Metody rozpoznawania obrazów
Zadanie 3 – raport

Mateusz Kocot

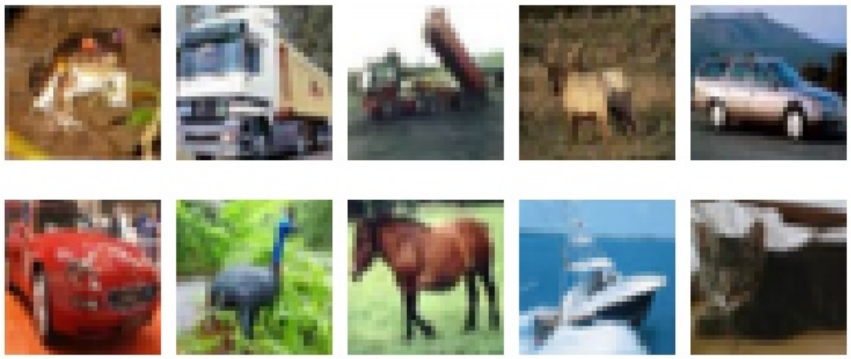
4 listopada 2021

Spis treści

1 Zbiór danych – Cifar10

1.1 Wczytanie zbioru

Cifar10 zawiera dane podzielone na 10 dość zróżnicowanych klas, w tym samoloty, koty, jelenie czy statki. Różnorodność tą widać już w przypadku pierwszych 10 obrazów (rys. 1). Co ważne, obiekty znajdują się w centrach obrazków, co znacząco ułatwia klasyfikację.



Rys. 1: Próbka 10 obrazów ze zbioru Cifar10

1.2 One hot encoding

W celu zamiany etykiet na wektory typu one hot wykorzystałem funkcję `tf.one_hot`, która przyjmuje etykiety oraz rozmiar końcowych wektorów.

2 Minimalna architektura

2.1 Implementacja

Do tego modelu, trudniejszego niż w zadaniu pierwszym, lecz wciąż prostego, idealnie nadała się klasa `tf.keras.Sequential`. Większość warstw, które użyłem, jest typowa, natomiast do przeskalowania wartości pikseli użyłem warstwy `Lambda` z funkcją dzielącą przez 255. Później zdałem sobie sprawę z istnienia warstwy `Rescaling`, ale nie zmieniałem już implementacji. Cały model wygląda następująco:

```
from tensorflow.keras import layers as tf_layers

model_1 = tf.keras.Sequential(layers=[
    tf.keras.Input(shape=(32, 32, 3)),
    tf_layers.Lambda(lambda x: x / 255), # normalizacja
    tf_layers.Conv2D(5, (3, 3), padding='same', activation='sigmoid'),
    tf_layers.Conv2D(5, (3, 3), padding='same', activation='sigmoid'),
    tf_layers.MaxPooling2D(pool_size=(8, 8)),
    tf_layers.Flatten(),
    tf_layers.Dense(10, activation='softmax')
], name='model_1')
```

Tak prosty model ma już 1180 warstw.

Przed rozpoczęciem trenowania podałem na wejście kilka zdjęć. Co ciekawe, uzyskane wektory prawdopodobieństw są bardzo podobne:

```
[0.3102289 , 0.04388418, 0.09348349, 0.11833623, 0.03337299,
    0.05916024, 0.05914203, 0.04586514, 0.13769099, 0.09883579],
[0.30853188, 0.04353305, 0.09168533, 0.12048813, 0.03336096,
    0.05900173, 0.05981053, 0.04543867, 0.13932769, 0.098822  ],
[0.31115592, 0.04354907, 0.09296943, 0.11759286, 0.03347881,
    0.05971997, 0.05897107, 0.04638271, 0.13820602, 0.09797422],
[0.30740398, 0.04365538, 0.09245769, 0.11884824, 0.03357116,
    0.05998491, 0.0596593 , 0.0460469 , 0.13896216, 0.09941032]
```

Wygląda na to, że domyślnie wagi modelu ustawione są w taki sposób, że wejście nie ma dużego wpływu na wyjście.

