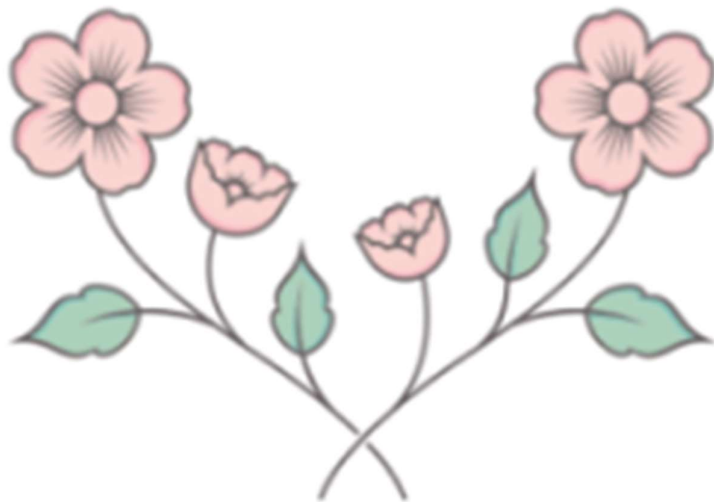


Flower_Match_

DOKUMENTACJA PROJEKTU

Justyna Mątewka

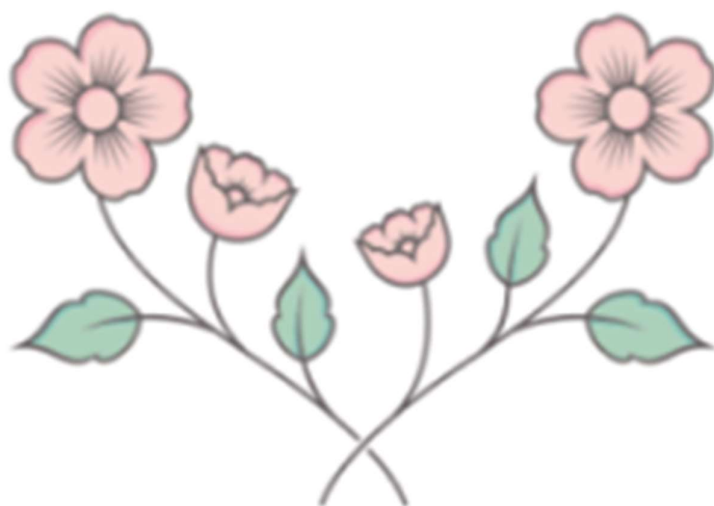


Spis treści

Opis ogólny	3
Wybór tematu	4
Rozpoznawane gatunki kwiatów	4
Wykorzystane technologie.....	5
Graf obrazujący plan ogólny wykorzystany do wykonania projektu.....	6
Przygotowanie danych i modelu	7
Baza danych	8
Colab	8
Przygotowanie środowiska na Colab do obsługi bazy danych i jej import	9
Parametry konfiguracyjne	10
Deklaracja podziału bazy danych na część treningową i walidacyjną przy użyciu TensorFlow	11
Prefetch danych.....	12
Model ResNet-18 – opis teoretyczny	13
Schemat ogólny dowolnego modelu.....	14
ResNet-18	15
Residual Learning	16
Residual Block	17
Analiza parametrów otrzymanych podczas uczenia modelu ResNet-18	18
Otrzymany rozkład prawdopodobieństwa i błędów na poszczególnych epokach podczas trenowania modelu ResNet-18	19
Wykresy wygenerowane na podstawie historii danych zebranych podczas treningu modelu.....	20
ONNX	22
Zapis modelu i konwersja do formatu ONNX.....	23
Strona internetowa	24
Uruchomienie	25
Wygląd	25
API	29
Struktura aplikacji	31
JSON w HTML.....	31

JSON w Python.....	32
Obróbka otrzymanego obrazu.....	32
Detekcja gatunku kwiatu	33
Adnotacje końcowe	34
Przykładowe źródła.....	34

Opis ogólny



Wybór tematu

Projekt "Flower_Match" ma ambitny cel - rozpoznawanie różnych gatunków kwiatów za pomocą zaawansowanych technik uczenia maszynowego. Pomysł na projekt narodził się, gdy sama chciałam skorzystać z tego typu aplikacji i okazało się, że wszystkie są płatne lub słabej jakości. To dało mi bardzo dużo motywacji do samodzielnego zgłębienia tematu i stworzenia aplikacji, którą (pewnie jeszcze po wielu udoskonaleniach) wykorzystam ja i moje koleżanki :)

Rozpoznawanie kwiatów jest niezwykle interesującym i zarazem skomplikowanym tematem. Na całym świecie istnieje wiele różnych gatunków kwiatów, a ich różnorodność i złożoność sprawiają, że ich odróżnienie może być trudne nawet dla doświadczonych botaników.

Rozpoznawane gatunki kwiatów

Wytrenowany przeze mnie model ResNet-18 rozpoznaje 4 gatunki kwiatów:

1. Dzwonki



2. Tulipany



3. Róże



4. Stokrotki



Wykorzystane technologie

Python 3.12 + biblioteki: glob, pandas, numPy, seaborn, matplotlib.pyplot, tfzonnx, onnxruntime, fastapi 0.111.0, opencv-python (cv2), jinja2 2.11.5

TensorFlow 2.0 + biblioteki: keras (models, layers, losses, metrics, optimizers, callbacks, regularizers), train

Colab – środowisko programistyczne

Kaggle - <https://www.kaggle.com/>

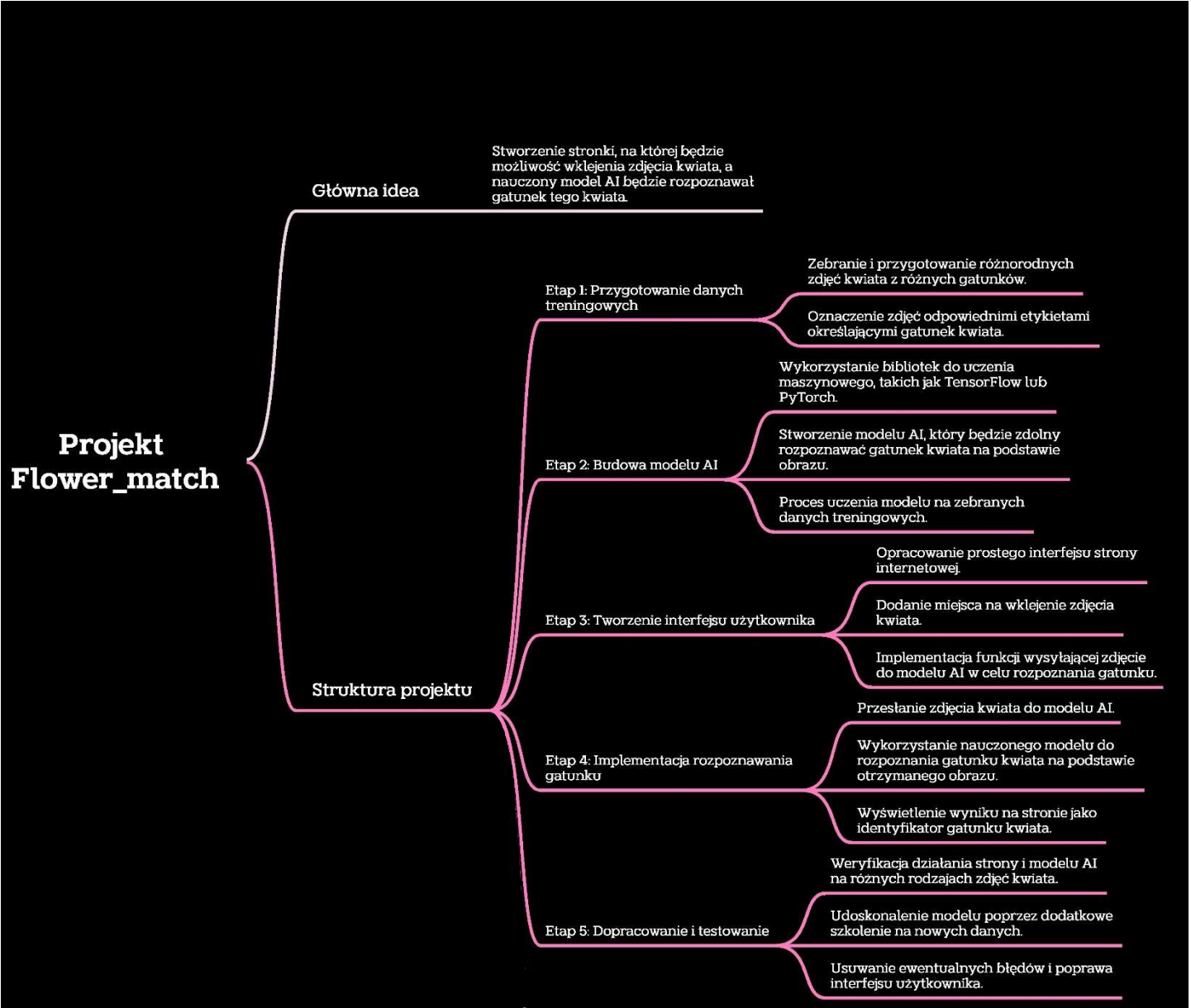
ONNX – format modelu używany na stronie internetowej

