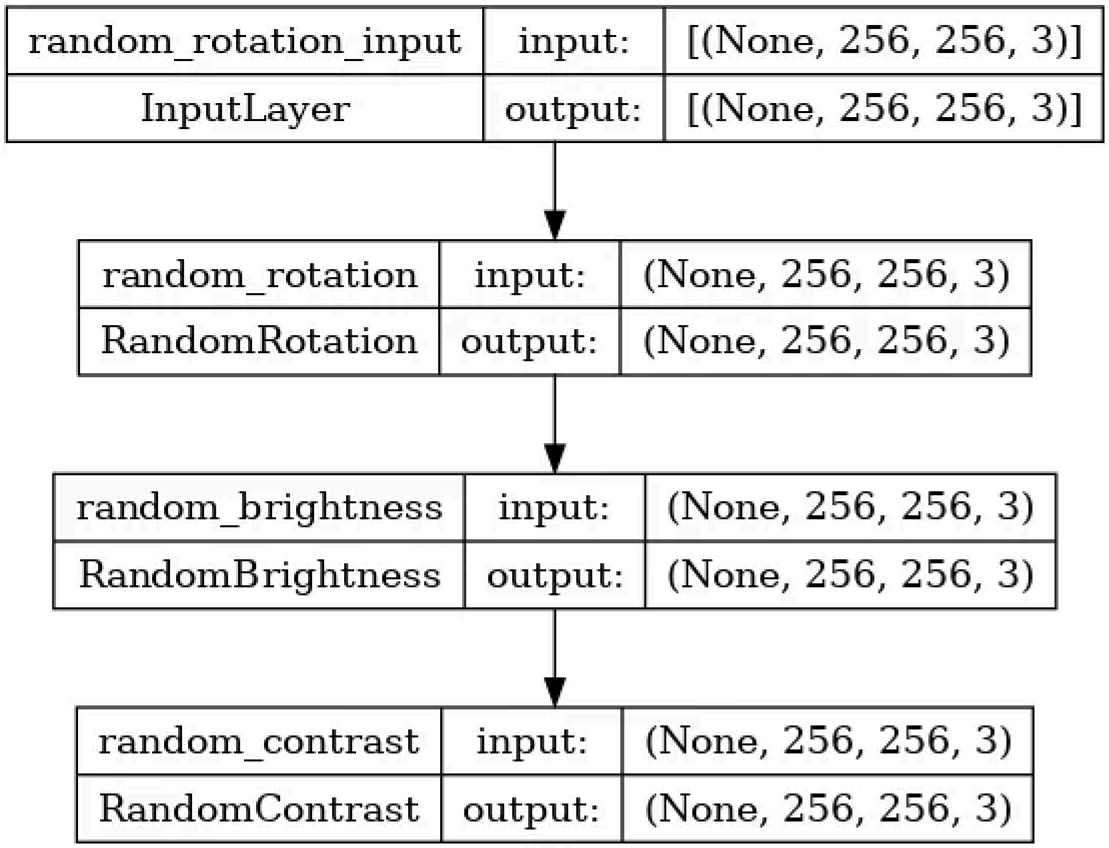

random_rotation_input | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
random_rotation | input: | (None, 256, 256, 3)
RandomRotation | output: | (None, 256, 256, 3)
random_brightness | input: | (None, 256, 256, 3)
RandomBrightness | output: | (None, 256, 256, 3)
random_contrast | input: | (None, 256, 256, 3)
RandomContrast | output: | (None, 256, 256, 3)

