Dog identification

Introduction

There are many dog breeds with similar characteristics where even the human eye cannot distinguish. This project, using the data set of a kaggle competition set 5 years ago, attempts to predict the breed of a dog in each picture.

Background

The problem seen above is a multi-classification problem, and relatively massive. Where usually the number of classes is around 5-10. This data set needs to be classified into 120 labels.
The given data set contains 10,222 pictures of dogs, in various positions and environments, and different image sizes. On average there's 85 images per label where the maximum sized label has 120 images and minimum sized label has 60 images, so the data can be more balanced but doesn't have to.

Project description

In the final attempt, the model was built using CNN and TensorFlow 2.

The highest accuracy model was built with 5 starting layers of:

- Convolution with increasing filter each time.
- Batch normalization.
- Max pooling.
- Spatial dropout.

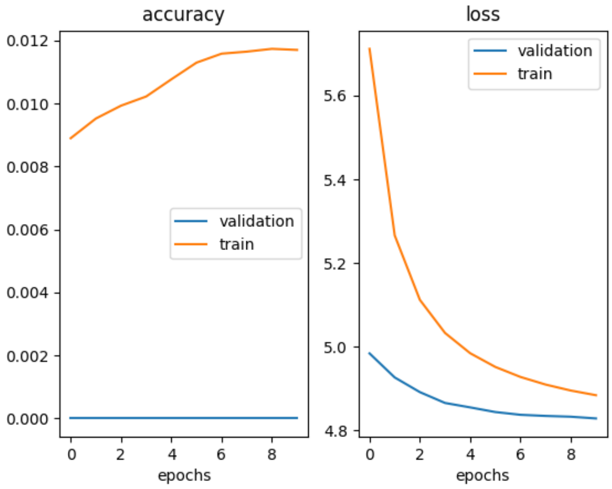
Then instead of using flatten, GlobalAveragePooling was used to collapse the given data from say [5 by 5 by 64] to [1 by 1 by 64].

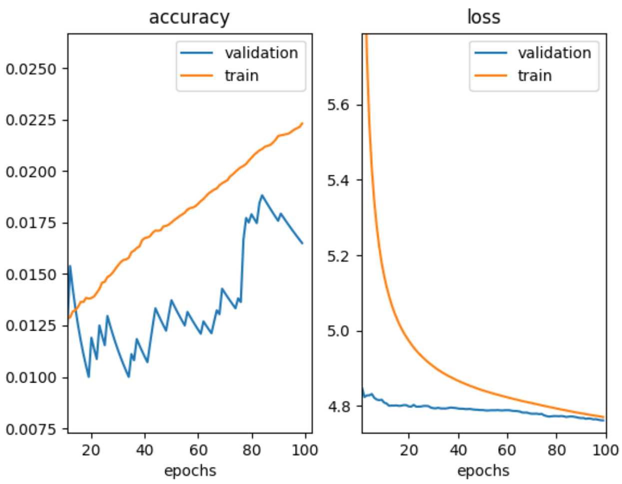
Then it is connected to a Dense layer, then a dropout, and finally connected to the final layer, activated by SoftMax.

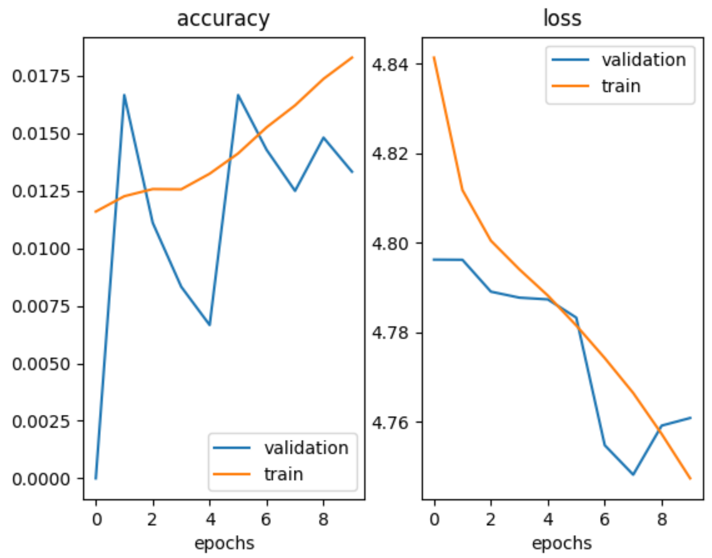
After iterating for around 2000 epochs the model reached a validation accuracy of around 17%. To prevent overfitting, we have used dropout, and image augmentation. For image augmentation we have used the built-in keras library to perform zoom and rotation on the input training images. This helped us also with the issue of a small sample size and imbalanced data, as mentioned earlier.