Anaconda3 – obsługa TensorFlow

FastAPI – API z stroną WWW połączoną z modelem

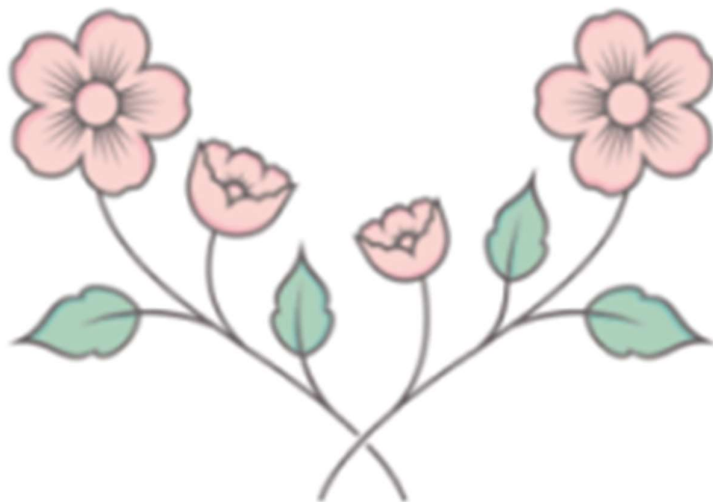
Jinja2 – umożliwia korzystanie z HTMX i CSS w FastAPI

Graf obrazujący plan ogólny wykorzystany do wykonania projektu



Rysunek 1 Diagram projektu

Przygotowanie danych i modelu



Baza danych

Projekt opiera się na bazie danych zawierającej przykładowe zdjęcia 4 gatunków kwiatów: dzwonków, róż, tulipanów i stokrotek, stworzonej przez "L3LLFF". Całkowita ilość wykorzystanych obrazów w 4 folderach to 4054.



Rysunek 2 Logo Kaggle

Link bazy na Kaggle: <https://www.kaggle.com/datasets/l3llff/flowers/data>

Colab

Darmowe środowisko programistyczne połączone z usługami Google, udostępniające ponad 100GB przestrzeni dyskowej, posiadające zainstalowane i skonfigurowane narzędzia do wykonywania projektów m. in. związanych z uczeniem maszynowym, tj.: Anaconda, Python, TensorFlow. Środowisko działa na systemie Linux.

Link do strony -> <https://colab.research.google.com/>

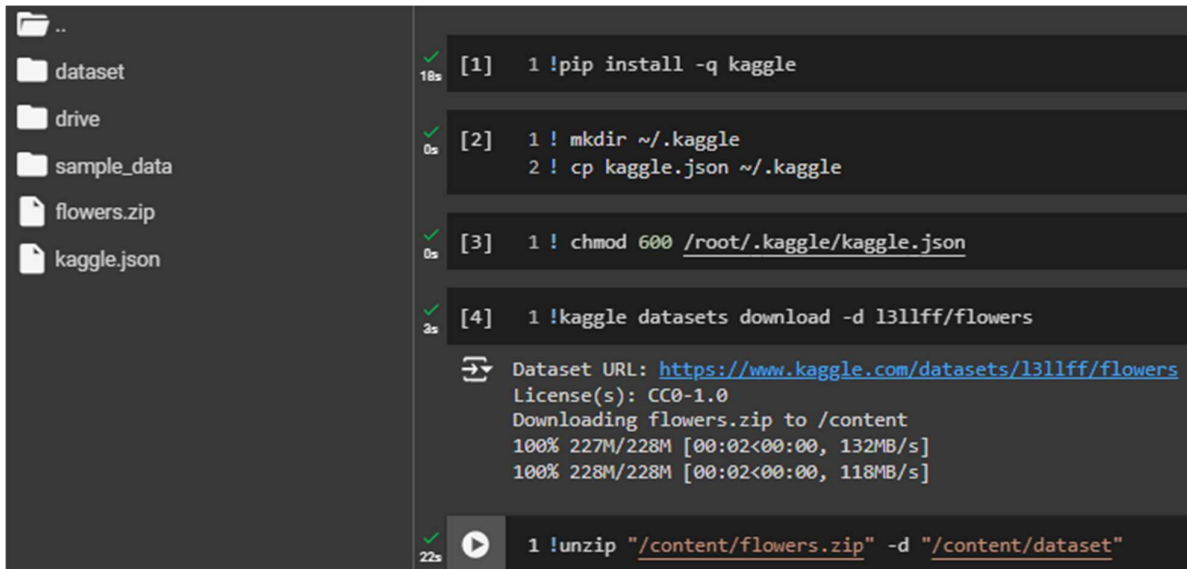


Rysunek 3 Logo Jupyter Notebook i Colab

Niestety dużym mankamentem Colab jest konieczność każdorazowego wykonania, wszystkich poleceń oraz transfer potrzebnych plików po rozłączeniu się środowiska wykonawczego. Skutkowało to niejednokrotnie koniecznością kilkukrotnego wywoływania tych samych poleceń. Po wytrenowaniu modelu w sposób satysfakcjonujący korzystanie z tego środowiska przestało być konieczne, a sam model zapisany w formacie onnx został podłączony do API.

Przygotowanie środowiska na Colab do obsługi bazy danych i jej import

Polecenia systemowe tj.: pobieranie i operacje na katalogach wykonywane są w składni Linuxowej z poprzedzającym je '!':



```
[1] 1 !pip install -q kaggle
18s

[2] 1 ! mkdir ~/.kaggle
2 ! cp kaggle.json ~/.kaggle
0s

[3] 1 ! chmod 600 /root/.kaggle/kaggle.json
0s

[4] 1 !kaggle datasets download -d l3l1ff/flowers
2s

Dataset URL: https://www.kaggle.com/datasets/l3l1ff/flowers
License(s): CC0-1.0
Downloading flowers.zip to /content
100% 227M/228M [00:02<00:00, 132MB/s]
100% 228M/228M [00:02<00:00, 118MB/s]

1 !unzip "/content/flowers.zip" -d "/content/dataset"
22s
```

Rysunek 4 Polecenia do przygotowania środowiska

Połączenie między środowiskiem programistycznym, a bazą danych znajdującą się na Kaggle, umożliwia wygenerowany przeze mnie token -> kaggle.json. Zapewnia on możliwość bezpośredniego pobrania bazy danych na Colab bez konieczności zajmowania fizycznej przestrzeni dyskowej prywatnego laptopa.

