**Opis Prac**

Prace poprzedziłem testami roboczymi gdzie sprawdzałem możliwości swojego komputera, testowałem rozmiar obrazków a także różne architektury modeli. Efekty które zamieściłem na publicznym repozytorium są prezentacją kroków które wykonałem oraz zawierają najlepszy model który udało mi się zbudować.

Prace podzieliłem na 3 etapy:

1. Własna nieskomplikowana architektura
2. Transfer learning z wykorzystaniem 2 faz uczenia
3. Niewielka aplikacja we Flask która pozwala na udostępnienie modelu

Wykorzystałem obrazki w rozmiarach 75x75, batch size ustawiony był na 32, a liczba obrazków przekazana w ramach każdej epoki stanowiła 40 krotności liczności obrazków którymi dysponowałem. Skorzystałem z data Data Generatora dostarczonego w ramach pakietu keras.

W trakcie uczenia zastosowałem „Early Stopping”, „ReduceLROnPlateau”, Regularyzację L2 oraz Dropout(0.3). Przeskalowałem wartości pixeli z 0-255 na 0-1.

**Własna nieskomplikowana architektura**

W pierwszym etapie zaproponowałem 10 Architektur które znajdują się w pliku „architectures.py”, zachęcam do zapoznania się z tym plikiem ponieważ lektura tamtych architektur może okazać się czytelniejsza. W każdej architekturze blok warstw konwolucyjnych miał ustawione parametry: kernel\_size=(3, 3), activation='relu', padding='same' i był zakończony MaxPoolingiem z pool size (2, 2). W warstwach ukrytych zastosowałem funkcje aktywacji ReLu, a w warstwie wynikowej funkcje softmax.

1. Conv8, Pool, Conv16, Pool, Conv32, Pool, Flatten, Dense128, Drop(0.3)
2. Conv32, Pool, Conv64, Pool, Conv128, Pool, Flatten, Dense128, Drop(0.3)
3. Conv32, Conv32, Pool, Conv64, Conv64, Pool, Conv128, Conv128, Pool, Flatten, Dense128, Drop(0.3)
4. Conv32, Conv32, Pool, Conv64, Conv64, Pool, Conv128, Conv128, Conv128, Pool, Conv128, Conv128, Conv128, Pool, Flatten, Dense128, Drop(0.3)
5. Conv32, Pool, Conv64, Pool, Conv128, Pool, Flatten, Dense512, Drop(0.3), Dense128, Drop(0.3)
6. Conv32, Conv32, Pool, Conv64, Conv64, Pool, Conv128, Conv128, Pool, Flatten, Dense512, Drop(0.3), Dense128, Drop(0.3)
7. Conv32, Conv32, Pool, Conv64, Conv64, Pool, Conv128, Conv128, Conv128, Pool, Conv128, Conv128, Conv128, Pool, Flatten, Dense512, Drop(0.3), Dense128, Drop(0.3)
8. Conv32, Pool, Conv64, Pool, Conv128, Pool, Flatten, Dense1024, Drop(0.3), Dense512, Drop(0.3), Dense128, Drop(0.3)
9. Conv32, Conv32, Pool, Conv64, Conv64, Pool, Conv128, Conv128, Pool, Flatten, Dense1024, Drop(0.3), Dense512, Drop(0.3), Dense128, Drop(0.3)
10. Conv32, Conv32, Pool, Conv64, Conv64, Pool, Conv128, Conv128, Conv128, Pool, Conv128, Conv128, Conv128, Pool, Flatten, Dense1024, Drop(0.3), Dense512, Drop(0.3), Dense128, Drop(0.3)

Wyniki:

|  |  |
| --- | --- |
| Nazwa architektury | Accuracy na zbiorze testowym |
| eval\_arch\_1 | 0.30 |
| eval\_arch\_2 | 0.39 |
| eval\_arch\_3 | 0.10 |
| eval\_arch\_4 | 0.10 |
| eval\_arch\_5 | 0.11 |
| eval\_arch\_6 | 0.08 |
| eval\_arch\_7 | 0.09 |
| eval\_arch\_8 | 0.09 |
| eval\_arch\_9 | 0.10 |
| val\_arch\_10 | 0.11 |

**Transfer learning z wykorzystaniem 2 faz uczenia**

Aby usystematyzować prace zdecydowałem na wybranie ujednoliconego podejścia do TL, postanowiłem że będę uczył modele w 2 fazach. W pierwszej zablokuje możliwość uczenia warstw konwolucyjnych i dołączę własną architekturę sieci głębokich (przygotowałem zunifikowane 3 architektury), a w drugiej fazie odblokuję ostatni blok warstw konwolucyjnych, tak by można było zmieniać w nim wagi, ustawię bardzo mały learning rate i uruchomię proces uczenia ponownie.

Przygotowałem w pliku „architectures.py” 4 funkcję które były pomocne w usystematyzowaniu prac. Trzy z nich zawierają zunifikowane architektury, a czwarta jest wykorzystywana w drugiej fazie uczenia.

W taki sposób przeuczyłem 3 modele VGG16, VGG19 oraz Xception. Każdy z modeli po 3 razy (dla każdej architektury). Z wyników niestety nie byłem zadowolony, więc zwiększyłem rozmiar obrazków do 224x224 i wytrenowałem VGG16 ponownie. Niestety mój komputer nie pozwala na uczenie bardziej złożonych modeli na tak dużych obrazkach, więc na wyniku 80% accuracy dla zbioru testowego poprzestałem.

