest kluczowe przy tego typu danych.
- CNN zapewnia translacyjną niezmienniczość oraz redukcję liczby parametrów w porównaniu do sieci w pełni połączonych.
- Struktura ta pozwala zachować rozsądną szybkość obliczeń przy wysokiej skuteczności.

2 Architektura programu i narzędzia

2.1 Wykorzystane biblioteki

Projekt zrealizowaliśmy w języku Python, korzystając z biblioteki PyTorch do budowy i trenowania sieci. Ponadto wykorzystaliśmy:

- **Torchvision**: do transformacji i normalizacji danych.
- **Kagglehub**: do automatycznego pobierania zbioru danych.
- **Matplotlib / Seaborn**: do generowania wykresów i wizualizacji wyników.
- **Scikit-learn**: do obliczania metryk klasyfikacji.

2.2 Struktura projektu

Zdecydowaliśmy się na pełne oddzielenie logiki modelu od jego parametrów. Strukturę projektu zaprojektowaliśmy w następujący sposób:

- `config.py`: Tutaj umieściliśmy kluczowe ustawienia, co pozwoliło nam zmieniać konfigurację (np. learning rate, batch size) bez ingerencji w kod sieci.
- `experiments.py`: Stworzyliśmy ten skrypt, aby zautomatyzować proces badawczy i uruchamianie serii scenariuszy testowych.
- `models/cnn.py`: W tym pliku zdefiniowaliśmy klasę `FashionCNN`.

Postanowiliśmy również wdrożyć obsługę rdzeni CUDA, aby umożliwić wybór między CPU a GPU, co pozwoliło nam skrócić czas treningu z minut do sekund.

3 Analiza i przygotowanie danych

3.1 Podział danych

Zgodnie z wymaganiami, podzieliliśmy zbiór danych na trzy zestawy:

- **Zbiór treningowy:** ok. 50 000 obrazów (służący do uczenia wag).
- **Zbiór walidacyjny:** ok. 10 000 obrazów (wykorzystywany przez mechanizm Early Stopping do monitorowania postępów).
- **Zbiór testowy:** 10 000 obrazów (użyty do finalnej oceny modelu).

Dane wejściowe znormalizowaliśmy do zakresu $[-1, 1]$ oraz sformatowaliśmy jako tensory o wymiarach $(Batch, 1, 28, 28)$.

4 Architektura sieci (Model)

Zaprojektowaliśmy model składający się z następujących bloków:

4.1 Warstwy konwolucyjne

Zastosowaliśmy 3 warstwy `Conv2d` o rosnącej liczbie filtrów: 32, 64 i 128.

- Do każdej warstwy dodaliśmy **Batch Normalization**, aby ustabilizować proces uczenia.
- Po konwolucjach zastosowaliśmy **Max Pooling**, aby stopniowo zmniejszać wymiarowość danych.

4.2 Warstwy w pełni połączone

Na końcu sieci dane są spłaszczane i trafiają do klasyfikatora liniowego.

- Dodaliśmy mechanizm **Dropout (0.5)**, który chroni nasz model przed "nauczeniem się obrazków na pamięć"(overfittingiem).
- Ostatnia warstwa mapuje cechy na 10 klas wyjściowych.

5 Trenowanie sieci i eksperymenty

Przeprowadziliśmy serię eksperymentów, korzystając ze skryptu automatyzującego, który dla każdego przebiegu trenował model, zapisywał najlepsze wagi i generował raport.

