Analiza Obrazów

Dokumentacja projektu - Klasyfikator ras psów

Kamil Jagodziński, Mateusz Nizio, Jakub Kraśniak

1. O programie:

Program wykonywalny:
Po pobraniu i rozpakowaniu projektu na urządzeniu z systemem linux, należy przejść do folderu 'Klasyfikator'. Znajduje się w nim plik wykonywalny 'out'. Sam program włączyć możemy z poziomu konsoli (komenda ./out) lub poprzez dwa kliknięcia w ikonkę 'out'.

Skrypt w języku python:
Aby skrypt działał niezbędne jest pobranie szeregu dodatkowych bibliotek pythona (lista znajduje się pod koniec punktu 1). Jeśli pliki dokumentacja.pdf, image_recognition.py oraz learned_model_from_transfer.pt znajdują się w jednym folderze wszytko powinno działać poprawnie zarówno na systemie windows jak i linux. Program uruchamiamy komendą 'python3 out.py'.

Czego nie należy robić:
- W przypadku programu wykonywalnego próba klasyfikacji bez wczytanego zdjęcia powoduje zakończenie programu.

- Nie należy dodawać zdjęć na których nie ma psów. Program nie weryfikuje czy użytkownik podał zdjęcie, na którym znajduje się pies - jeśli będzie ono w poprawnym formacie to zdjęcie zostanie przyjęte ale wynikiem klasyfikacji i tak będzie grupa ras psów.

- Nie należy opuszczać eksploratora plików bez wybrania zdjęcia, ponieważ to też powoduje zakończenie działania programu.

Wymagane biblioteki:
- PyQt5,
- Pillow (PIL),
- numpy,
- torchvision,
- pytorch,
- torch,
- os.

2. Opis projektu:

Informacje ogólne:
Obiektem projektu jest klasyfikator ras psów. Algorytm na podstawie zdjęcia psa zamieszczonego przez użytkownika wskaże rasę, do której najprawdopodobniej należy dany pies, oraz dwie inne rasy, które również mogą być prawdopodobne pod tym względem.

Dane wejściowe:
Baza danych użyta do realizacji projektu i nauczenia algorytmu składała się ze 133 ras psów, gdzie każdą rasę reprezentowało około 80/90 zdjęć. Każda rasa została umieszczona w innym folderze, którego nazwa odpowiada nazwie rasy. Zdjęcia zostały także podzielone na trzy kategorie: do trenowania, do testowania i do walidacji.

Wykorzystane technologie:
W projekcie korzystaliśmy z języka Python w wersji 3.9.7. Początkowo głównym modułem wykorzystanym do nauczania algorytmu był Tensorflow (keras). Jednak z powodu niskiej dokładności i problemami z wersją biblioteki zdecydowaliśmy się na zastosowanie modułu PyTorch. Dodatkowo do stworzenia interfejsu użytkownika wykorzystaliśmy bibliotekę PyQt5 oraz udostępnionego wraz z nią kreatora gui. Do utworzenia programu wykonywalnego użyliśmy biblioteki pyinstaller, która umożliwia utworzenie pliku wykonywalnego na podstawie skryptu napisanego w języku python.

3. Interfejs użytkownika:

Podstawy interfejsu zostały wykonane z użyciem programu PyQt5 Designer. Zaprojektowaliśmy w nim ogólny layout GUI, a następnie do wygenerowanego kodu dopisaliśmy obsługę “drag and drop”, funkcje wywoływane wciśnięciem każdego z przycisków oraz komunikację z algorytmem klasyfikatora będącym osobnym plikiem.

Interfejs użytkownika umożliwia:
- Dodawanie zdjęć na dwa sposoby (eksplorator plików oraz drag&drop).
- Podgląd zdjęcia w okienku (oraz przeskalowanie go gdyby miało się nie zmieścić w okienku podglądu / było zbyt małe).
- Reset okna aplikacji do stanu początkowego.
- Dokonanie klasyfikacji aktualnie załadowanego zdjęcia i wyświetlanie wyników pracy algorytmu.
- Wyświetlanie dokumentacji.

