1. Program roboczy

Gotowy program składa się z trzech plików: main.py – główny moduł programu, z którego uruchamiane jest testowanie, Model.py – plik z modelem, który wykonuje wszystkie obliczenia, model\_test.py – plik z kodem testu jednostkowego dla programu

1. OOP

Wszystko jest napisane przy użyciu klas, nawet testów.

Główny moduł programu został napisany w paradygmacie OOP, czyli został zaimplementowany jako klasa z metodami.

Również moduł testowy został zaimplementowany z wykorzystaniem dziedziczenia z biblioteki unittest

1. Git Hub

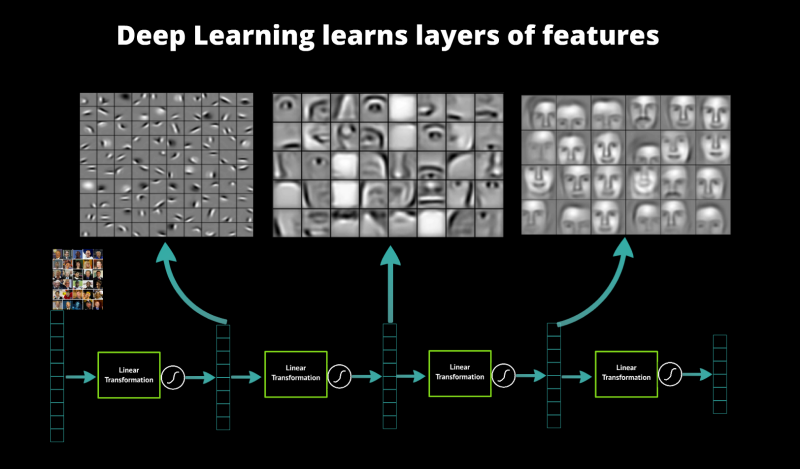
<https://github.com/gelious23-rgb/uczenie-maszynowe.git>

1. Analiza danych wyjściowych

Jako dane początkowe bierzemy zestaw danych składający się z obrazów psów i kotów. Zbiór danych musi zawierać co najmniej 15 000 obrazów. Wszystkie obrazy muszą być w tym samym formacie.

1. Feature engineering

Aby nasza sieć neuronowa nauczyła się odróżniać psy od kotów, musi określić, czym się różnią. Trening ten odbywa się dzięki temu, że sieć neuronowa tworzy warstwy z pewnymi szczegółami lub prymitywami graficznymi, przykład pokazano na obrazku poniżej.



Ponadto, aby poprawić wydajność algorytmu, można użyć gotowe warstwy bazowe.

1. Przygotowanie zestawów treningowych i testowych

Jako dane szkoleniowe bierzemy zestaw danych „Cats\_vs\_dogs” firmy Microsoft, składający się z obrazów psów i kotów. Zbiór danych składa się z 25 000 zdjęć kotów i psów. Wszystkie obrazy są już w tym samym formacie, więc nie ma potrzeby konwertowania do jednego formatu, ale rozmiary obrazów są różne, więc zmieniamy wszystkie rozmiary na 224x224 (ten rozmiar został wybrany ze względu na możliwość późniejszej optymalizacji).

W ramach testu inny zestaw danych o tej samej charakterystyce, tylko liczba obrazów będzie mniejsza.

1. Trening modelu

Trening modelu zaczynamy od zbudowania konwolucyjnego modelu do treningu (ponieważ pozwala analizować obrazy testowe w częściach, a nie całości na raz, jak zwykły perceptron). Początkowo trening rozpoczynamy bez warstw bazowych iz domyślnymi parametrami, przez okres 7 epok.



W efekcie otrzymujemy wynik na poziomie 60% dokładności dla 7 epok, z czego możemy wywnioskować, że sieć neuronowa jest trenowana długo i nieefektywnie.

1. Korzystanie z alternatywnych klasyfikatorów i parametrów (fine tuning)

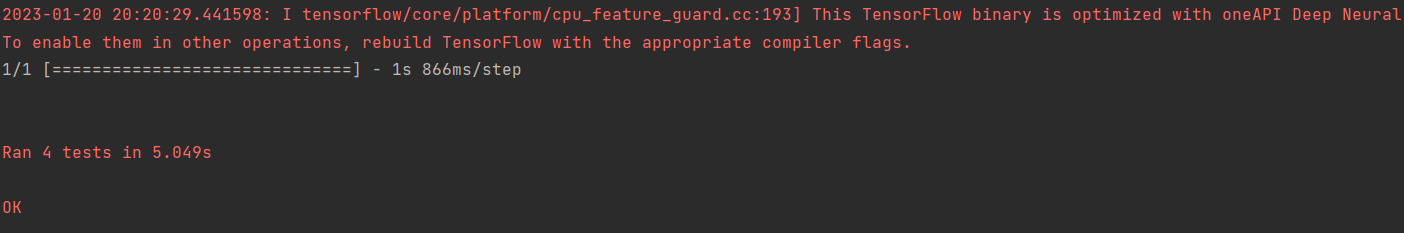
Aby poprawić dotychczasowe wyniki, zdecydowano się na dodanie warstw bazowych. W naszym przypadku wzięliśmy architekturę MobileNetV2 wbudowaną w tensorflow. Ta baza warstw zawiera zróżnicowaną bazę obrazów tysięcy obiektów. Dodaliśmy również wyłączenie 20% neuronów w każdej epoce, aby sieć neuronowa nie zapamiętywała obrazów, ale uczyła się na ich podstawie.