Previous attempts (Exercise 2)

To optimize the learning process and shorten the waiting time per session in these attempts, the images were resized to predetermined bounds(200x200) and saved as one color-channel image (black & white) before starting the learning process.

<table>
<tr>
<td>
</td>
<td>At first, the model was built with a given input (optimized image), and an output (1 x 120) with no layers in-between.

During the data iteration, it could be inferred that the training loss and accuracy increased, but due to the simplicity of the model, the accuracy of the validation group was non-existent. Which means this is a form of over-fitting.</td>
</tr>
<tr>
<td>
</td>
<td>In the next attempt, adding one layer between the input and output, the train graph has a good trend going on both in accuracy and loss. However, validation is messier but loss has shown decrease and overall accuracy seems to rise</td>
</tr>
</table>



For the final attempt we have placed 3 hidden layers between the input and output.

It can be inferred that it is an improvement but not enough, as the model was too simple for such a task.

This specific 5 layer mode was running for too low a number of epochs due to time constraints.

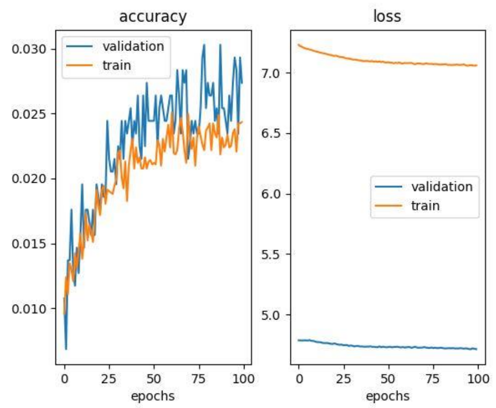
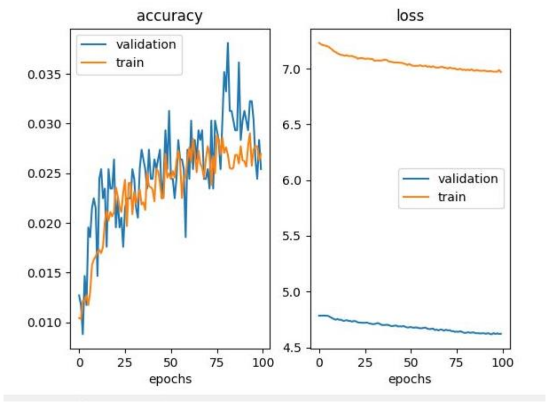
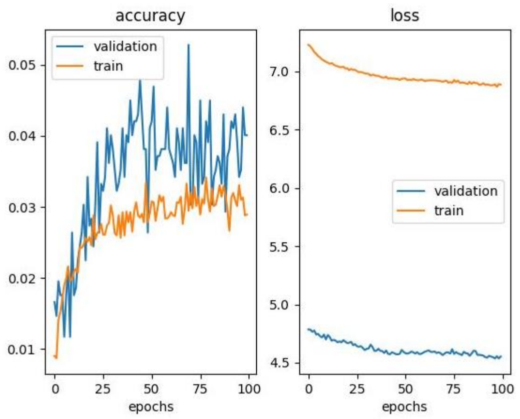
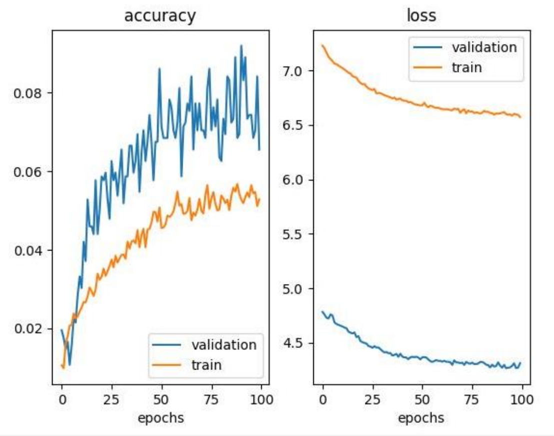
Experiments and their results

