Metody głębokiego uczenia
Konwolucyjne sieci neuronowe – raport

Jadwiga Słowik

17 kwietnia 2019

1 Cel zadania

Celem zadania jest implementacja konwolucyjnej sieci neuronowej (*CNN*) klasyfikującej obrazki ze zbioru *CIFAR-10* z jak największą wartością *accuracy* na zbiorze testowym.

2 Opis zbioru danych

CIFAR-10 (*Canadian Institute For Advanced Research*) to zbiór kolorowych obrazków o wymiarach 32×32 pixele. Każdy obrazek ma przydzieloną dokładnie jedną z 10-ciu następujących klas: samolot, samochód (*ang. automobile*), ptak, kot, jeleń, pies, żaba, koń, statek, ciężarówka. Warto zaznaczyć, że podane klasy są wykluczające.

Zbiór *CIFAR-10* został podzielony na następujące podzbiory:

1. Zbiór treningowy – 50 000 obrazków
   Składa się z pięciu równolicznych podzbiorów (*batchy*) obrazków ułożonych w losowej kolejności. To znaczy, że w obrębie danego *batcha* liczności obrazków zgrupowanych po klasie nie muszą być takie same. Aczkolwiek, sumarycznie, zbióry dla każdej z klas są równoliczne, tzn. jest tyle samo obrazków kotów, samochodów, itd.

2. Zbiór testowy – 10 000 obrazków
   Zbiór testowy jest zbalansowany, tzn. jest równomierny rozkład klas – po 1 000 wystąpień każdej klasy.

3 Opis konwolucyjnej sieci neuronowej

Proces uczenia konwolucyjnej sieci neuronowej (*CNN*) można podzielić na dwie główne części:

1. Wykrywanie charakterystycznych cech obrazka

2. Klasyfikacja

Wykrywanie charakterystycznych cech obrazka można podzielić na etap „konwolucji" (*ang. Convolution*), wprowadzenie nieliniowości ($ReLU$) oraz warstwę *Pooling*.

Klasyfikacja odbywa się na podstawie wykrytych (przez poprzedni etap) cech, które są analizowane przy pomocy sieci *MLP*.

Poniżej znajduje się szczegółowy opis każdego z wyżej wymienionych etapów:

1. *Convolution*
   Celem operacji *Convolution* jest wykrycie pewnych cech z danego obiektu/obrazka. Owa procedura polega na „przykładaniu" konkretnego filtra (macierzy, *ang. filter, kernel, feature detector*) do macierzy z pikselami (obrazka) i wykonywaniu operacji mnożenia element

po elemencie (*ang. element-wise*). Po wykonaniu owej operacji mnożenia, wynikowe elementy są sumowane i wynik (jedna liczba) jest zapisywany w komórce nowej macierzy. W trakcie operacji *Convolution* przesuwamy ów filtr o wybraną liczbę elementów (parametr *stride*).

Możemy mieć wiele filtrów (określa to parametr dt. głębokości (*ang. depth*)). Korzyścią, jaką osiągamy dzięki zastosowaniu większej liczby filtrów, jest możliwość „równoległego" wykrywania różnych cech obrazka np. krawędzie poziome, pionowe, itp.

Ostatecznie otrzymujemy macierz wynikową (bądź kilka macierzy), którą przekazujemy do następnego etapu.

2. Wprowadzenie nieliniowości
   Na elementach wynikowej macierzy wykonujemy wybraną funkcję nieliniową. Zwykle w tym przypadku stosuje się funkcję *ReLU*. Uważa się, że działa ona znacznie lepiej niż inne funkcje takie jak na przykład *sigmoid* lub *tanh*.

3. *Pooling* (*subsampling*, *downsampling*)
   Macierze poddaje się operacji *Pooling*, której celem jest redukcja wymiarowości macierzy otrzymanych w poprzednim etapie przy jednoczesnym zachowaniu istotnych cech potrzebnych w procesie klasyfikacji.

   Owa operacja polega na wykonywaniu wybranej funkcji (takiej jak np. maksimum, średnia, suma) na rozłącznych grupach sąsiadujących pikseli. W rezultacie otrzymujemy macierz, której elementami są wyniki zastosowania wspomnianej funkcji dla kolejnych grup sąsiadujących elementów.

   Uważa się, że dzięki operacji *Pooling* zmniejszamy prawdopodobieństwo przeuczenia i sprawiamy, że sieć jest mniej wrażliwa na małe transformacje obrazka.

4. Przetwarzanie w sieci gęstej (*ang. fully connected*)
   Ta warstwa jest zwykłą siecią neuronową *MLP*. Celem tego etapu jest klasyfikacja wynikowego (z poprzedniego etapu) zbioru cech.

Zbudowana przez nas sieć *CNN* może się składać z wielu warstw konwolucji, nieliniowości oraz *Poolingu*, które są ze sobą przeplatane. Dzięki posiadaniu wielu warstw, sieć neuronowa może lepiej uczyć się cech, należących do różnych klas abstrakcji. Mianowicie, pierwsze warstwy konwolucji mogą wykrywać jedynie konkretne krawędzie, podczas gdy kolejne mogą wyspecjalizować się w znajdowaniu bardziej złożonych obiektów takich jak np.uszy.

