Politechnika Wrocławska

Podstawy sieci neuronowych	
Kierunek: Informatyczne Systemy Automatyki	Prowadzący: Aneta Górniak
Imię, nazwisko, numer albumu: Vasilisa Semyanikhina (283047)	Data oddania: 18/01/2026

Rozpoznawanie ras psów na podstawie obrazów

Wstęp

Celem projektu było zaprojektowanie, zaimplementowanie i trenowanie sieci neuronowej do rozwiązania średniozaawansowanego problemu klasyfikacyjnego, aproksymacyjnego lub modelowania. W celu zrealizowania projektu został wybrany następujący temat: rozwiązanie problemu klasyfikacyjnego - rozpoznawanie ras psów na podstawie otrzymanego zdjęcia. Klasyfikacja ras psów należy do klasy *fine-grained image classification* - jednego z najtrudniejszych typów problemów w wizji komputerowej.

Dany problem możemy rozpatrywać jako średniozaawansowany, ponieważ projekt wykorzystuje zbiór danych zawierający 120 ras psów (ponad 20 000 zdjęć). Wiele ras różni się jedynie drobnymi szczegółami (ta sama rasa może wyglądać zupełnie inaczej w zależności od pozy, wieku i pielęgnacji psa). Zdjęcia w zbiorze danych są prawdziwe, co oznacza, że mogą zawierać ludzi, cienie, rozmycia i inne psy.

Projekt został zrealizowany przy pomocy języka programowania Python z wykorzystaniem następujących bibliotek:

- *Kagglehub* – do łatwego pobierania danych z platformy Kaggle
- *Os* – dla pracy z plikami, odczyt zdjęć z folderów
- *Tensorflow* – dla budowy, trenowania i uruchamiania sieci neuronowych
- *Matplotlib* – dla tworzenia wykresów i wizualizacji danych
- *Numpy* – dla wykonywania obliczeń numerycznych

- *tensorflow.keras* – dla uproszczenia tworzenia i trenowania modeli, pozwala na dodawanie warstw do modeli
- *keras.applications.mobilenet_v2* – do dekodowania przewidywań z modelu MobileNetV2 aby uprościć definiowania ras
- *pandas* – do wczytywania, czyszczenia, przetwarzania i analizowania danych w formie tabelarycznej

W projekcie w pierwszej kolejności opracowano autorską sieć konwolucyjną (CNN) w celu lepszego zrozumienia struktury i działania warstw sieci neuronowych. Następnie model ten został rozszerzony poprzez zastosowanie wstępnie wytrenowanej sieci EfficientNet, co pozwoliło na przyspieszenie oraz ulepszenie procesu uczenia. W celu analizy porównawczej proces treningu przeprowadzono z wykorzystaniem wstępnie wytrenowanych modeli EfficientNet oraz ResNet50. Po zakończeniu etapu trenowania wykonano testowanie modeli, a uzyskane wyniki zostały zaprezentowane w formie wykresów oraz map cieplnych (heatmap).

Analiza problemu

Do realizacji projektu wykorzystano zbiór danych *Stanford Dogs Dataset*, który zawiera obrazy 120 ras psów. Każde zdjęcie zostało przeskalowane do rozmiaru 224 × 224 pikseli z trzema kanałami kolorów (RGB), co daje łącznie 150 528 wartości liczbowych na obraz. Na podstawie tych wartości sieć neuronowa uczy się rozpoznawać cechy wizualne, takie jak kształt pyska, długość sierści, kolor oraz budowę uszu. Aby możliwe było skuteczne wykrywanie i łączenie tych cech w spójną reprezentację obrazu, zastosowano konwolucyjną sieć neuronową (CNN).

Convolutional Neural Network (CNN) to specjalny typ sieci neuronowej przeznaczony do pracy z obrazami. Każde zdjęcie składa się z pikseli, czyli liczb opisujących jasność i kolor w danym punkcie obrazu. CNN składa się z wielu warstw, gdzie każda warstwa ma własny filtr i używając tego filtra, każda warstwa w różny sposób odczytuje piksele obrazka. Te filtry przesuwają się po obrazie i wykrywają krawędzie, linie oraz tekstury. Używając CNN, warstwy opisują budowę modelu do analizy obrazów.

Warstwy CNN dzielą się na warstwy konwolucyjne, warstwy zbierające (pooling) oraz warstwy w pełni połączone.

- Na pierwszych warstwach **konwolucyjnych** sieć wykrywa proste elementy obrazu, takie jak linie, krawędzie i kontrasty. W kolejnych warstwach konwolucyjnych te proste cechy są łączone w bardziej złożone wzorce, na przykład kształty i fragmenty obiektów.
- Warstwy **zbierające** (pooling) zmniejszają rozmiar danych i uodparniają sieć na przesunięcia obrazu, zachowując najważniejsze informacje.
- W głębszych warstwach **w pełni połączonych**(dense) sieć tworzy abstrakcyjną reprezentację całego obrazu, która zawiera informacje o tym, jakie cechy wizualne

zostały wykryte. Na tej podstawie sieć podejmuje ostateczną decyzję dotyczącą rozpoznania obrazu.

Opracowanie danych

Zbiór danych składa się z dwóch plików: *„images”* oraz *„annotations”.* Plik *„images”* zawiera 120 folderów, z których każdy odpowiada jednej rasie psa i zawiera liczne zdjęcia przedstawiające psy danej rasy. Plik *„annotations”* zawiera pola ograniczające, które określają położenie psa na każdym obrazie. W niniejszym projekcie wykorzystano wyłącznie plik *„images”,* ponieważ proces uczenia będzie koncentrował się na klasyfikacji ras, a nie na detekcji obiektów.

Za pomocą funkcji *os.listdir()* odczytywana jest lista ras znajdujących się w folderze *„images”.* Ponieważ metoda ta zwraca elementy w kolejności losowej, w celu uzyskania stabilnej i powtarzalnej kolejności ras lista jest sortowana alfabetycznie przy użyciu funkcji *sorted().* Metoda *tf.keras.utils.image_dataset_from_directory()* umożliwia wczytanie plików obrazów i przypisanie każdemu z nich etykiety numerycznej odpowiadającej rasie. Sieć neuronowa nie operuje bezpośrednio na nazwach ras, lecz na tych etykietach, gdzie każda etykieta reprezentuje jedną rasę psa.

Podział danych

Każda sieć neuronowa potrzebuje trzy rodzaje zbiorów danych: treningowego, walidacyjnego i testowego.

- **Zbiór treningowy** jest używany do uczenia modelu. Sieć uczy się rozpoznawać wzorce w obrazach, zależności pomiędzy cechami i dalej tworzyć przewidywania. Podczas treningu model wielokrotnie zmienia swoje wagi, aby zminimalizować błąd treningu, wykorzystując techniki optymalizacji. Bez zbioru treningowego sieć nie byłaby w stanie nauczyć się żadnych zależności.
- **Zbiór walidacyjny** sprawdza jak dobrze sieć uczy się na zbiorze treningowym. Jeśli sieć byłaby uczona tylko na zbiorze treningowym, mogłaby zacząć zapamiętywać konkretne obrazy zamiast uczyć się ogólnych cech - co prowadzi do zjawiska *overfitting* (przeuczenia). Zbiór walidacyjny zawiera obrazy, które nie są używane do aktualizacji wag, ale są wykorzystywane do sprawdzania jakości modelu w trakcie uczenia. Dzięki temu możemy wykryć, czy model rzeczywiście się uczy, czy tylko zapamiętuje dane.
- **Zbiór testowy** jest używamy już po zakończeniu treningu. Jest to zbiór danych, którego sieć wcześniej nie widziała i na którym sprawdzamy, jak dobrze potrafi rozpoznawać nowe obrazy.

Każdy z tych zbiorów jest niezbędny w procesie tworzenia sieci i pełni inną, kluczową rolę. Dzięki temu możemy stworzyć dobrze wytrenowany model i uzyskać wysoką dokładność, która nie wynika z zapamiętywania obrazów przez sieć, lecz z jej rzeczywistej zdolności do rozpoznawania wzorców i obiektów.

