MASTER DEGREE IN DATA SCIENCE AND ADVANCED ANALYTICS, WITH SPECIALIZATION IN DATA SCIENCE

DEEP LEARNING PROJECT REPORT

LECTURER:

MAURO CASTELLI

GROUP 8

CATARINA CANDEIAS, M20200656
MOHAMED ELBAWAB, M20201102
RITA FERREIRA, M20200661
TIAGO GONÇALVES, M20201053

04/04/2021

Index

1. Introduction

We live in a time where information spreads faster than ever. One particular type is images – millions of pictures are uploaded every day on the internet. Many of these pictures are labelled by the users who upload them, whether on social media websites or blogs, and some of them later end up appearing as image web search results. Combined with mislabelling, one major consequence is that many of these search results contain images from subjects the user was not searching for.

Given this, we decided to work with images from two famous monuments that are frequently confused: Cristo Rei (Portugal) and Cristo Redentor (Brazil). Our project consists in developing a Deep Learning model that is able to correctly classify pictures from these two monuments as one or the other, recurring to Keras. The purpose of this system is acting as a filter that could enhance results of web searches – preventing the user from finding images from one monument when searching for the other. This is just an academic example and there are several other cases where a system of this nature could be applied.

2. Methodology

In this section we explain our approach and implementation details, while supporting some of the decisions that were made. We start by explaining how the data for our model was obtained, and then summarize the process of splitting the data into train and test folders. Subsequently, the performance measures used to assess our models are defined and, finally, we explain the process of building our models until achieving the best performance. The results obtained from this approach are then presented on section 3.

2.1 – Obtaining the data

To develop our model, we first needed to obtain the necessary data – images of both monuments. Using the web search engine Google Images, we automatically extracted the URLs of the search results of each monument into two text files (one for each), after inputting the monuments' names in Portuguese and in English. This resulted in 1119 URLs for Cristo Redentor, and 1196 URLs for Cristo Rei. The following step consisted in extracting the images from the respective links, which was done with Python code – *Download_Images* Notebook. Since some of the images could not be retrieved from the respective links, this led to the Cristo Redentor and Cristo Rei folders having 1050 and 995 files, respectively.

After having the folders with the images, some filtering steps were necessary to ensure that each of the folders only had pictures of the respective monument, and that there were no repeated files. Conscious of the limitations of the results of the image search engine, which have been described previously, we proceeded to do a manual filtering of the folders – deleting images that did not correspond to the respective monument or that had low quality. Consequently, the folders' sizes were reduced considerably – 539 and 565 files for the Brazilian and the Portuguese monuments, respectively. The last cleaning step consisted in filtering corrupt images that prevented the Deep Learning model from working. This was done automatically resorting to Python code and deleted a total of 131 images. The final filtered folders had 469 (Cristo Redentor) and 504 (Cristo Rei) image files.

2.2 – Train-test split

Our filtered dataset was composed of 973 images that were fairly balanced between the two binary target classes, being a relatively small sized dataset – which could be a problem when aiming to successfully classify these images. As an attempt to have enough data to train our model and still have the possibility to assess its performance, the images were split into train (80%) and test (20%) datasets, instead of also including a validation dataset.

Before splitting the dataset into two, the images were shuffled. Due to the way the images were retrieved - the first images of a Google search generally have better quality than the last results, this was an important step to guarantee that the distribution of input was similar between the two datasets. This was performed using the *split-folders* package*.*

The fact that the different models were tested on the test dataset and its hyperparameters and architectures adjusted accordingly to its results, could be a source of overfitting. As such, we were always critical when tuning and evaluating the models, implementing techniques to mitigate this effect.

2.3 – Performance Measures

As referred previously, one of our concerns when training the different models and evaluating them was to avoid overfitting (whether to the train or test data), while aiming to obtain the best model possible. The two main metrics used to evaluate the models' performances were the accuracy and the loss function (binary cross entropy). These were measured both on the train and on the test datasets.

Additionally, to determine the best number of epochs and assess up to which point the model stops improving to start overfitting the train dataset these values were plotted using the models' history.

As a final step of assessment, the confusion matrixes of the models' performance on the test dataset were calculated as well as the classification reports with the objective of evaluating their overall performance along with the discrimination ability between the two target classes.

2.4 – General approach – Building the model

Before starting to work on our model, we were conscious of the challenges that would be faced when trying to obtain a model with an ability to generalize well, despite the low quantity of data and the numerous differences among pictures of the same monument, such as different angles or distances.

