Convolutional Neural Network From Scratch

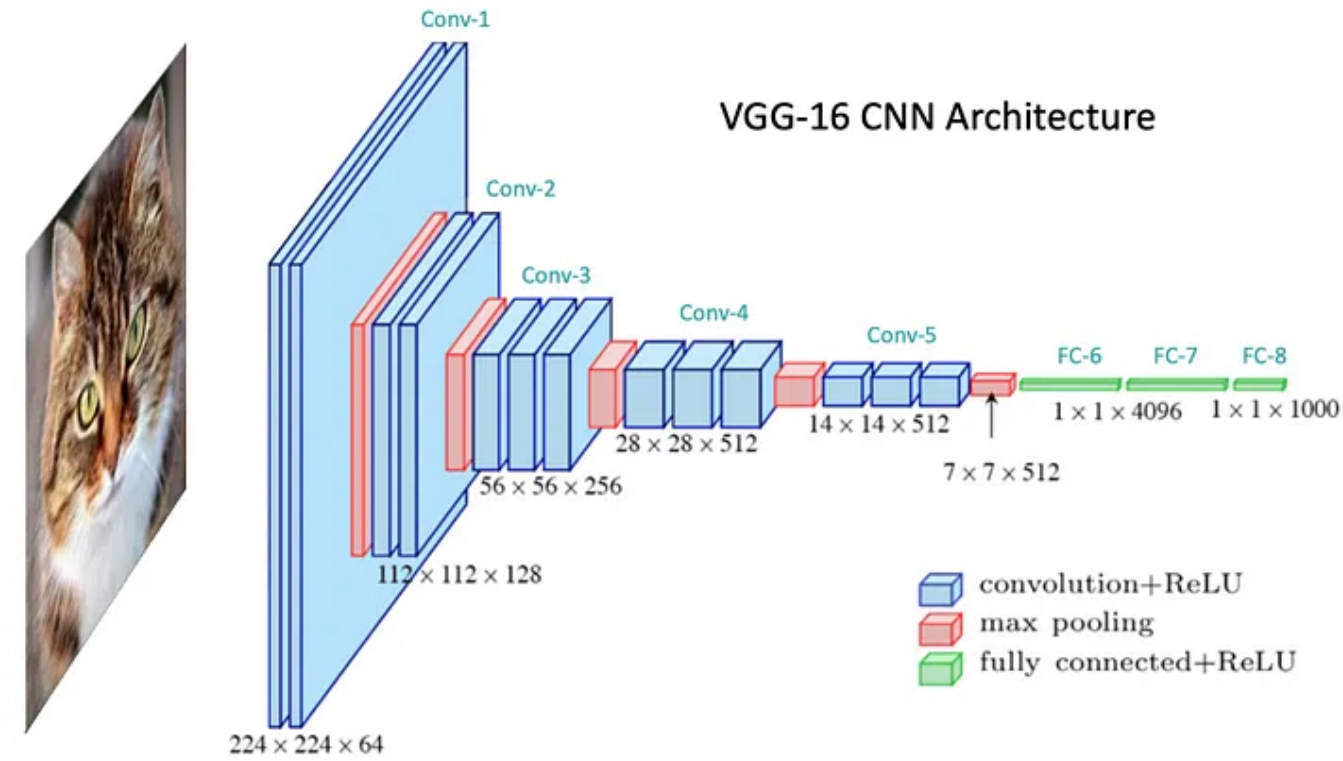
The most effective way of working with image data

Luís Fernando Torres · Follow

21 min read · Just now







VGG-16 Convolutional Neural Network. Source: LearnOpenCV

Note: This article can be better read on my Kaggle Notebook, 🧠 **Convolutional Neural Network From Scratch**.

Introduction

Convolutional Neural Networks (CNNs or ConvNets) are specialized neural architectures that is predominantly used for several **computer vision** tasks, such as image classification and object recognition. These neural networks harness the power of *Linear Algebra,* specifically through convolution operations, to identify patterns within images.

Convolutional neural networks have three main kinds of layers, which are:

• Convolutional layer

• Pooling layer

• Fully-connected layer

The convolutional layer is the first layer of the network, while the fully-connected layer is the final layer, responsible for the output. The first convolutional layer may be followed by several additional convolutional layers or pooling layers; and with each new layer, the more complex is the CNN.

As the CNN gets more complex, the more it excels in identifying greater portions of the image. Whereas earlier layers focus on the simple features, such as colors and edges; as the image progresses through the network, the CNN starts to recognize larger elements and shapes, until finally reaching its main goal.

The image below displays the structure of a CNN. We have an input image, followed by Convolutional and Pooling layers, where the feature learning process happens. Later on, we have the layers responsible for the task of classifying whether the vehicle in the input data is a car, truck, van, bicycle, etc.

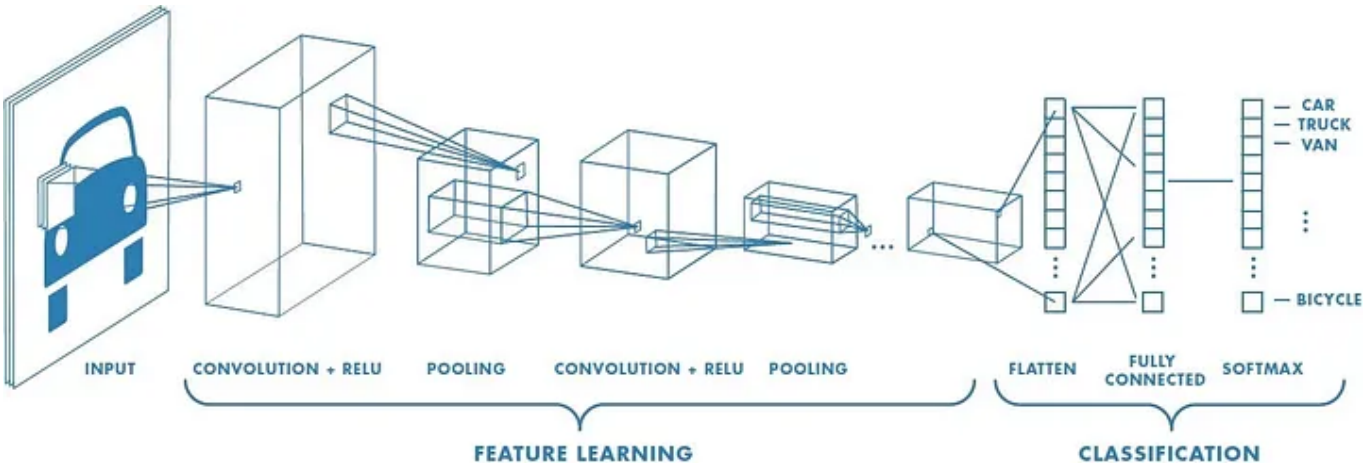


Image displaying the structure of a Convolutional Neural Networks.
Source: Understanding of Convolutional Neural Network (CNN) — Deep Learning

Convolutional Layer

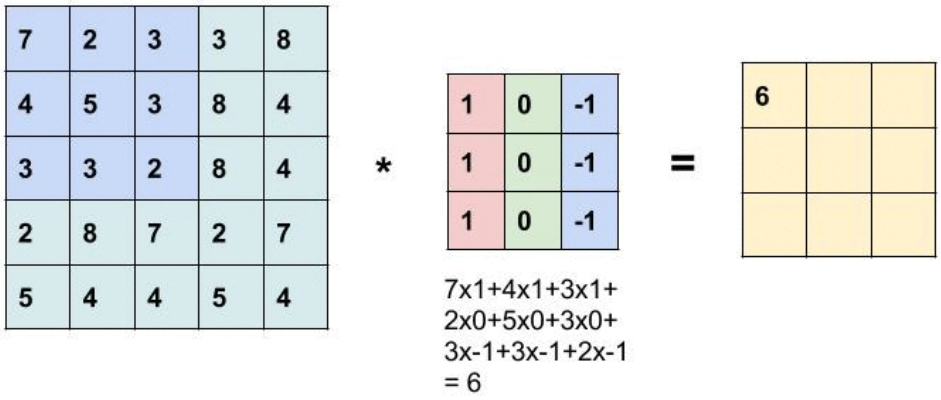
The convolutional layer is the most important layer of a CNN; responsible for dealing with the major computations. The convolutional layer includes **input data**, **a filter**, and **a feature map.**

To illustrate how it works, let's assume we have a color image as input. This image is made up of a matrix of pixels in 3D, representing the three dimensions of the image: height, width, and depth.

The filter — which is also referred to as kernel — is a two-dimensional array of weights, and is typically a 3×33×3 matrix. It is applied to a specific area of the image, and a **dot product** is computed between the input pixels and the weights in the filter. Subsequently, the filter shifts by a stride, and this whole

process is repeated until the kernel slides through the entire image, resulting in an output array.

