# Laboratorium: Analiza obrazów przy pomocy sieci konwolucyjnych

May 28, 2025

## 1 Zakres ćwiczeń

- Wykorzystanie konwolucyjnych sieci neuronowych (CNN) do analizy obrazu.
- Pobieranie gotowego modelu przy pomocy biblioteki Tensorflow Datasets.
- Przetwarzanie i udostępnianie danych przy pomocy *Dataset API*.
- Wykorzystanie gotowych modeli do uczenia transferowego.

## 2 Ćwiczenia

### 2.1 Ładowanie danych

Do załadowania danych skorzystamy z pakietu Tensorflow Datasets, który udostępnia wiele zbiorów przydatnych do uczenia maszynowego. Aby utrzymać względnie krótkie czasy uczenia, do ćwiczeń będziemy używać zbioru `tf_flowers`:

```
import tensorflow_datasets as tfds

[test_set_raw, valid_set_raw, train_set_raw], info = tfds.load(
    "tf_flowers",
    split=['train[:10%]', "train[10%:25%]", "train[25%:]"],
    as_supervised=True,
    with_info=True)
```

Kilka słów o argumentach metody `load`:

- `split` zapewnia odpowiedni podział zbioru (dlatego pierwszy element zwracanej krotki jest 3-elementowym słownikiem),
- `as_supervised` sprawia, że zwracane obiekty `tf.data.Dataset` mają postać krotek zawierających zarówno cechy, jak i etykiety,
- `with_info` dodaje drugi element zwracanej krotki.

Możemy łatwo wyekstrahować istotne parametry zbioru:

```
class_names = info.features["label"].names
n_classes = info.features["label"].num_classes
dataset_size = info.splits["train"].num_examples
```

Wyświetlmy kilka przykładowych obrazów:

```
plt.figure(figsize=(12, 8))
index = 0

sample_images = train_set_raw.take(9)

for image, label in sample_images:
    index += 1
    plt.subplot(3, 3, index)
    plt.imshow(image)
    plt.title("Class: {}".format(class_names[label]))
    plt.axis("off")

plt.show(block=False)
```

```
2025-05-27 19:01:23.017904: I
tensorflow/core/kernels/data/tf_record_dataset_op.cc:387] The default buffer
size is 262144, which is overridden by the user specified `buffer_size` of
8388608
2025-05-27 19:01:23.074651: W
tensorflow/core/kernels/data/cache_dataset_ops.cc:916] The calling iterator did
not fully read the dataset being cached. In order to avoid unexpected truncation
of the dataset, the partially cached contents of the dataset  will be discarded.
This can happen if you have an input pipeline similar to
`dataset.cache().take(k).repeat()`. You should use
`dataset.take(k).cache().repeat()` instead.
2025-05-27 19:01:23.076817: I tensorflow/core/framework/local_rendezvous.cc:407]
Local rendezvous is aborting with status: OUT_OF_RANGE: End of sequence
```


## 2.2 Budujemy prostą sieć CNN

W tym ćwiczeniu zbudujemy sieć o nieskompikowanej strukturze.

### 2.2.1 Przygotowanie danych

Sieć będzie przetwarzała obrazy o rozmiarze $224 \times 224$ pikseli, a więc pierwszym krokiem będzie przetworzenie. Obiekty `Dataset` pozwalają na wykorzystanie metody `map`, która przy uczeniu nadzorowanym będzie otrzymywała dwa argumenty (cechy, etykieta) i powinna zwracać je w postaci krotki po przetworzeniu.

Najprostsza funkcja będzie po prostu skalowała obraz do pożądanego rozmiaru:

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    return resized_image, label
```

Aplikujemy ją do pobranych zbiorów:

```
batch_size = 32

train_set = train_set_raw.map(preprocess).shuffle(dataset_size).
 ↪batch(batch_size).prefetch(1)
valid_set = valid_set_raw.map(preprocess).batch(batch_size).prefetch(1)
```