Parametry konfiguracyjne

Wartości części parametrów do konfiguracji pozostały domyślne. Moim celem w tym projekcie było sprawdzenie, czy uda mi się stworzyć funkcjonalny model klasyfikujący zdjęcia, więc nie skupiałam się za bardzo na próbach polepszenia wydajności i dokładności. Zastosowałam model, którego osiągi w wersji podstawowej wydawała mi się satysfakcjonująca.

```
1 directory = "/content/dataset/flowers"
2 result = ("/content/dataset/flowers/*")
3 CLASS_NAMES = []
4
5 for x in glob.glob(result):
6     CLASS_NAMES.append(x.split("/")[-1])
7
8 CONFIGURATION = {
9     "BATCH_SIZE": 32,
10    "IMAGE_SIZE": 256,
11    "LEARNING_RATE": 1e-3,
12    "N_EPOCHS": 10,
13    "DROPOUT_RATE": 0.0,
14    "REGULARIZATION_RATE": 0.0,
15    "N_FILTERS": 6,
16    "KERNEL_SIZE": 3,
17    "N_STRIDES": 1,
18    "POOL_SIZE": 2,
19    "N_DENSE_1": 1024,
20    "N_DENSE_2": 128,
21    "NUM_CLASSES": len(CLASS_NAMES),
22    "PATCH_SIZE": 16,
23    "PROJ_DIM": 768,
24    "CLASS_NAMES": CLASS_NAMES,
25 }
```

Rysunek 5 Zbiór parametrów konfiguracyjnych modelu

Za najistotniejsze uznałam:

BATCH_SIZE -> liczba przykładów (obrazów), które są przetwarzane jednocześnie przez model podczas pojedynczej iteracji treningu. Zastosowałam 32, czyli nie dużą ilość, wykorzystałam mniejszą ilość pamięci RAM do trenowania, ale przy tym liczyłam się z spadkiem dokładności. Jest to moim zdaniem najważniejszy parametr w konfiguracji

IMAGE_SIZE -> rozmiar jaki będzie miało każde zdjęcie

NUM_CLASSES -> ilość możliwych odpowiedzi modelu (tu rodzajów kwiatów)

CLASS_NAMES -> nazwy klas = nazwy katalogów w bazie ze zdjęciami rodzajów kwiatów

N_EPOCHS -> liczba (epok) iteracji po zestawie danych treningowych podczas uczenia modelu. Każda epoka składa się z następujących kroków:

1. Model przetwarza całą partię danych treningowych.
2. Obliczany jest błąd predykcji modelu na tej partii danych.
3. Wagi modelu są aktualizowane w celu zminimalizowania błędu.

Deklaracja podziału bazy danych na część treningową i walidacyjną przy użyciu TensorFlow

- Część treningowa - 80% bazy danych:

```
[ ] train_dataset = tf.keras.preprocessing.image_dataset_from_directory(  
    directory,  
    labels='inferred',  
    label_mode='categorical',  
    class_names=CLASS_NAMES,  
    color_mode='rgb',  
    batch_size=CONFIGURATION["BATCH_SIZE"],  
    image_size=(CONFIGURATION["IMAGE_SIZE"], CONFIGURATION["IMAGE_SIZE"]),  
    shuffle=True,  
    seed=99,  
    validation_split=0.2,  
    subset="training",  
    interpolation='bilinear'  
)  
  
Found 115944 files belonging to 299 classes.  
Using 92756 files for training.
```

Rysunek 6 Dataset treningowy

- Część walidacyjna - 20% bazy danych:

```
[9] val_dataset = tf.keras.preprocessing.image_dataset_from_directory(  
    directory,  
    labels='inferred',  
    label_mode='categorical',  
    class_names=CLASS_NAMES,  
    color_mode='rgb',  
    batch_size=CONFIGURATION["BATCH_SIZE"],  
    image_size=(CONFIGURATION["IMAGE_SIZE"], CONFIGURATION["IMAGE_SIZE"]),  
    shuffle=True,  
    seed=99,  
    validation_split=0.2,  
    subset="validation",  
    interpolation='bilinear'  
)  
  
Found 115944 files belonging to 299 classes.  
Using 23188 files for validation.
```

Rysunek 7 Dataset walidacyjny

Opis parametrów:

- `labels='inferred'` - zaznaczenie, że podpisy zdjęć są generowane z struktury katalogów
- `label_mode='categorical'` - rodzaj wyniku jaki dostaną obrazy na wyjściu, opis dokładny poniżej, dostępne są również opcje `'int'`, `'binary'`, `'None'`
- `color_mode='rgb'` - kolorowe inputy (modele z kolorowymi obrazami mają problem z rozpoznawaniem obrazów czarno-białych i odwrotnie)
- `shuffle=True` - „tasuje” elementy
- `seed=99` - zapewnia każdorazowo takie samo tasowanie
- `validation_split=0.2` - podział bazy: 80% część treningowa i 20% część testowa
- `subset="validation" / "training"` - oznaczenie, która to część
- `interpolation='bilinear'` - domyślnie wykorzystywana funkcja interpolacji podczas zmiany rozmiaru obrazów

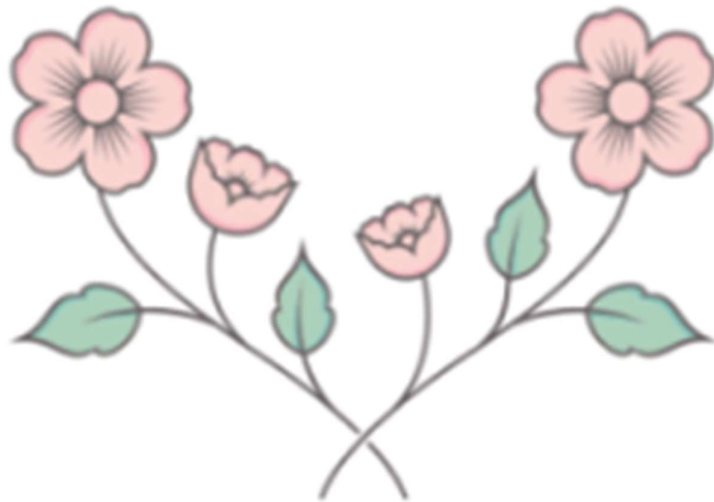
`label_mode='categorical'`:

Każdej z klas (nazw gatunków kwiatów) jest przyporządkowana liczba od 0 do 3 (są 4 gatunki w bazie danych). Parametr `label_mode='categorical'` zastosowany do przykładowego zdjęcia z bazy danych (inputu) zwraca listę zawierającą 4 pola, z czego pole o indeksie reprezentującym nazwę gatunku tego kwiatu jest oznaczone jako `'1'`, a wszystkie pozostałe pola jako `'0'`. Reprezentacja graficzna parametru `label_mode` -> dany obraz przedstawia kwiat z klasy o indeksie 1 - np `'rose'` - `[0, 1, 0, 0]`

Prefetch danych

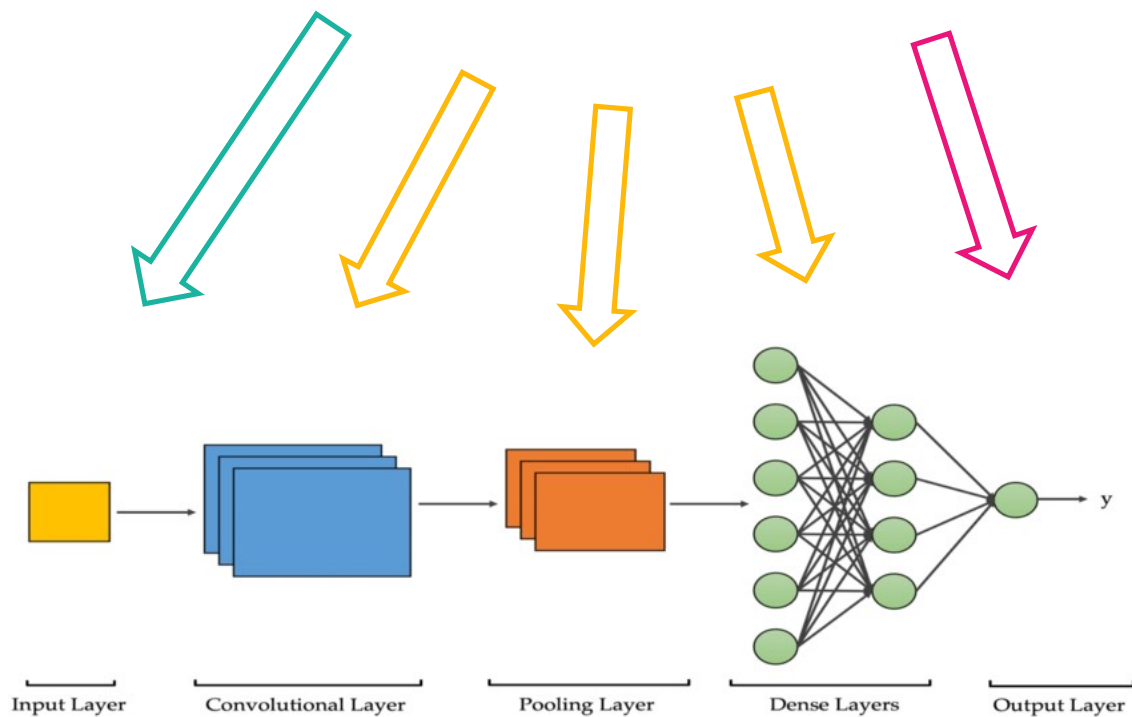
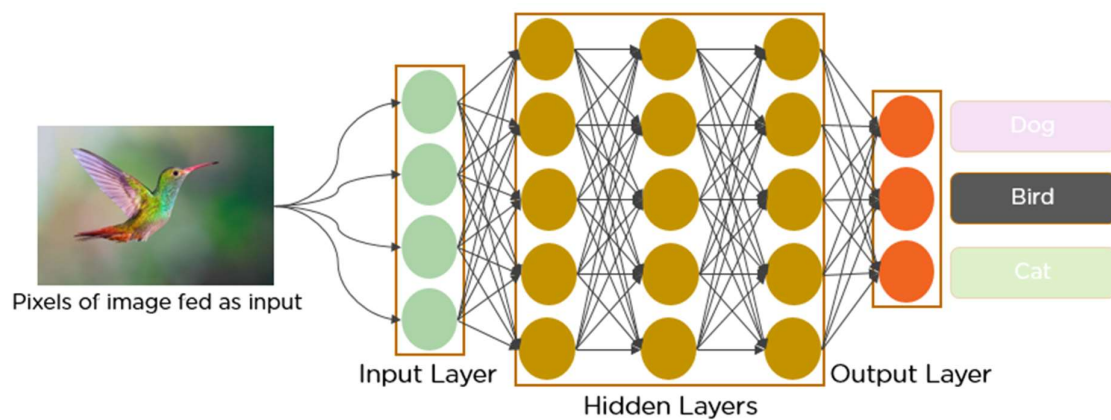
Prefetch danych to technika optymalizacji, która pozwala na asynchroniczne ładowanie następnych partii danych podczas przetwarzania bieżącej partii. Zwiększa to wydajność i zmniejsza opóźnienia podczas treningu modelu