The resulting output array is also known as a feature map, activation map, or convolved feature.



GIF displaying the convolutional process. First, we have a 5×55×5 matrix — pixels in the input image — with a 3×33×3 filter. The result of the operation is the output array.
Source: Convolutional Neural Networks

It is important to note that the weights in the filter remain fixed as it moves across the image. The weights values are adjusted during the training process due to backpropagation and gradient descent.

Besides the weights in the filter, we have other three important parameters that need to be set before the training begins:

• **Number of Filters:** This parameter is responsible for defining the depth of the output. If we have three distinct filters, we have three different feature maps, creating a depth of three.

• **Stride:** This is the distance, or number of pixels, that the filter moves over the input matrix.

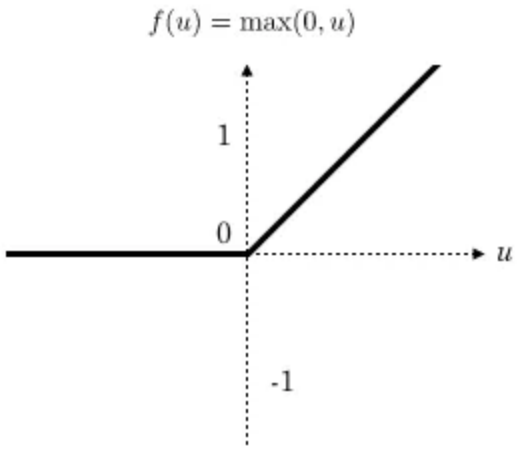
• **Zero-padding:** This parameter is usually used when the filters do not fit the input image. This sets all elements outside the input matrix to zero, producing a larger or equally sized output. There are three different kinds of padding:

■ **Valid padding:** Also known as *no padding*. In this specific case, the last convolution is dropped if the dimensions do not align.

■ **Same padding:** This padding ensures that the output layer has the exact same size as the input layer.

■ **Full padding:** This kind of padding increases the size of the output by adding zeros to the borders of the input matrix.

After each convolution operation, we have the application of a ***Rectified Linear Unit (ReLU)*** function, which transforms the feature map and introduces nonlinearity.



ReLU activation function:

$$f(u) = \begin{cases} 0 & \text{if } u \leq 0 \\ u & \text{if } u > 0 \end{cases}$$

Source: ResearchGate

As mentioned earlier, the initial convolutional layer can be followed by additional convolutional layers.

The subsequent convolutional layers can see the pixels within the receptive fields of the prior layers, which helps to extract and interpret additional patterns.

Pooling Layer

The pooling layer is responsible for reducing the dimensionality of the input. It also slides a filter across the entire input — without any weights — to populate the output array. We have two main types of pooling:

• **Max Pooling:** As the filter slides through the input, it selects the pixel with the highest value for the output array.

• **Average Pooling:** The value selected for the output is obtained by computing the average within the receptive field.

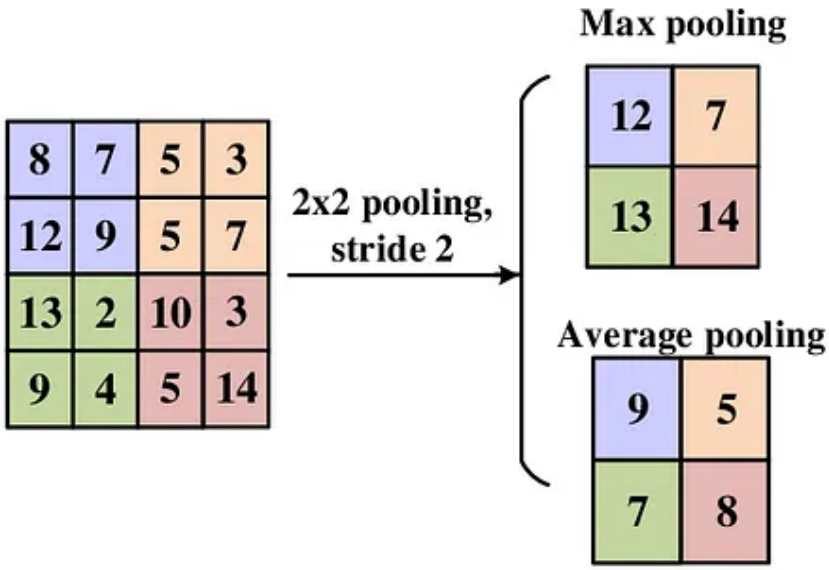


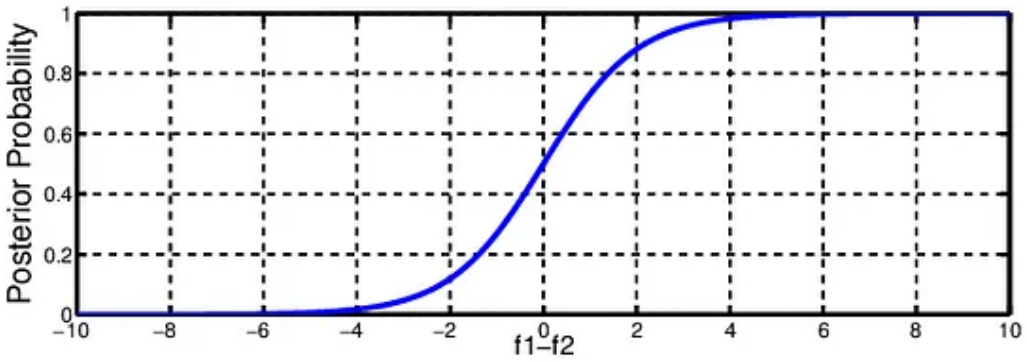
Illustration of the pooling process.

Source: ResearchGate

The pooling layer serves the purpose of reducing complexity, improving efficiency, and limiting the risk of overfitting.

Fully-connected Layer

This is the layer responsible for performing the task classification based on the features extracted during the previous layers. While both convolutional and pooling layers tend to use ***ReLU*** functions, fully-connected layers use the ***Softmax*** activation function for classification, producing a probability from *0* to *1*.



Softmax activation function graph.

Source: ResearchGate

CNNs and Computer Vision

Due to its power in image recognition tasks, CNNs have been highly effective in many fields related to ***Computer Vision***.

Computer Vision is a field of AI that enables computers to extract information from digital images, videos, and other visual inputs. Some common applications of computer vision today can be seen across several industries, including the following:

• **Social Media:** *Google*, *Meta*, and *Apple* use these systems to identify people in a photograph, making it easier to organize photo albums and tag friends.

• **Healthcare:** Computer vision models have been used to help doctors identifying cancerous tumors in patients, as well as other conditions.

• **Agriculture:** Drones equipped with cameras can monitor the health of vast farmlands to identify areas that need more water or fertilizers.

• **Security:** Surveillance systems can detect unusual and suspect activities in real time.

• **Finance:** Computer vision models may be used to identify relevant patterns in candlestick charts to predict price movements.

• **Automotive:** Computer vision is an essential component of the research leading to self-driving cars.

This Article

Nowadays, there are several *pre-trained* CNNs available for many tasks. Models like ***ResNet, VGG16, InceptionV3,*** as well as many others, are highly efficient in most computer vision tasks we currently perform across industries.

In this article, however, I would like to explore the process of building a simple, yet effective, Convolutional Neural Network from scratch. For this task, I will use ***Keras*** to help us build a neural network that can accurately identify diseases in a plant through images.

I am going to use the Plant Disease Recognition Dataset, which contains 1,530 images divided into train, test, and validation sets. The images are labeled as *"Healthy"*, *"Rust"*, and *"Powdery"* to describe the conditions of the plants.

Very briefly, each class means the following:

• **Rust:** These are plant diseases caused by Pucciniales fungi, which cause severe deformities to the plant.

• **Powdery:** Powdery mildews are caused by Erysphales fungi, posing a threat to agriculture and horticulture by reducing crop yields.

• **Healthy:** Naturally, these are the plants that are free from diseases.

Exploring the Data

Before building our Convolutional Neural Network, it is helpful to perform a brief, yet efficient, analysis of the data we have at hand. Let's start by loading the directories for each set.

```
# Loading training, testing, and validation directories
train_dir = '/kaggle/input/plant-disease-recognition-dataset/Train/Train'
test_dir = '/kaggle/input/plant-disease-recognition-dataset/Test/Test'
val_dir = '/kaggle/input/plant-disease-recognition-dataset/Validation/Validation
```

We may also count the files inside each subfolder to compute the total of data we have for training and testing, as well as measure the degree of class imbalance.

```
* * * * * Number of files in each folder * * * * *

Train/Healthy: 458

Train/Powdery: 430

Train/Rust: 434

  Total: 1322
--------------------------------------------------------------------------------

Test/Healthy: 50

Test/Powdery: 50

Test/Rust: 50
```

```
  Total: 150
--------------------------------------------------------------------------------

Validation/Healthy: 20

Validation/Powdery: 20

Validation/Rust: 20

  Total: 60
--------------------------------------------------------------------------------
```

We have a total of **1,322** files inside the `Train` directory and there are no large imbalances between classes. A small variation between them is fine, and a simple metric such as *Accuracy* may be enough to measure performance.