Proces uczenia polega na znalezieniu odpowiednich współczynników filtrów i wag sieci. Dlatego też, na początku inicjalizujemy wagi sieci *MLP* oraz wagi filtrów losowymi wartościami. Następnie, bierzemy obrazek (bądź zwykle cały *batch*) ze zbioru uczącego jako *input* i wykonujemy *feed forward* przechodząc przez wyżej wymienione etapy. Ostatecznie, obliczamy wartość funkcji kosztu i wykonujemy propagację wsteczną.

4 Opis rozwiązania

Konwolucyjne sieci neuronowe rozwiązujące problem z różną skutecznością zostały zaimplementowane przy użyciu biblioteki *Keras* w języku *Python*. Kod zamieszczony w *jupyter-notebooku* został wykonany na platformie *Google Colab*.

Dane dotyczące najlepszych rezultatów (tzn. modele osiągające najwyższe *accuracy*) zostały zapisane w plikach *JSON* wraz z informacją o procesie uczenia i predykcji oraz kilka z nich została również poddana serializacji do plików binarnych (przy pomocy operacji *pickle* pochodzącej z języka *Python*). Zapisane modele (i wyniki) znajdują się w paczce `models_data.zip`. Natomiast, wykorzystane kody w niniejszej pracy są w *jupyter-notebookach* o nazwach:

- `report_first_models.ipynb` – początkowe próby konstrukcji modelu sieci *CNN*
- `report_second.ipynb` – zaimplementowany generator testów i wywołania bardziej zaawansowanych modeli, których wyniki zostały zapisane w plikach znajdujących się w paczce z modelami
- `report_start_and_results_analysis.ipynb` – początkowe próby pracy z danymi, a później – analiza wyników zapisanych w paczce `models_data.zip`

Niestety, nie wszystkie wyniki wykonanych testów udało mi się zamieścić w notatnikach. Spowodowane to zostało przez następujące okoliczności:

1. część wykonywanych testów została niespodziewanie przerwana przez *Google Colab*, pomimo tego, że czas „bezczynności notatnika" był znacznie mniejszy niż dwie godziny
2. niektóre testy po dostatecznie dużej liczbie epok sama przerywałam i uruchamiałam ze zmienionymi parametrami, gdy zauważyłam, że sieć dalej nie uczyła się dostatecznie dobrze

Jednakże, wnioski pochodzące z wykonania „niepełnych" procesów uczenia również zostały zawarte w niniejszym raporcie.

4.1 Zastosowane podejście

Z danego zbioru uczącego został klasycznie wydzielony zbiór walidacyjny, który stanowił pierwsze 10% pierwotnego zbioru.

Poniżej zostanie opisana moja droga do ostatecznego rozwiązania: *accuracy* równe $81,91\%$ (`model7.json`).

4.1.1 Wybór optymalizatora

Na początku została wykonana seria eksperymentów mająca na celu wybranie najlepszego hiperparametru dotyczącego optymalizatora. Testowane były następujące optymalizatory:

- *SGD*
- *RMSprop*
- *Adagrad*
- *Adam*

Zdecydowanie najlepsze wyniki osiągał optymalizator *SGD*: dla danej sieci z kilkoma warstwami konwolucji i 20 epokami osiągał *accuracy* 67%, podczas gdy pozostałe optymalizatory dawały bardzo mierne wyniki: *accuracy* na poziomie 10-20%.

4.1.2 Wybór liczby warstw w *MLP*

Po wykonaniu serii testów (dla około 10 epok) doszłam do wniosku, że zastosowanie więcej niż jednej/dwóch warstw ukrytych pogarsza wyniki klasyfikacji. Wobec tego, ten hiperparametr ustawiłam na wartość równą 2. W każdej warstwie ukrytej znajdowało się 128 neuronów z funkcją aktywacji *ReLU*.

4.1.3 Wybór konfiguracji warstwy *poolingowo-konwolucyjnej*

Napisałam pomocnicze funkcje do generowania testów (kod jest zawarty w *jupyter-notebooku* `report_second`). Przy ich pomocy mogłam w łatwy sposób wywoływać testy z różnymi konfiguracjami (np. określona liczba warstw konwolucyjnych itp.).

Wyniki tej serii testów przedstawiają się następująco (parametr *learning rate* został ustawiony na wartość $0,01$):

id	conv_num	pooling_iter	epochs	accuracy
1	$1+3$	1	5	63%
2	$1+10$	1	5	61%
3	$1+30$	1	5	47%
4	$1+3$	4	5	62%
5	$1+8$	4	5	17%
6	$1+3$	4	40	76%

Tabela 1: Wyniki testów dla różnych konfiguracji warstwy *poolingowo-konwolucyjnej*

W kolumnie *conv_num* liczba warstw konwolucji została przedstawiona przy pomocy sumy, gdyż pierwsza warstwa miała filtry o wymiarach 3×3, natomiast następne warstwy miały wymiary 2×2. Głębokość (liczba *filtrów*) każdej warstwy konwolucji wynosiła 128.