4. Przygotowanie bazy danych

Dane wejściowe których zdecydowaliśmy się użyć w naszym zadaniu nie były od początku idealne. Problemem były „zepsute” zdjęcia, których rozmiar wynosił 0 kB. Zdjęcia miały także indeksy który nie były potrzebne w naszym programie,

więc oczyściliśmy bazę tak, aby pasowała do naszego projektu. Podczas projektu eksperymentowaliśmy także z różnymi bazami danych. W niektórych ras psów było znacznie więcej niż w innych, ale na zdjęciach występowały szumy (lub/i inne obiekty w tle), które zakłócały działanie algorytmu. Skrypt napisany w języku python pozwolił nam także na wyciągnięcie danych z pliku .csv(gdzie każdemu unikalnemu zdjęciu(id zdjęcia) odpowiadała jego rasa) i podzieleniu bazy na foldery gdzie nazwa folderu odpowiadała nazwie rasy. Ostatecznie podstawą naszych działań była baza: https://www.kaggle.com/c/dog-breed-identification/data, lecz została ona przez nas, metodą prób i błędów, urozmaicona (dodanie innych ras, usunięcie ras bardzo zbliżonych do siebie, zmniejsze/zwiekszenie ilości zdjęć i sprawdzenie jak wpłynie to na działanie algorytmu).

5. Konwolucyjna sieć neuronowa od podstaw

Architektura modelu

W naszym projekcie zdecydowaliśmy się na zbudowanie Konwolucyjnej Sieci Neuronowej(z ang. CNN) używającej 3 konwolucyjnych (splotowych) warstw. W końcowej wersji użyliśmy nawet pięciu warstw konwolucyjnych. Warstwy splotowe są podstawą w każdej sieci neuronowej. Zawierają one wyuczone filtry (kernele), które wyodrębniają cechy odróżniające od siebie różne obrazy. Filtr jest macierzą, dzięki której przeprowadzamy przekształcenia macierzowe (konwolucje). Przykład zastosowania filtra:

```
filtr = np.array([
    [1, 0, -1],
    [2, 0, -2],
    [1, 0, -1]])
apply_kernel_to_image(image, filtr, 'outline')
```

Rozmiar jądra w naszym algorytmie ustawiliśmy na 3x3, a wypełnienie na 1. Pierwszy z tych parametrów wpływa na dokładność wydobycia informacji z danych. Małe rozmiary są w stanie wydobyć ze zdjęć znacznie więcej informacji w lokalnych częściach zadania. Duże rozmiary pozwalają za to na lepsza generalizację problemu. Wypełnienie pozwala nam na obsługę obramowania danych. Osiągane jest to kosztem dodania na krawędziach sztucznych wag, najczęściej o wartości zero. Dzięki takiemu zabiegowi obraz wyjściowy będzie miał takie same rozmiary jak obraz wejściowy. Każda warstwa zawierała także funkcje aktywacji ReLU. Funkcja ta pozwala osiągnąć lepszą dokładność algorytmu dzięki wytworzeniu nieliniowych granic decyzyjnych- wynik nie może być zapisany jako liniowa kombinacja danych wejściowych. W projekcie używaliśmy także warstwy łączącej (pooling layer), która dzięki stopniowemu zmniejszaniu rozmiaru obrazu, zmniejsza także liczbę parametrów do wytrenowania (skraca czas działania). Działanie warstwy jest bardzo proste - z czterech sąsiadujących ze sobą pikseli wybieramy wartość największą, jednego z nich a następnie tworzymy nowy piksel o tej wartości. Warstwa porzucenia (dropout) pomogła uniknąć nadmiernego dopasowania danych wejściowych do danych treningowych (overfitting). Przy aktywacji, losowo wybrane neurony są ignorowane na określonym etapie co zapobiega budowaniu niewłaściwych połączeń. Używaliśmy także pojęcia kanału. Dla kolorowego zdjęcia, wyróżniamy trzy kanały barwne: czerwony, zielony niebieski. Dzięki temu możemy przedstawić zdjęcie jako macierz:

$$W \times H \times C$$

gdzie:
W - szerokość,
H - wysokość
C - liczba kanałów (u nas 3).

