Sean Melone

CPE-628

**Introduction:**

“Dogs vs. Cats” is a Kaggle challenge that began in 2013, the task seems quite simple, you must write an algorithm to effectively classify images of either dogs or cats into their respective class, where ‘1’ is a dog and ‘2’ is a cat. The images cannot contain both a dog and a cat in the same image. This report will go into detail on how I wrote my Python deep learning algorithm, as well as logging any issues I face along the way.

**Method and Results:**

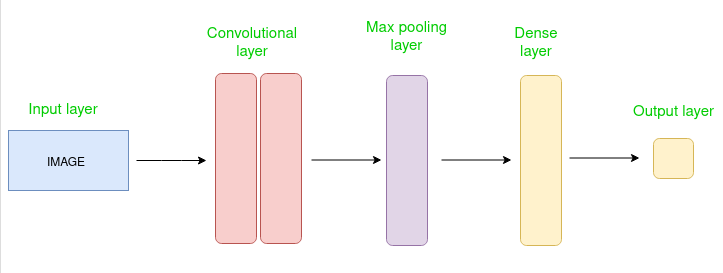
The very first task is to download and import the dataset. At first, I downloaded the data from <https://www.kaggle.com/c/dogs-vs-cats> and tried to run my notebook locally on a local Google Colab runtime, however, this was both complicated to set up, and slow, due to my standard MacBook M2 GPU. Compared to the hosted T4 GPU available to all Colab users, the M2 is very slow. Based on this information it using a hosted runtime on Google Colab was a no-brainer. As for how to get the images into Colab, there were two options I considered, the first was to locally download the dataset and upload it to Google Drive (which takes many hours given how large the dataset is), the next was to run a shell using python’s “os” library to download the dataset directly into the runtime. I did this using the “!wget” command and I learned it from watching a video on cats vs. dogs classification by “Google For Developers.” Link to video: <https://www.youtube.com/watch?v=nq7_ZYJPWf0>. I also referenced this video for a lot of other info on how to set up the program, especially for using the “os” library in Colab.

I wasn’t able to grab the exact version of the dataset provided by the Canvas assignment or the Kaggle challenge as both of those links would not work for me in Colab. Likely because you need to be logged in to access datasets on Kaggle, and Canvas is the same deal.

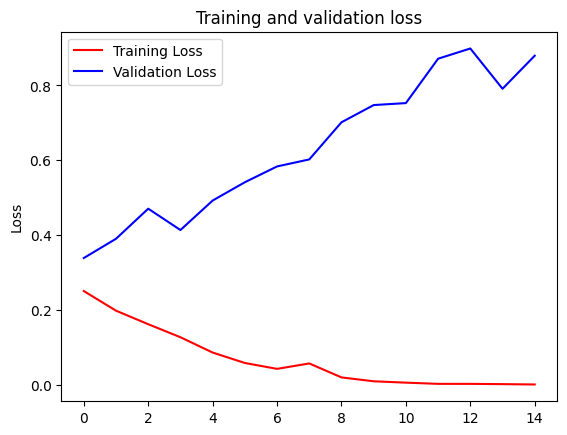
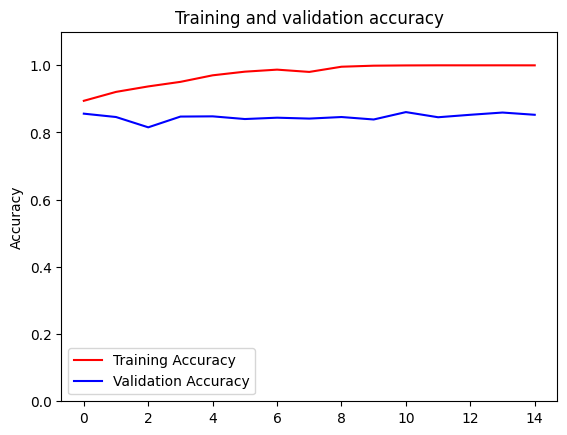
I chose to download the dataset from Microsoft, where it was organized into 2 folders named “cats”, and “dogs” each containing 12500 images for 25000 total. The last step needed before we can start having fun training the CNN is to preprocess the images using an image data generator. Without this step, all the images will be of varying sizes and we won’t have a uniform input shape for the CNN. We use the image data generator to convert each image to a uniform matrix of size 150x150. This is very easy to do with local runtimes as you can just use the Keras method “flow\_from\_directory” after sorting the cats and dogs of the training dataset into the proper folders, however it was trickier to do this on a cloud hosted runtime where I ran a script to download the dataset. I consulted the aforementioned Youtube video. It’s important to note that these are RGB images so color channels must be set to 3 in the generator.

