Rzeszów, 23.05.2024

Sztuczna Inteligencja

Raport z wykonania projektu:

Analiza sieci neuronowej CNN, rozpoznającej zdjęcia psów i kotów

Autor:

Maciej Nabożny

173678

2 EF-DI L06

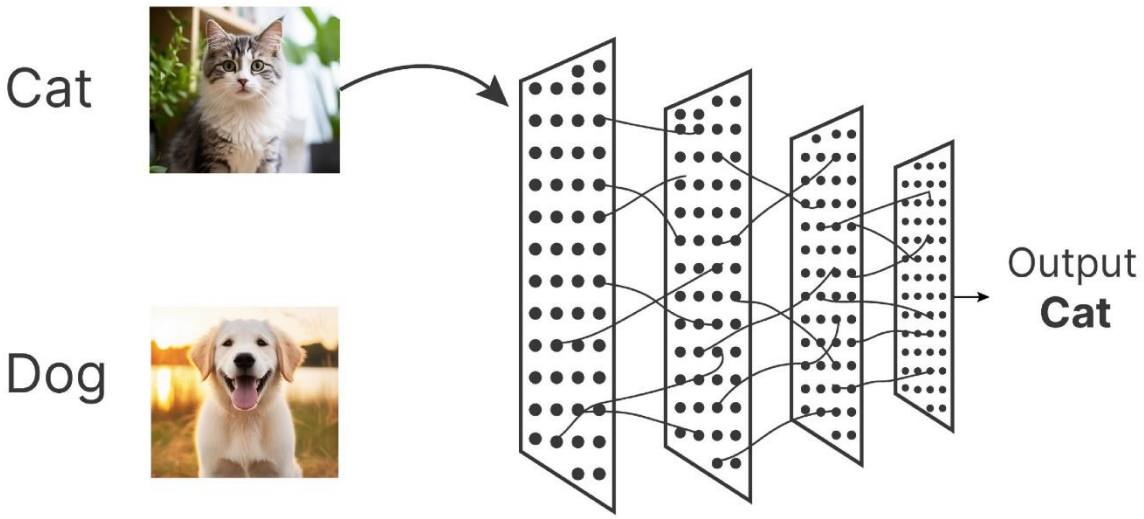
Spis treści

1 Wstęp

Algorytm CNN (Convolutional Neural Network) to rodzaj sztucznej sieci neuronowej, specjalizującej się w analizie danych o strukturze siatki, takich jak obrazy czy filmy, jego uproszczony schemat działania możemy zaobserowawać na Rysunku 1.1. Główna koncepcja CNN polega na wykorzystaniu warstw konwolucyjnych, które stosują filtry (jądra konwolucyjne) do ekstrakcji istotnych cech z danych wejściowych. W procesie tym, warstwy te przekształcają obraz w mapy cech, które są następnie poddawane operacjom nieliniowym, takim jak funkcje aktywacji (np. ReLU). Dodatkowe warstwy, takie jak warstwy poolingowe, redukują wymiary map cech, co zwiększa efektywność obliczeń i odporność na przesunięcia w danych. Na końcu sieci znajduje się warstwa w pełni połączona, która integruje wydobyte cechy i dokonuje klasyfikacji lub innej formy predykcji. Dzięki swojej strukturze i zdolności do automatycznego wykrywania i hierarchicznego reprezentowania cech, CNN stały się kluczowym narzędziem w zadaniach takich jak rozpoznawanie obrazów, analiza wideo i przetwarzanie języka naturalnego.

Problematyka projektu składa się z dogłębnego przedstawienia algorytmu CNN, analizy kodu źródłowego napisanego w języku Python i użytych w nim bibliotek takich jak np. Tensorflow oraz przedstawienie przykładowych wyników tego programu.

Program implementuje model CNN (Convolutional Neural Network) do klasyfikacji obrazów kotów i psów. Wykorzystując biblioteki TensorFlow i Keras, model przetwarza dane wejściowe, które są następnie dzielone na zestawy treningowe i testowe. Program normalizuje obrazy, przekształcając je do postaci odpowiedniej dla sieci neuronowej. Model składa się z kilku warstw konwolucyjnych, poolingowych i w pełni połączonych, co pozwala na ekstrakcję i analizę istotnych cech z obrazów. Po treningu modelu na zestawie treningowym, jego wydajność jest oceniana na zestawie testowym, a wyniki są wizualizowane za pomocą wykresów dokładności i stratności. Dodatkowo, program umożliwia predykcję klasy (kot lub pies) dla nowych, niewidzianych wcześniej obrazów.



Rysunek 1.1Obraz przedstawiajacy uproszczony schemat modelu sieci neuronowej. [analyticsvidhya[1]]

2 Dane

Bazą danych użytą w opisywanym algorytmie CNN jest **Cats vs Dogs** [14] która zawiera zdjęcia łącznie 25.000 psów i kotów, która jest podzielona na 2 foldery train i test:

- **Train** – zawiera 20.000 zdjęć psów i kotów z rozszerzeniem .jpg, jest to baza zdjęć służąca do trenowania modelu sieci
- **Test** - zawiera 5.000 zdjęć psów i kotów z rozszerzeniem .jpg, jest to baza zdjęć służąca do testowania modelu sieci

Zostały wczytane za pomocą skryptu w języku Python, który bazuje na bibliotece **Tensorflow:**

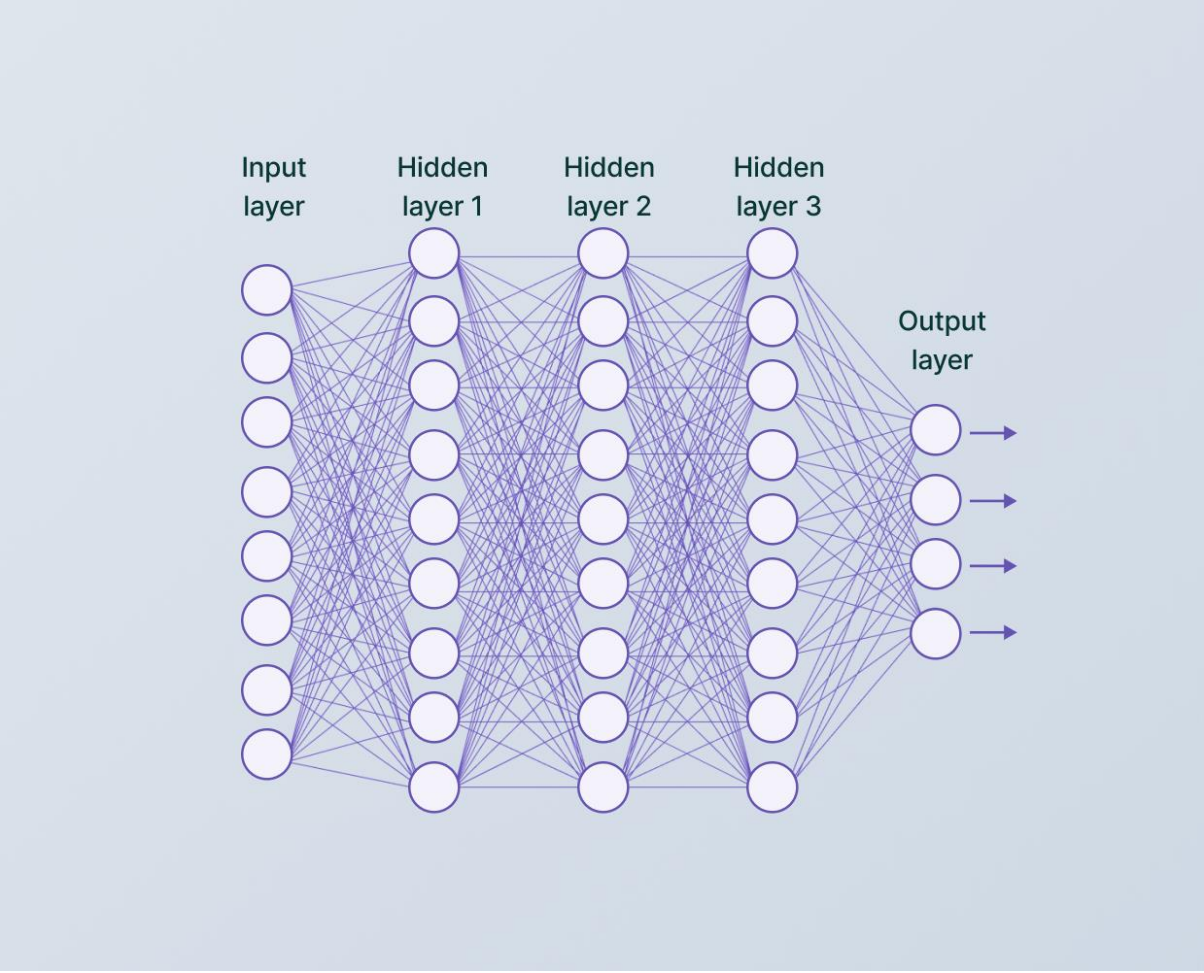
```
# Wczytywanie danych treningowych i testowych
train_ds = tf.keras.utils.image_dataset_from_directory(
   directory='./dogs_vs_cats/train',
   labels='inferred',
   label_mode='int',
   batch_size=32,
   image_size=(256, 256)
)

test_ds = tf.keras.utils.image_dataset_from_directory(
   directory='./dogs_vs_cats/test',
   labels='inferred',
   label_mode='int',
   batch_size=32,
   image_size=(256, 256)
)
# Funkcja normalizująca obrazy
def process(image, label):
   image = tf.cast(image / 255., tf.float32)
   return image, label

train_ds = train_ds.map(process)
test_ds = test_ds.map(process)
```

Listing 2.1Skrypt normalizujący. [opracowanie własne]

Skrypt z Listingu 2.1 wczytuje obrazy z dwóch katalogów **('train' i 'test')** i tworzy z nich zestawy danych, które są następnie używane do trenowania i testowania modelu uczenia maszynowego. Zestawy danych są wczytywane w „batchach”(podzbiórach danych treningowych / testowych) po 32 obrazy, a każdy obraz jest skalowany do rozmiaru 256x256 pikseli. Po czym na zmiennych przechowujących dane treningowe i testowe, wywoływana jest funkcja **process,** która normalizuje wartości pikseli obrazu i zmienia ich typ na tf.float32, a następnie zwraca przetworzony obraz wraz z oryginalną etykietą.

