Convolutional Neural Networks and Transfer Learning for Bird Images Classification

Deep Learning Project

Tzanis Nikolaos
mtn2217

Contents

Introduction

In the vast world of birds, our ability to identify the species of a bird based solely on its external characteristics has always been a significant challenge.

However, the progress of technology, particularly advancements in computer vision and deep learning, has brought us one step closer to achieving this goal. In fact, a significant breakthrough, especially in image classification, has been made with the application of Convolutional Neural Networks (CNN). By training these networks on datasets containing a large number of bird images, we can teach them to recognize specific features of birds that differentiate one species from another.

However, training CNNs on massive databases requires a considerable amount of time and computational power, making it impractical in many cases for developing such models.

The solution to this problem lies in using other neural networks that have already been trained on similar classification tasks. Neural networks such as ResNet, through the process of transfer learning, can help us quickly achieve high accuracy rates by utilizing the prior knowledge of these models.

In this exercise, I develop and optimize a CNN model for classifying a dataset containing approximately 80,000 bird images across 525 distinct bird species. Subsequently, I employ the model with transfer learning for classifying another dataset consisting of around 30,000 images of 25 bird species from India.

In the realm of transfer learning, I experiment with different techniques and eventually arrive at some interesting conclusions.

Necessary Libraries and Datasets

Libraries

The main libraries and modules used in this essay were:

- **Tensorflow:** A library that is commonly used for machine learning and deep learning tasks that provides a lot of tools for the design and training of neural networks.
- **Keras:** A module within TensorFlow that provides a high-level API for building and training neural networks.
- **Time:** This library is used to calculate the training time of the various models tested.
- **Matplotlib:** A library for creating visualizations and plots, which helped a lot to analyze and display the training results of the aforementioned models.

Datasets

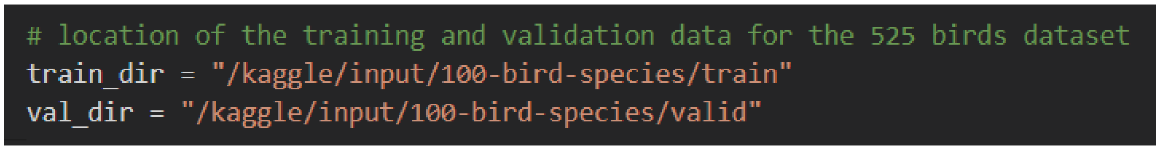
The two datasets used in this essay come from Kaggle.com and the get imported directly through the platform of that website for running machine learning and deep learning code.

It is important to note that both jupyter notebooks included in this essay require GPU usage, which is provided from Kaggle.

As a result, in order to run the notebooks, I recommend using Kaggle's code website while also opting to use the datasets directly from that website without downloading them locally.

Specifically, after uploading the notebook to Kaggle Code, there is an option on the right menu named "Add Data" that allows the user to add the dataset that will be used in the jupyter notebook.

The following code can make the datasets available to the notebook:



```
# location of the training and validation data for the 525 birds dataset
train_dir = "/kaggle/input/100-bird-species/train"
val_dir = "/kaggle/input/100-bird-species/valid"
```

General Information About the Datasets

The two datasets used in this essay were:

1. https://www.kaggle.com/datasets/gpiosenka/100-bird-species
2. https://www.kaggle.com/datasets/ichhadhari/indian-birds

The first dataset contains 84.635 bird images for the training set, divided into 525 classes (bird species).

The second dataset contains images of birds from India. In total, there are 37.000 images divided into 25 classes.

It is important to note that data augmentation has already been applied in the second dataset as a significant number of images are flipped (horizontally or vertically), tilted or subjected to other forms of augmentation. As a result, this dataset contains around 1400 images per class on the training dataset.

I need to note at this point that I considered applying data augmentation on the first dataset too, but after some experiments I realized that this further increases the size of an already huge dataset, thus increasing exponentially the training time. As a result I decided to keep the original dataset as is, and didn’t apply any data augmentation.

Creating the Training and Validation Sets and Exploring the Datasets

Parameters

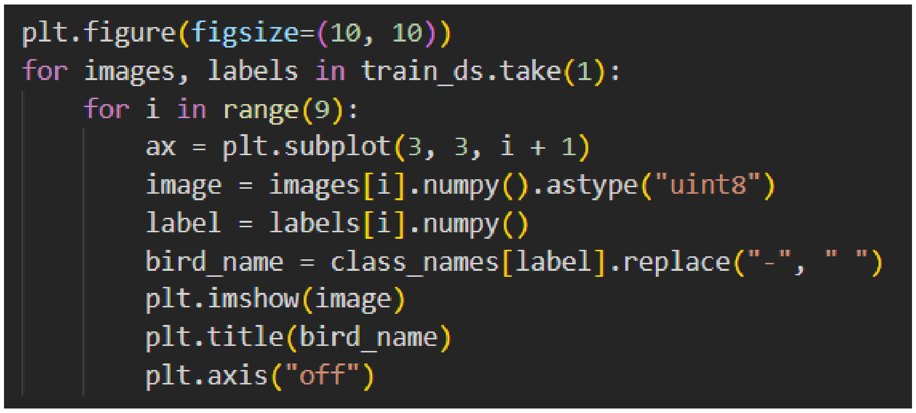
The photos used for the training and validation sets have been resized to 180x180 size, while the batch size is set to 64. While creating the training and validation sets with the image_dataset_from_directory function from keras a seed is also set, so that the results can be reproduced in the future.

Exploration of the Classes we’re Working with and the Images of the First Dataset

This dataset, as mentioned before, contains 84,635 photos in the training set and 2,625 photos in the validation set.

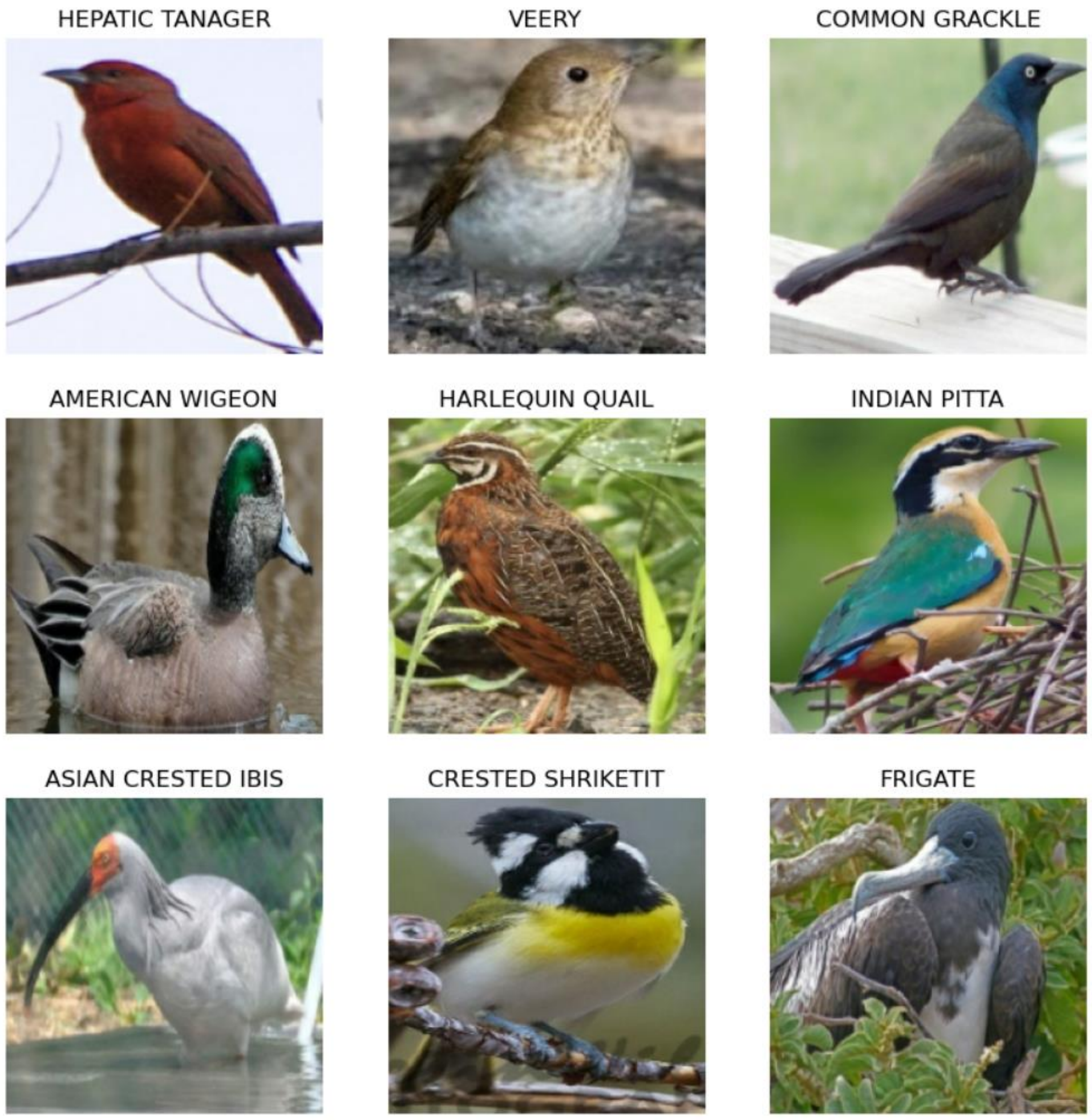
Practically, there are approximately 140-150 photos for each bird species. These photos have not undergone data augmentation and are all fundamentally different from each other. Ideally, data augmentation could have been applied for further optimization of the later-trained model. However, this approach significantly increased the training time, so as I mentioned earlier, I chose to use the original dataset without augmentation.

The following code snippet was used to extract 9 random photos from the training dataset:



```
plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        image = images[i].numpy().astype("uint8")
        label = labels[i].numpy()
        bird_name = class_names[label].replace("-", " ")
        plt.imshow(image)
        plt.title(bird_name)
        plt.axis("off")
```

Below we can see some of the sample images (the bird species is shown above each image):

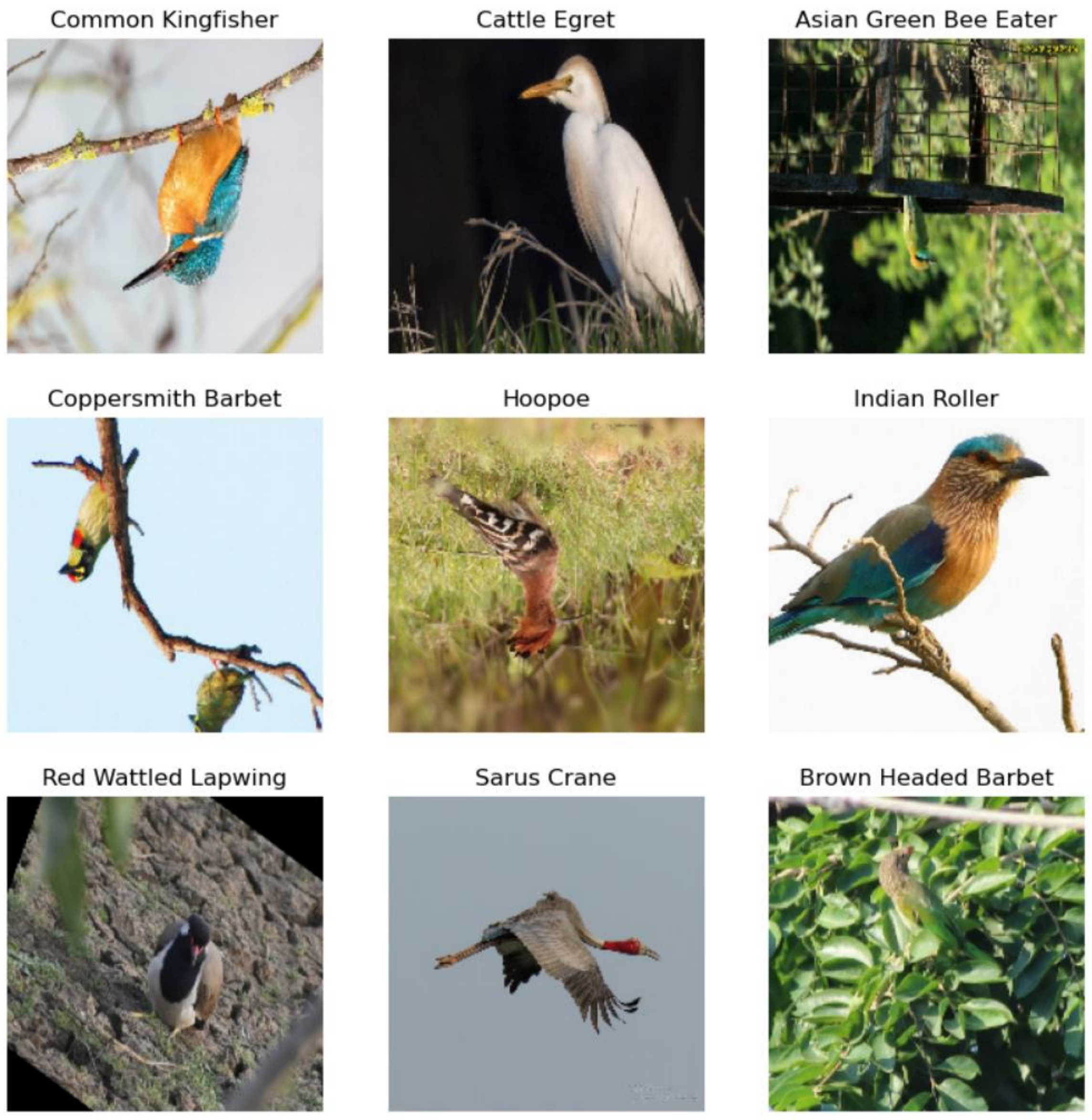


As we can see none of the above images has any indications that it could be the result of data augmentation.