2.2 Trenowanie

Najpierw przypisałem modelowi odpowiednią metodę optymalizacji oraz funkcję straty. Wskazałem także, by po każdej epoce była liczona metryka *accuracy*:

```
model.compile(optimizer=tf.keras.optimizers.SGD(
                    learning_rate=0.001, momentum=0.9),
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=[tf.keras.metrics.CategoricalAccuracy()])
```

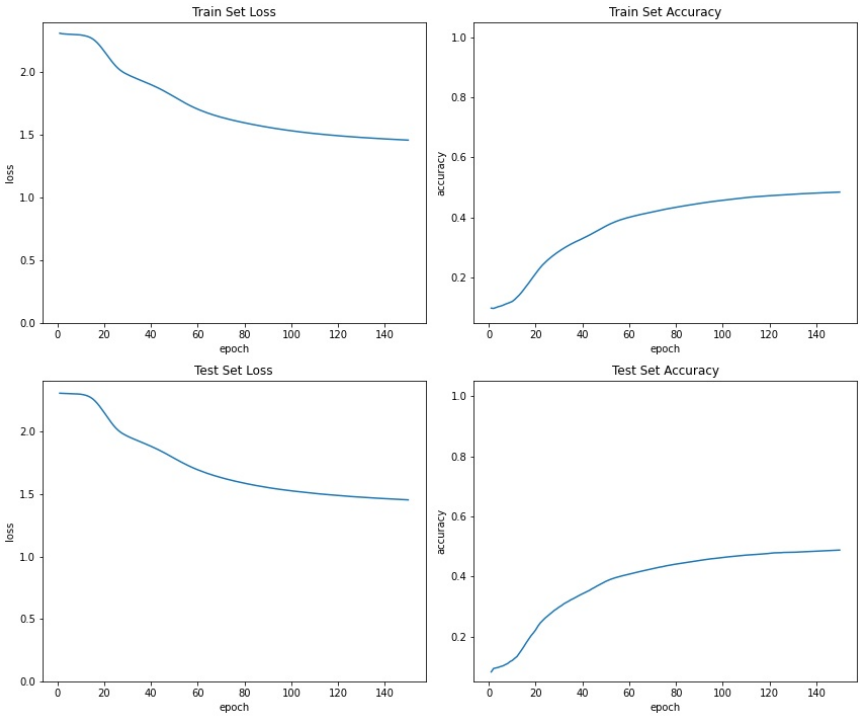
Następnie model został wytrenowany:

```
model.fit(train_images,
          train_labels_one_hot,
          batch_size=64,
          epochs=150,
          validation_data=(test_images, test_labels_one_hot))
```

Dzięki dodaniu `validation_data`, statystki po każdej epoce policzone zostały także na zbiorze testowym.

Model został wytrenowany w **435 s** co odpowiada średnio **2.9 s** na epokę. Warto zaznaczyć, że przed zmianą jednostki obliczeniowej z CPU na GPU czasy były kilkunastokrotnie większe. Odpowiednie wykresy przedstawiłem na rys. 2. W tym jak i wszystkich kolejnych przypadkach zastosowałem średnią kroczącą (średnia z ostatnich 10 wyników).

Po wytrenowaniu, zwracane prawdopodobieństwa mają już większy sens i widać, które klasy są najbardziej prawdopodobne. Oczywiście jakość predykcji pozostawia (jeszcze) sporo do życzenia, a większość obrazów nie jest klasyfikowana z prawdopodobieństwem większym niż 50%.