Model ResNet-18 – opis teoretyczny



Schemat ogólny dowolnego modelu

Poniższe rysunki przedstawiają schemat ogólny modelu uczenia maszynowego dla danych wejściowych będących obrazem. W ten sposób chciałam zobrazować ilość obliczeń niezbędnych do wykonania na każdym etapie uczenia modelu (każda czarna linia między warstwami symbolizuje kolejne wyliczenia). W moim modelu występuje 9 warstw filtrów / detekcji cech (Convolutional), 2 warstwy zmniejszania rozmiaru (pooling) i 1 warstwa w pełni połączona (Dense)



ResNet-18

Wybrany modelem do nauczania w tym projekcie jest ResNet w wersji ResNet-18.

Zdecydowałam się na taki wybór po zapoznaniu się z rankingami dla multiclass classification models (przy większej ilości klas - tu gatunków do określenia), obsłudze obrazów rgb oraz prostocie implementacji, przy założeniu, że model jest darmowy.

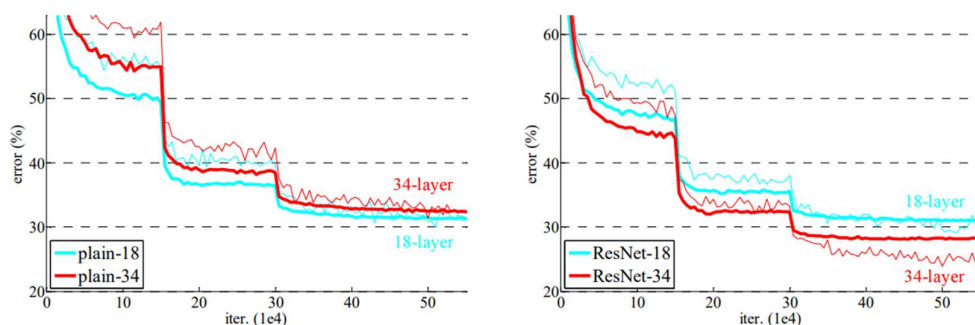
Do implementacji modelu wykorzystałam dane zawarte w oficjalnej dokumentacji modelu ResNet, dla wersji 18-layers (głębokość modelu), szczegóły przedstawia tabela poniżej:

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Tabela 1 Rodzaje modeli ResNet i ich właściwości

Model ResNet zawdzięcza swoje dobre wyniki wykorzystaniu funkcji Residual Learning (uczenie rezydualne). Pozwala ona na pogłębianie modelu bez utraty dokładności i efektu przetrenowania.

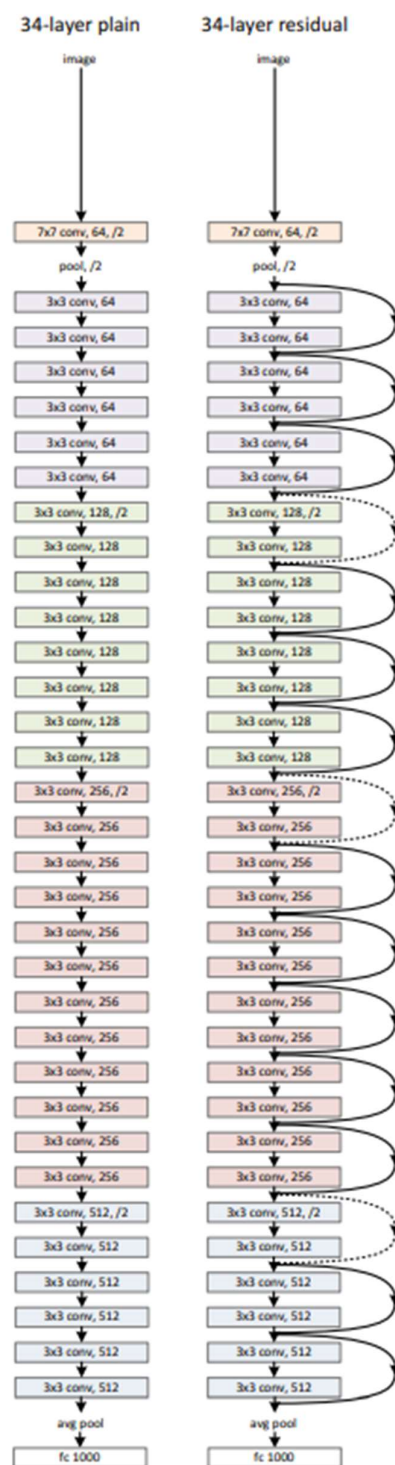
Wykresy poniżej przedstawiają spadek popełnianych błędów przez model po zastosowaniu funkcji Residual Learning:



Rysunek 8 Wykres porównania ilości błędów zwykłego modelu i modelu z użyciem uczenia rezyduального

Residual Learning

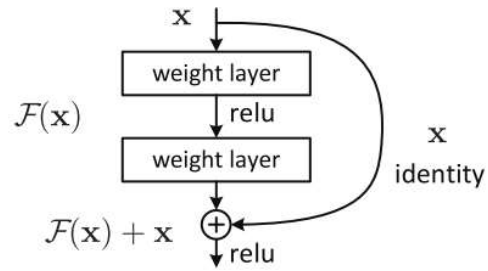
Tradycyjne sieci neuronowe mają problem z przetwarzaniem informacji przez wiele warstw, co prowadzi do przetrenowania – tu zaniku lub eksplozji gradientów (funkcja z czasem zbliża się do wartości 0, żeby ją przywrócić należy podbić tą wartość, a to wiąże się z spadkiem rzeczywistej dokładności, przekłamaniami) – ‘34-layer plain’ na schemacie po prawej. Residual Learning rozwiązuje ten problem, wprowadzając połączenia rezydualne (półokrągłe linie na prawym schemacie) między warstwami. Zamiast uczyć się bezpośrednio odwzorowania między wejściem a wyjściem, sieć uczy się odwzorowania rezydualnego (różnicy między wejściem a wyjściem). Dzięki temu informacja może łatwiej przepływać przez sieć, co pozwala na budowę głębszych modeli bez utraty wydajności.