```
test_set = test_set_raw.map(preprocess).batch(batch_size).prefetch(1)
```

Wykorzystujemy tu dodatkowe metody *Dataset API* tak aby dostarczanie danych nie stało się wąskim gardłem procesu uczenia:

- `shuffle` losowo ustawia kolejność próbek w zbiorze uczącym,
- `batch` łączy próbki we wsady o podanej długości (idealnie, powinna to być wielkość miniwsadu podczas uczenia),
- `prefetch` zapewnia takie zarządzanie buforem, aby zawsze przygotowane było $n$ próbek gotowych do pobrania (w tym przypadku chcemy, aby podczas przetwarzania miniwsadu przez algorytm uczenia zawsze czekał jeden przygotowany kolejny miniwsad).

Wyświetlmy próbkę danych po przetworzeniu:

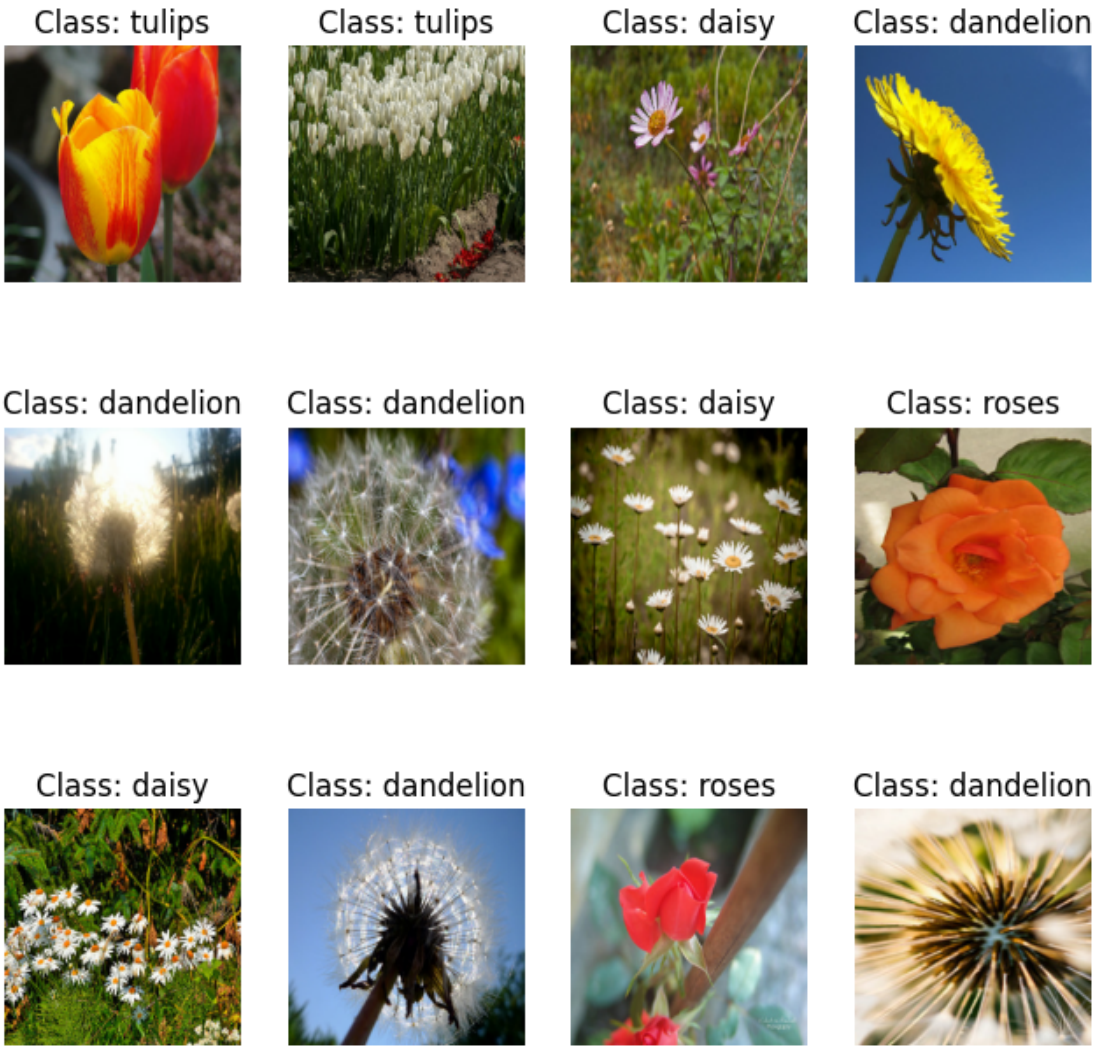
```
plt.figure(figsize=(8, 8))
sample_batch = train_set.take(1)
print(sample_batch)


for X_batch, y_batch in sample_batch:
    for index in range(12):
        plt.subplot(3, 4, index + 1)
        plt.imshow(X_batch[index]/255.0)
        plt.title("Class: {}".format(class_names[y_batch[index]]))
        plt.axis("off")

plt.show()
```

```
<_TakeDataset element_spec=(TensorSpec(shape=(None, 224, 224, 3),
dtype=tf.float32, name=None), TensorSpec(shape=(None,), dtype=tf.int64,
name=None))>
```