For the testing set, we have a total of **150** images, whereas the validation set consists of **60** images in total. Both sets have a perfect class balance.

Convolutional Neural Networks require a fixed size for all images we feed into it. This means that every single image in our dataset must be equally sized, either 128×128128×128, 224×224224×224, and so on.

We can also check if our data meets this requirement, or if it will be necessary to perform some preprocessing in this regard before modeling.

```
Found 8 unique image dimensions: {(4032, 3024),
(4000, 2672), (4000, 3000), (5184, 3456), (2592, 1728),
(3901, 2607), (4608, 3456), (2421, 2279)}
```

We have 8 different dimensions across the dataset. In the next cell, I am going to check the distribution of these dimensions across the data.

```
Dimension (4000, 2672): 1130 images

Dimension (4000, 3000): 88 images

Dimension (2421, 2279): 1 images

Dimension (2592, 1728): 127 images

Dimension (4608, 3456): 72 images

Dimension (5184, 3456): 97 images

Dimension (4032, 3024): 16 images

Dimension (3901, 2607): 1 images
```

t seems that most images have dimensions of *4000×2672*, which is a **rectangular shape**. We can conclude that, due to the differences in dimensions, we will need to apply some preprocessing to the data.

First, we are going to resize the images, so they all have the same shape. Then, we will transform the input from rectangular shape to **square** shape.

Another crucial consideration is verifying the pixel valye range of the images. In this case, all images should have pixel values spanning from **0** to **255**. This consistency simplifies the preprocessing step, since we often normalize pixel values in images to a range going from 0 to 1.

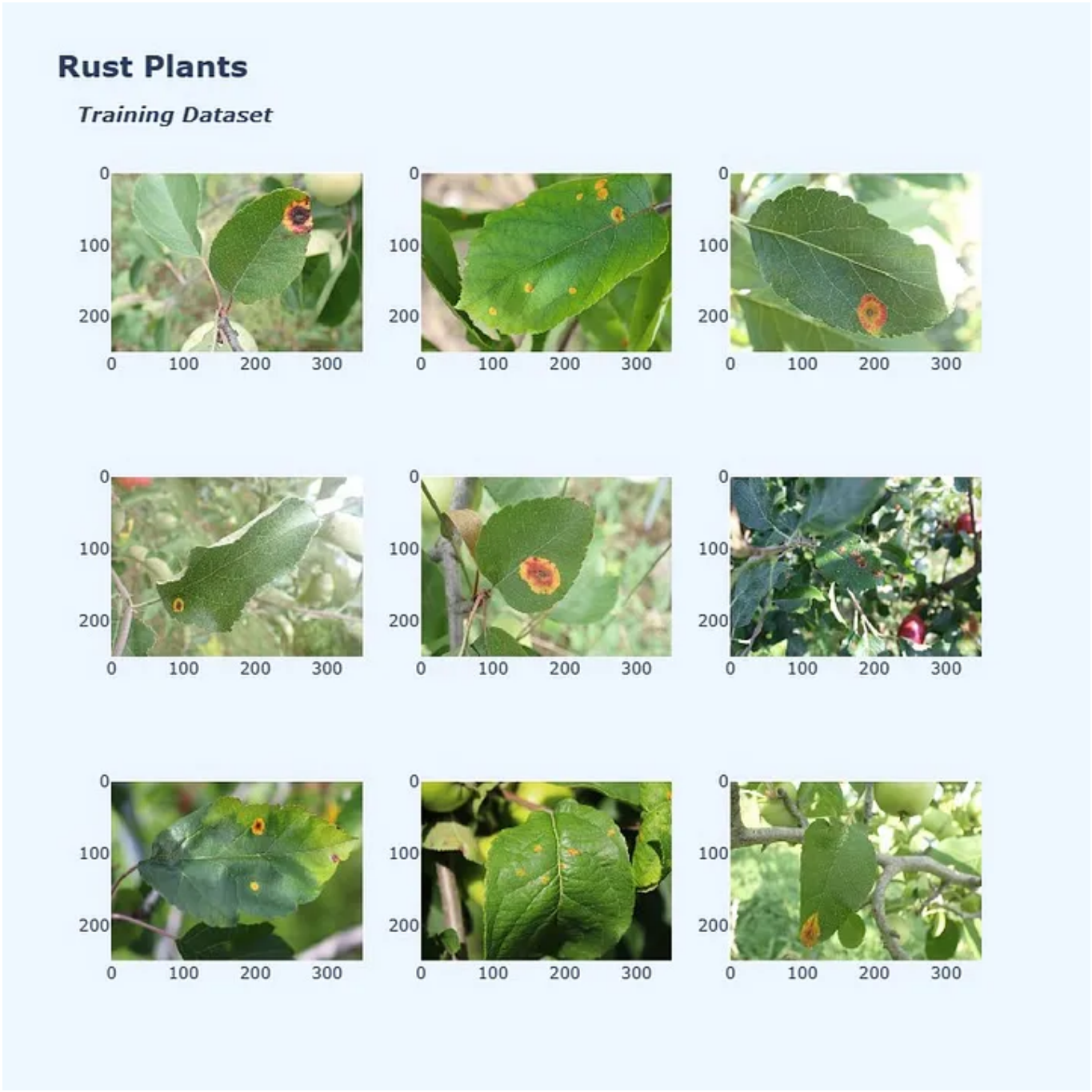
```
- Not all images are of data type uint8
```

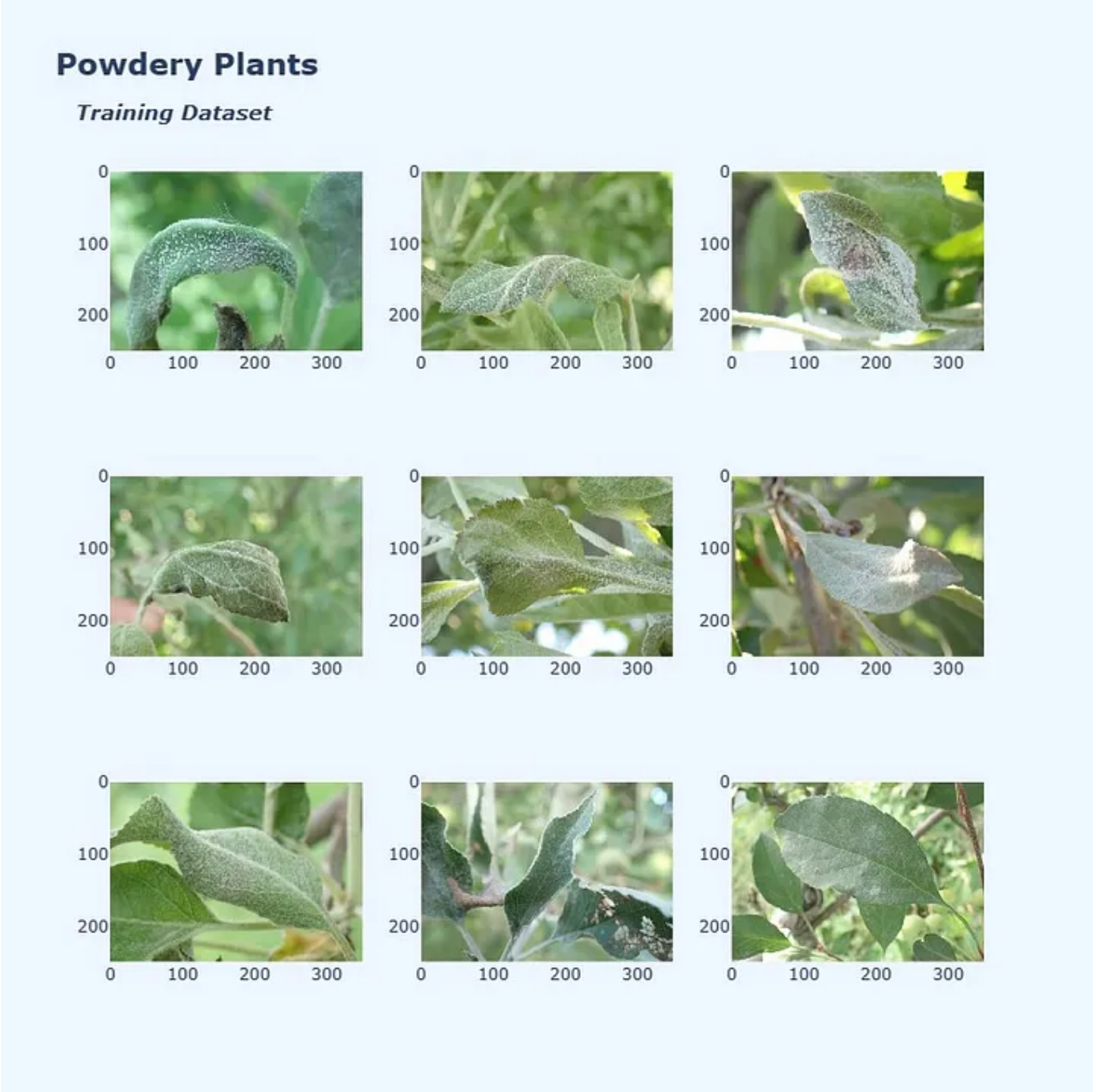
```
    - All images have pixel values ranging from 0 to 255
```

Even though not all images are of the same data type, `uint8`, it is fairly easy to guarantee that they will have the same data type once we load images into datasets. We confirmed, though, that all the images have pixel values ranging from 0 to 255, which is great news.

