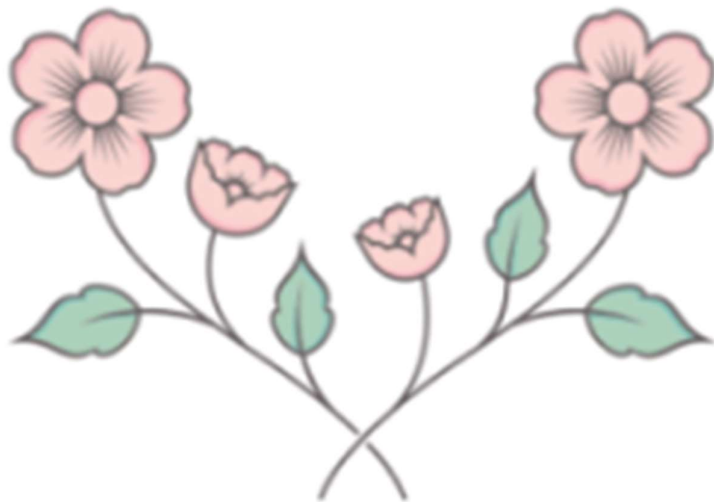


Flower_Match_

DOKUMENTACJA PROJEKTU

Justyna Mątewka



Opis ogólny i 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.