3 Algorytm CNN [Convolutional Neural Network]

Konwolucyjne Sieci Neuronowe (CNN) są zaawansowanym typem algorytmu głębokiego uczenia, który jest szczególnie skuteczny w analizie i przetwarzaniu danych obrazowych, takich jak zdjęcia czy filmy. Wykorzystują one hierarchiczną architekturę, gdzie kolejne warstwy uczą się coraz bardziej złożonych reprezentacji danych, od prostych krawędzi po skomplikowane obiekty. Architektura sieci CNN składa się z kliku rodzaju warstw, które współpracują, aby wydobywać i przetwarzać cechy z danych wejściowych. Można je podzielić na bardziej „ogólne" warstwy takie jak: **warstwa wejsciowa, warstwy ukryte oraz warstwa wyjściowa** (możemy je zaobserwować na Rysunku 3.1).



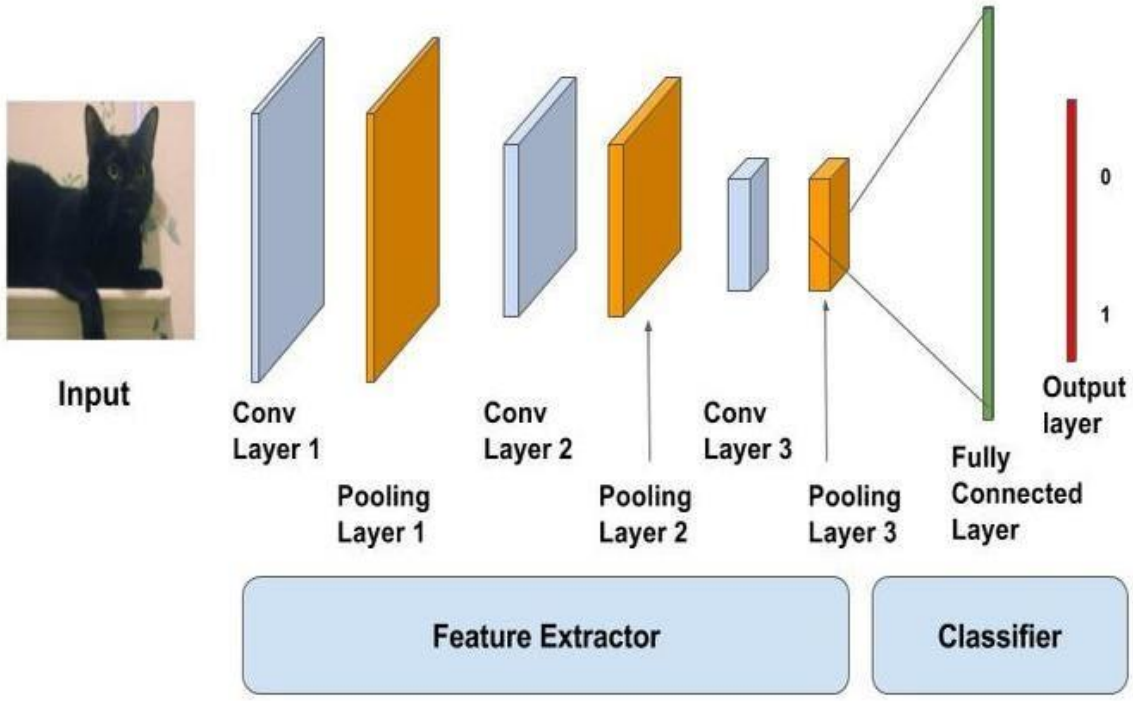
Rysunek 3.1 Warstwy sieci neuronowych CNN. [v7labs [2]]

Warstwa wejściowa przyjmuje surowe dane wejściowe, takie jak obrazy. Każdy piksel obrazu jest reprezentowany jako wartość w tej warstwie. Na przykład, dla kolorowego obrazu 256x256, warstwa wejściowa będzie miała 256x256x3 neuronów, gdzie 3 oznacza trzy kanały koloru (RGB).

Warstwa wyjściowa generuje ostateczne przewidywania na podstawie przetworzonych cech. W zależności od zadania, może to być pojedynczy neuron (dla klasyfikacji binarnej) lub wiele neuronów (dla klasyfikacji wieloklasowej). W tej warstwie często stosuje się funkcje

aktywacyjne, takie jak softmax (dla klasyfikacji wieloklasowej) lub sigmoid (dla klasyfikacji binarnej), aby uzyskać prawdopodobieństwa przynależności do poszczególnych klas.

Warstwy ukryte to główna część sieci CNN, gdzie następuje przetwarzanie i ekstrakcja cech. Są one podzielone na różne rodzaje, które odpowiadają za inną funkcjonalność.[5]



Rysunek 3.2 Obraz ukazujący bardziej dogłębny podział sieci neuronowej CNN. [medium [3]]

Na powyższym rysunku (Rysunek 3.2), jesteśmy w stanie dostrzec dokładniejszy podział architektury sieci CNN.

Warstwy konwolucyjne stosują filtry do wykrywania lokalnych wzorców w danych wejściowych poprzez operacje konwolucji, co skutkuje powstaniem **map cech**, na której używa się funkcji aktywacyjnych, które wprowadzają do nich nieliniowość. [6]Warstwy łączące (poolingowe) redukują wymiary danych i minimalizują ryzyko nadmiernego dopasowania, często poprzez operacje maksymalne łączenie (max pooling) lub średnie łączenie (average pooling). [11] Warstwy w pełni połączone (fully connected) łączą wszystkie neurony z poprzedniej warstwy z każdym neuronem w bieżącej warstwie, co umożliwia globalną integrację informacji i podejmowanie decyzji na podstawie całościowych danych wejściowych. Przed warstwą w pełni połączoną (fully connected) często dodaje się warstwę spłaszczającą (flattening), która przekształca wielowymiarowe dane wyjściowe z poprzednich warstw do jednowymiarowego wektora, umożliwiając w pełni połączonym warstwom integrację informacji i podejmowanie decyzji na podstawie całościowych danych wejściowych.[11]

Bardziej dokładne przedstawienie powyżej wymienionych warstw będzie opisane w następnych podsekcjach.

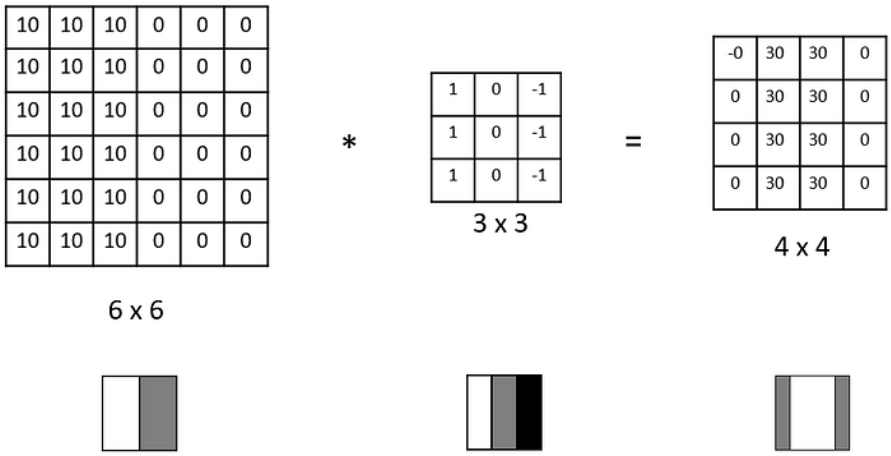
3.1 Warstwa Konwolucyjna (Convolutional Layer)

Warstwa konwolucyjna w sieciach neuronowych przetwarza dane wejściowe poprzez konwolucję, co pozwala na wyodrębnienie istotnych cech z obrazu.

Wprowadzenie warstwy konwolucyjnej zmienia kształt danych wejściowych z tensora o wymiarach (liczba wejść) × (wysokość wejścia) × (szerokość wejścia) × (liczba kanałów wejściowych) na tensor o wymiarach (liczba wejść) × (wysokość mapy cech) × (szerokość mapy cech) × (liczba kanałów mapy cech). Każda konwolucyjna jednostka przetwarza dane tylko dla swojego pola recepcyjnego, co jest analogiczne do reakcji neuronu w korze wzrokowej na określony bodziec.