Before moving on to the *Preprocessing* step, let's plot some images from each class to see what they look like.


Healthy Plants
Training Dataset


Rust Plants
Training Dataset



Preprocessing

For those familiar with tabular data, preprocessing is probably one of the most daunting steps of dealing with neural networks and unstructured data.

This task can be fairly easy by using TensorFlow's `image_dataset_from_directory`, which loads images from the directories as a

TensorFlow Dataset. This resulting dataset can be manipulated for batching, shuffling, augmentating, and several other preprocessing steps.

I suggest you check this link for more information on the `image_dataset_from_directory` function.

```
# Creating a Dataset for the Training data
train = tf.keras.utils.image_dataset_from_directory(
    train_dir,  # Directory where the Training images are located
    labels = 'inferred', # Classes will be inferred according to the structure o
    label_mode = 'categorical',
    class_names = ['Healthy', 'Powdery', 'Rust'],
    batch_size = 16,    # Number of processed samples before updating the model'
    image_size = (256, 256), # Defining a fixed dimension for all images
    shuffle = True,  # Shuffling data
    seed = seed,  # Random seed for shuffling and transformations
    validation_split = 0, # We don't need to create a validation set from the tr
    crop_to_aspect_ratio = True # Resize images without aspect ratio distortion
)
```

```
Found 1322 files belonging to 3 classes.
```

```
# Creating a dataset for the Test data
test = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    labels = 'inferred',
    label_mode = 'categorical',
    class_names = ['Healthy', 'Powdery', 'Rust'],
    batch_size = 16,
    image_size = (256, 256),
    shuffle = True,
    seed = seed,
    validation_split = 0,
    crop_to_aspect_ratio = True
)
```

```
Found 150 files belonging to 3 classes.
```

```
# Creating a dataset for the Test data
validation = tf.keras.utils.image_dataset_from_directory(
    val_dir,
    labels = 'inferred',
    label_mode = 'categorical',
    class_names = ['Healthy', 'Powdery', 'Rust'],
    batch_size = 16,
    image_size = (256, 256),
    shuffle = True,
    seed = seed,
    validation_split = 0,
    crop_to_aspect_ratio = True
)
```

```
Found 60 files belonging to 3 classes.
```

We have successfully captured all files within each set for each of the three classes. We can also print these datasets for a further understanding of their structure.

```
print('\nTraining Dataset:', train)
print('\nTesting Dataset:', test)
print('\nValidation Dataset:', validation)
```

```
Training Dataset: <_BatchDataset element_spec=(
TensorSpec(shape=(None, 256, 256, 3), dtype=tf.float32, name=None),
TensorSpec(shape=(None, 3), dtype=tf.float32, name=None))>

Testing Dataset: <_BatchDataset element_spec=(
TensorSpec(shape=(None, 256, 256, 3), dtype=tf.float32, name=None),
```

```
TensorSpec(shape=(None, 3), dtype=tf.float32, name=None))>

Validation Dataset: <_BatchDataset element_spec=(
TensorSpec(shape=(None, 256, 256, 3), dtype=tf.float32, name=None),
TensorSpec(shape=(None, 3), dtype=tf.float32, name=None))>
```

Let's explore a bit deeper what all the information above means.

• **_BatchDataset:** It indicates that the dataset returns data in batches.

• **element_spec:** This describes the structure of the elements in the dataset.

• **TensorSpec(shape=(None, 256, 256, 3), dtype=tf.float32, name = None):** This represents the features, in this case the images, in the dataset. `None` represents the batch size, which is *None* here because it can vary depending on how many samples we have in the last batch; `256, 256` represents the height and width of the images; `3` is the number of channels in the images, indicating they are RGB images. Last, `dtype=tf.float32` tells us that the data type of the image pixels is a 32-bit floating point.

• **TensorSpec(shape=(None, 3), dtype=tf.float32, name=None):** This represents the labels/targets of our dataset. Here, `None` refers to the batch size; `3` refers to the number of labels in the dataset; whilst `dtype=tf.float32` is also a 32-bit floating point.

By using the `image_dataset_from_directory` function, we have been able to automatically preprocess some aspects of the data. For instance, all the images are now of the same data type, `tf.float32`. By setting `image_size = (256, 256)`, we have ensured that all images have the same dimensions, 256×256256×256.

Another important step for preprocessing is ensuring that the pixel values of our images are within a 0 to 1 range. The `image_dataset_from_directory` method performed some transformations already, but the pixel values are still in the 0 to 255 range.

```
Minimum pixel value in the Validation dataset 0.0

Maximum pixel value in the Validation dataset 255.0
```

To bring the pixel values to the 0 to 1 range, we can easily use one of Keras' preprocessing layers, `tf.keras.layers.Rescaling`.

```
scaler = Rescaling(1./255) # Defining scaler values between 0 to 1
# Rescaling datasets
train = train.map(lambda x, y: (scaler(x), y))
test = test.map(lambda x, y: (scaler(x), y))
validation = validation.map(lambda x, y: (scaler(x), y))
```

Now we can once more visualize the minimum and maximum pixel values in the validation set.

```
Minimum pixel value in the Validation dataset 0.0

Maximum pixel value in the Validation dataset 1.0
```

Data Augmentation

When working with image data, it is usually a good practice to artificially introduce some diversity to the sample by applying random transformations to the images used in training. This is good because it helps to expose the model to a wider variety of images and avoids overfitting.

Keras has about seven different layers for image data augmentation. These are:

• **tf.keras.layers.RandomCrop**: This layer randomly chooses a location to crop images down to a target size.

• **tf.keras.layers.RandomFlip**: This layer randomly flips images horizontally and or vertically based on the `mode` attribute.

• **tf.keras.layers.RandomTranslation**: This layer randomly applies translations to each image during training according to the `fill_mode` attribute.

• **tf.keras.layers.RandomBrightness**: This layer randomly increases/reduces the brightness for the input RGB images.

• **tf.keras.layers.RandomRotation**: This layer randomly rotates the images during training, and also fills empty spaces according to the `fill_mode` attribute.

• **tf.keras.layers.RandomZoom**: This layer randomly zooms in or out on each axis of each image independently during training.

• **tf.keras.layers.RandomContrast**: This layer randomly adjusts contrast by a random factor during training in or out on each axis of each image

independently during training.