Given the nature of the problem – image classification – the ideal type of deep learning model for this task was naturally Convolutional Neural Networks. Our approach was performed in a recursive and systematic manner. Starting with a simple model to assess the initial performance on the data, we improved it step by step until arriving at a final one – the one that allowed us to obtain the best performance. Callbacks were implemented to save the models depending on their performance, specifically: *EarlyStopping* - which monitored the values of the test accuracy and stopped the fitting process if the value did not improve after a certain number of epochs; *ModelCheckpoint* – which saved the model in case there were improvements on the test accuracy score.

In order to feed the images to the models, *Keras' ImageDataGenerator* was used. In later stages of model development, it was also used to perform data augmentation, reducing the probabilities of overfitting given the small size of the dataset. When used, data augmentation applied a random set of transformations to the images before feeding them to the model, consequently augmenting the initial dataset and improving the model's generalization ability. When performed, this step was only applied to the train dataset and not to the test dataset, to avoid bias on the results. On the test dataset, *ImageDataGenerator* was only used for rescaling purposes. The batch size used was always 30.

The first model (1) had a simple architecture and our main goal was to have an initial idea of the performance level without any kind of data augmentation or problem-specific architectural choices. Starting with an image input shape of 128 by 128, the model contained two convolution layers, each of them followed by a pooling layer. It also contained a flattening layer, and two interconnected dense layers, of which the second one had only one unit and a sigmoid activation function, since this is a binary classification problem. The summaries of the models can be consulted on the appendix, and their architectures are specified on the delivered notebook.

For the second model (2), we increased the input shape to 256 by 256, with the purpose of allowing the network to capture details that improved its classification ability. We also introduced data augmentation into model (2), defining a set of random transformations that included width and height shifts, zooming and brightness changes, and shearing transformations. Excepting the input shape, the architecture of the second model was the same as the first one.

On the subsequent model – model (3), we used a slightly different architecture, that contained an additional convolution layer, followed by one more pooling layer. The data augmentation set of transformations used was the same as previously. Given the random nature of the training process, this model was trained several times. After this process, the model that returned the best results was saved. The performances achieved with each model are presented and discussed on the next section.

Trying to improve the performance of the previous model, a tuning process was implemented, using the *Keras Tuner* package. Using the same architecture, we defined ranges of values for the number of filters of the first and third convolution layers, as well as for the first dense layer. Besides this, we also tested including a Dropout layer after the Flatten one, with the Dropout rate set to vary between 0 and 0.2. The specific values for the ranges and the personalized model architecture are on the appendix – Tuner Ranges**.** Being a computational expensive and time-consuming step, only a few trials were performed, from which we retrieved the best performing model: model (4).

Finally, we performed transfer-learning on a pre-trained network to compare the performance of model (4) with the performance of an already developed model. The model used was *Xception*. We started by loading it with weights pre-trained on *ImageNet* and setting the input shape with the previously used value (256,256,3). We then froze the base model and created a new model on top of that, to which a Dense layer with one unit was added, in order to perform the binary classification. This model (5) was then fit on the data using the same data augmentation process. The callbacks were slightly different: *EarlyStopping* monitored the test loss instead of the test accuracy, and *ReduceLROnPlateau* was introduced to reduce the learning rate when the test accuracy did not improve.

Aiming to understand which particular shapes and patterns allowed our own built network to distinguish between one monument and the other, we also visualized its heatmaps of class activation. Model (3) was used and some picture examples that resulted from this step are presented on the next section.

3. Results & Discussion

In this section, the results obtained from the previously described approach are presented and discussed. Furthermore, it is important to extract the insights provided by each outcome, as well as the reasons behind them.

Regarding model (1), the simplest one, with 30 epochs executed, the test accuracy reaches its highest value on the 16th epoch: 77.43%. Since data augmentation was not used when training this model, it is easy to check on Figure 1 that the model overfits very rapidly, achieving a training accuracy of 100% after just 10 epochs, which stabilizes after that. It is noticeable that there is also a stabilization of the test accuracy approximately after the 10th epoch. Although a simple model that overfits the data, it was useful to set an initial performance benchmark to which we could compare the subsequent models.