Sieć neuronowa przyjmuje plik o wymiarach $W \times H \times C$, jako parametr wywołania skryptu a następnie, generuje na wyjściu plik $W' \times H' \times C'$. Liczba kanałów na wejściu to C a na wyjściu to C'. Filtrem dla takiego przekształcenia jest tensor o wymiarach $F \times F \times C \times C'$, gdzie F to rozmiar filtra.

Wykorzystaliśmy także normalizację zdjęć (metoda BatchNorm2D), co pozwoliło nam na przyspieszenie działania algorytmu i zwiększenie jego dokładności (umożliwiamy CNN znormalizowanie danych wejściowych we wszystkich warstwach).

```
class ModelFromScratch(nn.Module):
    def __init__(self):
        super(ModelFromScratch, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.norm1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.norm2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.norm3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
        self.norm4 = nn.BatchNorm2d(128)
        self.conv5 = nn.Conv2d(128, 256, 3, padding=1)
        self.norm5 = nn.BatchNorm2d(256)

  self.dropout = nn.Dropout(0.25)
        self.fc2 = nn.Linear(1024, breeds_number)
```

Gdzie Conv2D (in_channels, out_channels, kernel_size, padding). Czyli np. dla Conv2D(3,16,3, padding=1) otrzymujemy 16 kerneli (filtrów) o wymiarach 3x3 każdy mający 3 kanały co łącznie daje 16(3*3*3+1) parametrów. Lista parametrów zawiera wagi jakie wygenerował tensor.

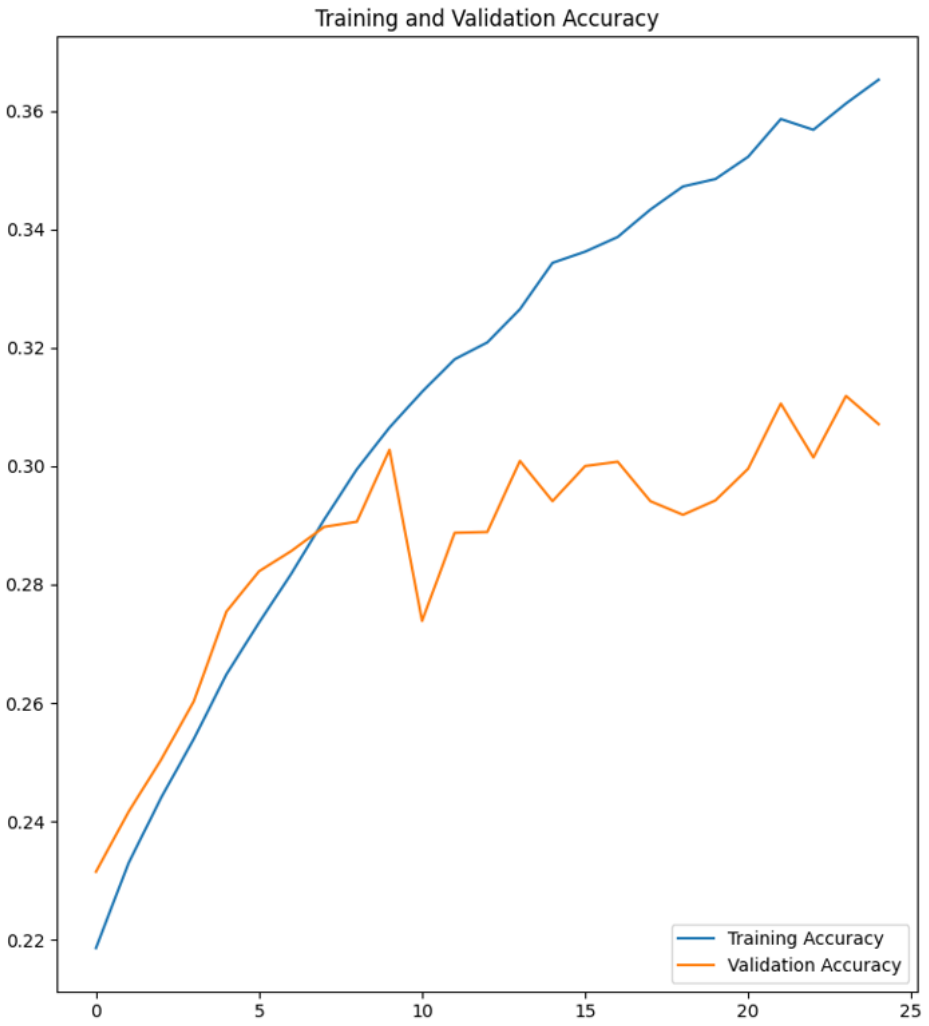
BatchNorm2D(num_features) jako argument przyjmuje liczbę kerneli warstwy.