Rysunek 9 Schemat zwykłego modelu i modelu z użyciem uczenia rezydualnego

Residual Block

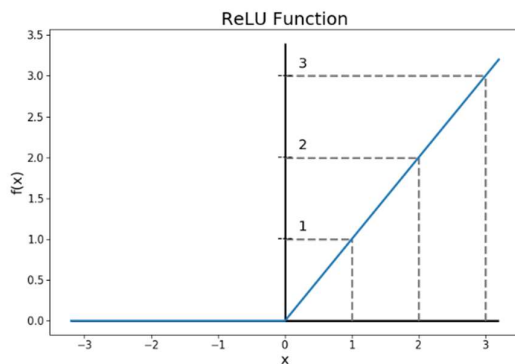
Blok rezydualny to podstawowy element architektury ResNet. Składa się z dwóch warstw konwolucyjnych (weight layer), z których każda jest poprzedzona normalizacją wsadową i aktywacją ReLU. Wyjście drugiej warstwy konwolucyjnej jest dodawane do wejścia bloku



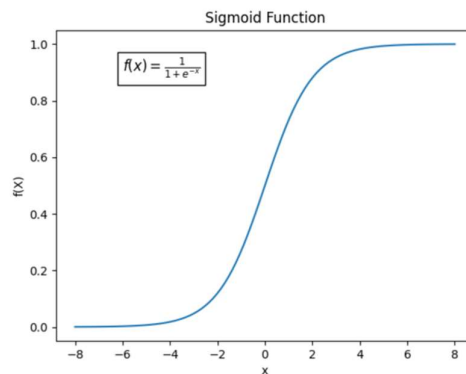
Równanie 1 Blok rezydualny

(połączenie rezydualne). Jeśli wymiary wejścia i wyjścia różnią się, stosowana jest dodatkowa warstwa konwolucyjna do dopasowania wymiarów przed dodaniem (patrz schemat z poprzedniej strony, oznaczone linią przerywaną).

Upraszczając można powiedzieć, że wynikiem modelu ResNet jest rozkład prawdopodobieństwa dla każdej klasy otrzymany przez wywołanie funkcji softmax na wyniku ostatniego bloku rezydualnego (patrz schemat z poprzedniej strony biały prostokąt na dole)



Równanie 2 Funkcja Relu



Równanie 3 Funkcja Sigmoid

Analiza parametrów otrzymanych podczas uczenia modelu ResNet-18



Otrzymany rozkład prawdopodobieństwa i błędów na poszczególnych epokach podczas trenowania modelu ResNet-18

	Część treningowa			Część walidacyjna		
Nr.Epoch	loss	accuracy	top_k_accuracy	loss	accuracy	top_k_accuracy
1.	1.4673	0.5487	0.7939	97.2995	0.2833	0.5821
2.	0.8226	0.6824	0.9067	2.7610	0.4667	0.7090
3.	0.7607	0.7093	0.9170	2.4652	0.4487	0.6667
4.	0.7061	0.7391	0.9260	0.9377	0.7038	0.9346
5.	0.6186	0.7763	0.9365	0.8243	0.7269	0.8910
6.	0.5712	0.7949	0.9433	0.7542	0.7077	0.9051
7.	0.5446	0.8080	0.9494	1.5598	0.6205	0.9372
8.	0.4983	0.8224	0.9484	0.7755	0.7590	0.9269
9.	0.5157	0.8170	0.9462	0.5870	0.7808	0.9513
10.	0.4301	0.8439	0.9587	0.7067	0.7590	0.9205

Agenda:

loss -> dąży do 0.0

accuracy i top_k_accuracy -> dążą do 1.0

top_k_accuracy -> średnia x najlepszych wyników (u mnie x=2)

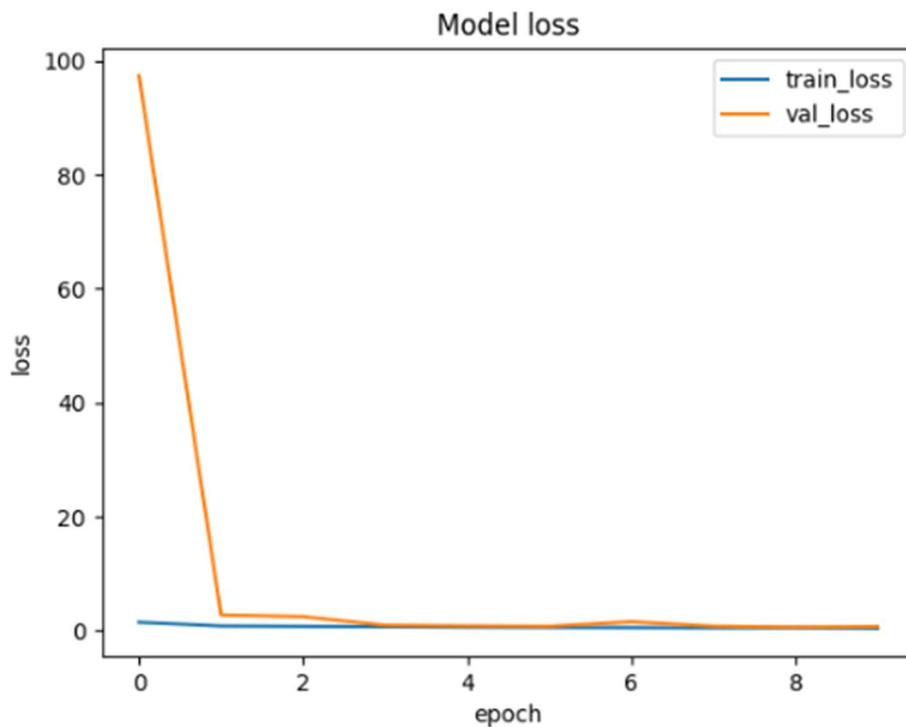
Wartości ostateczne jakie otrzymał mój model ResNet-18

loss: 0.7067	accuracy: 75.9%	top_k_accuracy: 92.05%
--------------	-----------------	------------------------

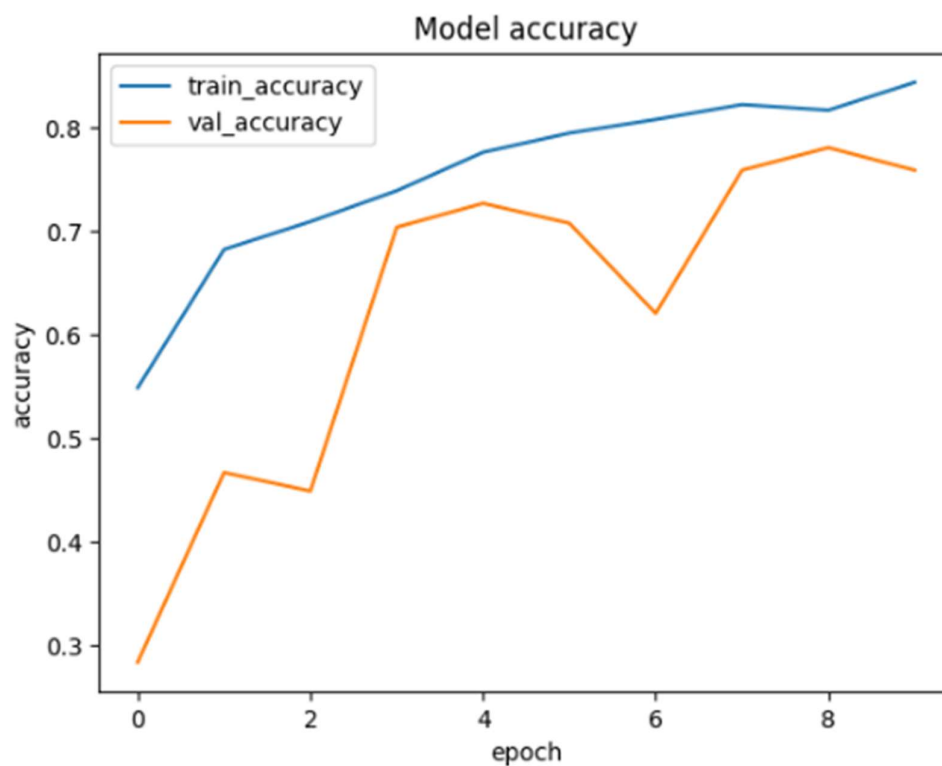
```
[30] 1 resnet_34_model = resnet_34.evaluate(validation_dataset)
25/25 [=====] - 86s 3s/step - loss: 0.7067 - accuracy: 0.7590 - top_k_accuracy: 0.9205
```

Rysunek 10 Zrzut ekranu ze sprawdzeniem dokładności po wytrenowaniu modelu

Wykresy wygenerowane na podstawie historii danych zebranych podczas treningu modelu



Rysunek 11 Wykres zmiany ilości błędów w kolejnych epokach podczas treningu modelu ResNet-18



Rysunek 12 Wykres zmiany dokładności predykcji w kolejnych epokach podczas treningu modelu ResNet-18

Pomimo, że accuracy dla części treningowej jest na bardzo dobrym poziomie – ponad 90% wartość walidacyjna dokładności jest poniżej 80%, więc model można uznać za niedotrenowany, jednak warunki sprzętowe nie pozwoliły mi na poprawę tych wyników.