Warstwy konwolucyjne redukują liczbę wolnych parametrów w porównaniu do w pełni połączonych warstw, co pozwala na budowę głębszych sieci neuronowych. Dla obrazu o rozmiarze 100 × 100, w pełni połączona warstwa wymagałaby 10,000 wag dla każdego neuronu w drugiej warstwie, podczas gdy konwolucja z regionem o wymiarach 5 × 5 wymaga jedynie 25 neuronów. [5]

Przykład mnożenie macierzy obrazu (danych wejściowych) i filtru w sieci CNN (Rysunek 3.3):



Rysunek 3.3 Grafika ukazująca mnożenie macierzy obrazu z filtrem sieci CNN cz.1. [medium [4]]

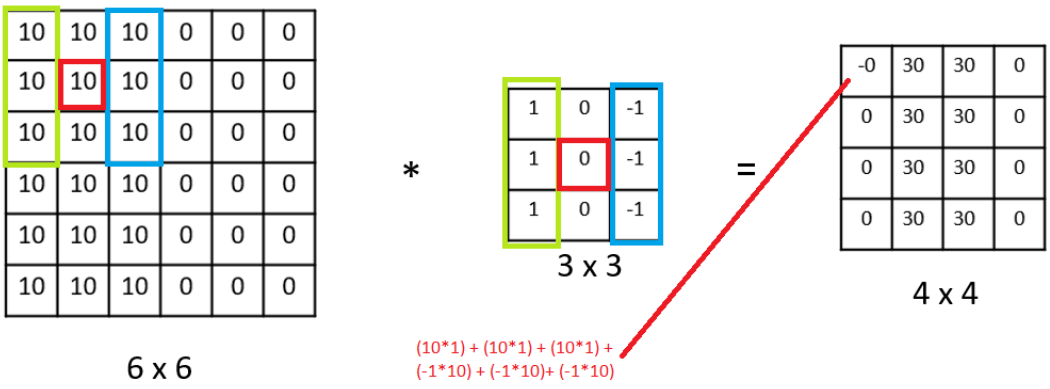
Powyżej mamy przedstawienie mnożenia macierzy orazu o rozdzielczosci 6x6 z filtrem o rozmiarze 3x3.

Aby obliczyć rozmiar wyjściowej mapy cech należy użyć wzoru [16] :

$$\boldsymbol{R = \frac{W - F + 2F}{S} + 1}$$

Gdzie:

- R – rozmiar wysciowy mapy cech
- W – rozmiar wejściowej macierzy (jedna wartość)
- F – rozmiar filtra
- P – padding
- S - stride



Rysunek 3.4 Grafika ukazująca mnożenie macierzy obrazu z filtrem sieci CNN cz.2. [medium [4]]

Mnożenie takich macierzy polega na przesuwaniu się macierzy filtru po wejściowym obrazie, gdzie mnoży odpowiadające sobie części macierz po czym je sumuje i wynik takiej operacji zostaje zapisany w określonym polu **mapy cech,** co można zaobserwować na Ryskunku 3.4. Początkowe wartości są ustawiane losowo, a następnie są dopasowywane podczas dalszych etapów uczenia się sieci. Proces ten wykorzystuje propagację wsteczną, która oblicza gradienty błędów w stosunku do wag. Następnie zastosowanie zejścia gradientowego pozwala na aktualizację wag w celu zminimalizowania funkcji kosztu. Dzięki temu sieć uczy się optymalnych wag, które najlepiej wykrywają wzorce w danych wejściowych.[2] [6]

3.2 Funckja Aktywacji (Activation Function)

Z każdym stworzeniem mapy cech, trzeba na niej użyć **ReLU (Rectified Linear Unit**), jest to funkcja aktywacji używana w sieciach neuronowych, w szczególności w konwolucyjnych sieciach neuronowych (CNN), która jest definiowana jako:

$$\boldsymbol{ReLU(x) = \max(0, x)}$$

Gdzie:

- **x oznacza wartość danej wejściowej**

Jeśli $0 < x$, funkcja zwraca x, a jeśli $0 \geq x$, funkcja zwraca 0. ReLU jest stosowane z kilku powodów: wprowadza nieliniowość do modelu, co jest istotne dla modelowania złożonych zależności, ma tendencję do prowadzenia do sparsity w aktywacjach, co może prowadzić do bardziej efektywnego obliczeniowo modelu i pomagać w unikaniu przeuczania, oraz jest bardzo efektywna obliczeniowo, ponieważ jej obliczanie jest proste i szybkie w porównaniu do innych funkcji aktywacji, takich jak sigmoida czy tanh.

Obliczenie ReLU polega na zastosowaniu definicji :

$$\boldsymbol{ReLU(x_{ij}) = \max(0, x_{ij})}$$

Zastosowanie tej definicji do każdej wartości w macierzy wyników z warstwy konwolucyjnej lub w pełni połączonej, co prowadzi do wynikowej macierzy o tych samych wymiarach,

ale z wartościami przekształconymi według funkcji **ReLU**; na przykład, jeśli mamy macierz wyników z poprzedniej warstwy:

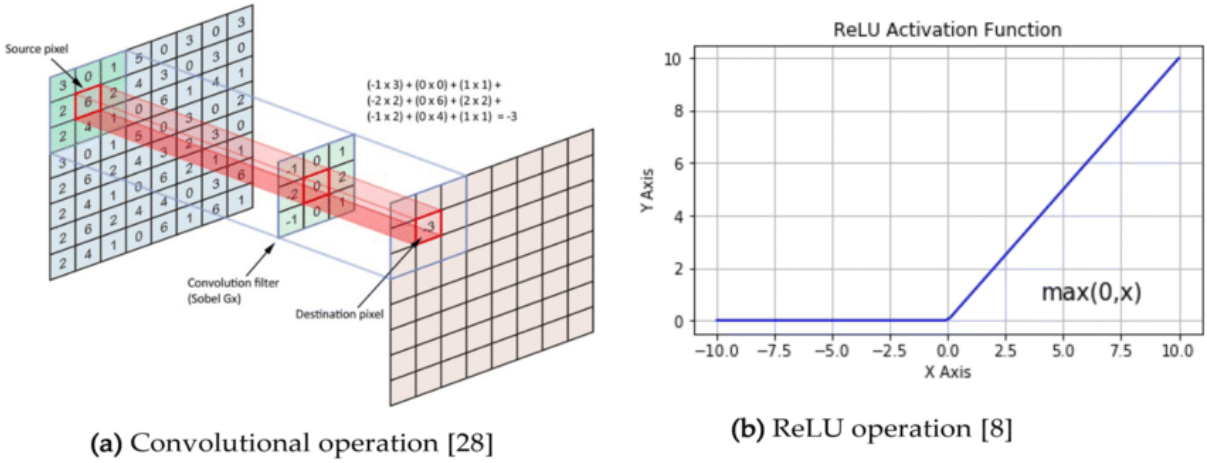
$$\boldsymbol{X} = \begin{bmatrix} \mathbf{-1} & \mathbf{2} & \mathbf{-3} \\ \mathbf{4} & \mathbf{-5} & \mathbf{6} \end{bmatrix}$$

Po zastosowaniu funkcji ReLU mamy:

$$\boldsymbol{X} = \begin{bmatrix} \mathbf{0} & \mathbf{2} & \mathbf{0} \\ \mathbf{4} & \mathbf{0} & \mathbf{6} \end{bmatrix}$$

ReLU stosuje się najczęściej po każdej warstwie konwolucyjnej lub w pełni połączonej, przed kolejnymi operacjami lub warstwami, ponieważ pomaga w propagacji gradientów w trakcie treningu, co jest korzystne dla głębokich sieci, redukując problem zanikania gradientu, który może wystąpić przy użyciu innych funkcji aktywacji, takich jak sigmoida czy tanh.[7]

Wykres funkcji aktywacji ReLU widoczny jest na Rysunku 3.5.



(a) Convolutional operation [28]

(b) ReLU operation [8]

Rysunek 3.5Grafika obrazująca ReLU. [researchgate [8]]

3.3 Warstwa Łącząca (Pooling Layer)

Warstwa łączeniowa, znana również jako warstwa poolingowa, jest kluczowym komponentem w architekturze sieci konwolucyjnej (CNN), której głównym zadaniem jest redukcja wymiarów danych wejściowych poprzez agregację informacji w małych regionach mapy cech, co zmniejsza liczbę parametrów i obliczeń w sieci, a także pomaga w uogólnieniu modelu i zapobieganiu przeuczaniu. Stosowane są 2 główne typy wartw łączących, **Max Pooling** oraz **Average Pooling** (ich wizualizacja widoczna jest na rysuknu 3.6).

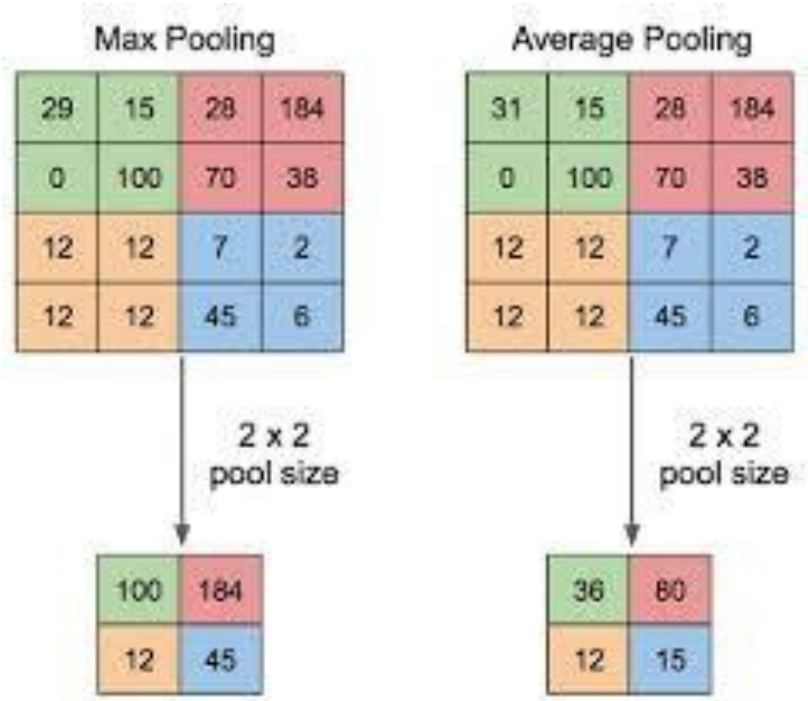
Max Pooling i Average Pooling to dwa główne rodzaje operacji łączeniowych stosowanych w warstwach poolingowych konwolucyjnych sieci CNN, które służą do redukcji wymiarów map cech.