Funkcje kosztu(loss function) i optymalizatory:
Jako optymalizatora użyliśmy funkcji Adagrad(), a naszą funkcją kosztu była CrossEntropyLoss. Optymalizator to algorytm, który poprzez zmianę wag i atrybutów sieci neuronowej minimalizuje straty podczas nauczania. Funkcja kosztu oblicza różnice między aktualnym wynikiem oraz spodziewanym wynikiem.

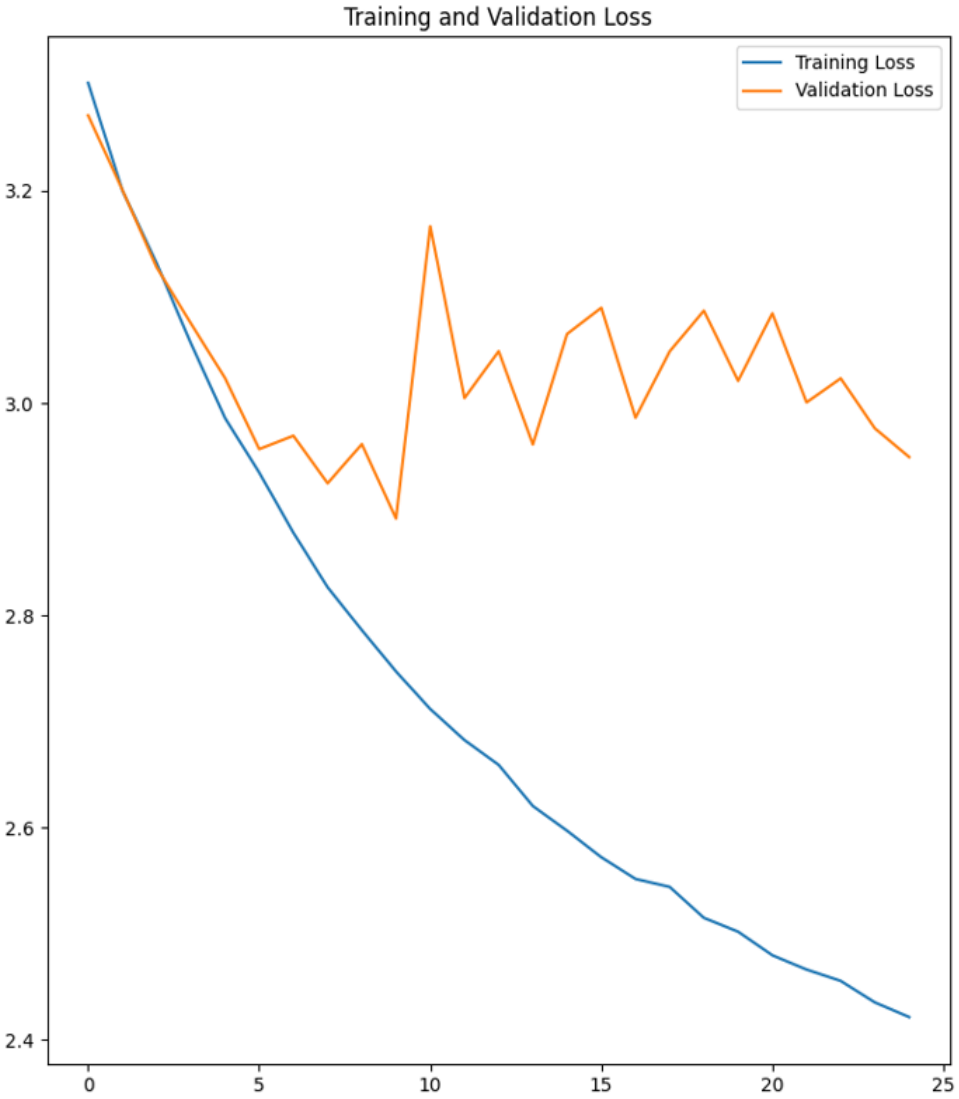
```
function_loss= nn.CrossEntropyLoss()

optimizer = optim.Adagrad(model.parameters(), lr=0.01)
```