5.1 Zestawienie wyników

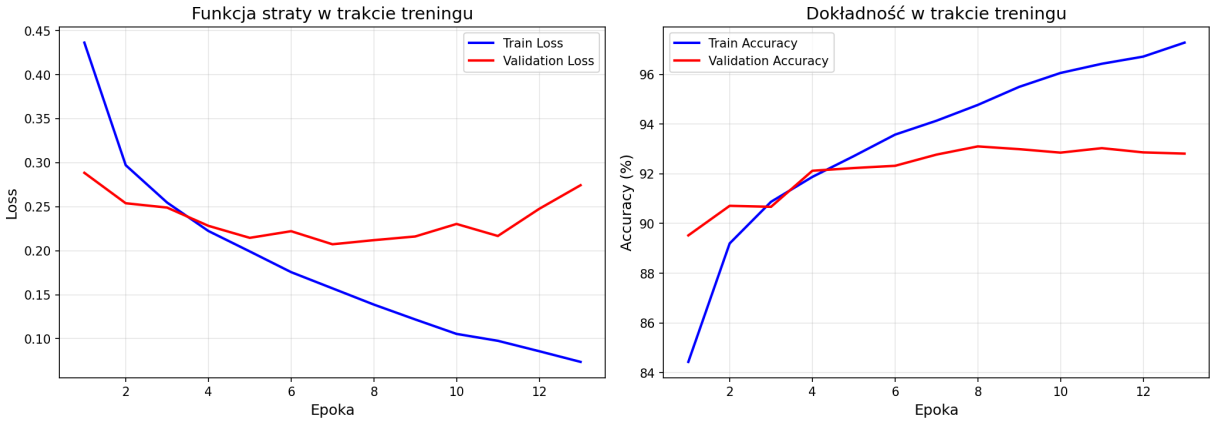
Przeanalizowaliśmy wpływ różnych konfiguracji na wyniki końcowe.

Eksperyment	Test Acc	Val Acc	Epoki	Nasze obserwacje
Baseline	92.14%	93.10%	13	Punkt odniesienia
High LR (0.01)	91.34%	91.83%	20	Niestabilność treningu
Large Batch (128)	92.84%	93.22%	19	Stabilniejsze gradienty
More Filters	**92.87%**	**93.41%**	17	**Najlepszy wynik**
Less Dropout (0.25)	92.78%	93.12%	17	Ryzyko overfittingu

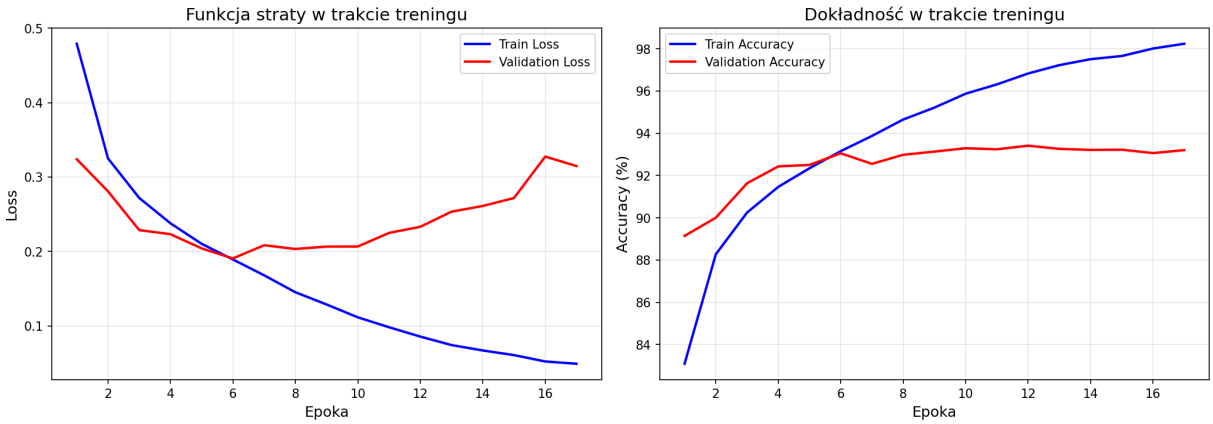
Tabela 1: Porównanie wyników uzyskanych w naszych eksperymentach.