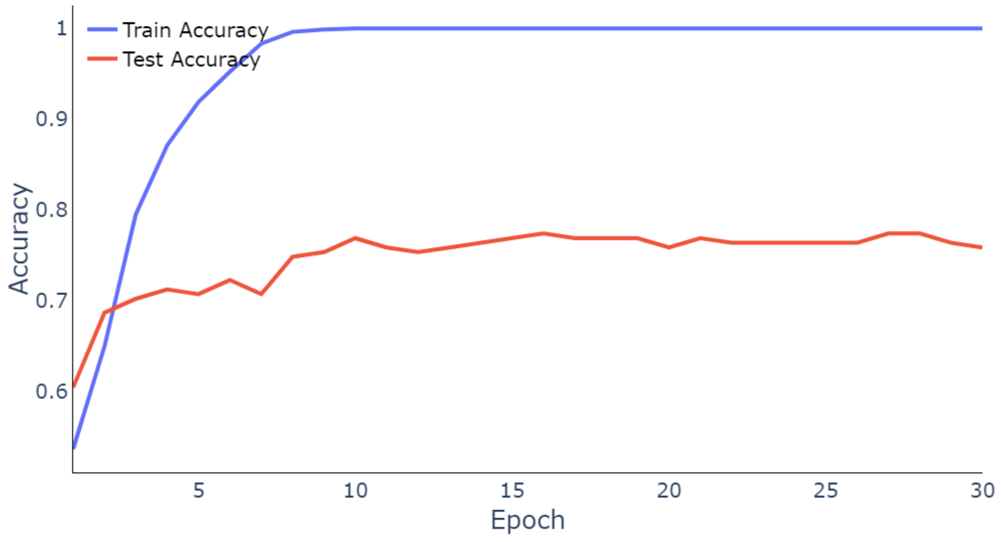

Figure 1 - Model (1) Accuracy plot

Concerning model (2), as explained previously, data augmentation was introduced as well as an increase on the input shape. This second model returned higher test accuracy values, reaching 87.69% on the 70th epoch. This increase in test accuracy from model (1) to model (2) was accompanied by a reduction on the training accuracies, given the introduction of data augmentation. When test accuracy reached its highest value, the training accuracy was 93.19%. Throughout all the epochs, it reached a maximum value of 93.57% on the 62nd epoch. Respecting to the test loss, it reached its minimum on the 34th epoch. These results confirmed what was expected – data augmentation reduced overfitting and improved the models' generalization ability.

The third model had two additional layers – one convolution and one pooling. As explained on the previous section, it was fitted several times (100 epochs each time) and the version of this model that returned the best test accuracy was saved. The accuracy and loss plots obtained from the saved version of model (3) are pictured on Figures 2 and 3, respectively. On the 81st epoch, the test accuracy reached a maximum value of 91.79%, with a training accuracy of 96.20% on this epoch. It is also possible to observe that the test loss reached its lowest value on the 47th epoch. After that, its irregular behaviour is easily observable with multiple spikes both up and downwards.