After importing and preprocessing the dataset, I used a convolutional neural network (CNN) to complete the challenge. CNNs are an extremely powerful tool for image classification and it seemed the most reasonable way to solve the problem. I chose to start with a very basic CNN implementation, using Conv2D, MaxPooling2D, Flatten, and Dense. With the adam optimizer and binary\_crossentropy loss as we have 2 classes.

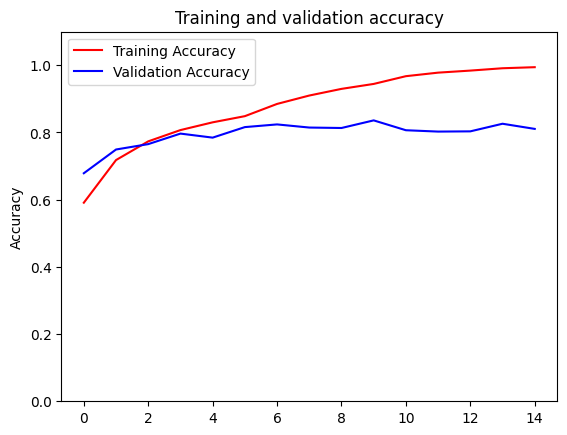
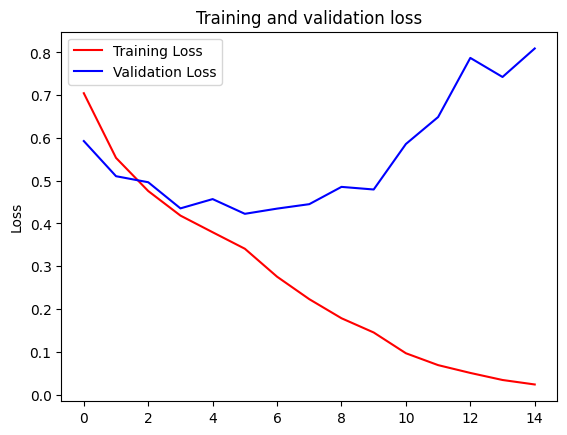
***Figure 1*** *(next pg): A basic visualization of the convolutional neural network via* [*GeeksForGeeks*](https://www.geeksforgeeks.org/introduction-convolution-neural-network/)

For the first run of the program I chose a simple model with 2 hidden layers and relatively high units. 32 units for the input, and 64 and 128 for the next 2 Conv2D layers, respectively. Lastly, 512 units of Dense were used after Flatten. With this model at 15 epochs, I was getting very high overfitting, and the accuracy left much to be desired. However, the goal of this very model was more to get an idea of how it behaves rather than achieve maximum accuracy. After examining the results of this model it was easier to understand what I need to tweak to reduce the overfitting and improve the accuracy.

