DOKUMENTACJA PROJEKTU

Sztuczna inteligencja

Rozpoznawanie Gatunków Zwierząt Morskich

Julia Kałuża

Spis treści

1. *Wstęp teoretyczny* 2
   - 1.1. *Sieci neuronowe* 2
   - 1.2. *Konwolucyjna sieć neuronowa* 3
   - 1.3. *Architektura EfficientNetB1* 3
   - 1.4. *Dlaczego EfficientNetB1?* 4
2. *Implementacja - Python* 5
   - 2.1. *Biblioteki i komponenty modelu* 5
   - 2.2. *Ścieżki i parametry* 6
   - 2.3. *Budowa modelu* 6
   - 2.4. *Zamrożenie części warstw (transfer learning)* 7
   - 2.5. *Kompilacja modelu* 7
   - 2.6. *Przygotowanie generatorów danych* 7
   - 2.7. *Ustawienie EarlyStopping* 8
   - 2.8. *Trening* 8
   - 2.9. *Zapis modelu* 9
3. *Wyniki treningu i najlepszy model* 9
4. *Predykcja dla danych testowych* 10
5. *Wykresy* 11
   - 5.1. *Macierz pomyłek* 11
   - 5.2. *Precyzja dla każdej klasy* 12
6. *Interfejs użytkownika i obsługa aplikacji* 13
   - 6.1. *Opis działania – krok po kroku* 13
   - 6.2. *Testowanie modelu przy użyciu aplikacji* 16
7. *Wnioski* 18
8. *Źródła* 18

1. Wstęp teoretyczny

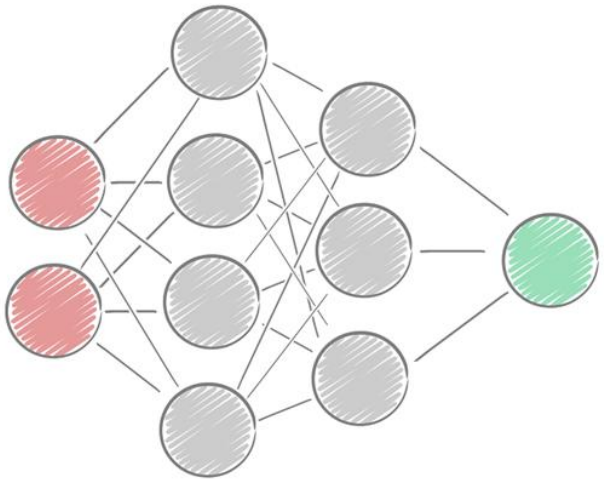
W naszym projekcie wytrenowałyśmy model, który rozpoznaje gatunki zwierząt morskich na podstawie zdjęcia. Model potrafi odróżnić **13 różnych gatunków**, takich jak np. delfin, krab, meduza czy rekin. Do realizacji zadania wykorzystałyśmy konwolucyjną sieć neuronową (CNN), ponieważ dobrze radzi sobie z rozpoznawaniem obrazów. Do nauki użyłyśmy podejścia uczenia z nadzorem – model uczy się na podstawie zdjęć, które mają przypisane poprawne etykiety (nazwy gatunków). Dzięki temu może porównywać swoje przewidywania z prawidłowymi odpowiedziami i stopniowo się poprawiać.

Zbiór danych (wynoszący 7300 zdjęć – co daje ok. 550 zdjęć na gatunek) został podzielony na trzy części:

- **Train (treningowy – 70% zbioru)** – do nauki modelu,
- **Validation (walidacyjny – 15% zbioru)** – do sprawdzania postępów w czasie treningu,
- **Test (testowy – 15% zbioru)** – do sprawdzenia, jak dobrze działa gotowy model.

1.1. Sieci neuronowe

Sztuczna sieć neuronowa to model matematyczny inspirowany działaniem ludzkiego mózgu. Składa się z warstw połączonych ze sobą "neuronów", które przetwarzają dane wejściowe i uczą się zależności między nimi a oczekiwanym wynikiem. Każdy neuron wykonuje proste operacje matematyczne, ale duża liczba warstw oraz odpowiednie funkcje aktywacji pozwalają sieci uczyć się bardzo złożonych wzorców i struktur w danych.



Rysunek 1: Sieć neuronowa. Źródło: https://miroslawmamczur.pl/czym-jest-i-jak-sie-uczy-sztuczna-siec-neuronowa/

1.2. Konwolucyjna sieć neuronowa

Konwolucyjne sieci neuronowe (CNN) to rodzaj sztucznych sieci neuronowych, które zostały stworzone specjalnie do rozpoznawania obrazów. Ich działanie opiera się na automatycznym wykrywaniu cech, takich jak krawędzie, kształty czy wzory.

CNN składają się z kilku typów warstw:

- **Warstwa splotowa (konwolucyjna)** – wyszukuje istotne cechy obrazu przy pomocy filtrów,
- **Warstwa aktywacji** – wprowadza nieliniowość, pozwalając sieci uczyć się złożonych wzorców,
- **Warstwa łączenia (pooling)** – zmniejsza rozmiar danych, zostawiając najważniejsze informacje,
- **Warstwa końcowa (gęsta)** – odpowiada za ostateczną klasyfikację obrazu.

Dzięki takiej budowie CNN bardzo dobrze radzą sobie z analizą zdjęć – jak w naszym projekcie – do rozpoznawania gatunków zwierząt ze zdjęć.

1.3. Architektura EfficientNetB1

EfficientNetB1 to jeden z modeli należących do rodziny EfficientNet, zaprojektowanej przez Google. Architektura ta została opracowana tak, aby przy stosunkowo niewielkiej liczbie parametrów osiągać bardzo dobrą dokładność w zadaniach klasyfikacji obrazów.