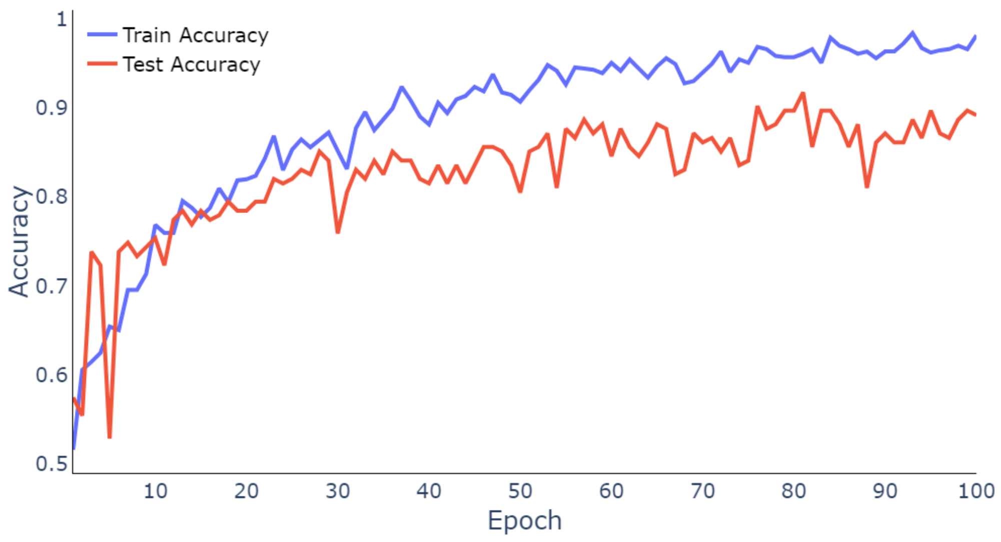

Figure 2- Model (3) Accuracy plot

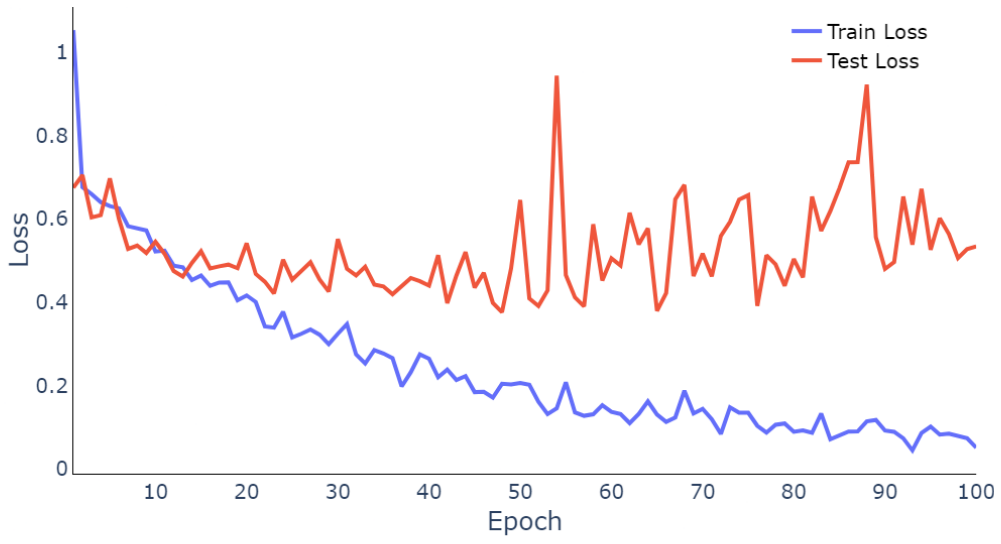

Figure 3 - Model (3) Loss plot

As mentioned previously, the tuning process was a time-consuming task. The resulting model's performance did not overtake the one of model (3), achieving a maximum value for the test accuracy of 85.64%. However, there are several plausible reasons for not getting improvements from the previous model performance, such as a non-optimal choice of parameters' ranges, as well as the low number of trials performed. Although a time-consuming and computationally expensive process, this is a step that is open for improvement, after which a better version of model (3) can be obtained.

Lastly, we used the pre-trained model *Xception* after implementing the transfer-learning process. With this model, 25 epochs were enough to achieve satisfactory test accuracy values, reaching a maximum value of 94.44% on the 20th epoch and a minimum test loss of 0.2246 on the 24th epoch. Furthermore, it is possible to observe on Figures 4 and 5 that the test and training accuracies and losses have a synchronized behaviour across all epochs, something that did not happen with the previous models.