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)
```

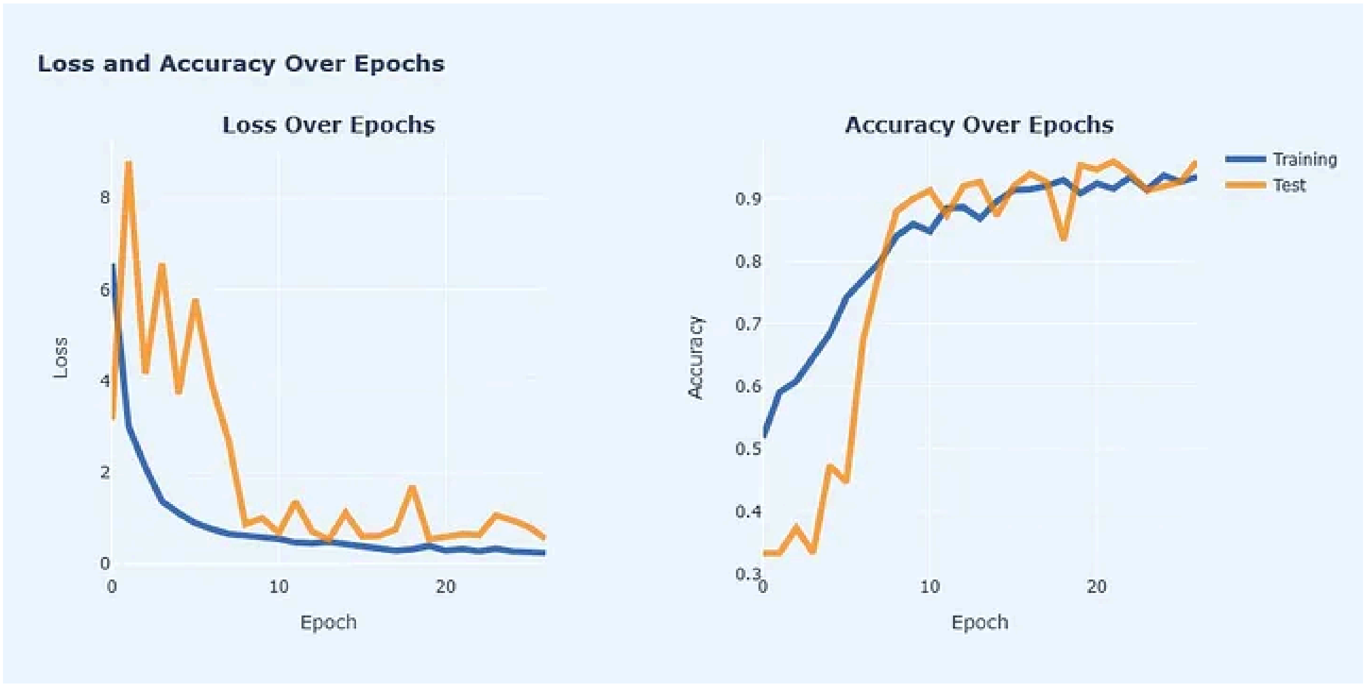
We may now use `model.fit()` to start the training and testing process.

```
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.

