|  |  |  |  |
| --- | --- | --- | --- |
|  | Politechnika Bydgoska im. J. J. Śniadeckich  Wydział Telekomunikacji,  Informatyki i Elektrotechniki  **Zakład Systemów Teleinformatycznych** | |  |
| **Przedmiot** | Wdrażanie technik uczenia maszynowego | | |
| **Prowadzący** | mgr inż. Martyna Tarczewska | | |
| **Temat** | Klasyfikacja obrazów - pingwiny vs. żółwie | | |
| **Studenci (+ nr indeksów)** | Dorota Mockiewicz (1188287), Oliwia Paliwoda (118836) | | |
| **Ocena** |  | **Data oddania spr.** | …. |

Uwaga! Sprawozdanie z laboratorium powinno zawierać:

* wypełnioną tabelę,
* link do kodu,
* wykresy,
* wyniki,
* wnioski,
* co można ulepszyć

Spis treści

[Specyfikacja projektu 3](#_Toc168852325)

[Eksploracyjna analiza danych 4](#_Toc168852326)

[Wczytanie danych do tensora 8](#_Toc168852327)

[Funkcje pomocnicze do treningu i walidacji modelu 10](#_Toc168852328)

[Trening i walidacja 12](#_Toc168852329)

[Front z użyciem biblioteki gradio 17](#_Toc168852330)

[Wygląd aplikacji 18](#_Toc168852331)

[Uruchomienie programu przez Docker 19](#_Toc168852332)

[CI/CD 19](#_Toc168852333)

[Wrzucenie obrazu dockerowego na dowolną chmurę 20](#_Toc168852334)

## Specyfikacja projektu

Link do danych: <https://www.kaggle.com/datasets/abbymorgan/penguins-vs-turtles/data>

Github:

Temat: Klasyfikacja obrazów - pingwiny vs. żółwie.

Celem projektu jest stworzenie modelu uczenia maszynowego, który klasyfikuje obrazy na dwie kategorie: pingwiny lub żółwie. Zbiór danych zawiera 572 unikalne obrazki, z których 500 stanowi zbiór treningowy, a 72 zbiór walidacyjny. Klasy są zrównoważone, z równą liczbą obrazów dla każdej kategorii. Każdy obrazek przedstawia tylko jeden obiekt klasyfikowany i jest zapisany w formacie JPG.

## Eksploracyjna analiza danych

DataFrame df zawiera dane do zbioru treningowego, natomiast DataFrame df\_valid zawiera dane do zbioru testowego.

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

Rozmiary ramek danych są zgodne z oczekiwaniami:

Obraz zawierający tekst, Czcionka, zrzut ekranu, numer

Opis wygenerowany automatycznie

W obu zbiorach danych nie występują żadne brakujące wartości, co oznacza, że wszystkie oczekiwane wpisy są kompletnie wypełnione:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Czcionka

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Obraz zawierający tekst, Czcionka, zrzut ekranu, design

Opis wygenerowany automatycznie

Identyfikatory obrazków w obu zbiorach były unikalne i obejmowały wartości z przedziału od 0 do wielkości zbioru. Dane były sklasyfikowane zgodnie z dwiema kategoriami, przy czym wartość 1 odpowiadała pingwinowi, a wartość 2 żółwiowi. Obie klasy były równomiernie zbalansowane.

DataFrame train zawiera ścieżki do obrazków oraz identyfikatory obrazków, które pochodzą ze zbioru treningowego. Natomiast DataFrame valid zawiera ścieżki i identyfikatory obrazków ze zbioru walidacyjnego (testowego). Rozmiary zgadzają się z ramkami zawierającymi etykiety.