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

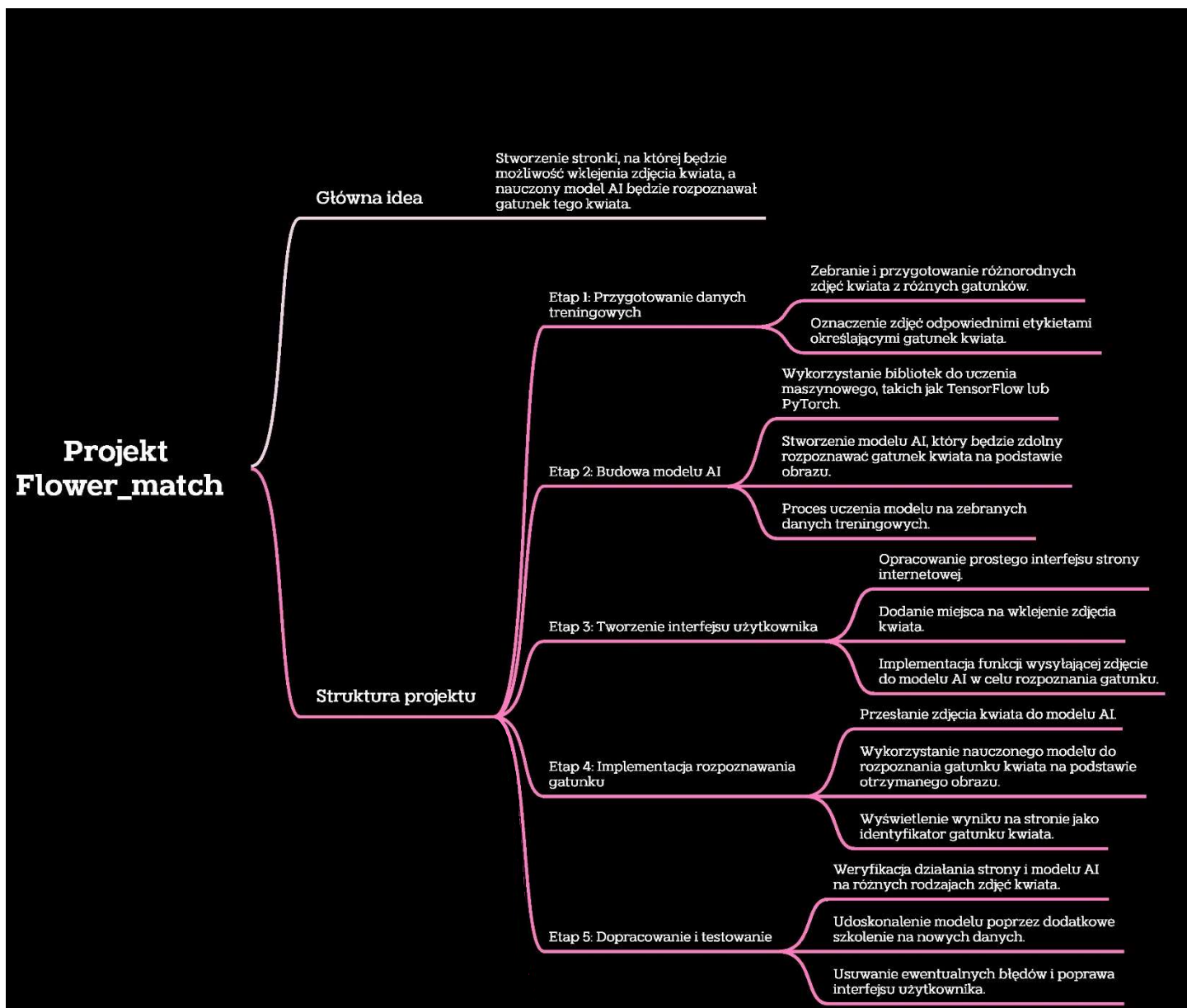
Colab – środowisko programistyczne

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

Anaconda3 – obsługa TensorFlow

FastAPI – API z stroną WWW połączoną z modelem

Graf obrazujący plan ogólny wykorzystany do wykonania projektu:



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.



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.

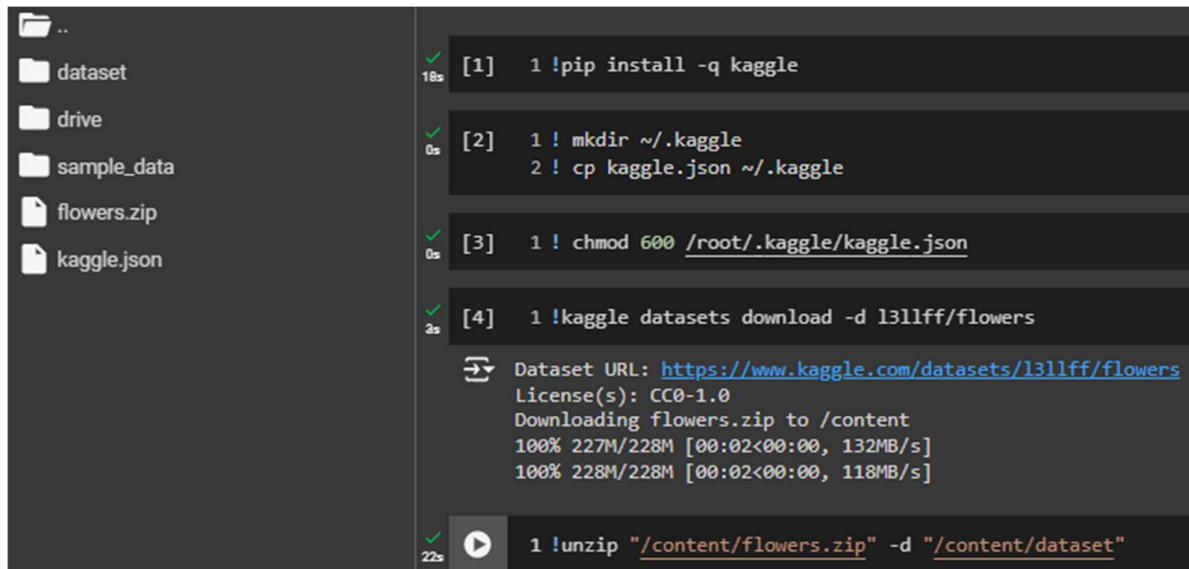
Link do strony -> <https://colab.research.google.com/>



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 '!':