Poniżej prezentuję accuracy dla wszystkich modeli które wytrenowałem, bardziej szczegółowe wyniki jak np. różnica między accuracy na zbiorze walidacyjnym i treningowym są dostępne w notebooku „Prepare\_model”. Architektury z sufiksem „\_2„ to wyniki drugiej fazy uczenia TL, a np. określenie arch2 sugeruje że była zastosowana 2 architektura. Arch1 – Dense(128), Arch2 – Dense(512), arch3 - Dense(512)Dense(128)

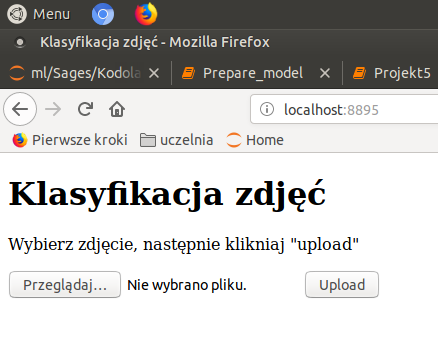
|  |  |
| --- | --- |
| Nazwa architektury | Nazwa architektury |
| eval\_VGG16\_arch1 | 0.41 |
| eval\_VGG16\_arch2 | 0.42 |
| eval\_Xception\_arch3 | 0.43 |
| eval\_VGG19\_arch1 | 0.43 |
| eval\_VGG19\_arch2 | 0.45 |
| eval\_VGG19\_arch3 | 0.46 |
| eval\_VGG19\_arch1\_2 | 0.46 |
| eval\_VGG19\_arch3\_2 | 0.47 |
| eval\_Xception\_arch1 | 0.47 |
| eval\_Xception\_all\_arch2 | 0.48 |
| eval\_VGG16\_arch2\_2 | 0.49 |
| eval\_VGG16\_arch3 | 0.50 |
| eval\_VGG16\_arch3\_2 | 0.50 |
| eval\_VGG19\_arch2\_2 | 0.50 |
| eval\_Xception\_arch2 | 0.52 |
| eval\_VGG16\_arch1\_2 | 0.52 |
| eval\_Xception\_arch3\_2 | 0.52 |
| eval\_Xception\_arch2\_2 | 0.54 |
| eval\_VGG16\_224\_arch3 | 0.68 |
| eval\_VGG16\_224\_arch3\_2 | 0.80 |

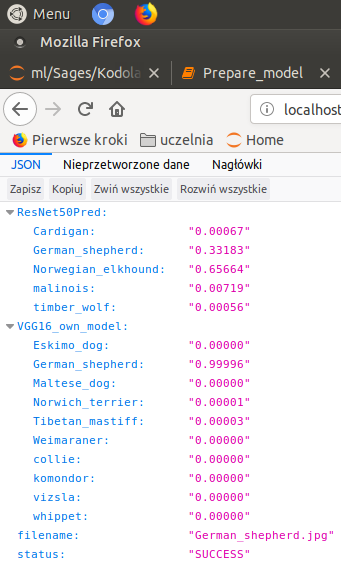
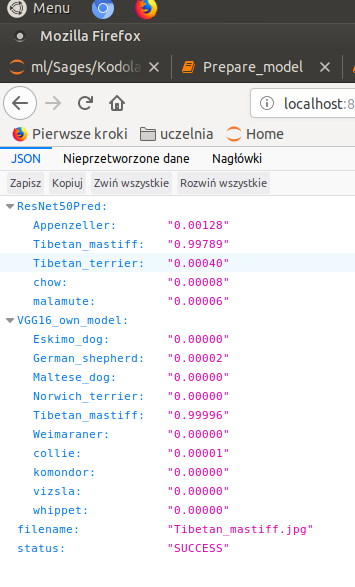
**Niewielka aplikacja we Flask która pozwala na udostępnienie modelu**

W pliku „Flask\_dog\_breed.ipynb” znajduje się niewielka aplikacja która pozwala na udostępnienie modelu i rozpoznanie razy psa na załączonym obrazku. Z racji tego że model który wytrenowałem po picklowaniu zajmuje 270 MB załączam jedynie screenshoty. Aplikacja zwraca predyjcję 2 modeli, mojego oraz ResNet50.

Do testów skorzystałem z 3 obrazków ras psów które wytrenowałem (obrazki pochodzą ze zbioru testowego):

Mamy tutaj 3 rasy: German\_shepherd, Maltese\_dog, Tibetan\_mastiff. A poniżej prezentuję działanie aplikacji :

Wita nas prosty ekran proszący a upload zdjęcia:

W odpowiedzi otrzymujemy dane w formacie JSON pochodzące z 2 modeli:

Dla Tiberian Mastiff oba modele są zgodne i poprwnie przewidują z prawdopodobieństwem przekraczającym 99 procent.

Dla German Shephard, nasz model przewidział poprawnie natomiast ResNet dopiero na 2 miejscu wskazał właściwą odpowiedź.

Dla Maltańczyka ResNet z prawdopodobieństwem 54% wskazał właściwą odpowiedź, a nasz model dał właściwej razie tylko 1% i wskazał na Mastifa.

Porównanie z ResNet oczywiście nie wskazuje na to że nasz model jest lepszy, mamy tylko 10 klas.