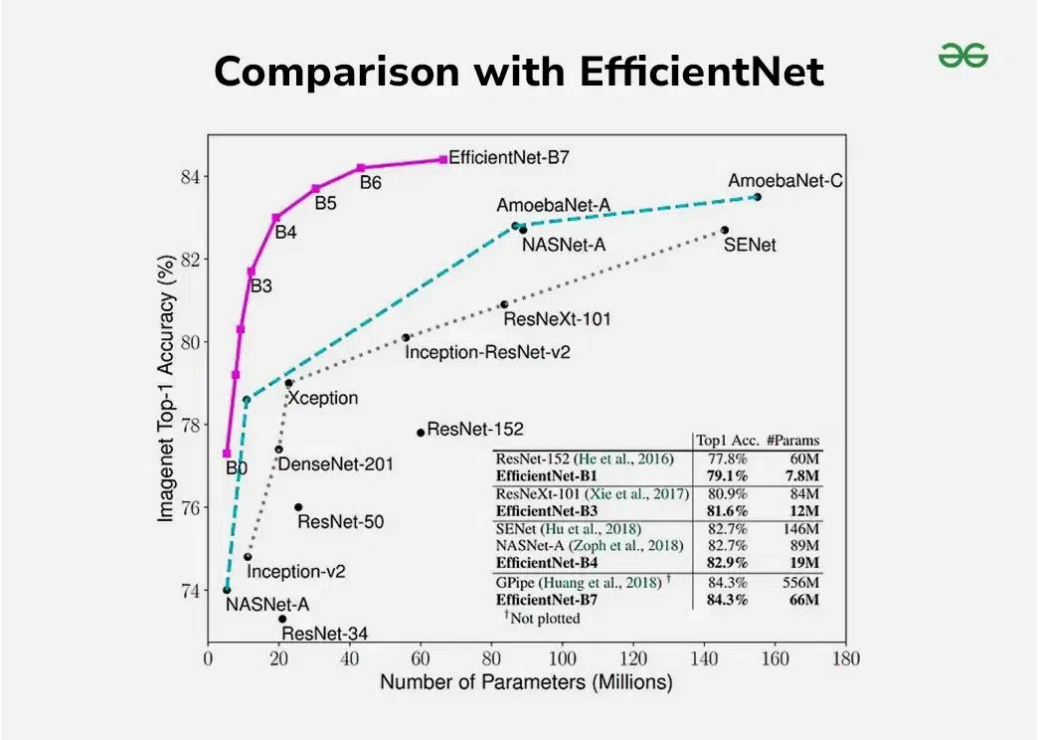
EfficientNet opiera się na tzw. skalowaniu złożonym (compound scaling), które równocześnie zwiększa trzy parametry sieci: głębokość (liczbę warstw), szerokość (liczbę neuronów) i rozdzielczość wejściowego obrazu. Dzięki temu model może lepiej wykrywać cechy w obrazach, nie zużywając przy tym zbyt dużo pamięci ani czasu obliczeniowego.

EfficientNetB1 zawiera warstwy konwolucyjne, warstwy normalizacji i aktywacji oraz końcowe warstwy gęste do klasyfikacji. Model został wcześniej wytrenowany na zbiorze ImageNet, co umożliwia jego wykorzystanie w transfer learningu.

1.4. Dlaczego EfficientNetB1?

Zdecydowałyśmy się na model EfficientNetB1, ponieważ oferuje bardzo dobry balans między jakością a wydajnością. W porównaniu do starszych architektur, takich jak DenseNet-201 czy ResNet50, EfficientNetB1 uzyskuje lepsze wyniki przy mniejszej liczbie parametrów, co przyspiesza trening i zmniejsza zużycie zasobów.

Wersja B1 jest nieco dokładniejsza niż podstawowy EfficientNetB0, ale nadal lekka i łatwa do uruchomienia na typowym komputerze. Dodatkowo dostępność wstępnie wytrenowanych wag (ImageNet) pozwoliła zastosować transfer learning, co znacząco poprawiło jakość modelu przy ograniczonej liczbie danych treningowych.



Rysunek 2: Porównanie dokładności i liczby parametrów różnych modeli CNN na zbiorze ImageNet. EfficientNetB1 osiąga wyższą dokładność niż ResNet-152 czy DenseNet-201, przy znacznie mniejszej liczbie parametrów. Źródło: https://www.geeksforgeeks.org/efficientnet-architecture/

2. Implementacja - Python

Najpierw podzieliłyśmy zdjęcia na zbiory: treningowe, walidacyjne i testowe:

```
import splitfolders
# dzielimy dane w proporcjach: 70% train, 15% val, 15% test
splitfolders.ratio("data_deleted", output="data_deleted_split", seed=42,
ratio=(.7, .15, .15))
```

2.1. Biblioteki i komponenty modelu

Biblioteki

```
import os
import tensorflow as tf
import matplotlib.pyplot as plt
```

- **os** – umożliwia odczyt struktury katalogów w celu pobrania nazw klas.
- **tensorflow** – biblioteka do budowy i trenowania modelu głębokiego uczenia.
- **matplotlib.pyplot** – do wizualizacji wyników treningu.

Komponenty

```
#komponenty modelu
EfficientNetB1 = tf.keras.applications.efficientnet.EfficientNetB1
ImageDataGenerator = tf.keras.preprocessing.image.ImageDataGenerator
Model = tf.keras.Model
Dense = tf.keras.layers.Dense
Dropout = tf.keras.layers.Dropout
GlobalAveragePooling2D = tf.keras.layers.GlobalAveragePooling2D
Adam = tf.keras.optimizers.Adam
EarlyStopping = tf.keras.callbacks.EarlyStopping
```

- **EfficientNetB1** – pretrenowany model konwolucyjny z rodziny EfficientNet. Użyty do ekstrakcji cech ze zdjęć.
- **ImageDataGenerator** – generator do ładowania i augmentacji obrazów w czasie rzeczywistym.
- **Model** – klasa służąca do budowy kompletnego modelu na podstawie wejścia i wyjścia.
- **Dense** – warstwa w pełni połączona. Używana w warstwach ukrytych i wyjściowej (softmax).
- **Dropout** – warstwa regularizacyjna – losowo wyłącza część neuronów podczas treningu.
- **GlobalAveragePooling2D** – redukuje wymiary przestrzenne do jednej wartości na kanał – służy do spłaszczania danych.
- **Adam** – optymalizator oparty na adaptacyjnym tempie uczenia. Szybki i skuteczny.
- **EarlyStopping** – callback zatrzymujący trening, jeśli metryka walidacyjna przestanie się poprawiać.

2.2. Ścieżki i parametry

Ścieżki - do danych treningowych i walidacyjnych oraz do pliku, w którym zostanie zapisany wytrenowany model.

```
TRAIN_DIR = 'dataset/train'
VAL_DIR = 'dataset/val'
MODEL_SAVE_PATH = 'model/saved_model/model_efficientnetB1_13.keras'
```

Parametry

```
CLASSES = sorted(os.listdir(TRAIN_DIR))
IMG_SIZE = (240, 240)
BATCH_SIZE = 32
```

- **CLASSES** – lista nazw klas (np. nazw folderów ze zdjęciami), sortowana alfabetycznie.
- **IMG_SIZE** – rozmiar, do którego przeskalowywane są wszystkie obrazy.
- **BATCH_SIZE** – liczba przykładów w jednej partii danych podczas treningu.