Max Pooling polega na wybieraniu maksymalnej wartości z określonego obszaru (np. 2×3×3) w mapie cech i przesuwaniu się o określony krok (stride), co skutkuje utrzymaniem

najbardziej aktywowanych cech w danym obszarze. To sprawia, że Max Pooling jest skuteczny w ekstrakcji istotnych cech, zachowując przy tym lokalną strukturę obrazu.

Z kolei, **Average Pooling** oblicza średnią wartość z określonego obszaru w mapie cech. Ta operacja ma tendencję do wygładzania mapy cech, redukując jej wymiary, ale zachowując informacje o uśrednionych cechach w danym obszarze.

Porównując oba podejścia, Max Pooling skupia się na wybieraniu najbardziej istotnych cech w danym obszarze, podczas gdy Average Pooling bardziej uśrednia cechy w obszarze, co może być użyteczne w przypadku, gdy chcemy bardziej ogólnie analizować obszar cech. [10]



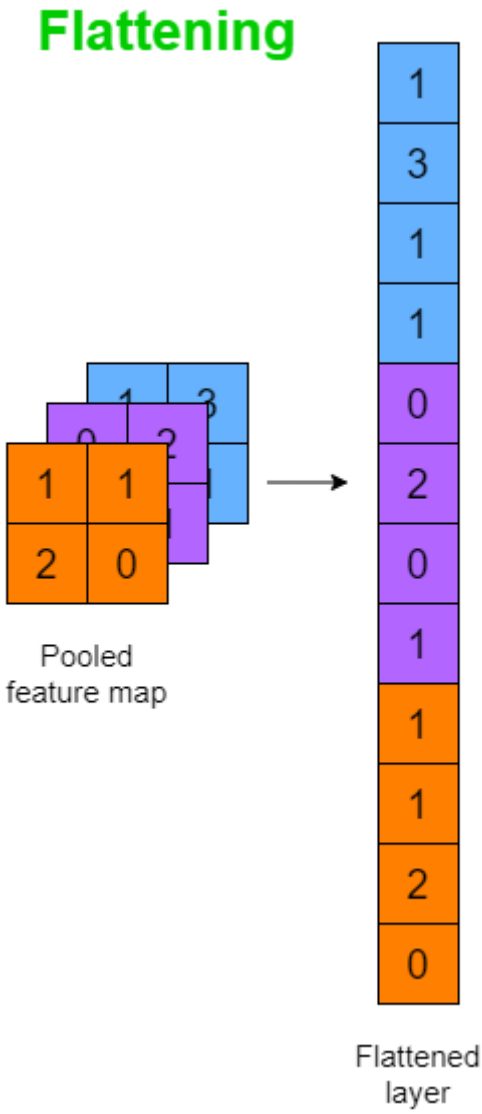
Rysunek 3.6Grafika obrazująca Max pooling i Average Pooling. [researchgate [9]]

3.4 Warstwa Słaszczająca (Flattening Layer)

Warstwa spłaszczająca, zwana również warstwą Flatten, jest używana w konwolucyjnych sieciach neuronowych (CNN) do przekształcenia wielowymiarowych map cech na płaski wektor, który może być przekazany do warstw w pełni połączonych. Jej działanie polega na zamianie każdego piksela w mapach cech na pojedynczą wartość w wektorze wyjściowym. Na przykład, jeśli wejściowa mapa cech ma wymiary n × m × d, gdzie n i m to wymiary przestrzenne, a d to liczba kanałów, warstwa spłaszczająca przekształca tę mapę w jednowymiarowy wektor o długości n × m × d.

Wizualizacja tego precesu widoczna jest na Rysunku 3.7.

Ten proces jest istotny, ponieważ pozwala na przekształcenie map cech z warstw konwolucyjnych do postaci, która może być przetwarzana przez warstwy w pełni połączone, co umożliwia klasyfikację lub regresję na podstawie wyodrębnionych cech. Warstwa spłaszczająca jest zwykle stosowana przed warstwą w pełni połączoną w końcowych etapach CNN. [11]



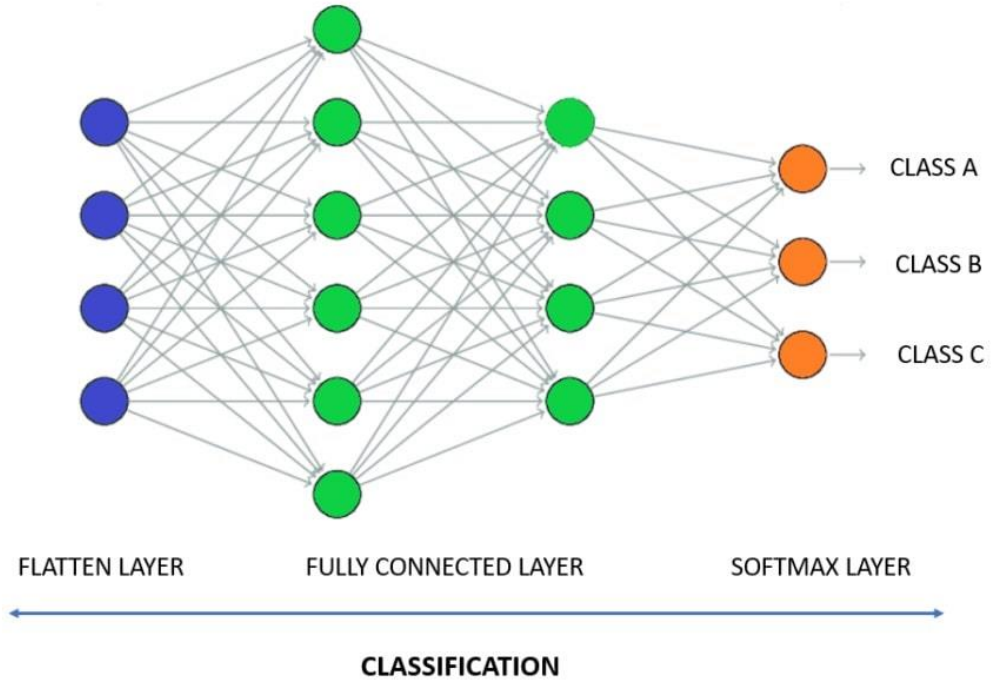
Rysunek 3.7 Wizualizacja Warstwy Spłaszczającej. [towardsdatascience [11]]

3.5 Warstwa w Pełni Połączona (Fully connected Layer)

Warstwa w pełni połączona jest kluczowym elementem konwolucyjnych sieci neuronowych (CNN), który integruje cechy wyekstrahowane z poprzednich warstw, aby przeprowadzić klasyfikację lub regresję. Każdy neuron w warstwie w pełni połączonej jest połączony z każdym neuronem z poprzedniej warstwy, tworząc kompletnie połączoną sieć neuronową. Warstwa ta pełni rolę klasyfikatora, decydując o przynależności obrazu do poszczególnych klas.

Po obliczeniu sumy ważonych wejść, zazwyczaj stosuje się funkcję aktywacji, np. ReLU, dla każdego neuronu w warstwie. Ostatnim krokiem w warstwie w pełni połączonej często jest zastosowanie funkcji **softmax**, która normalizuje wyjścia neuronów, przekształcając je w prawdopodobieństwa przynależności do poszczególnych klas. **Softmax** jest szczególnie przydatny w problemach wieloklasowej klasyfikacji, ponieważ przekształca wartości wyjściowe na rozkład prawdopodobieństwa, co ułatwia interpretację wyników. Jest ona określana wykresem:[12]

W przypadku problemów klasyfikacji, warstwa w pełni połączona z funkcją softmax na wyjściu jest zwykle ostatnią warstwą w modelu, której wyniki decydują o przynależności obrazu do poszczególnych klas. Warstwa w pełni połączona jest kluczowa zarówno dla problemów klasyfikacji, jak i regresji w sieciach neuronowych, ponieważ umożliwia modelowi uczenie się złożonych zależności między danymi wejściowymi a docelowymi wyjściami.[12]



Rysunek 3.9 Wizualizacja warstwy spłaszczającej, w pełni połączonej oraz Softmax. [indiantechwarrior [13]]

4 Model Sieci Neuronowej CNN

Model opisywanej sieci CNN importuje potrzebne biblioteki takie jak **keras**, która pomaga w ustalaniu kształtu danych wejściowych (obrazów). Jest ona używana również do wczytywania i normalizowania danych treningowych i testowych (jak można zaobserwować w 2 podpunkcie, gdzie dane testowe i treningowe zostają zapisane do zmiennych **train_ds** i **test_ds** z bazy danych **Cats vs Dogs**, co widoczne jest na listingu 4.1).

```
# Wczytywanie danych treningowych i testowych
train_ds = tf.keras.utils.image_dataset_from_directory(
    directory='./dogs_vs_cats/train',
    labels='inferred',
    label_mode='int',
    batch_size=32,
    image_size=(256, 256)
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    directory='./dogs_vs_cats/test',
    labels='inferred',
    label_mode='int',
    batch_size=32,
    image_size=(256, 256)
)

# Funkcja normalizująca obrazy
def process(image, label):
    image = tf.cast(image / 255., tf.float32)
    return image, label

train_ds = train_ds.map(process)
test_ds = test_ds.map(process)
```

