

**WYDZIAŁ  
ELEKTROTECHNIKI  
I INFORMATYKI  
POLITECHNIKI RZESZOWSKIEJ**

L8 2EF-DI  
Daniel Kleczyński

**Temat: Badanie niektórych parametrów  
konwolucyjnej sieci głębokiej używając zbioru  
obrazów WHU-RS19**

**Prowadzący:**

dr hab. inż. Roman Zajdel, prof. PRz

Rzeszów, 2023

# Spis treści

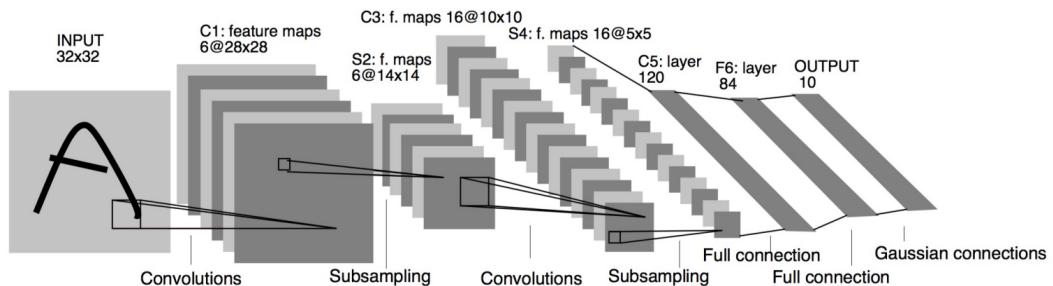
|  |           |
|--|-----------|
| <b>Wprowadzanie</b>  | <b>3</b>  |
| <b>1. Problematyka oraz charakterystyczne rozwiązania w głębokich sieciach CNN w zastosowaniu klasyfikacji zdjęć</b> | <b>5</b>  |
| 1.1. Problematyka zanikającego gradnietu   | 5         |
| 1.2. Charakterystyczne rozwiązania w głębokich sieciach CNN  | 6         |
| 1.2.1. Funkcja aktywacji ReLU  | 6         |
| 1.2.2. Inicjalizacja wag przy użyciu Xavier/Glorot.  | 7         |
| 1.2.3. Normalizacja danych lub warstw  | 8         |
| 1.2.4. Architektur sieci, które umożliwiają przepływ gradientu w głąb sieci  | 8         |
| <b>2. Dane uczące oraz ich przygotowanie</b>   | <b>10</b> |
| <b>3. Przedstawienie oraz omówienie własnej architektury CNN</b>   | <b>13</b> |
| <b>4. Przedstawienie oraz omówienie architektury sieci tf mobilenetv3 small minimal 100</b>                          | <b>16</b> |
| <b>5. Omówienie klasy LightningModule oraz optymalizatora ADAM</b>   | <b>18</b> |
| <b>6. Eksperyment I. badanie parametru rozmiaru wsadu oraz wielkości zdjęcia</b>                                     | <b>19</b> |
| 6.1. Metoda badawcza   | 21        |
| 6.2. Wyniki  | 22        |
| <b>7. Eksperyment II. Uczenie i porównanie modeli po 200 epokach</b>   | <b>24</b> |
| 7.1. Metoda badawcza   | 24        |
| 7.2. Wyniki  | 25        |
| <b>8. Podsumowanie</b>   | <b>29</b> |
| <b>Literatura</b>  | <b>31</b> |

# Wprowadzanie

Głębokie sieci konwolucyjne (ang. deep convolutional neural networks, CNN) są rodzajem modeli głębokiego uczenia maszynowego, które są szczególnie skuteczne w analizie i przetwarzaniu danych wizyjnych, takich jak obrazy i filmy. Ich zastosowanie obejmuje wiele dziedzin, takich jak rozpoznawanie obrazów, segmentacja, detekcja obiektów, klasyfikacja i wiele innych.

Sieci konwolucyjne składają się z kilku warstw konwolucyjnych, które ekstrahują cechy z danych wejściowych poprzez przeprowadzenie operacji konwolucji na obrazach. Operacje konwolucji polegają na nakładaniu filtrów na obrazy i obliczaniu ich odpowiedzi. Filtry te uczą się automatycznie w procesie trenowania, aby wyodrębnić istotne wzorce i cechy z obrazów.

Głębokie sieci konwolucyjne składają się z wielu warstw konwolucyjnych, które są stosowane sekwencyjnie. Zazwyczaj zaczynają się od prostych filtrów, które wykrywają podstawowe cechy, takie jak krawędzie i tekstury, a następnie kolejne warstwy wykrywają coraz bardziej skomplikowane wzorce. W miarę postępu w głąb sieci, abstrakcyjność cech rośnie, co pozwala na bardziej zaawansowane rozumienie i interpretację obrazów stosuje się również klasyczne warstwy neuronów które mogą posłużyć do przekształcenia na klasyfikator przykładowy schemat sieci CNN rysunek 0.1 .



Rysunek 0.1: Architektura sieci konwolucyjnych

Zastosowanie głębokich sieci konwolucyjnych ma szerokie spektrum. W dziedzinie rozpoznawania obrazów, CNN może być wykorzystywany do klasyfikacji obiektów na obrazach, np. rozpoznawanie gatunków zwierząt lub identyfikowanie przedmiotów. Mogą również być stosowane do detekcji obiektów, gdzie sieć może wykrywać obiekty na obrazie i określać ich położenie.

Sieci konwolucyjne są również wykorzystywane w segmentacji obrazu, gdzie mogą dzielić obraz na różne regiony lub piksele należące do różnych klas. Ponadto, CNN znalazły zastosowanie w dziedzinach takich jak rozpoznawanie mówcy, analiza medyczna, przetwarzanie naturalnego języka i wiele innych.

# 1. Problematyka oraz charakterystyczne rozwiązania w głębokich sieciach CNN w zastosowaniu klasyfikacji zdjęć

W celu uczenia algorytmów coraz bardziej skomplikowanych zależność wielkość warstw oraz głępokość sieci była stale powiększana w przypadku algorytmów MLP skutkowało to bardzo dużą ilością parametrów do uczenia. Rozważaniem były warstwy konwolucyjne oraz inne rozwiązania.

## 1.1. Problematyka zanikającego gradientu

Problem zanikającego gradientu występuje, gdy w trakcie propagacji wstecznej gradient maleje wraz z głębokością sieci, co prowadzi do trudności w uczeniu głębokich warstw. Wzór na zanikający gradient można przedstawić w następujący sposób:

Niech  $\frac{\partial J}{\partial W}$  oznacza gradient funkcji kosztu  $J$  względem wag  $W$ . Wówczas gradient dla danej warstwy można obliczyć jako iloczyn gradientu warstwy powyżej i pochodnej funkcji aktywacji. Przykładowo, dla dwóch warstw:

$$\delta_2 = \frac{\partial J}{\partial A_2} \cdot g'(z_2)$$

$$\delta_1 = W_2^T \cdot \delta_2 \cdot g'(z_1)$$

gdzie:

- $\delta_2$  to gradient dla warstwy drugiej (bliżej wyjścia)
- $\delta_1$  to gradient dla warstwy pierwszej (bliżej wejścia)
- $\frac{\partial J}{\partial A_2}$  to gradient funkcji kosztu  $J$  względem wyjścia warstwy drugiej
- $g'(z)$  to pochodna funkcji aktywacji względem wejścia  $z$
- $z_2$  to wejście do warstwy drugiej
- $W_2$  to macierz wag dla warstwy drugiej
- $z_1$  to wejście do warstwy pierwszej

Problem zanikającego gradientu występuje, gdy pochodna funkcji aktywacji  $g'(z)$  jest mała (bliska 0) dla większości wartości  $z$  w obszarze, w którym znajduje

się większość danych. W takiej sytuacji, iloczyn  $W_2^T \cdot \delta_2 \cdot g'(z_1)$  może również stać się bardzo mały, co prowadzi do zanikania gradientu dla warstwy pierwszej. To z kolei powoduje, że wagi w warstwie pierwszej nie są aktualizowane w odpowiedni sposób, co utrudnia efektywne uczenie sieci.

## 1.2. Charakterystyczne rozwiązania w głębokich sieciach CNN

Aby radzić sobie z problemem zanikającego gradientu, stosuje się różne techniki, takie jak:

- Użycie innych funkcji aktywacji, które nie mają tendencji do zanikania gradientu, na przykład funkcji ReLU (Rectified Linear Unit).
- Inicjalizacja wag w sposób bardziej odpowiedni, na przykład przy użyciu inicjalizacji Xavier/Glorot.
- Normalizacja danych lub warstw w celu zmniejszenia zakresu wartości wejściowych.
- Wykorzystanie architektur sieci, które umożliwiają przepływ gradientu na większe odległości, takie jak sieci rekurencyjne (RNN) lub połączenia pomijające (skip connections) w sieciach konwolucyjnych.

Dzięki tym technikom można zminimalizować problem zanikającego gradientu i umożliwić skuteczne uczenie głębokich sieci neuronowych.