Jako że sieci były uruchamiane na platformie *Google Colab*, gdzie wystąpują ograniczenia czasowe związane z wykonywanymi obliczeniami, musiałam znacząco okroić liczbę konfiguracji sieci, które miałam zamiar testować, gdyż niemożliwe było zostawienie dłuższych obliczeń bez nadzoru.

Intuicyjnie, sieć o numerze porządkowym równym 4 zachowywała się bardzo obiecująco przy małej liczbie epok, dlatego też zdecydowałam się na sprawdzenie jej skuteczności dla większej liczby. Ostatecznie, postanowiłam optymalizować inne hiperparametry tej właśnie sieci.

4.1.4 Zastosowanie metod regularyzacji

Zastosowanie metod regularyzacji ma na celu wyeliminowanie zjawiska przeuczenia się sieci (*overfittingu*), czyli zbyt dużego dopasowania się do danych treningowych bez zdolności generalizacji na inne przypadki.

Podjęłam się zastosowania następujących metod:

- *Kernel regularizer* dla wartstw *fully connected* i warstw konwolucyjnych
- *Dropout* (prawdopodobieństwo wygaszenia połączenia równe $0,2$)
  Ideą metody *dropout* jest wygaszanie losowych połączeń w sieci neuronowej. Zastosowałam *dropout* dla każdej warstwy sieci.
- *Batch Normalization*
  Owa metoda poddaje normalizacji wynik (*aktywację*) poprzedniej warstwy.

Wykorzystanie pierwsze metody spowodowało znaczący spadek skuteczności klasyfikacji do rzędu 20%, natomiast że zastosowanie dwóch ostatnich metod poprawiło wynik o kilka punktów procentowych (*accuracy* rzędu *79%*).

4.1.5 Pierwsza próba zastosowania *data augmentation*

Podjęłam pierwszą próbę zastosowania *data augmentation* (przy pomocy obiektu *ImageDataGenerator* z biblioteki *Keras*) z następującymi operacjami na obrazkach:

- `width_shift_range` $= 0,1$

- `height_shift_range` $= 0,1$
- `shear_range` $= 0,2$
- `zoom_range` $= 0,3$
- `horizontal_flip` oraz `vertical_flip`

Zauważyłam, że od pewnego momentu wartości *accuracy* dla zbioru walidacyjnego i treningowego prawie nie ulegają zmianie – utrzymują się w okolicach 78%. Po wykonaniu 150 epok, skuteczność sieci dla danych testowych była podobna.

Podejrzewam, że zatrzymanie skuteczności uczenia mogło być spowodowane przez dwa następujące czynniki:

- Ustawiłam zbyt dużo różnych przekształceń obrazka, w wyniku czego generowane obrazki nie były dostatecznie użyteczne dla procesu uczenia sieci i również, z powodu dużej liczby możliwości – niektóre (być może bardziej przydatne transformacje) miały mniejsze prawdopodobieństwo bycia wygenerowanym
- Została zastosowana jedna wartość *learning rate*
  Przypuszczam, że w tym przypadku zastosowanie mniejszego *learning rate* dla większych numerów epok mogłoby przynieść poprawę skuteczności klasyfikacji

4.1.6 Wybór *stałej uczenia*

Na początku testowałam zachowanie sieci w przypadku, gdy nie zmieniałam *explicite* wartości *stałej uczenia*. Ustawiałam ją na następujące wartości: $0,01,\ 0,001,\ 0,0005$. Zauważyłam jednak, że przy ustawianiu mniejszej wartości sieć wolniej się uczy i wymaga większej liczby epok, aby osiągnąć dobrą skuteczność.

Wobec tego, zdecydowałam się na zastosowanie zmiennej wartości stałej uczenia w zależności od numeru epoki, która jest aktualnie wykonywana. Mianowicie, dla numeru epoki:

- w przedziale $[1, 30)$: *learning rate* równy $0,01$
- w przedziale $[30, 60)$: *learning rate* równy $0,005$
- w przedziale $[60, \infty)$: *learning rate* równy $0,003$

Niestety, zastosowany zabieg nie poprawił jakości klasyfikacji. Testowany był dla 80 epok (`model8.json`). Być może, zwiększenie liczby epok poprawiłoby jakość *accuracy*. Niestety, fizycznie nie byłabym w stanie wykonać obliczeń trwających ponad parenaście godzin pracując na platformie *Google Colab*.

4.1.7 Druga próba zastosowania *data augmentation*

Wyciągnowszy wnioski z powyżej opisanego podejścia, postanowiłam wykonać drugą próbę. Postanowiłam *douczyć* wcześniej wytrenowany model (`model7.json` o skuteczności ok. 82%) dodając *data augmentation*.