Rys. 2: Wyniki minimalnego modelu

3 Ewolucja

3.1 Zwiększenie liczby filtrów w warstwach konwolucyjnych

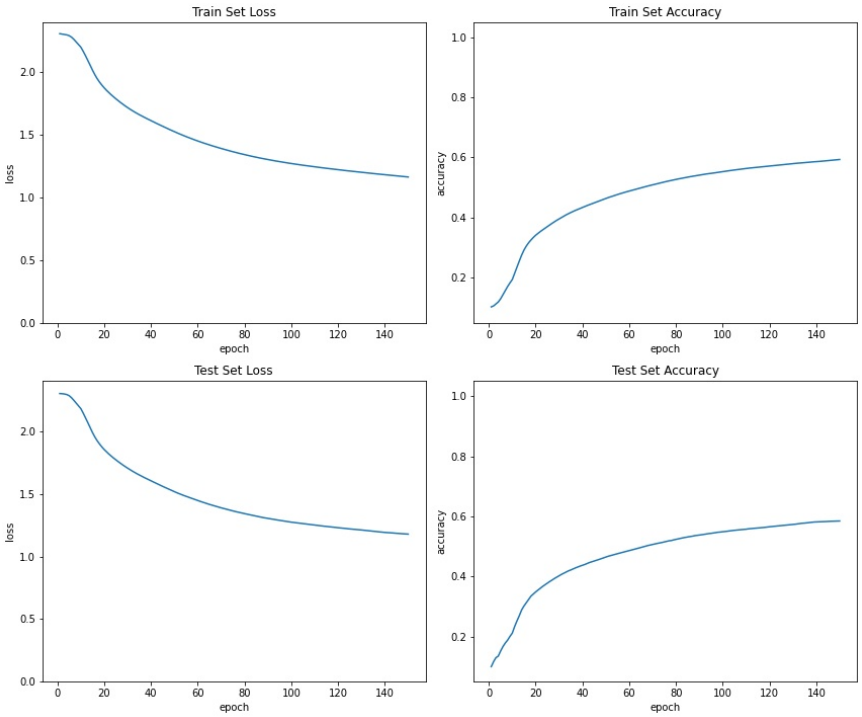
Po zwiększeniu liczby filtrów w warstwach konwolucyjnych liczba parametrów znacząco wzrosła – z 1180 do 7390. Wzrósł także czas trenowania. Tym razem cały trening zajął **501 s**, co daje średnio **3.34 s** na epokę. Wyniki przedstawiłem na rys. 3. Nie można odnotować jednak znaczącej poprawy.

3.2 Modyfikacja bloku konwolucyjnego

Przygotowałem funkcję zwracającą cały blok konwolucyjny:

```
def convolution_block(n_filters, activation='relu'):
    return [tf_layers.Conv2D(n_filters, (3, 3),
                             padding='same',
                             activation=activation),
            tf_layers.Conv2D(n_filters, (3, 3),
                             padding='same',
                             activation=activation),
            tf_layers.MaxPooling2D(pool_size=(2, 2))]
```

Oprócz parametru określającego liczbę filtrów `n_filters`, przyszłościowo dodałem także parametr `activation`. Na razie jednak będzie on ustawiany na `'sigmoid'`.



Rys. 3: Wyniki modelu po zwiększeniu liczby filtrów w warstwach konwolucyjnych

3.2.1 Dwa bloki konwolucyjne

Mając funkcję `convolution_block`, zmodyfikowałem model:

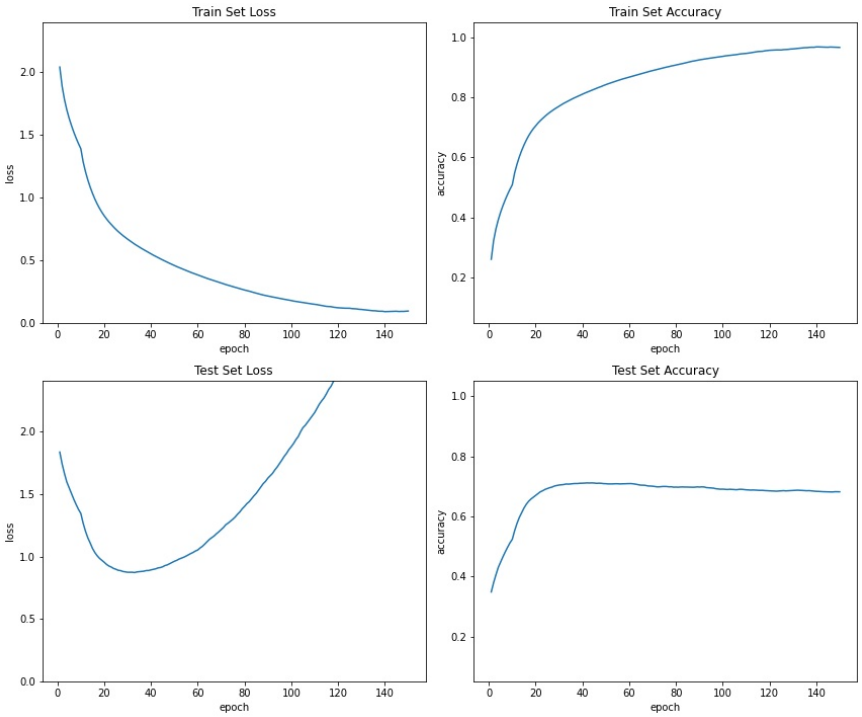
```
model_3_1 = tf.keras.Sequential(layers=[
    tf.keras.Input(shape=(32, 32, 3)),
    tf_layers.Lambda(lambda x: x / 255),
    *convolution_block(20, 'sigmoid'),
    *convolution_block(40, 'sigmoid'),
    tf_layers.Flatten(),
    tf_layers.Dense(10, activation='softmax')
], name='model_3_1')
```

Uruchomiłem 15 epok treningu nowego modelu. Efekty nie wyglądają jednak obiecująco. Wartości *accuracy* po 15 epokach są gorsze niż w przypadku poprzedniej sieci na tym samym etapie.

3.2.2 Zmiana aktywacji na ReLU

Wyniki znacząco powinno polepszyć zastosowanie ReLU. Jedyna zmiana w kolejnym modelu to usunięcie parametru `'sigmoid'` w `convolution_block`, dzięki czemu jest on ustawiany na domyślną wartość, tj. `'relu'`. Wyniki przedstawiłem na rys. 4.

Widać znaczący wzrostu *accuracy* (ok. 97%!) dla zbioru treningowego. Niestety, potężne jest także przetrenowanie skutkujące overfittingiem, co w konsekwencji powoduje spadek



Rys. 4: Wyniki modelu po zmianie większości aktywacji na ReLU

accuracy zbioru testowego mniej więcej od 30 epoki.

Czas trenowania ponownie wzrósł. Tym razem wyniósł **647 s**, tj. ok. **4.31 s** na epokę.

3.2.3 Kolejne bloki – 20, 40, 80, 160

Na tym etapie, dodanie kolejnych bloków nie usprawniło predykcji dla zbioru testowego, a wręcz spowodowało szybsze i bardziej efektowne przetrenowanie (rys. 5).

Czas trenowania tym razem wyniósł **954 s**, czyli **6.36 s** na epokę.