W danym projekcie dane zostały podzielone w następujących proporcjach: 70% na zbiór treningowy, 15% na zbiór walidacyjny i 15% na zbiór testowy. Do tworzenia zbiorów danych użyto funkcji *tf.keras.utils.image_dataset_from_directory()*. Zwraca ona obiekty typu *tf.data.Dataset*, które w każdej iteracji dostarczają pary *(image, label),* gdzie *image* to obraz psa, a *label* to liczbowa etykieta odpowiadająca danej rasie. Podczas tworzenia zbiorów danych określono następujące parametry:

- *directory* - ścieżka do folderu zawierającego obrazy ras psów
- *validation_split* - w jakich proporcach będziemy wykonywać podział na zbiór walidacyjny i treningowy, wpisujemy wartość pomiędzy [0,1] i ta wartość opisuje część danych przeznaczona na walidację
- *subset* - określa, który podzbiór ma zostać zwrócony (*"training"* lub "*validation"*)
- *seed* - stała wartość losowa zapewniająca powtarzalność podziału danych (w projekcie użyto 420)
- *image_size* - rozmiar, do którego mają zostać zmienione obrazy po odczytaniu ich w folderów. Ponieważ potok przetwarza partie obrazów, które muszą mieć ten sam rozmiar, należy podać ten parametr. W danym projekcie wpisujemy (224,224)
- *batch_size* - liczba obrazów w jednej partii danych (domyślnie 32)
- *class_names* - lista klas (ras), która pozwala kontrolować przypisanie etykiet liczbowych do ras

Przygotowanie danych

Sieć neuronowa przetwarza obrazy jako macierze liczb, dlatego na etapie przygotowania danych konieczne jest przekształcenie ich do formatu wygodnego dla sieci. Każdy obraz zawiera piksele o wartościach z przedziału [0,255] i może mieć różne oświetlenie, kontrast, skalę oraz orientację. Aby sieć mogła uczyć się rzeczywistych cech obiektów, a nie warunków wykonania zdjęcia, wszystkie obrazy muszą zostać ujednolicone pod względem formatu i rozmiaru.

Przetwarzanie danych składa się z normalizacji i augmentacji.

- **Normalizacja** to proces skalowania wartości pikseli do ustalonego zakresu liczbowego. W przypadku własnej sieci CNN zastosowano skalowanie do zakresu [0,1] za pomocą warstwy *Rescaling(1./255),* co stabilizuje proces uczenia i zapobiega zapamiętywaniu zdjęć na podstawie właściwości wizualnych.
- **Augmentacja** to sztuczne rozszerzenie zbioru treningowego poprzez losowe modyfikacje obrazów, takie jak obrót, odbicie, skalowanie, zmiana kontrastu i jasności. Celem augmentacji jest zwiększenie różnorodności danych oraz poprawa zdolności uogólniania modelu.

Wszystkie obrazy zostały przeskalowane do rozmiaru 224×224 piksele i zapisane jako tensory RGB, co jest wymagane przez architekturę sieci neuronowej.

Każdy obraz w partii danych (batchu) przechodzi przez wszystkie warstwy augmentacji, przy czym parametry transformacji są losowe i niezależne dla każdego obrazu. Dzięki temu sieć widzi za każdym razem nieco inne wersje tych samych zdjęć. W projekcie zastosowano następujące transformacje: odbicie horyzontalne, obrót ±0.2 radiana, skalowanie ±30%, zmianę kontrastu ±30% oraz zmianę jasności ±20%.

Pierwsza architektura - własna CNN

Tworzenie modelu

Rozpoczniemy od stworzenia własnej struktury CNN. W tej strukturze podstawowymi warstwami są:

- ***Convolutional Layers*:** te warstwy uczą się wykrywać różne cechy na obrazie poprzez zastosowanie filtrów. Macierz 3×3 (filtr) przesuwa się po obrazie, wykonując konwolucję: w każdej pozycji obliczana jest ważona suma pikseli, w wyniku czego powstaje nowa mapa cech (*feature map*). Z wszystkich tych liczb powstaje nowy „obraz cech", który jest przekazywany do następnej warstwy.

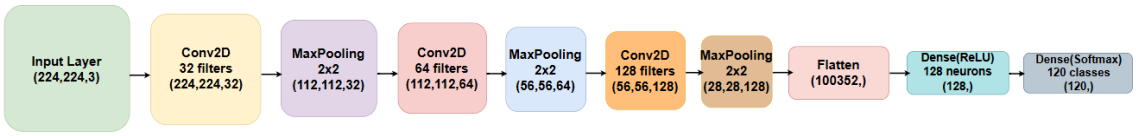
  Przykład: pierwsza warstwa ma 4 filtry, każdy filtr tworzy jedną mapę cech. Na wyjściu otrzymujemy 4 obrazy, które trafiają do drugiej warstwy, w której znajdują się 4 filtry. Każdy filtr analizuje wszystkie cztery obrazy i tworzy jeden obraz wyjściowy. W rezultacie ponownie otrzymujemy 4 obrazy, które trafiają do trzeciej warstwy. Trzecia warstwa ma 2 filtry, co oznacza, że na wyjściu będą 2 obrazy, które trafiają do warstwy pullującej.
  Liczba obrazów wyjściowych = liczba filtrów w warstwie.
- ***Pooling Layers*:** otrzymany obraz jest zmniejszany za pomocą *Max pooling*. Z każdego małego okna (na przykład 2×2) wybierana jest maksymalna wartość. Zmniejsza to rozmiar danych, zachowuje najważniejsze cechy i przyspiesza uczenie się.
- ***Dropout Layers*** podczas uczenia się część neuronów jest losowo zerowana, aby sieć nie zaczęła zapamiętywać konkretnych obrazów i nie uczyła się ponownie.
- ***Fully Connected Layers(Dense):*** zbierają informacje z poprzednich warstw, tworząc ogólny obraz i podejmując ostateczną decyzję. Zwracają wektor składający się ze 100 parametrów, gdzie każda liczba oznacza prawdopodobieństwo wystąpienia konkretnego gatunku. Suma 100 liczb = 1.

Tworzymy model przy użyciu *models.Sequential()*, co pozwala nam utworzyć sekwencję warstw.

Zaczynamy od dodania warstw augmentacji i normalizacji. Struktura prostej modelu CNN składa się z kilku par warstw *Convolutional* + *MaxPooling2D*, a następnie warstwy *Flatten*, która przekształca macierz cech w jednowymiarowy wektor. Wektor ten jest przekazywany do warstwy *Dense* z aktywacją *relu*, gdzie wartości ujemne są zerowane, a dodatnie zachowywane. Wprowadza to nieliniowość i pozwala modelowi uczyć się złożonych zależności. Następnie następuje warstwa *Dropout*, która losowo zeruje część

parametrów wektorze, aby sieć nie zapamiętywała konkretnych kombinacji. Na końcu ostatnia warstwa *Dense* z aktywacją *softmax* zwróci tablicę prawdopodobieństw dla każdej rasy.

Logikę tych warstw można schematycznie opisać w następujący sposób:



Rysunek 1: Diagram architektury sieci neuronowej dla własno stworzonej CNN

Warstwa wejściowa (*input layer*) otrzymuje obraz RGB z trzema kanałami, następnie na obraz nakładane są 32 filtry splotowe, w wyniku czego powstaje 32 mapy cech (*feature maps*). W kolejnych warstwach na uzyskane mapy cech nakładane są nowe filtry, które generują coraz bardziej złożone reprezentacje obrazu.

Warstwa *Flatten* przekształca trójwymiarowe mapy cech w jednowymiarowy wektor o długości 100 352 cech. Następnie warstwa *Dense* z funkcją aktywacji *ReLU* łączy i przekształca te cechy w bardziej abstrakcyjne i precyzyjne wzorce (np. „plamista sierść + długie uszy + średni wzrost" → może wskazywać na dalmatyńczyka).

W ostatniej warstwie *Dense* każdy z 120 neuronów otrzymuje na wejściu wszystkie 128 cech i oblicza własną kombinację, odpowiadającą prawdopodobieństwu przynależności obrazu do danej rasy.

Jak przechodzimy od 128 do 120? *Dense* pobiera cały wektor wejściowy i oblicza sumy ważone dla każdego nowego neuronu

$$y_1 = w_{11}x_1 + w_{21}x_2 + \ldots + w_{128}*x_{128} + b_1$$
$$y_{120} = w_{1201}x_1 + \ldots + w_{12820}x_{128} + b_{120}$$

Każdy z 120 neuronów otrzymuje wszystkie 128 liczb na wejściu i oblicza swoją kombinację. Następnie stosujemy '*softmax*', aby uzyskać prawdopodobieństwa, których suma wynosi 1.

Kompilacja modelu

W trakcie treningu ważne jest opisanie, w jaki sposób sieć się uczy. Proces uczenia się polega nie tylko na rozpoznawaniu wzorców i łączeniu ich w kombinacje, ale także na korekcie błędów. Podczas uczenia się sieć popełnia błędy i musimy określić, w jaki sposób błędy te będą mierzone i korektowane, czyli jak będą zmieniać się wagi neuronów, aby model przewidywał prawidłowo.