Zastosowałam zmienny (w zależności od numeru epoki) *learning rate* (liczony według algorytmu opisanego we wcześniejszym punkcie) oraz ograniczyłam możliwości transformacji obrazka do następujących parametrów:

- `rotation_range` $= 30$
- `width_shift_range` i `width_shift_range` równe $0,1$
- `horizontal_flip` i `vertical_flip`

Proces uczenia został ustawiony na 100 epok. Niestety, *Google Colab* niespodziewanie przerwał moje obliczenia w 75-tej epoce pomimo faktu, że około 10 minut wcześniej dokonywałam modyfikacji w *jupyter-notebooku*. Jednakże, wyniki uzystane do 75-tej epoki nie wyglądają optymistycznie: od pewnego momentu *accuracy* oscyluje w granicach 80% i sieć ma spore trudności, aby tę wartość przekroczyć.

4.2 Opis ostatecznego rozwiązania

W niniejszym rozdziale zostanie opisane rozwiązanie dające najwyższą skuteczność.

4.2.1 Architektura

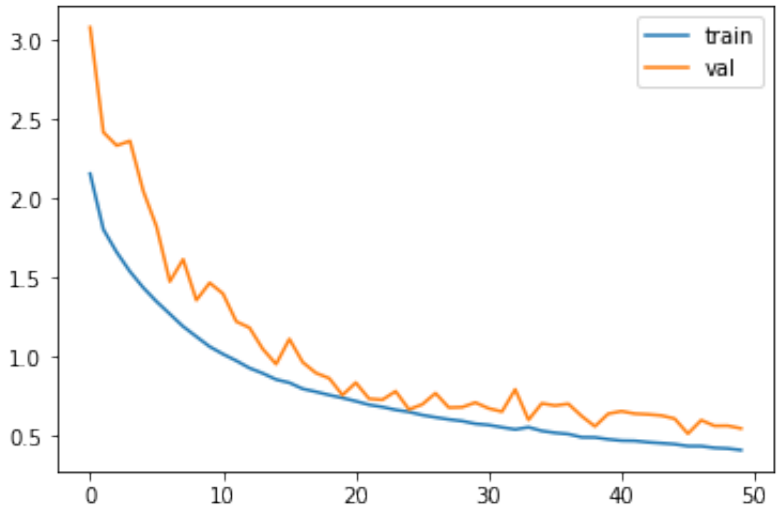
Sieć uzyskująca najwyższą skuteczność na zbiorze testowym $(81,91\%)$ ma następującą architekturę:

1. Jedna warstwa konwolucji o głębokości równej 128 i wymiarze filtra 3×3 i przesunięciu 1
2. Nieliniowość *ReLU*
3. 4 warstwy *MaxPooling* o wymiarze 2×2 i przesunięciu równym 2 (operacja *pooling* była wykonywana po każdej warstwie konwolucyjnej opisanej poniżej). W obrębie każdej warstwy:
   (a) Warstwa konwolucji: głębokość równa 128, wymiar filtra 2×2 oraz wartość przesunięcia (*ang. stride*) równa 1
   (b) Nieliniowość *ReLU*
   (c) *Dropout* z prawdopodobieństwem $0,2$
   (d) *BatchNormalization*
   (e) Warstwa konwolucji: głębokość równa 128, wymiar filtra 2×2 oraz wartość przesunięcia (*ang. stride*) równa 1
   (f) Nieliniowość *ReLU*
   (g) *Dropout* z prawdopodobieństwem $0,2$
   (h) *BatchNormalization*
   (i) Warstwa konwolucji: głębokość równa 128, wymiar filtra 2×2 oraz wartość przesunięcia (*ang. stride*) równa 1
   (j) Nieliniowość *ReLU*
   (k) *Dropout* z prawdopodobieństwem $0,2$
   (l) *BatchNormalization*
4. Warstwa *Fully-connected* z liczbą neuronów równą 128 i funkcją aktywacji *ReLU*
5. *Dropout* równy $0,2$
6. *BatchNormalization*
7. Warstwa *Fully-connected* z liczbą neuronów równą 128 i funkcją aktywacji *ReLU*
8. *Dropout* równy $0,2$
9. *BatchNormalization*
10. Wartstwa wyjściowa z liczbą neuronów równą 10 (liczba klas) i funkcją aktywacji *softmax*

Proces uczenia powyższej sieci trwał 50 epok i wartość *learning rate* była stała (o wartości $0,01$).