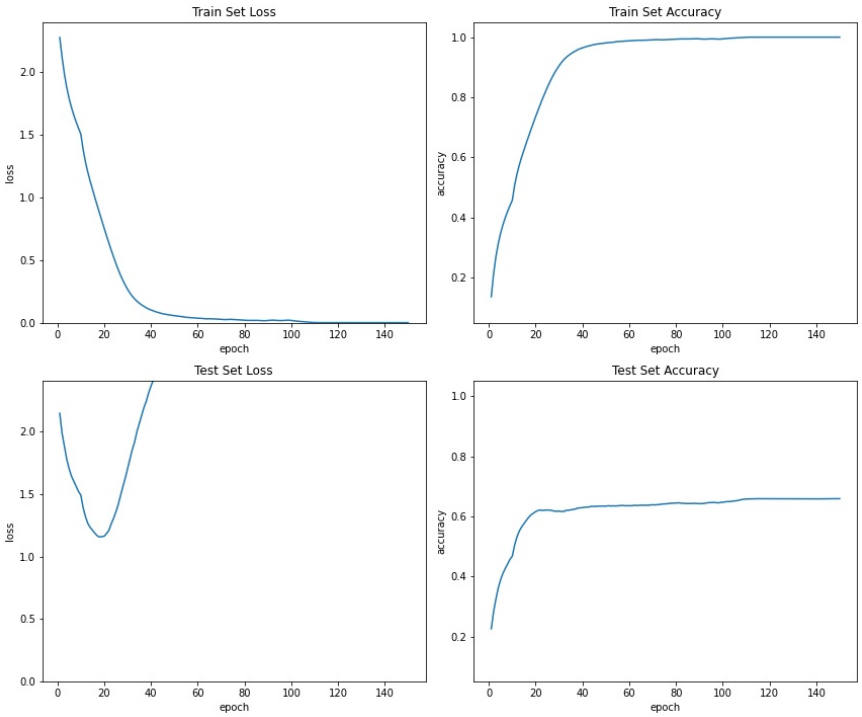
3.3 Batch normalization

Po każdej warstwie konwolucyjnej w bloku dodałem `tf_layers.BatchNormalization()`. Wprowadzenie warstw *batch normalization* wspomogło trenowanie. Przetrenowanie jest tu mniej widoczne niż w poprzednim modelu, a ostateczne `accuracy` na zbiorze testowym jest nieco większe (rys. 6).

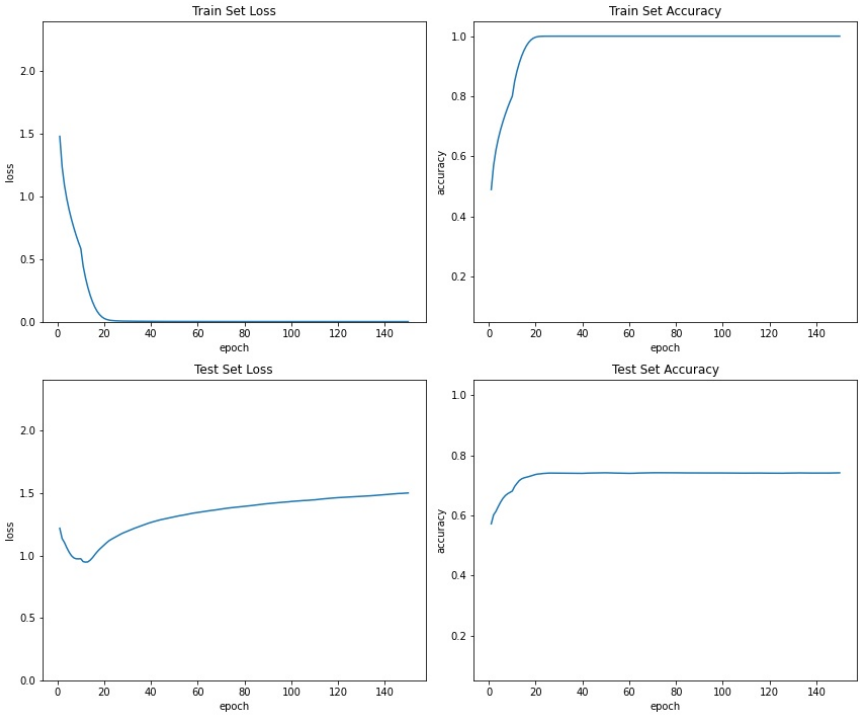
Czas trenowania wyniósł **1142 s**, co daje ok. **7.61 s** na epokę.

3.4 Dropout

Warstwę *dropout* dodałem na końcu każdego bloku konwolucyjnego modyfikując po raz kolejny funkcję tworzącą te bloki poprzez dodanie na końcu `tf_layers.Dropout(dropout_rate)`.