### 1.2.1. Funkcja aktywacji ReLU

Funkcja ReLU jest popularną funkcją aktywacji w sieciach neuronowych, ponieważ ma prostą i efektywną implementację oraz pomaga w rozwiązyaniu problemu zanikającego gradientu. Funkcja zwraca wartość zero dla wszystkich wartości wejściowych mniejszych niż zero, a dla wartości nieujemnych zwraca wartość równą wejściu.

$$f(x) = \begin{cases} 0, & \text{dla } x < 0, \\ x, & \text{dla } x \geq 0. \end{cases}$$

Funkcja ta jest prostą nieliniową funkcją, która działa jako prosta przekształcenie dla wartości nieujemnych i przepuszcza je bez zmian. Jeśli wartość wejściowa

jest ujemna, funkcja zwraca 0. Graficznie, funkcja ReLU można przedstawić jako prostą linię przechodzącą przez punkt  $(0,0)$ , a dla wartości ujemnych osiąga wartość 0, a dla wartości nieujemnych zachowuje się jak funkcja tożsamościowa. Funkcja ReLU jest popularna ze względu na jej prostotę obliczeniową oraz zdolność do rozwiązywania problemu zanikającego gradientu. Ponadto, ReLU jest nieodwracalna, co może wprowadzić do modelu nieliniowość i zdolność do wykrywania i reprezentowania złożonych wzorców i relacji w danych.

### **1.2.2. Inicjalizacja wag przy użyciu Xavier/Glorot.**

Xavier (również znane jako inicjalizacja Glorot) to popularna metoda inicjalizacji wag w sieciach neuronowych. Metoda ta ma na celu zapewnienie odpowiedniej skali wag na początku uczenia, aby uniknąć zanikającego gradientu.

Inicjalizacja Xavier/Glorot opiera się na rozkładzie normalnym lub jednostajnym w zależności od funkcji aktywacji. W przypadku rozkładu normalnego, wagi są inicjalizowane zgodnie z następującym wzorem:

$$W \sim N\left(0, \frac{1}{\sqrt{n_{in}}}\right)$$

gdzie:

- $W$  to macierz wag,
- $N(\mu, \sigma)$  to rozkład normalny o średniej  $\mu$  i odchyleniu standardowym  $\sigma$ ,
- $n_{in}$  to liczba neuronów w poprzedniej warstwie.

Natomiast w przypadku rozkładu jednostajnego, wagi są inicjalizowane zgodnie z następującym wzorem:

$$W \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right)$$

gdzie:

- $U(a, b)$  to rozkład jednostajny o przedziale  $(a, b)$ ,
- $n_{out}$  to liczba neuronów w bieżącej warstwie.

Inicjalizacja Xavier/Glorot pozwala na równomierne rozprowadzenie wariancji wejść i wyjść neuronów, co sprzyja stabilnemu uczeniu sieci neuronowej. Dzięki temu

metoda ta może pomóc w uniknięciu zanikającego lub eksplodującego gradientu na początku uczenia.

### **1.2.3. Normalizacja danych lub warstw**

Normalizacja danych lub warstw jest powszechną techniką stosowaną w sieciach neuronowych w celu zmniejszenia zakresu wartości wejściowych. Ma to na celu ułatwienie uczenia się modelu poprzez zapewnienie bardziej stabilnego i efektywnego procesu optymalizacji. Istnieją różne metody normalizacji danych lub warstw, a niektóre z najpopularniejszych to:

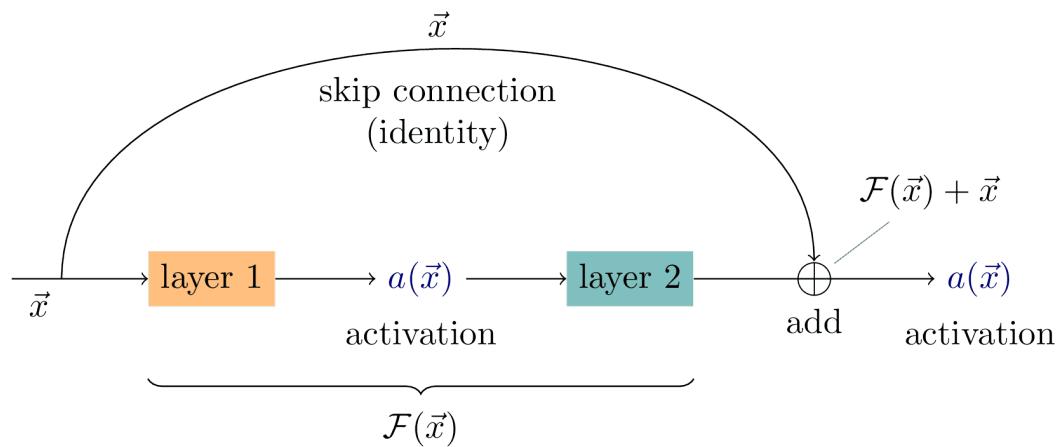
- Normalizacja z-score (standardowa normalizacja)
- Normalizacja min-max Ta metoda przekształca wartości danych w zakres od 0 do 1.
- Batch Normalization (normalizacja wsadowa)
- Layer Normalization (normalizacja warstwowa)

### **1.2.4. Architektur sieci, które umożliwiają przepływ gradientu w głąb sieci**

Sieci konwolucyjne z połączonymi pomijającymi (skip connections) są rozwińciem tradycyjnych sieci konwolucyjnych, które mają na celu rozwiązanie problemu zanikającego gradientu oraz wzmacnieniu nisko abstrakcyjnych cech obrazu.

W sieciach konwolucyjnych z połączonymi pomijającymi wprowadza się dodatkowe połączenia, które tworzą skoki w przepływie informacji. Te połączenia łączą warstwy wcześniejsze bezpośrednio z warstwami późniejszymi, pomijając niektóre pośrednie warstwy konwolucyjne rysunek 1.2. Dzięki temu połączeniu informacja o niskopoziomowych cechach może być bezpośrednio przenoszona do warstw wyższego poziomu.

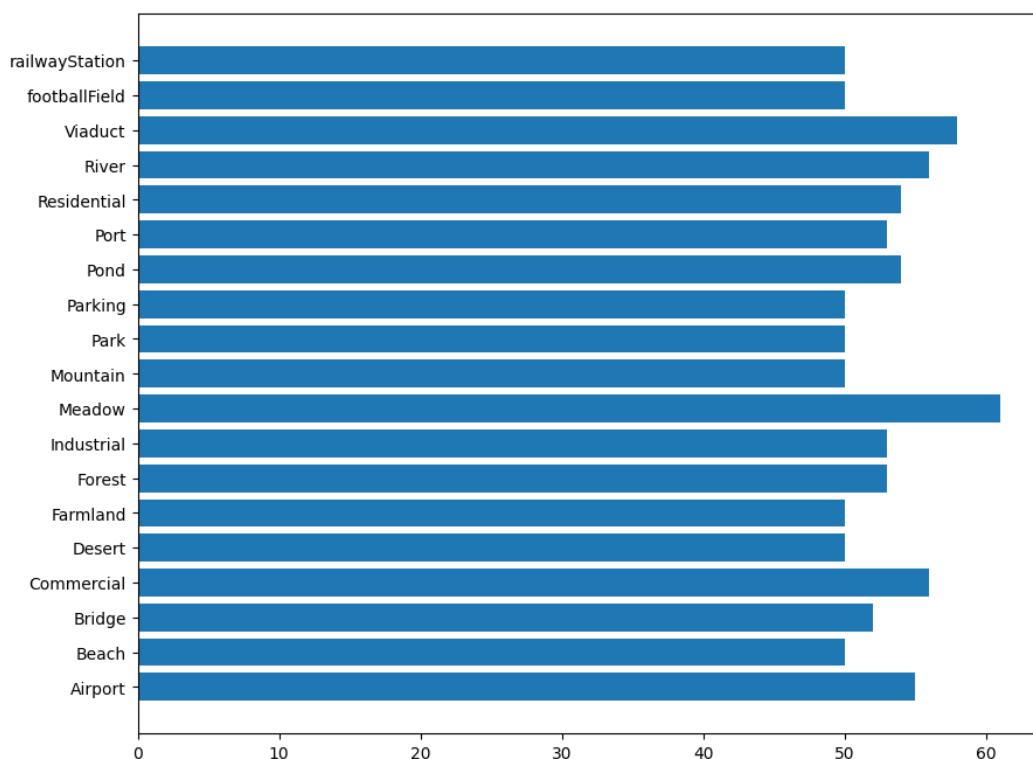
Główną korzyścią wynikającą z wykorzystania połączeń pomijających jest możliwość przekazywania i zachowywania informacji o niskopoziomowych cechach, które są istotne dla skutecznego uczenia się sieci. Dzięki temu sieć może efektywniej wykorzystać te cechy do dokładniejszej klasyfikacji lub predykcji. Ponadto, połączenia pomijające umożliwiają propagację gradientu wstecznie z wyższych warstw do niższych warstw, co ułatwia proces uczenia się i przeciwdziała zanikaniu gradientu.