2.3. Budowa modelu

```
base_model = EfficientNetB1(weights='imagenet', include_top=False,
input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3))
```

Tworzony jest bazowy model EfficientNetB1, wczytany z pretrenowanymi wagami z ImageNet. Ustawienie include_top=False umożliwia dołączenie własnych warstw klasyfikujących.

```
x = base_model.output # pobieramy ostatnie wyjście z EfficientNet
x = GlobalAveragePooling2D()(x) # spłaszczamy cechy przestrzenne do wektora
x = Dropout(0.3)(x) # zastosowanie Dropoutu - wyłączenie 30% neuronów
losowo
x = Dense(384, activation='relu')(x) # gęsta warstwa ukryta z aktywacją
ReLU
predictions = Dense(len(CLASSES), activation='softmax')(x) # warstwa
wyjściowa z aktywacją softmax
model = Model(inputs=base_model.input, outputs=predictions) # budujemy
kompletny model
```

- **GlobalAveragePooling2D** – redukuje liczbę parametrów i ryzyko przeuczenia.
- **Dropout** (30%) – wyłącza losowo część neuronów w trakcie treningu, dzięki czemu jest mniejsze ryzyko przeuczenia się modelu.
- **Dense(384, relu)** – warstwa ukryta z 384 jednostkami, zapewnia większą zdolność reprezentacyjną.
- Warstwa wyjściowa z softmax dopasowana do liczby klas – zwraca rozkład prawdopodobieństw.

2.4. *Zamrożenie części warstw*

```
for layer in base_model.layers[:80]:
    layer.trainable = False
```

Zamrożenie pierwszych 80 warstw pozwala na zachowanie ogólnych cech wyuczonych na ImageNet (potrafią już rozróżniać krawędzie i kolory na zdjęciach). Dzięki temu model szybciej koncentruje się na nauce specyfiki danych dotyczących zwierząt morskich.

2.5. *Kompilacja modelu*

```
loss_fn = tf.keras.losses.CategoricalCrossentropy(label_smoothing=0.05) #
funkcja straty z wygładzeniem etykiet
model.compile(optimizer=Adam(learning_rate=1e-4), loss=loss_fn,
metrics=['accuracy']) # kompilacja
```

Funkcja straty **CategoricalCrossentropy** z wygładzaniem etykiet (label_smoothing) zapobiega nadmiernemu dopasowaniu modelu do etykiet treningowych.

Użycie optymalizatora Adam z learning_rate=0.0001 jest typowe dla transfer learningu.

2.6. *Przygotowanie generatorów danych*

```
train_datagen = ImageDataGenerator(
    rescale=1./255, # przeskalowanie pikseli do zakresu 0-1
    rotation_range=20, # obrót obrazu o losowy kąt
    zoom_range=0.2, # losowe przybliżenia
    width_shift_range=0.1, # przesunięcie w poziomie
    height_shift_range=0.1, # przesunięcie w pionie
    horizontal_flip=True # odbicie w poziomie
)
```

train_datagen – przekształca obrazy w czasie rzeczywistym, stosując augmentację:

- rotacje, przesunięcia, przybliżenia i odbicia,
- normalizacja pikseli do przedziału [0, 1].

```
val_datagen = ImageDataGenerator(rescale=1./255) # walidacja bez
augmentacji
```

Dane walidacyjne są jedynie normalizowane w **val_datagen** – bez augmentacji, aby zapewnić obiektywną ocenę modelu.

```
train_gen = train_datagen.flow_from_directory(
    TRAIN_DIR, target_size=IMG_SIZE, batch_size=BATCH_SIZE,
class_mode='categorical')

val_gen = val_datagen.flow_from_directory(
    VAL_DIR, target_size=IMG_SIZE, batch_size=BATCH_SIZE,
class_mode='categorical')
```

Obie funkcje wczytują obrazy z katalogów i przygotowują je do treningu w partiach (batch_size).

2.7. Ustawienie EarlyStopping

```
# EarlyStopping - wcześniejsze zatrzymanie treningu jeśli nie ma poprawy
wyników
early_stop = EarlyStopping(monitor='val_accuracy', patience=7,
restore_best_weights=True)
```

Mechanizm EarlyStopping zatrzymuje trening, jeśli dokładność walidacyjna nie poprawia się przez 7 epok. Przywracane są najlepsze wagi i model się zapisuje.

2.8. Trening

Faza 1 (zamrożone warstwy) – trening przez 10 epok:

```
#faza 1: trening z zamroxonymi warstwami
history1 = model.fit(train_gen, epochs=10, validation_data=val_gen)
```

W pierwszej fazie uczone są tylko nowo dodane warstwy klasyfikujące, natomiast bazowe warstwy EfficientNet (pierwsze 80 warstw) pozostają niezmienione.

Faza 2 - Fine-tuning (odmrożenie kolejnych warstw) – trening przez kolejne 30 epok:

```
for layer in base_model.layers[80:]:
    layer.trainable = True
```

Druga faza treningu polega na stopniowym dostrajaniu głębszych warstw modelu do danych domenowych.

```
#kompilacja z mniejszym learning rate do delikatnego dostrajania
model.compile(optimizer=Adam(learning_rate=1e-5), loss=loss_fn,
metrics=['accuracy'])
#kontynuacja treningu z EarlyStopping
history2 = model.fit(train_gen, epochs=30, validation_data=val_gen,
callbacks=[early_stop])
```

Dla fine-tuningu użyłyśmy mniejszego współczynnika uczenia (0.00001), aby zachować stabilność wyuczonych wag.