Realizujemy to na etapie kompilacji. Określamy:

- **Funkcja strat** (*loss function*) – określa błąd między prognozami modelu a rzeczywistymi wartościami docelowymi, podając jedną liczbę („straty”), na podstawie której optymalizator decyduje, jak skorygować wagi.
- **Optymalizator** (*optimizer*) – algorytm, który aktualizuje wagi sieci na podstawie wartości funkcji strat, aby zminimalizować błędy. Zadaniem optymalizatora jest znalezienie takich wag, które sprawiają, że funkcja strat jest jak najmniejsza.
- **Metryki** (*metrics*) – wskaźniki, które śledzimy podczas uczenia się, aby zrozumieć, jak dobrze model wykonuje zadanie (na przykład dokładność ‘*accuracy’*)

Do kompilacji użyliśmy optymalizatora ‘*adam’*, który automatycznie dobiera zmiany wag modelu, funkcję strat ‘*sparse_categorical_crossentropy’*, która dobrze nadaje się w klasyfikacji wieloklasowej z etykietami numerycznymi, oraz metrykę ‘*accuracy’*, która pokazuje odsetek poprawnie sklasyfikowanych obrazów na każdym etapie uczenia.

```
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Rysunek 2: Kompilacja modelu

Trenowanie modelu

Model jest trenowany za pomocą metody *model.fit()*. Przekazujemy zbiór danych treningowych, zbiór danych walidacyjnych i liczbę epok.

Jedna **epoka** to jedno pełne przejście przez cały zbiór danych treningowych. Dane nie są przetwarzane jednocześnie, ale partiami (batchami). Po tym, jak model przetworzy kolejno wszystkie partie, a tym samym cały zbiór danych, epoka zostaje uznana za zakończoną. Na przykład, jeśli mamy 1000 obrazów i *batch_size=32*, model przetworzy wszystkie 1000 obrazów w ciągu jednej epoki, dzieląc je na partie po 32 obrazy.

Podczas każdej epoki dzieje się następujące:

1. Cały zbiór danych jest dzielony na partie (*batching*), małe grupy obrazów, w naszej sytuacji grupy po 32 obrazy.
2. Przepływ bezpośredni (*forward pass*): każdy obraz z partii przechodzi przez wszystkie warstwy sieci, a na wyjściu otrzymujemy wektor prawdopodobieństw.
3. Obliczanie strat: dla każdej partii obliczana jest różnica między przewidywaniami modelu a rzeczywistymi oznaczeniami.
4. Przepływ wsteczny (*backpropagation*): na podstawie funkcji strat optymalizator koryguje wagi wszystkich warstw, aby zmniejszyć błąd.
5. Aktualizacja wag: po każdej partii wagi warstw są aktualizowane, a model stopniowo „uczy się” rozpoznawać cechy i wzorce.

Po przetworzeniu wszystkich partii w epoce model dokonuje oceny na zestawie walidacyjnym: nie zmienia wag, a jedynie sprawdza, jak dobrze wyszkolony model radzi sobie z nowymi, wcześniej nieznanymi danymi. Ten krok pomaga monitorować nadmierne dopasowanie (*overfitting*): jeśli dokładność na zestawie treningowym rośnie, a na zestawie walidacyjnym spada, oznacza to, że model zapamiętuje obrazy zamiast uczyć się rozpoznawać cechy.

Po zakończeniu uczenia metoda *model.fit()* zwraca obiekt *history*, który zawiera następujące wartości:

- *loss* - błąd na zbiorze treningowym,
- *val_loss* - błąd na zbiorze walidacyjnym,
- *accuracy* - dokładność modelu na zbiorze treningowym,
- *val_accuracy* - dokładność modelu na zbiorze walidacyjnym.

```
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10 #1 epoch = one full pass over the entire training dataset
)
```

Rysunek 3: Uczenie się własno stworzonej sieci na 10 epokach

Za pomocą obiektu *history* możemy wizualizować proces uczenia się za pomocą wykresów, analizować parametry uczenia się i w ten sposób wybierać najlepszy model.

Wynik trenowania

Przykładowo, uczenie odbywało się na pierwszych 10 rasach, a w modelu wykonano 10 epok. W ciągu tych 10 epok dokładność na zbiorze treningowym szybko rosła od 11% w pierwszej epoce do 99% w dziesiątej. Pokazuje to, że model z sukcesem „zapamiętuje" dane treningowe. Jednak dokładność na zestawie walidacyjnym pozostawała niska i praktycznie nie rosła, około 28-29% pod koniec trenowania. Jednocześnie funkcja strat na walidacji znacznie wzrosła, co wskazuje na przetrenowanie: model nauczył się rozpoznawać konkretne obrazy z zestawu treningowego, ale nie nauczył się uogólniać na nowe obrazy.