Kompilacja I testowanie:

Po dodaniu jeszcze kilku niezbędnych rzeczy, skompilowaliśmy nasz algorytm. Wyniki nie były jednak bardzo zadowalające. Średnia dokładność algorytmu wyniosła 28% przy jednocześnie dużej utracie - wartości “loss” (po wykonaniu 15 epok).



Na osi X przedstawiono liczbę epok a na osi Y dokładność algorytmu w skali (0 - 1). Niebieska krzywa informuje o postępie podczas nauczania a żółta podczas walidacji. (Rys.1)



Na osi X przedstawiono liczbę epok a na osi Y wartości strat jakie program ponosił podczas nauczania (błąd między danymi wejściowymi z programu z oczekiwaną wartością). Im niższa wartość tym lepiej. (Rys.2)

Powiększenie bazy danych- data augmentation

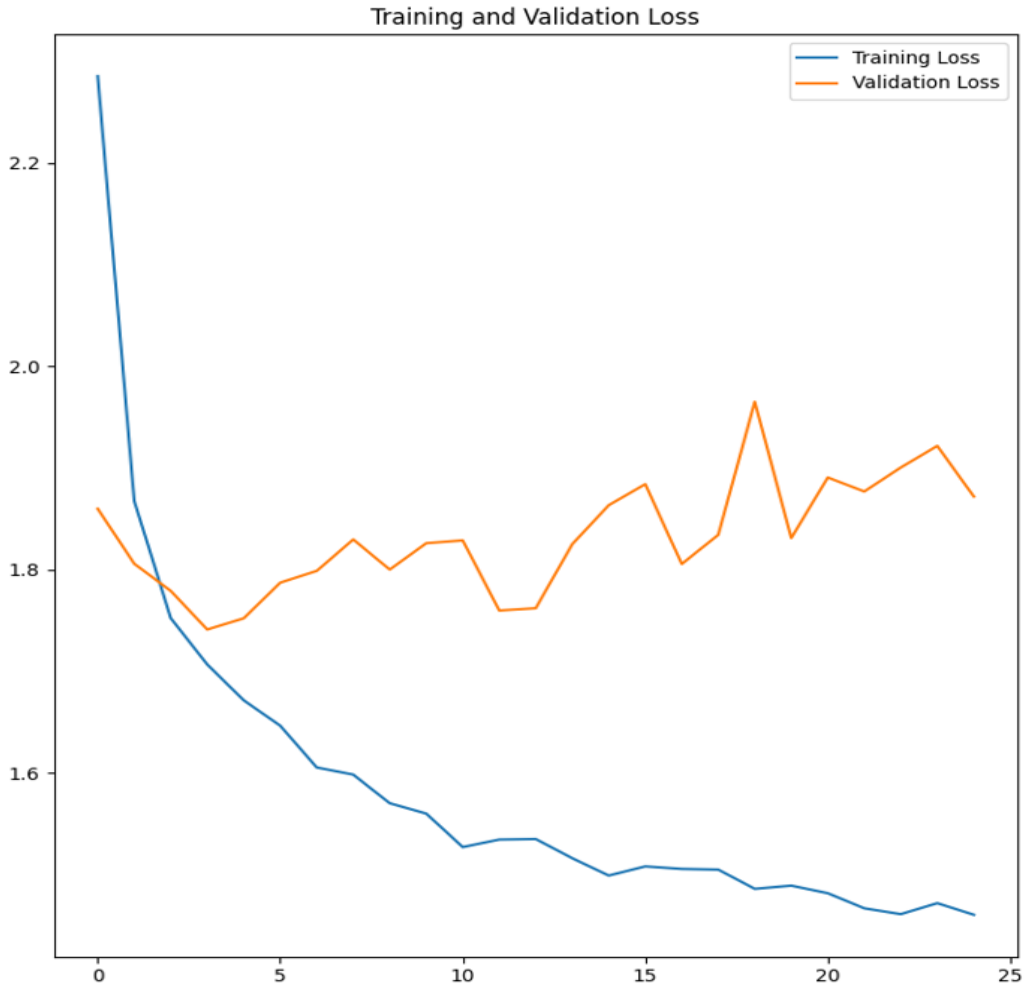
Aby poradzić sobie ze słabymi wynikami algorytmu postanowiliśmy zastosować metodę „data augmentation". Przed wysłaniem zdjęć do algorytmu poddawaliśmy je przekształceniom takim jak:

-losowanie obracanie
-przycinanie,
-przybliżanie.

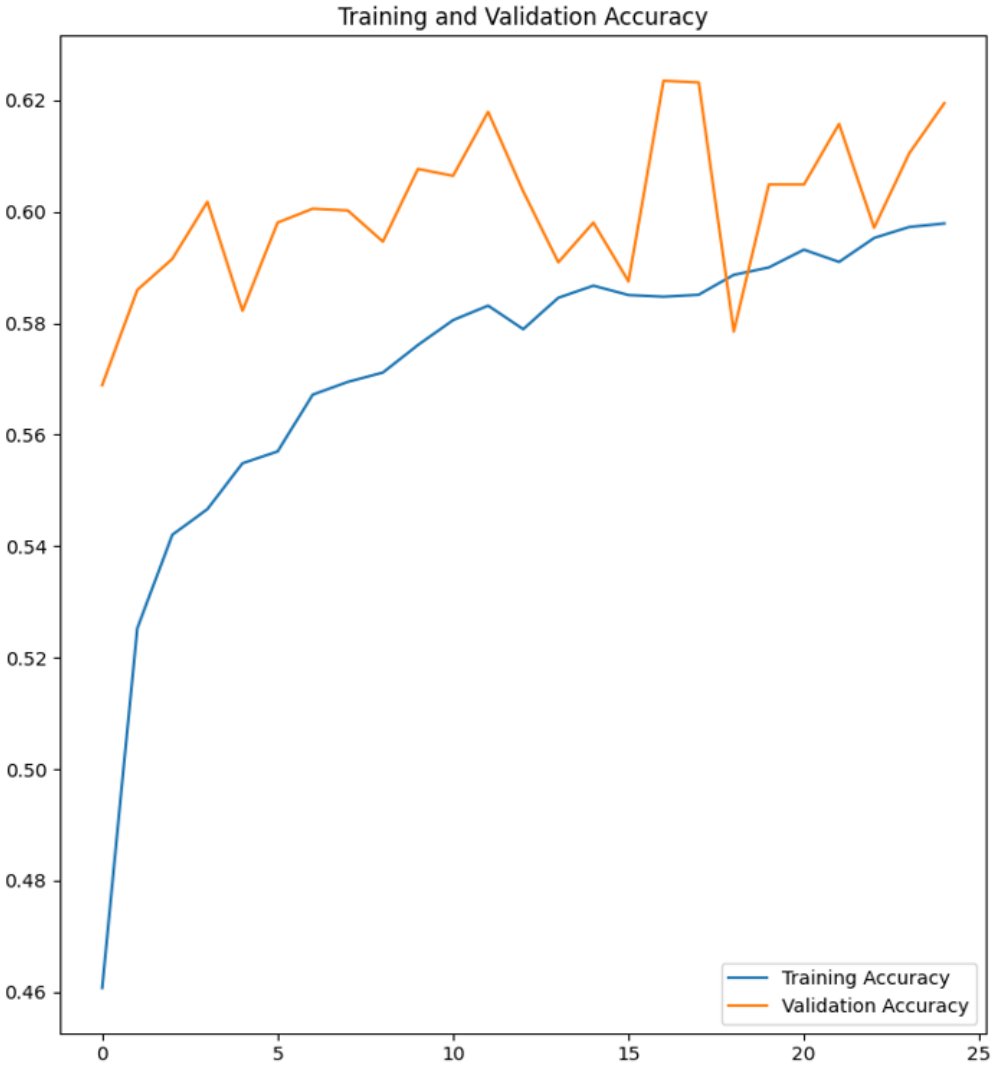
Rozmiar zdjęć został także ujednolicony do wymiaru 256x256, a następnie każde z nich zostało znormalizowane. Wszystkie te zabiegi zostały wykonane po to, aby poszerzyć naszą bazę danych i uporać się z problemem overfittingu.

```
transform = transforms.Compose([transforms.RandomRotation(10),
                            transforms.RandomResizedCrop(224),
                            transforms.RandomHorizontalFlip(),
                            transforms.ToTensor(),
                            transforms.Normalize(mean=[0.485,0.456])
```