Exploration of the Classes we're Working with and the Images of the Second (Indian) Dataset

This dataset consists of a total of 30,000 images in the training set and 7,000 images in the validation set, representing 25 different bird species from India. Practically, there are approximately 1,400 images per bird species, although many of these images are the result of data augmentation.

Several random images from this dataset are seen below:

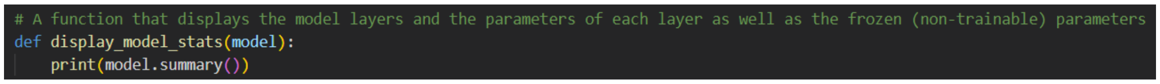


As we can see, some of these images are flipped horizontally or vertically or tilted. All of the above are practices used in data augmentation.

Classification Models

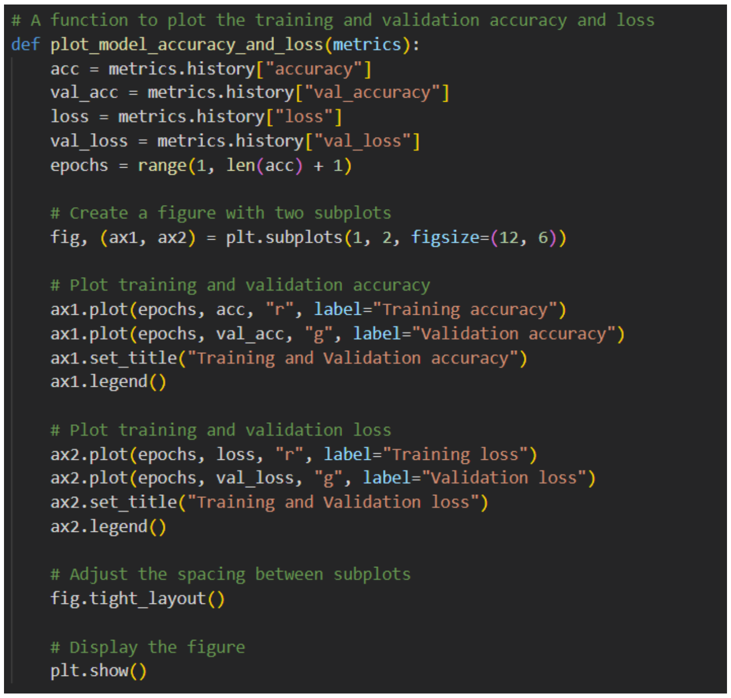
Helper Functions

For each of the neural network models examined, the following helper functions were used to facilitate various operations.



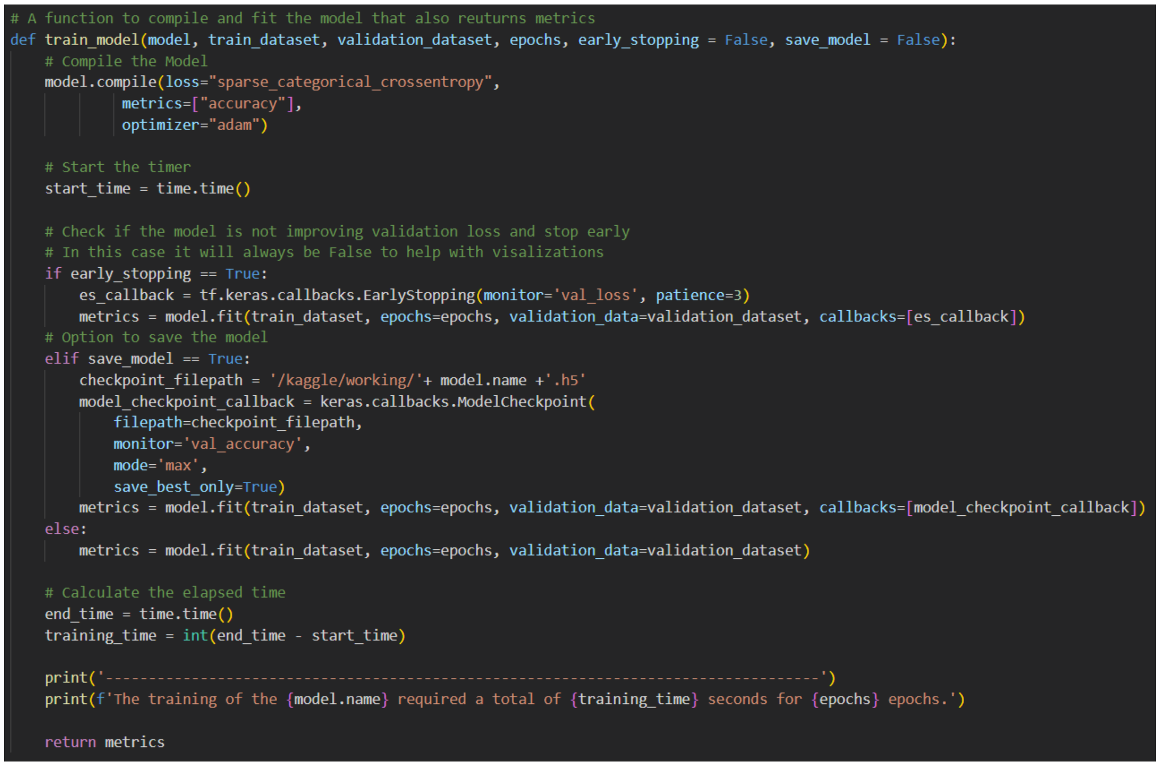
```python
# A function that displays the model layers and the parameters of each layer as well as the frozen (non-trainable) parameters
def display_model_stats(model):
    print(model.summary())
```

The above function returns the structure of the neural network and prints the number of parameters in each layer. It also shows the total number of parameters in the model and how many of them are trainable.



```python
# A function to plot the training and validation accuracy and loss
def plot_model_accuracy_and_loss(metrics):
    acc = metrics.history["accuracy"]
    val_acc = metrics.history["val_accuracy"]
    loss = metrics.history["loss"]
    val_loss = metrics.history["val_loss"]
    epochs = range(1, len(acc) + 1)

    # Create a figure with two subplots
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

    # Plot training and validation accuracy
    ax1.plot(epochs, acc, "r", label="Training accuracy")
    ax1.plot(epochs, val_acc, "g", label="Validation accuracy")
    ax1.set_title("Training and Validation accuracy")
    ax1.legend()

    # Plot training and validation loss
    ax2.plot(epochs, loss, "r", label="Training loss")
    ax2.plot(epochs, val_loss, "g", label="Validation loss")
    ax2.set_title("Training and Validation loss")
    ax2.legend()

    # Adjust the spacing between subplots
    fig.tight_layout()

    # Display the figure
    plt.show()
```

This function displays two plots: the training and validation accuracy, as well as the training and validation loss per training epoch of the neural network.



```python
# A function to compile and fit the model that also reuturns metrics
def train_model(model, train_dataset, validation_dataset, epochs, early_stopping = False, save_model = False):
    # Compile the Model
    model.compile(loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"],
                  optimizer="adam")

    # Start the timer
    start_time = time.time()

    # Check if the model is not improving validation loss and stop early
    # In this case it will always be False to help with visalizations
    if early_stopping == True:
        es_callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)
        metrics = model.fit(train_dataset, epochs=epochs, validation_data=validation_dataset, callbacks=[es_callback])
    # Option to save the model
    elif save_model == True:
        checkpoint_filepath = '/kaggle/working/'+ model.name +'.h5'
        model_checkpoint_callback = keras.callbacks.ModelCheckpoint(
            filepath=checkpoint_filepath,
            monitor='val_accuracy',
            mode='max',
            save_best_only=True)
        metrics = model.fit(train_dataset, epochs=epochs, validation_data=validation_dataset, callbacks=[model_checkpoint_callback])
    else:
        metrics = model.fit(train_dataset, epochs=epochs, validation_data=validation_dataset)

    # Calculate the elapsed time
    end_time = time.time()
    training_time = int(end_time - start_time)

    print('-----------------------------------------------------------------------------------')
    print(f'The training of the {model.name} required a total of {training_time} seconds for {epochs} epochs.')

    return metrics
```

The last helper function compiles the model using "sparse_categorical_crossentropy" as the loss function, which was preferred over "categorical_crossentropy" because the former is better suited for classification problems with multiple distinct classes.

The optimizer used was Adam, and during the training process, we want to track the accuracy (both training and validation).

Following that, a timer is started, which will ultimately measure the time taken for training the neural network (in seconds).

The function includes an option for early stopping if there is no improvement, but I never used this option because I wanted it to be clear in the plots what was happening with each neural network.

There is also an option to save the model.

Subsequently, the data is fitted to the model, the time taken for training is printed, and the function returns information from the training process, which is ultimately displayed using the previous function.

A Simple CNN Model with 4 Convolutional Layers

The first neural network I'm examining is a CNN with 4 convolutional layers.

In this network, the first layer is a Rescaling layer, which normalizes the images by dividing them by 255 to bring the values within the range [0,1]. This preprocessing step is quite common in image processing models.

After this initial layer, there are 4 convolutional layers, each followed by a 2D max pooling layer.

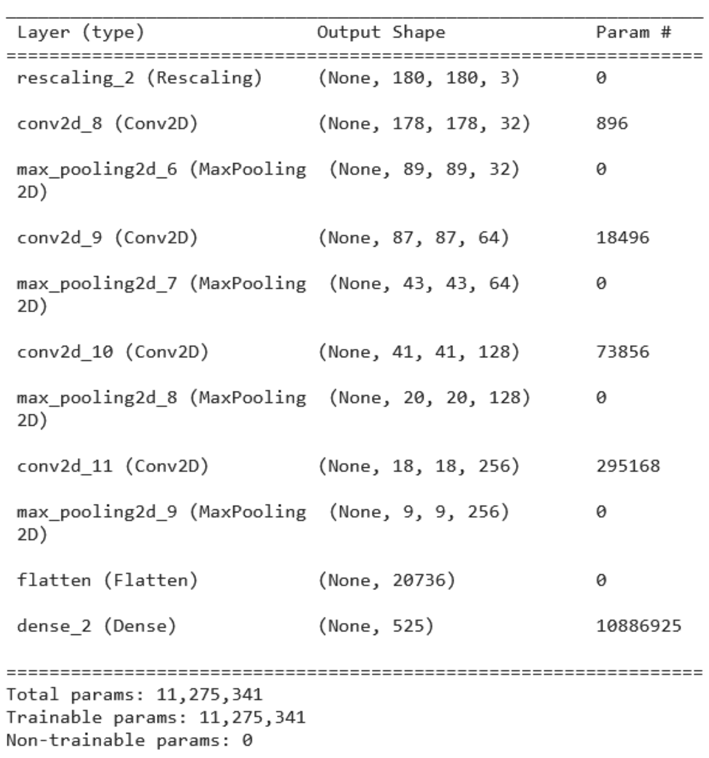
The first convolutional layer has 32 filters of size 3x3 and uses the ReLU activation function, allowing us to introduce non-linearity into the network.

The next 3 convolutional layers have progressively increasing numbers of filters (64, 128, and 256) to extract more complex features from the images.

Next, we add a Flatten layer, which effectively converts the output of the convolutional layers into a one-dimensional vector, preparing the data for the final (classification) layer.