2.9. Zapis modelu

```
model.save(MODEL_SAVE_PATH)
print("\u2705 Model zapisany jako:", MODEL_SAVE_PATH)
```

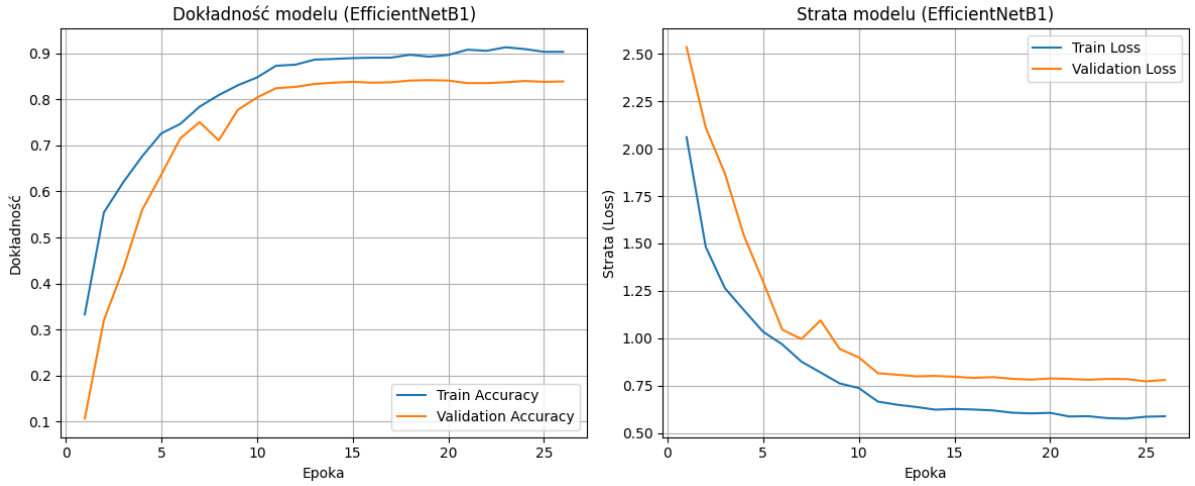
Model jest zapisywany do formatu .keras, który jest zalecany w TensorFlow 2.x jako domyślny sposób przechowywania modelu i wag.

3. Wyniki treningu i najlepszy model

Po przeprowadzeniu treningu modelu przy użyciu opisanego wyżej skryptu, uzyskałyśmy najlepszy wynik na zbiorze walidacyjnym przy następujących parametrach:

- **Model bazowy**: EfficientNetB1 (transfer learning z ImageNet)
- **Zamrożone warstwy**: 80 pierwszych
- **Rozmiar obrazu**: 240 x 240 pikseli
- **Batch size**: 32
- **Faza 1**: learning rate: 1e-4 (trening nowych warstw), 10 epok
- **Faza 2**: learning rate: 1e-5 (fine-tuning), 30 epok
- **Funkcja straty**: CategoricalCrossentropy (label smoothing = 0.05)
- **EarlyStopping**: monitorowanie val_accuracy, patience = 7 (zatrzymuje jeśli przez 7 epok wynik na zbiorze walidacyjny się nie poprawił).

Testowałyśmy różne ustawienia powyższych parametrów, aż w końcu udało nam się uzyskać model o dokładności walidacyjnej: 84%:



Rysunek 3: Wykres dokładności modelu i funkcji straty. Trening trwał 26 epok – zatrzymany przez EarlyStopping.

4. Predykcja dla danych testowych

Poniższy kod odpowiada za ocenę skuteczności wytrenowanego modelu na niewidzianych wcześniej danych testowych. Wykonywana jest predykcja, porównanie z rzeczywistymi etykietami oraz wygenerowanie raportu klasyfikacji, macierzy pomyłek i wykresów precyzji.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import sklearn.metrics

#ścieżki
MODEL_PATH = 'model/saved_model/model_efficientnetB1_13.keras'
TEST_DIR = 'dataset/test'
IMG_SIZE = (240, 240)
BATCH_SIZE = 32 #liczba obrazów, które model przetwarza jednocześnie

#zaladuj model
model = tf.keras.models.load_model(MODEL_PATH)

#generator danych testowych
ImageDataGenerator = tf.keras.preprocessing.image.ImageDataGenerator
test_datagen = ImageDataGenerator(rescale=1./255)
test_gen = test_datagen.flow_from_directory(
    TEST_DIR,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=False
)

#klasy
class_indices = test_gen.class_indices
class_labels = list(class_indices.keys())

#predykcje
predictions = model.predict(test_gen)
y_pred = np.argmax(predictions, axis=1)
y_true = test_gen.classes

#raport klasyfikacji
report = sklearn.metrics.classification_report(
    y_true, y_pred, target_names=class_labels, output_dict=True
)
print(f"Dokładność ogólna (accuracy): {accuracy:.2f}")
```

Wynik:

```
Dokładność ogólna (accuracy): 0.84


Process finished with exit code 0
```

Rysunek 4: Wynik ogólnej dokładności modelu na danych testowych – 84%

5. Wykresy

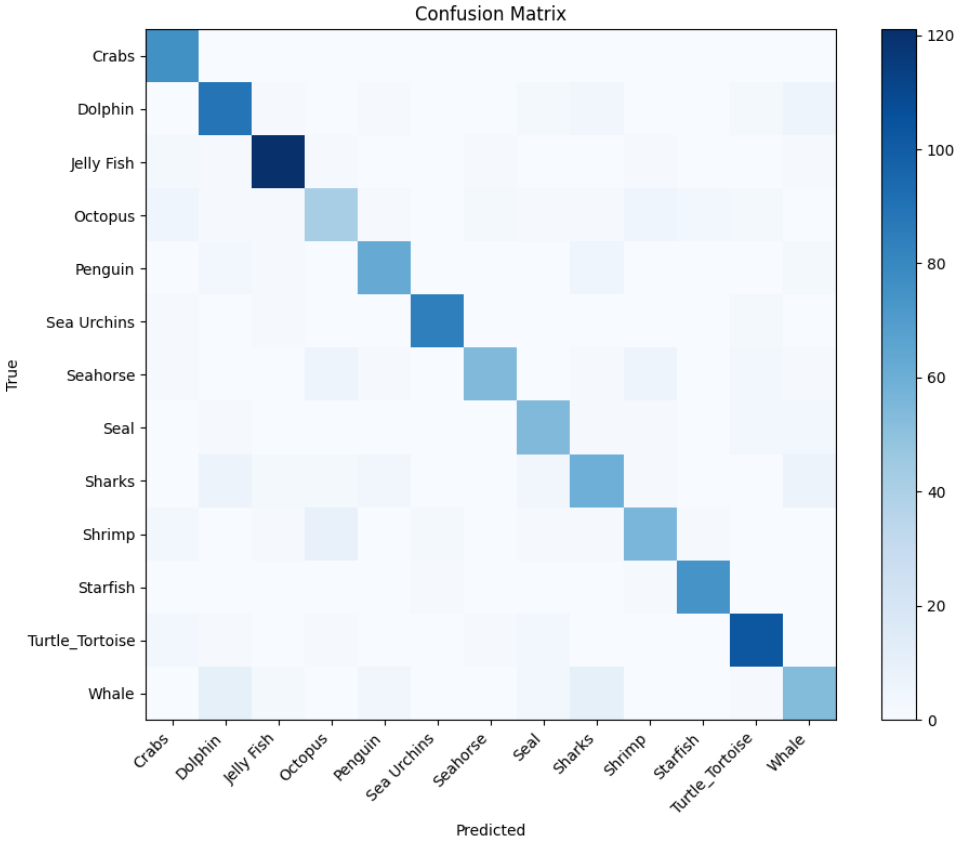
Na podstawie powyższego skryptu do oceny modelu na danych testowych, wygenerowałyśmy następujące wykresy:

5.1. Macierz pomyłek

Macierz pomyłek pozwala zobaczyć, które klasy są najczęściej ze sobą mylone.

Skrypt:

```
#confusion matrix
conf_matrix = sklearn.metrics.confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 8))
plt.imshow(conf_matrix, cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.xticks(ticks=np.arange(len(class_labels)), labels=class_labels,
rotation=45, ha='right')
plt.yticks(ticks=np.arange(len(class_labels)), labels=class_labels)
plt.colorbar()
plt.tight_layout()
plt.savefig('confusion_matrix_B1_13.png')
plt.show()
```



Rysunek 5: Macierz pomyłek – na zbiorze testowym.

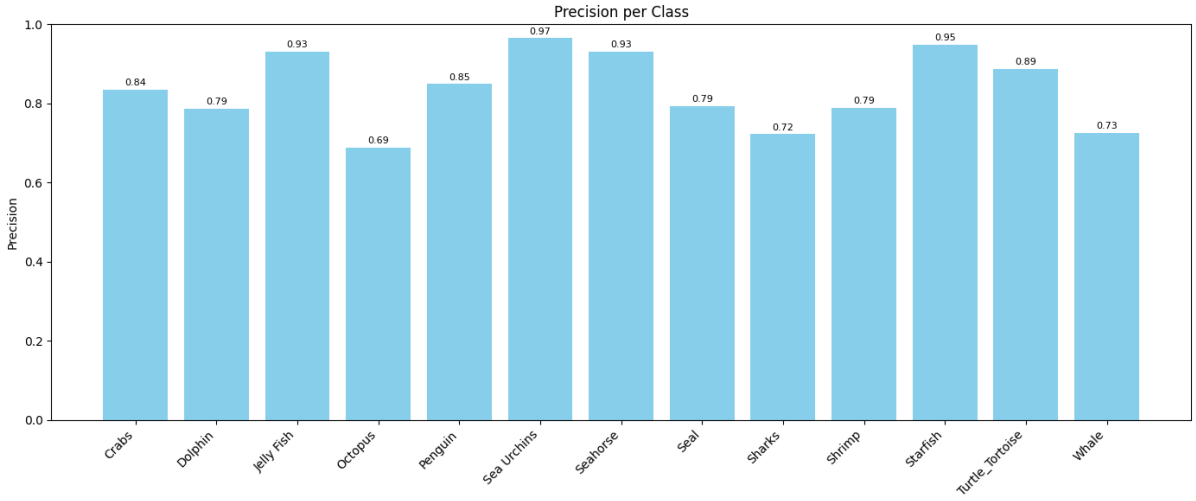
5.2. *Precyzja dla każdej klasy*

Dla każdej klasy (np. delfin, rekin, żółw) rysowany jest słupek z wartością precyzji. Dodatkowo wartości są wypisywane nad słupkami.

Skrypt:

```
#precyzja dla danej klasy
precisions = [report[label]['precision'] for label in class_labels]
plt.figure(figsize=(14, 6))
bars = plt.bar(class_labels, precisions, color='skyblue')
plt.xticks(rotation=45, ha='right')
plt.ylim(0, 1)
plt.ylabel('Precision')
plt.title('Precision per Class')
for bar, val in zip(bars, precisions):
    plt.text(bar.get_x() + bar.get_width() / 2, val + 0.01, f'{val:.2f}',
ha='center', fontsize=8)
plt.tight_layout()
plt.savefig('precision_per_class_B1_13.png')
plt.show()
```

Wynikowy wykres:



Rysunek 6: Wykres precyzji predykcji danej klasy - na zbiorze testowym.

Najwyższe precyzje:

Nazwa gatunku	Wartość
Sea Urchins	97%
Starfish	95%
Jelly Fish i Seahorse	93%

Najniższe precyzje:

Nazwa gatunku	Wartość
Octopus	69%
Whale	73%
Sharks	72%

Słabsze precyzje predykcji mogą wynikać z wizualnego podobieństwa między klasami lub zbyt małej ilości zdjęć do treningu dla niektórych gatunków.

6. Interfejs użytkownika i obsługa aplikacji

W ramach projektu stworzyłyśmy aplikację graficzną umożliwiającą użytkownikowi klasyfikację zdjęć zwierząt morskich przy pomocy naszego wytrenowanego modelu EfficientNetB1. Aplikacja została stworzona w języku Python z użyciem biblioteki CustomTkinter.

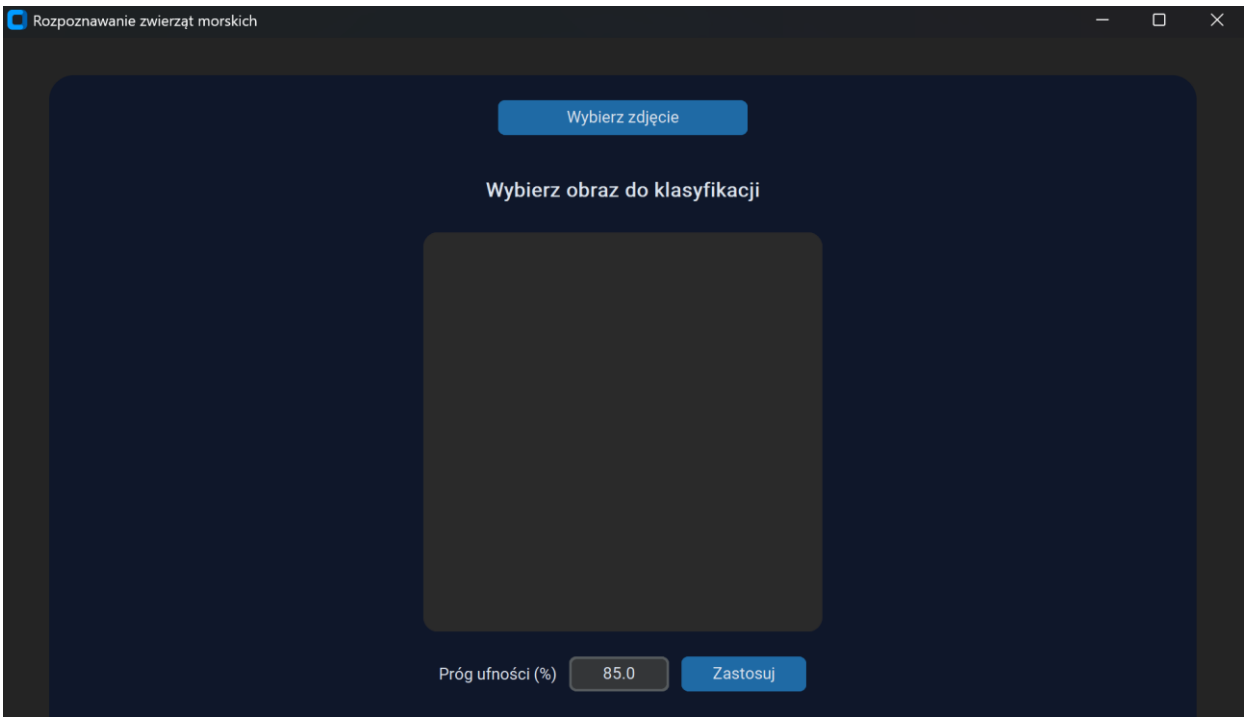
Naszym celem było stworzenie przyjaznego interfejsu użytkownika do testowania wytrenowanego modelu bez potrzeby korzystania z wiersza poleceń czy środowisk programistycznych. Aplikacja pozwala użytkownikowi:

- załadować zdjęcie ze swojego urządzenia,
- uzyskać informacje o rozpoznanym gatunku (jeśli nie rozpozna to o tym informuje),