Rysunek 1.2: Architektura z połączeniami pomijającymi

## 2. Dane uczące oraz ich przygotowanie

W procesie uczenia będzie używany zbiór WHU-RS19 o liczecności 1005 jest to zbiór Kwadratowych kolorowych zdjęć o rozmiarach 600x600 px w formacie .jpg. Zrobione Satelitarnie Zdjęcia przedsatiają obiekty takie jak 'Airport', 'Beach', 'Bridge', 'Commercial', 'Desert', 'Farmland', 'Forest', 'Industrial', 'Meadow', 'Mountain', 'Park', 'Parking', 'Pond', 'Port', 'Residential', 'River', 'Viaduct', 'footballField', 'railwayStation' ich liczecnoś jest zaprzentowana na wykresie 2.4



Rysunek 2.3: Zestawienie liczebności zdjęć w poszczególnych klasach

Dane w takiej konfiguracji nie są odpowiednie do uczenia składa się na to kilka cech takich jak zbyt duży rozmiar zdjęcia co powoduje znaczne zwiększenie danych wejściowych oraz obecny format który nie jest w stanie być przetwarzany na jednostkach obliczeniowych 'cuda' a to również wydłuża proces ucznia. Na zdjęciach zostanie przeprowadzona transformacja do typu danych jakim są tensorsy o rozmiarach `tensor(1, 3, 384, 384)` Poszczególne wymiary mają następujące znaczenie:

- (1): Odpowiada liczbie przykładów w zbiorze danych. W tym przypadku mamy tylko jeden przykład.

- (3): Odpowiada liczbie kanałów koloru. W przypadku obrazów w formacie RGB.
- (384): Odpowiada liczbie pikseli w pionie (wysokość obrazu) przedział wartości.
- (384): Odpowiada liczbie pikseli w poziomie (szerokość obrazu).

odpowiada za to taka transformacja:

```

1  self.transform = transforms.Compose([
2      transforms.Resize((384, 384)),
3      transforms.ToTensor(),
4  ])

```

Listing 1: Transforamcja zdjęć

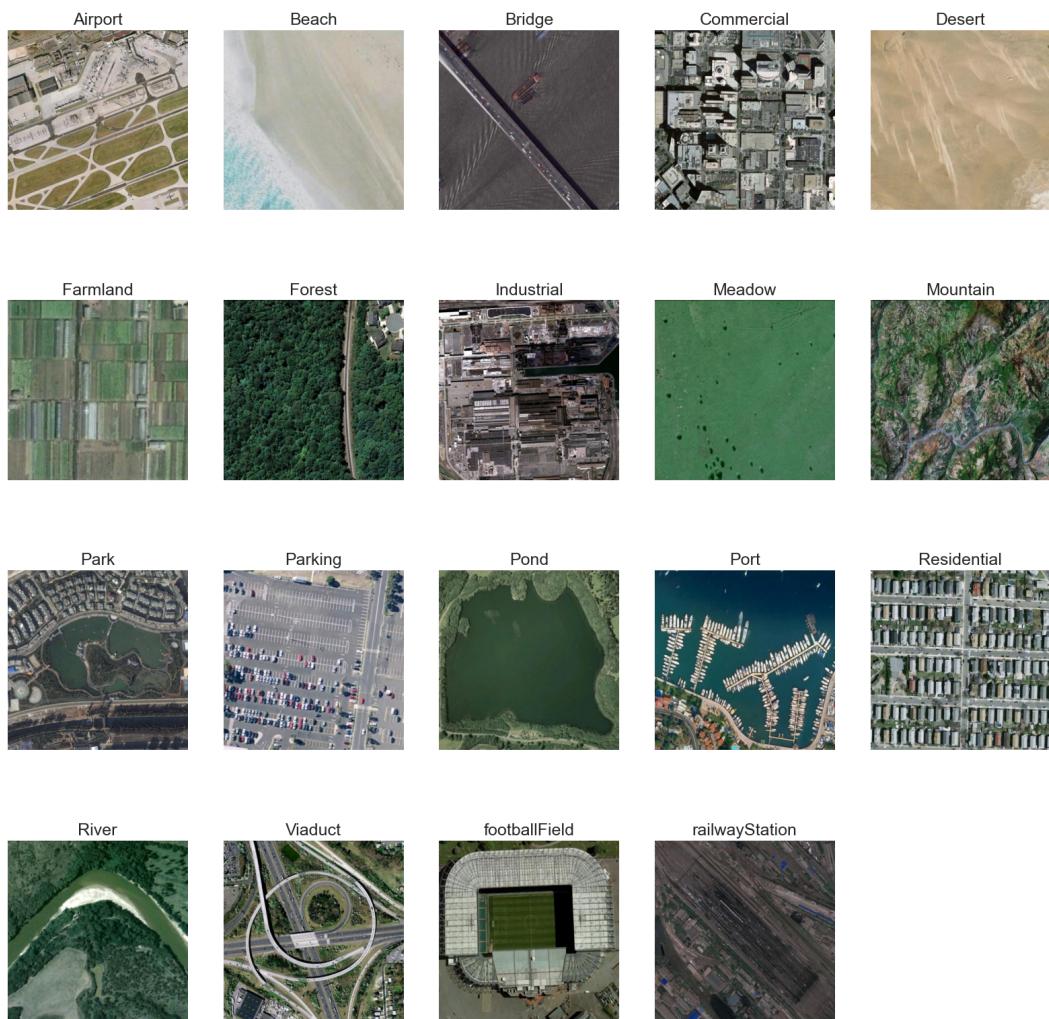
Typ danych przechowywanych w tensorach to `float32` z zakresu od 0 do 1 (po normalizacji przed użyciem w sieci).

przykład pojedynczego tensora pokazanie tylko wymiarów 2 i 3 :

```

tensor([[[0.2510, 0.2196, 0.2157, ..., 0.2392, 0.2275, 0.2196],
        [0.2980, 0.2667, 0.2549, ..., 0.2863, 0.2784, 0.2784],
        [0.3294, 0.3098, 0.3059, ..., 0.3333, 0.3294, 0.3373],
        ...,
        [0.5843, 0.4902, 0.4627, ..., 0.2980, 0.3059, 0.3176],
        [0.5569, 0.5020, 0.4667, ..., 0.3020, 0.3020, 0.3137],
        [0.5490, 0.5176, 0.4745, ..., 0.3098, 0.3059, 0.3059]],...)

```



Rysunek 2.4: Przykłady zdjęć

### 3. Przedstawienie oraz omówienie własnej architektury CNN

Architektura sieci Lstlisting 2 jest oparta na rozwiązaniach biblioteki PyTorch posiada ona 197 000 parametrów do nauki, 3 warstwy konwolucyjne , 1 warstwę neuronów ReLU, 1 warstwę neuronów linowych, 2 bloki Sequential gdzie każda z nich składa się z 2 warstw Relu oraz 2 warstw konwolucyjnych

```
1  class MyModel1(nn.Module):
2      def __init__(self, num_classes=19):
3          super().__init__()
4          self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1,
5          padding=1)
6          self.relu1 = nn.ReLU()
7          self.conv2 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
8          padding=1)
9          self.residual1 = self._make_residual_block(64, 64)
10         self.residual2 = self._make_residual_block(64, 64)
11         self.skip_connection = nn.Conv2d(128, 64, kernel_size=1,
12          stride=1)
13         self.max_pool = nn.MaxPool2d(kernel_size=2)
14         self.fc = nn.Linear(64, num_classes)
15         self._initialize_weights()
16     def forward(self, x):
17         out = self.conv1(x)
18         out = self.relu1(out)
19         out = self.conv2(out)
20         residual1 = self.residual1(out)
21         residual2 = self.residual2(residual1)
22         skip_out = self.skip_connection(torch.cat([residual1,
23             residual2], dim=1))
24         skip_out = self.max_pool(skip_out)
25         out = self.max_pool(out) + skip_out
26         out = out.mean(dim=(2, 3))
27         out = self.fc(out)
28         return out
29     def _make_residual_block(self, in_channels, out_channels):
30         return nn.Sequential(
31             nn.Conv2d(in_channels, out_channels, kernel_size=3,
32             stride=1, padding=1),
33             nn.ReLU(),
34             nn.Conv2d(out_channels, out_channels, kernel_size=3,
35             stride=1, padding=1),
36             nn.ReLU())
37     def _initialize_weights(self):
38         for m in self.modules():
39             if isinstance(m, nn.Conv2d):
40                 nn.init.xavier_uniform_(m.weight)
41                 if m.bias is not None:
42                     nn.init.constant_(m.bias, 0)
43             elif isinstance(m, nn.Linear):
44                 nn.init.xavier_uniform_(m.weight)
```

Listing 2: Struktura głębokiej sieci konwolucyjnej MyModel1

Klasa `MyModel1` reprezentuje model sieci neuronowej, która składa się z warstw konwolucyjnych, bloków powtórzeń, połączeń pomijających, warstw poolingowych oraz warstwy wyjściowej.