Obraz zawierający tekst, Czcionka, zrzut ekranu, numer

Opis wygenerowany automatycznie

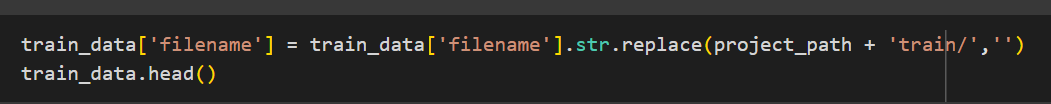
Ramki danych zawierające ścieżki do obrazków są kompatybilne z ramkami zawierającymi etykiety, każdej ścieżce odpowiada jedna etykieta.Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

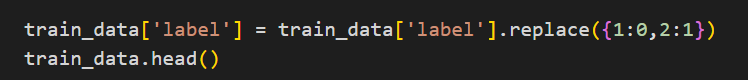
Opis wygenerowany automatycznie

Połączono oba zbiory, w wyniku czego otrzymano ramkę zawierającą ścieżkę oraz etykietę.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Czcionka

Opis wygenerowany automatycznie

W dalszej części przetwarzania, ścieżki dostępu do plików zostały zastąpione nazwami plików. Ponadto dokonano przekształcenia etykiet w taki sposób, że wartości zaczynają się od zera: 0 oznacza klasę pingwina, a 1 klasę żółwia. 



Na zakończenie procesu analizy, wyświetlono sześć losowo wybranych obrazków wraz z przypisanymi etykietami, aby zweryfikować zawartość zbioru danych oraz poprawność przypisania etykiet.

Obraz zawierający tekst, zrzut ekranu, wyświetlacz, Czcionka

Opis wygenerowany automatycznie



## Wczytanie danych do tensora

Zaimplementowano transformacje, które dostosowują wczytywane obrazy do wymaganego formatu i rozmiaru. Ponadto, transformacje te zapewniają odpowiednią normalizację, dostosowaną do standardów używanych w bazie ImageNet.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Kolejno wczytano obrazy oraz etykiety ze zbioru treningowego do struktury tensorów, a następnie sprawdzono ich rozmiary, aby potwierdzić poprawność danych. Po weryfikacji, oba tensory zostały połączone w jeden dataset.Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Obraz zawierający tekst, Czcionka, zrzut ekranu, typografia

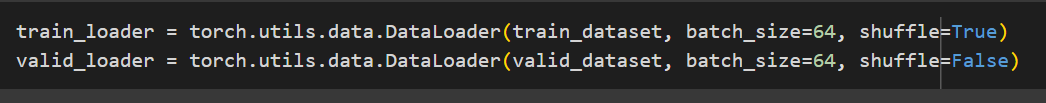
Opis wygenerowany automatycznie

Analogiczne działania wykonano również na zbiorze testowym.

Obraz zawierający tekst, Czcionka, zrzut ekranu, typografia

Opis wygenerowany automatycznie

Stworzono obiekty typu DataLoader z wielkością batch-a równą 64.



## Funkcje pomocnicze do treningu i walidacji modelu

Zaimplementowano funkcję treningową, która co pięć epok wyświetla wartość funkcji straty dla danych treningowych oraz testowych, a także accuracy dla danych testowych:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, System operacyjny

Opis wygenerowany automatycznie

Kolejno zaimplementowano funkcję, która umożliwia sprawdzenie dokładności (accuracy) modelu zarówno dla zbioru treningowego, jak i testowegoObraz zawierający tekst, elektronika, zrzut ekranu, wyświetlacz

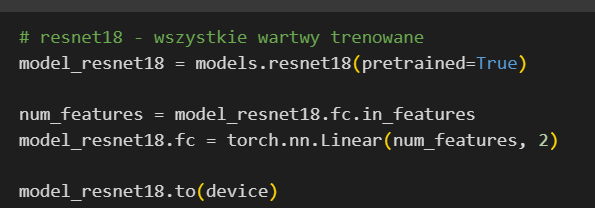
Opis wygenerowany automatycznie

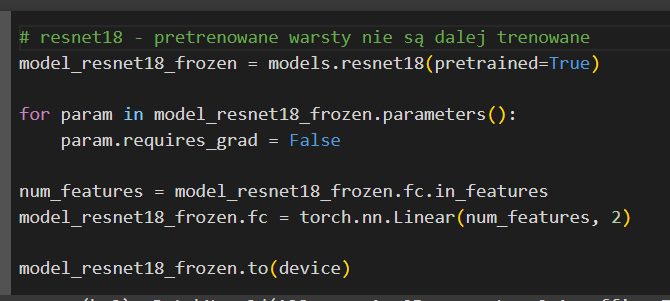
Na koniec zaimplementowano funkcję umożliwiającą wizualną walidację poprawności modelu. Funkcja ta wyświetla dziesięć losowo wybranych obrazków ze zbioru testowego wraz z ich rzeczywistymi i przewidywanymi etykietami.Obraz zawierający tekst, zrzut ekranu, oprogramowanie, wyświetlacz