- zmieniać próg ufności predykcji (domyślnie ustawiony na 85%)
- zobaczyć szczegóły klasyfikacji (trzy najbardziej prawdopodobne gatunki).

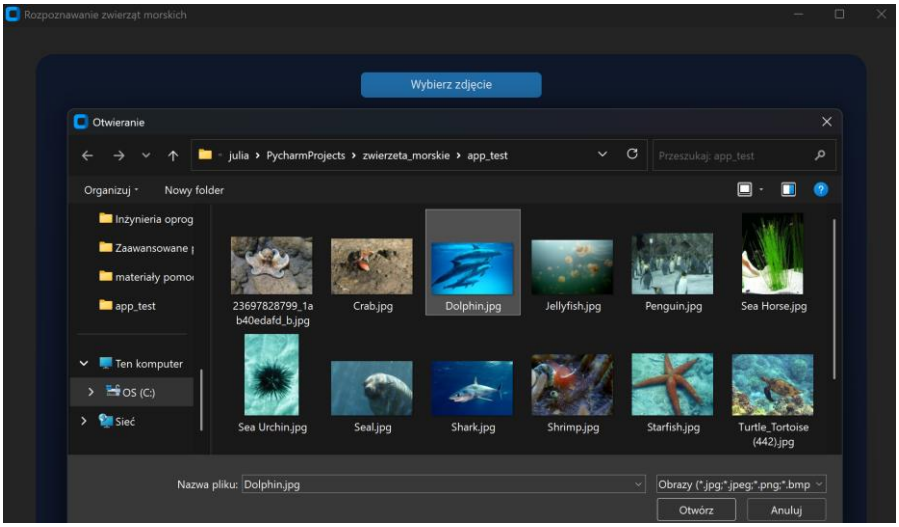
6.1. Opis działania – krok po kroku

Poniżej zaprezentujemy co znajduje się w interfejsie naszej aplikacji.

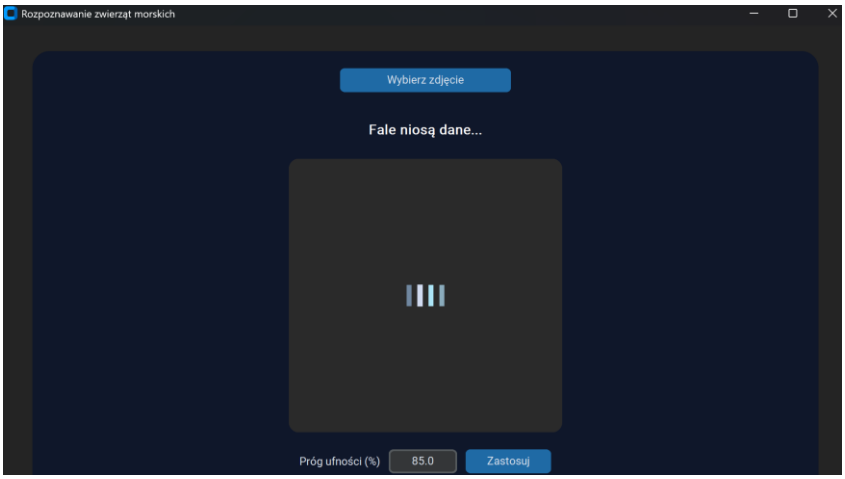
1. Użytkownik uruchamia aplikację i klika „Wybierz zdjęcie”.



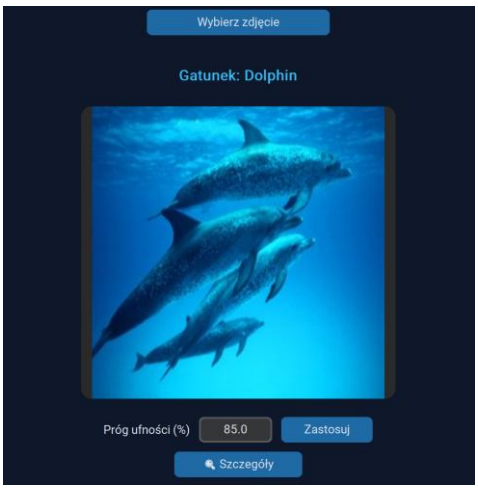
2. Możemy wybrać teraz zdjęcie z naszego urządzenia → Obraz jest skalowany do rozmiaru 240×240 pikseli i wysyłany do modelu.



3. Czekamy na wynik i podziwiamy interfejs...

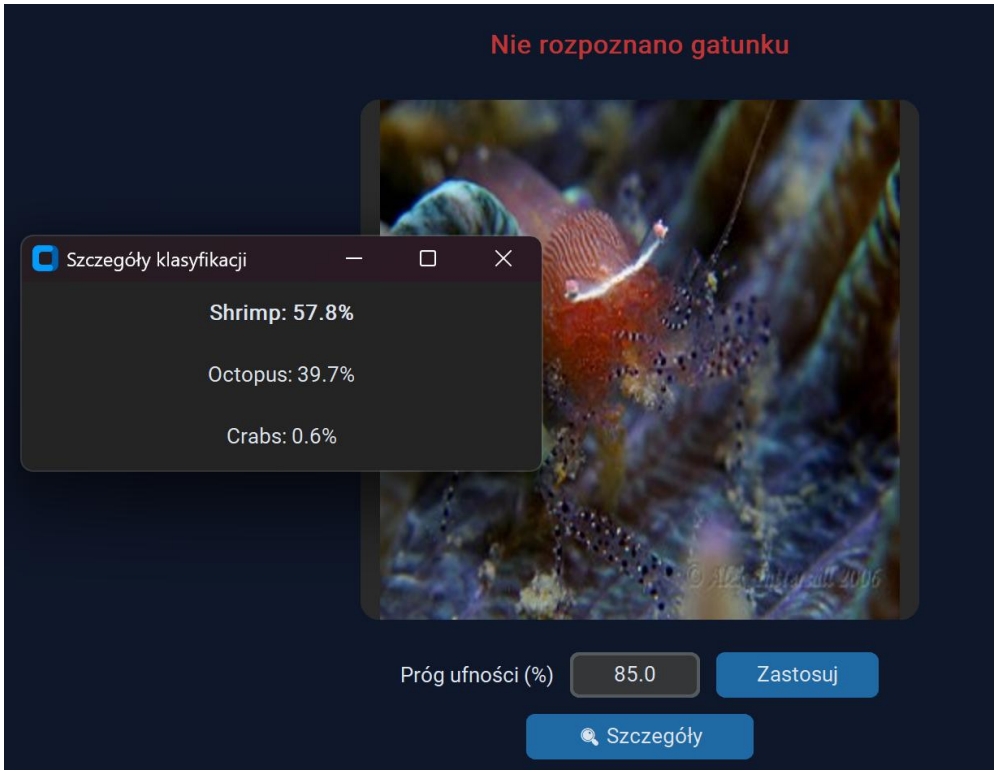


4. Po przetworzeniu, wynik predykcji wyświetlany jest w formie tekstowej. → Jeśli wynik jest pewny – użytkownik widzi nazwę gatunku.

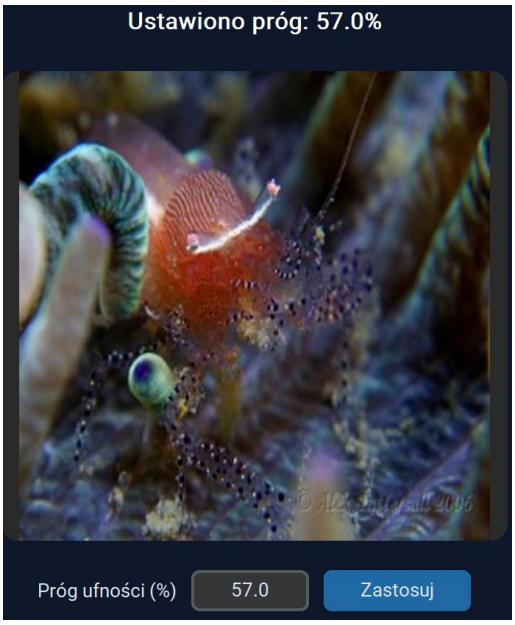


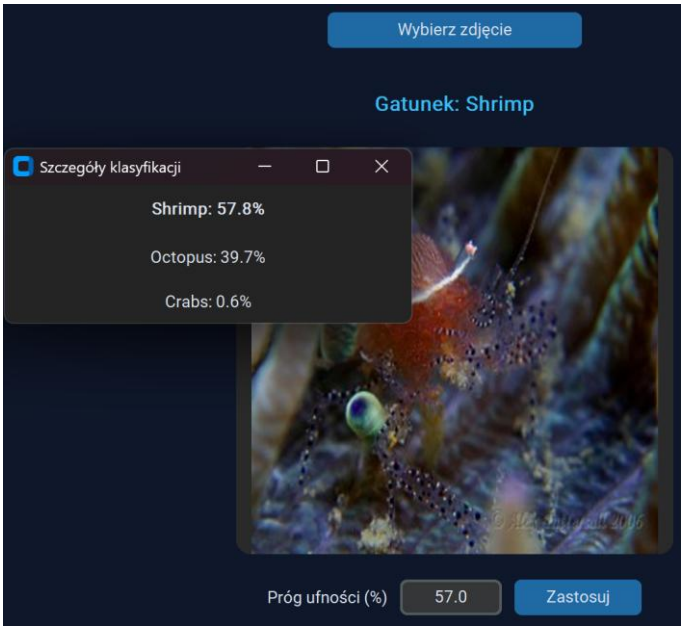
5. Jeśli wynik jest poniżej progu – pojawia się informacja o braku rozpoznania.

6. Opcjonalnie użytkownik może kliknąć „Szczegóły”, by zobaczyć trzy najlepsze dopasowania modelu z ich procentową oceną.

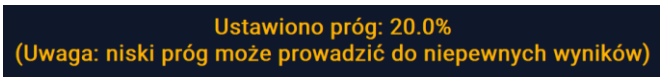


7. Próg ufności można dowolnie zmieniać. (wartość od 0.0 do 100.0) → Aby zapisać zmianę progu, należy kliknąć „Zastosuj”.





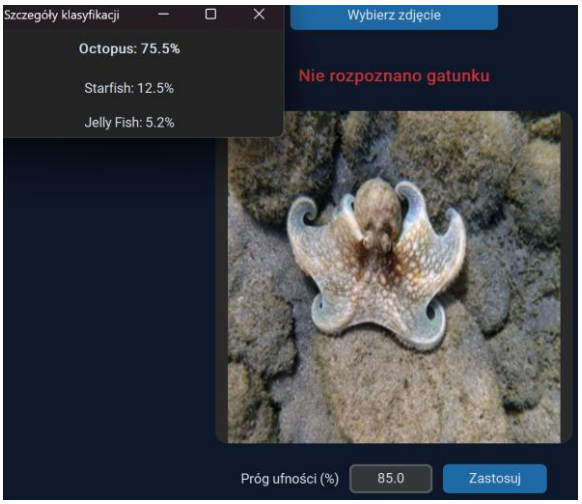
8. Jeśli próg nie mieści się w przedziale od 0.0 do 100.0 lub jest zbyt niski to jest informacja zwrotna.

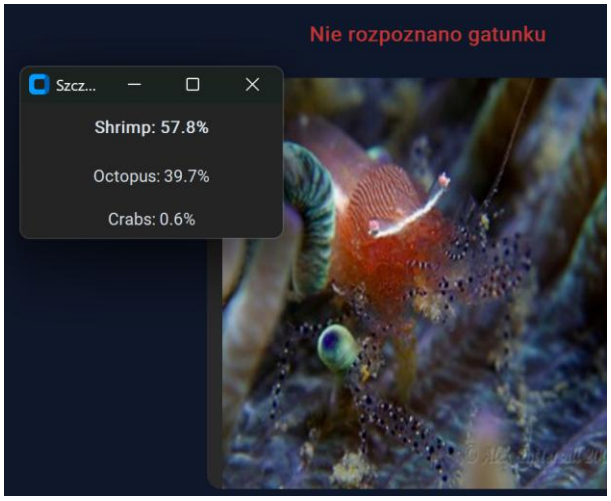


6.2. *Testowanie modelu przy użyciu aplikacji*

Zostawiamy próg ufności na poziomie 85%, bo chcemy żeby wynik był jak najbardziej pewny i testujemy wszystkie gatunki – 13.

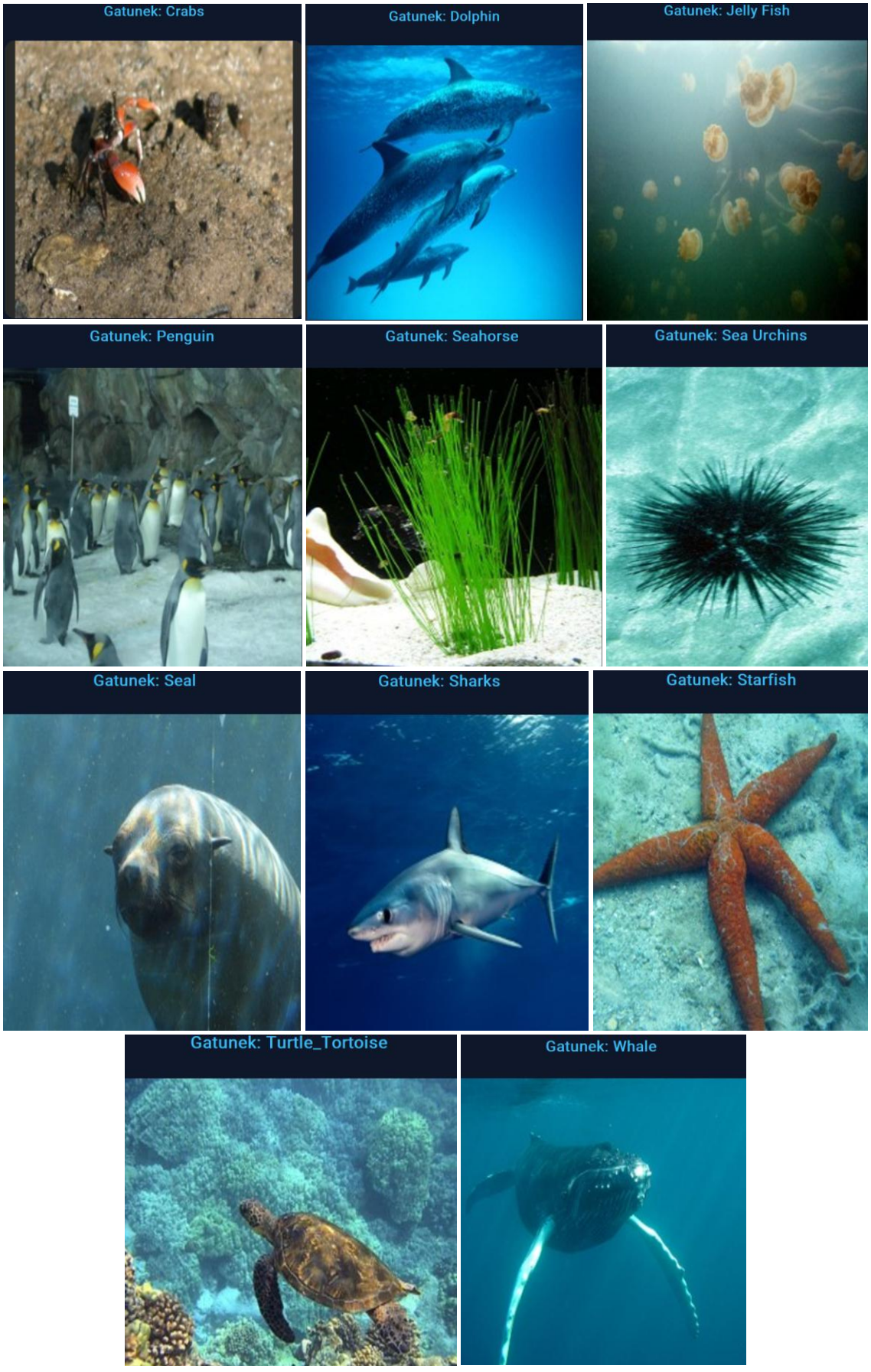