The screenshot displays the Google Colab interface. On the left, a file explorer shows the directory structure: a parent folder '..', and subfolders 'dataset', 'drive', and 'sample_data'. Below these are files 'flowers.zip' and 'kaggle.json'. The main terminal area on the right shows four executed commands, each preceded by a green checkmark and a time indicator. Command [1] installs Kaggle. Command [2] creates a '.kaggle' directory and copies 'kaggle.json' into it. Command [3] sets permissions for 'kaggle.json'. Command [4] downloads the 'l31lff/flowers' dataset. Below the command, the terminal shows the dataset URL, license, and download progress for 'flowers.zip'. The final command is partially visible, showing the unzipping of the downloaded file into the 'dataset' folder.

```
[1] 1 !pip install -q kaggle
[2] 1 ! mkdir ~/.kaggle
    2 ! cp kaggle.json ~/.kaggle
[3] 1 ! chmod 600 /root/.kaggle/kaggle.json
[4] 1 !kaggle datasets download -d l31lff/flowers
Dataset URL: https://www.kaggle.com/datasets/l31lff/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"
```

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 }
```

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": 256 -> 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 po konwersji do formatu wymaganego przez 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.

- 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.

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

ResNet-18:

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

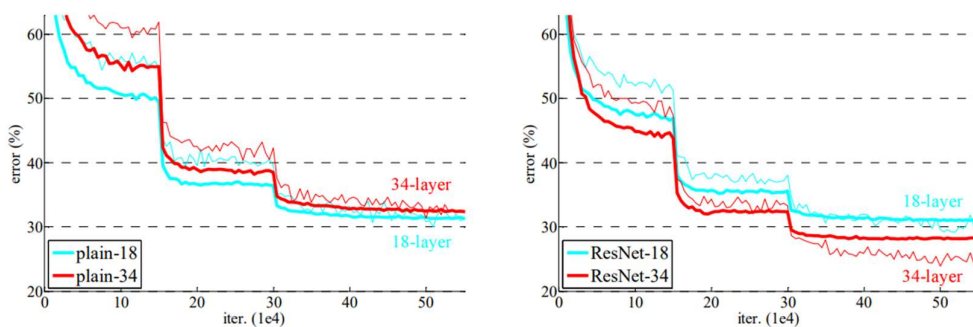
Zdecydowałam się na taki wybór po zapoznaniu się z rankingami dla multiclass classification models (przy większej ilości klas - tu gatunków do określenia), obsłudze obrazów rgb oraz prostocie implementacji, przy założeniu, że model jest darmowy.

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

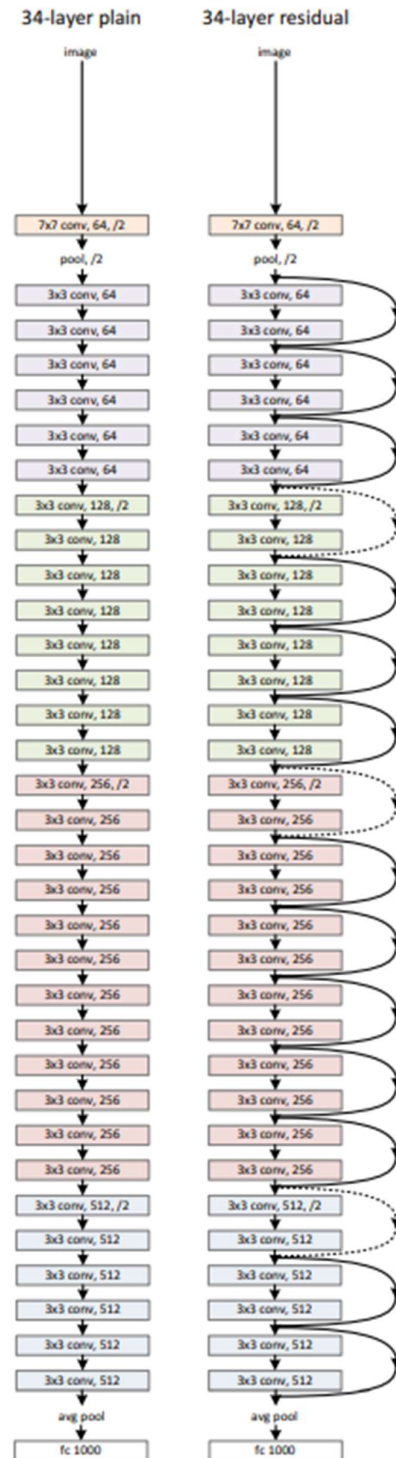
Model ResNet zawdzięcza swoje dobre wyniki wykorzystaniu funkcji Residual Learning (uczenie rezydualne). Pozwala ona na pogłębianie modelu bez utraty dokładności i efektu przetrenowania.

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



Residual Learning:

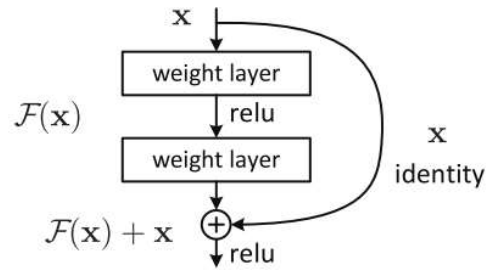
Tradycyjne sieci neuronowe mają problem z przetwarzaniem informacji przez wiele warstw, co prowadzi do przetrenowania – tu zaniku lub eksplozji gradientów (funkcja z czasem zbliża się do wartości 0, żeby ją przywrócić należy podbić tą wartość, a to wiąże się z spadkiem rzeczywistej dokładności) – ‘34-layer plain’ na schemacie. 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.