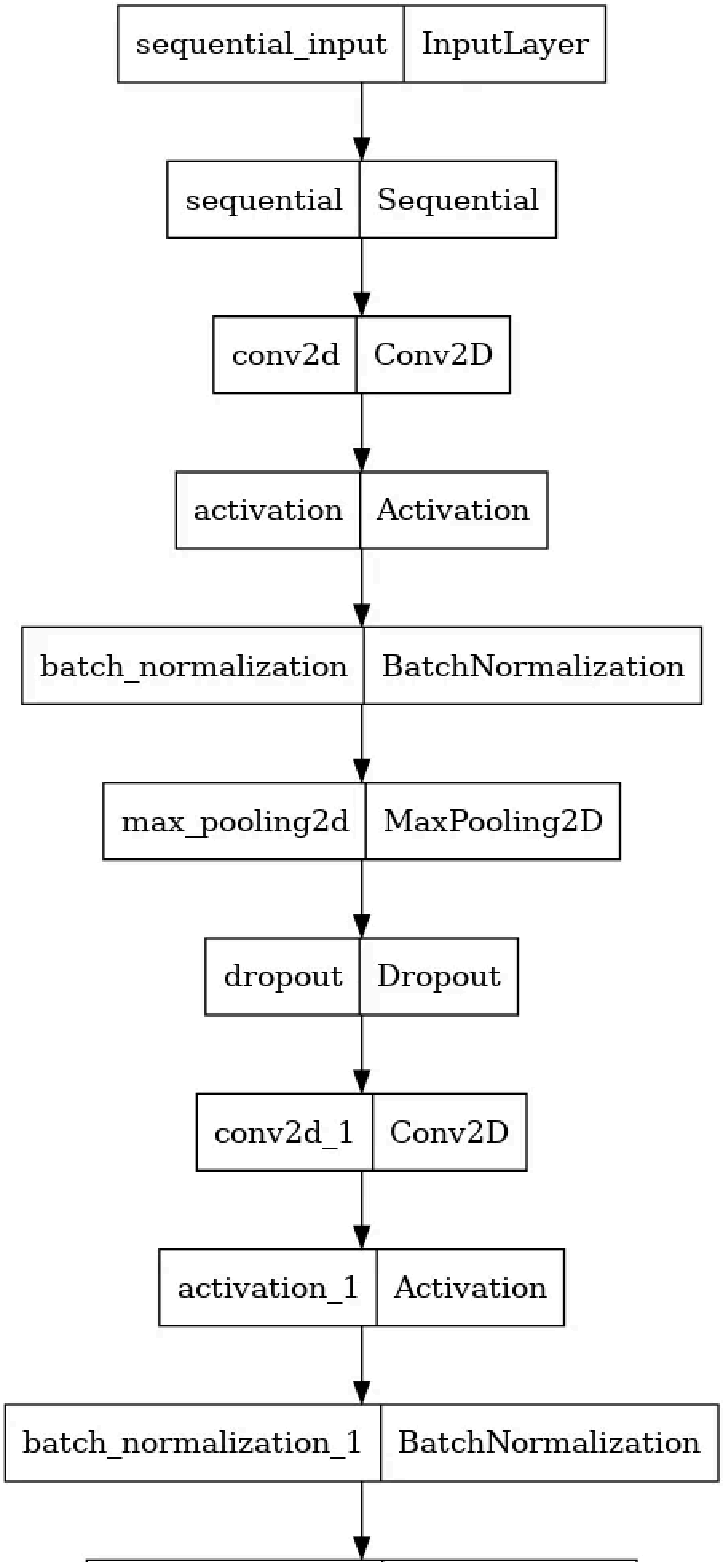


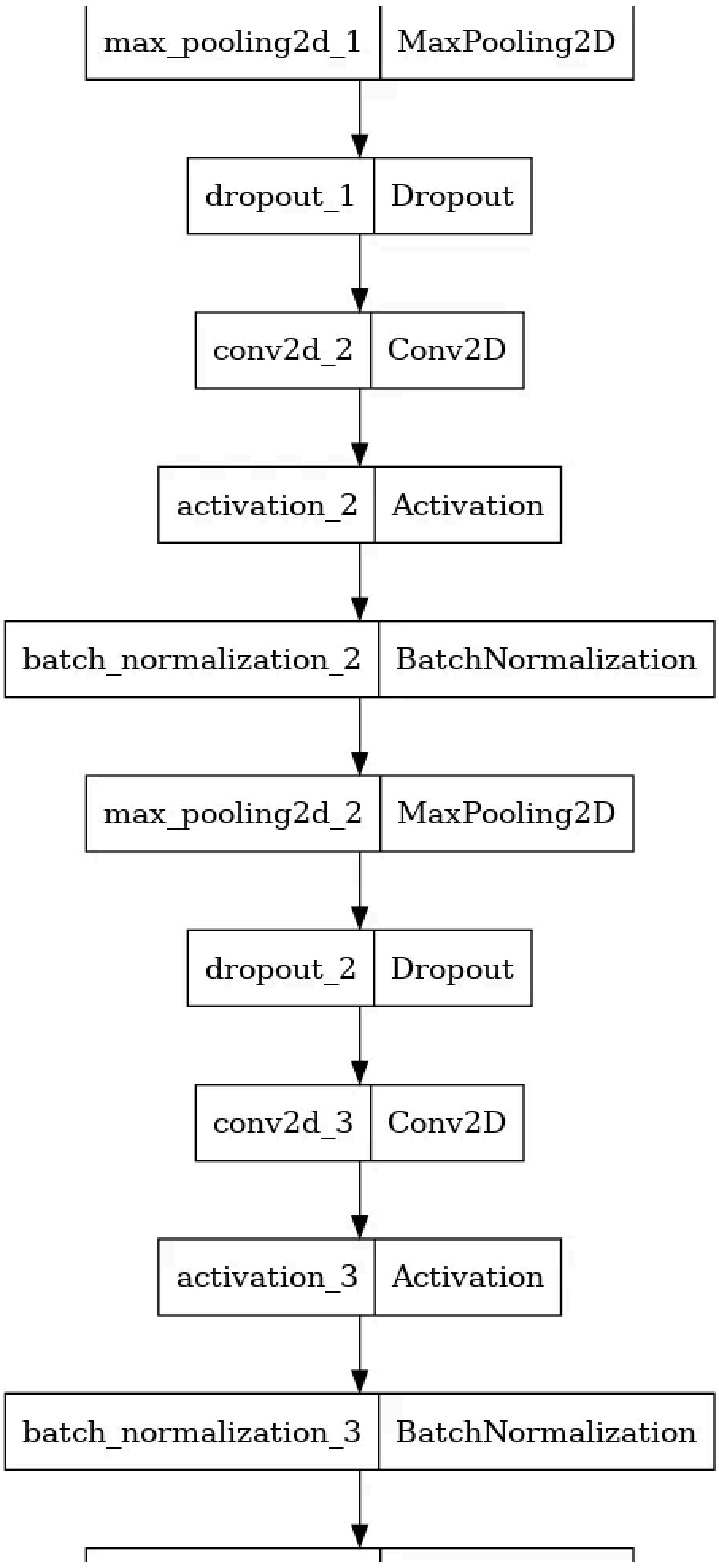
Loss and Accuracy over Epochs for Both Sets

