

## 02\_PREPROCESSING

**Date:** 06-10-2025

**Goals:** Build a preprocessing pipeline to turn the images into compatible data for the upcoming models.

Let's build the pipeline function, which will take the manifest filepath in (containing labels and image filepaths), and return the train and validation dataset.

```
In [1]: # imports
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import train_test_split
from pathlib import Path

try:
    PROJECT_ROOT = Path(__file__).resolve().parents[2]
except NameError:
    PROJECT_ROOT = Path.cwd().resolve().parent
DATA_PATH = PROJECT_ROOT / "data" / "labels"

labels_and_paths_csv_fp = DATA_PATH / "labels_manifest_1000.csv" #labels filepath

labels_fp = PROJECT_ROOT / "data" / "processed" / "manifest_train_and_val.csv"
```

First, the function will read and extract the needed columns (filepath, label) from the manifest, and then use the `train_test_split()` function to split the dataset.

```
In [2]: df = pd.read_csv(labels_fp) # read the labels csv file
labels_and_fp_cols = df[["derived_label", "filepath"]] # select needed labels

train_df, valid_df = train_test_split(labels_and_fp_cols, # split the data for t
                                      test_size=0.2,
                                      stratify=labels_and_fp_cols["derived_l
                                      random_state=37)
```

Next, let's translate categorical data (words) to integers. I assign a number to every type of galaxy. For example: 0 corresponds to elliptical, 1 to spiral, and so on.

```
In [13]: labels_to_indexes = {} # create a dictionary with labels and indexes (elliptical
unique_labels = labels_and_fp_cols["derived_label"].unique()
for i in range (unique_labels):
    labels_to_indexes[labels_and_fp_cols["derived_label"].unique()[i]] = i
```

Now, we need a python list of all the data so far (paths and labels separate lists for each train/validation set). It is a necessary step, as the later functions require this data type.

```
In [14]: train_paths_list = train_df["filepath"].apply(lambda x: str(PROJECT_ROOT / x)).t
valid_paths_list = valid_df["filepath"].apply(lambda x: str(PROJECT_ROOT / x)).t
train_labels_list = []
valid_labels_list = []
```

```

for i in train_df["derived_label"]:
    train_labels_list.append(labels_to_indexes[i])
for i in valid_df["derived_label"]:
    valid_labels_list.append(labels_to_indexes[i])

```

It is crucial to define a function that resizes and normalizes every image to sets of numbers between 0 and 1, representing each pixel's intensity.

```

In [15]: def PREPROCESS(path, label): # function to flatten the images and one-hot encode
    image = tf.io.read_file(path)
    image = tf.image.decode_image(image, channels=3)
    image.set_shape([None, None, 3])
    image = tf.image.resize(image, [224, 224])
    image = tf.cast(image, tf.float32) / 255.0 # cast the image to a tensor with
    labels = tf.one_hot(label, unique_labels) # one-hot encoding the labels
    return image, labels

```

Finally, we need to create the datasets, suitable for training, using the previous lists we crafted.

AUTOTUNE lets us optimize the process by adding parallel computation.

The main data pipeline consists of four steps:

- mapping using our earlier written PREPROCESS function
- shuffling the data to avoid biased distribution
- batch all the data into small chunks
- prefetch for optimization (AUTOTUNE)

```

In [16]: AUTOTUNE = tf.data.AUTOTUNE # AUTOTUNE variable for an optimized performance

train_ds = tf.data.Dataset.from_tensor_slices((train_paths_list, train_labels_list))
train_ds = (
    train_ds.shuffle(buffer_size=len(train_paths_list), seed=37)
    .map(PREPROCESS, num_parallel_calls=AUTOTUNE)
    .batch(32)
    .prefetch(AUTOTUNE)
)

valid_ds = tf.data.Dataset.from_tensor_slices((valid_paths_list, valid_labels_list))
valid_ds = (
    valid_ds.shuffle(buffer_size=len(valid_paths_list), seed=37)
    .map(PREPROCESS, num_parallel_calls=AUTOTUNE)
    .batch(32)
    .prefetch(AUTOTUNE)
)

```

Now all we have left to do is return train\_ds and valid\_ds! The preprocessing is complete and the resulting datasets can be passed straight to the model at fitting time.

Note: it is only possible to do most of these functions because the dataset has already been checked for quality, and missing filepaths/labels are absent.

The full function is saved to data\_loader.py.

Next, I will build the first baseline classifier model (see 03\_baselines.ipynb).