```
2025-05-27 19:01:56.902630: I tensorflow/core/framework/local_rendezvous.cc:407]
Local rendezvous is aborting with status: OUT_OF_RANGE: End of sequence
```




### 2.2.2 Budowa sieci

Zaprojektuj prostą sieć konwolucyjną, która pozwoli na uzyskanie przyzwoitej dokładności klasyfikacji przetwarzanego zbioru.

Pamiętaj o istotnych zasadach:

1. W przypadku naszych danych, ponieważ składowe RGB pikseli mają wartości z zakresu 0–255, musimy pamiętać o normalizacji danych; można użyć do tego warstwy skalującej wartości.
2. Część wykrywająca elementy obrazu składa się z warstw konwolucyjnych, najczęściej przeplatanych warstwami zbierającymi:
   - głównymi parametrami warstw konwolucyjnych są liczba filtrów i rozmiar filtra; zazwyczaj zaczynamy od względnie niskiej liczby filtrów (np. 32) o większym rozmiarze (np. $7 \times 7$), aby wykryć elementarne komponenty obrazu, a na kolejnych warstwach łączymy je w bardziej złożone struktury – kombinacji jest więcej, a więc mamy coraz więcej filtrów, ale mogą być mniejszego rozmiaru (np. $3 \times 3$),

- zwyczajowo na jedną warstwę konwolucyjną przypadała jedna warstwa zbierająca (zmniejszająca rozmiar „obrazu"), ale często stosujemy też kilka (np. 2) warstw konwolucyjnych bezpośrednio na sobie.

3. Po części konwolucyjnej typowo następuje część gęsta, złożona z warstw gęstych i opcjonalnie regularyzacyjnych (dropout?):
   - część gęsta musi być poprzedzona warstwą spłaszczającą dane, gdyż spodziewa się 1-wymiarowej struktury,
   - ostatnia warstwa musi być dostosowana do charakterystyki zbioru danych.

Skompiluj model z odpowiednimi parametrami, tak aby zbierana była metryka dotycząca dokładności.

Przeprowadź uczenie przez 10 epok.

Sprawdź jakość modelu. Model powinien zapewniać dokładność *dla zbioru testowego* co najmniej na poziomie ok. 60%.

Zapisz wynik ewaluacji dla zbioru uczącego, walidacyjnego i testowego w postaci krotki `(acc_train, acc_valid, acc_test)` do pikla `simple_cnn_acc.pkl`.

6 pkt.

```
model.save('simple_cnn_flowers.keras')
```

Zapisz model do pliku `simple_cnn_flowers.keras`.

6 pkt.

## 2.3 Uczenie transferowe

Tym razem wykorzystamy gotową, dużo bardziej złożoną sieć. Dzięki temu, że sieć przeszła już proces uczenia, a więc wagi nie są w niej losowe, możemy znacząco skrócić czas uczenia.

Jako bazową wykorzystamy względnie nowoczesną sieć Xception. Jest ona dostępna w pakiecie `tf.keras.applications.xception`.

Wykorzystamy wcześniej już załadowane surowe zbiory danych (`..._set_raw`).

### 2.3.1 Przygotowanie danych

Gotowe modele często dostarczają własnych funkcji przygotowujących wejście w sposób zapewniający optymalne przetwarzanie. Musimy więc zmienić nieco funkcję przygotowującą dane, dodając wywołanie odpowiedniej metody.

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = tf.keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

Zobaczmy jak tym razem wyglądają wstępnie przetworzone dane; zwróć uwagę, że ponieważ teraz wartości należą już do zakresu $(-1, 1)$, musimy je odpowiednio przeskalować (ale w sieci nie będziemy potrzebowali warstwy skalującej):

```
plt.figure(figsize=(8, 8))
sample_batch = train_set.take(1).repeat()
for X_batch, y_batch in sample_batch:
    for index in range(12):
        plt.subplot(3, 4, index + 1)
        plt.imshow(X_batch[index] / 2 + 0.5)
        plt.title("Class: {}".format(class_names[y_batch[index]]))
        plt.axis("off")

plt.show()
```

### 2.3.2 Budowa sieci

Utwórz model bazowy przy pomocy odpowiedniej metody:

```
base_model = tf.keras.applications.xception.Xception(
    weights="imagenet",
    include_top=False)
```

Wyjaśnienie:

- argument `weights` zapewnia inicjalizację wag sieci wynikami uczenia zbiorem ImageNet,
- argument `include_top` sprawi, że sieć nie będzie posiadała górnych warstw (które musimy sami dodać, gdyż są specyficzne dla danego problemu).

Możesz wyświetlić strukturę załadowanej sieci:

`tf.keras.utils.plot_model(base_model)`

Korzystając z API funkcyjnego Keras dodaj warstwy:

- uśredniającą wartości wszystkich „pikseli”,
- wyjściową, gęstą, odpowiednią dla problemu.

Utwórz model korzystając z odpowiedniego konstruktora, podając jako wejścia (`inputs`) wejście modelu bazowego, a jako wyjścia – utworzoną warstwę wyjściową.

Przeprowadź uczenie w dwóch krokach:

1. Kilka (np. 5) epok, podczas których warstwy sieci bazowej będą zablokowane; ten krok jest konieczny aby zapobiec „zepsuciu” wag dostarczonych wraz z siecią bazową ze względu na spodziewane duże błędy wynikające z braku przyuczenia „nowych” warstw:

   ```
   for layer in base_model.layers:
       layer.trainable = False
   ```

2. Kolejne epoki, już z odblokowanymi do uczenia warstwami bazowymi.

Aby ograniczyć czas uczenia (i oceniania), ustaw niezbyt wysoką liczbę iteracji w drugiej fazie. Przeprowadź ewaluację modelu analogicznie do tej w poprzednim zadaniu.

Zapisz wynik ewaluacji dla zbioru uczącego, walidacyjnego i testowego w postaci krotki `(acc_train, acc_valid, acc_test)` do pikla `xception_acc.pkl`.

6 pkt.

Zapisz model do pliku `xception_flowers.keras`.

6 pkt.