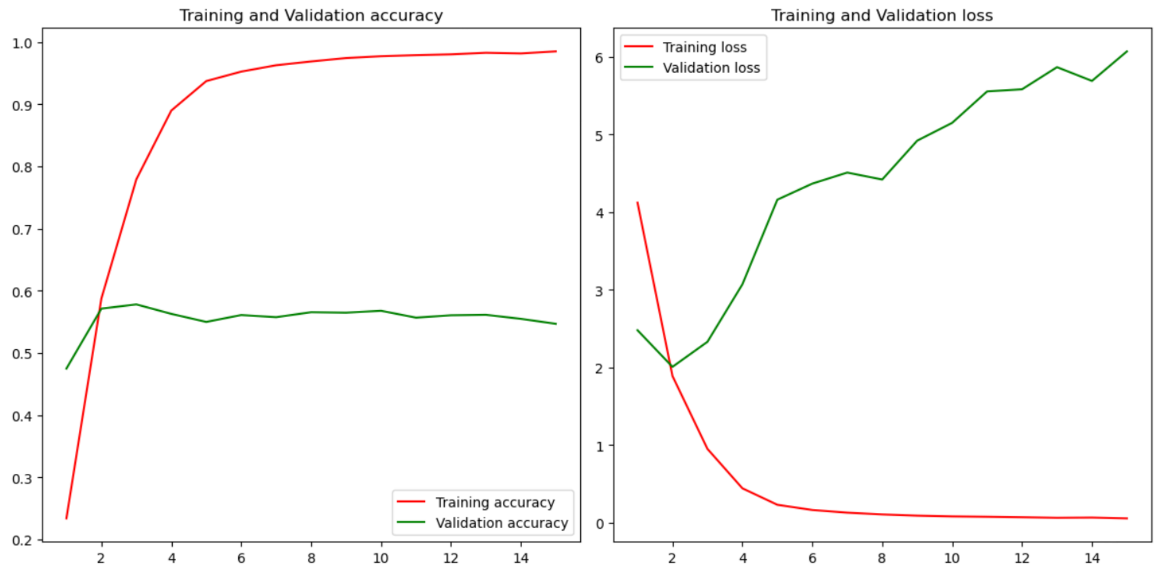
The classification layer we use is a Dense layer with 525 neurons, each corresponding to a class in our problem. The activation function of this layer is softmax, which allows us to obtain a probabilistic distribution of classes for each image.

The final architecture of the model is seen below:



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_2 (Rescaling)     (None, 180, 180, 3)       0

 conv2d_8 (Conv2D)           (None, 178, 178, 32)      896

 max_pooling2d_6 (MaxPooling  (None, 89, 89, 32)       0
 2D)

 conv2d_9 (Conv2D)           (None, 87, 87, 64)        18496

 max_pooling2d_7 (MaxPooling  (None, 43, 43, 64)       0
 2D)

 conv2d_10 (Conv2D)          (None, 41, 41, 128)       73856

 max_pooling2d_8 (MaxPooling  (None, 20, 20, 128)      0
 2D)

 conv2d_11 (Conv2D)          (None, 18, 18, 256)       295168

 max_pooling2d_9 (MaxPooling  (None, 9, 9, 256)        0
 2D)

 flatten (Flatten)           (None, 20736)             0

 dense_2 (Dense)             (None, 525)               10886925

=================================================================
Total params: 11,275,341
Trainable params: 11,275,341
Non-trainable params: 0
```

After training the model for 15 epochs we can see that it has clearly overfit the training data as the validation accuracy reaches a plateau after only 2 epochs while the training accuracy continues to increase and gets really close to perfect. The validation loss also increases, another indicator that the model doesn't generalize well.

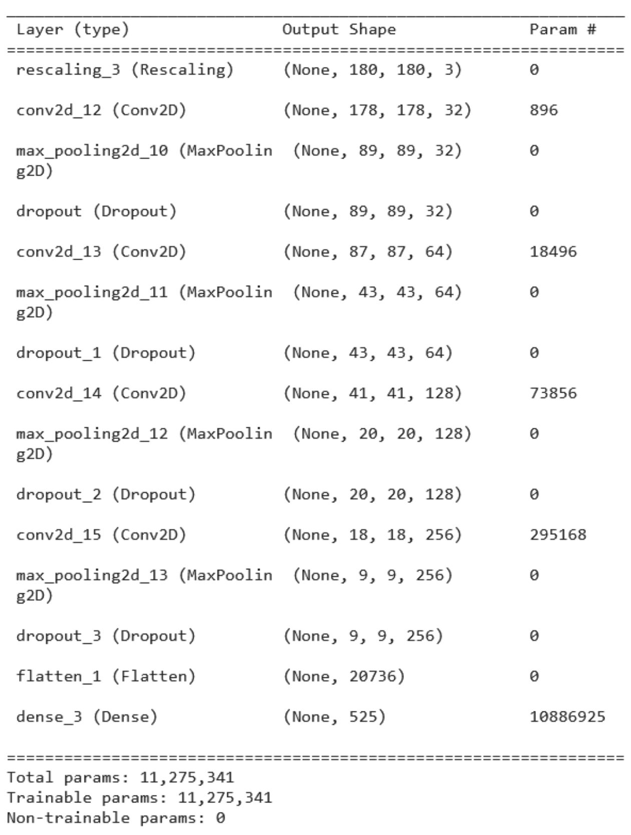


Adding Dropout Layers to Reduce Overfitting

The first approach to reduce overfitting is to add a Dropout layer after each colvolutional layer.

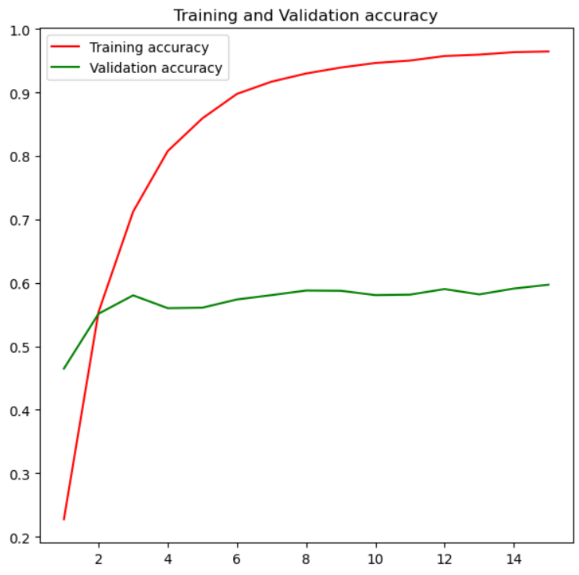
Dropout layers serve two primary purposes:

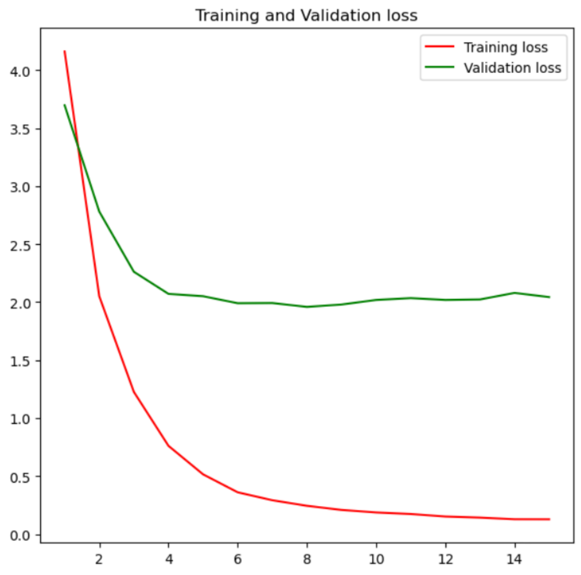
- First, by dropping out neurons, the model becomes less reliant on any single neuron or set of neurons with perceived greater value. This encourages the model to learn more robust and generalizable representations of the data.
- Second, dropout acts as a form of regularization by introducing noise and reducing co-adaptation between neurons, thereby preventing overfitting and improving the model's ability to generalize to unseen data.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 rescaling_3 (Rescaling)     (None, 180, 180, 3)       0         
                                                                 
 conv2d_12 (Conv2D)          (None, 178, 178, 32)      896       
                                                                 
 max_pooling2d_10 (MaxPoolin  (None, 89, 89, 32)       0         
 g2D)                                                            
                                                                 
 dropout (Dropout)           (None, 89, 89, 32)        0         
                                                                 
 conv2d_13 (Conv2D)          (None, 87, 87, 64)        18496     
                                                                 
 max_pooling2d_11 (MaxPoolin  (None, 43, 43, 64)       0         
 g2D)                                                            
                                                                 
 dropout_1 (Dropout)         (None, 43, 43, 64)        0         
                                                                 
 conv2d_14 (Conv2D)          (None, 41, 41, 128)       73856     
                                                                 
 max_pooling2d_12 (MaxPoolin  (None, 20, 20, 128)      0         
 g2D)                                                            
                                                                 
 dropout_2 (Dropout)         (None, 20, 20, 128)       0         
                                                                 
 conv2d_15 (Conv2D)          (None, 18, 18, 256)       295168    
                                                                 
 max_pooling2d_13 (MaxPoolin  (None, 9, 9, 256)        0         
 g2D)                                                            
                                                                 
 dropout_3 (Dropout)         (None, 9, 9, 256)         0         
                                                                 
 flatten_1 (Flatten)         (None, 20736)             0         
                                                                 
 dense_3 (Dense)             (None, 525)               10886925  
                                                                 
=================================================================
Total params: 11,275,341
Trainable params: 11,275,341
Non-trainable params: 0
```

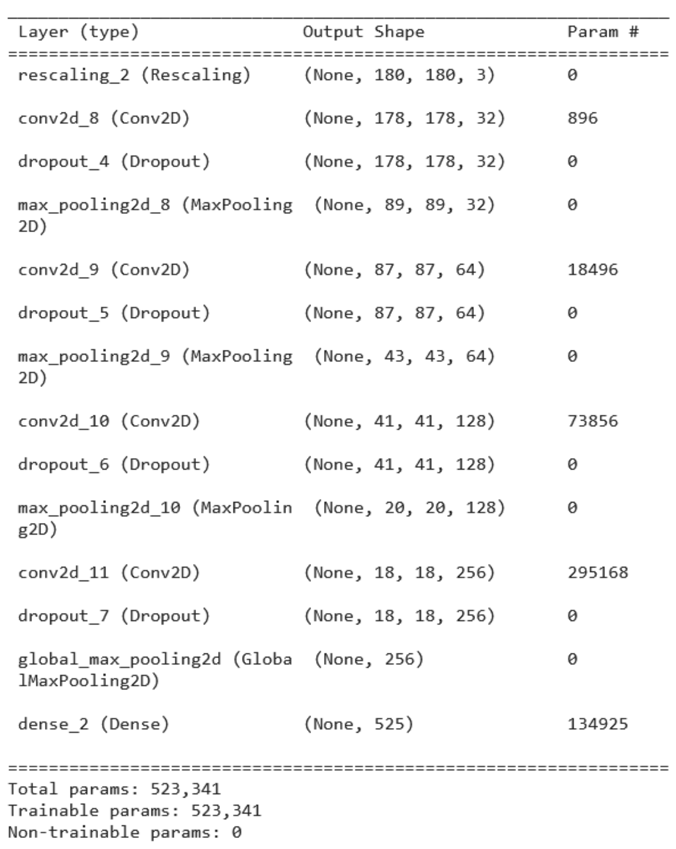
The addition of dropout layers managed to reduce the validation loss significantly, though it didn't offer much in terms of accuracy. The training time was also reduced marginally.





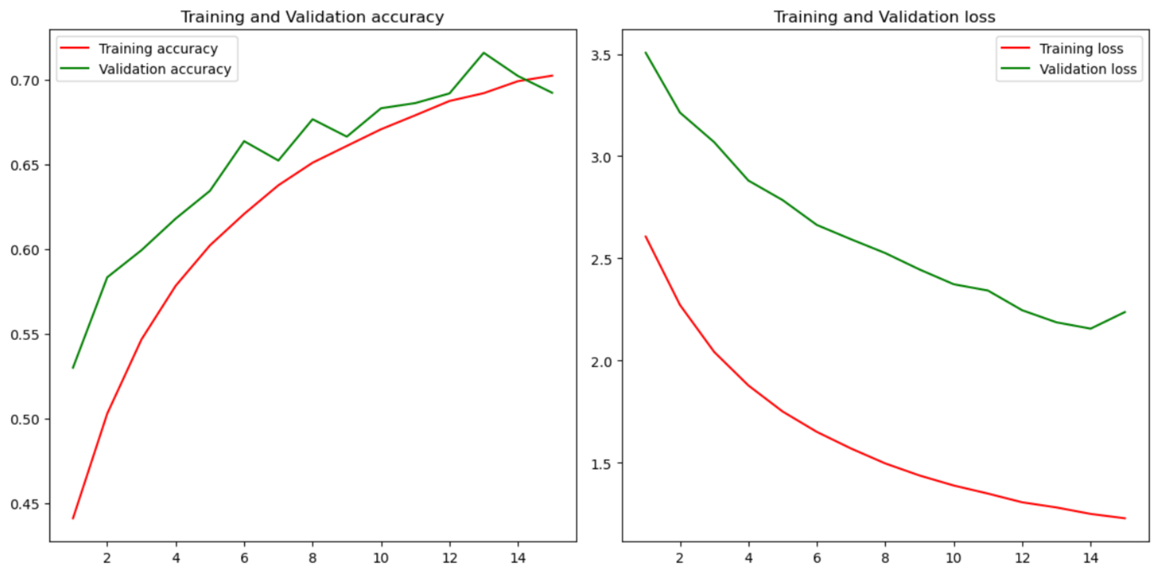
Replacing the Flatten Layer with a GlobalMaxPooling2D Layer

The next step is to replace the Flatten layer before the final Dense layer with a Global Max Pooling 2D layer. Compared to the Flatten layer, which converts the 2D feature maps into a 1D vector, GlobalMaxPooling2D reduces the dimensionality of the feature maps while still preserving the most important spatial information. A major advantage of the GlobalMaxPooling2D layer compared to the Flatten layer is that the former considers the entire feature map, enabling the network to capture global context information. This global perspective is particularly useful for tasks that require a holistic understanding of the input, such as the current classification problem. The addition of the GlobalMaxPooling2D layer will also decrease the number of parameters significantly.