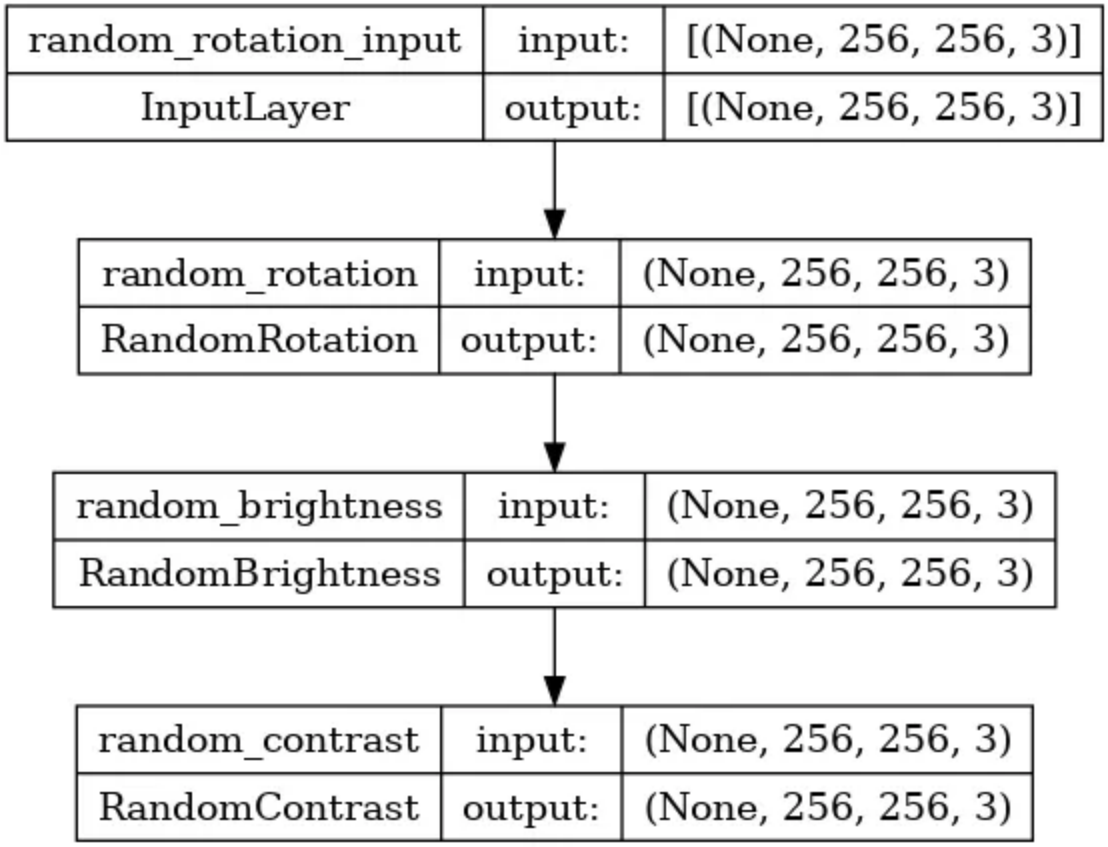
For this task, I am going to apply `RandomRotation`, `RandomContrast`, as well as `RandomBrightness` to our images.

```
# Creating data augmentation pipeline
augmentation = tf.keras.Sequential(
    [
        tf.keras.layers.RandomRotation(
        factor = (-.25, .3),
        fill_mode = 'reflect',
        interpolation = 'bilinear',
        seed = seed),


        tf.keras.layers.RandomBrightness(
        factor = (-.45, .45),
        value_range = (0.0, 1.0),
        seed = seed),

        tf.keras.layers.RandomContrast(
        factor = (.5),
        seed = seed)
    ]
)
```

We can also use an `input_shape` as example to build the pipeline above and plot it below to illustrate how it looks.



Architecture of the Data Augmentation Pipeline

Building the Convolutional Neural Network

To build the Convolutional Neural Network with Keras, we are going to use the `Sequential` class. This class allows us to build a linear stack of layers, which is essential for the creation of neural networks.

Besides the Convolutional, Pooling, and Fully-Connected Layers, which we have previously explored, I am also going to add the following layers to the network:

• **BatchNormalization:** This layer applies a transformation that maintains the mean output close to 00 and the standard deviation close to 11. It normalizes its inputs and is important to help convergence and generalization.

• **Dropout**: This layer randomly sets a fraction of input units to 00 during training, which helps to prevent overfitting.

• **Flatten**: This layer transforms a multi-dimensional tensor into a one-dimensional tensor. It is used when transitioning from the **Feature Learning** segment — Convolutional and Pooling layers — to the fully-connected layers.

I plan to use different kernel sizes, both 3×33×3 and 5×55×5. This may allow the network to capture features at multiple scales.

I am also gradually increasing the *dropout rates* as we advance through the process and the increase in the number of kernels.

With that being said, let's go ahead and build our ConvNet.

```
# Initiating model on GPU
with strategy.scope():
    model = Sequential()

    model.add(augmentation) # Adding data augmentation pipeline to the model

    # Feature Learning Layers
    model.add(Conv2D(32,                  # Number of filters/Kernels
                     (3,3),               # Size of kernels (3x3 matrix)
                     strides = 1,         # Step size for sliding the kernel acr
                     padding = 'same',    # 'Same' ensures that the output featu
                    input_shape = (256,256,3) # Input image shape
                    ))
    model.add(Activation('relu'))# Activation function
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.2))

    model.add(Conv2D(64, (5,5), padding = 'same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.2))
```

```
model.add(Conv2D(128, (3,3), padding = 'same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
model.add(Dropout(0.3))

model.add(Conv2D(256, (5,5), padding = 'same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
model.add(Dropout(0.3))

model.add(Conv2D(512, (3,3), padding = 'same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
model.add(Dropout(0.3))

# Flattening tensors
model.add(Flatten())

# Fully-Connected Layers
model.add(Dense(2048))
model.add(Activation('relu'))
model.add(Dropout(0.5))

# Output Layer
model.add(Dense(3, activation = 'softmax')) # Classification layer
```

By using Keras' `compile` method, we can prepare our neural network for training. This method has several parameters, the ones we will be focusing here are:

• **optimizer**: In this parameter, we define the algorithms to adjust the weight updates. This is an important parameter, because choosing the right optimizer is essential to speed convergence. We are going to use `RMSprop`, which is the best optimizer I've found during the tests I ran.

• **loss**: This is the loss function we're trying to minimize during training. In this case, we are using `categorical_crossentropy`, which is a good choice for classification tasks with over two classes.

• **metrics**: This parameter defines the metric that will be used to evaluate performance during training and validation. Since our data is not heavily unbalanced, we may use `accuracy` for this, which is a very straightforward metric given by the following formula:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

```
# Compiling model
model.compile(optimizer = tf.keras.optimizers.RMSprop(0.0001), # 1e-4
              loss = 'categorical_crossentropy', # Ideal for multiclass tasks
              metrics = ['accuracy']) # Evaluation metric
```

After compiling the model, I am going to define an **Early Stopping** and a **Model Checkpoint**.

Early Stopping serves the purpose of interrupting the training process when a certain metric stops improving over a period of time. In this case, I am going to configure the `EarlyStopping` method to monitor the accuracy in the test set, and stop the training process if we don't have any improvement on it after 5 epochs.

Model Checkpoint will ensure that only the best weights get saved, and we're also going to define the *best weights* according to the accuracy of the model in the test set.

```
# Defining an Early Stopping and Model Checkpoints
early_stopping = EarlyStopping(monitor = 'val_accuracy',
                               patience = 5, mode = 'max',
                               restore_best_weights = True)

checkpoint = ModelCheckpoint('best_model.h5',
                             monitor = 'val_accuracy',
                             save_best_only = True)
```