```
Epoch 1/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 7s 93ms/step - accuracy: 0.1143 - loss: 2.9111 - val_accuracy: 0.1979 - val_loss: 2.2890
Epoch 2/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 7s 64ms/step - accuracy: 0.1610 - loss: 2.2691 - val_accuracy: 0.1875 - val_loss: 2.2079
Epoch 3/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.2805 - loss: 2.0512 - val_accuracy: 0.2917 - val_loss: 2.0610
Epoch 4/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 3s 63ms/step - accuracy: 0.4639 - loss: 1.5699 - val_accuracy: 0.2847 - val_loss: 2.1213
Epoch 5/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 3s 70ms/step - accuracy: 0.7182 - loss: 0.9473 - val_accuracy: 0.2639 - val_loss: 2.6475
Epoch 6/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 3s 70ms/step - accuracy: 0.8571 - loss: 0.4789 - val_accuracy: 0.2951 - val_loss: 3.9949
Epoch 7/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 3s 79ms/step - accuracy: 0.9433 - loss: 0.1710 - val_accuracy: 0.2465 - val_loss: 5.1747
Epoch 8/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 3s 70ms/step - accuracy: 0.9596 - loss: 0.1150 - val_accuracy: 0.2708 - val_loss: 4.6461
Epoch 9/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 3s 64ms/step - accuracy: 0.9809 - loss: 0.0884 - val_accuracy: 0.2743 - val_loss: 5.8164
Epoch 10/10
42/42 ━━━━━━━━━━━━━━━━━━━━ 6s 78ms/step - accuracy: 0.9922 - loss: 0.0520 - val_accuracy: 0.2882 - val_loss: 5.2538
```

Rysunek 4: Wynik uczenia się sieci

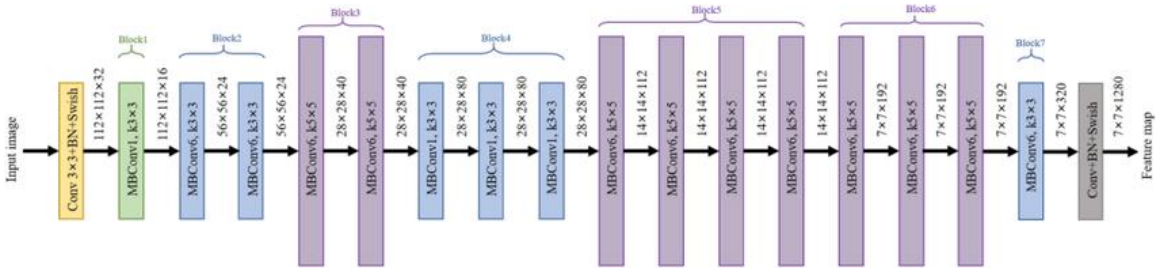
Wynika to z faktu, że architektura własnej CNN zawiera tylko kilka warstw, co jest niewystarczające do dokładnego rozpoznawania dużej liczby różnych ras psów. Ponieważ CNN uczy się od zera, sieć potrzebowałaby bardzo dużo czasu, aby najpierw nauczyć się odróżniać proste wzorce, a dopiero potem rozpoznawać 120 ras psów. Stworzona struktura działa poprawnie, sieć zaczęła się uczyć, ale dla tak dużego i zróżnicowanego zestawu danych jest zbyt słaba. Rozwiązaniem może być wykorzystanie już wstępnie nauczonej sieci (*pretrained model*), co pozwoli przyspieszyć proces uczenia się, skuteczniej rozróżniać bardziej skomplikowane wzorce między rasami psów oraz zmniejszyć ryzyko przeuczenia.

Transfer Learning z EfficientNet

Transfer learning - metoda wykorzystania już wstępnie wytrenowanego modelu jako część naszego modelu. Wykorzystujemy wiedze nauczonego modelu i dostosowujemy ją do naszego problemu. Taki model już umie rozpoznawać proste wzorce takie jak krawędzie, tekstury, futro, oczy, twarze, nawet umię rozpoznawać zwierząt. Powinnyśmy tylko zmienić niektóre warstwy aby dostosować model do naszych potrzeb.

Będziemy wykorzystać EfficientNet-B0 - podstawowa splotowa sieć neuronowa CNN z rodziny EfficientNet znana dzięki wysokiej dokładności i wydajności. Wykorzystuje metodę skalowania złożonego do równomiernego skalowania głębokości, szerokości i rozdzielczości sieci, zapewniając optymalną wydajność w zadaniach takich jak klasyfikacja ImageNet.

EfficientNet składa się z sekwencji warstw *convolution+pooling*. Każda kolejna warstwa zmniejsza rozdzielczość przestrzenną obrazu wejściowego, zachowując jednocześnie najbardziej informacyjne cechy. Podczas przetwarzania obrazu jego rozmiar jest sukcesywnie zmniejszany: 224×224 → 112×112 → 56×56 → 28×28 → 14×14 → 7×7.



Rysunek 5: Diagram architektury wytrenowanej sieci neuronowej EfficientNet-B0

Link do obrazu: https://www.researchgate.net/figure/The-detailed-architecture-of-EfficientNet-B0-EfficientNet-B0-consists-of-seven-blocks_fig2_361521546

Każda z 49 komórek przechowuje 1280 różnych filtrów, z których każdy reprezentuje prawdopodobieństwo wystąpienia określonego wzorca.

Do zmiennej *base* ładujemy już nauczoną sieć neuronową EfficientNet. Ta zmienna przechowuje całą „pamięć wzrokową" sieci EfficientNet - wszystkie jej filtry i wagi, których nauczyła się podczas uczenia się na ogromnym zbiorze danych ImageNet. Usuwamy ostatnią warstwę klasyfikacyjną, która potrafi rozpoznawać obiekty, aby zamiast tego sieć rozpoznawała tylko rasy psów. Zamiast tego dodamy naszą warstwę klasyfikacyjną. Wskazujemy, że rozmiar obrazu wejściowego będzie wynosił (224,224,3). W parametrze *weights* wskazujemy '*imagenet'*. Imagenet to duża baza danych składająca się z obrazów, na których trenowano EfficientNet, co oznacza, że sieć ta potrafi rozpoznać ponad 1000 klas. Ponieważ EfficientNet potrafi rozpoznać te obrazy, oznacza to, że podczas procesu uczenia się zostały ustawione prawidłowe wagi. W ten sposób ładujemy wstępnie wytrenowane wagi sieci, uzyskane podczas uczenia się na zbiorze danych ImageNet.

Następnie *base.trainable = False* zamrażamy wszystkie warstwy EfficientNet, co oznacza, że nie będziemy zmieniać wag modelu, a jedynie wykorzystamy je do nauki. Ciąg *base.training=False* przełącza model EfficientNet w tryb inferencji. Zapobiega to zniszczeniu już wyuczonych rozkładów cech ImageNet.

Zamiast warstw normalizacji i argumentacji wskazujemy zmiany niezbędne dla EfficientNet. EfficientNet został nauczony na pikselach w zakresie [-1,1]. Tworzymy warstwę, która będzie częścią naszego nowego modelu i będzie stosować te zmiany do każdego obrazu.

Tworzenie modelu

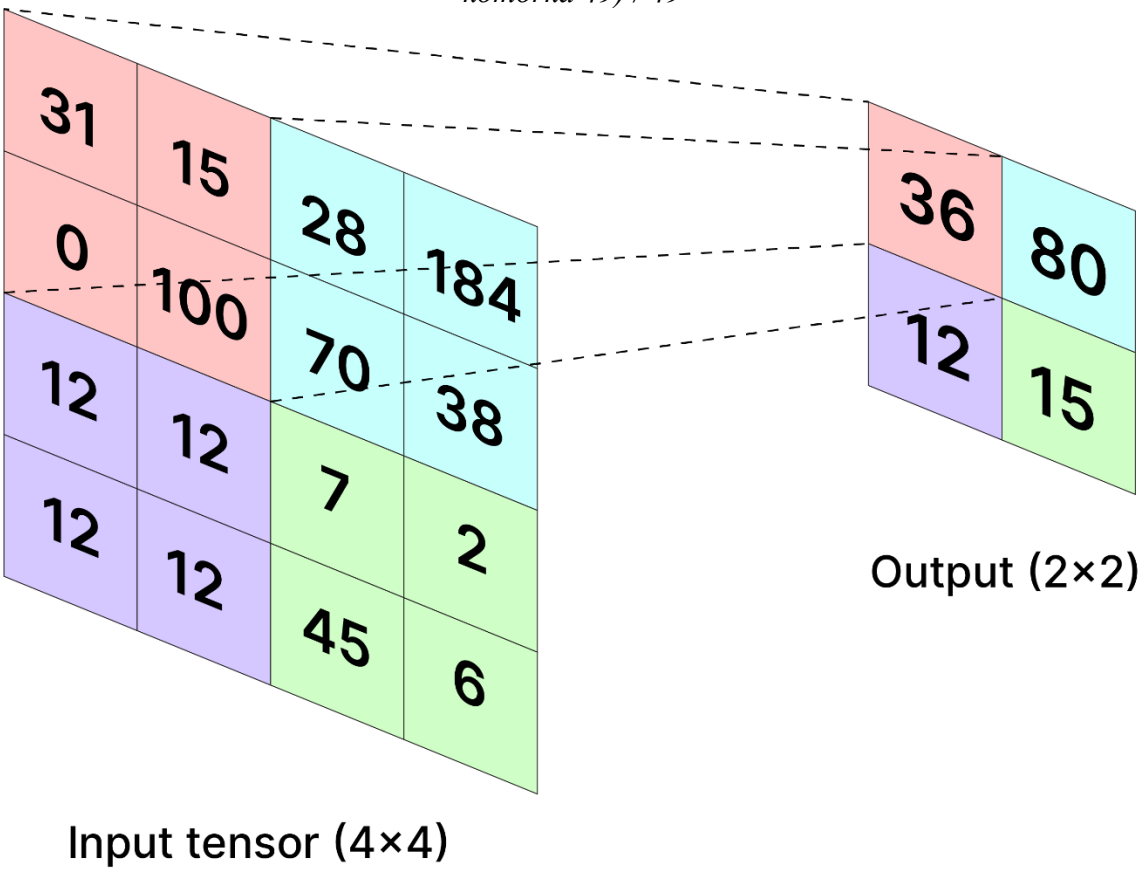
Architektura nowego modelu będzie następująca: zaczynamy od określenia rozmiaru obrazów wejściowych i warstwy normalizacji, która będzie przekształcać obrazy do zakresu [-1,1]. Następnie pojawia się warstwa wstępnie wytrenowanego modelu, warstwa *base*.

Warstwa *base* zwraca 7×7×1280, czyli macierz 7×7, gdzie każdy element jest tablicą zawierającą 1280 wartości. Wartości te odpowiadają prawdopodobieństwu wystąpienia

konkretnego wzorca. Na przykład: *feature_map[3][5] = [0,1, 0,93, 0,02, 0,8, ... 1280 liczb] = [czy jest ucho, czy jest futro, czy jest oko, czy jest nos, ...].*

Kolejna warstwa *GlobalAveragePooling2D()*. W *EfficientNet* są setki warstw konwolucji. Ostatni blok konwolucji ma 1280 filtrów, gdzie jeden filtr to jeden wzór (np. ucho, oko, plamy, sierść...). Oznacza to, że każda liczba jest prawdopodobieństwem wystąpienia konkretnej cechy. *Dense* nie może pracować z macierzą, działa tylko z wektorami. *GlobalAveragePooling2D()* pobiera każdy z 1280 filtrów i uśrednia go na całym obrazie. Mówiąc inaczej

Średnia pierwszego filtra = (wartość tego filtra w komórce 1 + komórka 2 + ... + komórka 49) / 49



Rysunek 6: Przykład działania warstwy zbierającej
Link do obrazu: https://iq.opengenus.org/global-average-pooling/

I tak dla każdego z 1280 filtrów, po czym otrzymujemy wektor składający się z 1280 wartości, gdzie każda wartość oznacza, jak silnie każdy z 1280 wzorców wizualnych jest obecny na obrazie.

Kolejna warstwa *Dropout(0.4)* losowo zeruje 40% z tych 1280 cech. Następnie *Dense(256, activation='relu')* z 1280 cech łączy parametry i zwraca wektor składający się z 256 liczb. Te 256 liczb to cechy rasy wysokiego poziomu (kształt pyska, typ uszu, typ sierści, proporcje). Następnie ponownie stosujemy *Dropout(0.3)*, zerując kolejne 30% z 256

wartości. Ten wektor trafia do ostatniej warstwy *Dense* z aktywacją *softmax*, która zwraca wektor 120 prawdopodobieństw.

Trenowanie modelu

Proces uczenia się był taki sam jak w poprzednim modelu, uczenie odbywało się na wszystkich 120 rasach psów i w 20 epokach. Wynik ostatniej epoki był następujący:

Epoch 20/20: step - accuracy: 0.9117 - loss: 0.2531 - val_accuracy: 0.8451 - val_loss: 0.5916

Po sprawdzeniu na zestawie danych testowych otrzymaliśmy następujący wynik:

```
test_loss, test_acc = model.evaluate(test_ds)
print("Test accuracy:", test_acc)
```

```
97/97 ━━━━━━━━━━━━━━━━━━━━ 25s 213ms/step - accuracy: 0.8228 - loss: 0.6945
Test accuracy: 0.8339780569076538
```

Rysunek 7: Wynik rozpoznawania obrazów przez sieć neuronową na zbiorze testowym

Mamy więc teraz dokładność treningu ≈ 0,84 i dokładność testu ≈ 0,83. Model działa dobrze zarówno na zbiorze danych, jak i na nowych danych, co oznacza że sieć uczy się, a nie tylko zapamiętuje.

Jeśli dodamy więcej epok, istnieje ryzyko obniżenia dokładności testu, ponieważ gęste warstwy zaczną adaptować się do szumu (ryzyko przeuczenia, czyli „zapamiętywania").

Aby ulepszyć nasz model, możemy odblokować ostatnie warstwy sieci Efficientnet i powoli je uczyć. Teraz ostatnie warstwy sieci Efficientnet wyglądają jak „kot, wilk, lis, pies - prawie to samo". Po dostrojeniu zidentyfikuje psy, takie jak „husky i malamuty mają drobne różnice w kształcie oczu i sierści".

Fine-tuning

Fine-tuning jest procesem dalszego dostrajania już wytrenowanej sieci neuronowej. W naszym projekcie proces uczenia został podzielony na dwa etapy:

1. Transfer learning: Zamrożona sieć *EfficientNet* + *nowe warstwy Dense*.
   Na tym etapie sieć wykorzystuje ogólną wiedzę EfficientNet zdobytą na ImageNet i uczy się mapować wykryte cechy na rasy psów, np. „kształt oczu + rodzaj sierści + kształt pyska → konkretna rasa".

2. Fine-tuning: Częściowo odmrożona sieć *EfficientNet* + *warstwy Dense*.
   Na tym etapie pozwalamy ostatnim warstwom EfficientNet dostosować się do zadania rozpoznawania ras psów. Oznacza to, że zamiast rozpoznawać ogólne

obiekty (zwierzęta, twarze, futro), sieć zaczyna uczyć się drobnych różnic między rasami.

EfficientNet został wytrenowany na zbiorze ImageNet, który zawiera wiele klas zwierząt, w tym psy, wilki i lisy. Jednak ImageNet nie jest specjalizowany w rozróżnianiu ras psów. Dlatego zastosowano fine-tuning, aby dostroić ostatnie warstwy EfficientNet do bardziej precyzyjnego rozpoznawania cech typowych dla różnych ras psów (np. kształt uszu, struktura sierści, proporcje pyska).

EfficientNet ma około 230 warstw. Warstwy od 0 do 160 przedstawiają kąty, tekstury, futro i kształty, a warstwy od 161 do 229 przedstawiają zwierzęta i twarze. Najpierw odmrażamy wszystkie warstwy EfficientNet. Następnie obliczamy, ile warstw EfficientNet stanowi 70% wszystkich warstw i zamrażamy je, pozostawiając tylko ostatnie 30%, czyli w przybliżeniu warstwy 161–229. Przeuczymy te warstwy z rozpoznawania dowolnych obiektów na rozpoznawanie tylko psów.

Do kompilacji użyjemy tej samej funkcji straty, tej samej metryki i tego samego optymalizatora, ale zastosujemy wolniejszy proces uczenia, ponieważ sieć już posiada dobre wagi wyuczone na ImageNet. Zbyt szybkie uczenie mogłoby zniszczyć te wartości i doprowadzić do pogorszenia generalizacji (*overfittingu*). Celem fine-tuning nie jest radykalna zmiana wag, lecz ich delikatne dostosowanie do nowego zadania.

```
base.trainable = True

#how many layers is in 70% of the EfficientNet
fine_tune_at = int(len(base.layers) * 0.7)

#don't train layers from [0-fine_tune_at]
for layer in base.layers[:fine_tune_at]:
    layer.trainable = False

model.compile(
    # we need to choose slow learning rate in order to change efficientnet vision a bit, not to break it
    optimizer=tf.keras.optimizers.Adam(1e-5),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

#teach 10 more epoches in order to reach higher accuracy
#first 20 epoches = classification, next 10 epoches = fine tuning
history_fine = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=30,
    initial_epoch=history.epoch[-1]
)
```

Rysunek 8: Implementacja warstw dla realizacji fine-tuning

Trenowanie z fine-tuning

W procesie fine-tuningu wykorzystano ten sam zbiór treningowy i walidacyjny, jednak całkowita liczba epok została zwiększona do 30. Ponieważ zasadnicza część sieci została

już wytrenowana w pierwszych 20 epokach etapu transfer learning, na etapie fine-tuningu uczenie odbywa się wyłącznie w przedziale epok od 20 do 30, podczas którego dostrajane są wybrane warstwy sieci EfficientNet.

Po treningu wynik na zbiorze danych testowych uległ pogorszeniu, a mianowicie:

```
test_loss, test_acc = model.evaluate(test_ds)
print("Test accuracy after fine-tuning:", test_acc)
```