Residual Block:

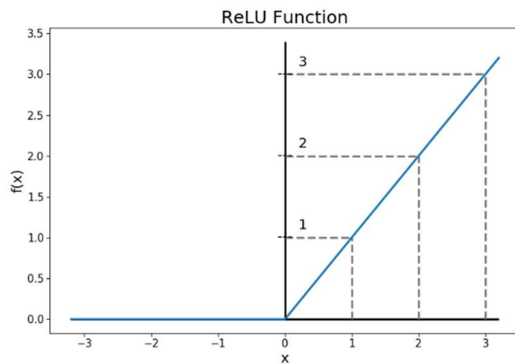
Blok rezydualny to podstawowy element architektury ResNet. Składa się z dwóch warstw konwolucyjnych (weight layer), z których każda jest poprzedzona normalizacją wsadową i aktywacją ReLU. Wyjście drugiej warstwy

konwolucyjnej jest dodawane do wejścia bloku (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).

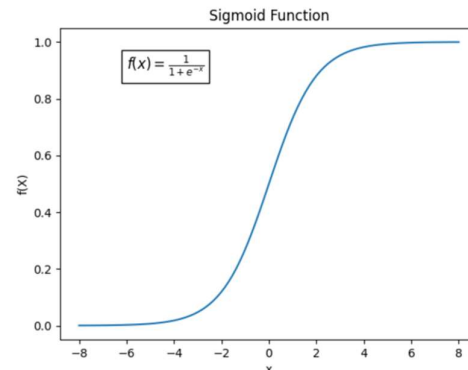


Upraszczając można powiedzieć, że wynikiem modelu ResNet jest rozkład prawdopodobieństwa dla każdej klasy otrzymany przez wywołanie funkcji softmax na wyniku ostatniego bloku rezydualnego (patrz schemat z poprzedniej strony biały prostokąt na dole)

Funkcja ReLU:



Funkcja Sigmoid:



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

Nr.Epoch	Część treningowa			Część walidacyjna		
	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:

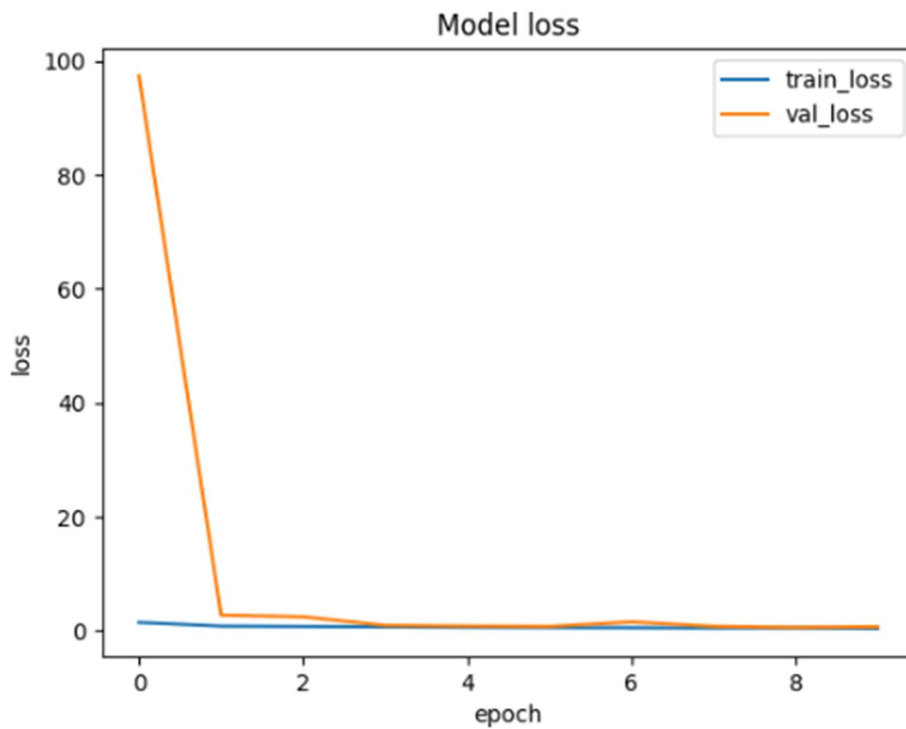
loss: 0.7067

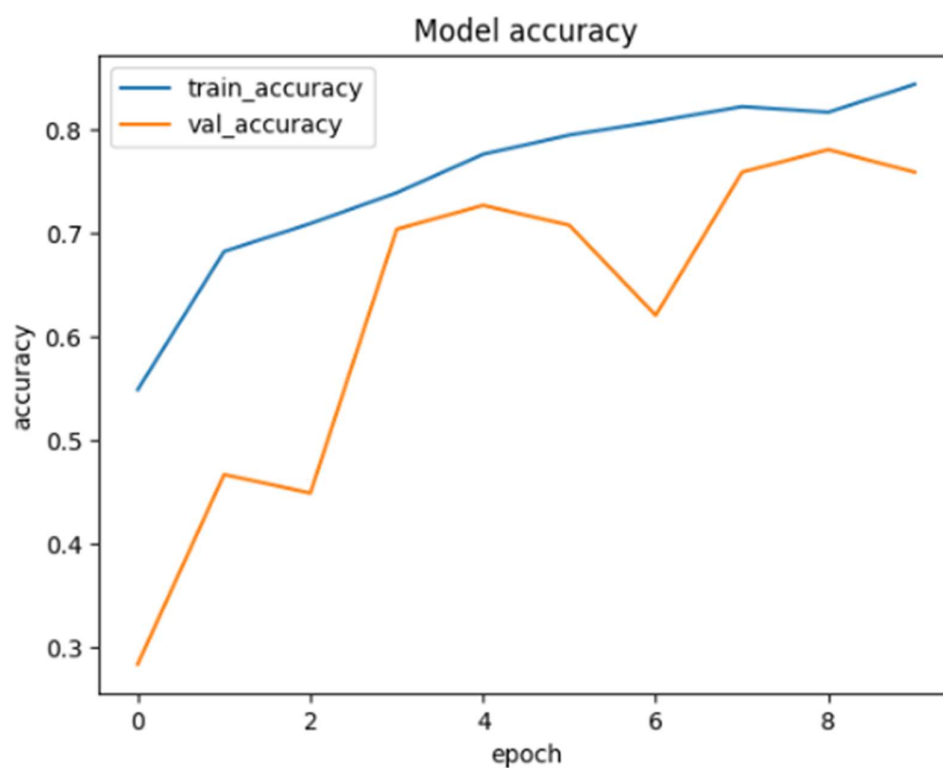
accuracy: 75.9%

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
```