Opis wygenerowany automatycznie

## Trening i walidacja

Wczytano wcześniej wytrenowane modele ResNet18 i VGG16. Dla każdego z modeli stworzono dwie wersje: pierwsza, 'frozen', gdzie pretrenowane warstwy nie są dalej trenowane, oraz druga, gdzie wszystkie warstwy są aktywne w procesie uczenia. Dodatkowo dostosowano liczbę wyjść w ostatniej warstwie, aby była zgodna z liczbą klas w zbiorze danych.

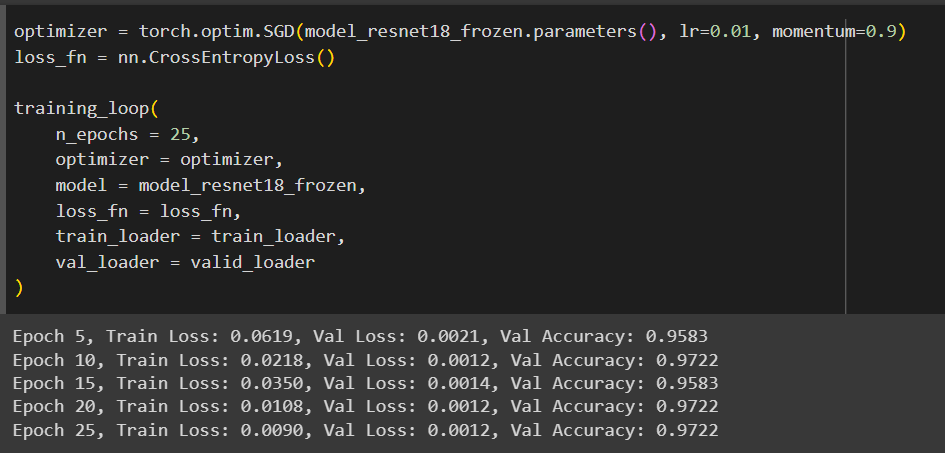


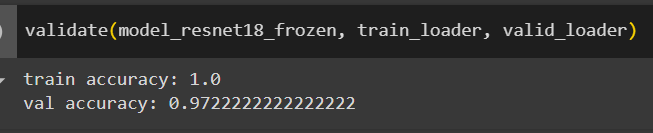


Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Czcionka

Opis wygenerowany automatycznie

Najpierw przeprowadzono trening modelu ResNet18 z zamrożonymi pretrenowanymi warstwami. Jako funkcję straty zastosowano entropię krzyżową, a jako optymalizator wybrano SGD learning rate ustawionym na 0,01 i momentum na 0,9. Po 25 epokach treningu osiągnięto dokładność (accuracy) na poziomie 97% na zbiorze testowym. 