ONNX



Zapis modelu i konwersja do formatu ONNX

Export modelu z Colab na szczęście był bardzo prosty - polegał na zapisaniu go na rzeczywistym nośniku i jednolinijkowej konwersji do pliku '.onnx'.

```
resnet_18.save('/content/drive/MyDrive/resnet_18_model')  
  
!python -m tf2onnx.convert --saved-model /content/drive/MyDrive/resnet_18_model/ --output resnet_18_model.onnx
```

Rysunek 13 Polecenia do zapisu i konwersji modelu z TensorFlow na format ONNX

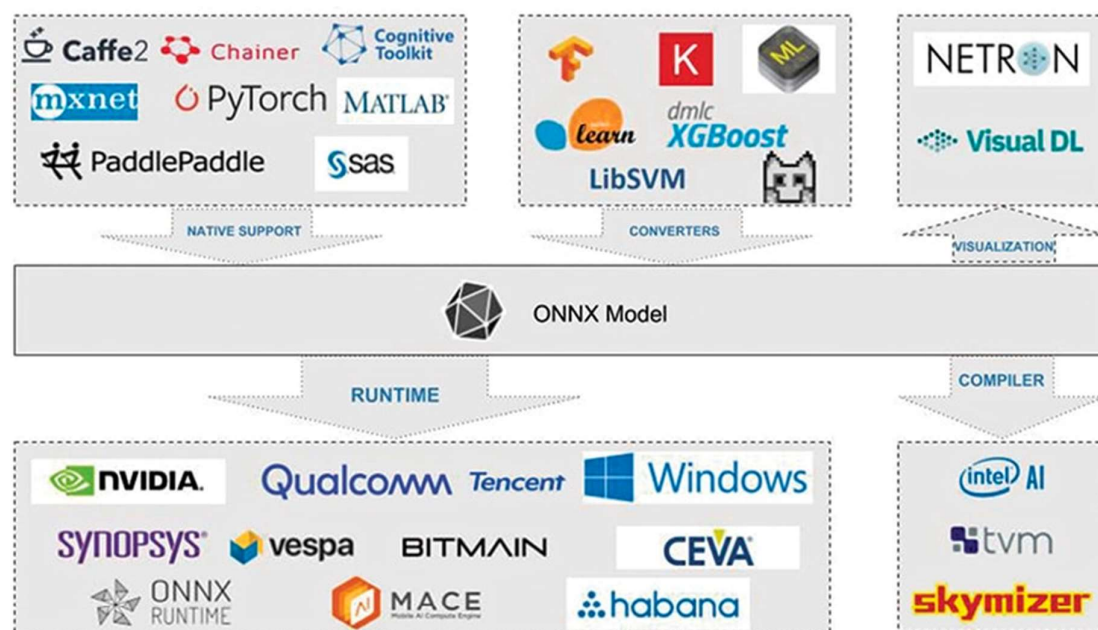
Wielkie wrażenie wywarła na mnie zmiana rozmiaru modelu ResNet-18, a mianowicie po tej operacji z 132KB zmniejszył się on do zaledwie 43KB.

ONNX – Open Neural Network Exchange



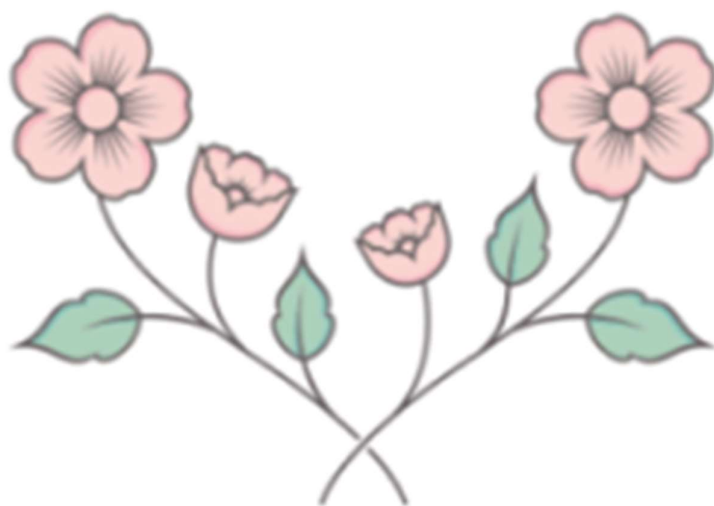
Rysunek 14 Logo ONNX

Format modelu interpretowany przez większość języków używanych do uczenia maszynowego, kompilatorów i frameworków, umożliwia np. wykorzystanie modelu napisanego przy użyciu TensorFlow do modyfikacji w PyTorch. Inną ważną zaletą ONNX jest współpraca z różnymi silnikami jak Nvidia, Windows, ONNX Runtime, co umożliwiło mi w łatwy sposób podpiąć model do strony.



Rysunek 15 Schemat kompatybilności wstecz i w przód modelu ONNX

Strona internetowa



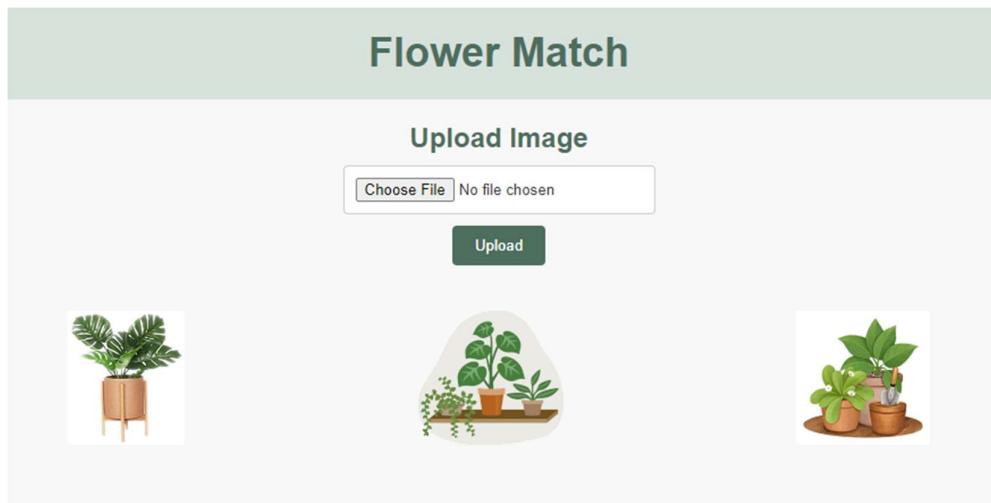
W celu ułatwienia użytkownikowi końcowemu integracji z wytrenowanym przeze mnie modelem stworzyłam stronę internetową. Zawiera ona nazwę projektu, miejsce na załączenie pliku – zdjęcia kwiatu oraz detale estetyczne. Zwracany jest gatunek kwiatu oraz podane przez użytkownika zdjęcie.

Uruchomienie

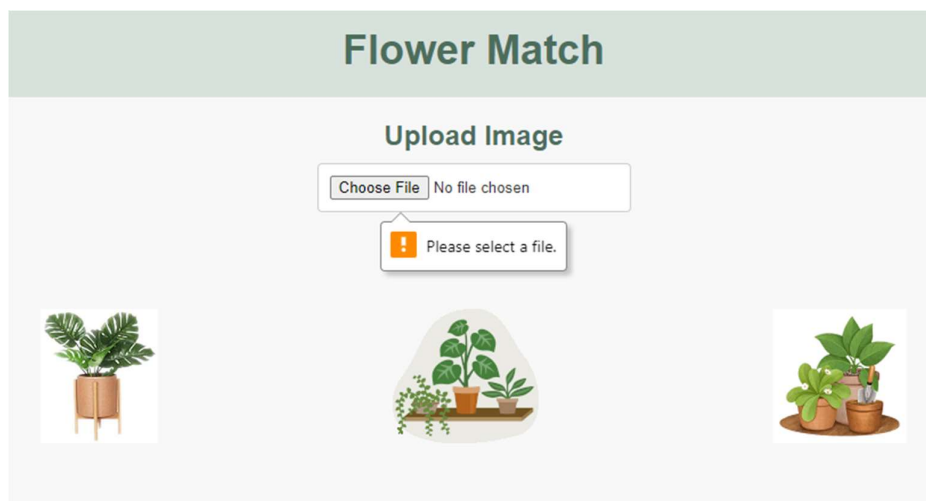
Należy w konsoli wprowadzić polecenie: „**fastapi dev main.py --reload**”, w katalogu, w którym znajduje się plik `main.py`. Następnie po wpisaniu do przeglądarki adresu: <http://127.0.0.1:8000/index/> zostanie wyświetlona strona internetowa.