Listing 4.1Skrypt normalizujący. [opracowanie własne]

Aby rozpocząć budowanie modelu musimy najpierw wywołać funkcję **Sequential () (**Listing 4.2 **)** na zmiennej **model**, która będzie przechowywać model naszej sieci. Wyżej wymieniona funkcja tworzy model, do którego można dodawać warstwy w kolejności, co umożliwia zbudowanie sieci neuronowej przez dodawanie kolejnych warstw jedna po drugiej.

```
# Tworzenie modelu CNN

model = Sequential()
```

Listing 4.2 Użycie Sequential. [opracowanie własne]

Biblioteka **keras** zawiera również funkcje, które ułatwiają implementowanie różnych warstw sieci neuronowej takie jak np. **warstwa konwolucyjna** czy **warstwa łącząca**, które możemy dodać do opisywanego modelu za pomocą metody **add()**, co można zobaczyć na listingu 4.3**.**

```
model.add(Input(shape=input_shape))

model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu'))
model.add(Conv2D(32, kernel_size=(3, 3), strides=(2, 2), padding='valid', activation='relu'))

model.add(AveragePooling2D(pool_size=(2, 2), strides=2, padding='valid'))

model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu'))
model.add(Conv2D(32, kernel_size=(3, 3), strides=(2, 2), padding='valid', activation='relu'))

model.add(AveragePooling2D(pool_size=(2, 2), strides=2, padding='valid'))

model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu'))
model.add(Conv2D(32, kernel_size=(3, 3), strides=(2, 2), padding='valid', activation='relu'))

model.add(AveragePooling2D(pool_size=(2, 2), strides=2, padding='valid'))

model.add(Flatten())

model.add(Dense(64, activation='relu'))
model.add(Dropout(0.1))

model.add(Dense(32, activation='relu'))
model.add(Dropout(0.1))

model.add(Dense(2, activation='softmax'))
```

Listing 4.3 Warstwy modułu. [opracowanie własne]

Na początku ustawiamy domyślny **shape** (kształt danych wejściowych) na wartość która została zapisana w zmiennej **input_shape** (listing 4.4):

```
input_shape = (256, 256, 3)
```

Listing 4.4 Wielkość „Inputu". [opracowanie własne]

Dodajemy do naszego modelu dodajemy:

- **Warstwę konwolucyjną** (Conv2D) z 32 filtrami, rozmiarem jądra 3x3, krokiem 1x1, wypełnieniem 'valid' i aktywacją 'relu', która wykonuje operację konwolucji na obrazie, wykrywając cechy lokalne.

- **Warstwę konwolucyjną** (Conv2D) z 32 filtrami, rozmiarem jądra 3x3, krokiem 2x2, wypełnieniem 'valid' i aktywacją 'relu', która również wykrywa cechy, ale z większym krokiem, zmniejszając rozmiar obrazu.

- **Warstwę średniego próbkowania** (AveragePooling2D) z rozmiarem puli 2x2, krokiem 2 i wypełnieniem 'valid', która redukuje rozmiar obrazu, uśredniając wartości w sąsiadujących pikselach.

- **Warstwę konwolucyjną** (Conv2D) z 32 filtrami, rozmiarem jądra 3x3, krokiem 1x1, wypełnieniem 'valid' i aktywacją 'relu', która ponownie wykrywa cechy lokalne.

- **Warstwę konwolucyjną** (Conv2D) z 32 filtrami, rozmiarem jądra 3x3, krokiem 2x2, wypełnieniem 'valid' i aktywacją 'relu', zmniejszającą rozmiar obrazu.

- **Warstwę średniego próbkowania** (AveragePooling2D) z rozmiarem puli 2x2, krokiem 2 i wypełnieniem 'valid', która dalej redukuje rozmiar obrazu przez uśrednianie.

- **Warstwę konwolucyjną** (Conv2D) z 32 filtrami, rozmiarem jądra 3x3, krokiem 1x1, wypełnieniem 'valid' i aktywacją 'relu', wykrywającą cechy lokalne.

- **Warstwę konwolucyjną** (Conv2D) z 32 filtrami, rozmiarem jądra 3x3, krokiem 2x2, wypełnieniem 'valid' i aktywacją 'relu', która zmniejsza rozmiar obrazu.

- **Warstwę średniego próbkowania** (AveragePooling2D) z rozmiarem puli 2x2, krokiem 2 i wypełnieniem 'valid', dalej redukującą rozmiar obrazu przez uśrednianie.

- **Warstwę spłaszczającą** (Flatten), która przekształca dane z 2D do 1D, przygotowując je do warstw gęstych.

- **Warstwę gęstą** (Dense) z 64 neuronami i aktywacją 'relu', która wprowadza nieliniowość i wykrywa skomplikowane wzorce.

- **Warstwę dropout** (Dropout) z wartością 0.1, która losowo wyłącza 10% neuronów, zapobiegając przeuczeniu.

- **Warstwę gęstą** (Dense) z 32 neuronami i aktywacją 'relu', zwiększającą zdolność modelu do wykrywania wzorców.

- **Warstwę dropout** (Dropout) z wartością 0.1, ponownie zapobiegającą przeuczeniu.

- **Warstwę gęstą** (Dense) z 2 neuronami i aktywacją 'softmax', która klasyfikuje obrazy do dwóch kategorii, zwracając prawdopodobieństwo przynależności do każdej z nich.

Po dodaniu powyższych warstw definiujemy zmienną **initial_learning_rate**, za pomocą której można ustawiać wartość stopy uczenia. W nastepnym kroku za pomocą **ExponentialDecay** definujemy harmonogram spadku stopy uczenia po czym używamy funckji **compile** aby zkompilować nasz model (listing 4.5).

```
initial_learning_rate = 0.001
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
   initial_learning_rate=initial_learning_rate,
   decay_steps=10000,
   decay_rate=0.9,
   staircase=True
)

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr_schedule),
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])
```

Listing 4.5Skrypt odpowiadający za „learning rate". [opracowanie własne]

Poszczególne parametry **ExponentialDecay** odpowiadają za:

- **Initial_learning_rate** - początkowa wartość stopy uczenia, która jest równa zmiennej o tej samej nazwie, w opisywanym przypadku wynosi ona 0.001.
- **Decay_steps** - określa co ile kroków (iteracji) ma nastąpić zmiana stopy uczenia. W opisywanym przypadku wynosi 10000.
- **Decay_rate** - współczynnik spadku, który określa o ile stopa uczenia ma być zmniejszana po każdych decay_steps. W opisywanem przypadku wynosi 0.9.
- **Staircase -** określa, czy spadek stopy uczenia ma być skokowy (True) czy ciągły (False). W opisywanem przypadku ma wartość True.

Powyższy fragment kodu definiuje harmonogram spadku stopy uczenia, który zmniejsza stopę uczenia eksponencjalnie co 10000 kroków, z współczynnikiem spadku równym 0.9, i spadkiem w sposób skokowy.

Poniżej w **model.compile:**

- **Optimizer** – dzięki biblioteki keras, learning_rate ma wartość zmiennej lr_schedule poprzez zmienienie jej przez optymalizator **Adam**, który pomaga w dostosowywaniu stopy uczenia się dla każdego parametru. Przyspiesza to proces uczenia się modelu.
- **Loss** – została ona ustawiona na '**sparse_categorical_crossentropy**', która oblicza stratę między etykietami a prognozowanymi prawdopodobieństwami dla klasy wyjściowej
- **Metrics** - została ustawiona na '**accuracy**', który jest wskaźnikiem oceny wydajności modelu podczas treningu.

W dalszej części modelu sieci neuronowej, definiujemy funkcję test_image, która jako agrumenty przyjmuje, model sieci, oraz ścieżkę do obrazu, który zostanie użyty do sprawdzenia poprawności sieci CNN (listing 4.6).

```
def test_image(model, image_path):
    test_img = cv2.imread(image_path)
    test_img = cv2.resize(test_img, (256, 256))
    test_input = test_img.reshape((1, 256, 256, 3))
    prediction = model.predict(test_input)
    predicted_class = np.argmax(prediction)
    class_labels = ['cat', 'dog']
    predicted_label = class_labels[predicted_class]
    print(f'Prediction for {image_path}:', prediction)
    print('Predicted label:', predicted_label)
    print("------")

# Testowanie różnych obrazów
image_paths = ['cat.1817.jpg', 'cat.194.jpg', 'dog.35.jpg', 'cat.1817.jpg', 'dog.119.jpg', 'dog.14.jpg', 'cat.77.jpg']
for image_path in image_paths:
    test_image(model, image_path)
```

Listing 4.6 Funkcja test_image i jej wywołanie. [opracowanie własne]

Powyższa funkcja ładuje obraz z podanej ścieżki, zmienia jego rozmiar na 256x256 pikseli, przekształca go w odpowiedni format wejściowy dla modelu, przewiduje klasę obrazu za pomocą wytrenowanego modelu, przypisuje etykietę 'cat' lub 'dog' na podstawie przewidywanej klasy, wyświetla wynik predykcji i przewidywaną etykietę, a następnie testuje tę funkcję na kilku obrazach z listy **image_paths**, wywołując **test_image** dla każdej ścieżki obrazu.