Proces ten rzeczywiście wpłynął na poprawę działania programu, jednak wciąż nie był on idealny.



Na Osi X przedstawiono liczbę epok, na osi Y wartość strat jakie program ponosi podczas nauczania(oblicza błąd między danymi wyjściowymi z programu, a oczekiwaną testową wartością). Im wartość strat jest bliższa zeru, tym algorytm jest lepszy. (Rys.3)



Na osi X przedstawiono liczbę epok, na osi Y dokładność algorytmu (od 0 do 1). Niebieska krzywa - postęp nauczania, żółta - postęp walidacji. (Rys.4)

6. Transfer learning

Pracując nad algorytmem nie byliśmy już w stanie bardziej poprawić jego skuteczności. Ostatnim więc krokiem jaki postanowiliśmy zastosować było użycie techniki “transfer learning”. Pomysł który się za tym kryje to wykorzystanie sieci neuronowych które zostały wcześniej opracowane przez profesjonalistów. Zdecydowaliśmy się na użycie algorytmu DenseNet161, który został wytrenowany na bazie danych ImageNet. W naszym zadaniu zmieniamy model tak aby nie uczył się od nowa, a następnie modyfikujemy ostatnią warstwę tak żeby jej wynik końcowy był równy 133(ilość ras psów w naszej bazie). Dzięki transfer learningowi mogliśmy użyć początkowych warstw programu, a ostatnią warstwę przeszkolić za pomocą algorytmu DenseNet.

```
model_2_transfer_learning = models.densenet161(pretrained=True)
```

Zastosowanie takie rozwiązania okazało się bardzo owocne. Po wykonaniu 15 epok, algorytm osiągnął skuteczność około 80% co było zadowalającym nas wynikiem.

7. Co można było zmienić

Pracując nad algorytmem dowiedzieliśmy się kilku rzeczy o uczeniu maszynowym.

Po pierwsze, wybór odpowiedniej bazy danych jest bardzo ważny. Samo przygotowanie zdjęć do algorytmu wymaga zastanowienia się, jaki jest nasz główny cel i jak dokładne chcemy otrzymać wyniki. Zamiast ponad 100 ras psów, moglibyśmy zawęzić bazę do np. 20 ras, skracając czas uczenia algorytmu i ewentualnie zwiększając jego skuteczność (o ile nie mówimy o problemie overfittingu). Z drugiej strony, przy większej liczbie klas algorytm działa lepiej dla użytkownika, ponieważ powinien dostarczać dokładniejszych danych o rasie jego psa (możliwe do wyboru więcej ras).

Aby ulepszyć nasz algorytm, moglibyśmy spróbować użyć większej bazy danych (na przykład takiej, która zawiera 120 000 zdjęć), ale przy stosunkowo wolnych komputerach zajęłoby nam to bardzo dużo czasu. Eksperymentowaliśmy także z liczbą epok i rozmiarem obrazów. Okazało się, że zwiększenie rozmiaru obrazu wejściowego nie wydaje się poprawiać dokładności modelu, ale znacznie wydłuża czas potrzebny na nauczanie. Zwiększenie liczby epok również nie zawsze poprawia skuteczność algorytmu. W pewnym momencie dokładność nauczania już się nie zmienia (lub zmienia się o mniej niż 1% na epokę).