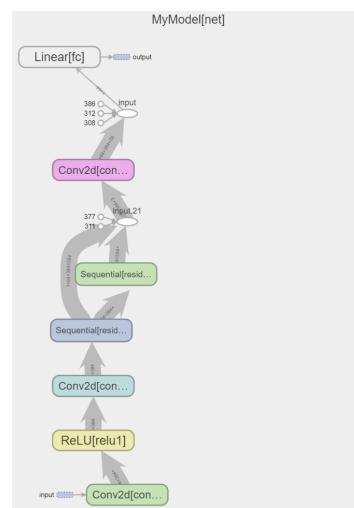
Struktura sieci:

- Warstwa konwolucyjna `conv1` o wejściowym kanale o rozmiarze 3, wyjściowym kanale o rozmiarze 64, jądrze o rozmiarze  $3 \times 3$ , kroku 1 i paddingu 1.
- Warstwa ReLU `relu1`.
- Warstwa konwolucyjna `conv2` o wejściowym kanale o rozmiarze 64, wyjściowym kanale o rozmiarze 64, jądrze o rozmiarze  $3 \times 3$ , kroku 1 i paddingu 1.
- Blok powtórzenia `residual1` składający się z dwóch warstw konwolucyjnych, z których każda ma wejściowy kanał o rozmiarze 64 i wyjściowy kanał o rozmiarze 64, jądro o rozmiarze  $3 \times 3$ , krok 1 i padding 1. Po każdej warstwie konwolucyjnej występuje warstwa ReLU.
- Blok powtórzenia `residual2` o takiej samej strukturze jak `residual1`.
- Połączenie pomijające `skip_connection` przy użyciu warstwy konwolucyjnej o wejściowym kanale o rozmiarze 128, wyjściowym kanale o rozmiarze 64 i jądrze o rozmiarze  $1 \times 1$ .
- Warstwa `MaxPool2d max_pool` o jądrze o rozmiarze  $2 \times 2$ .
- Warstwa wyjściowa `fc` typu `Linear` o wejściu o rozmiarze 64 i wyjściu o rozmiarze `num_classes`.

Działanie sieci:

- 1) Dane wejściowe  $x$  przechodzą przez warstwę konwolucyjną `conv1`.
- 2) Wynik jest poddawany funkcji aktywacji ReLU `relu1`.
- 3) Wynik przechodzi przez warstwę konwolucyjną `conv2`.
- 4) Wynik przechodzi przez blok powtórzenia `residual1`.

- 5) Wynik przechodzi przez blok powtórzenia `residual2`.
- 6) Wyniki `residual1` i `residual2` są konkatenowane wzduż wymiaru kanałów i przechodzą przez połaczenie pomijające `skip_connection`.
- 7) Wynik z warstwy `conv2` jest sumowany z wynikiem z połączenia pomijającego.
- 8) Wynik przechodzi przez warstwę poolingową `max_pool`.
- 9) Wynik jest poddawany operacji Global Average Pooling, obliczając średnią po wymiarach (2, 3).
- 10) Wynik przechodzi przez warstwę wyjściową `fc`.



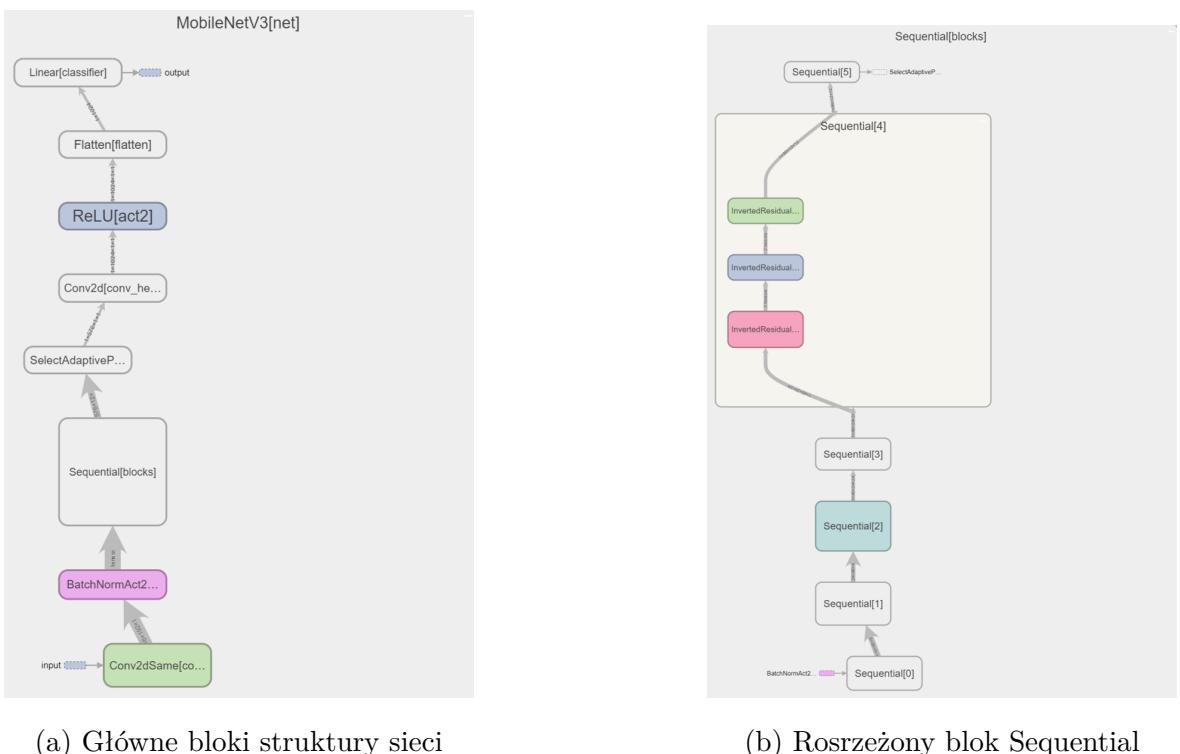
Rysunek 3.5: Schemat Blokowy Sieci MyModel

## **4. Przedstawienie oraz omówienie architektury sieci tf\_mobilenetv3 small minimal 100**

Sieć tf\_mobilenetv3\_small\_minimal\_100 to zredukowana wersja modelu MobileNetV3 w implementacji dla TensorFlow. Jest to model o architekturze sieci konwolucyjnej z milionem parametrów do uczenia, który został zoptymalizowany pod kątem wydajności i ma mniejszą liczbę parametrów niż pełna wersja MobileNetV3.

Główne cechy sieci tf\_mobilenetv3\_small\_minimal\_100 to:

- Warstwy konwolucyjne: Sieć wykorzystuje zestaw warstw konwolucyjnych o różnych rozmiarach jądra (3x3, 5x5 itp.) w celu ekstrakcji cech z obrazu wejściowego. Warstwy konwolucyjne są stosowane sekwencyjnie, tworząc głębsze reprezentacje obrazu.
- Bloki Bottleneck: Wykorzystuje bloki Bottleneck, które składają się z sekwencji warstw konwolucyjnych o mniejszej liczbie filtrów, aby zmniejszyć liczbę parametrów. Te bloki pomagają zachować wydajność i dokładność modelu przy ograniczonych zasobach obliczeniowych.
- Ekspresywność: Model ma na celu osiągnięcie wysokiej ekspresywności przy niskiej złożoności. Wykorzystuje różne techniki, takie jak konwolucje separowalne, tzw. "squeeze-and-excite"(SE) oraz aktywacje "hard-swish", aby zwiększyć reprezentatywność modelu.
- Minimalne rozmiary: Sieć tf\_mobilenetv3\_small\_minimal\_100 ma mniejszą liczbę parametrów niż pełna wersja MobileNetV3, co sprawia, że jest bardziej odpowiednia do zastosowań o ograniczonych zasobach obliczeniowych, takich jak urządzenia mobilne.
- Celem tej sieci jest dostarczenie efektywnego modelu o mniejszej liczbie parametrów, który może być wykorzystywany w zasobnościami środowiskach, jednocześnie zachowując odpowiednią dokładność w zadaniach klasyfikacji obrazów.



Rysunek 4.6: Struktura sieci tf\_mobilenetv3\_small\_minimal\_100

## 5. Omówienie klasy `LightningModule` oraz optymalizatora `ADAM`

Klasa `LightningModule` służy między innymi do rejsteracji Hiperparametrów sieci, obróbki danych wejściowych, oraz procesu uczenia się sieci.

Optymalizator Adam łączy w sobie zalety dwóch innych algorytmów optymalizacji: AdaGrad, który dostosowuje krok uczenia dla każdego parametru na podstawie historycznych gradientów, i RMSProp, który stosuje średnią ruchomą gradientu.

Kluczowe aspekty działania optymalizatora Adam są następujące:

- Średnia ruchoma gradientu (first moment): Algorytm Adam przechowuje średnią ruchomą gradientu, która jest obliczana na podstawie historycznych gradientów. Ta średnia ruchoma reprezentuje estymację momentu pierwszego (optymalizacja kierunku).
- Kwadrat średniej ruchomej gradientu (second moment): Algorytm Adam przechowuje średnią ruchomą kwadratu gradientu, która jest obliczana na podstawie historycznych kwadratów gradientów. Ta średnia ruchoma reprezentuje estymację momentu drugiego (optymalizacja kroku).
- Bias korekcyjny: Ze względu na inicjalne przesunięcie obliczeń średnich ruchomych, Adam stosuje bias korekcyjny w celu uwzględnienia tego efektu i poprawienia stabilności w początkowych iteracjach optymalizacji.
- Aktualizacja parametrów: Algorytm Adam aktualizuje parametry sieci na podstawie estymacji momentu pierwszego i momentu drugiego. Wykorzystuje również dodatkowy hiperparametr, tzw. współczynnik uczenia (learning rate), który kontroluje wielkość kroku aktualizacji.
- Działanie optymalizatora Adam może być podsumowane w kilku krokach: obliczanie gradientu dla aktualnych parametrów, obliczanie estymacji momentu pierwszego i momentu drugiego, uwzględnianie biasu korekcyjnego, obliczanie nowych wartości parametrów na podstawie estymacji momentów i współczynnika uczenia.