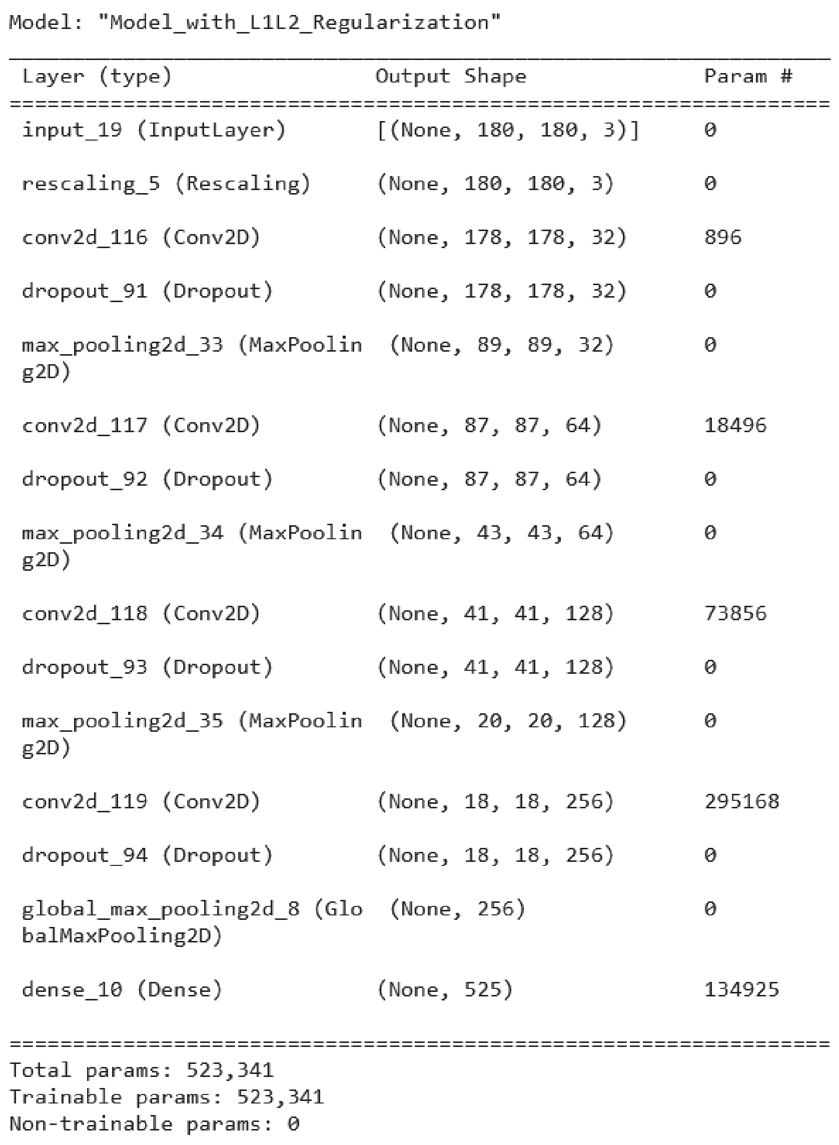
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_2 (Rescaling)     (None, 180, 180, 3)       0

 conv2d_8 (Conv2D)           (None, 178, 178, 32)      896

 dropout_4 (Dropout)         (None, 178, 178, 32)      0

 max_pooling2d_8 (MaxPooling  (None, 89, 89, 32)       0
 2D)

 conv2d_9 (Conv2D)           (None, 87, 87, 64)        18496

 dropout_5 (Dropout)         (None, 87, 87, 64)        0

 max_pooling2d_9 (MaxPooling  (None, 43, 43, 64)       0
 2D)

 conv2d_10 (Conv2D)          (None, 41, 41, 128)       73856

 dropout_6 (Dropout)         (None, 41, 41, 128)       0

 max_pooling2d_10 (MaxPoolin  (None, 20, 20, 128)      0
 g2D)

 conv2d_11 (Conv2D)          (None, 18, 18, 256)       295168

 dropout_7 (Dropout)         (None, 18, 18, 256)       0

 global_max_pooling2d (Globa  (None, 256)              0
 lMaxPooling2D)

 dense_2 (Dense)             (None, 525)               134925

=================================================================
Total params: 523,341
Trainable params: 523,341
Non-trainable params: 0
```

The addition of the GlobalMaxPooling2D layer drastically improved the validation accuracy, while at the same time reducing the training accuracy, which show that our model no loger overfits the data. As a matter of fact it seems like the validation accuracy would continue to improve if the model was trained for more epochs.



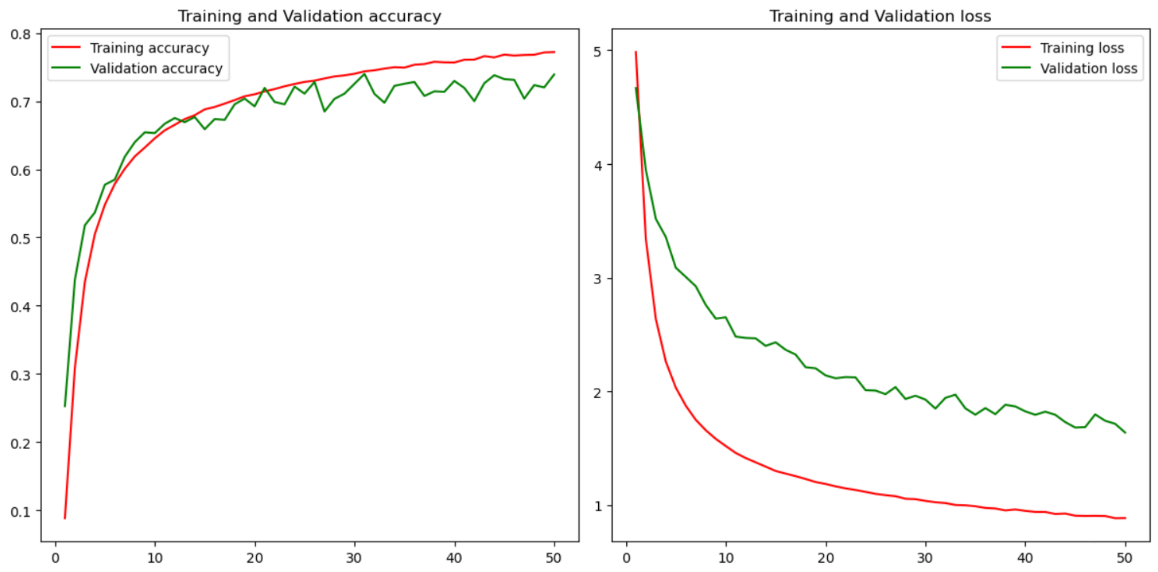
Further Improving the Model with L1/L2 Regularization

In the final model, L1/L2 regularization is inserted in the first 2 convolutional layers. Through the penalties applied to the loss function this approach manages to keep the overall weights lower. Specifically, L1 regularization promotes sparsity and feature selection, while L2 regularization encourages overall weight reduction.
So, the L1/L2 regularization is expected to further reduce overfitting and improve the model's ability to generalize.



```
Model: "Model_with_L1L2_Regularization"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_19 (InputLayer)       [(None, 180, 180, 3)]     0

 rescaling_5 (Rescaling)     (None, 180, 180, 3)       0

 conv2d_116 (Conv2D)         (None, 178, 178, 32)      896

 dropout_91 (Dropout)        (None, 178, 178, 32)      0

 max_pooling2d_33 (MaxPoolin  (None, 89, 89, 32)       0
 g2D)

 conv2d_117 (Conv2D)         (None, 87, 87, 64)        18496

 dropout_92 (Dropout)        (None, 87, 87, 64)        0

 max_pooling2d_34 (MaxPoolin  (None, 43, 43, 64)       0
 g2D)

 conv2d_118 (Conv2D)         (None, 41, 41, 128)       73856

 dropout_93 (Dropout)        (None, 41, 41, 128)       0

 max_pooling2d_35 (MaxPoolin  (None, 20, 20, 128)      0
 g2D)

 conv2d_119 (Conv2D)         (None, 18, 18, 256)       295168

 dropout_94 (Dropout)        (None, 18, 18, 256)       0

 global_max_pooling2d_8 (Glo  (None, 256)              0
 balMaxPooling2D)

 dense_10 (Dense)            (None, 525)               134925

=================================================================
Total params: 523,341
Trainable params: 523,341
Non-trainable params: 0
```

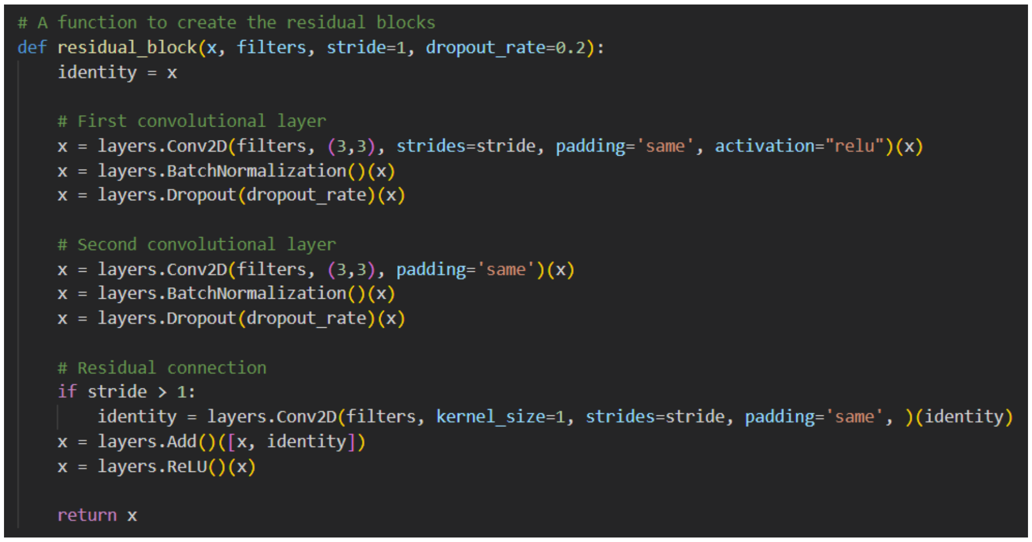
After training this model for 50 epochs we can see that the validation accuracy was marginally improved (from 71% to 74%) compared to the previous best model. Given the large number of classes this improvement might seem small but it is actually significant.



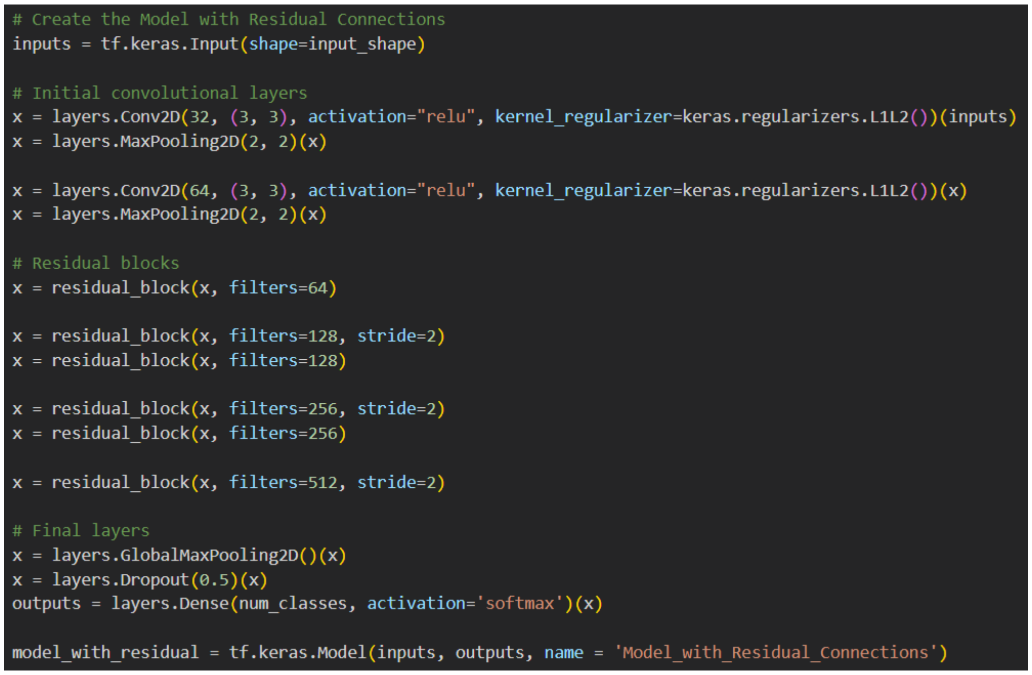
Adding Residual Connections

The final step is to add residual connections to the model. After the two first convolutional layers, this network contains 6 more blocks of convolutional layers with residual connections. Every block contains two convolutional layers followed by a BatchNormalization layer and a Dropout layer. We are also removing the dropout from the first two convolutional layers in this model and instead adding it only to the 6 blocks. The 6 blocks of residual connections are followed by a GlobalMaxPooling2D layer, a Dropout layer and the final classification layer.

Note: This model is too lengthy to fit in a screenshot, so I will provide here the code snippet used for the residual blocks as well as the code snippet that creates the model.

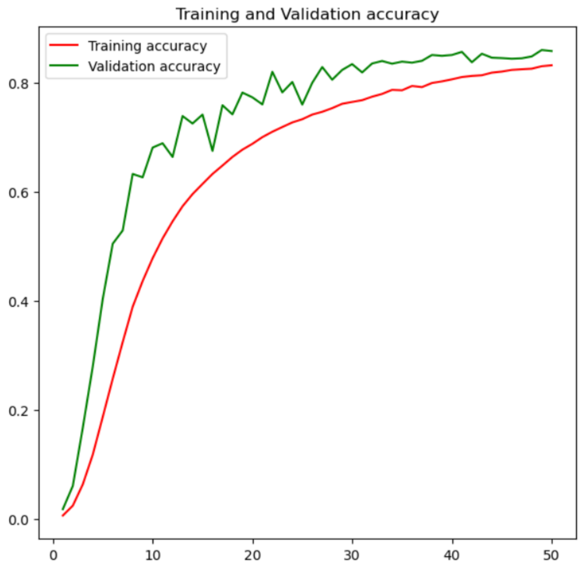


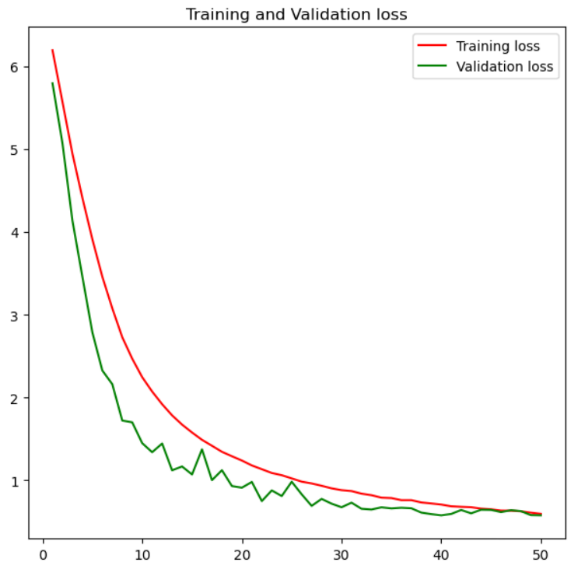
```python
# A function to create the residual blocks
def residual_block(x, filters, stride=1, dropout_rate=0.2):
    identity = x