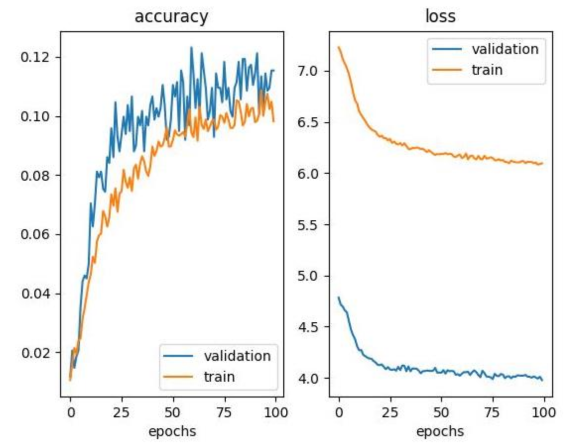
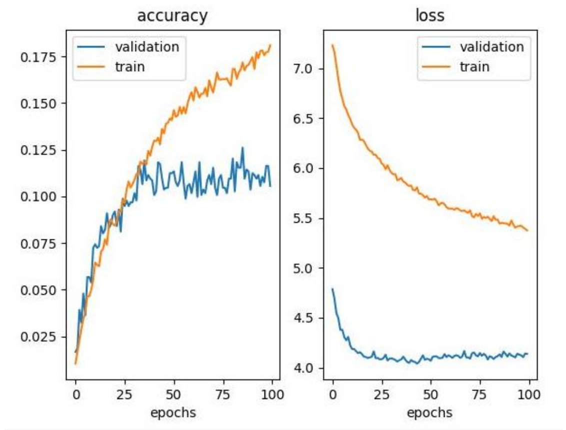
In this attempt, using CNN network, the approach was starting from small and simple, and each time the model was changed slightly.

Starting with one layer of convolution, max pool, into spatial dropout. Each time adding a layer of convolution, max pool and dropout until no increase in performance is seen.

Following that, a fully connected layer of dense, to drop out, to the final layer activated by softmax and giving the output.

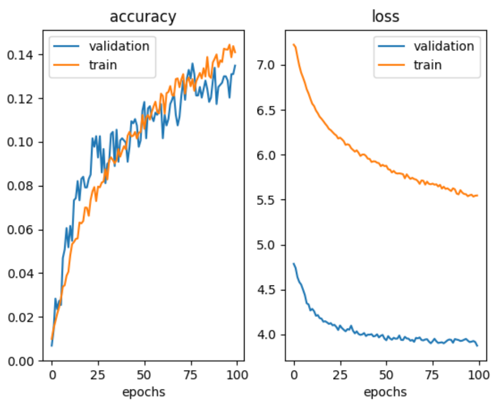
After constructing the model, class weights were added to reduce the effect of the data imbalance. The class with the highest number of pictures is valued at 1.
The rest are valued at a number greater than 1, depending how low their amount of pictures is. For example, the class with the least amount of pictures is the most valuable.

	
one layer, has better performance then the simple MLP.	Two layers, minor improvement.
	
Three layers, higher accuracy than earlier, minor under-fitting.	four layers, a clear indication of under-fitting, meaning more layers can be added.