---

```

input :  $\gamma$  (lr),  $\beta_1, \beta_2$  (betas),  $\theta_0$  (params),  $f(\theta)$  (objective)
        $\lambda$  (weight decay), amsgrad, maximize
initialize :  $m_0 \leftarrow 0$  ( first moment),  $v_0 \leftarrow 0$  (second moment),  $\widehat{v}_0^{max} \leftarrow 0$ 

for  $t = 1$  to ... do
  if maximize :
     $g_t \leftarrow -\nabla_{\theta} f_t(\theta_{t-1})$ 
  else
     $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ 
  if  $\lambda \neq 0$ 
     $g_t \leftarrow g_t + \lambda \theta_{t-1}$ 
   $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ 
   $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ 
   $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ 
   $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ 
  if amsgrad
     $\widehat{v}_t^{max} \leftarrow \max(\widehat{v}_t^{max}, \widehat{v}_t)$ 
     $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t / (\sqrt{\widehat{v}_t^{max}} + \epsilon)$ 
  else
     $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ 


---


return  $\theta_t$ 

```

---

Rysunek 5.7: Matematyczny zapis Optymalizatora ADAM

## 6. Eksperyment I. badanie parametru rozmiaru wsadu oraz wielkości zdjęcia

Rozmiar wsadu (batch size) jest parametrem określającym liczbę próbek, które są przetwarzane jednocześnie podczas jednej iteracji algorytmu uczenia maszynowego. Zmiana rozmiaru wsadu może mieć wpływ na proces uczenia się z kilku powodów:

- Wydajność obliczeniowa: Większy rozmiar wsadu może przyspieszyć proces uczenia, ponieważ możliwe jest równoległe przetwarzanie większej liczby próbek na jednostce obliczeniowej. Dzieje się tak, ponieważ przetwarzanie wsadów odbywa się z wykorzystaniem efektywnych operacji macierzowych, które mogą być zoptymalizowane na poziomie sprzętowym.
- Stabilność uczenia: Przy mniejszych rozmiarach wsadu, gradienty obliczane na podstawie próbek są bardziej losowe i bardziej zmienne, co może prowadzić do większej niestabilności procesu uczenia. Większy rozmiar wsadu może przyczynić się do lepszego uczenia, ale może również prowadzić do nadwrażliwości na szum.

nić się do bardziej stabilnego uczenia poprzez wygładzenie gradientów i zmniejszenie wpływu pojedynczych próbek na aktualizację wag.

- Zużycie pamięci: Większy rozmiar wsadu wymaga większej ilości pamięci do przechowywania danych i gradientów. Jeśli dostępna pamięć jest ograniczona, konieczne może być dostosowanie rozmiaru wsadu tak, aby zmieścił się w dostępnej pamięci.
- Ogólna dokładność modelu: Wybór rozmiaru wsadu może mieć wpływ na ogólną dokładność modelu. Przy większych rozmiarach wsadu model może mieć lepszą zdolność uogólniania, ponieważ jest trenowany na większej liczbie różnorodnych przykładów. Jednak mniejszy rozmiar wsadu może prowadzić do bardziej szczegółowych aktualizacji wag, które mogą doprowadzić do osiągnięcia lepszej lokalnej dokładności.

Wielkość zdjęcia ma istotny wpływ na proces uczenia się w modelach opartych na sieciach neuronowych. Oto kilka aspektów, które należy wziąć pod uwagę:

- Ilość dostępnej informacji: Większe zdjęcia zawierają więcej pikseli, co oznacza większą ilość informacji. Model może zyskać dostęp do bardziej szczegółowych cech obrazu, co może przyczynić się do lepszej wydajności w zadaniach takich jak rozpoznawanie obiektów czy segmentacja.
- Złożoność obliczeniowa: Przetwarzanie większych obrazów wymaga większej mocy obliczeniowej. Modele uczące się na większych obrazach mogą być bardziej wymagające obliczeniowo i potrzebować większych zasobów, takich jak pamięć GPU i czas treningu.
- Overfitting: Jeśli model jest zbyt skomplikowany lub ma zbyt mało dostępnych danych treningowych w porównaniu do wielkości obrazów, istnieje ryzyko przeuczenia (overfittingu). Model może nauczyć się "zapamiętywać" konkretne obrazy treningowe zamiast generalizować cechy. W takich przypadkach można rozważyć zastosowanie technik regularyzacji, takich jak augmentacja danych lub wczytywanie obrazów w mniejszych rozmiarach.
- Efektywność obliczeniowa: Przetwarzanie większych obrazów może być bardziej czasochłonne, zarówno podczas treningu, jak i podczas wykonywania predykcji

na nowych danych. Dla niektórych zastosowań, takich jak detekcja obiektów w czasie rzeczywistym, może być konieczne dostosowanie rozmiaru zdjęcia do wymagań czasowych.

## 6.1. Metoda badawcza

Celem tego eksperymentu jest znalezienie optymalnych parametrów dla sieci neuronowych poprzez zmianę rozmiaru partii (batch size) oraz rozmiaru obrazu. Prze prowadzony zostanie szereg testów, gdzie batch size będzie przyjmował wartości: 2, 4, 8, 16, 32, 64, 128, 256, a rozmiar obrazu będzie wynosił: 10, 65, 121, 177, 232, 288, 344, 400.

W trakcie każdego testu, oba modele (netV3 i MyModel) zostaną wytrenowane przy użyciu odpowiednich wartości batch size i rozmiaru obrazu. Następnie, dokładność (accuracy) oraz funkcja straty (loss) zostaną zmierzone i zarejestrowane .

```
1 lr=1e-4
2 max_epochs=10
3 for batch_size in vec_batch_size:
4     for img_size in vec_img_size:
5
6         model = ClasyficatorCNN(slect_model_timm = select_model ,
7             size = img_size ,batch_size=batch_size ,lr=lr, pretrained=
8             pretrained)
9         logger = TensorBoardLogger("lightning_logs", name=
10             select_model+ "img_size = " + str(img_size)+ "batch_size = "+
11             str(batch_size), default_hp_metric=False)
12         trainer = Trainer(
13             max_epochs=max_epochs ,
14             logger=logger)
15         trainer.fit(model)
16
17         vec_loss = np.append(vec_loss , model.loss)
18         vec_acc = np.append(vec_acc , model.valid_acc)
19         vec_pres = np.append(vec_pres , model.val_precision)
20         vec_acu = np.append(vec_acu , model.val_ACU)
21         vac_sens = np.append(vac_sens , model.val_specificitys)
22
23
24         model = ClasyficatorCNN(slect_model = MyModel() , size =
25             img_size ,batch_size = batch_size,lr=lr, pretrained=
26             pretrained)
27         logger = TensorBoardLogger("lightning_logs", name=
28             select_model+ "img_size = " + str(img_size)+ "batch_size = "+
29             str(batch_size) , default_hp_metric=False)
30         trainer = Trainer(
31             max_epochs=max_epochs ,
32             logger=logger)
33         trainer.fit(model)
34         vec_loss_Mymodel = np.append(vec_loss , model.loss)
```

```

27     vec_acc_Mymodel = np.append(vec_acc , model.valid_acc)
28     vec_pres_Mymodel = np.append(vec_pres , model.
29     val_precision)
30     vec_acu_Mymodel = np.append(vec_acu , model.val_ACU)
31     vac_sens_Mymodel = np.append(vac_sens , model.
32     val_specificity)

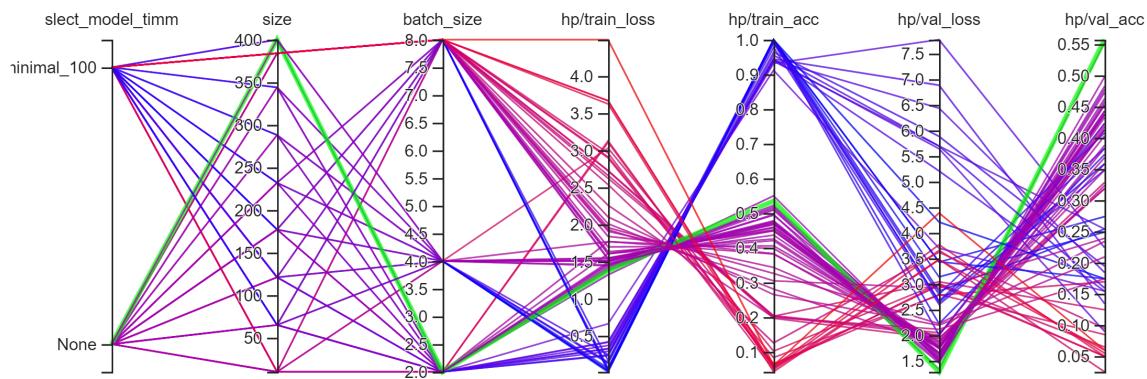
```