    # First convolutional layer
    x = layers.Conv2D(filters, (3,3), strides=stride, padding='same', activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Dropout(dropout_rate)(x)

    # Second convolutional layer
    x = layers.Conv2D(filters, (3,3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Dropout(dropout_rate)(x)

    # Residual connection
    if stride > 1:
        identity = layers.Conv2D(filters, kernel_size=1, strides=stride, padding='same', )(identity)
    x = layers.Add()([x, identity])
    x = layers.ReLU()(x)

    return x
```



```python
# Create the Model with Residual Connections
inputs = tf.keras.Input(shape=input_shape)

# Initial convolutional layers
x = layers.Conv2D(32, (3, 3), activation="relu", kernel_regularizer=keras.regularizers.L1L2())(inputs)
x = layers.MaxPooling2D(2, 2)(x)

x = layers.Conv2D(64, (3, 3), activation="relu", kernel_regularizer=keras.regularizers.L1L2())(x)
x = layers.MaxPooling2D(2, 2)(x)

# Residual blocks
x = residual_block(x, filters=64)

x = residual_block(x, filters=128, stride=2)
x = residual_block(x, filters=128)

x = residual_block(x, filters=256, stride=2)
x = residual_block(x, filters=256)

x = residual_block(x, filters=512, stride=2)

# Final layers
x = layers.GlobalMaxPooling2D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(num_classes, activation='softmax')(x)

model_with_residual = tf.keras.Model(inputs, outputs, name = 'Model_with_Residual_Connections')
```

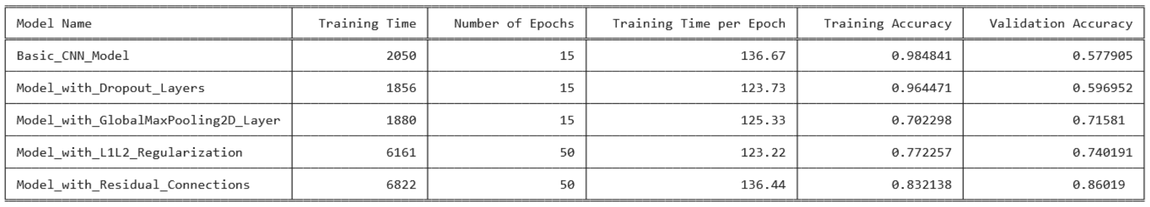
As we can see the addition of residual blocks increased the accuracy of the model significantly but also increased the training time as the model parameters also increased by a lot. Specifically, the validation accuracy improved by around 12% compared to the previous best model (from 74% to 86%) and it would probably reach an even higher number if the training was longer than 50 epochs.





Conclusions from the Training of the CNN Model on the 525 Bird Species Dataset

The following table shows that the earlier models with flatten layer and wiehout dropout tend to overfit alot and learn the training data too well. Both the addition of Dropout layers and the replacement of the Flatten layer with a GlobalMaxPooling 2D layer was enough to stop the model from overfitting and improve the validation accuracy. The L1/L2 regularization also improved the accuracy while keeping the training times relatively low. Finally, thw addition of residual blocks improved the accuracy by a lot but it also increased the training time as we are now training more parameters.



Model Name	Training Time	Number of Epochs	Training Time per Epoch	Training Accuracy	Validation Accuracy
Basic_CNN_Model	2050	15	136.67	0.984841	0.577905
Model_with_Dropout_Layers	1856	15	123.73	0.964471	0.596952
Model_with_GlobalMaxPooling2D_Layer	1880	15	125.33	0.702298	0.71581
Model_with_L1L2_Regularization	6161	50	123.22	0.772257	0.740191
Model_with_Residual_Connections	6822	50	136.44	0.832138	0.86019

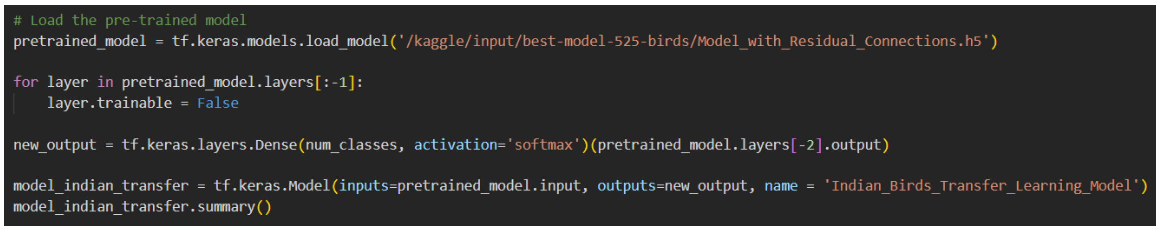
Transfer Learning

Transfer Learning with the Best Model

In the first experiment we are going to explore if the best model we saved from Part 1 can classify this new dataset with only the last (classification) layer trainable. What we're gonna do is freeze all the layers of this model and replace the final Dense layer with a new layer that will classify the 25 bird species of the Indian Birds dataset.

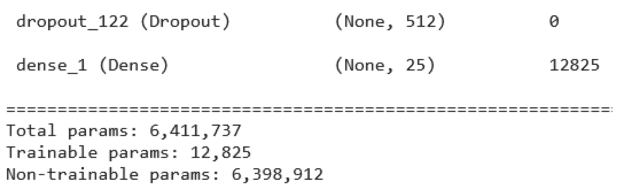
The first step is to load the model. Same as before, this code is recommended to run in Kaggle's platform, so the model has to be uploaded there and imported through the load_model function. The code presumes that the model is uploaded in a folder named "best-model-525-birds". The model can be found in the folder "models" within this repository. In order to load the model locally, the path provided to the load_model function should be changed to the model's location.

After loading the model, the next step is to freeze (make non-trainable) all layers except the last one (classification layer). It is interesting to note that the new classifier layer has a significantly smaller number of parameters as the classes that need to be classified are alot less than our original dataset (525 vs 25).



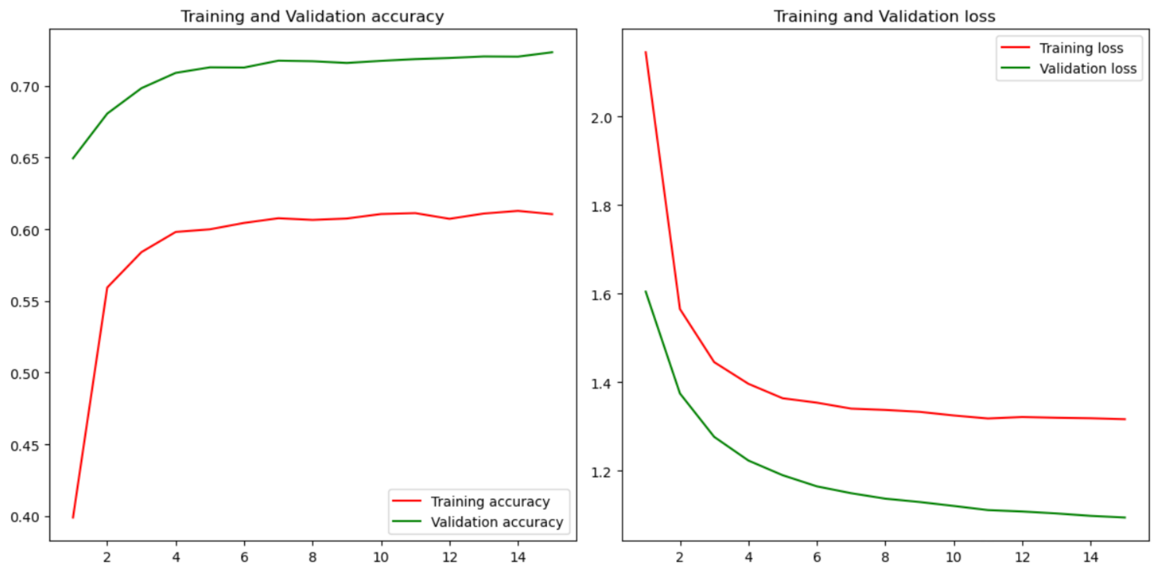
```
# Load the pre-trained model
pretrained_model = tf.keras.models.load_model('/kaggle/input/best-model-525-birds/Model_with_Residual_Connections.h5')

for layer in pretrained_model.layers[:-1]:
    layer.trainable = False

new_output = tf.keras.layers.Dense(num_classes, activation='softmax')(pretrained_model.layers[-2].output)

model_indian_transfer = tf.keras.Model(inputs=pretrained_model.input, outputs=new_output, name = 'Indian_Birds_Transfer_Learning_Model')
model_indian_transfer.summary()
```

After applying the above steps we can see that only the last layer’s parameters are trainable:



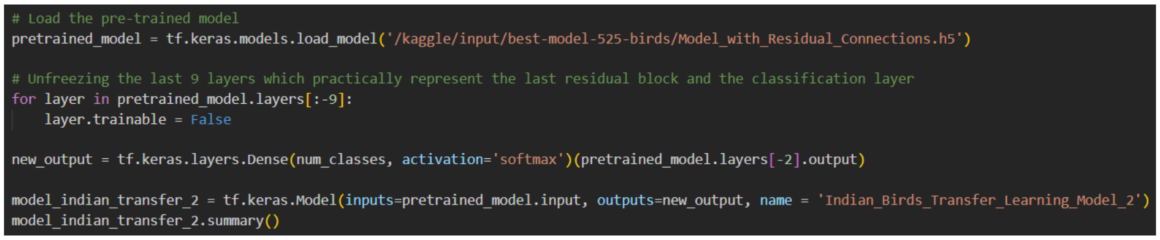
```
 dropout_122 (Dropout)          (None, 512)          0