Rys. 5: Wyniki modelu po dołożeniu kolejnych dwóch bloków konwolucyjnych

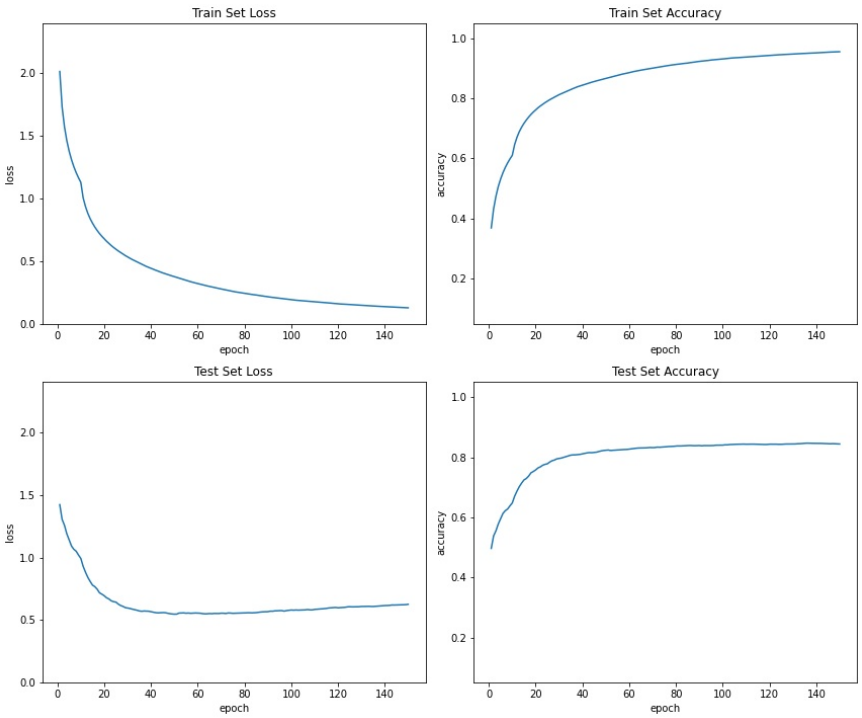


Rys. 6: Wyniki modelu po dodaniu warstw *batch normalization*

`dropout_rate` ustawiłem na 0.1, 0.2, 0.3 i 0.4 odpowiednio dla kolejnych bloków konwolu-

cyjnych

Dodanie warstw `dropout` w znaczący sposób ograniczyło przetrenowanie oraz zwiększyło jakość klasyfikacji na zbiorze testowym do ok. 84%. Wyniki uczenia przedstawiłem na rys. 7.



Rys. 7: Wyniki modelu po dodaniu warstw *dropout*

Czas trenowania wyniósł **1233 s**, co daje ok. **8.22 s** na epokę.

3.5 GAP

Zaimplementowałem ostateczną wersję funkcji tworzącej blok konwolucyjny dodając możliwość zamiany warstwy `MaxPooling2D` na warstwę `GlobalAveragePooling2D`. Funkcja wygląda teraz następująco:

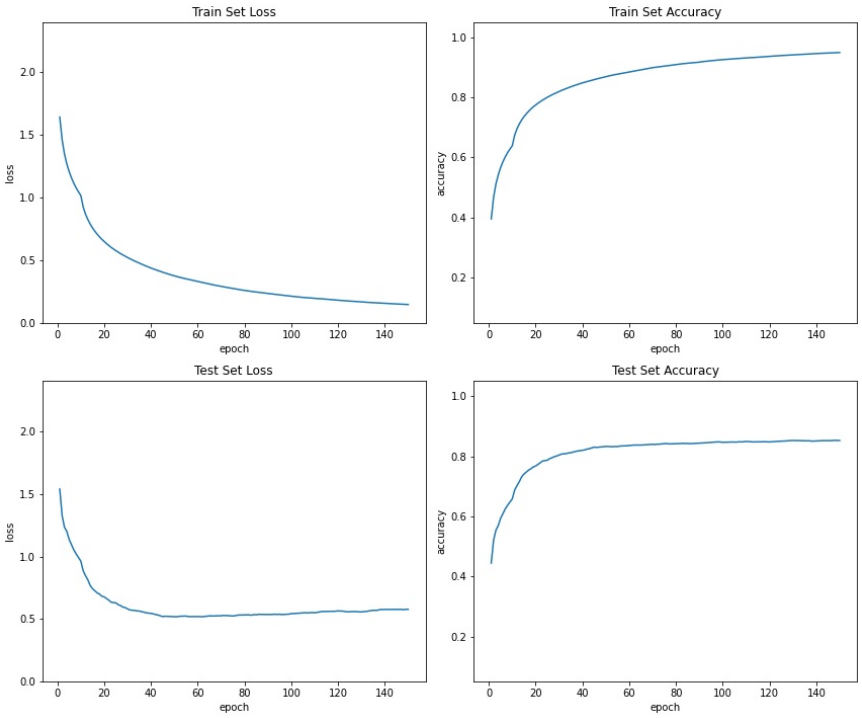
```
def convolution_block_with_opt_gap(n_filters, dropout_rate, gap=False,
                                   activation='relu'):
    if gap:
        pool_or_gap_layer = tf_layers.GlobalAveragePooling2D()
    else:
        pool_or_gap_layer = tf_layers.MaxPooling2D(pool_size=(2, 2))

    return [tf_layers.Conv2D(n_filters, (3, 3), padding='same',
                             activation=activation),
            tf_layers.BatchNormalization(),
```

```
                tf_layers.Conv2D(n_filters, (3, 3), padding='same',
                                 activation=activation),
                tf_layers.BatchNormalization(),
                pool_or_gap_layer,
                tf_layers.Dropout(dropout_rate)]
```