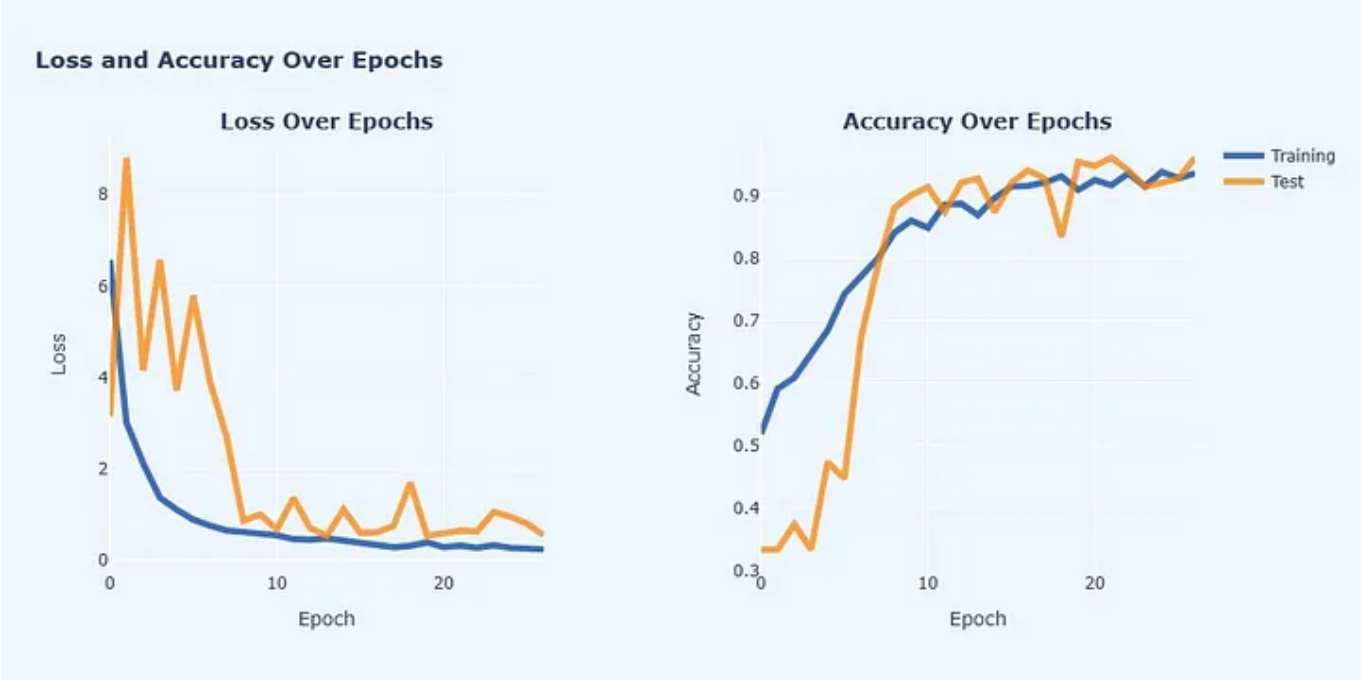
We may now use `model.fit()` to start the training and testing process.

```
Epoch 1/50
83/83 [==============================] - 82s 760ms/step - loss: 6.5686 - accurac
Epoch 2/50
83/83 [==============================] - 69s 768ms/step - loss: 3.0173 - accurac
Epoch 3/50
83/83 [==============================] - 70s 774ms/step - loss: 2.1228 - accurac
Epoch 4/50
83/83 [==============================] - 67s 727ms/step - loss: 1.3750 - accurac
Epoch 5/50
83/83 [==============================] - 67s 744ms/step - loss: 1.1113 - accurac
Epoch 6/50
83/83 [==============================] - 68s 746ms/step - loss: 0.8958 - accurac
Epoch 7/50
83/83 [==============================] - 70s 765ms/step - loss: 0.7605 - accurac
Epoch 8/50
83/83 [==============================] - 72s 792ms/step - loss: 0.6549 - accurac
Epoch 9/50
83/83 [==============================] - 72s 794ms/step - loss: 0.6207 - accurac
Epoch 10/50
83/83 [==============================] - 73s 803ms/step - loss: 0.5761 - accurac
Epoch 11/50
83/83 [==============================] - 73s 800ms/step - loss: 0.5478 - accurac
Epoch 12/50
83/83 [==============================] - 68s 749ms/step - loss: 0.4660 - accurac
```

```
Epoch 13/50
83/83 [==============================] - 68s 744ms/step - loss: 0.4503 - accurac
Epoch 14/50
83/83 [==============================] - 69s 766ms/step - loss: 0.4796 - accurac
Epoch 15/50
83/83 [==============================] - 69s 757ms/step - loss: 0.4338 - accurac
Epoch 16/50
83/83 [==============================] - 69s 763ms/step - loss: 0.3859 - accurac
Epoch 17/50
83/83 [==============================] - 71s 781ms/step - loss: 0.3487 - accurac
Epoch 18/50
83/83 [==============================] - 68s 747ms/step - loss: 0.2876 - accurac
Epoch 19/50
83/83 [==============================] - 68s 754ms/step - loss: 0.3202 - accurac
Epoch 20/50
83/83 [==============================] - 70s 772ms/step - loss: 0.3956 - accurac
Epoch 21/50
83/83 [==============================] - 65s 708ms/step - loss: 0.2890 - accurac
Epoch 22/50
83/83 [==============================] - 65s 716ms/step - loss: 0.3251 - accurac
Epoch 23/50
83/83 [==============================] - 70s 762ms/step - loss: 0.2763 - accurac
Epoch 24/50
83/83 [==============================] - 69s 760ms/step - loss: 0.3304 - accurac
Epoch 25/50
83/83 [==============================] - 69s 763ms/step - loss: 0.2737 - accurac
Epoch 26/50
83/83 [==============================] - 69s 769ms/step - loss: 0.2629 - accurac
Epoch 27/50
83/83 [==============================] - 68s 754ms/step - loss: 0.2416 - accurac
```

The highest accuracy for the testing set has been reached at the 22nd epoch at 0.9600, or 96%, and didn't improve after that.

With the `history` object, we can plot two lineplots showing both the loss function and accuracy for both sets over epochs.