5.2 Wizualizacja procesu uczenia

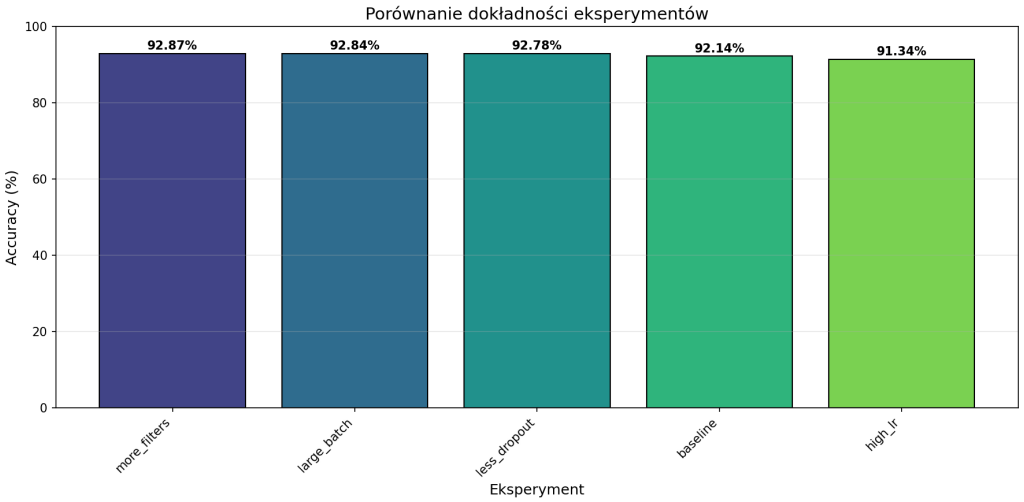
Poniżej prezentujemy wizualizacje procesu uczenia się, które wygenerowaliśmy dla wybranych eksperymentów.



Rysunek 1: Krzywe uczenia, które uzyskaliśmy dla modelu referencyjnego (Baseline).



Rysunek 2: Krzywe uczenia dla naszego najlepszego modelu (More Filters).



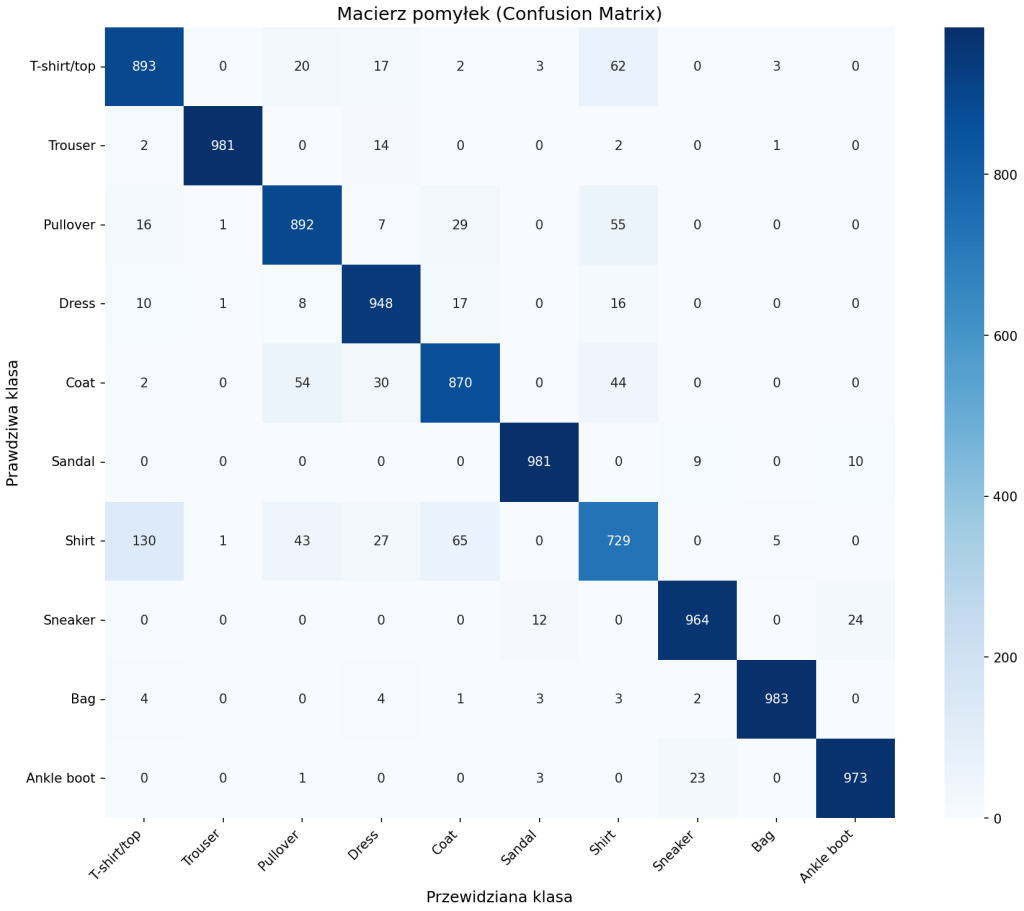
Rysunek 3: Porównanie dokładności testowej wszystkich przeprowadzonych przez nas eksperymentów.