Błędne przewidywania (2):





W obu przypadkach zbyt niska pewność predykcji.

Prawidłowe przewidywania (11):



7. Wnioski

- Uzyskana dokładność na zbiorze testowym wyniosła około 84%, co uznajemy za bardzo dobry wynik jak na nasz zakres danych (po około 550 zdjęć na gatunek).
- Zastosowanie transfer learningu z gotowym modelem EfficientNetB1 pozwoliło nam osiągnąć wysoką skuteczność bez potrzeby trenowania sieci od podstaw.
- Zauważyłyśmy, że największe trudności pojawiły się przy klasie Octopus, gdzie model częściej się mylił. Może to wynikać z faktu, że ta klasa zawierała około 350 zdjęć, podczas gdy inne miały po około 550. Nierównowaga w danych mogła wpłynąć na jakość predykcji.
- Projekt dał nam możliwość wykorzystania w praktyce wiedzy z zakresu sztucznej inteligencji, analizy obrazów i programowania aplikacji. Praca nad nim była dla nas wartościowym i rozwijającym doświadczeniem.

8. Źródła

[1] *Sea Animals Image Dataset*. Kaggle

https://www.kaggle.com/datasets/vencerlanz09/sea-animals-image-dataste

[2] Mamczur, M. (2020). *Czym jest i jak się uczy sztuczna sieć neuronowa?*

https://miroslawmamczur.pl/czym-jest-i-jak-sie-uczy-sztuczna-siec-neuronowa/

[3] Mamczur, M. (2021). *Jak działają konwolucyjne sieci neuronowe (CNN)?*

https://miroslawmamczur.pl/jak-dzialaja-konwolucyjne-sieci-neuronowe-cnn/

[4] GeeksforGeeks. *EfficientNet Architecture*.

https://www.geeksforgeeks.org/efficientnet-architecture/

[5] Keras Documentation. *Image Classification with EfficientNet and Fine-Tuning*.

https://keras.io/examples/vision/image_classification_efficientnet_fine_tuning/