Przykładowe wywołanie **test_image**:

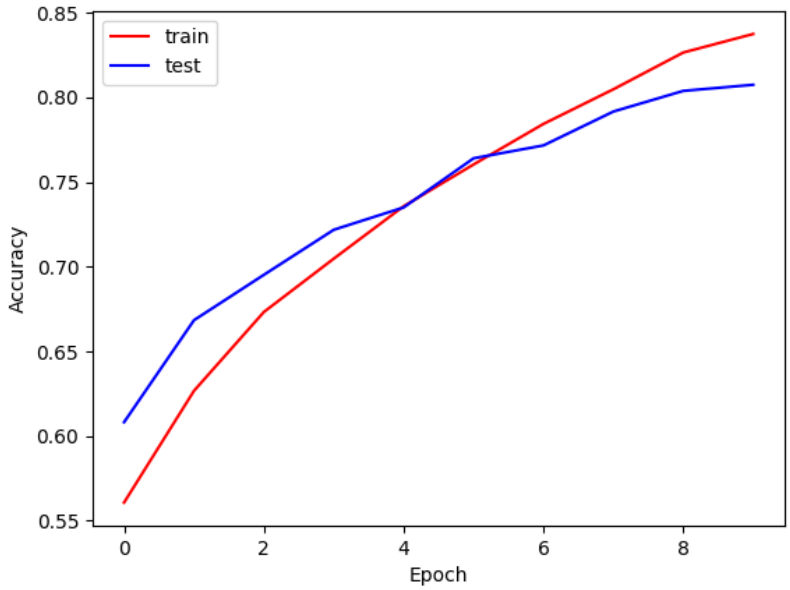
```
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
Prediction for dog.2.jpg: [[2.0346500e-04 9.9979657e-01]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
Prediction for dog.3.jpg: [[0. 1.]]
Predicted label: dog
------
```

Listing 4.7 Przykładowy wynik. [opracowanie własne]

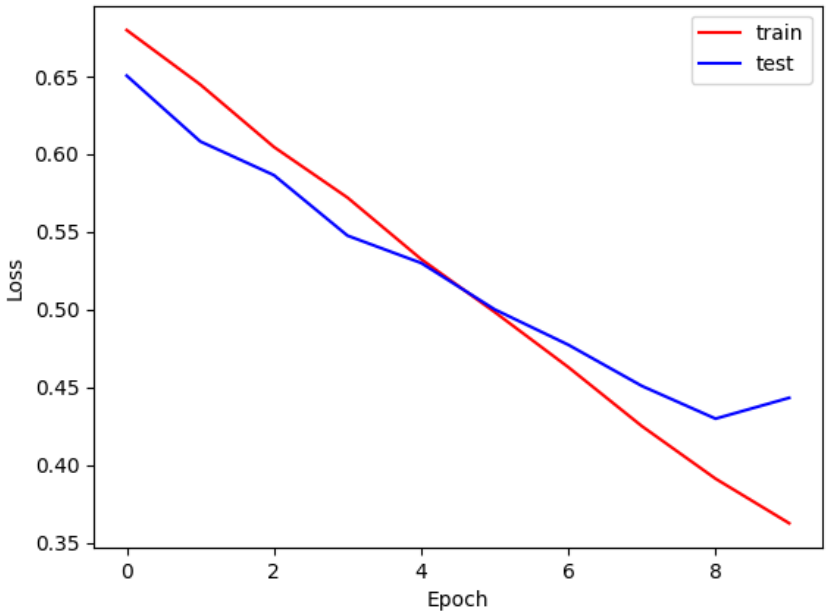
W końcowej części kodu za pomocą biblioteki **matplotlib,** program wyświetla 2 wykresy, jeden wypisujący krzywą dokładność (accuracy) uczenia się modelu na przestrzeni epok, a drugi krzywą straty (loss) na przestrzeni epok (każdy wykres przedstawia dwie krzywe dla

danych treningowych i testowych). Te wizualizacje pomagają ocenić, jak dobrze model uczy się i generalizuje na dane walidacyjne w miarę postępu treningu.

Przykładowe wykresy:



Listing 4.8 Przykładowy wykres „accuracy”. [opracowanie własne]



Listing 4.9 Przykładowy wykres „loss”. [opracowanie własne]

4.1 Dobór parametrów

W opisywanym modelu sieci neuronowej użyto **warstw konwolucyjnych** Conv2D z 32 filtrami i rozmiarem jądra 3x3, ponieważ jest to standardowy wybór dobrze równoważący dokładność detekcji cech i złożoność obliczeniową, pierwsza warstwa ma kroki przesunięcia (strides) (1,1), co oznacza przemieszczenie filtra o jeden piksel w obu kierunkach, a druga warstwa (2,2), co zmniejsza wymiar obrazu o połowę, przy paddingu 'valid', co oznacza stosowanie filtra tylko tam, gdzie mieści się w całości, zmniejszając wymiar obrazu.

Kolejne bloki **warstw konwolucyjnych** Conv2D stosują te same parametry, aby stopniowo redukować wymiary i zwiększać liczbę detekcji cech, zachowując spójność architektury. Warstwy AveragePooling2D z rozmiarem okna 2x2 i krokiem 2 uśredniają wartości w oknie, co może być bardziej odpowiednie dla danych, gdzie każda cecha jest ważna, skutecznie zmniejszając rozmiar danych i zachowując istotne cechy.

Warstwa Flatten przekształca dane z postaci 2D do 1D, co jest wymagane przed przekazaniem danych do warstw Dense. Warstwa Dense z 64 neuronami i aktywacją ReLU, standardową w większości architektur, oraz kolejna warstwa Dense z 32 neuronami, pomagają w stopniowej redukcji wymiarów i złożoności modelu. Stosowanie Dropout z prawdopodobieństwem 0.1 pomaga zapobiegać przeuczeniu modelu poprzez losowe wyłączanie 10% neuronów w czasie treningu.

Warstwa wyjściowa z 2 neuronami, odpowiadająca liczbie klas w problemie klasyfikacji, oraz aktywacja softmax konwertująca wyniki na prawdopodobieństwa dla każdej klasy.

Harmonogram uczenia z początkową szybkością 0.001, powszechnie używaną dla optymalizatora **Adam,** oraz **Exponential Decay**, który redukuje stopniowo szybkość uczenia w trakcie treningu, co pomaga w bardziej stabilnej i dokładnej konwergencji; decay steps określają, co ile kroków następuje zmniejszenie szybkości uczenia, a decay rate to mnożnik stosowany do zmniejszenia szybkości uczenia.

Kompilacja modelu używa optymalizatora **Adam**, dostosowując szybkość uczenia dla każdego parametru, z funkcją straty **sparse categorical crossentropy** dla problemów klasyfikacji wieloklasowej, gdzie etykiety są kodowane jako liczby całkowite, oraz metryką accuracy do oceny skuteczności modelu.

5 Badania

Opisywana sieć neuronowa zostanie poddana badaniom na podstawie których zostanie wywnioskowane jaka wielkość **learning rate** jest najbardziej optymalna dla modelu. Przeanalizowane zostaną końcowe wartości **accuracy** i **loss** dla każdego przykładu, wraz z trafnościa w określeniu czy na danym obrazie znajduje się pies bądź kot.

Obrazami na podstawie których model będzie wnioskował wynik są pliki z roszerzeniem .jpg:

Listing 5.1 cat_1.jpg [kaggle [15]]

Listing 5.2 cat_2.jpg [kaggle [15]]

Listing 5.2 cat_3.jpg [kaggle [15]]

Listing 5.4 dog_1.jpg [kaggle [15]]

Listing 5.5 dog_2.jpg [kaggle [15]]

Listing 5.5 dog_3.jpg [kaggle [15]]

5.1 Eksperyment I

Na początku w pierwszym ekperymencie sprawdzamy model sieci neuronowej z wartością zmiennej **initial_learning_rate** równą 0.001 oraz warstwą łączącą **AveragePooling**.

Po skończeniu nauki modelu otrzymujemy końcowe accuracy i loss:

```
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 167s 267ms/step - accuracy: 0.8713 - loss: 0.2973 - val_accuracy: 0.8174 - val_loss: 0.4342
------
```

Listing 5.6 Wynik eksperymentu nr 1 (accuracy, loss). [opracowanie własne]