4.2.2 Funkcja kosztu

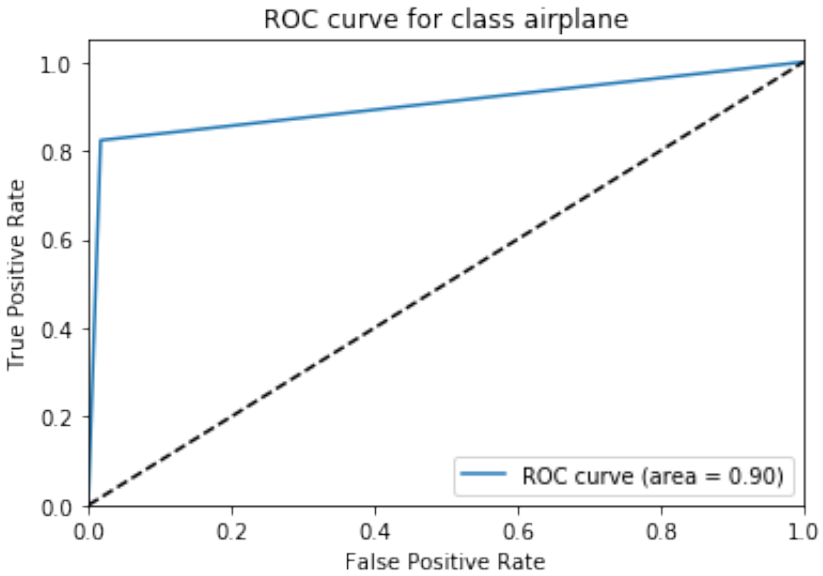
Wykres funkcji kosztu obliczanej w trakcie trenowania sieci dla zbioru treningowego i walidacyjnego przedstawia się następująco:



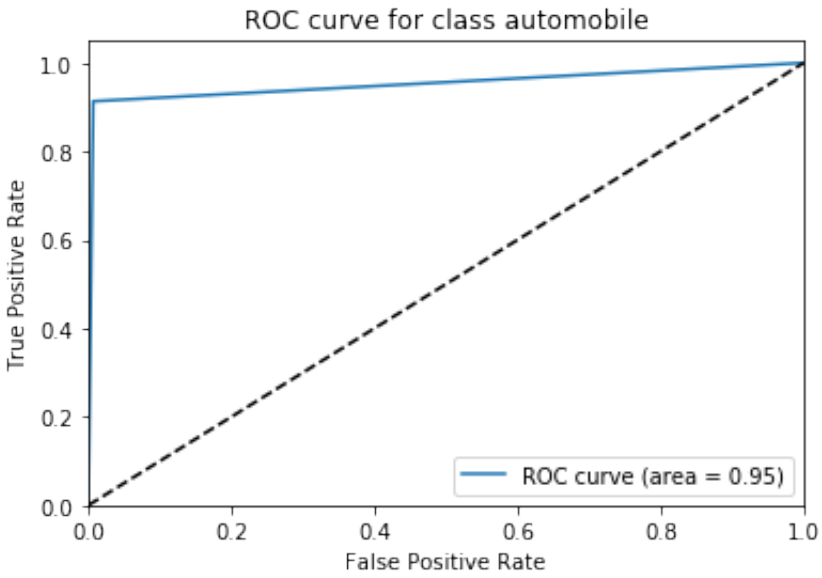
Rysunek 1: Wykres funkcji kosztu

4.2.3 Wykresy *ROC*

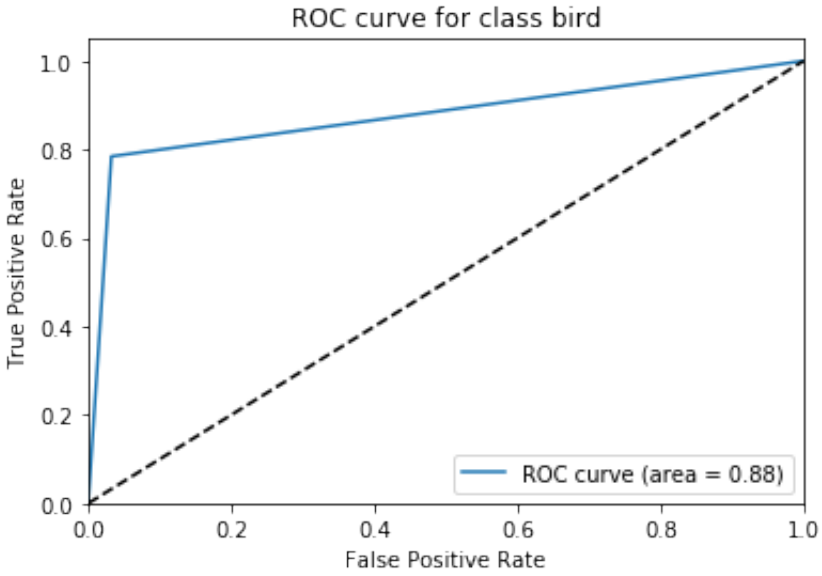
Poniżej znajdują się wykresy *ROC* dla każdej klasy:



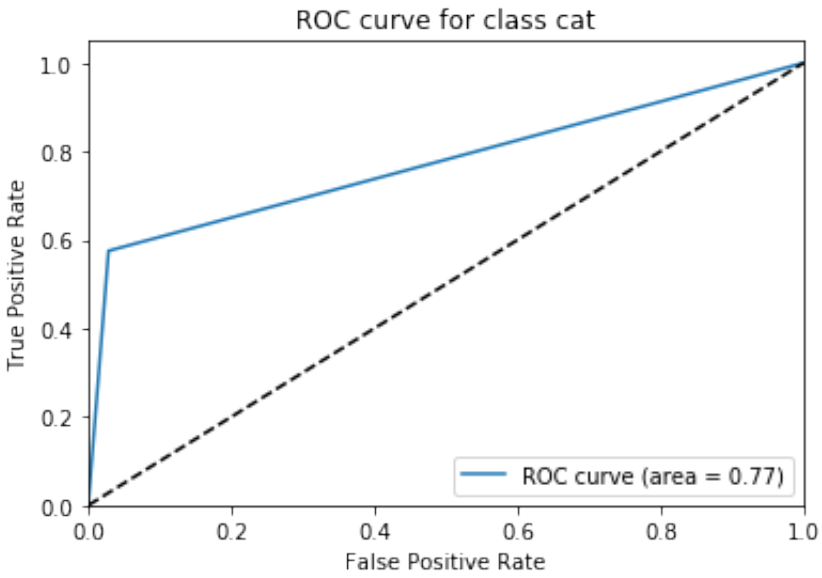
Rysunek 2: Wykres *ROC* z wartościami *AUC* dla klasy *airplane*



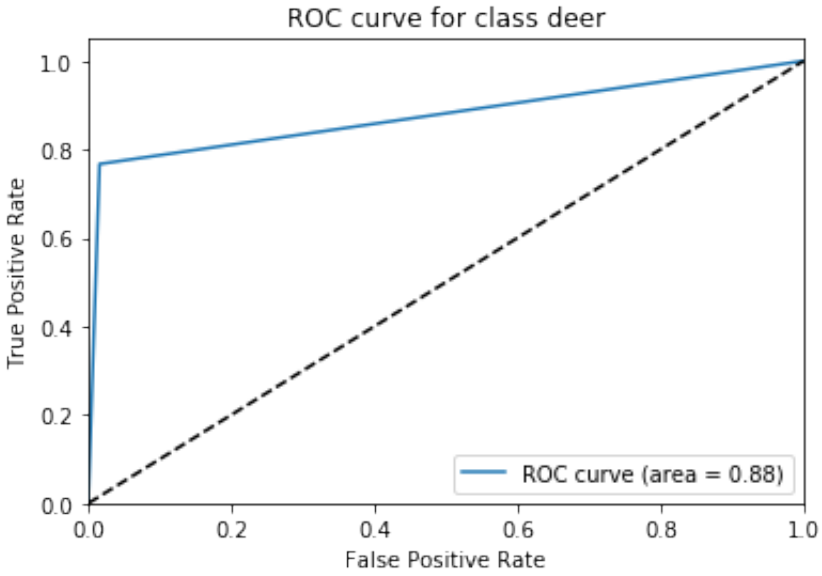
Rysunek 3: Wykres *ROC* z wartościami *AUC* dla klasy *automobile*



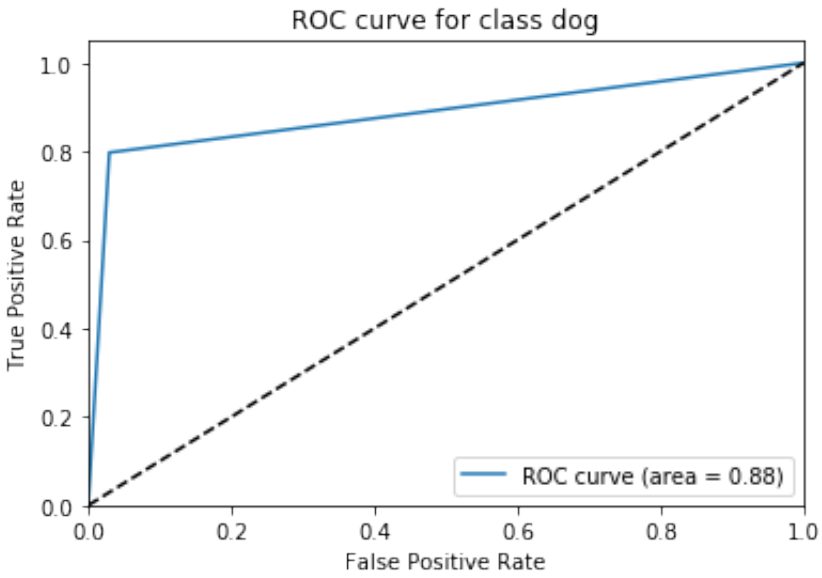
Rysunek 4: Wykres *ROC* z wartościami *AUC* dla klasy *bird*