Warstwy wymagają do działania obrazów 224 x 224, dlatego przed treningiem musimy przekonwertować wszystkie obrazy do tej rozdzielczości.

1. Testy jednostkowe

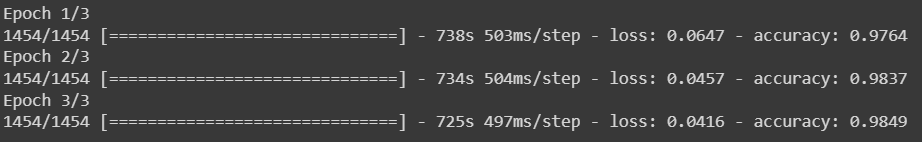
Testy jednostkowe zostały napisane dla głównych funkcji, kod i wyniki przedstawiono poniżej.

class ModelTest(TestCase):  
 def test\_predict\_one(self):  
 m = model.Model()  
 m.load\_trained(r'D:\py\ML\_work\trained\_models\model1.h5')  
 self.assertEqual(m.predict\_one(r'D:\py\ML\_work\test\_images\cats\cat.4003.jpg'), ('КОТ', 'D:\\py\\ML\_work\\test\_images\\cats\\cat.4003.jpg'))  
  
 def test\_predict\_file\_not\_found(self):  
 with self.assertRaises(FileNotFoundError) as e:  
 m = model.Model()  
 m.load\_trained(r'D:\py\ML\_work\trained\_models\model1.h5')  
 m.predict\_one(r'1.jpg')  
 self.assertEqual(2, e.exception.args[0])  
  
 def test\_load\_model(self):  
 m = model.Model()  
 m.load\_trained(r'D:\py\ML\_work\trained\_models\model1.h5')  
 self.assertIsNotNone(m.\_model)  
  
 def test\_load\_model\_no\_file(self):  
 with self.assertRaises(OSError) as e:  
 m = model.Model()  
 path = r'D:\py\ML\trained\_models\model1.h5'  
 m.load\_trained(path)  
 self.assertEqual(f'No file or directory found at {path}', e.exception.args[0])



1. Analiza metryk oceny modeli i ocena wyników

Po dodanych usprawnieniach współczynnik dokładności sieci neuronowej wzrósł do 98,49% już po 3 epokach treningowych, co jest całkiem dobrym wynikiem. Efekty uczenia się przedstawiono poniżej.



Aby sprawdzić wyniki, pobrano testowy zestaw 1000 zdjęć psów i 1000 zdjęć kotów. Stworzono również metodę testowania test, której kod podano poniżej.

def test(self):  
 dir\_path\_cats = r'test\_images\cats'  
 dir\_path\_dogs = r'test\_images\dogs'  
 mistakes = 0  
 m = Model()  
 # cats  
 for file in os.listdir(dir\_path\_cats):  
 if os.path.isfile(os.path.join(dir\_path\_cats, file)):  
 img\_path = os.path.join(dir\_path\_cats, file)  
 img = load\_img(img\_path)  
 img\_array = img\_to\_array(img)  
 \_ = None  
 img\_resized, \_ = self.resize\_image(img\_array, \_)  
 img\_expended = np.expand\_dims(img\_resized, axis=0)  
 prediction = self.\_model.predict(img\_expended)[0][0]  
 pred\_label = 'КОТ' if prediction < 0.5 else 'СОБАКА'  
  
 if pred\_label == 'СОБАКА':  
 mistakes += 1  
  
 # dogs  
 for file in os.listdir(dir\_path\_dogs):  
 if os.path.isfile(os.path.join(dir\_path\_dogs, file)):  
 img\_path = os.path.join(dir\_path\_dogs, file)  
 img = load\_img(img\_path)  
 img\_array = img\_to\_array(img)  
 \_ = None  
 img\_resized, \_ = self.resize\_image(img\_array, \_)  
 img\_expended = np.expand\_dims(img\_resized, axis=0)  
 prediction = self.\_model.predict(img\_expended)[0][0]  
 pred\_label = 'КОТ' if prediction < 0.5 else 'СОБАКА'  
  
 if pred\_label == 'КОТ':  
 mistakes += 1  
  
 return (2000 - mistakes) \* 100 / 2000 # accuracy

W wyniku testów otrzymaliśmy wynik z 98,95% dokładnością.