Obraz zawierający tekst, Czcionka, zrzut ekranu

Opis wygenerowany automatycznie

Obraz zawierający ptak wodny, pingwin, ptak, tekst

Opis wygenerowany automatycznie

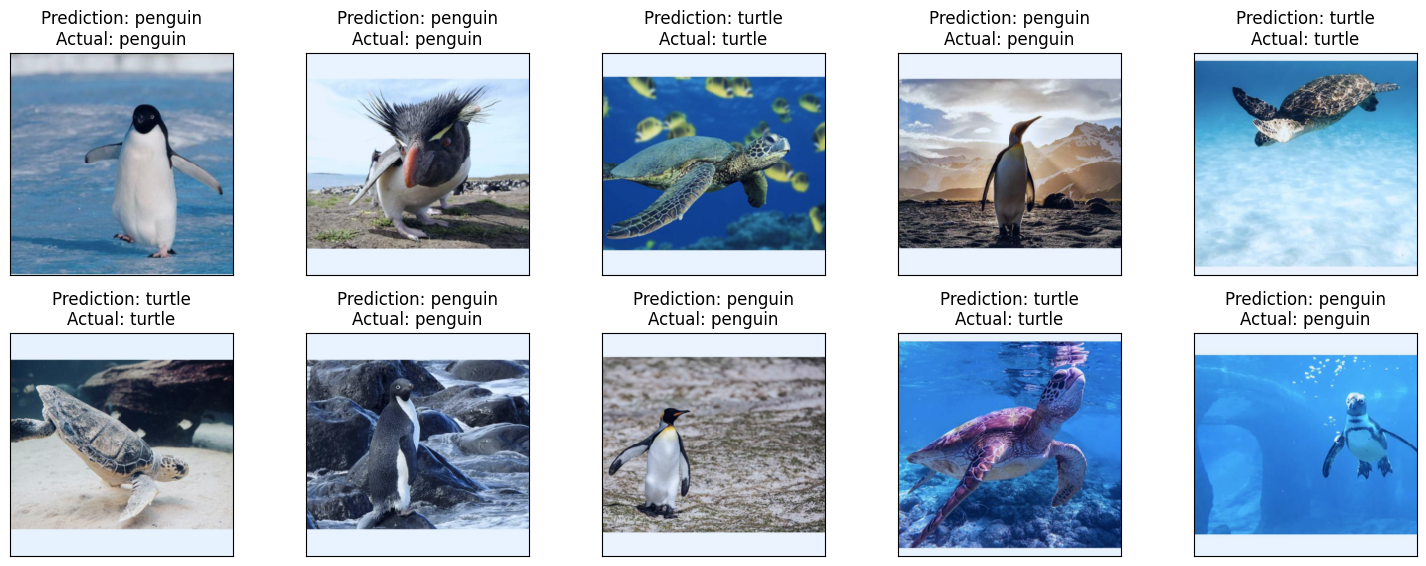
Zwiększenie liczby epok treningowych nie przyniosło poprawy dokładności modelu. Zmniejszenie wartości learning rate również nie wpłynęło na wyniki, podobnie jak jego zwiększenie. Zmiana optymalizatora na Adam także nie przyniosła poprawy.

Następnie przeprowadzono trening modelu ResNet18 w wersji, gdzie wszystkie warstwy były aktywne. Po 25 epokach udało się osiągnąć dokładność na poziomie 98,6

%. Modyfikacja parametrów nie pozwoliła osiągnąć wyższego wyniku, wręcz przyniosła wynik odwrotny. Zmiana optymalizatora na Adam zaowocowała początkową dokładnością na poziomie 50% po 25 epokach, a maksymalnie osiągniętą dokładnością podczas dalszego treningu wyniosła 75%.Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Czcionka

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

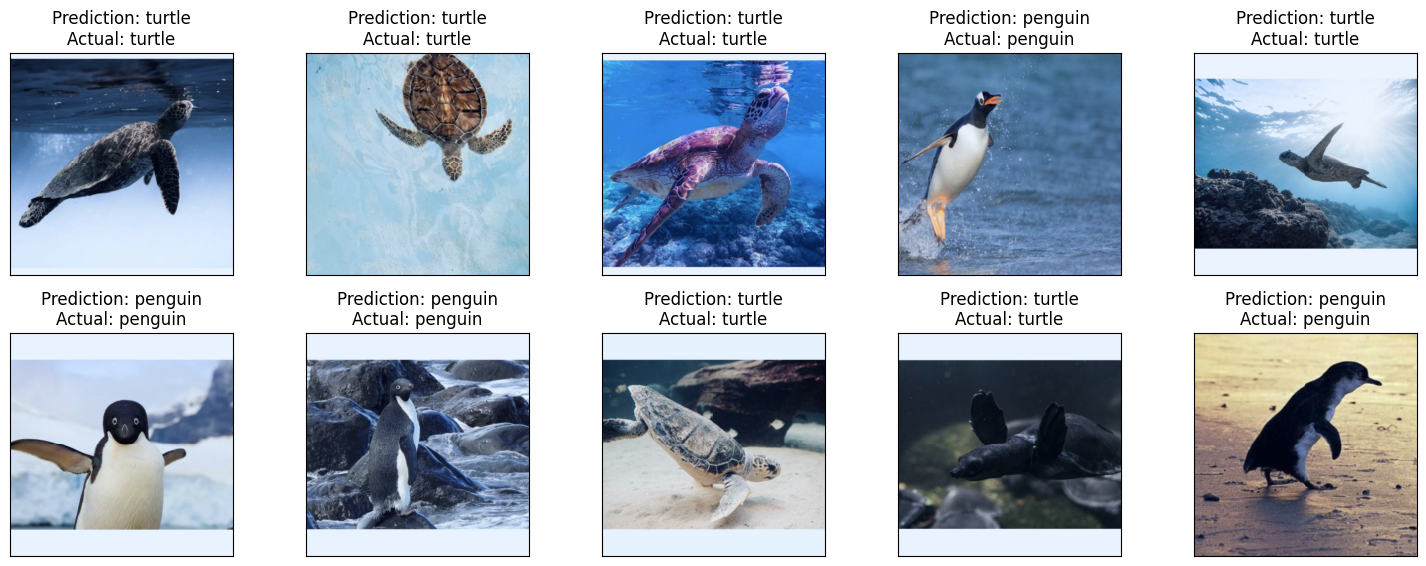
Model VGG16 z zamrożonymi pretrenowanymi warstwami osiągnął dokładność na poziomie 98,6% na zbiorze testowym po 25 epokach treningu. Dalsze modyfikacje hiperparametrów oraz zmiana optymalizatora nie przyniosły wyższego wyniku.