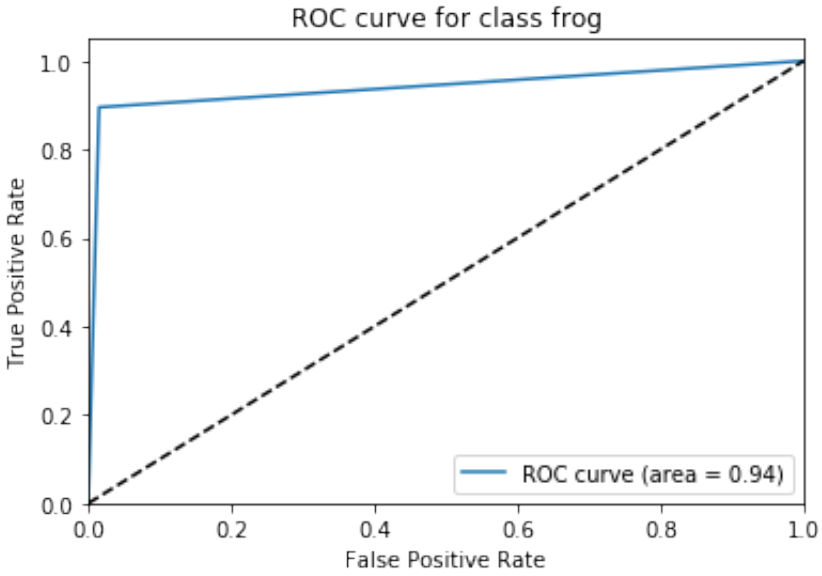
Rysunek 5: Wykres *ROC* z wartościami *AUC* dla klasy *cat*



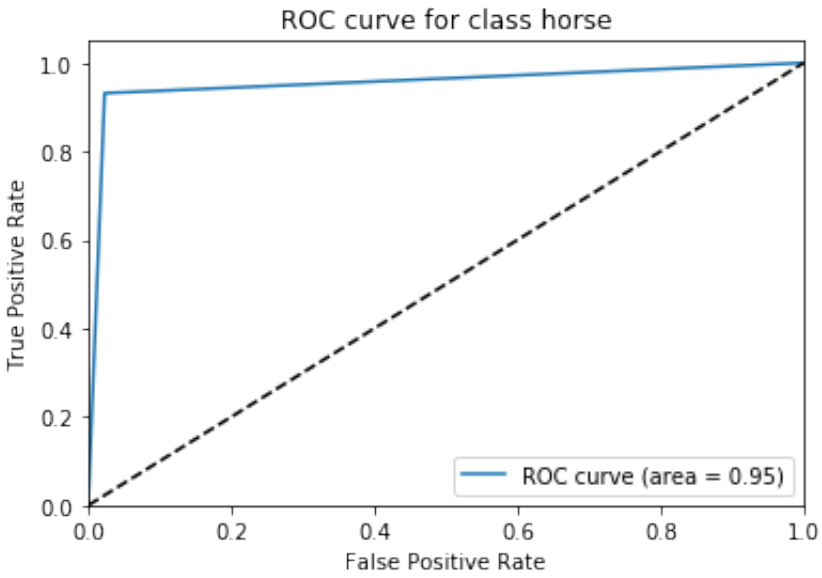
Rysunek 6: Wykres *ROC* z wartościami *AUC* dla klasy *deer*



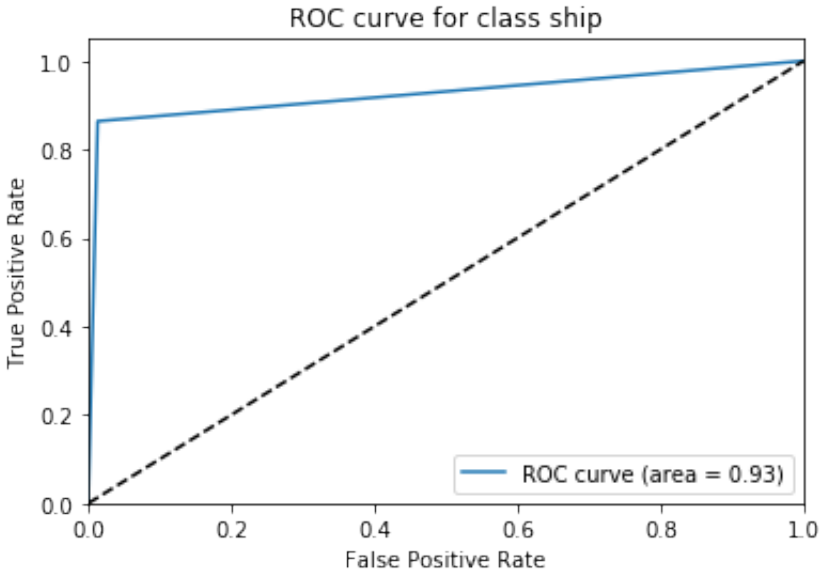
Rysunek 7: Wykres *ROC* z wartościami *AUC* dla klasy *dog*



