Image Classifier reflection

In the development of the Image Classifier, several choices and adjustments were made in the development to achieve the optimal result. These will be covered in this reflection where the process is explained as parameters and values are adjusted.

The first stage was to download the dataset provided and look through the images to delete the ones irrelevant, messy and corrupted to better provide a “clean” dataset for the model. None of the classes were similar, but one could already see how a deer image with a lot of background could be mistaken for a mountain, or how a bicycle with a human could be a confusing aspect for the model. Therefore, the images were cleaned up to restrict those mistakes. To provide the model with enough data, more pictures were downloaded from Kaggle to include additional data. The aim was to include around 300 images in each folder/class to be classified.

The project given was to “design, write, and train an ML (Machine Learning) model that will perform classification of the provided image data.” The choice was made to use Keras in the creation of said model as it is easy to read and write, as well as good for smaller data sets, compared to the difficulty of learning TensorFlow. Keras acts as an interface for the TensorFlow library.

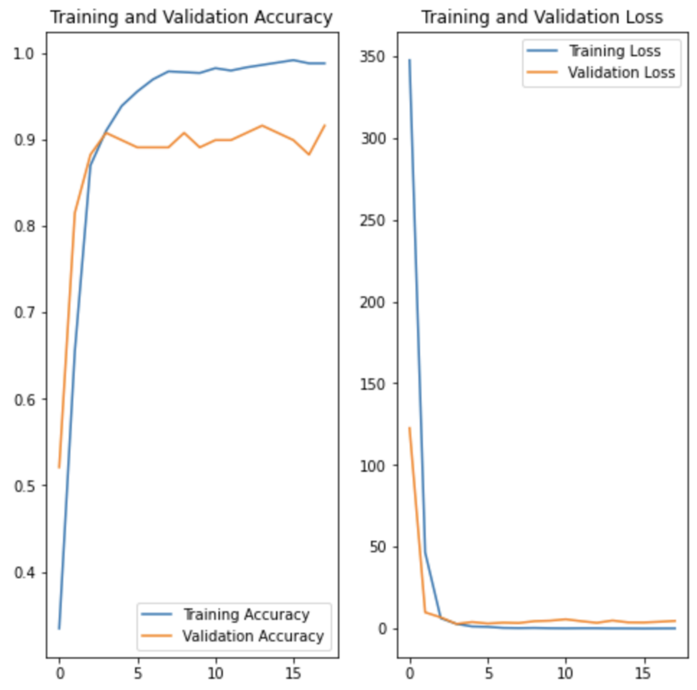
Previously in the semester I was unable to use Jupyter and ended up using Colab instead, proving to be easier to use. I then loaded the images by importing google drive and started preprocessing to improve overall data quality by resizing images, making sure they all have the same image extension and color space. Images were resized to 150x150 to avoid the increasing of size and further obscuring of key features for the model to learn. It was then time to load the images where labels were chosen to be inferred and categorical to correspond with the loss function. Batch size was an original 32 but changed to 64 as it increased the accuracy. Class names were then defined and printed to later be used in the prediction of an input image.

The loading of the model has a lot of parameters to experiment with. Experiments were not made as to number of pooling, or dense, layers, but rather on the values. For the convolutional 2D layer, the kernel size was two different sizes to see the effect on results. There’s a preference for smaller size kernels, so the sizes implemented were 5x5 and 3x3. With the size of 5x5, the validation accuracy went down, so 3x3 was shown to be the better option. ReLU was chosen as it’s most used, as well as fast and easy to compute by not allowing for the activation of all neurons at the same time. Max pooling extracts features of images and the attempt was made to retain more details by lowering the pool size to 3x3m, yet it resulted in the validation accuracy going down, even though the validation loss also resulted in lower values. The pool size was then increased to 5x5 to rid more details, yet the accuracy was still low, resulting in the 4x4 value, as proven to be the best option in keeping validation accuracy high. The next value to experiment with was the percentage of the dropout layer. There was originally augmentation in the form of random flip, rotation and zoom, yet the dropout layer seemed extensive enough, resulting in the removal of said layers. Changing of dropout layer value did not result in any substantial change, other than the validation accuracy hitting a plateau at 0.88 for six epochs and a lower validation loss. Yet with the value set to 0.5, the validation accuracy would in some epochs go as high as over 0.91 and was therefore the final value. In the dense layer, the SoftMax activation function was chosen as it applies to multi class problems. Yet, there was some confusion around Sigmoid and how it could be another alternative. However, as it is a multi-label classification, SoftMax activation function was the better choice. 4 was the number of units chosen in the layer as the changing of it to another number, ex. 16, required the loss function to be changed to sparse categorical cross entropy. It was therefore kept at 4.

Next step was then to compile the model by optimizing it to find the “just right” of the model. Adam was chosen as the optimizer by being computationally efficient by adding to the advantages of Adadelta and RMSprop. It is also an adaptive algorithm self-tuning during training without the need of changing hyperparameters to tune. However, the value was changed from the default 0.001 to 0.1 to see how it would affect the results, with an extreme dip in validation accuracy to 0.37. The loss function was set to categorical cross entropy as it's a multi class classification.

It was then time for the time consuming process of training the model. The number of epochs depends on the complexity of the dataset, yet the starting point with 20. One goal was to keep the training time from being incredibly long, so an attempt was made to decrease the number of epochs while keeping accuracy. When decreased to 15 the gap between training and validation accuracy increased and validation loss increased. When upped to 30 epochs, the gap between training and validation increased a great deal in both validation and loss. The loss in training remained low while in validation increased, as well as training accuracy remained high while validation accuracy was on a lower scale fluctuating. This could be a sign of overfitting. 18 epochs provided about the same accuracy as 20, yet the time difference in training was not significant, so the final number of epochs ended up at the original 20 as it resulted in best accuracy.

The model was then visualized in a simple graph showing the development of training accuracy and loss, as well as the validation accuracy and loss from each epoch. It was then saved for further use. Results were a validation accuracy of 0.91 and a validation loss of 4.7.



In a separate file, libraries and google drive were imported to be able to then load the image classifier model previously developed. Further, the data needed for the prediction was loaded, as well as a picture uploaded as input into the model. A prediction was then printed as to what class the image classifier believed the image should be classified as. The results were very accurate. However, testing of the model's understanding of the images was done to see how it would classify images with greater complexity. Examples as to wrong classifications were when a human was riding a bicycle. Humans were not included in the training dataset and led to the model not being familiar with that feature. It ended up classifying the picture belonging in the bicycle class as a deer. Another example was a picture of a deer with a larger percentage of background mistaken for a mountain instead. In the training dataset, deer images were primarily closer images to better focus on the deer and not the background of the image. When a larger percentage of the image was nature, the model wrongly classified it as a mountain.