Accuracy: 87.13%, Loss: 29.73%

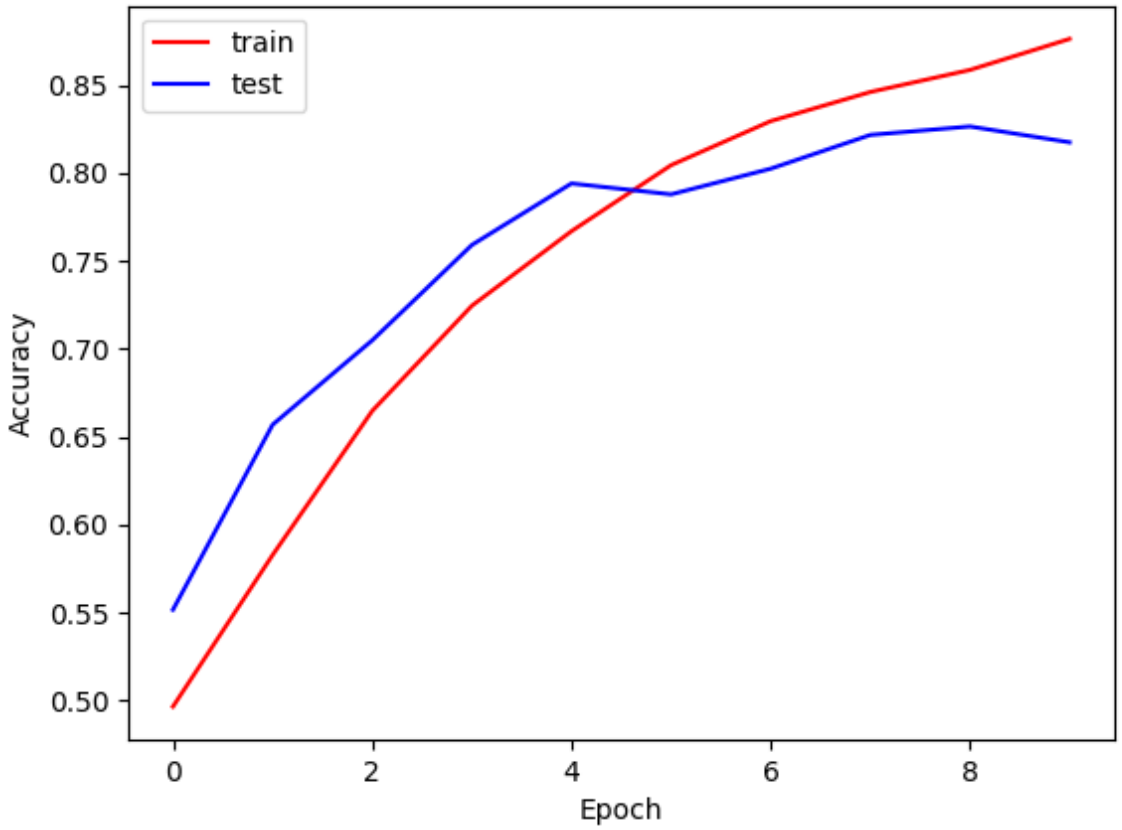
Wyniki predykcji psów i kotów:

```
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step
Prediction for cat.1.jpg: [[1. 0.]]
Predicted label: cat
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Prediction for cat.2.jpg: [[1.322486e-04 9.998677e-01]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
Prediction for cat.3.jpg: [[1. 0.]]
Predicted label: cat
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Prediction for dog.1.jpg: [[3.0850464e-10 1.0000000e+00]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
Prediction for dog.2.jpg: [[0. 1.]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
Prediction for dog.3.jpg: [[0. 1.]]
Predicted label: dog
------
```

Listing 5.7 Wynik eksperymentu nr 1 (predykcje). [opracowanie własne]

Otrzymane wyniki pokazują, że podczas predykcji wystąpił 1 błąd ,więc 5/6 zwierząt zostało poprawnie przydzielone.

Wykres accuracy na przestrzeni epok:



Listing 5.8 Wynik eksperymentu nr 1 (wykres). [opracowanie własne]

5.2 Eksperyment II

Na początku w pierwszym ekperymencie sprawdzamy model sieci neuronowej z wartością zmiennej **initial_learning_rate** równą 0.01 oraz warstwą łączącą **AveragePooling.**

Po skończeniu nauki modelu otrzymujemy końcowe accuracy i loss:

```
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 167s 267ms/step - accuracy: 0.4922 - loss: 0.6937 - val_accuracy: 0.5000 - val_loss: 0.6935
------
```

Listing 5.9 Wynik eksperymentu nr 2 (accuracy, loss). [opracowanie własne]

Accuracy: 49.22%, Loss: 69.37%

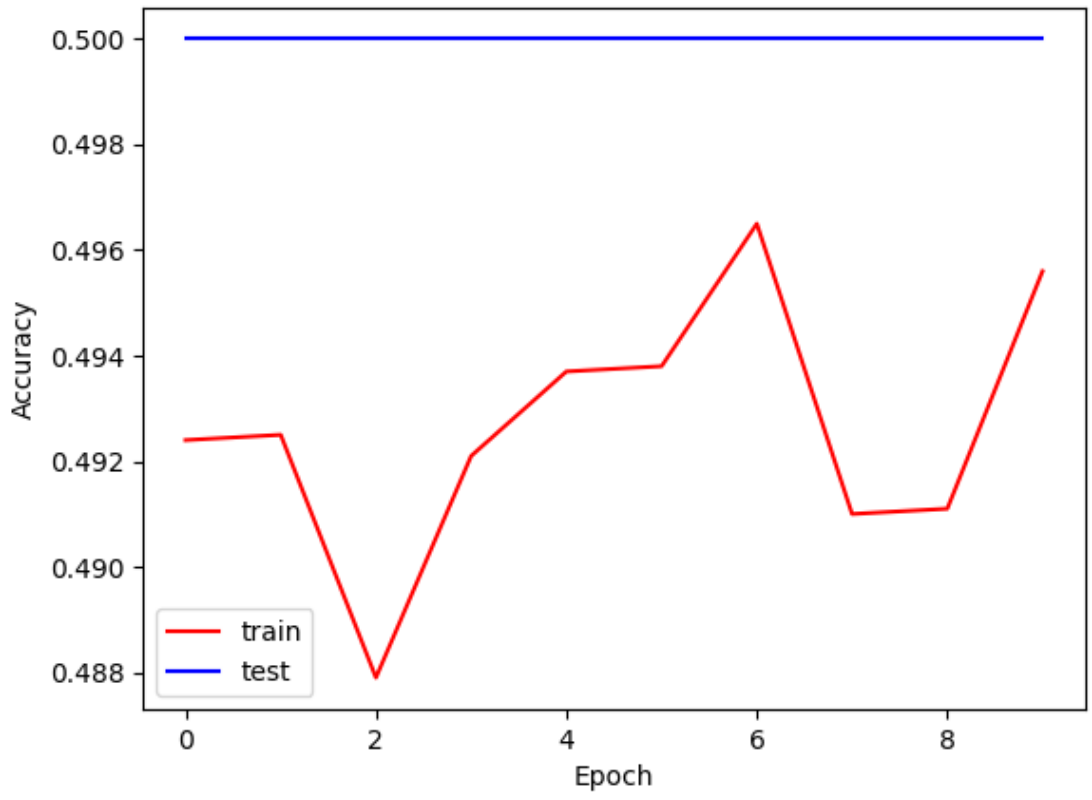
Wyniki predykcji psów i kotów:

```
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 122ms/step
Prediction for cat.1.jpg: [[0.4863028 0.5136972]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Prediction for cat.2.jpg: [[0.4863028 0.5136972]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
Prediction for cat.3.jpg: [[0.4863028 0.5136972]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Prediction for dog.1.jpg: [[0.4863028 0.5136972]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
Prediction for dog.2.jpg: [[0.4863028 0.5136972]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step
Prediction for dog.3.jpg: [[0.4863028 0.5136972]]
Predicted label: dog
------
```

Listing 5.10 Wynik eksperymentu nr 2 (predykcje). [opracowanie własne]

Otrzymane wyniki pokazują, że podczas predykcji wystąpił 3 błędy ,więc 3/6 zwierząt zostało poprawnie przydzielone.

Wykres accuracy na przestrzeni epok:



Listing 5.11 Wynik eksperymentu nr 1 (wykres). [opracowanie własne]

5.3 Eksperyment II

Na początku w pierwszym ekperymencie sprawdzamy model sieci neuronowej z wartością zmiennej **initial_learning_rate** równą 0.1 oraz warstwą łączącą **AveragePooling.**.

Po skończeniu nauki modelu otrzymujemy końcowe accuracy i loss:

```
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 163s 261ms/step - accuracy: 0.4920 - loss: 0.6973 - val_accuracy: 0.5000 - val_loss: 0.6968
------
```

Listing 5.12 Wynik eksperymentu nr 3 (accuracy, loss). [opracowanie własne]

Accuracy: 49.20%, Loss: 69.73%

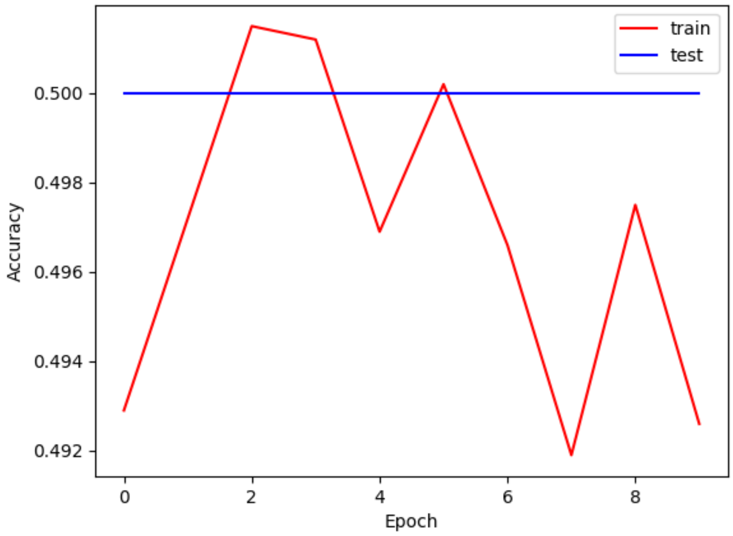
Wyniki predykcji psów i kotów:

```
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step
Prediction for cat.1.jpg: [[0.45725885 0.5427412 ]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Prediction for cat.2.jpg: [[0.45725885 0.5427412 ]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
Prediction for cat.3.jpg: [[0.45725885 0.5427412 ]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
Prediction for dog.1.jpg: [[0.45725885 0.5427412 ]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
Prediction for dog.2.jpg: [[0.45725885 0.5427412 ]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Prediction for dog.3.jpg: [[0.45725885 0.5427412 ]]
Predicted label: dog
------
```

Listing 5.13 Wynik eksperymentu nr 3 (predykcje). [opracowanie własne]

Otrzymane wyniki pokazują, że podczas predykcji wystąpił 3 błędy ,więc 3/6 zwierząt zostało poprawnie przydzielone.