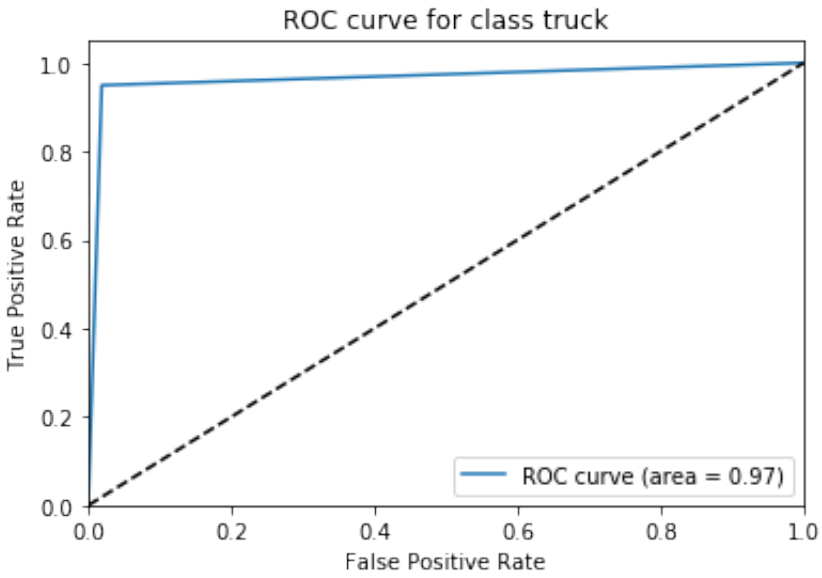
Rysunek 8: Wykres *ROC* z wartościami *AUC* dla klasy *frog*



Rysunek 9: Wykres *ROC* z wartościami *AUC* dla klasy *horse*



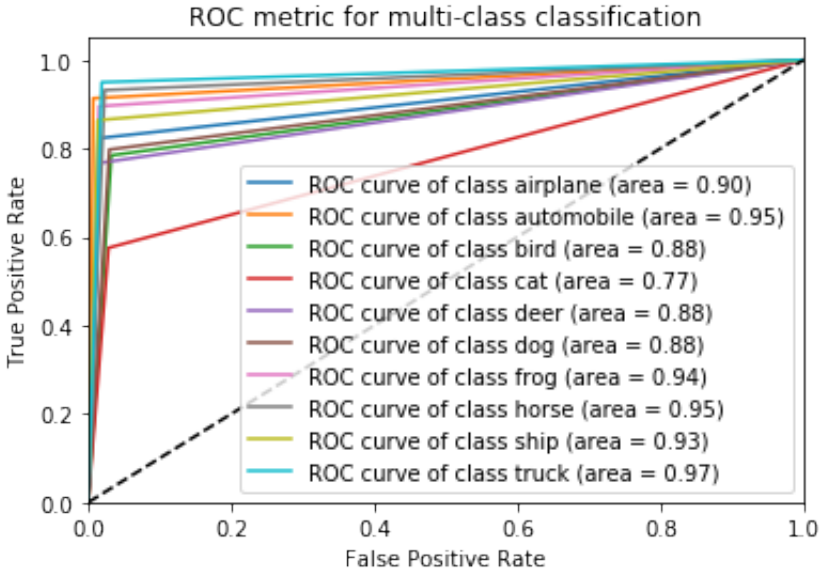
Rysunek 10: Wykres *ROC* z wartościami *AUC* dla klasy *ship*



Rysunek 11: Wykres *ROC* z wartościami *AUC* dla klasy *truck*

Na podstawie powyższych wykresów można stwierdzić, że zdecydowanie najgorszy wynik osiąga klasyfikacja kotów. Lista wszystkich źle sklasyfikowanych obrazków znajduje się w pliku *jupyter-notebooka* `report_start_and_tests_analysis`.

Poniżej znajduje się zbiorczy wykres *ROC*.



Rysunek 12: Wykres *ROC* z wartościami *AUC* dla każdej klasy

4.2.4 Macierz pomyłek

Macierz pomyłek (*ang. confusion matrix*) przedstawia się następująco

	0	1	2	3	4	5	6	7	8	9
0	844	9	63	16	4	4	4	11	40	30
1	7	937	0	3	1	0	2	0	15	61
2	34	2	705	39	33	33	25	20	4	4
3	35	3	94	753	60	174	64	58	47	21
4	16	1	71	40	859	31	24	68	2	8
5	0	2	24	99	7	732	8	44	1	1
6	3	2	31	27	18	10	863	3	3	4
7	5	1	5	11	12	15	3	790	0	6
8	49	21	5	9	6	1	7	3	881	38
9	7	22	2	3	0	0	0	3	7	827

Rysunek 13: Macierz pomyłek

Wiersze macierzy oznaczają wartość oczekiwaną, natomiast kolumny – wartość przewidzianą przez model. Z powyższej macierzy można wywnioskować, że bardzo dużo pomyłek występuje przy próbie klasyfikacji obrazka z kotem.

4.3 Pomysły na ulepszenie rozwiązania

Gdybym miała więcej czasu i lepsze warunki obliczeniowe, to spróbowałabym dodać do powyższej architektury *data augmentation*, ze zmienną (malejącą wraz ze wzrostem numeru epoki) wartością *learning rate* oraz znacznie większą liczbą epok (ponad 200).

Również, ciekawym pomysłem byłoby wykorzystanie pretrenowanej sieci i zastosowanie tzw. *transfer learning*.

Ponadto, następnym razem, aby zabezpieczyć się przez niespodziewanym zamknięciem sesji przez *Google Colab*, mogłabym zapisywać model, co pewną, wybraną liczbę epok.