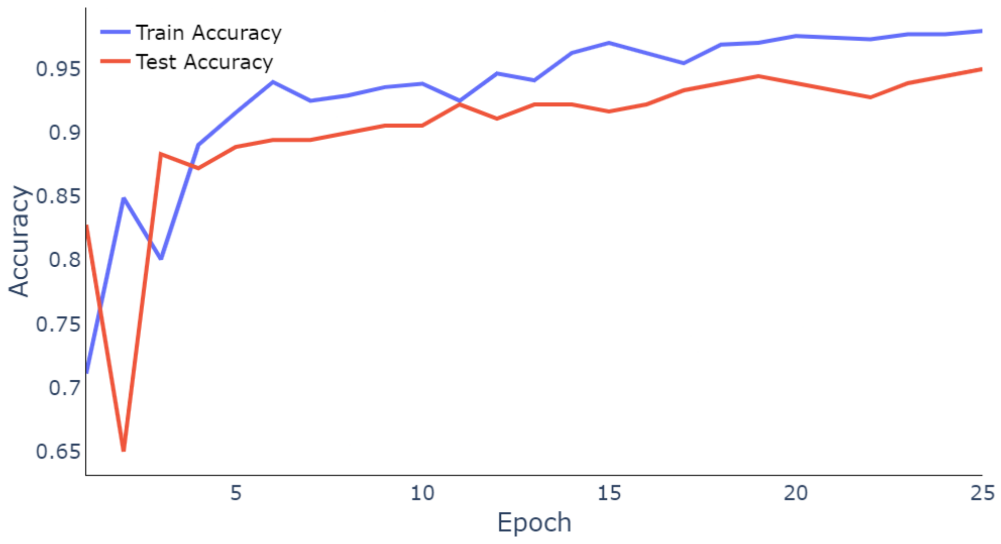

Figure 4 - Model (5) Accuracy plot

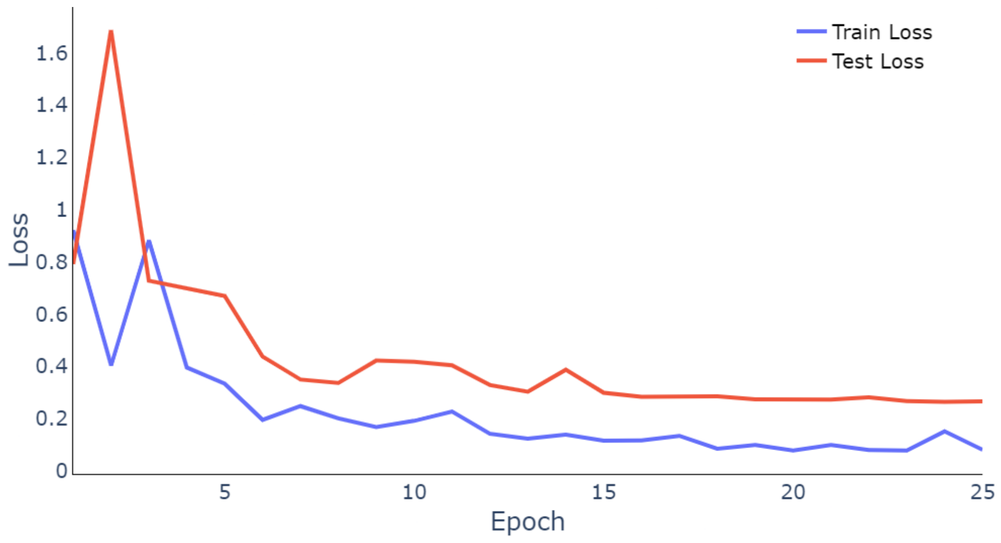

Figure 5 - Model (5) Accuracy plot

From this extensive process resulted two main models: Model (3), with an architecture set by us, and Model (5), a pre-trained model that was fit to our data via a transfer-learning process. Model (5) slightly outperformed Model (3), achieving a higher test accuracy. The classification reports of both models on the test data are below, on Table 1. Conscious of the limitations of not having train, validation and test data, these are still good performance indicators. The availability of more image data and a good filtration process are vital to ensure a more rigorous performance assessment that includes the 3 split categories.

	Model (3)			**Model (5)**			
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Support
Redentor	0.91	0.91	0.91	0.93	0.97	0.95	94
Rei	0.92	0.92	0.92	0.97	0.93	0.95	101
Accuracy			0.92			0.95	195
Macro Avg	0.92	0.92	0.92	0.95	0.95	0.95	195
Weighted Avg	0.92	0.92	0.92	0.95	0.95	0.95	195

Table 1 – Classification Reports of Models (3) and (5)

Besides comparing the results between the different models, we decided to visualize the heatmaps of class activation of our own built model (3). Below, examples of pictures from the test data are displayed with a heatmap of class activation on top of them.