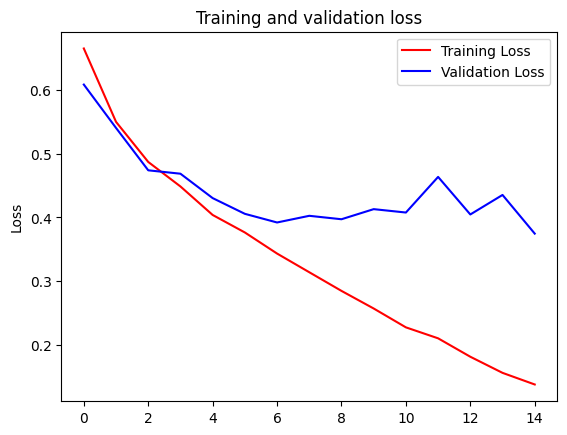
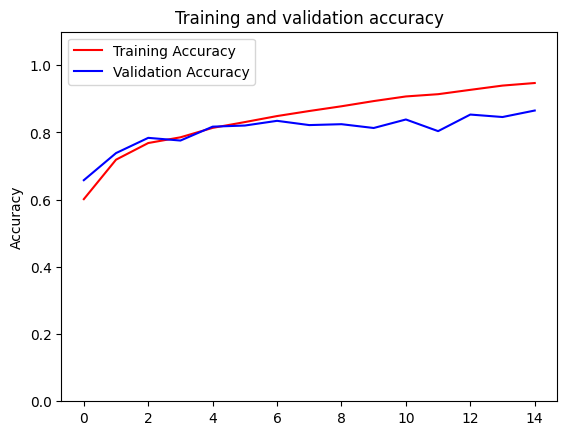
***Figures 2 and 3:*** Results for first model with 15 epochs.

As can be seen above, the model consistently overfits, we can even see the training accuracy reach 100% by epoch 8 while validation accuracy is hovering just above 80%. The first change I made to the model was to significantly lower the complexity. I did this by reducing the number of units in each Conv2D by 50%, this reduced the overfitting but had virtually no effect on the validation accuracy. In fact, it was slightly worse, but I still believed overfitting was the problem.

***Figures 4 and 5:*** Results for new model with above changes.



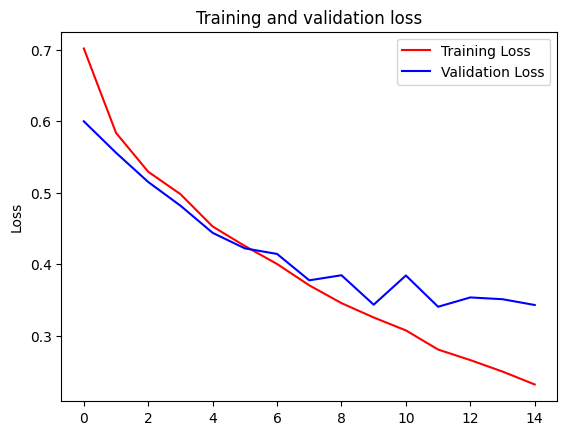
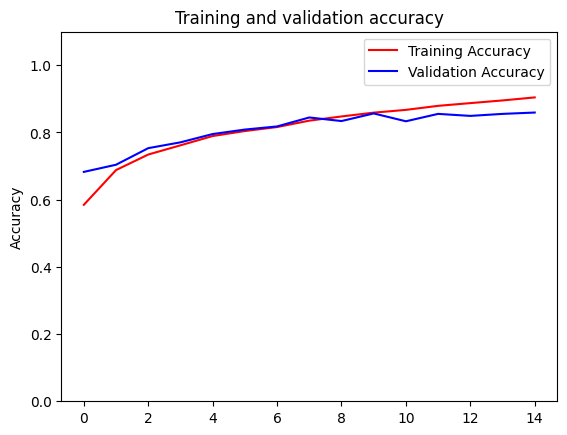
My next idea was to add some dropout layers. I chose to start with 2 layers of dropout with a rate of 50% and this significantly improved the fitting. The model achieved a validation accuracy of ~87% with these changes. The model was still overfitting but it was performing much better.

***Figures 6 and 7:*** Results for new model with added dropout.

Seeing the large effect it had on the fit was hardly a surprise, but I was blown away by just how well it performed with no further tuning. From here, I was happy with the dropout rate and decided to shift focus towards tweaking other parameters such as the hidden layer count or the optimizer.

The very first thing I tried after dropout was tweaking the Dense layer by reducing its units to 256, this returned slightly worse validation accuracy but it seems to have solved the overfitting problem.