Obraz zawierający tekst, elektronika, zrzut ekranu, oprogramowanie

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

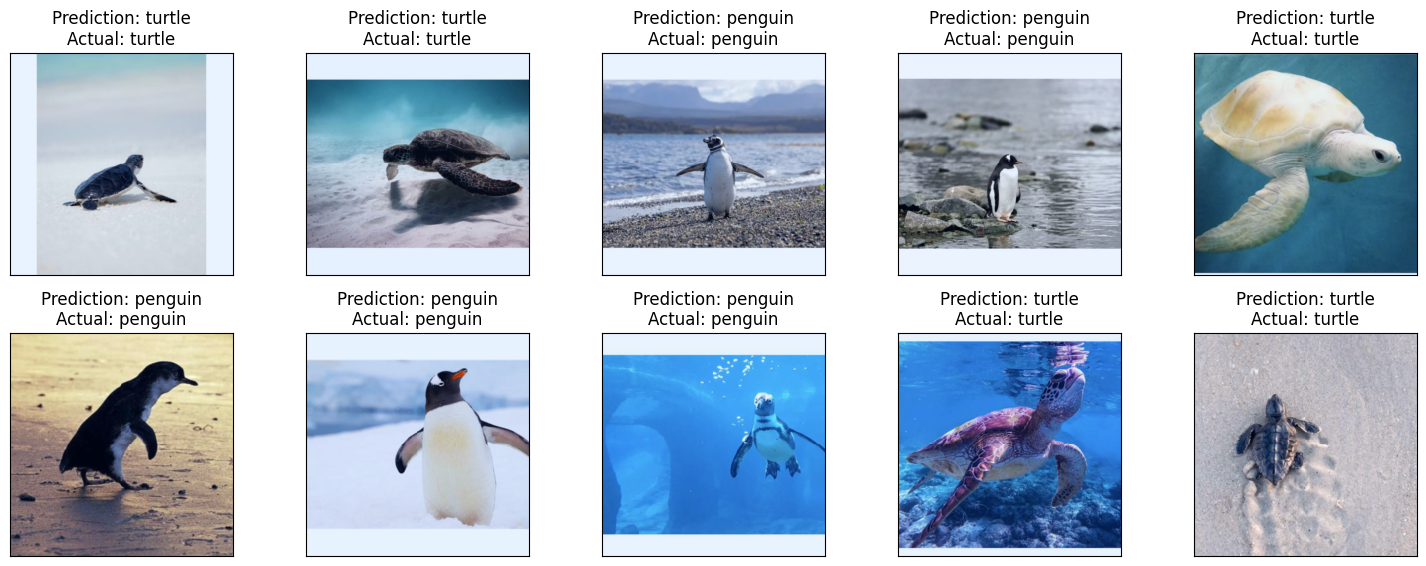


Trenowanie modelu VGG16 z aktywnymi wszystkimi warstwami było bardzo zasobożerne i trwało dłużej niż w przypadku pozostałych metod. Po 25 epokach udało się osiągnąć accuracy na poziomie 93%, co jest najniższym wynikiem wśród testowanych konfiguracji. Wartość straty dla zbioru treningowego była natomiast bliska zeru, co może sugerować że model uległ przeuczeniu.Obraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie



Ze względu na wysokie koszty procesu oraz osiągnięcie lepszych wyników w krótszym czasie w innych modelach, zdecydowano o zaniechaniu dalszego treningu modelu VGG16 z innymi konfiguracjami hiperparametrów.

## Front z użyciem biblioteki gradio

import gradio as gr  
import torch  
from torchvision import transforms, models  
from PIL import Image  
import numpy as np  
  
  
# Funkcja do przetwarzania obrazu przed podaniem go do modelu  
def preprocess\_image(image):  
 if isinstance(image, np.ndarray):  
 image = Image.fromarray(image.astype('uint8'), 'RGB')  
 # Konwersja obrazu PIL na tensor i normalizacja  
 preprocess = transforms.Compose([  
 transforms.Resize(256),  
 transforms.CenterCrop(224),  
 transforms.ToTensor(),  
 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),  
 ])  
 input\_tensor = preprocess(image)  
 # Dodanie wymiaru wsadowego  
 input\_batch = input\_tensor.unsqueeze(0)  
 return input\_batch  
  
  
# Funkcja do klasyfikacji obrazu za pomocą modelu  
def classify\_image(image):  
 # Przetworzenie obrazu  
 input\_tensor = preprocess\_image(image)  
 # Sprawdzenie dostępności urządzenia GPU  
 device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
  
 # Zdefiniowanie i załadowanie modelu ResNet18  
 model\_resnet18 = models.resnet18(weights=None)  
 model\_resnet18.fc = torch.nn.Linear(model\_resnet18.fc.in\_features,  
 2) # Zakładamy, że model ma 2 klasy (Pingwin i Żółw)  
 checkpoint\_resnet18 = torch.load('model\_resnet18.pth', map\_location=device)  
 model\_resnet18.load\_state\_dict(checkpoint\_resnet18['model\_state\_dict'])  
 model\_resnet18.to(device)  
 model\_resnet18.eval()  
  
 # Zdefiniowanie i załadowanie modelu VGG16  
 model\_vgg16 = models.vgg16(weights=None)  
 num\_features = model\_vgg16.classifier[6].in\_features  
 model\_vgg16.classifier[6] = torch.nn.Linear(num\_features, 2) # Zakładamy, że model ma 2 klasy (Pingwin i Żółw)  
 checkpoint\_vgg16 = torch.load('model\_vgg16.pth', map\_location=device)  
 model\_vgg16.load\_state\_dict(checkpoint\_vgg16['model\_state\_dict'])  
 model\_vgg16.to(device)  
 model\_vgg16.eval()  
  
 # Klasyfikacja obrazu za pomocą modelu ResNet18  
 with torch.no\_grad():  
 input\_tensor = input\_tensor.to(device)  
 output\_resnet18 = model\_resnet18(input\_tensor)  
 \_, predicted\_resnet18 = torch.max(output\_resnet18, 1)  
  
 # Klasyfikacja obrazu za pomocą modelu VGG16  
 with torch.no\_grad():  
 input\_tensor = input\_tensor.to(device)  
 output\_vgg16 = model\_vgg16(input\_tensor)  
 \_, predicted\_vgg16 = torch.max(output\_vgg16, 1)  
  
 # Zwracanie predykcji z obu modeli  
 prediction\_resnet18 = "Penguin" if predicted\_resnet18.item() == 0 else "Turtle"  
 prediction\_vgg16 = "Penguin" if predicted\_vgg16.item() == 0 else "Turtle"  
 return prediction\_resnet18, prediction\_vgg16  
  