Wygląd

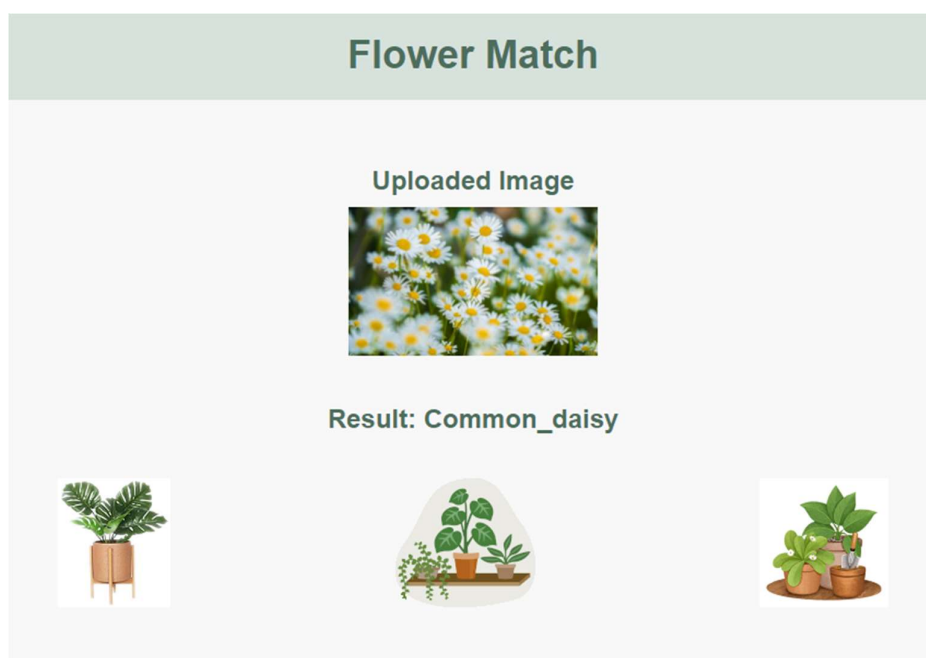
Poniższe zrzuty ekranu przedstawiają wygląd i funkcjonalności przygotowanej strony internetowej



Rysunek 16 Zrzut ekranu interfejsu użytkownika



Rysunek 17 Do uruchomienia jest wymagane zdjęcie



Rysunek 18 Zrzut ekranu poprawnie rozpoznanego gatunku

Flower Match

Uploaded Image



Result: Not recognized



Rysunek 19 Zrzut ekranu nierozpoznanego gatunku

Flower Match

Uploaded Image

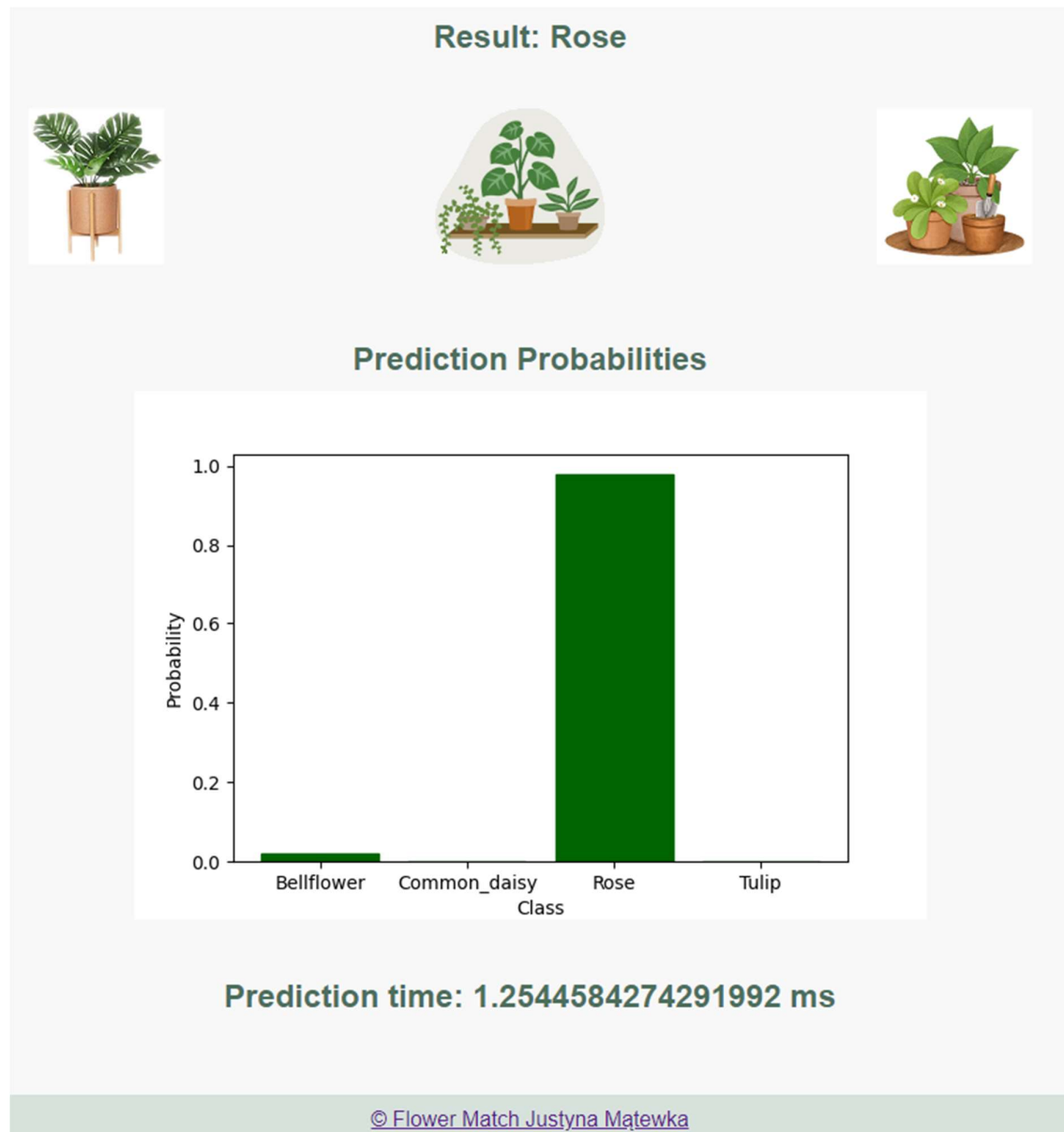


Result: Rose



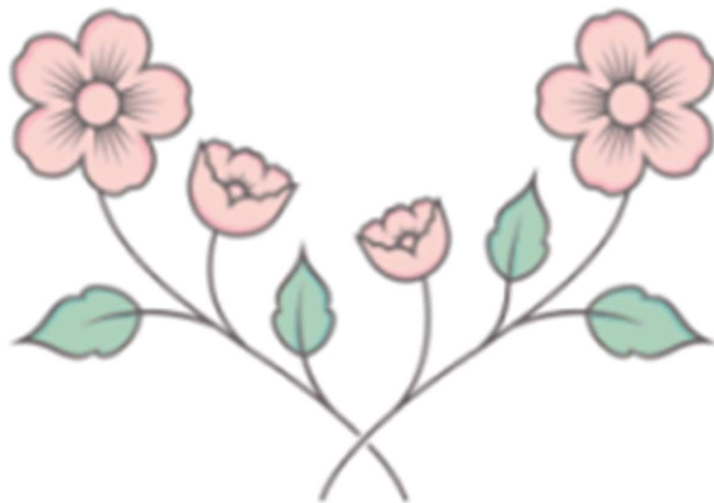
Rysunek 20 Zrzut ekranu błędnie rozpoznanego gatunku

Każda analiza przesłanego zdjęcia jest obrazowana również na prostym wykresie słupkowym przedstawiającym rozkład prawdopodobieństwa między poszczególnymi gatunkami znanymi modelowi oraz czas potrzebny na wykonanie detekcji ('Prediction time' podawany w milisekundach).