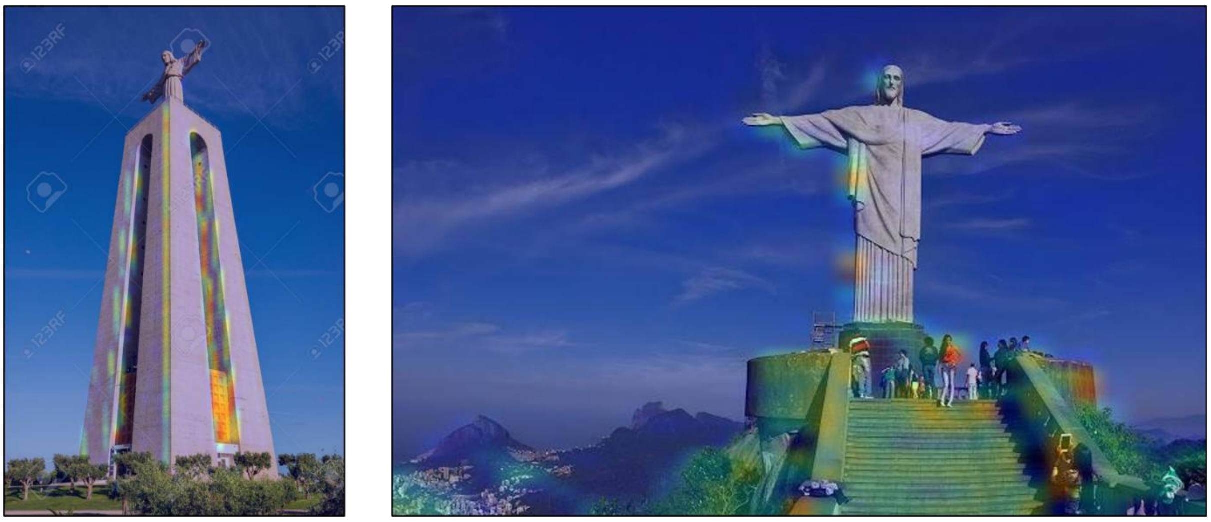

Figure 6 - Heatmap visualizations of Cristo Rei (left) and Cristo Redentor (right)

These heatmap examples help us to understand some of the patterns that allow the network to correctly classify the images. Patterns from the monuments or even from their close surroundings are clearly highlighted on the pictures. Nonetheless, there is some variability on the highlighted pattern from picture to picture of the same monument.

4. Conclusion

Developing an accurate model proved to be challenging from the beginning. Retrieving data from Google Image's search results was a time-consuming task due to the manual filtering process that we needed to apply to ensure data quality and coherence, which would not be possible with a large-sized dataset. After that, several models were tested and their performances were assessed, where we could check the consequences of having different architectures and including measures to reduce overfitting such as data augmentation. Our limited computational resources meant that some of the tests took a considerable amount of time to be performed, which transformed the development of the final mode into a slow-paced process.

Some of the results were already expected – such as reduced overfitting when including data augmentation or better performances when using more complex models. In the end, the best performing model was Model (5), a version of the pre-trained model *Xception* that was trained on our dataset. The resulting test accuracy was 94.44%, a favourable value given the considerable in-class variation between pictures of the same monument, as well as the low quantity of data.

Summing up, the system we proposed achieved satisfactory results, even though the low quantity of data did not allow for a more rigorous performance assessment that included train, validation and test data, instead of only train and test. Although a simple academic example, limited by the previously mentioned constraints, we believe that a system of this kind would be useful and could be applied to image web search engines to enhance the coherence and quality of the retrieved results.

5. Appendix

Model Summaries

Model (1)

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 126, 126, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 63, 63, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 61, 61, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 30, 30, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 57600)             0
_________________________________________________________________
dense (Dense)                (None, 256)               14745856
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 14,765,505
Trainable params: 14,765,505
Non-trainable params: 0
_________________________________________________________________
```

Model (2)

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 254, 254, 32)      896
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 127, 127, 32)      0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 125, 125, 64)      18496
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 62, 62, 64)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 246016)            0
_________________________________________________________________
dense_4 (Dense)              (None, 256)               62980352
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 257
=================================================================
Total params: 63,000,001
Trainable params: 63,000,001
Non-trainable params: 0
_________________________________________________________________
```

Model (3)

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 255, 255, 32)      416
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 127, 127, 32)      0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 126, 126, 64)      8256
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 63, 63, 64)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 62, 62, 128)       32896
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 31, 31, 128)       0
_________________________________________________________________
flatten_4 (Flatten)          (None, 123008)            0
_________________________________________________________________
dense_8 (Dense)              (None, 256)               31490304
_________________________________________________________________
dense_9 (Dense)              (None, 1)                 257
=================================================================
Total params: 31,532,129
Trainable params: 31,532,129
Non-trainable params: 0
_________________________________________________________________
```

Model (5)

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 256, 256, 3)]     0
_________________________________________________________________
xception (Functional)        (None, 8, 8, 2048)        20861480
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 2049
=================================================================
Total params: 20,863,529
Trainable params: 2,049
Non-trainable params: 20,861,480
_________________________________________________________________
```

Tuner Ranges

Conv2D Layer 1 – Number of filters: [32, 64], step = 16.

Conv2D Layer 3 – Number of filters: [64, 128], step = 16.

Dropout Layer – Dropout rate: [0, 0.2], step = 0.05.

Dense Layer 1 – Number of units: [64, 512], step = 64.