Z modelu usunąłem warstwę `Input`, dzięki czemu może on przyjmować obrazki o dowolnym rozmiarze. Ostatecznie model zaimplementowałem następująco:

```
model_6 = tf.keras.Sequential(layers=[
    tf_layers.Lambda(lambda x: x / 255),
    *convolution_block_with_opt_gap( 20, 0.1),
    *convolution_block_with_opt_gap( 40, 0.2),
    *convolution_block_with_opt_gap( 80, 0.3),
    *convolution_block_with_opt_gap(160, 0.4, gap=True),
    tf_layers.Flatten(),
    tf_layers.Dense(10, activation='softmax')
], name='model_6')
```

Sieć osiąga wyniki bardzo podobne do poprzedniej wersji (rys. 8).



Rys. 8: Ostateczne wyniki modelu po dodaniu warstwy *GAP*

Czas trenowania wyniósł **1261 s**, co daje ok. **8.40 s** na epokę.

Ostatecznie, udało się osiągnąć **85%** ***accuracy*** na zbiorze testowym. Dzięki dodaniu warstwy *GAP*, model obsługuje teraz także obrazki o innych rozmiarach. Jako przykład, przygo-

towałem dość prosty do klasyfikacji, lecz nietypowy obiekt klasy 0, tj. samolot Airbus Beluga widoczny na rys. 9.

Rys. 9: Airbus Beluga (Don-vip, CC BY-SA 3.0)

Nowy obrazek przeskalowałem do rozmiaru 43×32.

Sieć z niemal 100% prawdopodobieństwem wskazała na poprawną klasę 0. Mimo nietypowego wyglądu, na zdjęciu widać sporo cech samolotu jak silnik czy skrzydła, więc nie dziwi tak dobry wynik.

4 Wnioski

Ostatecznie udało się za pomocą kilkunastu linijek kodu zaimplementować sieć, która osiąga aż 85% *accuracy* na zbiorze Cifar10. Jest to imponujące!

Okazało się, że nie wystarczy po prostu dodać kilku warstw konwolucyjnych i sieci neuronowej na końcu, gdyż nawet z wykorzystaniem aktywacji ReLU wyniki nie były dobre. Pomogło zastosowanie warstw *batch normalization* oraz *dropout*, dzięki którym sieć została „ustabilizowana" i zmniejszyła się znacząco jej skłonność do przetrenowania. Trzeba jednak pamiętać, że zbiór Cifar10 jest łatwy, tzn. zawiera tylko obrazki 32×32 z wycentrowanymi obiektami i stworzona w ramach tego zadania sieć na pewno nie sprawdziła by się tak dobrze w przypadku bardziej „życiowych" danych.