6 Analiza wyników

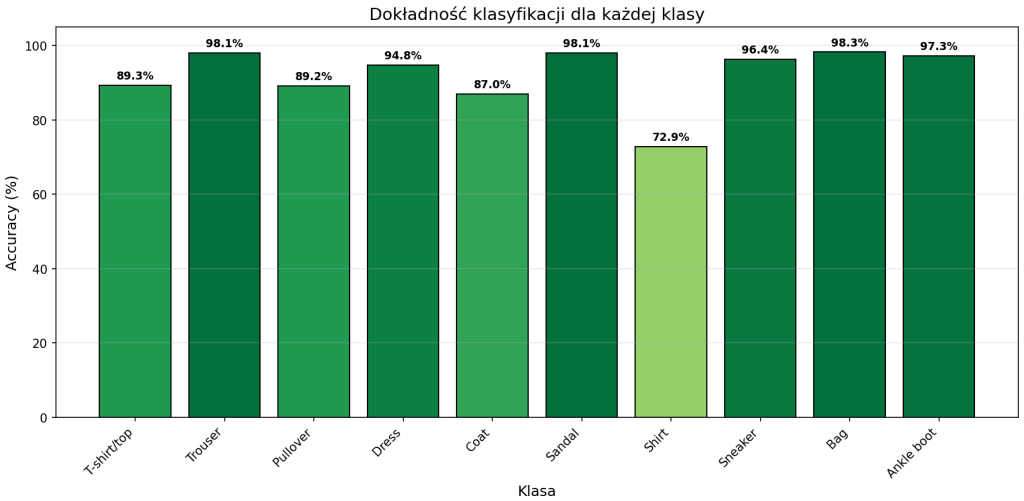
W toku analizy ustaliliśmy, że najlepsze rezultaty przyniósł eksperyment `more_filters`, osiągając 92.87% dokładności na zbiorze testowym.

6.1 Macierz pomyłek i analiza błędów

Przygotowaliśmy macierze pomyłek, aby zrozumieć, gdzie nasz model popełnia błędy.



Rysunek 4: Macierz pomyłek (Confusion Matrix) dla modelu Baseline.