  
# Interfejs Gradio  
input\_component = gr.components.Image(label="Wczytaj obraz")  
output\_component\_resnet18 = gr.components.Label(label="Predykcja z ResNet18")  
output\_component\_vgg16 = gr.components.Label(label="Predykcja z VGG16")  
  
gr.Interface(fn=classify\_image, inputs=input\_component, outputs=[output\_component\_resnet18, output\_component\_vgg16]).launch(server\_name="0.0.0.0", server\_port=8081)

Do stworzenia graficznego interfejsu użytkownika wykorzystano bibliotekę gradio. Załadowane zostały przygotowane wcześniej modele ResNet18 i VGG16. Użytkownik po uruchomieniu aplikacji ma możliwość załadować plik z komputera. Obraz po wstępnym przetworzeniu zostaje sklasyfikowany przez oba modele, a następnie wyświetlone są wyniki klasyfikacji: „Penguin” lub „Turtle”, z podpisem od którego modelu otrzymano odpowiedź.

### Wygląd aplikacji

Obraz zawierający pingwin, ptak, Oprogramowanie multimedialne, tekst

Opis wygenerowany automatycznie

## Uruchomienie programu przez Docker

Utworzony został następujący Dockerfile:

FROM python:3.9  
  
WORKDIR /pythonProject  
  
COPY requirements.txt /pythonProject/  
RUN pip install --no-cache-dir -r requirements.txt  
  
COPY . /pythonProject  
  
EXPOSE 8081  
  
CMD ["python", "main.py","-e", "production"]  
ENTRYPOINT ["python", "main.py", "-e", "production"]

Plik requirements.txt:

gradio  
torch  
torchvision  
Pillow  
numpy

Po uruchomieniu i wykonaniu następujących komend  
Buildowanie obrazu:

docker build -t my\_image\_name .

Uruchomienie kontenera:

docker run -p 8081:8081 my\_image\_name

aplikacja dostępna jest na serwerze localhost:8081.

## CI/CD

Głównym założeniem wykorzystanego CI/CD jest po każdej aktualizacji na repozytorium automatyczne tworzenie obrazu Dockerowego aplikacji, a następnie wysyłanie go na DockerHub, by stamtąd mógł zostać pobrany do chmury.

Zawartość pliku ci-cd.yml:

name: CI/CD  
  
on:  
 push:  
 branches:  
 - main  
  
jobs:  
 build-and-deploy:  
 runs-on: ubuntu-latest  
  
 steps:  
 - name: Checkout repository  
 uses: actions/checkout@v2  
  
 - name: Display Docker credentials  
 run: |  
 echo "DOCKER\_USERNAME: $DOCKER\_USERNAME"  
 echo "DOCKER\_PASSWORD: $DOCKER\_PASSWORD"  
 env:  
 DOCKER\_USERNAME: ${{ secrets.DOCKER\_USERNAME }}  
 DOCKER\_PASSWORD: ${{ secrets.DOCKER\_PASSWORD }}  
  
 - name: Build Docker image  
 run: docker build -t wtumprojekt:latest ./pythonProject  
  
 - name: Log in to Docker Hub  
 uses: docker/login-action@v1  
 with:  
 username: ${{ secrets.DOCKER\_USERNAME }}  
 password: ${{ secrets.DOCKER\_PASSWORD }}  
  
 - name: Push Docker image to Docker Hub  
 run: |  
 docker tag wtumprojekt:latest ${{ secrets.DOCKER\_USERNAME }}/wtumprojekt:latest  
 docker push ${{ secrets.DOCKER\_USERNAME }}/wtumprojekt:latest

Omówienie najważniejszych kroków:  
  
Build Docker image - tworzenie/aktualizowanie obrazu dockerowego.

Log in to Docker Hub – automatyczne logowanie się do Docker Hub za pomocą loginu i hasła zaszyfrowanego w Repository Secrets.

Push Docker image to Docker Hub – wysyłanie utworzonego obrazu na Docker Hub.

## Wrzucenie obrazu dockerowego na dowolną chmurę

Użyto chmury railway.app. Obraz jest automatycznie pobierany ze wszystkimi aktualizacjami z repozytorium Docker Hub i dostępny pod wygenerowaną domeną.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Opis wygenerowany automatycznie