Kolejną rzeczą, która okazała się bardzo ważna, było zastosowanie techniki „data augmentation". Dzięki niej poprawiliśmy dokładność i zapobiegliśmy przeuczeniu (overfitting) algorytmu. Dzięki prostym operacjom, takim jak obracanie lub przycinanie, znacznie zwiększyliśmy różnorodność naszych danych. Możemy stwierdzić, że procedura danych jest bardzo ważna i powinna być wykonywana w każdym algorytmie ML.

Również w celu usprawnienia algorytmu, korzystając z augmentacji danych, zamiast losowo przycinać obraz, moglibyśmy spróbować precyzyjnie wyciąć fragment z psem (zdjęcia mają szumy tła, np. drzewa itp.).

Kolejną zmianą, którą moglibyśmy wprowadzić, jest próba użycia innego optymalizatora (Adam, Adamax, Momentum, SGD itp.). Każda taka zmiana wymagałaby jednak skompilowania programu od nowa, co przy naszej ograniczonej mocy obliczeniowej zajmowało około 10 godzin.

Na koniec podczas transfer learningu wybraliśmy algorytm Dense-Net 161 , ale moglibyśmy również spróbować skorzystać z innych modeli, takich jak: Resnet, Inception v3 lub MobileNet (MobileNet v2).

8. Źródła

Wskutek niedostatecznej wiedzy na temat CNN, podczas pisania projektu korzystaliśmy z wielu źródeł internetowych:

Machine Learning i informacje ogólne o budowaniu sieci neuronowej:

https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://www.tensorflow.org/tutorials/images/classification

https://www.tensorflow.org/tutorials/images/transfer_learning

https://www.youtube.com/watch?v=ofh27VwCoLE

https://web.stanford.edu/class/cs231a/prev_projects_2016/output%20(1).pdf

https://www.mi-research.net/en/article/doi/10.1007/s11633-020-1261-0

https://www.kaggle.com/reukki/pytorch-cnn-tutorial-with-cats-and-dogs

https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-to-classify-photos-of-dogs-and-cats/

https://towardsdatascience.com/build-your-first-deep-learning-classifier-using-tensorflow-dog-breed-example-964ed0689430

https://idiotdeveloper.com/dog-breed-classification-using-transfer-learning-in-tensorflow/

http://blizzard.cs.uwaterloo.ca/iss4e/wp-content/uploads/2017/10/yerbol_aussat_cs698_project_report.pdf

https://discuss.pytorch.org/t/how-the-pytorch-freeze-network-in-some-layers-only-the-rest-of-the-training/7088

https://medium.com/@ankitsingh540/dog-breed-classification-using-a-pre-trained-cnn-model-32d0d8b9cc26

Transformacje zdjęć:

https://www.programcreek.com/python/?code=jindongwang%2Ftransferlearning%2Ftransferlearning-master%2Fcode%2Fdeep%2Ffinetune_AlexNet_ResNet%2Fdata_loader.py

https://pytorch.org/vision/stable/transforms.html

Loss function:

https://neptune.ai/blog/pytorch-loss-functions

https://medium.datadriveninvestor.com/visualizing-training-and-validation-loss-in-real-time-using-pytorch-and-bokeh-5522401bc9dd

(Testowane) Bazy danych:

https://www.kaggle.com/c/dog-breed-identification/data

(główna baza wokół której powstała ostateczna wersja bazy wykorzystana w naszym programie)

https://www.kaggle.com/jessicali9530/stanford-dogs-dataset

https://www.kaggle.com/enashed/dog-breed-photos

https://www.kaggle.com/abhinavkrjha/dog-breed-classification

https://www.kaggle.com/gpiosenka/70-dog-breedsimage-data-set

https://vision.stanford.edu/aditya86/ImageNetDogs