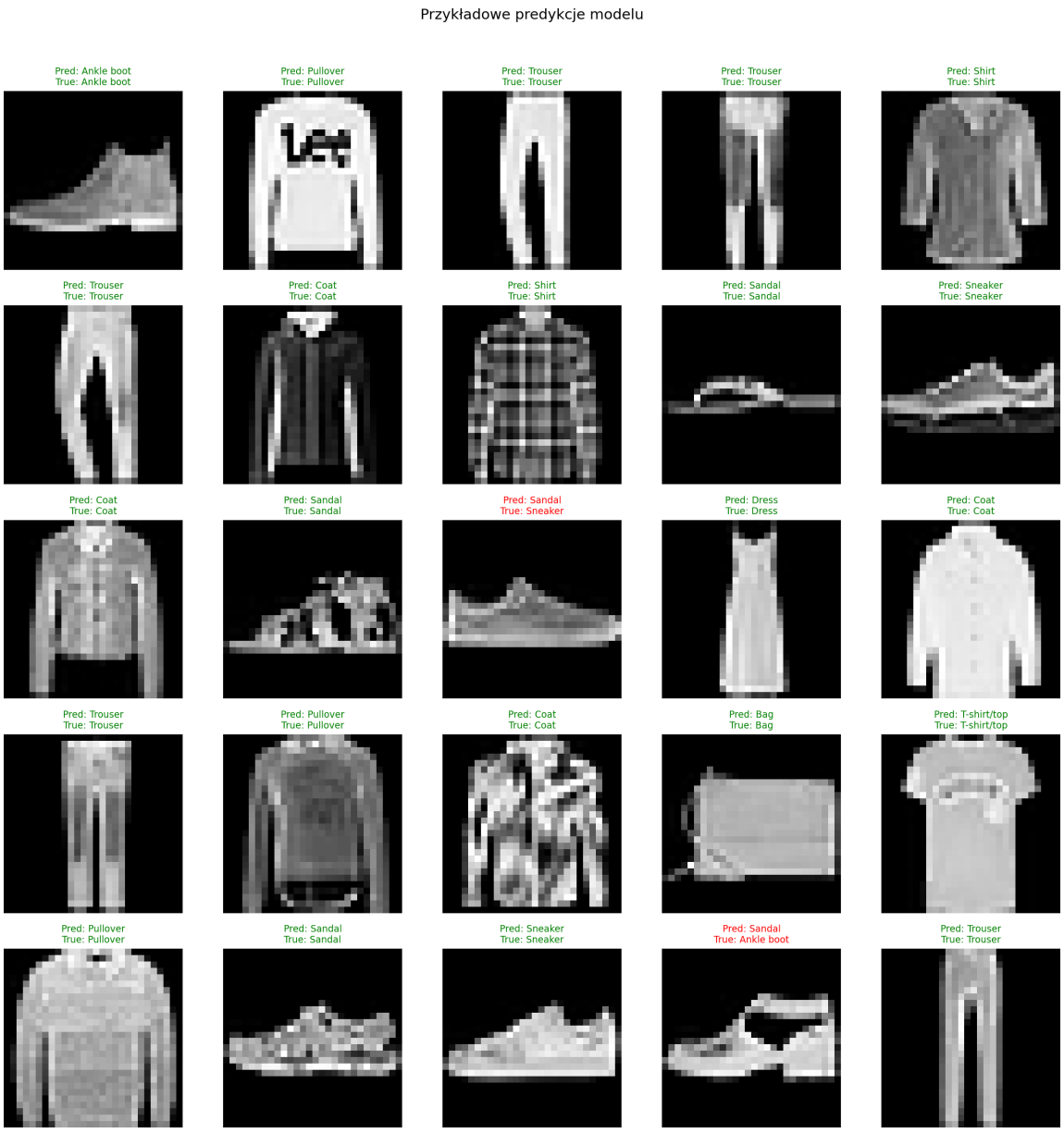


Rysunek 5: Dokładność klasyfikacji per klasa, którą zmierzyliśmy dla modelu Baseline.

Zauważyliśmy, że model najczęściej myli klasy wizualnie podobne, takie jak *T-shirt* z *Shirt* czy *Pullover* z *Coat*.

6.2 Przykładowe predykcje

Wygenerowaliśmy wizualizację predykcji, aby ocenić jakość klasyfikacji "okiem".



Rysunek 6: Przykładowe predykcje naszego modelu (zielone = poprawne, czerwone = błędne).

6.3 Sukcesy, porażki i napotkane problemy

Podczas realizacji projektu napotkaliśmy i rozwiązaliśmy następujące problemy:

- **Problem z formatem danych:** Napotkaliśmy błąd przy wczytywaniu plików Gzip. Rozwiązaliśmy go, implementując automatyczne wykrywanie formatu pliku.
- **Wydajność:** Trening na CPU był zbyt wolny, dlatego zaimplementowaliśmy automatyczne wykrywanie i wykorzystanie GPU.
- **Zbyt wysoki Learning Rate:** W eksperymencie z LR=0.01 zaobserwowaliśmy pogorszenie wyników, co pozwoliło nam ustalić, że wartość 0.001 jest optymalna.

7 Wnioski końcowe

Nasz projekt zakończył się sukcesem, a otrzymane wyniki spełniły nasze oczekiwania (accuracy $> 90\%$).

1. Potwierdziliśmy, że **Batch Normalization** skutecznie stabilizuje trening.
2. Wykorzystanie parametru `patience` w mechanizmie **Early Stopping** pozwoliło nam automatycznie przerywać trening w optymalnym momencie.
3. Eksperymenty wykazały, że zwiększenie liczby filtrów w warstwach konwolucyjnych przynosi największą poprawę dokładności.

Link do naszego repozytorium: https://github.com/ilszew/PSN_CNN