Listing 3: Kod do eksperymentu II

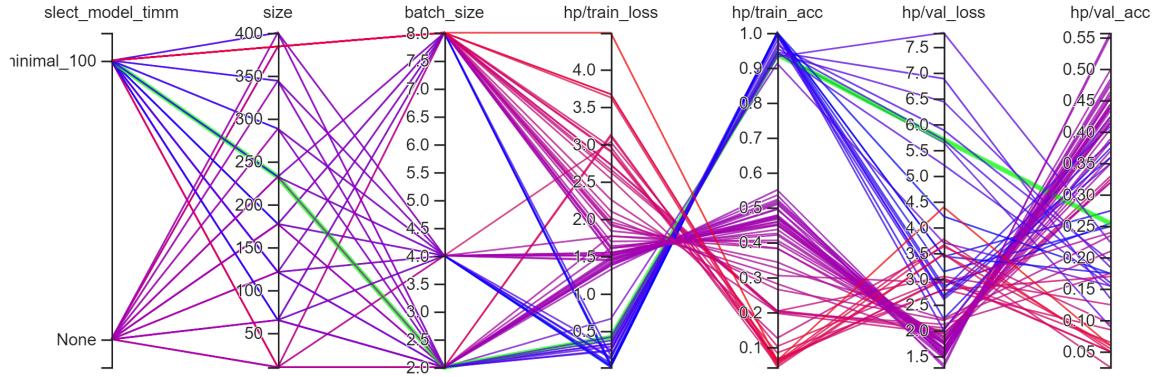
## 6.2. Wyniki

Nie udało się przeprowadzić całego testu został on przerwany na wielkość wsadu = 8. Przerwanie to uniemożliwia optymalne dobranie parametrów, jednak można zauważać kilka interesujących zjawisk. Na przykład, mały rozmiar wsadu wpływa na szybkie przeuczenie modelu, niezależnie od rozmiaru zdjęcia.

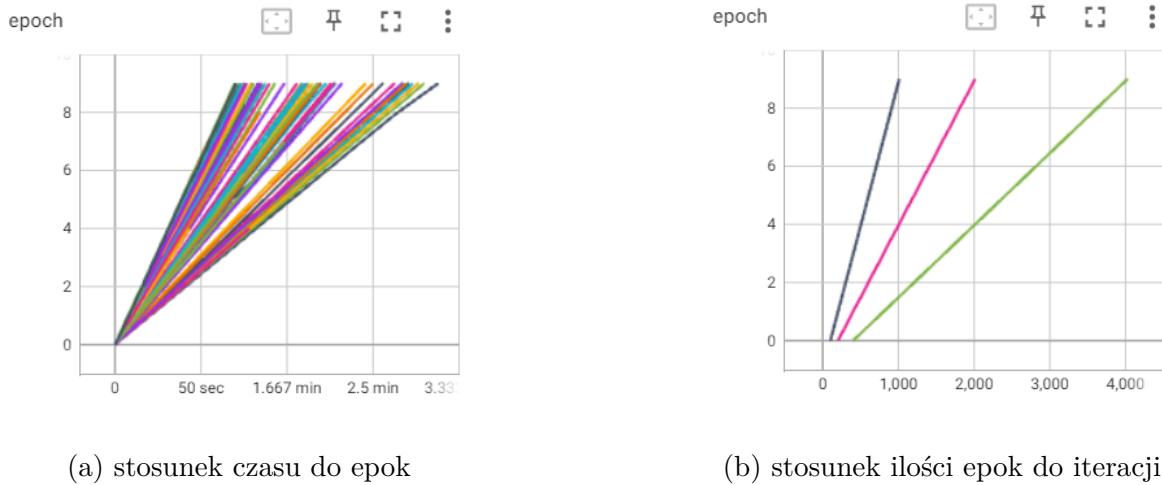
Ponadto, można zauważyc różnicę w liczbie potrzebnych iteracji do przejścia przez jedną epokę w zależności od wartości batch size. Przy tej samej liczbie epok, ilość iteracji rośnie odwrotnie proporcjonalnie do rozmiaru wsadu.



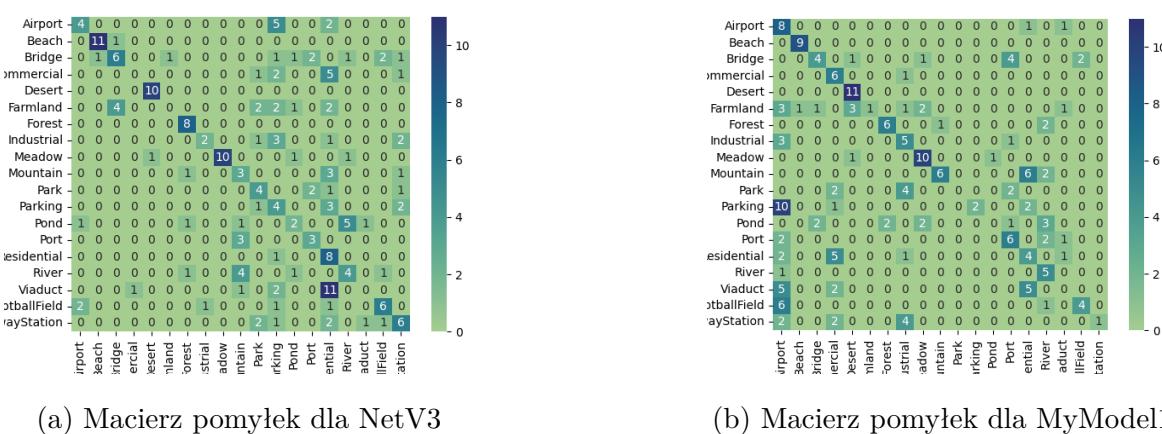
Rysunek 6.8: Najlepsze zbadane rozwiązania dla MyModel1



Rysunek 6.9: Nalepsze możliwe rozwiązanie dla NetV3



Rysunek 6.10: Zestawienie szybkości nauczania



Rysunek 6.11: zstawienie macierzy pomyłek

## 7. Eksperyment II. Uczenie i porównanie modeli po 200 epokach

W celu porównania skuteczności modeli MyModel i NetV3, przeprowadzimy eksperyment polegający na uczeniu obu modeli przez 200 epok. Porównamy wyniki dla modelu MyModel oraz modelu NetV3

### 7.1. Metoda badawcza

Na podstawie wyników z porzedniego zostały dobrane parametry sieci dla architektury Mymodel1 batch\_size = 8 img\_size = 400, dla archtektury tf\_mobilenetv3\_small\_minimal\_100 batch\_size = 2 img\_size = 232. Po przeprowadzenia procesu uczenia Listing 4 zostanie przepowadzona anaaliza wyników szybkości (ilości iteracji podczas uczenia) oraz taki hiper parametrów jak dokładność, czułość , precyzja, oraz pole pod krrzywą ROC

```
1    lr = 3e-4
2    batch_size = 2
3    model = ClasyficatorCNN(slect_model= MyModel1(), size = 400
, batch_size=8 ,lr=lr)
4    logger = TensorBoardLogger("lightning_logs", name="My model
5      bbatch_size:" , default_hp_metric=True)
6    trainer = Trainer(
7        max_epochs=200 ,
8        logger=logger
9    )
10   trainer.fit(model)

11   select_model = 'tf_mobilenetv3_small_minimal_100'
12   model = ClasyficatorCNN(slect_model_timm = select_model ,
13   size = 232 ,batch_size=2,lr=lr,pretrained=False)
14   logger = TensorBoardLogger("lightning_logs", name=
15   select_model , default_hp_metric=False)
16   trainer = Trainer(
17       max_epochs=200 ,
18       logger=logger
19   )
20   trainer.fit(model)

21   model = ClasyficatorCNN(slect_model_timm = Kod do
22   eksperymentu II , size = 232 ,batch_size=2,lr=lr,pretrained=
23   True)
24   logger = TensorBoardLogger("lightning_logs", name=
25   select_model , default_hp_metric=False)
26   trainer = Trainer(
```

---

Listing 4: Kod do eksperymentu II.

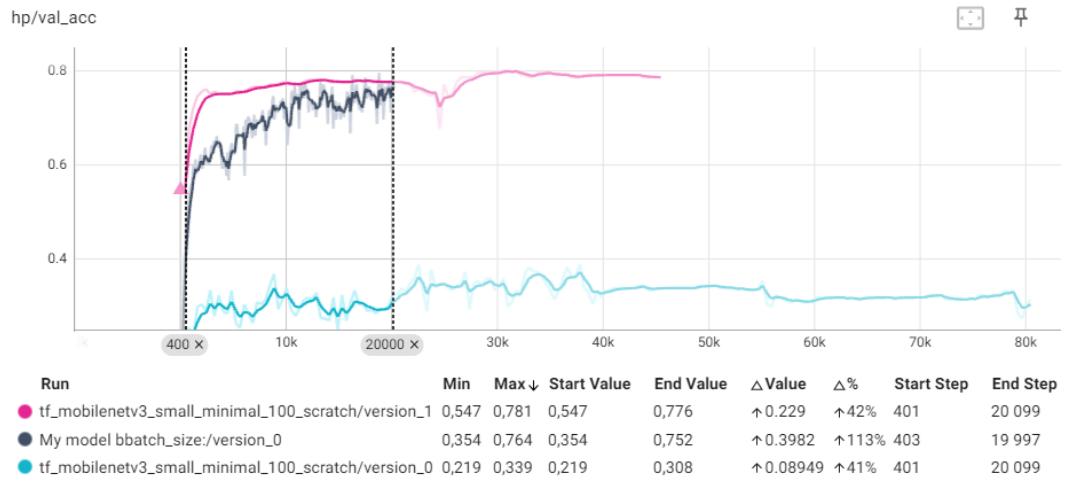
## 7.2. Wyniki

Podczas procesu uczenia zastosowano trzy modele sieci oparte na architekturze `tf_mobilenetv3_small_minimal_100`. Analiza wyników pokazała, że tylko dwa z tych modeli osiągnęły dokładność na poziomie bliskim 0.8, natomiast trzeci model był dotknięty efektem przeuczenia.