 dense_1 (Dense)                (None, 25)           12825

=================================================================
Total params: 6,411,737
Trainable params: 12,825
Non-trainable params: 6,398,912
```

The model we imported that was trained on the previous dataset managed to achieve around 70% accuracy in this new dataset after only 3-4 epochs. Though, after that the accuracy stopped increased something that was more or less expected as the most parameters of the model are frozen and cannot be trained. So practically, the rest of the epochs provided no value in terms of accuracy. A way to avoid this wasted training time is to add an early stopping callback while training the model. In this case I opted not to do this in order to explain what's happening more easily.



Transfer Learning while Unfreezing More Layers

In this experiment I will try also unfreezing the last convolutional block before the final Dense layer and see if that changes how fast the model trains or how quickly it can reach a high accuracy score.

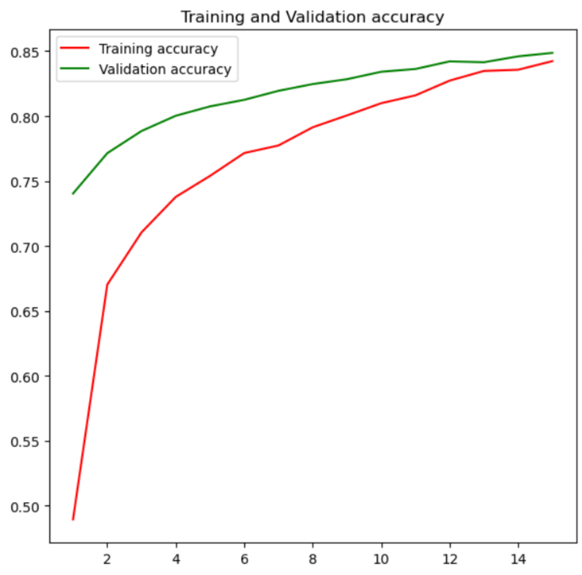


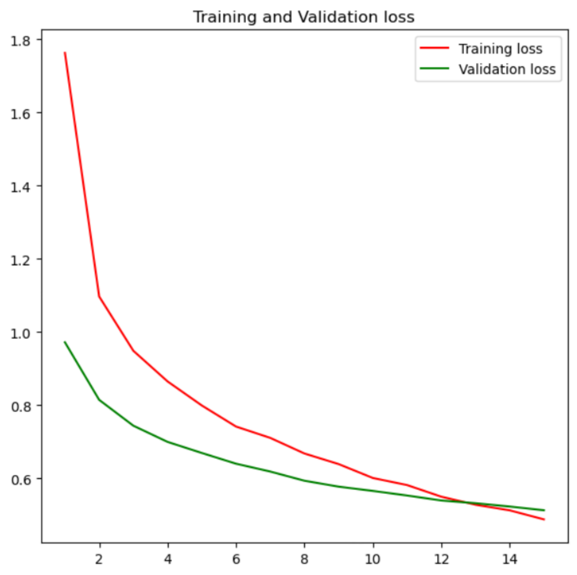
```python
# Load the pre-trained model
pretrained_model = tf.keras.models.load_model('/kaggle/input/best-model-525-birds/Model_with_Residual_Connections.h5')

# Unfreezing the last 9 layers which practically represent the last residual block and the classification layer
for layer in pretrained_model.layers[:-9]:
    layer.trainable = False

new_output = tf.keras.layers.Dense(num_classes, activation='softmax')(pretrained_model.layers[-2].output)

model_indian_transfer_2 = tf.keras.Model(inputs=pretrained_model.input, outputs=new_output, name = 'Indian_Birds_Transfer_Learning_Model_2')
model_indian_transfer_2.summary()
```

As we can see, now the trainable params are way more than the previous model:

```
 dropout_120 (Dropout)          (None, 6, 6, 512)    0

 conv2d_152 (Conv2D)            (None, 6, 6, 512)    2359808

 batch_normalization_78 (BatchN  (None, 6, 6, 512)   2048
 ormalization)

 dropout_121 (Dropout)          (None, 6, 6, 512)    0

 conv2d_153 (Conv2D)            (None, 6, 6, 512)    131584

 add_44 (Add)                   (None, 6, 6, 512)    0


 re_lu_54 (ReLU)                (None, 6, 6, 512)    0

 global_max_pooling2d_10 (Globa  (None, 512)         0
 lMaxPooling2D)

 dropout_122 (Dropout)          (None, 512)          0

 dense_2 (Dense)                (None, 25)           12825

=============================================================
Total params: 6,411,737
Trainable params: 2,505,241
Non-trainable params: 3,906,496
```

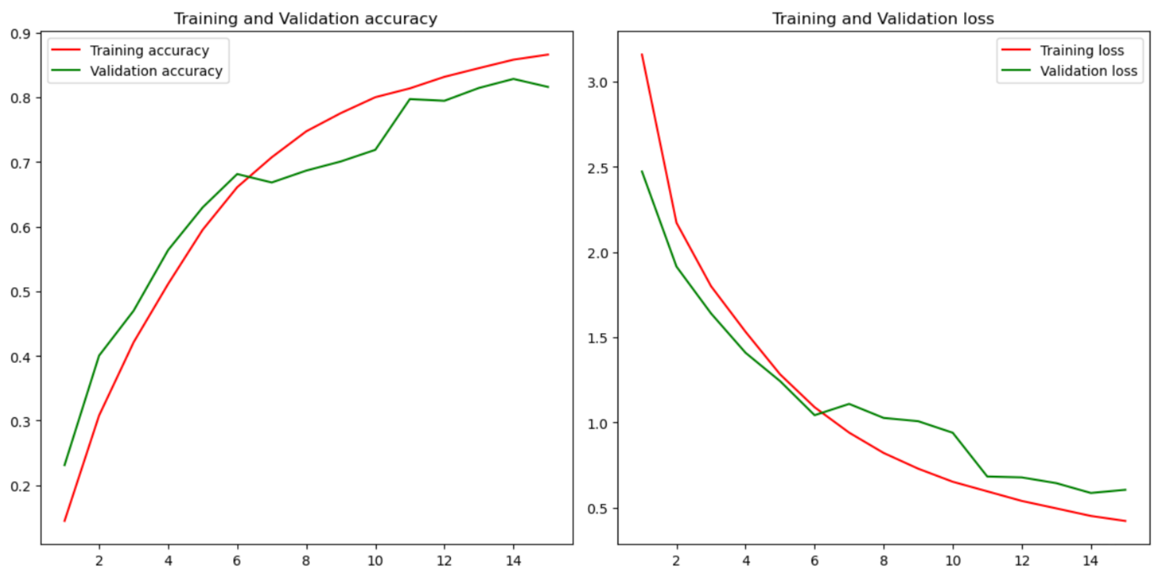
As a result, the model managed to use the prior knowledge of the model we imported which was trained on a similar task and achieved close to 85% percentage in this new, different dataset.





Training all the Layers of the Final Model from Scratch to Compare with the Transfer Learning Results

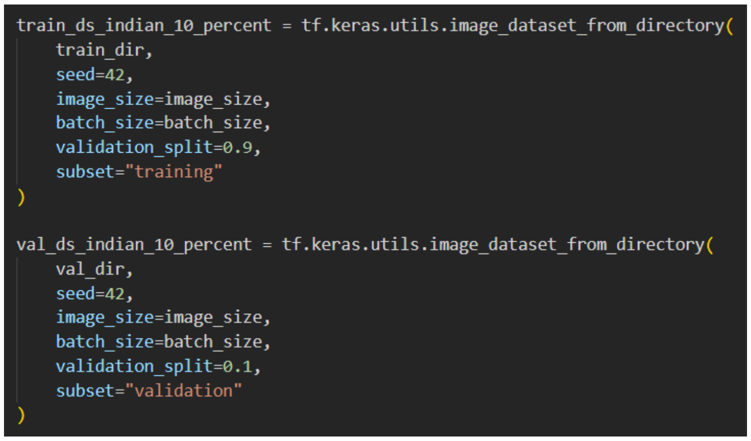
The question is, though, what would happen if we tried to train a new model from scratch on the indan birds dataset? Would this be more or less efficient regarding both accuracy and training time? Practically what we're doing is using the same model architecture that yielded the best results in the 525 bird species dataset and then training this model in the new dataset without using any pretrained weights.



After training this model for 15 epochs we can see that the validation accuracy is lower than the one achieved in the transfer learning model. Also the model took more time to train those 15 epochs as it had to adjust a greater number of parameters. From the following plot it is obvious, tough, that if the training were to resume for more epochs we would probably surpass the pretrained model's accuracy. The main benefit of transfer learning is that we can use the weight of a model that was trained on a similar task to achieve high accuracy on a similar dataset, thus reducing the training time significantly. Of course, training a model from scratch is expected to give better results after a large number of epochs, but this requires a lot of computational power and more time that can be reduced significantly if we use transfer learning.

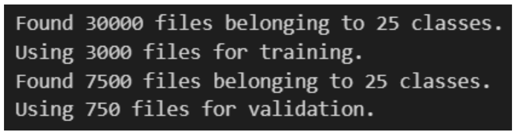
Transfer Learning with the Final Model and Using only 10% of the Dataset

In the following 3 experiments I'm going to try reducing the size of the training and validation datasets to 10% of the original size and see how that affects the results from the 3 models we examined earlier.



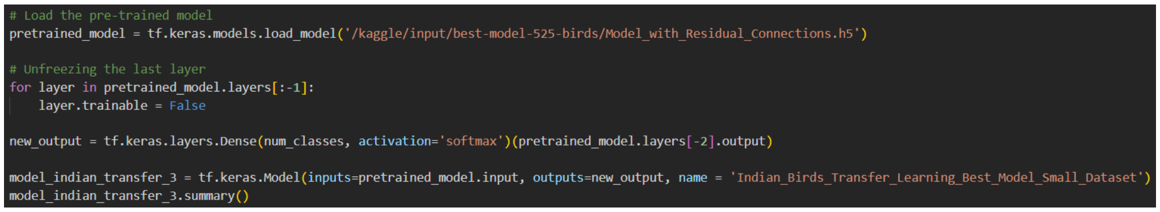
```
train_ds_indian_10_percent = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    seed=42,
    image_size=image_size,
    batch_size=batch_size,
    validation_split=0.9,
    subset="training"
)

val_ds_indian_10_percent = tf.keras.utils.image_dataset_from_directory(
    val_dir,
    seed=42,
    image_size=image_size,
    batch_size=batch_size,
    validation_split=0.1,
    subset="validation"
)
```

To achieve that I split the training and validation sets and use only 10% of each:



```
Found 30000 files belonging to 25 classes.
Using 3000 files for training.
Found 7500 files belonging to 25 classes.
Using 750 files for validation.
```

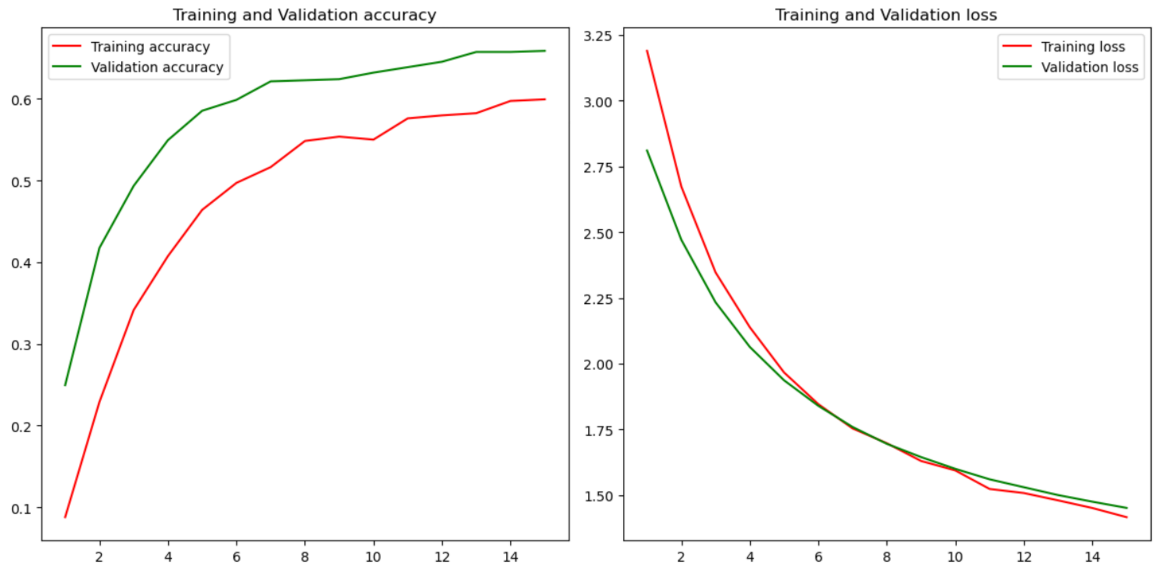
Then I import the pretrained model again:



```
# Load the pre-trained model
pretrained_model = tf.keras.models.load_model('/kaggle/input/best-model-525-birds/Model_with_Residual_Connections.h5')

# Unfreezing the last layer
for layer in pretrained_model.layers[:-1]:
    layer.trainable = False

new_output = tf.keras.layers.Dense(num_classes, activation='softmax')(pretrained_model.layers[-2].output)

model_indian_transfer_3 = tf.keras.Model(inputs=pretrained_model.input, outputs=new_output, name = 'Indian_Birds_Transfer_Learning_Best_Model_Small_Dataset')
model_indian_transfer_3.summary()
```

And train the model with only the last layer trainable.

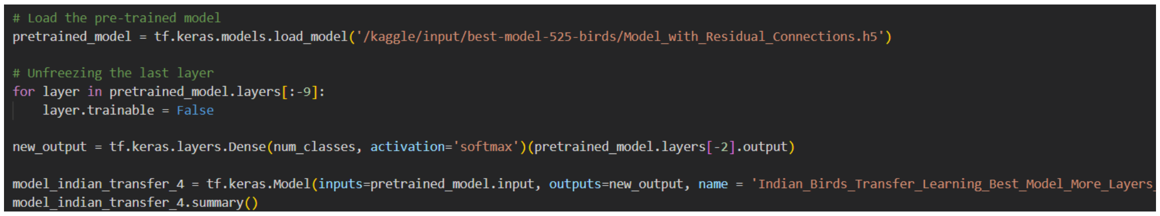
Obviously, reducing the size of the dataset also slashed the training time by a significant margin, though the accuracy of the model was still close to 66%. This shows how transfer learning can help leverage prior knowledge to make effective models even when the dataset we're working with is relatively small.



Transfer Learning with the Final Model and Using Only 10% of the Dataset with More Layers Trainable

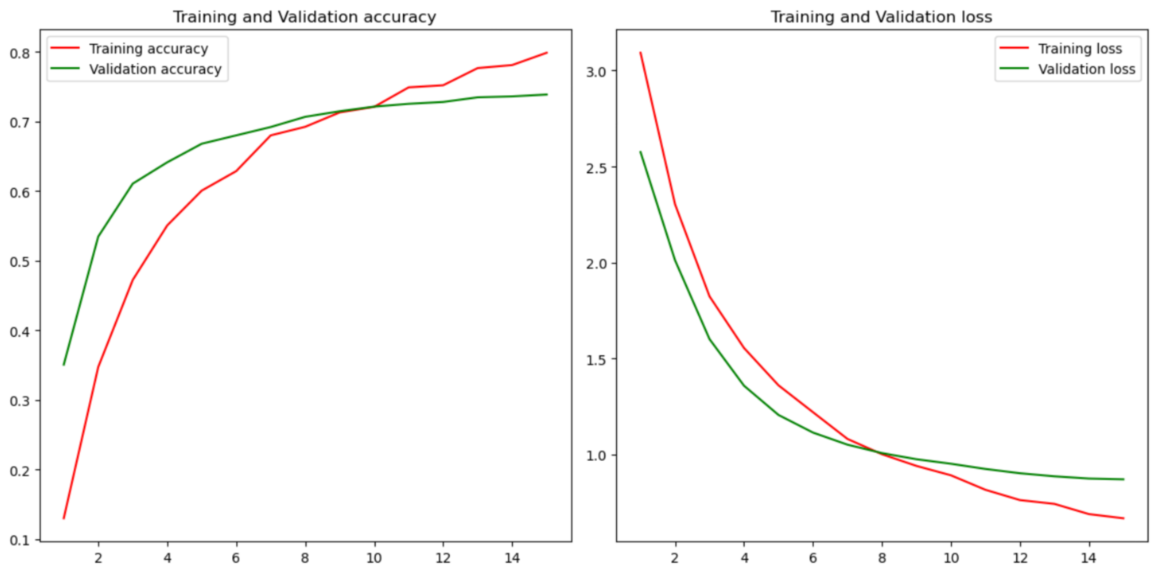
In this experiment we are goint to find out how the pretrained model fares in this new smaller dataset if it can also train the last residual block except the final classifier layer.

I import the model again and this time unfreeze the last 9 layers which correspond to the last residual block and the dense layer:



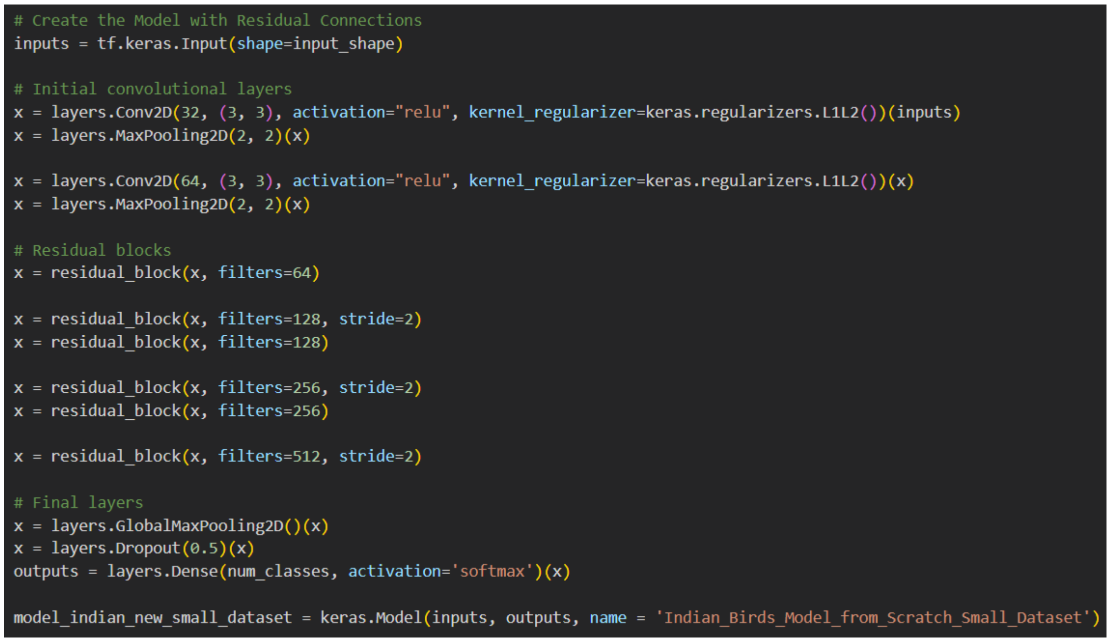
```
# Load the pre-trained model
pretrained_model = tf.keras.models.load_model('/kaggle/input/best-model-525-birds/Model_with_Residual_Connections.h5')

# Unfreezing the last layer
for layer in pretrained_model.layers[:-9]:
    layer.trainable = False

new_output = tf.keras.layers.Dense(num_classes, activation='softmax')(pretrained_model.layers[-2].output)

model_indian_transfer_4 = tf.keras.Model(inputs=pretrained_model.input, outputs=new_output, name = 'Indian_Birds_Transfer_Learning_Best_Model_More_Layers_
model_indian_transfer_4.summary()
```

Unfreezing of the last residual block increased the accuracy of the transfer learning model by a significant margin without increasing the training time. This time the model approaches an accuracy of 74% after only 15 epochs of training on the smaller dataset.



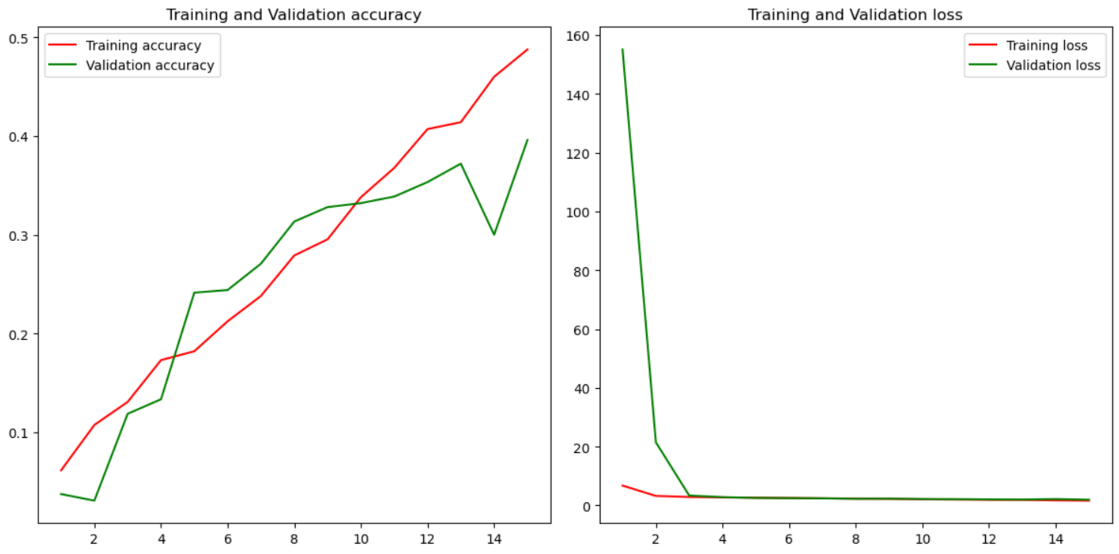
Training the Final Model from Scratch on the Indian Dataset Using Only 10% of the Dataset

Now it is time to see if it is possible to train a new model on the smaller dataset. Would this model compare to the results we achieved with the transfer learning models? Again I'm using the same model architecture but this time there are no pretrained weights.



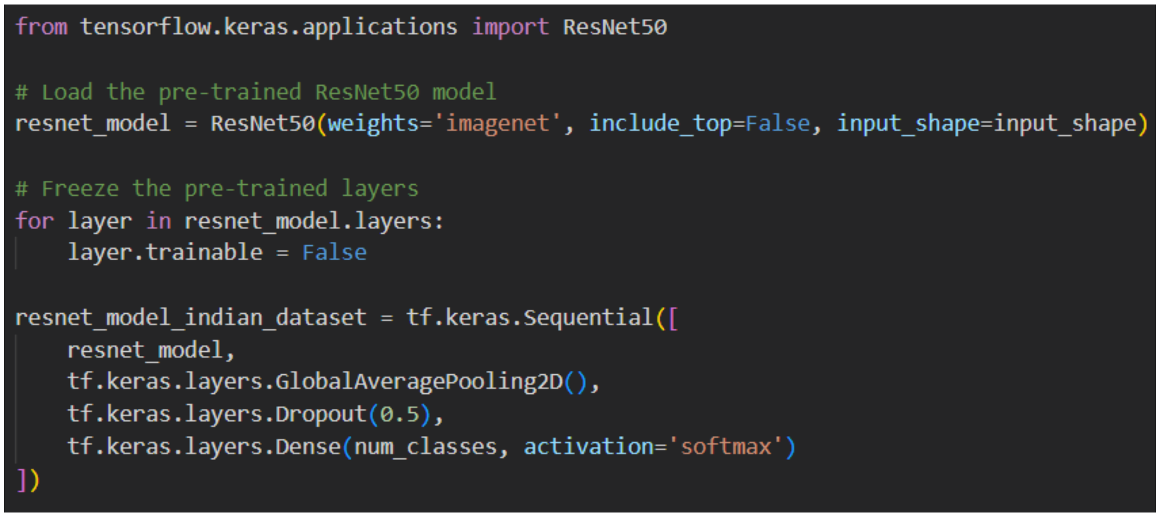
```
# Create the Model with Residual Connections
inputs = tf.keras.Input(shape=input_shape)

# Initial convolutional layers
x = layers.Conv2D(32, (3, 3), activation="relu", kernel_regularizer=keras.regularizers.L1L2())(inputs)
x = layers.MaxPooling2D(2, 2)(x)

x = layers.Conv2D(64, (3, 3), activation="relu", kernel_regularizer=keras.regularizers.L1L2())(x)
x = layers.MaxPooling2D(2, 2)(x)

# Residual blocks
x = residual_block(x, filters=64)

x = residual_block(x, filters=128, stride=2)
x = residual_block(x, filters=128)

x = residual_block(x, filters=256, stride=2)
x = residual_block(x, filters=256)

x = residual_block(x, filters=512, stride=2)