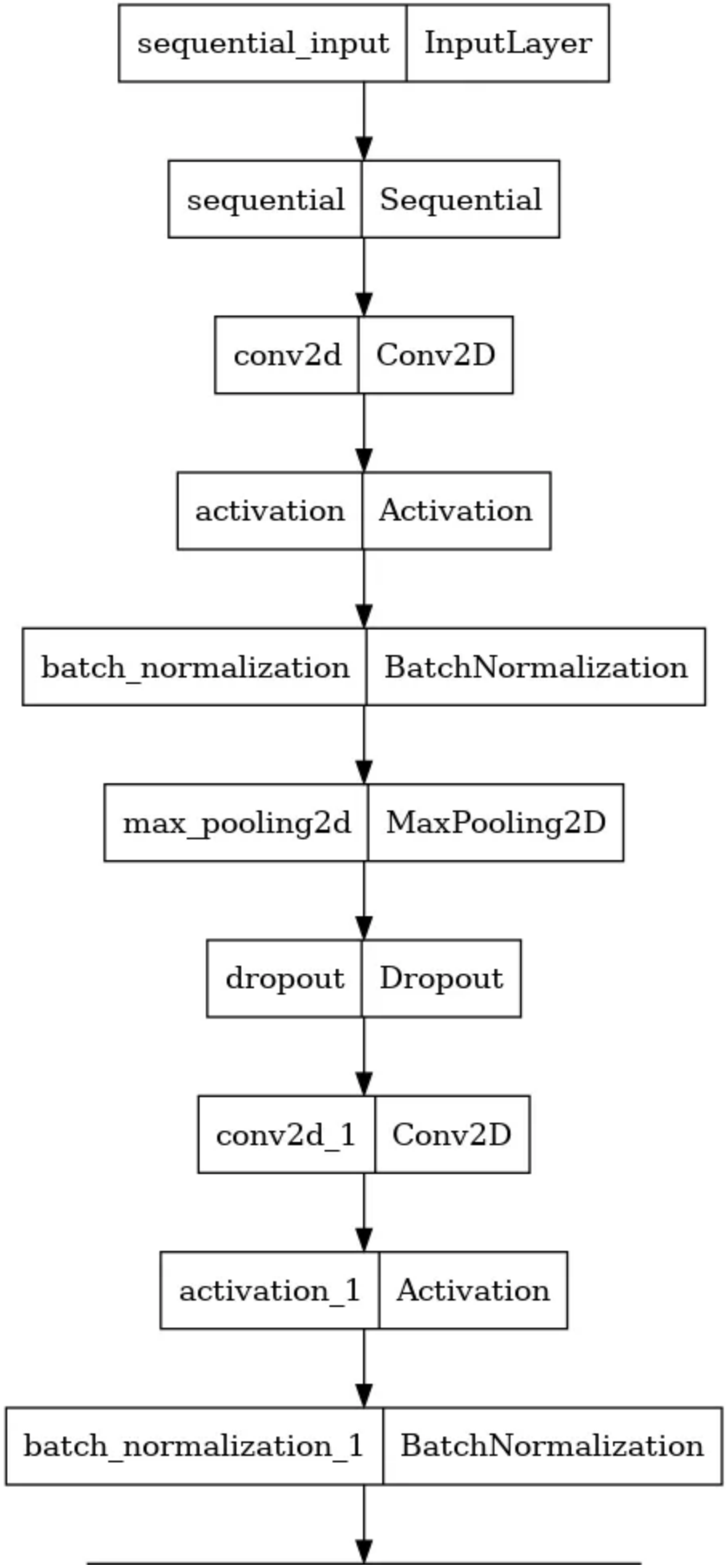


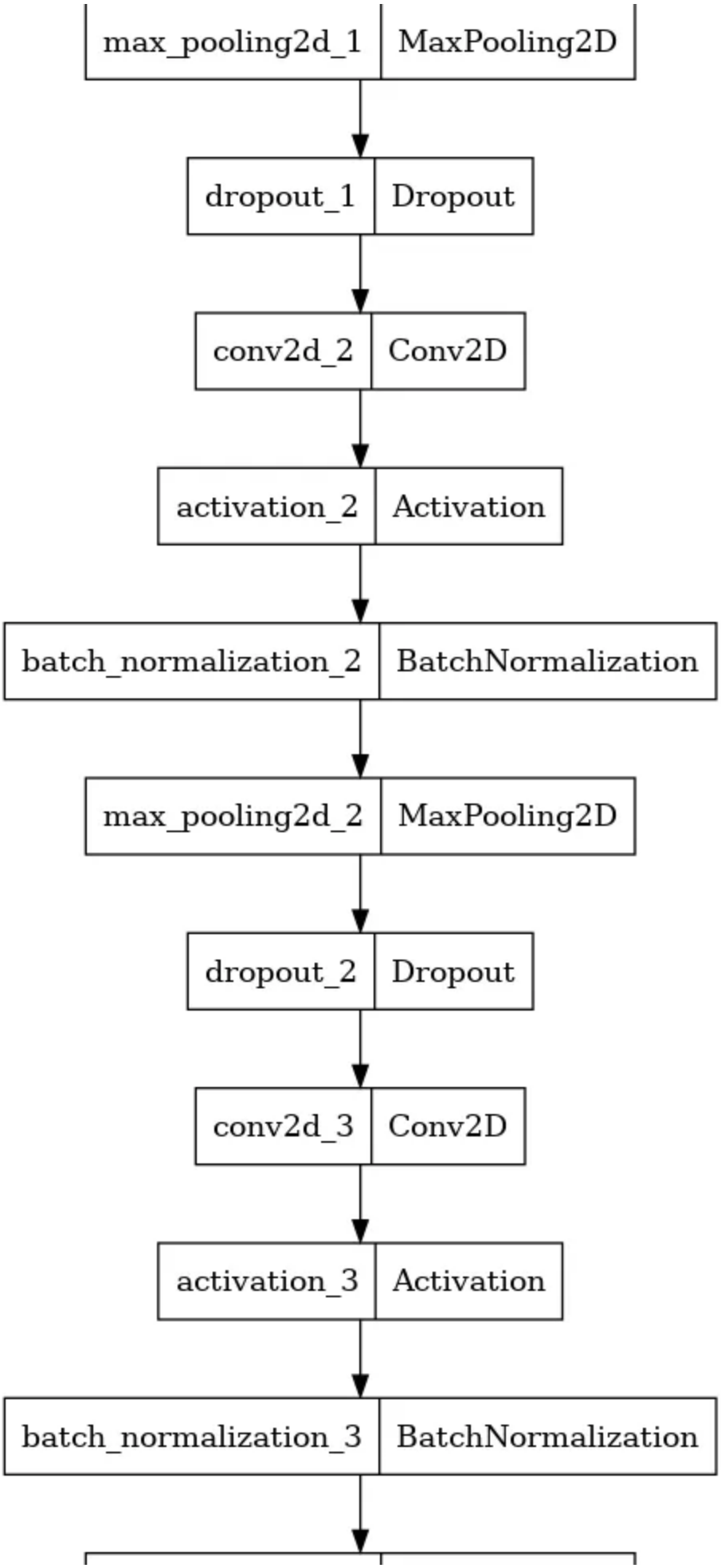
Loss and Accuracy over Epochs for Both Sets

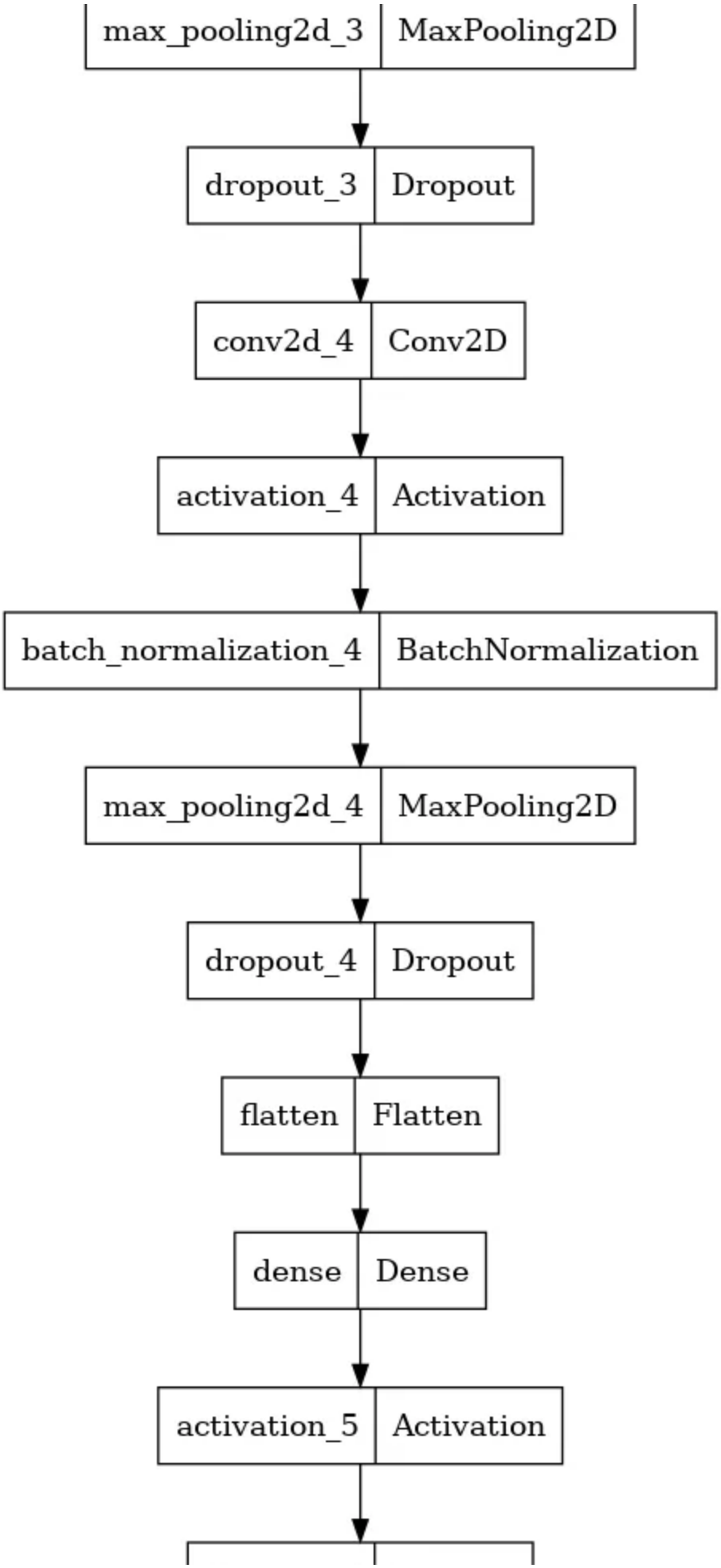
It is possible to see that the loss of the training set decreases continuously over epochs, whereas its accuracy increases. This happens because, at each epoch, the model starts to become more and more aware of the training set's patterns and particularities.

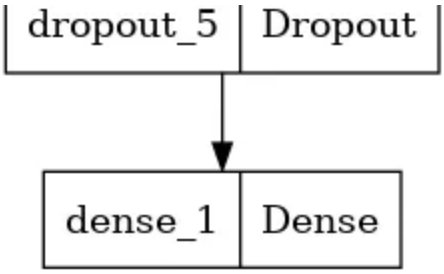
For the test set, however, this process is a bit more slower. Overall, the lowest loss for the test set happened at epoch 14 at 0.5319, while the accuracy was at its peak at epoch 22, at 0.9600.

Now that our model is built, trained, and tested, we can also plot its architecture, as well as summary to better understand it.


sequential_input
InputLayer
sequential
Sequential
conv2d
Conv2D
activation
Activation
batch_normalization
BatchNormalization
max_pooling2d
MaxPooling2D
dropout
Dropout
conv2d_1
Conv2D
activation_1
Activation
batch_normalization_1
BatchNormalization


max_pooling2d_1 MaxPooling2D
dropout_1 Dropout
conv2d_2 Conv2D
activation_2 Activation
batch_normalization_2 BatchNormalization
max_pooling2d_2 MaxPooling2D
dropout_2 Dropout
conv2d_3 Conv2D
activation_3 Activation
batch_normalization_3 BatchNormalization


max_pooling2d_3 MaxPooling2D
dropout_3 Dropout
conv2d_4 Conv2D
activation_4 Activation
batch_normalization_4 BatchNormalization
max_pooling2d_4 MaxPooling2D
dropout_4 Dropout
flatten Flatten
dense Dense
activation_5 Activation



Architecture of the CNN we've built

In the image, it is possible to visualize the sequential process of the Convolutional Neural Network. First we have a 2D Convolutional Layer, with ***ReLU*** activation function, followed by a BatchNormalization Layer and then a MaxPooling 2D Layer. Finally, we have a Dropout Layer to avoid overfitting. This same pattern repeats a few times until we reach the Flatten Layer, which connects the output of the Feature Learning process to the Dense Layers for the final classification task.