Wykresy wygenerowane na podstawie historii danych zebranych podczas treningu modelu:





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.

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'.

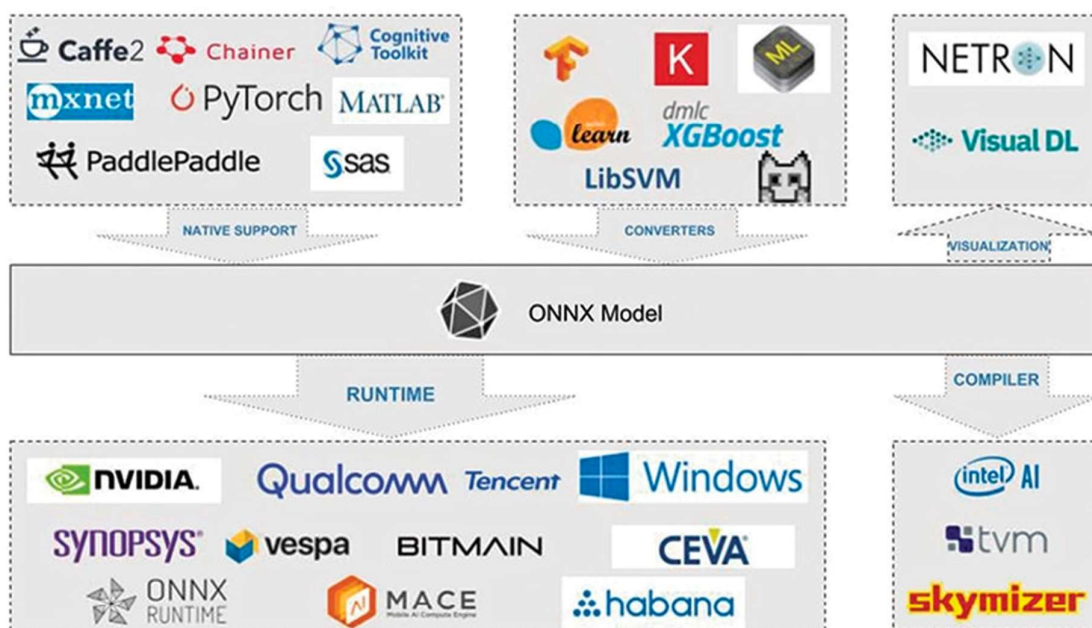
```
resnet_18.save('/content/drive/MyDrive/resnet_18_model')  
  
!python -m tf2onnx.convert --saved-model /content/drive/MyDrive/resnet_18_model/ --output resnet_18_model.onnx
```

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



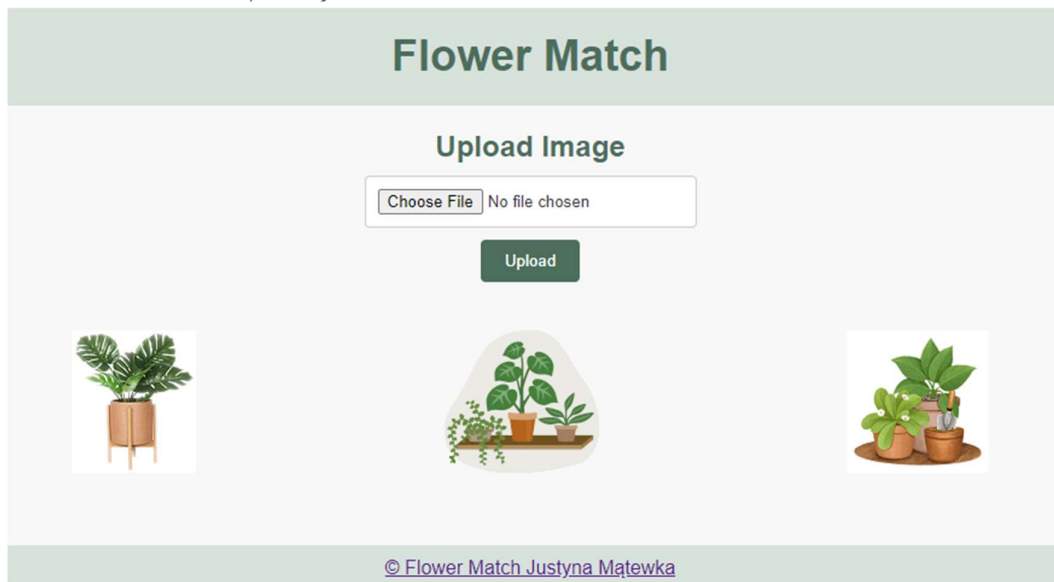
Format modelu interpretowany przez większość języków używanych do uczenia maszynowego, kompilatorów i frameworków, umożliwia np. wykorzystanie modelu napisanego przy użyciu TensorFlow do modyfikacji w PyTorch. Inną ważną zaletą ONNX jest współpraca z różnymi silnikami jak Nvidia, Windows, ONNX Runtime,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