***Figures 8 and 9:*** Results for tweaked Dense layer

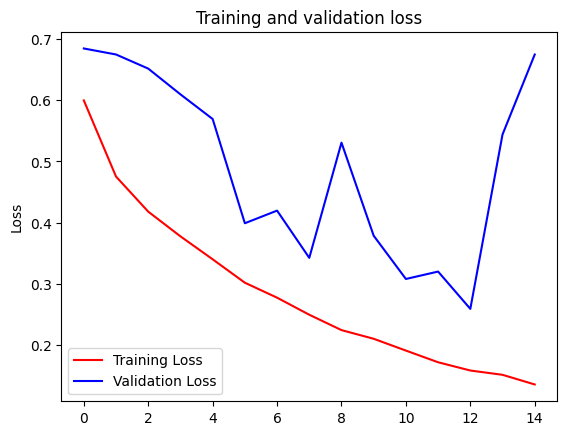
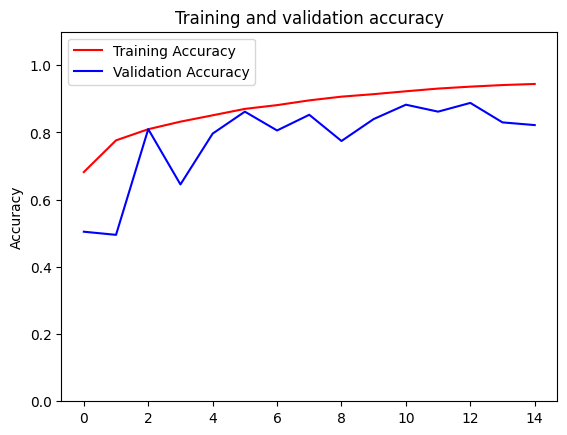
From here, there are a lot of changes that can be made to the model but the focus should be on improving the model capacity

My next idea was adding an additional hidden dense layer. Now that we have solved the overfitting, I want to improve the model capacity. For this, I first tried adding a hidden dense layer with 128 units and a dropout layer, but this did not improve the accuracy.

I decided to remove the hidden dense layer in favor of trying something else. The next thing I tried was adding a hidden convolutional layer with 128 units. This gave me very promising results and seemed to be just what my model needed to approch the 90% accuracy bar.

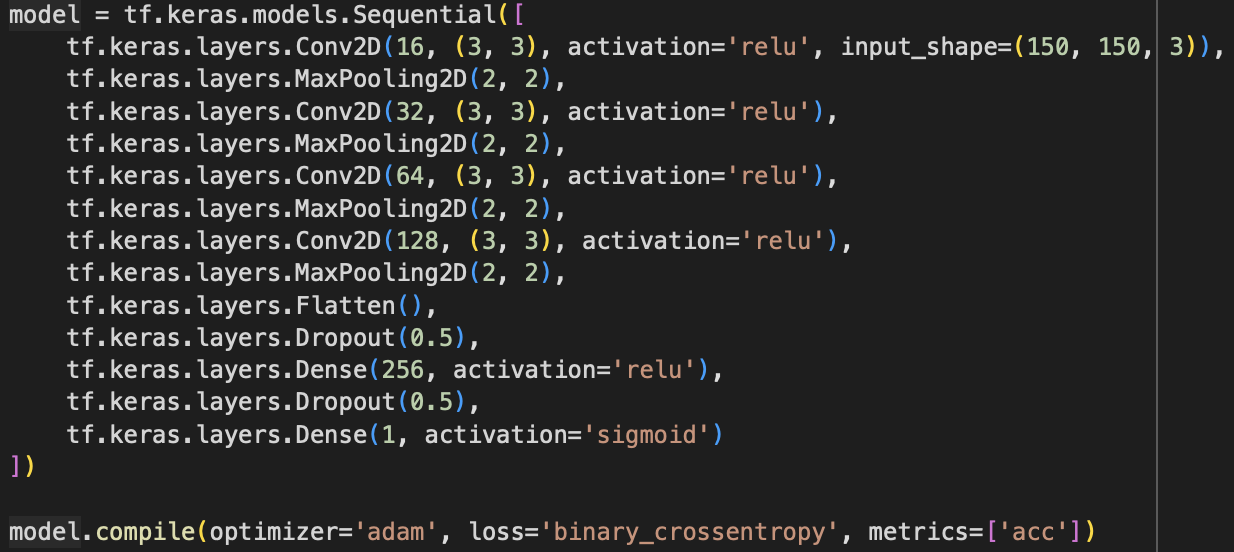
***Figures 10 and 11 (next pg):*** Results for added hidden convolutional layer