```
97/97 ━━━━━━━━━━━━━━━━━━━━ 14s 103ms/step - accuracy: 0.8138 - loss: 0.6924
Test accuracy after fine-tuning: 0.8152804374694824
```

Rysunek 9: Wynik rozpoznawania obrazów przez sieć neuronową na zbiorze testowym po dodaniu fine-tuning

Przed fine-tuningiem uzyskano *test accuracy = 0,834*, natomiast po jego zastosowaniu wartość test *accuracy* spadła do 0,815. Przed fine-tuningiem EfficientNet dostarczał wysokiej jakości, ogólnych cech wizualnych, a dodane warstwy *Dense* nauczyły się mapować te cechy na konkretne rasy psów.

Po zastosowaniu fine-tuningu zmieniono wagi ostatnich warstw EfficientNet, w wyniku czego sieć zaczęła dostosowywać już dobrze wyuczone filtry do specyficznych obrazów z naszego zbioru danych, co doprowadziło do przeuczenia modelu.

W takiej sytuacji EfficientNet już skutecznie rozpoznaje psy, dlatego modyfikacja jego wewnętrznych filtrów zmniejszyła zdolność generalizacji modelu. W związku z tym najlepiej działającym rozwiązaniem okazał się model bez fine-tuningu.

Analiza wyników

Wynik trenowania modelu przy użyciu wstępnie wytrenowanej sieci EfficientNet na testowym zbiorze danych wykazał dokładność 0,834. Dla porównania przeprowadzono analogiczne trenowanie przy użyciu wstępnie wytrenowanego modelu ResNet50.

ResNet50 to popularna architektura głębokiej konwolucyjnej sieci neuronowej (CNN) o głębokości 50 warstw. Jest ona często wykorzystywana jako model bazowy lub w głębszych sieciach do zadań klasyfikacji obrazów. W tym eksperymencie wykorzystano również 120 klas (ras psów) i 20 epok trenowania. Wynik wyniósł 0,750, co jest wartością znacznie niższą niż w przypadku modelu opartego na EfficientNet.

Analiza uzyskanych wyników pokazuje, że EfficientNet wykazuje lepszą wydajność w zadaniu klasyfikacji ras psów. Wynika to z bardziej efektywnej struktury sieci, dostosowanej do wydobywania cech z obrazów przy mniejszej liczbie parametrów w porównaniu z ResNet50. Ponadto EfficientNet lepiej radzi sobie z różnorodnością cech wizualnych ras psów, w tym kształtem pyska, uszu i teksturą sierści.

W ten sposób wybór wstępnie wyszkolonego modelu EfficientNet okazał się bardziej odpowiedni dla tego zadania, zapewniając wyższą dokładność i stabilność działania sieci na danych testowych.

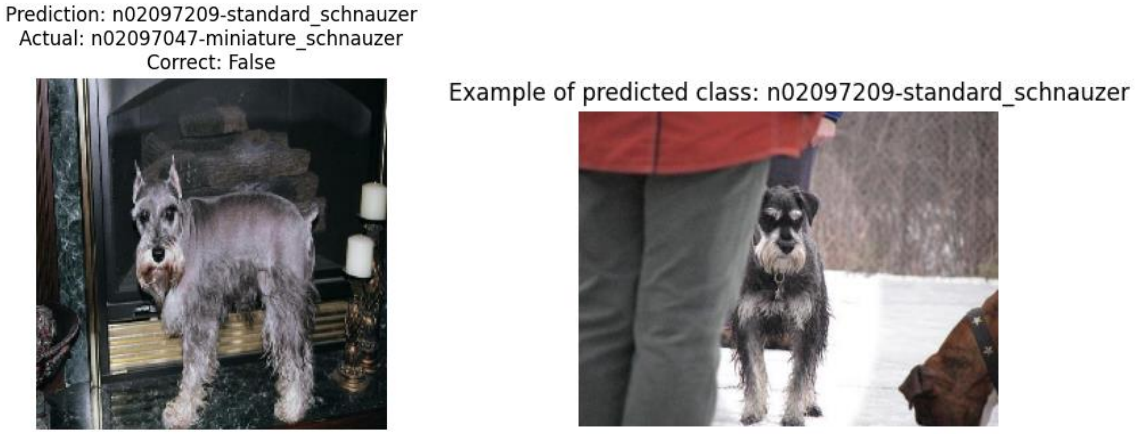
Wizualizacje

Aby wizualnie przedstawić wynik uczenia się sieci na podstawie testowego zestawu danych, wybierane są losowe obrazy, które są przekazywane do modelu w celu klasyfikacji. Dla każdego obrazu wyświetlany jest wynik pracy modelu wraz z przewidywaną i rzeczywistą rasą, a także informacją o poprawności przewidywania.