Wykres accuracy na przestrzeni epok:



Listing 5.14 Wynik eksperymentu nr 3 (wykres). [opracowanie własne]

5.4 Eksperyment IV

W ramach czwartego eksperymentu, sprawdzimy model sieci neuronowej kiedy użyjemy w niej warstwy łączącej **MaxPooling**

```
model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu'))
model.add(Conv2D(32, kernel_size=(3, 3), strides=(2, 2), padding='valid', activation='relu'))
# model.add(AveragePooling2D(pool_size=(2, 2), strides=2, padding='valid'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'))
model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu'))
model.add(Conv2D(32, kernel_size=(3, 3), strides=(2, 2), padding='valid', activation='relu'))
# model.add(AveragePooling2D(pool_size=(2, 2), strides=2, padding='valid'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'))
model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu'))
model.add(Conv2D(32, kernel_size=(3, 3), strides=(2, 2), padding='valid', activation='relu'))
# model.add(AveragePooling2D(pool_size=(2, 2), strides=2, padding='valid'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(2, activation='softmax'))
```

Listing 5.15 Model sieci neuronowej z użyciem MaxPooling [opracowanie własne]

Po skończeniu nauki modelu otrzymujemy końcowe accuracy i loss:

```
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 163s 261ms/step - accuracy: 0.4950 - loss: 0.6932 - val_accuracy: 0.5000 - val_loss: 0.6932
------
```

Accuracy: 49.50%, Loss: 69.32%

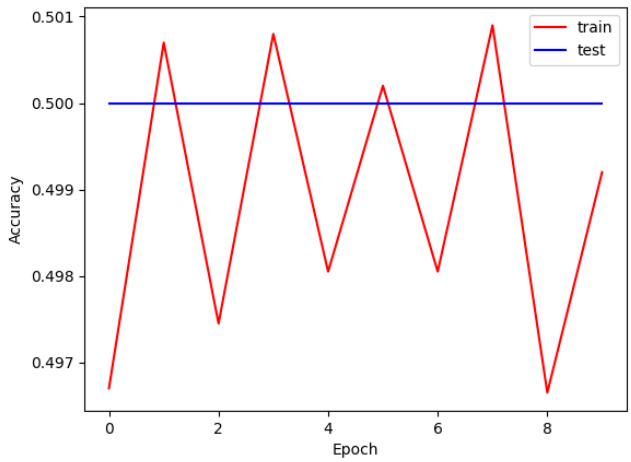
Wyniki predykcji psów i kotów:

```
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step
Prediction for cat.1.jpg: [[0.47228715 0.5277129 ]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
Prediction for cat.2.jpg: [[0.41629055 0.5837094 ]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
Prediction for cat.3.jpg: [[0.40679047 0.5932095 ]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Prediction for dog.1.jpg: [[0.41489992 0.5851001 ]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
Prediction for dog.2.jpg: [[0.48214808 0.51785195]]
Predicted label: dog
------
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
Prediction for dog.3.jpg: [[0.4408049 0.5591951]]
Predicted label: dog
------
```

Listing 5.13 Wynik eksperymentu nr 4(predykcje). [opracowanie własne]

Otrzymane wyniki pokazują, że podczas predykcji wystąpił 3 błędy ,więc 3/6 zwierząt zostało poprawnie przydzielone.

Wykres accuracy na przestrzeni epok:



Listing 5.14 Wynik eksperymentu nr 4 (wykres). [opracowanie własne]

5.5 Podsumowanie wyników eksperymentów i wnioski

Po przeprowadzeniu eksperymentów z trzema różnymi wartościami szybkości uczenia (learning rate): 0.001, 0.01 oraz 0.1, aby ocenić ich wpływ na dokładność modelu klasyfikującego obrazy psów i kotów, najlepsze wyniki osiągnięto przy szybkości uczenia 0.001, która uzyskała wysoką dokładność (accuracy). Model trenowany z tą wartością szybkości uczenia poprawnie sklasyfikował 5 z 6 obrazów zwierząt. Szybkość uczenia 0.001 pozwoliła modelowi na bardziej stabilne i dokładne dopasowanie do danych, co przełożyło się na lepszą skuteczność w klasyfikacji.

Dla tych eksperymentów zastosowano warstwę łączącą typu **AveragePooling**, która dawała najlepsze rezultaty w porównaniu do max pooling. Warstwa **AveragePooling** pozwoliła na uzyskanie bardziej stabilnych wyników w trakcie treningu, co wpłynęło na lepszą dokładność końcową.

W przeciwieństwie do tego, modele trenowane z wartościami learning rate 0.01 oraz 0.1 osiągnęły znacznie gorsze wyniki, z dokładnością bliską 50%. Oznacza to, że te modele poprawnie sklasyfikowały tylko 3 z 6 obrazów zwierząt, co wskazuje na problem z konwergencją lub nadmiernym dopasowaniem do danych przy wyższych szybkościach uczenia. Szybkość uczenia 0.01 i 0.1 mogła spowodować zbyt duże wahania w procesie uczenia, co skutkowało mniejszą precyzją w klasyfikacji obrazów psów i kotów.

Wyniki z przeprowadzonych eksperymentów wskazują, że odpowiednia szybkość uczenia (dla opisywanego algorytmu), taka jak 0.001, jest kluczowa dla skutecznego treningu modelu, ponieważ zapewnia stabilne i dokładne dopasowanie do danych, podczas gdy zbyt wysokie szybkości uczenia (0.01 i 0.1) prowadzą znacznych spadków w dokładności klasyfikacji. Warstwa łącząca typu **AveragePooling** okazała się skuteczniejsza niż **MaxPooling**.

6 Podsumowanie i wnioski

Raport z analizy sieci neuronowej CNN, rozpoznawającej zdjęcia psów i kotów, przedstawił niezbędną teorie o modelu wyżej wspomnianej sieci, dostarczając informacje jej warstwach i budowie.

Model sieci CNN napisany w języku Python przy pomocy bibliotek takich jak **Keras** czy **Tensorflow**, oraz teorii związanej z tematyką modelu, pozwolił na przedstawienie przykładowego rozwiązania dla takiego programu. Dzięki użyciu warstwy łączącej **AveragePooling** zamiast **MaxPooling** oraz doborze odpowiedniej wartości zmiennej odpowiedzialnej za szybkość nauki, byłem w stanie osiągnąć dokładność w wysokości **87.13%,** co pozwoliło w pierwszym eksperymencie na poprawne rozpoznanie 5 z 6 zwierząt. Dobór wartości 10 epok, w której model miał za zadanie nauczyć się na dobranej bazie, został wybrany z tego powodu, że środowisko w którym opisywany program był kompilowany miał zainstalowaną wersję **Tensorflow 2.16** która nie obsługuję GPU tylko CPU. W dużym stopniu wydłużyło to czas nauki modelu.

Podsumowując, raport pozwolił na zgłębienie teorii związanej z tematyką CNN oraz przedstawił przykładowy spoób napisania algorytmu dla tej sieci.

Algorytm CNN jest niezastąpiony w zadaniach związanych z rozpoznawaniem obrazów, dzięki swojej zdolności do automatycznego ekstraktowania istotnych cech i precyzyjnego klasyfikowania danych wizualnych. Jego efektywność i dokładność czynią go kluczowym narzędziem w dziedzinie przetwarzania obrazów i uczenia maszynowego.

7 Bibliografia

[1] https://www.analyticsvidhya.com/blog/2021/06/beginner-friendly-project-cat-and-dog-classification-using-cnn/ Data dostępu: 22.05.2023.

[2] https://www.v7labs.com/blog/convolutional-neural-networks-guide Data dostępu: 23.05.2023.

[3] https://medium.com/analytics-vidhya/image-classification-cats-and-dogs-pre-trained-neural-network-vs-constructed-6370d5c79fde Data dostępu: 23.05.2023.

[4] **https://medium.com/analytics-vidhya/convolutional-neural-networks-cnn-a78e78c1ba94** Data dostępu: 23.05.2023.

[5] https://en.wikipedia.org/wiki/Convolutional_neural_network Data dostępu: 23.05.2023.

[6] https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/ Data dostępu: 24.05.2023.

[7] https://www.baeldung.com/cs/ml-relu-dropout-layers Data dostępu: 23.05.2023.

[8] https://www.researchgate.net/figure/Functionalities-of-convolution-and-ReLU-layers-of-CNN_fig3_339342379 Data dostępu: 23.05.2023

[9] https://www.researchgate.net/figure/Illustration-of-Max-Pooling-and-Average-Pooling-Figure-2-above-shows-an-example-of-max_fig2_333593451 Data dostępu: 23.05.2023

[10] https://jacobheyman702.medium.com/different-pooling-layers-for-cnn-4652a5103d62

Data dostępu: 23.05.2023

[11] https://towardsdatascience.com/convolutional-neural-network-cnn-architecture-explained-in-plain-english-using-simple-diagrams-e5de17eacc8f Data dostępu: 23.05.2023

[12] https://botpenguin.com/glossary/softmax-function Data dostępu: 23.05.2023

[13] https://indiantechwarrior.com/fully-connected-layers-in-convolutional-neural-networks/ Data dostępu: 23.05.2023

[14] https://www.kaggle.com/datasets/salader/dogs-vs-cats Data dostępu: 23.05.2023

[15] https://www.kaggle.com/datasets/nagraj0308/dogs-vs-cats-small Data dostępu: 23.05.2023

[16] https://www.baeldung.com/cs/convolutional-layer-size Data dostępu: 23.05.2023