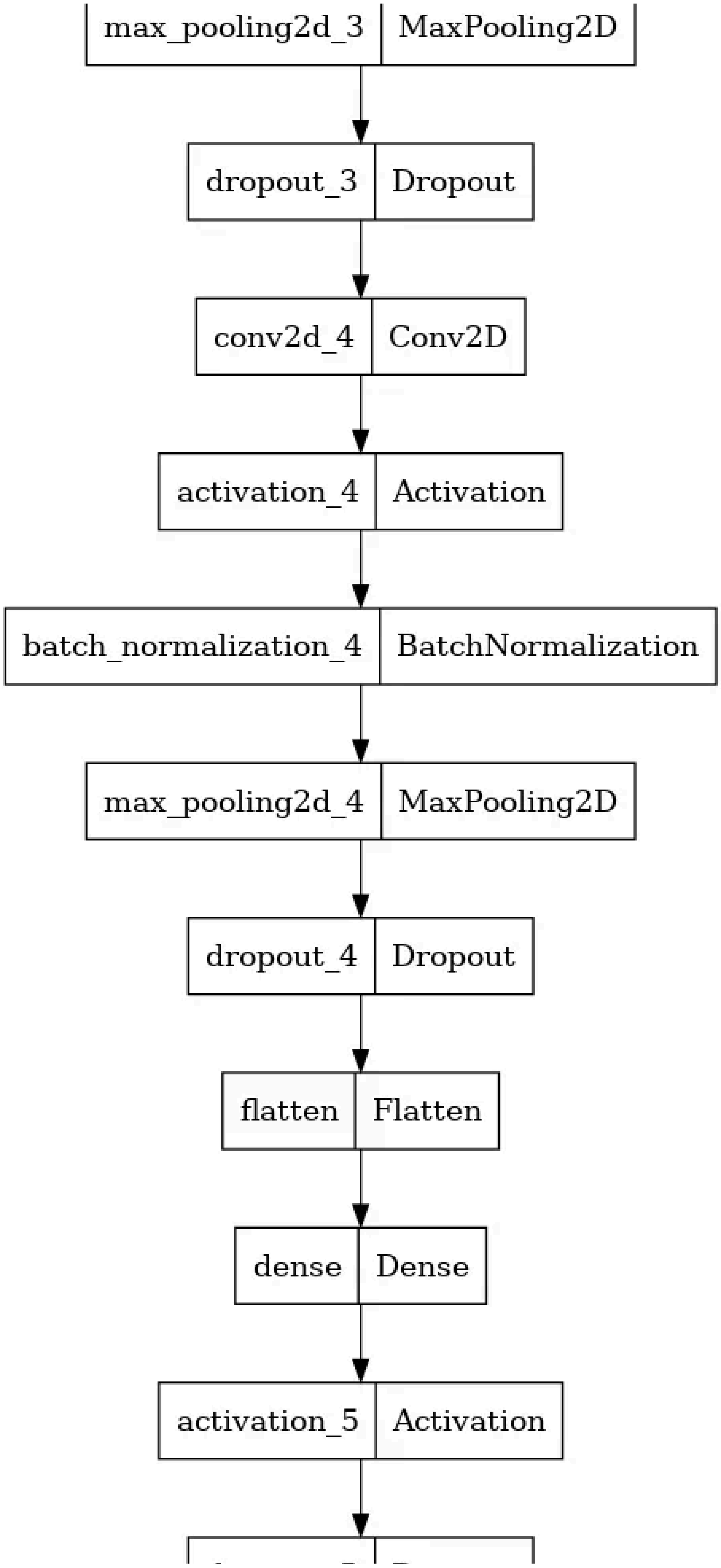
It is possible to see that the loss of the training set decreases continuously over epochs, whereas its accuracy increases. This happens because, at each epoch, the model starts to become more and more aware of the training set's patterns and particularities.

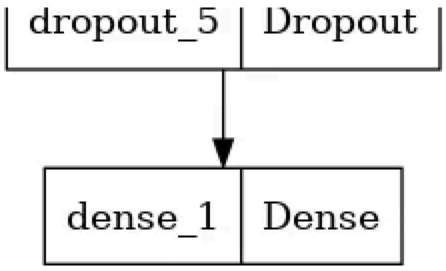
For the test set, however, this process is a bit more slower. Overall, the lowest loss for the test set happened at epoch 14 at 0.5319, while the accuracy was at its peak at epoch 22, at 0.9600.

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
```