W przypadku nieprawidłowego przewidywania dodatkowo wyświetlany jest przykład obrazu z klasy, która została przewidziana przez model. Pozwala to przeprowadzić jakościową analizę błędów i ocenić wizualne podobieństwo między rasami, które mogło wpłynąć na decyzję modelu.

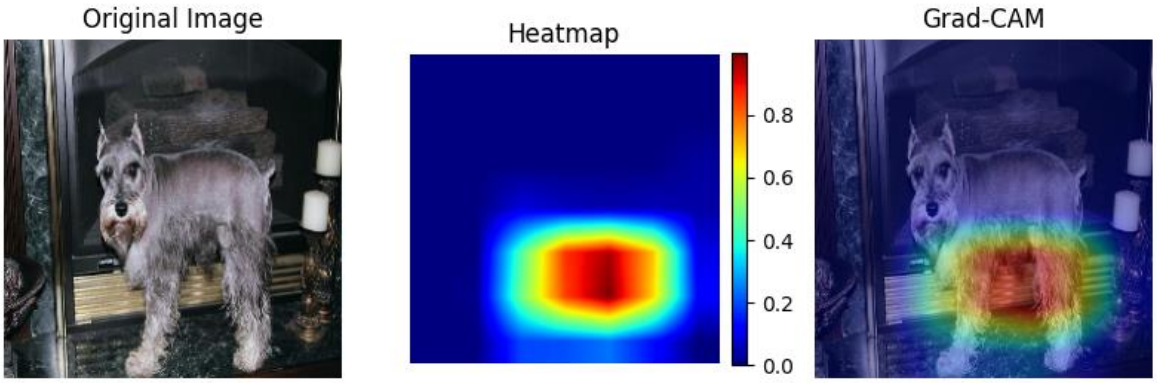
Do interpretacji przewidywań wykorzystuje się metodę Grad-CAM, opartą na analizie gradientów warstwy wyjściowej modelu w odniesieniu do aktywacji ostatniej warstwy konwolucyjnej EfficientNetB0. W rezultacie powstaje mapa cieplna, odzwierciedlająca obszary obrazu, które są najbardziej istotne dla podjęcia decyzji o przynależności do konkretnej klasy.

Otrzymana mapa cieplna jest wizualizowana oddzielnie i nakładana na obraz wyjściowy, co pozwala ocenić, na które części obrazu model zwraca największą uwagę. Takie podejście zwiększa interpretowalność modelu i pozwala dodatkowo ocenić poprawność jego działania podczas klasyfikacji obrazów.

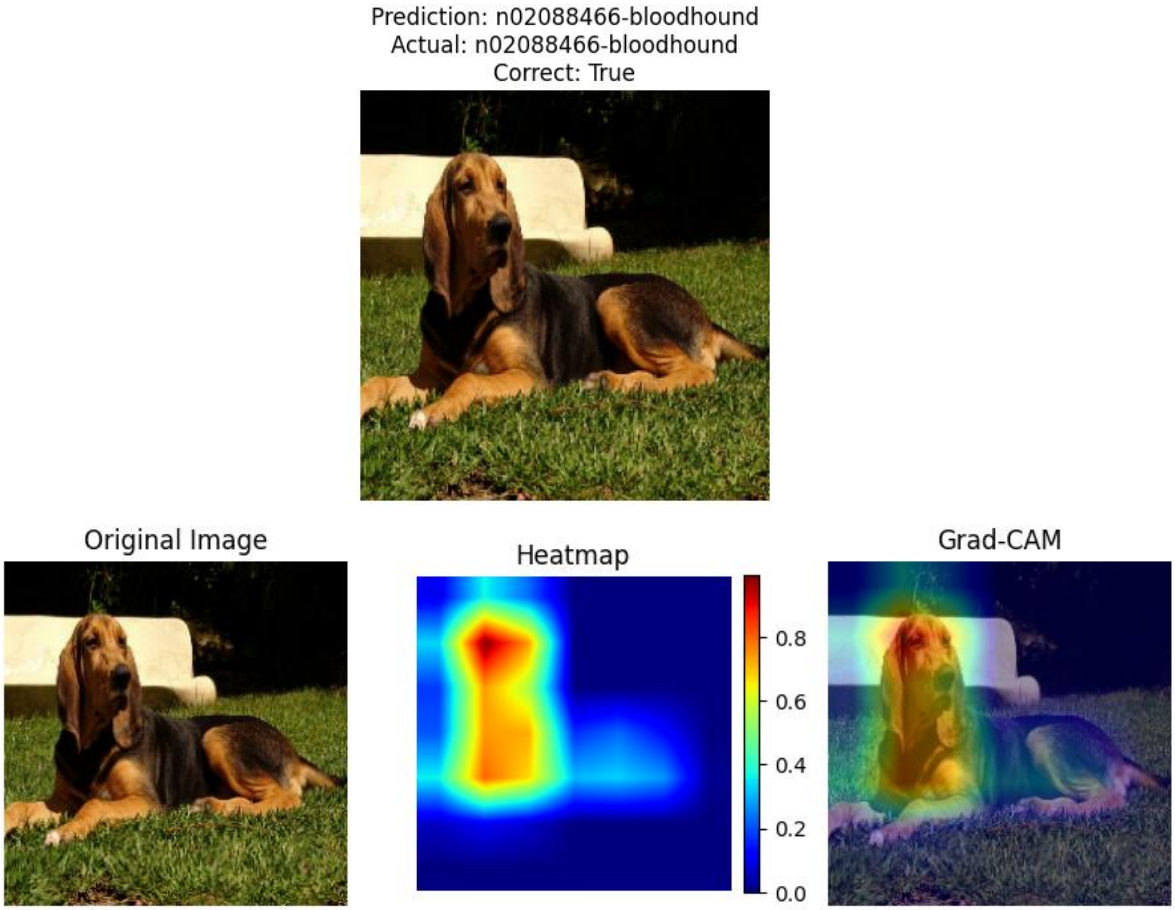


Przykład 1: Rozpatrzymy jeden z błędów w działaniu naszego modelu. Zaobserwowano, że model popełnił pomyłkę przy rozróżnianiu Standard Schnauzera i Miniature Schnauzera. Wynika to z faktu, że rasy te należą do tej samej rodziny (o czym wskazuje również ich nazwa), z tym że Miniature Schnauzer jest rasą mniejszą. Na niektórych

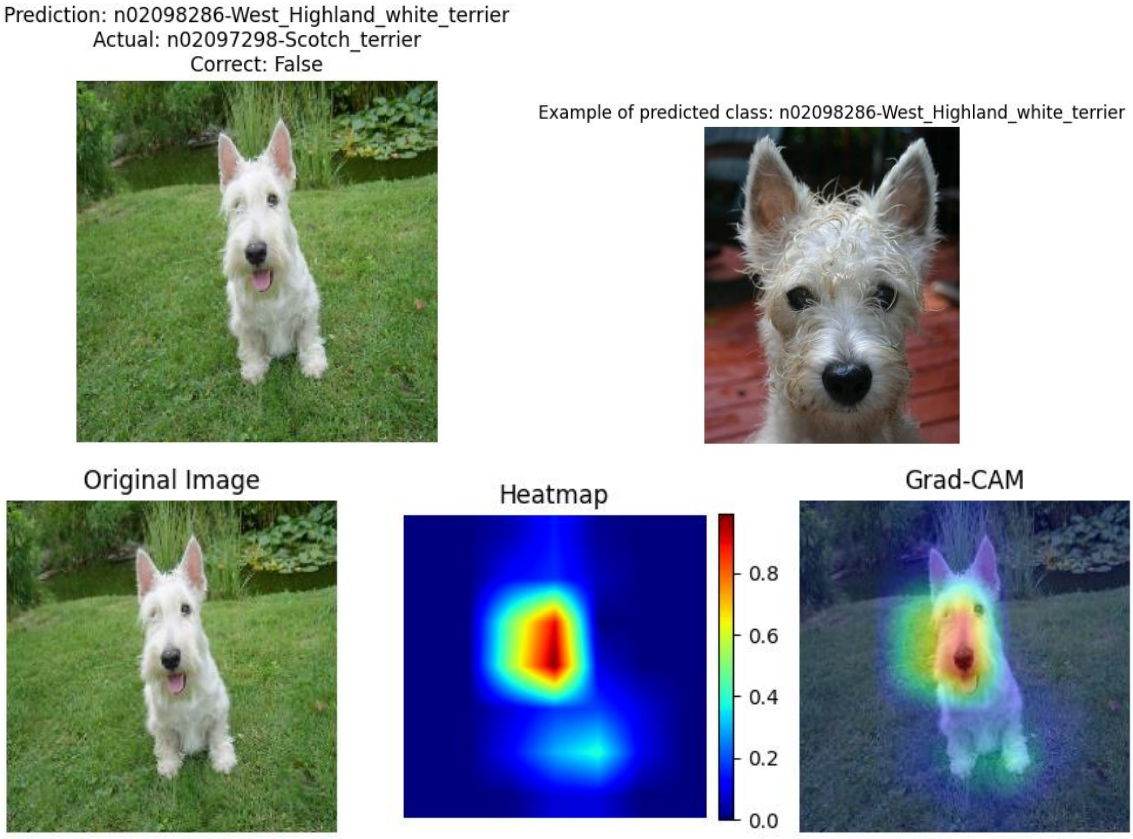
zdjęciach, szczególnie w omawianym przypadku, trudno jednoznacznie określić rzeczywisty rozmiar psa, co spowodowało, że model dokonał błędnej klasyfikacji.



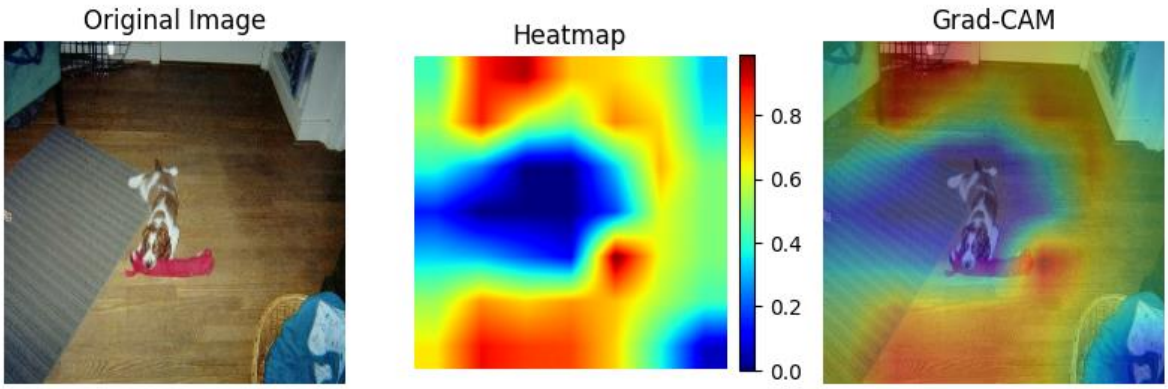
Przykład 2: tutaj natomiast widzimy fajny przykład tego, że model dobrze się nauczył. U psów rasy Bloodhound bardzo wyróżnia się głowa, dlatego nasz model zwraca najwięcej uwagi na głowę przy zgadywaniu psów tej rasy.



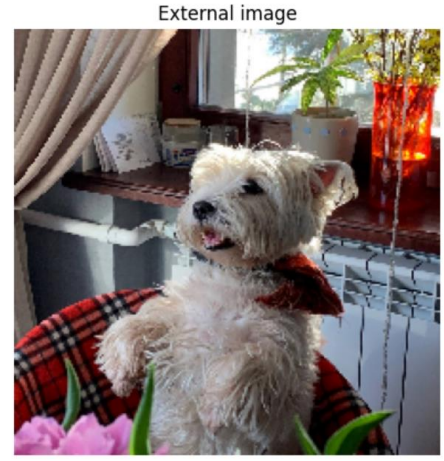
Przykład 3: Kolejny przykład pomyłki modelu, tym razem taksamo wynikający z tego faktu, iż te dwie rasy są z jednej rodziny, i je psy są bardzo podobne. Oba psa są białe z czarnymi oczami i czarnym nosem. Nawet oceniając to samodzielnie łatwo się pomylić



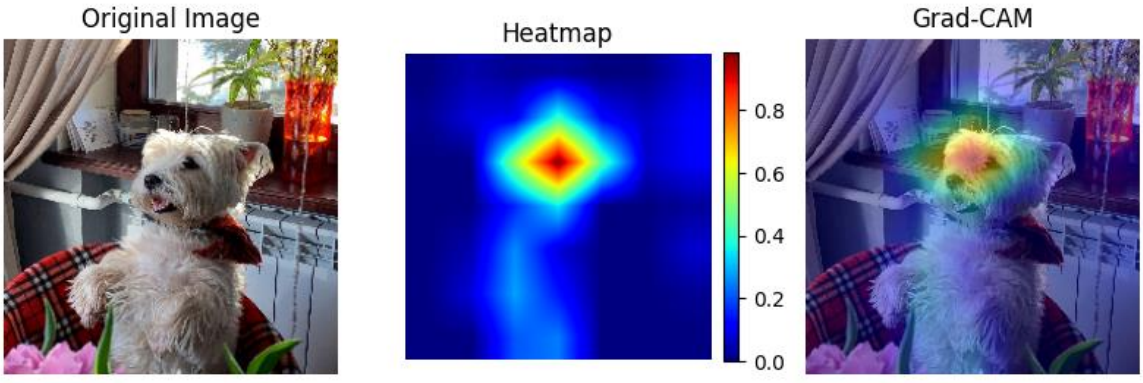
Przykład 4: Mapa trafień pokazuje, że model zgadywał nie na podstawie wyglądu psa, ale na podstawie jego otoczenia, co jest niepoprawne. Może to wskazywać, że model został przetrenowany i po prostu zapamiętał obrazy. Można to uzasadnić tym, że pies jest bardzo mały w porównaniu do otaczających go obiektów, więc sieć nadała otoczeniu większe znaczenie niż psu.



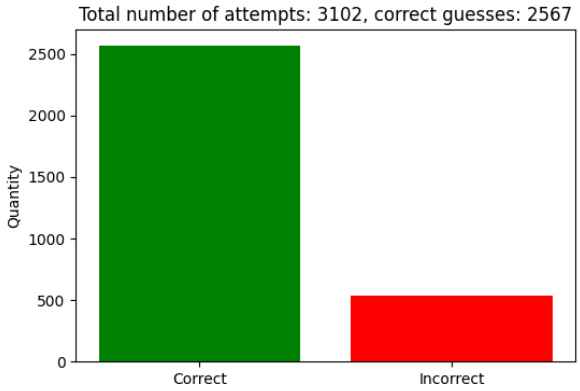
Przykład 5: W ostatnim przykładzie spróbowaliśmy dodać własne zdjęcie psa rasy West Highland White Terrier, aby sprawdzić jak sieć neuronowa rozpoznaje obraz, który nie pochodzi ze zbioru danych uczących.



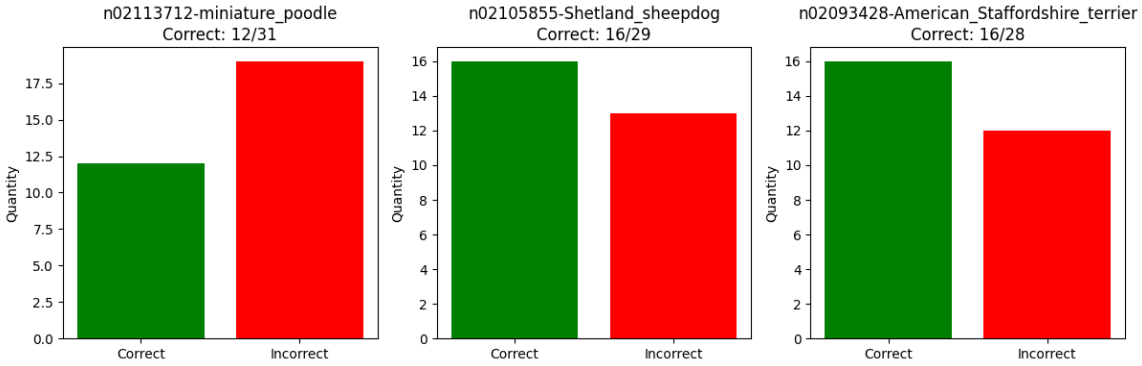