Here we can see that the model is slightly overfitting, but the validation accuracy is much improved. We can also see it might be time to reduce the number of epochs as the best performance usually occurs before the 13th epoch, but I would like to first try adding a layer of batch normalization first to reduce the overfitting and see how we do.

***Figures 12 and 13:*** Added Batch normalization  


Batch normalization ended up being quite a good solution. In the first 5 epochs, the validation accuracy aggressively fluctuated, but as we approached the 13th epoch it got a lot better, then in the final two epochs, the validation accuracy tanked. It’s quite an interesting solution, but I would need a lot more time to optimize it. Perhaps there is an optimal stopping point around 11 or twelve epochs before the model starts wildly overfitting. There’s potential with batch normalization, but I chose not to use it in my final model in favor of the accuracy provided by the model shown in figures 10 and 11. Before finalizing my CNN I played around with Root-Mean-Squared Propagation and Adamax, however, RMSprop requires tweaking of parameters such as learning rate and momentum for maximum performance, whereas adam works very well out of the box from my limited experience.

***Figure 14:*** Implementation of the final convolutional neural network.

**

***Figure 15:*** Final plot of accuracy.

Epoch 15/15

90/90 [==============================] - 51s 566ms/step - loss: 0.1623 - acc: 0.9355 - val\_loss: 0.2465 - val\_acc: 0.8847

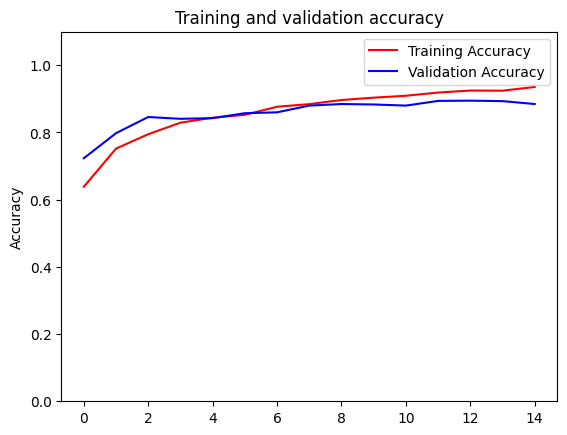


Image classification tasks:

1/1 [==============================] - 0s 19ms/step

Dog image: 10849.jpg

10849.jpg is a dog

1/1 [==============================] - 0s 20ms/step

Dog image: 1368.jpg

1368.jpg is a dog

1/1 [==============================] - 0s 22ms/step

Dog image: 1239.jpg

1239.jpg is a dog

1/1 [==============================] - 0s 23ms/step

Dog image: 11353.jpg

11353.jpg is a dog

1/1 [==============================] - 0s 21ms/step

Dog image: 246.jpg

246.jpg is a dog

1/1 [==============================] - 0s 21ms/step

Cat: 8757.jpg

8757.jpg is a cat

1/1 [==============================] - 0s 22ms/step

Cat: 3706.jpg

3706.jpg is a cat

1/1 [==============================] - 0s 24ms/step

Cat: 6457.jpg

6457.jpg is a dog

1/1 [==============================] - 0s 19ms/step

Cat: 3874.jpg

3874.jpg is a dog

1/1 [==============================] - 0s 18ms/step

Cat: 974.jpg

974.jpg is a cat

**Conclusion:**

There is no real “correct” way to solve this Kaggle challenge, many methods can eclipse 90% accuracy, I only tried out a few. I didn’t even try one of the most popular methods for this challenge: transfer learning. Overall, this was a great learning experience for me and I am much more confident in the field of deep learning after this experience.