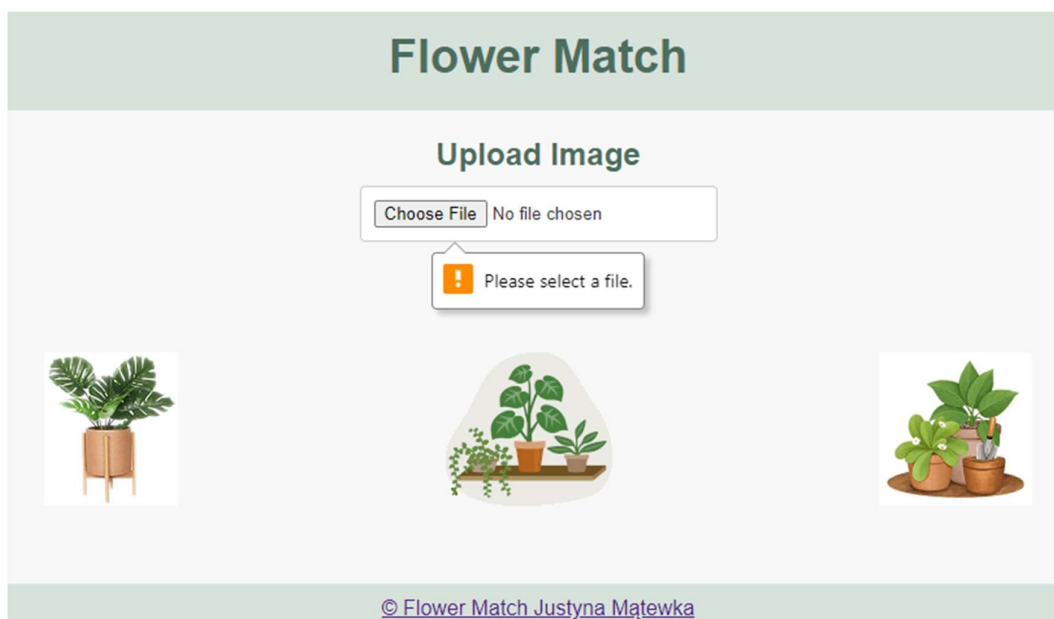
Interface użytkownika:

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.

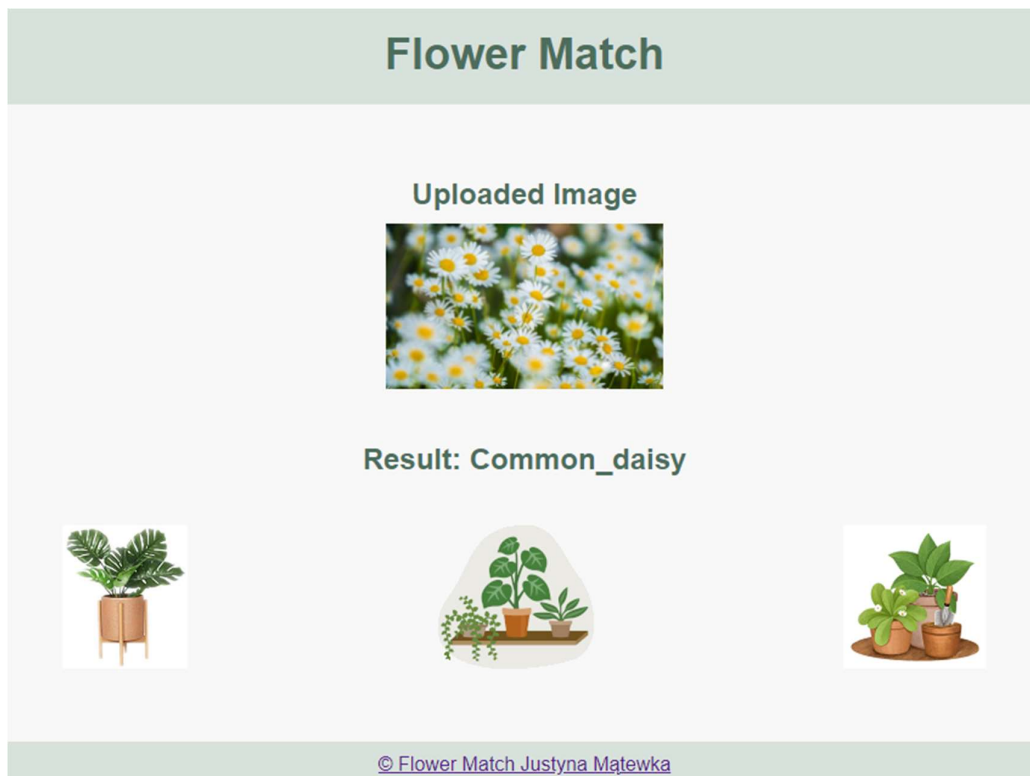
Zrzut ekranu interfejsu użytkownika:



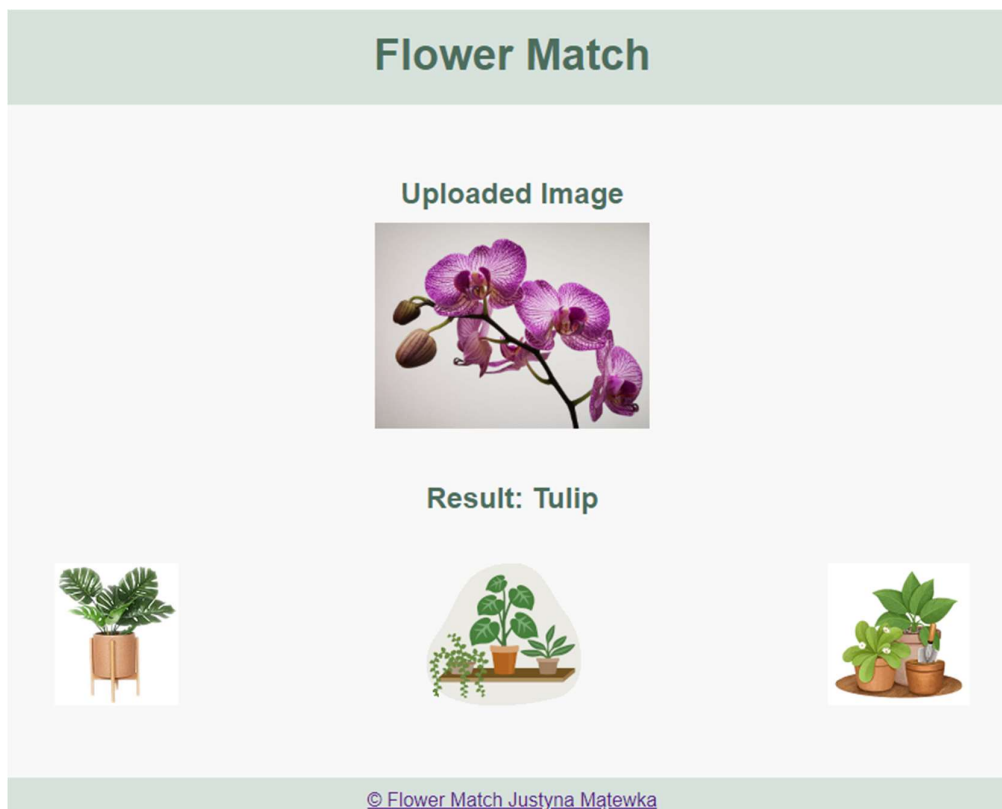
Do uruchomienia jest wymagane zdjęcie:



Zrzut ekranu poprawnie rozpoznanego zdjęcia:



Zrzut ekranu błędnie rozpoznanego zdjęcia:



API:

Funkcjonalności strony internetowej są realizowane przy użyciu frameworku FastAPI oraz silnika szablonów Jinja2

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



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



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.



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>