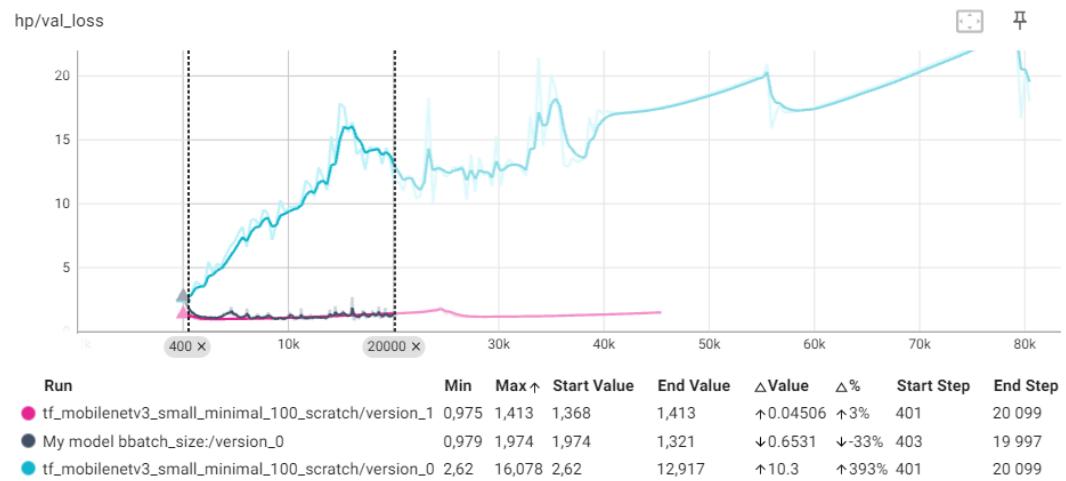
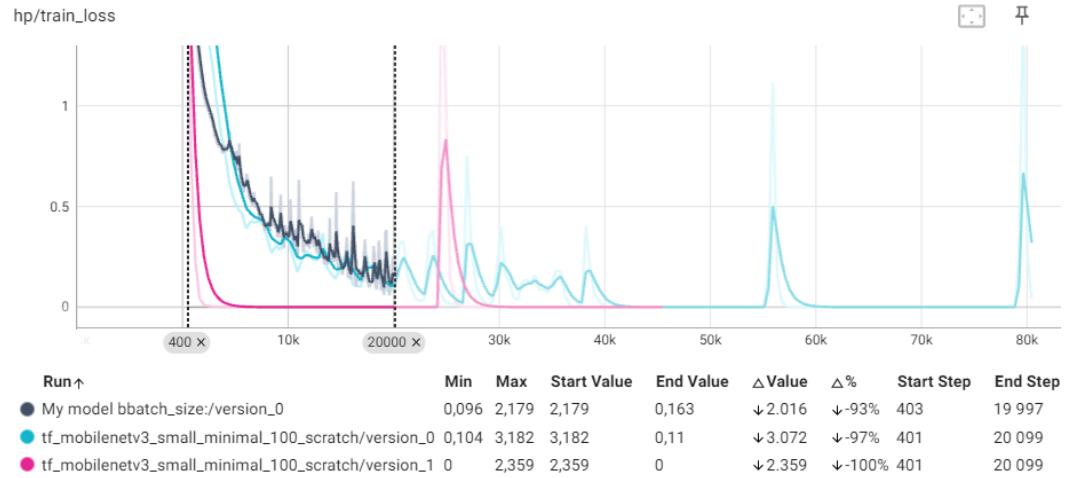
Okazało się, że modele oparte na architekturze `tf_mobilenetv3_small_minimal_100` nie były w stanie wykorzystać w pełni swojego potencjału, co można przypisać złemu doborowi parametrów lub zbyt małemu zbiorowi uczącemu. W przypadku modelu `version_0`, nieprzetrenowanego, nie był w stanie uogólniać cech, skupiając się zbyt mocno na dopasowaniu do danych uczących. Wskazuje na to porównanie funkcji błędów dla danych uczących i testowych. Aby rozwiązać ten problem, można by rozważyć zwiększenie rozmiaru zbioru uczącego.

Drugi model, `version_1`, oparty na architekturze `tf_mobilenetv3_small_minimal_100`, okazał się przetrenowany, co oznacza, że był w stanie bardzo szybko osiągnąć wysoką dokładność na danych uczących. Niemniej jednak, istnieje ryzyko, że taki model może mieć trudności z ogólną zdolnością do uogólniania na nowe dane spoza zbioru uczącego.

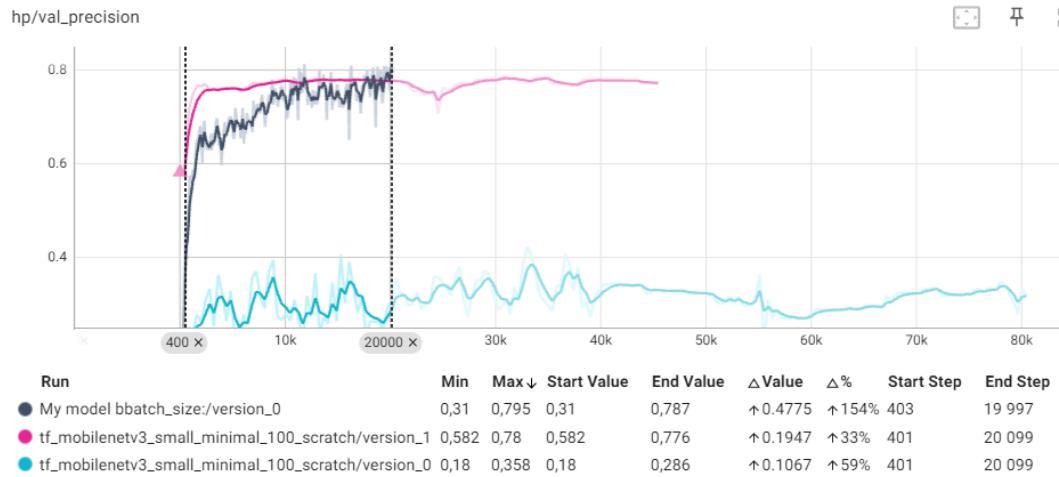
Trzeci model, oparty na własnej architekturze `MyModel1`, również osiągnął dobre wyniki podczas procesu uczenia, ale jego tempo nauki było znacznie wolniejsze. Mimo to, model ten zdolny był efektywnie uczyć się i osiągnąć wysoką dokładność.



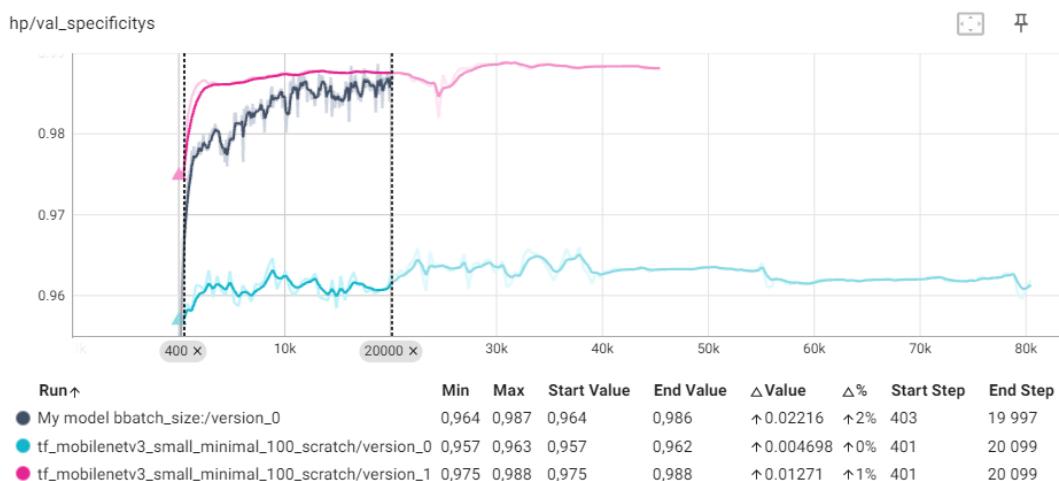
Rysunek 7.12: Zestawienie dokładności poszczególnych modeli