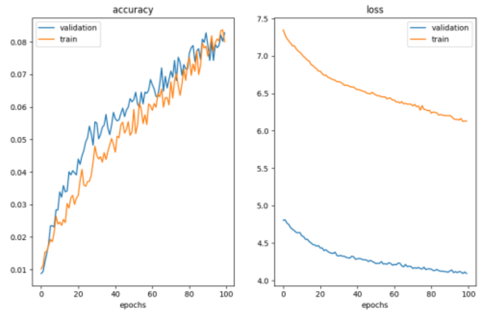
<table>
<tr>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td>five layers, more balanced trend among both train and validation.
Losses have a similar curve.</td>
<td>Six layers, overfitting, Network is too deep.

The previous model will be improved upon.</td>
</tr>
</table>

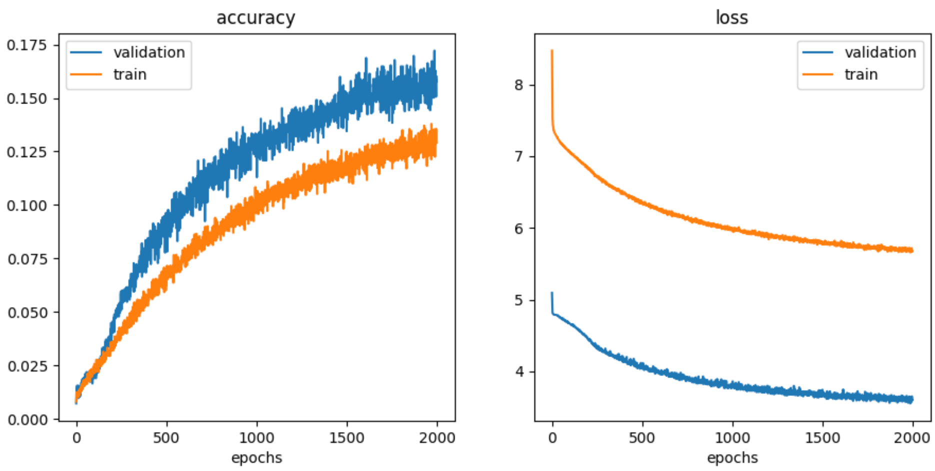
Using the 5-layer model



Adding a dense layer of 1200 neurons.



Adding batch normalization between convolutions and dense.



Adding regularizer L2 to the Dense layer.
Final, best result.

Conclusions

Although the final model is still at a low accuracy percentage, 17% of 120 classes is reasonable achievement for a non-pretrained model. Furthermore, it has shown reasonable, generalized learning.

Code:

```
def get_model():
    activ = ReLU()
    reg = regularizers.l2(0.001)
    drop_rate = 0.25
    filter_size = 8
    ## Sequential : Single inupt --> Single output (Image -> breed)
    model = Sequential()
    weight_init = tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.01, seed=4)
    bias_init = tf.keras.initializers.Zeros()

    ## Add Layers
    model.add(Conv2D(filter_size, kernel_size=(3, 3), strides=(1, 1), padding='same',
                     kernel_initializer=weight_init, kernel_regularizer=reg,
                     bias_initializer=bias_init,
                     input_shape=(IMG_HEIGHT, IMG_WIDTH, CHANNELS)))
    model.add(activ)
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=2, padding='same'))
    model.add(SpatialDropout2D(drop_rate))

    ## Add several layers, each time increase the filter size to extract different features.
    while filter_size < 128:
        filter_size *= 2
        model.add(Conv2D(filter_size, kernel_size=(3, 3), strides=(1, 1), padding='same',
                         kernel_initializer=weight_init,
                         kernel_regularizer=reg, bias_initializer=bias_init))
        model.add(activ)
        model.add(BatchNormalization())
        model.add(MaxPooling2D(pool_size=2, padding='same'))
        model.add(SpatialDropout2D(drop_rate))

    model.add(GlobalAveragePooling2D())
    model.add(Dense(1200, activation=activ, kernel_initializer=weight_init,
                     kernel_regularizer=reg, bias_initializer=bias_init))

    model.add(Dropout(drop_rate))

    model.add(Dense(BREEDS_NUM, activation="softmax"))
    return model
```

More can be seen [here.]()