Rysunek 21 Analiza danych z modelu

API



Funkcjonalności strony internetowej są realizowane przy użyciu frameworku FastAPI oraz silnika szablonów Jinja2 w połączeniu ze skryptem Pythonowym, gdzie przy użyciu ONNX Runtime wykorzystywany jest wytrenowany przeze mnie model ResNet-18

FastAPI -> nowoczesny, szybki (wysokowydajny) framework webowy do tworzenia interfejsów API w Pythonie z wykorzystaniem wstawek JSON.



Rysunek 22 Logo FastAPI

Jinja2 -> w pełni funkcjonalny silnik szablonów HTTP dla Pythona. Posiada pełną obsługę unikodu, opcjonalne zintegrowane środowisko, jest powszechnie używany z FastAPI.



Rysunek 23 Logo Jinja

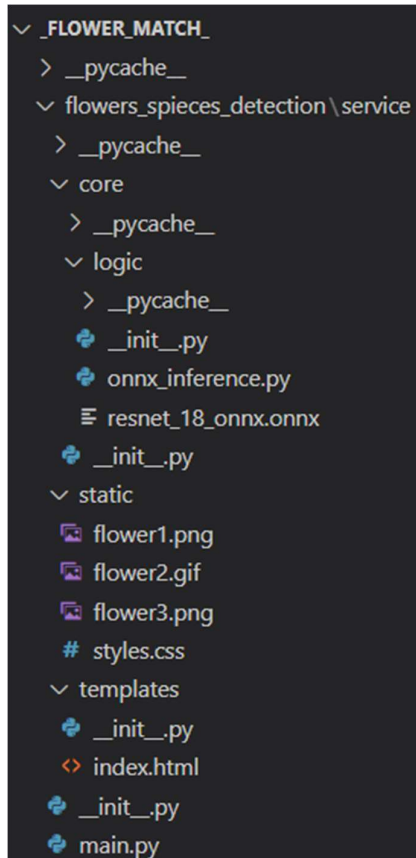
Realizując projekt wykorzystałam tylko najbardziej podstawowe funkcjonalności. Zdecydowałam się na wykorzystanie FastAPI, po opiniach, które porównywały go do następnika DJANGO, w którym pisaliśmy projekt na studia i faktycznie mogę ocenić ten framework jako bardziej przyjemny, a wdrożenie zajęło mi minimalną ilość czasu.

ONNX Runtime -> środowisko uruchomieniowe dla modeli ONNX, które umożliwia efektywne wykonywanie inferencji na różnych platformach sprzętowych i systemach operacyjnych.



Rysunek 24 Logo ONNX Runtime

Struktura aplikacji



Rysunek 25 Struktura katalogów i plików w FastAPI

Struktura katalogów w FastAPI składa się z:

- kat. 'service' - część główna z plikiem 'main.py' zawierającym obsługę rządań POST i GET ze strony internetowej
- kat. 'service/core/logic' - zawierający model 'resnet_18_onnx.onnx' i plik 'onnx_inference.py' z funkcją do testowania podanego obrazu na modelu
- kat. 'service/static' - zawiera zdjęcia będące elementami dekoracyjnymi strony i jej style CSS
- kat. 'service/templates' - zawiera plik z HTML z szablonem mojej strony

Struktura katalogów została z góry narzucona przez framework i jest wymagana w takiej formie do poprawnego działania aplikacji.

JSON w HTML

```
<section class="result">
    {% if result %}
    <h2>Result: {{ result }}</h2>
    {% endif %}
</section>
```

Rysunek 26 Wyświetlenie zmiennej 'result' na stronie WWW

Wymiana parametrów między HTML, a Python jest możliwa dzięki wstawkom w formacie JSON. Fragment kodu po lewej obrazuje takie wywołanie dla zwracanego gatunku kwiatu. Sekcja pojawia się na stronie internetowej tylko w momencie zwrócenia przez 'main.py' zmiennej 'result'.

JSON w Python

Wartości zmiennych z kodu w Python przekazywane są do pliku 'index.html' w postaci słownika o strukturze parametrów JSON.

```
context = {
    "request": request,
    "result": result["emotion"],
    "image": image_base64
}
return templates.TemplateResponse("index.html", context)
```

Rysunek 27 Zwracanie wartości z kodu w Python do kodu w HTML

Obróbka otrzymanego obrazu

Otrzymane od użytkownika zdjęcie jest konwertowane do kolorowego formatu (przydatne, jeżeli otrzymano obraz czarno-biały, bo wtedy brakowałoby jednego z parametrów przy obliczeniach modelu). Następnie zdjęcie jest przekształcane na postać macierzy i w takim formacie przekazywane do funkcji określającej gatunek kwiatu.

Dodatkowo, aby umożliwić wyświetlenie zwrotne obrazu na stronie internetowej musi on zostać przekonwertowany na rozpoznawany przez kod HTML format.

```
@app.post("/detect/", response_class=HTMLResponse)
async def detect(request: Request, flowerImage: UploadFile = File(...)):
    contents = await flowerImage.read()
    image = Image.open(BytesIO(contents)).convert('RGB')
    image_np = np.array(image)

    result = flower_detector(image_np)

    buffered = BytesIO()
    image.save(buffered, format="PNG")
    image_base64 = base64.b64encode(buffered.getvalue()).decode('utf-8')
```

Rysunek 28 Zmiany na otrzymanym obrazie

Detekcja gatunku kwiatu

Sposób rozpoznawania gatunku kwiatu został zaimplementowany w 'flower_detector' wywoływanej na przekazanej macierzy.

Podłączenie modelu przy użyciu ONNX Runtime i deklaracja, że on być wywołany przy użyciu procesora CPU.

```
provider = ['CPUExecutionProvider']
model = rt.InferenceSession(r"resnet_18_onnx.onnx", providers=provider)
```

Rysunek 29 Dołączenie modelu do API

Dalsze zmiany na przekazanym obrazie – zmiana rozmiaru na obsługiwany przez model, zmienia format danych w macierzy na zmiennoprzecinkowy oraz liczbę próbek równą 1. Tak przekazany obraz ma postać: liczba próbek, wysokość, szerokość, kanały (3, bo RGB).

```
test_image = cv2.resize(img_array, (256, 256))
im = np.float32(test_image)
img_array = np.expand_dims(im, axis = 0)
```

Rysunek 30 Dalsze zmiany inputu

Uruchamiany jest model ResNet-18 na liście dostępnych klas (output_names) i słowniku danych wejściowych zawierającym tylko przekazany obraz.

```
output_names = [output.name for output in model.get_outputs()]
onnx_pred = model.run(output_names, {'input_1': img_array})
```

Rysunek 31 Uruchomienie modelu

Słowna deklaracja nazw klas i wyciągnięcie predykcji modelu oraz jej zwrócenie. W wypadku, gdy wartość predykcji jest **mniejsza niż 85%** zwracany jest komunikat „Not recognized”

```
CLASS_NAMES = ['Bellflower', 'Common_daisy', 'Rose', 'Tulip']
# CLASS_NAMES = ['Dzwonek', 'Stokrotka', 'Róża', 'Tulipan']
if max(onnx_pred[0][0]) < 0.85:
    emotion = "Not recognized"
else:
    emotion = CLASS_NAMES[np.argmax(onnx_pred[0][0])]

return {"emotion" : emotion}
```

Rysunek 32 Predykcja modelu

Adnotacje końcowe

1. Czas potrzebny na wytrenowanie modelu okazał się być dłuższy niż przewidywałam i dla 14 gatunków kwiatów na modelu ResNet-34 przy epoch = 20 wynosiłby ponad 48h co przy niestabilnym środowisku Colab było niemożliwym do osiągnięcia. W związku z tym zdecydowałam się na zmniejszenie głębokości modelu do 18 warstw – implementację modelu ResNet-18, dodatkowo zmniejszyłam epoch do 10.
2. Dokładność wytrenowanego modelu jest na poziomie ~76%, co jest zauważalne przy manualnym testowaniu na randomowych zdjęciach kwiatów (spoza wykorzystanej bazy danych), dość często się myli w wynikach

Przykładowe źródła

- https://www.tensorflow.org/api_docs/python/tf/keras
- <https://arxiv.org/pdf/1512.03385.pdf>
- <https://arxiv.org/pdf/1712.09913.pdf>
- <https://fastapi.tiangolo.com/tutorial/>
- <https://www.youtube.com/watch?v=IA3WxTTPXqQ>
- https://youtu.be/i_LwzRVP7bg
- <https://www.tensorflow.org/tutorials/images/cnn?hl=pl>
- <https://mco-mnist-draw-rwpkxka3zaa-ue.a.run.app/>