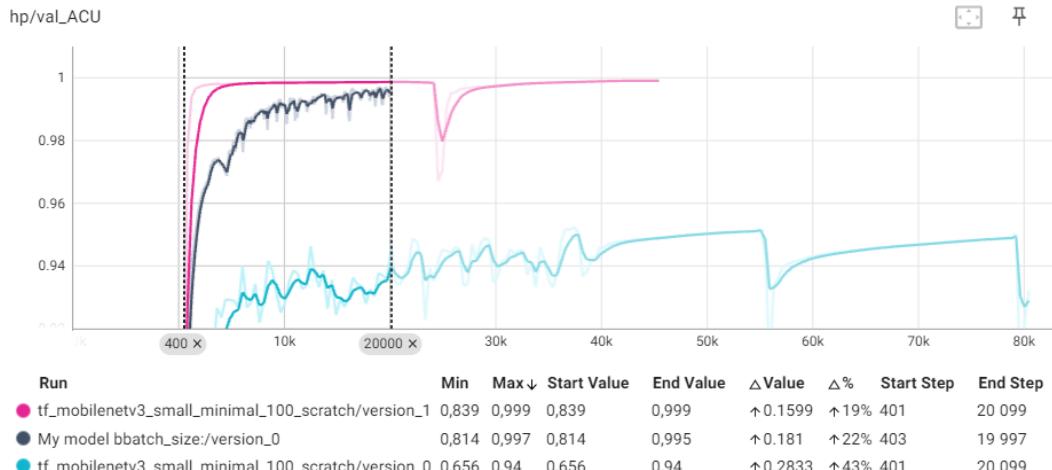
Rysunek 7.13: Porównanie wykresów błędów dla danych validacyjnych oraz treningowych



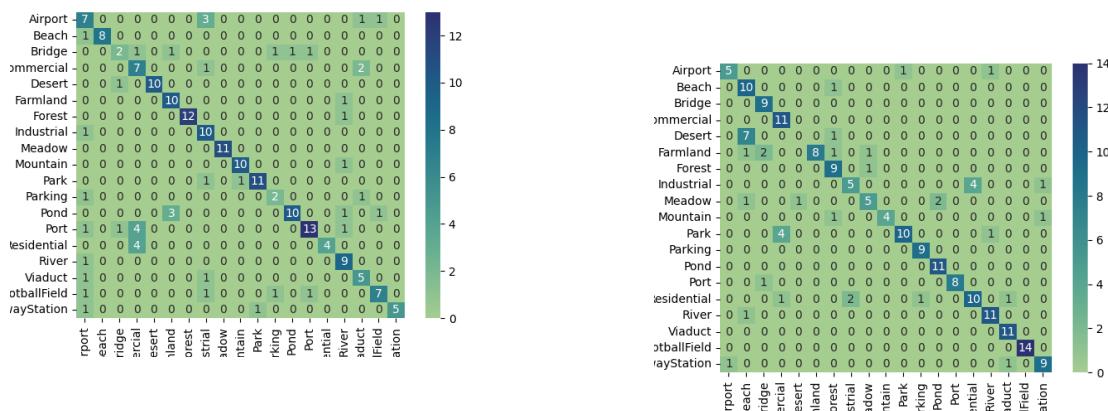
Rysunek 7.14: Zestawienie precyzji poszczególnych modeli



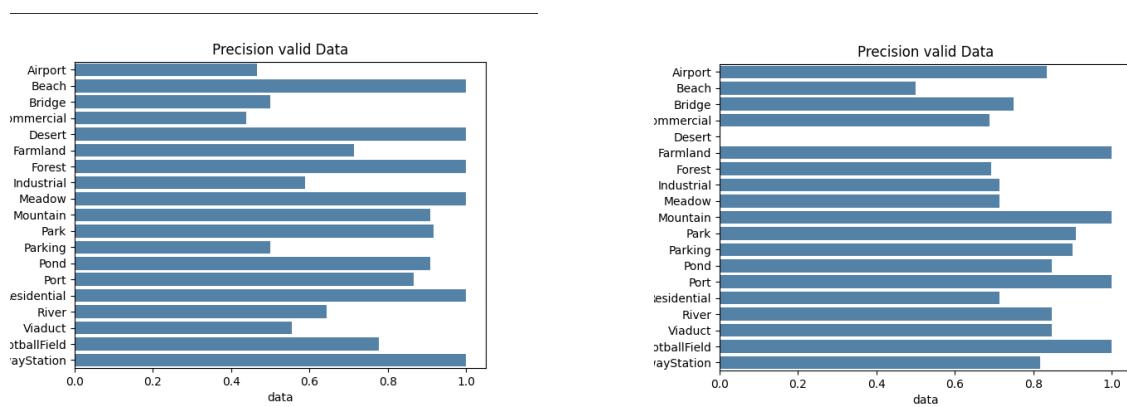
Rysunek 7.15: Zestawienie czułości poszczególnych modeli



Rysunek 7.16: Zestawienie wartości pola pod krzywą ROC poszczególnych modeli



Rysunek 7.17: zstwawienie macierzy pomyłek



Rysunek 7.18: zstwawienie wykresów precyzji dla poszczególnych klas

## 8. Podsumowanie

Niniejsza praca miała na celu zgłębienie problematyki i rozważenie różnych rozwiązań związanych z głębkimi sieciami konwolucyjnymi (CNN) w kontekście klasyfikacji obrazów. W toku badań skupiono się na problemie zanikającego gradientu, który często utrudnia efektywne uczenie się w głębkich sieciach. Przedstawiono i omówiono różnorodne techniki mające na celu przeciwdziałanie temu zjawisku, takie jak stosowanie funkcji aktywacji ReLU, inicjalizacja wag przy użyciu metody Xavier/Glorot, normalizacja danych lub warstw oraz wykorzystanie architektur sieci z połączonymi pomijającymi.

Funkcja aktywacji ReLU okazała się popularnym wyborem w głębkich sieciach neuronowych, ze względu na jej zdolność do skutecznego rozwiązywania problemu zanikającego gradientu. Metoda inicjalizacji wag za pomocą Xavier/Glorot umożliwia odpowiednie skalowanie wag na początku procesu uczenia, zapobiegając tym samym zanikaniu gradientu. Wykorzystanie normalizacji danych lub warstw przyczynia się do zmniejszenia zakresu wartości wejściowych i ułatwia proces uczenia. Architektury sieci z połączonymi pomijającymi umożliwiają przekazywanie informacji o niskopoziomowych cechach, co sprzyja skutecznemu uczeniu sieci.

W ramach pracy zaprezentowano również dwie konkretne architektury sieci: własną architekturę CNN oraz architekturę sieci tf mobilenetv3 small minimal 100. Szczegółowo omówiono kluczowe cechy tych struktur i ich zastosowanie w kontekście klasyfikacji obrazów.

Przeprowadzono również dwa eksperymenty, których celem było zbadanie wpływu różnych parametrów na skuteczność uczenia sieci. Pierwszy eksperiment koncentrował się na badaniu rozmiaru wsadu oraz rozmiaru zdjęć, podczas gdy drugi eksperiment porównywał modele po 200 epokach uczenia. Wyniki eksperymentów dostarczyły istotnych informacji na temat optymalnych parametrów oraz skuteczności uczenia sieci.

W pracy omówiono również klasę LightningModule oraz optymalizator ADAM, które zostały wykorzystane w procesie uczenia sieci.

Do przeprowadzenia badań wykorzystano zbiór danych WHU-RS19, który zawierał różne klasy obiektów na zdjęciach. Przeprowadzono odpowiednie przetwarzanie danych, takie jak zmiana rozmiaru zdjęć i konwersja ich do formatu tensorów, aby umożliwić efektywne uczenie sieci.

W trakcie przeprowadzanych eksperymentów zastosowano dwa modele: netV3 oraz MyModel. Każdy z modeli był trenowany przy użyciu określonych wartości batch size i rozmiaru obrazu. Celem eksperymentów było zmierzenie dokładności (accuracy) oraz funkcji straty (loss) i zbadanie wpływu różnych czynników na proces uczenia.

Niestety, nie udało się przeprowadzić pełnego testu, ponieważ został on przerwany przy batch size równym 8. Przerwanie eksperymentu uniemożliwiło optymalne dobranie parametrów, jednakże obserwowane zjawiska są nadal interesujące. Na przykład, mały rozmiar batch size prowadził do szybkiego przeuczenia modelu, niezależnie od rozmiaru zdjęcia.

Dodatkowo, zaobserwowano różnicę w liczbie iteracji potrzebnych do przejścia przez jedną epokę w zależności od wartości batch size. Przy tej samej liczbie epok, ilość iteracji wzrastała odwrotnie proporcjonalnie do rozmiaru batch size.

W przypadku trzeciego modelu, opartego na własnej architekturze MyModel1, osiągnięto dobre wyniki podczas procesu uczenia, ale tempo nauki było znacznie wolniejsze. Mimo to, model ten był zdolny do skutecznego uczenia się i osiągnięcia wysokiej dokładności.

## **Literatura**

- [1] <https://pytorch.org/docs/stable/index.html>. Dostęp 29.05.2023.
- [2] <https://lightning.ai/docs/pytorch/stable/>. Dostęp 29.05.2023.
- [3] <https://lightning.ai/docs/pytorch/latest/>. Dostęp 29.05.2023.
- [4] Kluska j.: Prezentacja z cyklu wykładów S.I. pt. Classifiers assessment methods ,Rzeszow University of Technology.