```
Top predictions:
n02098286-West_Highland_white_terrier – 91.91%
n02096177-cairn – 6.95%
n02085936-Maltese_dog – 0.46%
n02098105-soft-coated_wheaten_terrier – 0.23%
n02097474-Tibetan_terrier – 0.18%
```



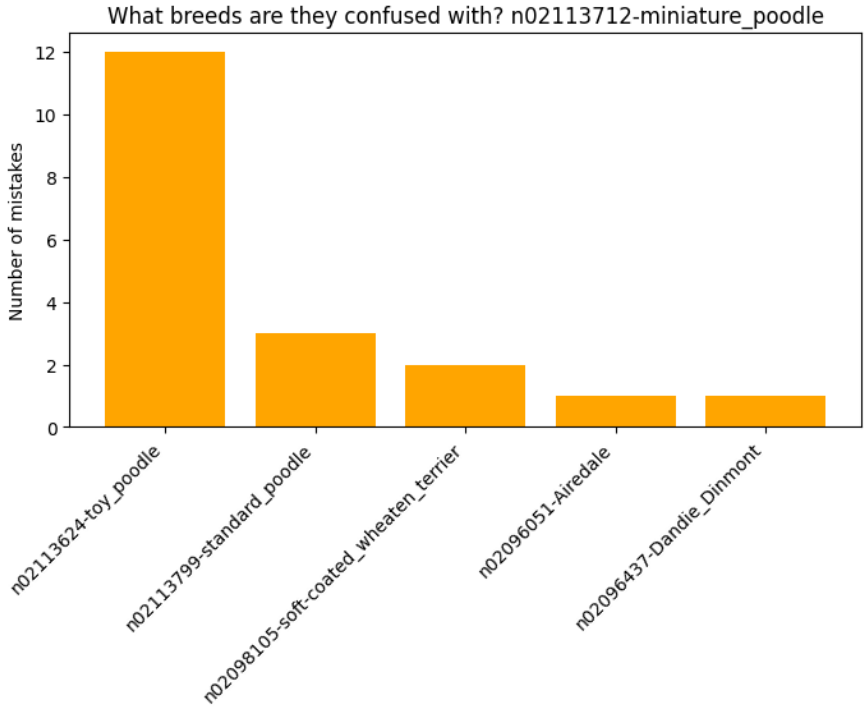
Poniżej przedstawiono schematyczne wykresy ilustrujące porównania oraz zestawienia statystyczne.



Rysunek 10: Całkowita liczba poprawnych i niepoprawnych przewidywań modelu w zbiorze testowym



Rysunek 11: Trzy rasy z największą liczbą błędów



Rysunek 12: Jakie inne rasy model najczęściej myli z rasą najbardziej błędną

Wnioski

Projekt polegał na stworzeniu i wytrenowaniu sieci neuronowej do rozwiązania średniozaawansowanego problemu klasyfikacji obrazów, jakim jest rozpoznawanie ras psów. Na początku została zaimplementowana prosta konwolucyjna sieć neuronowa (CNN), zbudowana od podstaw. Model ten działał poprawnie, jednak okazał się zbyt prosty i zbyt słaby, aby skutecznie rozwiązać tak złożone zadanie, jak klasyfikacja 120 ras psów na podstawie około 20 000 obrazów. Zbudowanie własnej sieci pozwoliło jednak lepiej zrozumieć działanie poszczególnych warstw oraz drogę, jaką przechodzi obraz od wejścia do modelu aż do etapu predykcji.

W kolejnym etapie do modelu została dodana wstępnie wytrenowana sieć EfficientNetB0. Zastosowanie transfer learningu znacząco przyspieszyło proces uczenia oraz poprawiło dokładność rozpoznawania ras psów. Następnie przeprowadzono próbę fine-tuningu, polegającą na dalszym trenowaniu wybranych warstw EfficientNetB0. Jednak w tym przypadku fine-tuning doprowadził do pogorszenia wyników na zbiorze testowym, dlatego został on ostatecznie wyłączony, a jako model końcowy pozostawiono wersję opartą wyłącznie na transfer learningu.

Nieudana próba fine-tuningu dostarczyła jednak cennego doświadczenia w zakresie dostrajania wstępnie wytrenowanych sieci oraz doboru parametrów uczenia dla ich ostatnich warstw. Pokazała również, że fine-tuning nie zawsze prowadzi do poprawy jakości modelu, zwłaszcza gdy model wstępnie wytrenowany jest już dobrze dopasowany do danego zadania.

Dla porównania przeprowadzono także trening z wykorzystaniem innej wstępnie wytrenowanej architektury – ResNet50. Uzyskane wyniki były gorsze niż w przypadku EfficientNetB0, co potwierdziło, że wybór EfficientNetB0 był trafny dla tego problemu.

Na ostatnim etapie projektu dokonano wizualnej analizy działania modelu. Za pomocą map cieplnych (Grad-CAM) zaprezentowano, na które fragmenty obrazu sieć zwraca uwagę podczas podejmowania decyzji. Dodatkowo przygotowano wykresy statystyczne pokazujące, które rasy psów są najczęściej mylone oraz ile poprawnych i błędnych klasyfikacji wykonuje model po 20 epokach treningu.

W przyszłości model można ulepszyć na kilka sposobów. Przede wszystkim warto zastosować większy i bardziej zróżnicowany zbiór danych treningowych, co pozwoliłoby zmniejszyć ryzyko przeuczenia i poprawić zdolność generalizacji modelu. Kolejnym krokiem mogłoby być użycie bardziej zaawansowanych technik augmentacji danych, takich jak losowe przycinanie, zmiany oświetlenia czy perspektywy, aby model lepiej radził sobie z różnorodnymi zdjęciami psów.

Możliwe jest również przetestowanie innych nowoczesnych architektur, takich jak EfficientNet w większych wariantów (np. B3 lub B4) albo Vision Transformers. Fine-tuning mógłby zostać ponownie rozważony przy użyciu mniejszego learning rate i

mniejszej liczby odblokowanych warstw, co mogłoby pozwolić na delikatne dostosowanie modelu bez ryzyka nadmiernego dopasowania do danych treningowych.

Literatura

https://gitlab.com/agorniak/podstawy-sieci-neuronowych/-/tree/main?ref_type=heads
https://www.geeksforgeeks.org/python/get-current-directory-python/
https://keras.io/api/applications/efficientnet/
https://medium.com/@sanjay_dutta/designing-your-own-convolutional-neural-network-cnn-model-a-step-by-step-guide-for-beginners-4e8b57836c81
https://deeplizard.com/learn/video/ZjM_XQa5s6s
https://medium.com/analytics-vidhya/this-blog-post-aims-at-explaining-the-behavior-of-different-algorithms-for-optimizing-gradient-46159a97a8c1
https://www.geeksforgeeks.org/machine-learning/training-vs-testing-vs-validation-sets/
https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image_dataset_from_directory
https://towardsdatascience.com/how-to-easily-draw-neural-network-architecture-diagrams-a6b6138ed875/