Using `model.summary()`, we can extract some extra info on the neural network.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential (Sequential)     (None, 256, 256, 3)       0

 conv2d (Conv2D)             (None, 256, 256, 32)      896

 activation (Activation)     (None, 256, 256, 32)      0

 batch_normalization (BatchN  (None, 256, 256, 32)     128
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 128, 128, 32)     0
 )

 dropout (Dropout)           (None, 128, 128, 32)      0

 conv2d_1 (Conv2D)           (None, 128, 128, 64)      51264
```

```
activation_1 (Activation)   (None, 128, 128, 64)      0

batch_normalization_1 (Batc  (None, 128, 128, 64)     256
hNormalization)

max_pooling2d_1 (MaxPooling  (None, 64, 64, 64)       0
2D)

dropout_1 (Dropout)         (None, 64, 64, 64)        0

conv2d_2 (Conv2D)           (None, 64, 64, 128)       73856

activation_2 (Activation)   (None, 64, 64, 128)       0

batch_normalization_2 (Batc  (None, 64, 64, 128)      512
hNormalization)

max_pooling2d_2 (MaxPooling  (None, 32, 32, 128)      0
2D)

dropout_2 (Dropout)         (None, 32, 32, 128)       0

conv2d_3 (Conv2D)           (None, 32, 32, 256)       819456

activation_3 (Activation)   (None, 32, 32, 256)       0

batch_normalization_3 (Batc  (None, 32, 32, 256)      1024
hNormalization)

max_pooling2d_3 (MaxPooling  (None, 16, 16, 256)      0
2D)

dropout_3 (Dropout)         (None, 16, 16, 256)       0

conv2d_4 (Conv2D)           (None, 16, 16, 512)       1180160

activation_4 (Activation)   (None, 16, 16, 512)       0

batch_normalization_4 (Batc  (None, 16, 16, 512)      2048
hNormalization)

max_pooling2d_4 (MaxPooling  (None, 8, 8, 512)        0
2D)

dropout_4 (Dropout)         (None, 8, 8, 512)         0

flatten (Flatten)           (None, 32768)             0

dense (Dense)               (None, 2048)              67110912
```

```
 activation_5 (Activation)   (None, 2048)              0

 dropout_5 (Dropout)         (None, 2048)              0

 dense_1 (Dense)             (None, 3)                 6147

=================================================================
Total params: 69,246,659
Trainable params: 69,244,675
Non-trainable params: 1,984
_________________________________________________________________
```

The summary displays the output shapes for each layer, as well as the number of parameters. We can clearly see, for instance, that the output shape for the first layer is `(None, 256,256,3)` where 256256 represents both height and width, while 33 represents the RGB color. In the last dense layer, however, the output shape is `(None, 3)`, where 33 represents the three classes for classification.

We can also see that the model has over 69 million parameters, where 99.99% of them are trainable. The non-trainable parameters are the ones from the BatchNormalization layers.

Validating Performance

After finishing the training and testing phase, we may go ahead and validate our model on the validation set. To load the best weights achieved during training, we simply use the `load_weights` method. These weights will be saved with the same name we've given during the `ModelCheckpoint` configuration, when we set `ModelCheckpoint('best_model.h5')`.

```
# Loading best weights
```

```
model.load_weights('best_model.h5')
```

```
preds = model.predict(validation)  # Running model on the validation dataset
val_loss, val_acc = model.evaluate(validation) # Obtaining Loss and Accuracy on

print('\nValidation Loss: ', val_loss)
print('\nValidation Accuracy: ', np.round(val_acc * 100), '%')
```

```
4/4 [==============================] - 3s 108ms/step
4/4 [==============================] - 3s 14ms/step - loss: 0.7377 - accuracy: 0

Validation Loss:  0.7376963496208191

Validation Accuracy:  97.0 %
```

The output for `model.predict()` consists of probabilities for each class, while `model.evaluate()` returns loss and accuracy values.

It is clear that the model correctly predicts *97%* of the labels of the images in the validation set.

I am going to load some images from the validation test and run predictions on them individually, so we can see how the model performs according to each picture.

```
Picture of a Powdery Plant:
```

```
1/1 [==============================] - 0s 283ms/step

Predicted Class: Powdery

Confidence Score: 0.9999980926513672
```

The model is about 99.9% confident that the plant in the picture belongs to the ***Powdery*** class, which is correct.

```
Picture of a Rust Plant:
```

```
1/1 [==============================] - 0s 71ms/step

Predicted Class: Rust

Confidence Score: 1.0
```

The model is 100% certain that the plant in the picture belongs to the ***Rust*** class, which is also correct.

```
Picture of a Healthy Plant:
```

```
1/1 [==============================] - 0s 64ms/step

Predicted Class: Healthy

Confidence Score: 1.0
```

The model is 100% certain that the plant in the picture belongs to the ***Healthy*** class, which is also correct.

After running several tests with other pictures, I could identify that the current model is performing fairly well in classifying all the three classes.

To save the current weights, so you can deploy this model or continue working with it later on, you can simply use Keras' `.save()` method. This is

going to save your model as an HDF5 file.

```
model.save('plant_disease_classifier.h5') # Saving model
```

Conclusion

In this article, we explored the basics of Convolutional Neural Networks. We delved deeper into the main layers — Convolutional, Pooling, etc. — , activation functions, as well as many other techniques to work with image data and CNNs for image classification.

Even though many tasks nowadays can be efficiently done with pre-trained models, that can be easily accessible via platforms such as TensorFlow Hub and HuggingFace, it is still essential to understand what is the role of each layer inside a Convolutional Neural Network and how they interact with each other. This is why this notebook have the intention of guiding you through the process of building a CNN from scratch, and I plan to bring more notebooks such as this one for other *Deep Learning* tasks and architectures.

Our model scored 97.0% in accuracy while predicting labels for the validation dataset, which is a great performance, and it was competent to identify relevant patterns across all the classes in the dataset.

I hope that this notebook serves as an introduction to those that are still just starting to explore ConvNets, or even help veterans to refine their knowledge on some of the basics. Please, feel free to copy this notebook and edit it as you wish, specially to try your own improvements for higher performance and testings.

Thank you so much for reading. Your feedback, upvotes, and suggestions are always much welcome!

Luis Fernando Torres

Let's connect! 🔗
LinkedIn • Kaggle • HuggingFace

Convolutional Network | Deep Learning | Image Processing | Machine Learning | AI

Written by Luís Fernando Torres

Follow

853 Followers

Data Scientist | Machine Learning | Commodities Trader & Investor |
https://luuisotorres.github.io/

More from Luís Fernando Torres

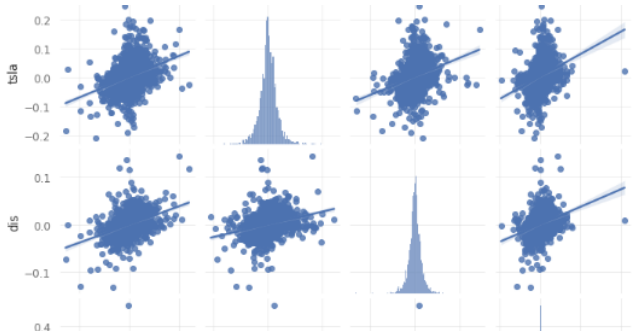
Luís Fernando Torres in LatinXinAI

How I Deployed a Machine Learning Model for the First Time

Going beyond Jupyter Notebooks 🚀

13 min read · Jul 18

778 7



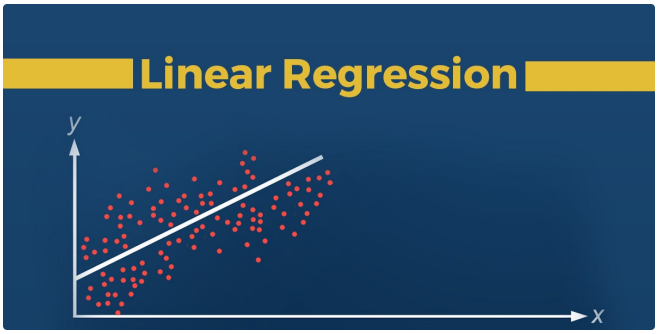
Luís Fernando Torres in InsiderFinance Wire

Introduction to Quant Investing with Python

Introduction

15 min read · Mar 30

990 11



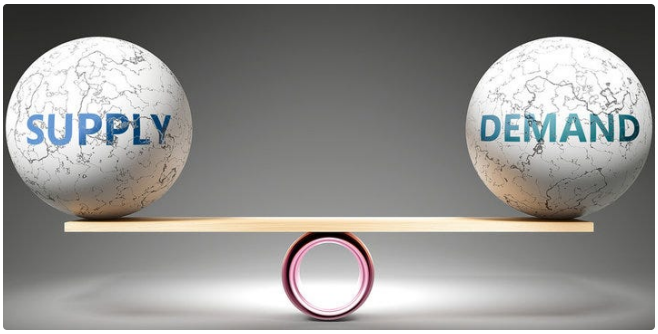
Luís Fernando Torres in LatinXinAI

Mastering Linear Regression with Statsmodels

Note: This article is based on my Kaggle Notebook: 📒 📈 Mastering Linear...

16 min read · Sep 17

158



Luís Fernando Torres in InsiderFinance Wire

Volatility-Based Supply & Demand Levels Forecasting

Introduction

9 min read · Sep 21

143

See all from Luís Fernando Torres