# Final layers
x = layers.GlobalMaxPooling2D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(num_classes, activation='softmax')(x)

model_indian_new_small_dataset = keras.Model(inputs, outputs, name = 'Indian_Birds_Model_from_Scratch_Small_Dataset')
```

Training this model from scratch on such a small dataset didn't even managed to breach the 40% accuracy barrier as it is obvious from the following plots. Would this accuracy improve with further training of the model? We cannot say for sure but it seems like the model was probably too complex for this small dataset so I guess the pretrained models got this one.



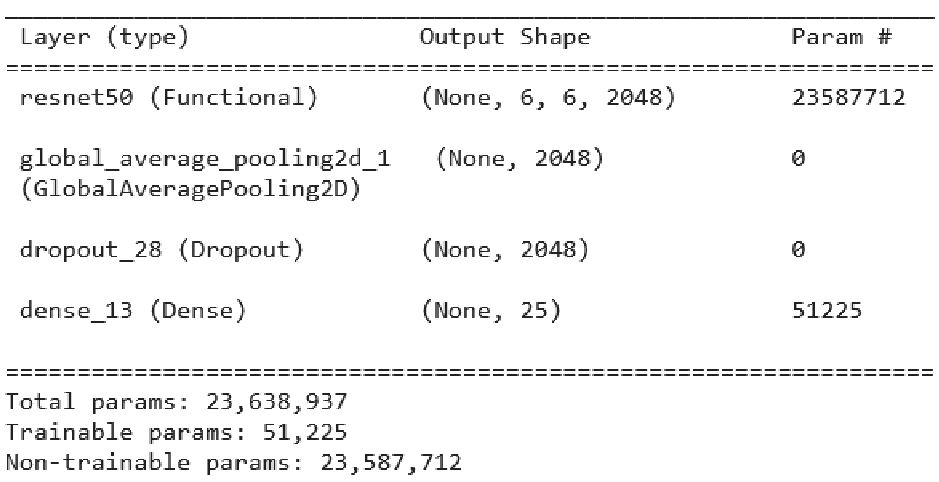
Transfer Learning with the ResNet Model on the Indian Dataset

For this last part I'm going to examine one of the most popular models for image classification, namely the ResNet model trained on the ImageNet dataset. When importing the model, I'm discarding the classifier layer, and adding a GlobalMaxPooling2D layer and a Dropout layer before the final classification layer for the 25 bird species of India.



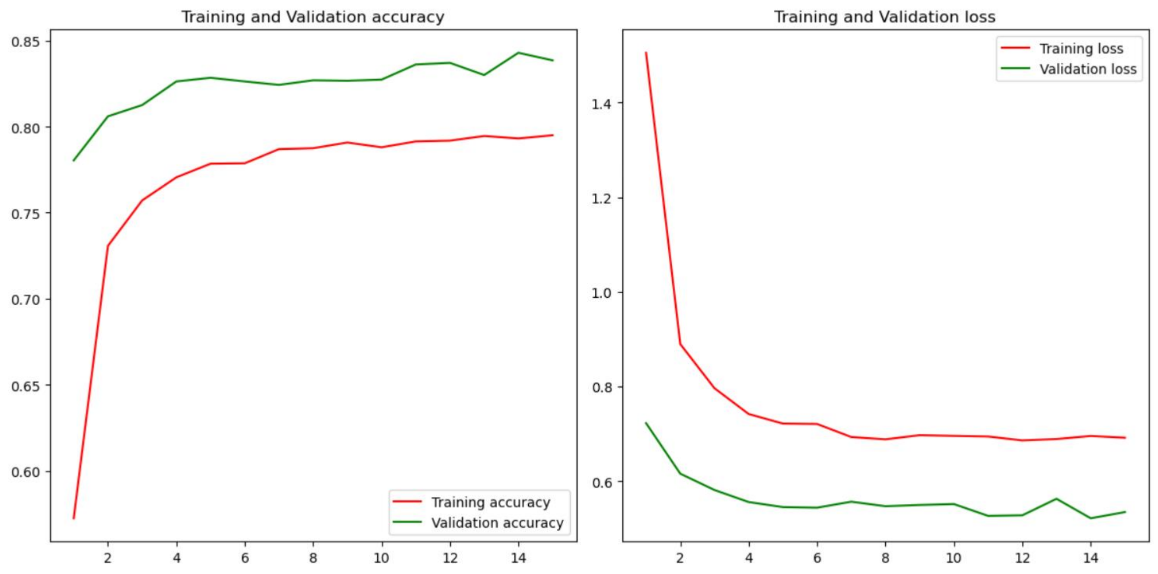
```python
from tensorflow.keras.applications import ResNet50

# Load the pre-trained ResNet50 model
resnet_model = ResNet50(weights='imagenet', include_top=False, input_shape=input_shape)

# Freeze the pre-trained layers
for layer in resnet_model.layers:
    layer.trainable = False

resnet_model_indian_dataset = tf.keras.Sequential([
    resnet_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])
```

Once again, only the last layer will be trainable and I'm going to explore the metrics of this model.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 6, 6, 2048)        23587712

 global_average_pooling2d_1   (None, 2048)             0
 (GlobalAveragePooling2D)

 dropout_28 (Dropout)        (None, 2048)              0

 dense_13 (Dense)            (None, 25)                51225

=================================================================
Total params: 23,638,937
Trainable params: 51,225
Non-trainable params: 23,587,712
```

After only 15 epochs, the ResNet model achieved around 84% in the Indian birds dataset. Actually, it is obvious from the plot that after only 2 epochs, the model already has above 80% accuracy on the dataset.

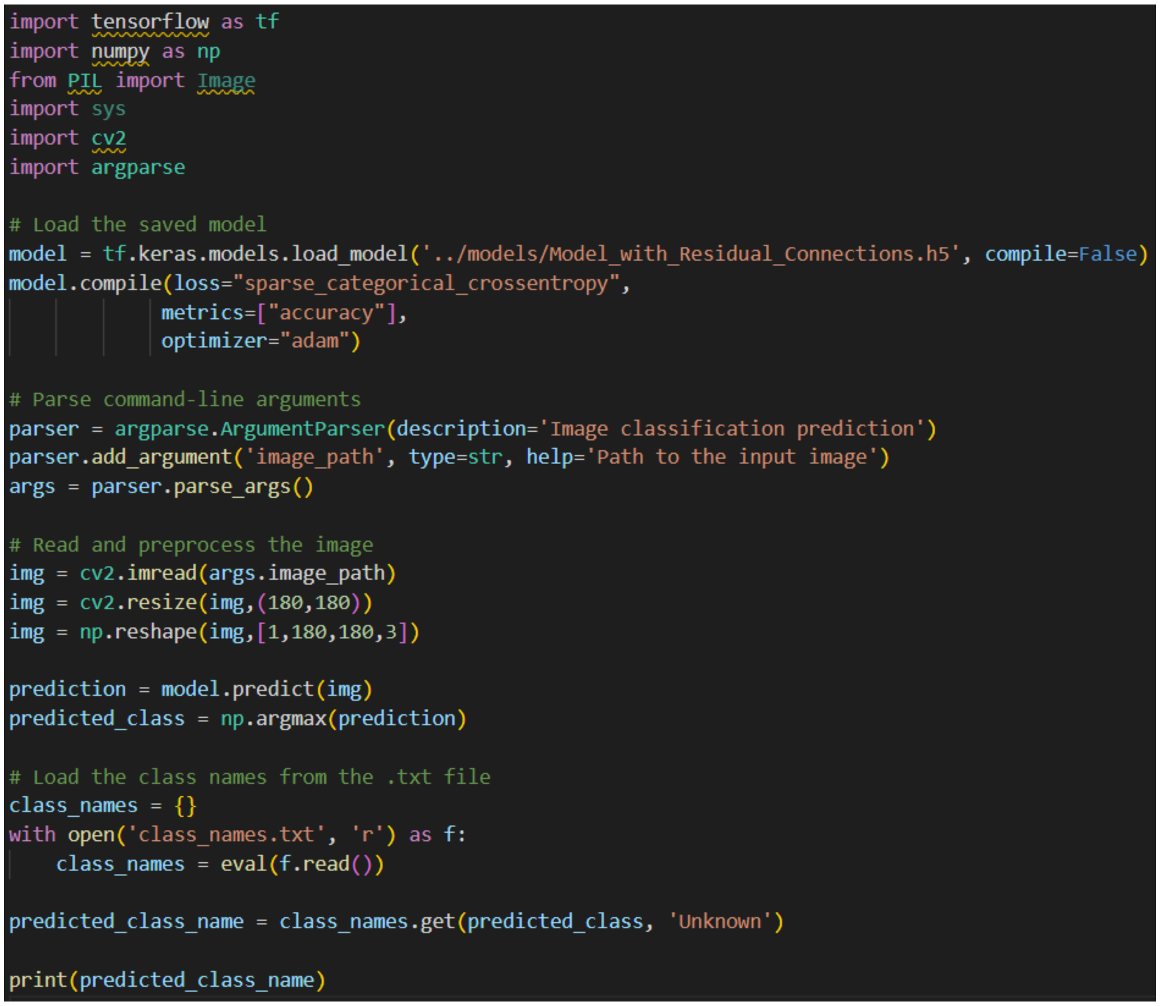


Final Observations and Conclusion

As a closing statement, we could say that in most cases it is better to finetune a pretrained model on a similar task than train a new model from scratch on most datasets. Transfer learning allows us to use the prior knowledge of similar models, thus cutting down both training time and computational requirements. Especially for smaller datasets, transfer learning models can achieve good results while this is not the case for most models without prior knowledge.

Demo Code

The repository contains also a python script to test the model with residual connections that achieved the highest validation accuracy on the first dataset:



```python
import tensorflow as tf
import numpy as np
from PIL import Image
import sys
import cv2
import argparse

# Load the saved model
model = tf.keras.models.load_model('../models/Model_with_Residual_Connections.h5', compile=False)
model.compile(loss="sparse_categorical_crossentropy",
              metrics=["accuracy"],
              optimizer="adam")

# Parse command-line arguments
parser = argparse.ArgumentParser(description='Image classification prediction')
parser.add_argument('image_path', type=str, help='Path to the input image')
args = parser.parse_args()

# Read and preprocess the image
img = cv2.imread(args.image_path)
img = cv2.resize(img,(180,180))
img = np.reshape(img,[1,180,180,3])

prediction = model.predict(img)
predicted_class = np.argmax(prediction)

# Load the class names from the .txt file
class_names = {}
with open('class_names.txt', 'r') as f:
    class_names = eval(f.read())

predicted_class_name = class_names.get(predicted_class, 'Unknown')

print(predicted_class_name)
```

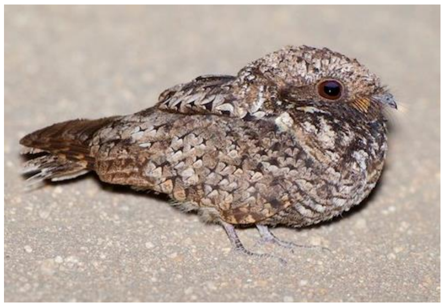
To test this code type in the command line:

python demo.py ‘image path’

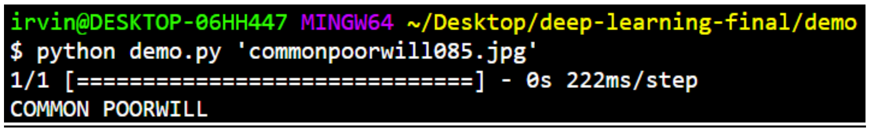
The image path should be replaced with the location of the image that has to be checked. The code also uses a text file that is located in the same folder to return an actual bird species instead of the encoded number.

Below are some results of the code execution with random images from the internet:

Input: (Common Poorwill)

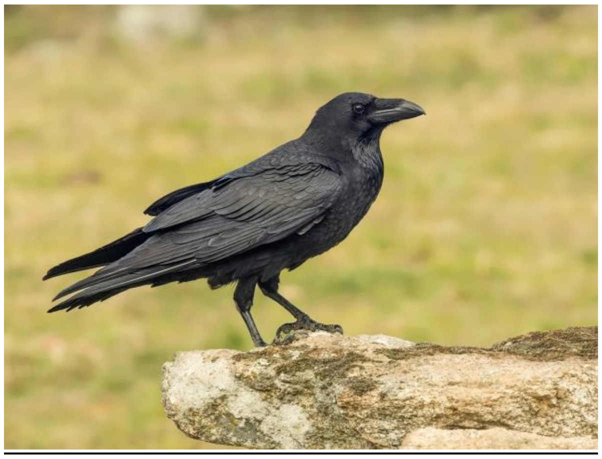


Prediction:

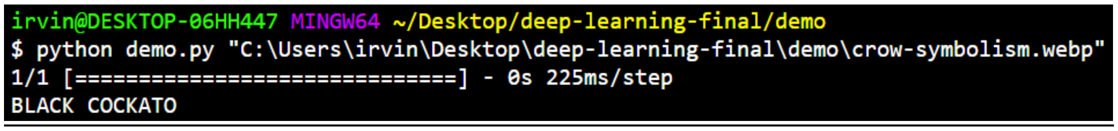


```
irvin@DESKTOP-06HH447 MINGW64 ~/Desktop/deep-learning-final/demo
$ python demo.py 'commonpoorwill085.jpg'
1/1 [==============================] - 0s 222ms/step
COMMON POORWILL
```

Input: (Crow)



Prediction:



```
irvin@DESKTOP-06HH447 MINGW64 ~/Desktop/deep-learning-final/demo
$ python demo.py "C:\Users\irvin\Desktop\deep-learning-final\demo\crow-symbolism.webp"
1/1 [==============================] - 0s 225ms/step
BLACK COCKATO
```