Neural Networks & Genetic Algorithms Project
Cats vs Dogs Image recognition AI

The Data Set

For this project, I chose the Cats vs Dogs dataset from Kaggle (https://www.kaggle.com/competitions/dogs-vs-cats) as it is a widely used and popular benchmark in the field of computer vision and image recognition. It consists of a large collection of images depicting cats and dogs, with the task of classifying each image into one of the two categories. This dataset serves as a fundamental example of binary classification and has been instrumental in evaluating the performance of image recognition algorithms.

The dataset is made of 25000 color images of varying sizes, in the JPEG format. The images depict domestic cats and dogs in various poses, environments, and backgrounds. The diversity of images in terms of lighting conditions, angles, and appearances presents a realistic and challenging scenario for image recognition models.

To expand the dataset and improve model performance, data augmentation techniques are often applied. These techniques involve applying transformations such as rotation, scaling, flipping, and cropping to generate additional training samples, increasing the diversity of the dataset and helping the models learn robust features.

The Implementation

I started this project from an image recognition model I found on the web, designed to recognize the letters of the sign language (https://www.analyticssteps.com/blogs/hand-gesture-classification-using-deep-learning-keras), along with an image recognition model that was meant to differentiate between the rock, paper, scissors hand sign game (https://www.analyticsvidhya.com/blog/2021/07/step-by-step-guide-for-image-classification-on-custom-datasets/). From there, I changed different attributes and functions until I reached a few options that I compared.

The implementation of this application is comprised of four stages: the pre-processing of the images, the model building, the model training and the model evaluation.

To properly explain the application and how it works, I will take each of the stages and expand on them separately.

Pre-Processing of the Images

This dataset consists of 25 000 colored images of cats and dogs, out of which 12 500 are pictures of cats and 12 500 are pictures of dogs. The images are all in one folder named ‘train’ and each image is named ‘animal.no.jpg’.

To separate the train and test data, I chose an additional cats vs dogs dataset (https://www.kaggle.com/datasets/tongpython/cat-and-dog) to be the test set. This set contains 2023 images out of which 1011 are ctas and 1012 are dogs, each category separated in its own folder.

For the train folder, as all the images were all in one big folder, I created an additional cat and dog folder and manually sorted the images in each one. The validation set is manually created as well as a directory separated into two folders containing the images. The validation directory has 5000 images while the train directory has 20 000 images.

Model Building

For the model I chose to build a sequential model as it provides a straightforward and intuitive way to stack layers one after another in a sequential manner.

The Sequential method serves as a container for linear stacks of layers, where the output of one layer becomes the input of the next layer. It is primarily used for building sequential models, where the flow of data moves forward through the network without any branching or skipping connections. The order in which the layers are added determines the flow of data through the network.

For the layers, I added convolution, max pooling, flatten and dense as main components and I experimented with adding batch normalization and dropout.

Convolution takes in an input image, assigns importance (learnable weights and biases) to various aspects/objects in the image, and is able to differentiate one from the other. It involves sliding a small filter/kernel over the entire input image and computing element-wise multiplications and summations at each position. The filter is typically a small matrix with weights. The size of the filter is smaller than the input image.

In the case of color images, each image typically consists of three color channels (red, green, and blue). Convolution is applied independently to each channel of the image, and the resulting feature maps are stacked together to create a 3D tensor.

Max Pooling is a pooling operation that calculates the maximum value for patches of a feature map, and uses it to create a downsampled (pooled) feature map. It adds a small amount of translation invariance - meaning translating the image by a small amount does not significantly affect the values of most pooled outputs.

Flatten converts a multi-dimensional matrix to a single dimensional matrix. It collapses the multi-dimensional input into a single dimension while preserving all the elements.

Dense Layer is a simple layer of neurons in which each neuron receives input from all the neurons of the previous layer, thus called dense. Dense Layer is used to classify images based on output from convolutional layers**.** The input to each neuron is a weighted sum of the outputs from the previous layer, which is passed through an activation function.

The last layer is a Dense layer and is used as the output layer for multi-class classification problems, where the number of units matches the number of classes. Softmax activation function is used when we have 2 or more than 2 classes. Softmax converts logits into probabilities. RMSprop optimizer uses an adaptive learning rate instead of treating the learning rate as a hyperparameter. This means that the learning rate changes over time.

Normalization is a data pre-processing tool used to bring the numerical data to a common scale without distorting its shape.

Batch normalization is a process to make neural networks faster and more stable through adding extra layers in a deep neural network. The new layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer. As a result, the next layer receives a “reset” of the output distribution from the preceding layer, allowing it to analyze the data more effectively.

Dropout refers to data, or noise, that's intentionally dropped from a neural network to improve processing and time to results.

The term “dropout” refers to dropping out the nodes (input and hidden layer) in a neural network. All the forward and backwards connections with a dropped node are temporarily removed, thus creating a new network architecture out of the parent network. The nodes are dropped by a dropout probability of p. A dropout is a regularization approach that prevents overfitting by ensuring that no units are codependent with one another.

The last part of the model is the compilation. For this I used the compile function To configure the compilation step. For this I specified the loss function which is categorical cross-entropy as it is commonly used for multi-class classification problems, where the target variable has multiple classes, the optimizer RMSprop(Root Mean Square Propagation), as it is an adaptive learning rate optimization algorithm and it helps accelerate convergence and handle different learning rates for different weights, and the metrics accuracy since it measures the proportion of correctly predicted samples over the total number of samples.

For the categorical cross-entropy, each predicted class probability is compared to the actual class desired output 0 or 1 and a score/loss is calculated that penalizes the probability based on how far it is from the actual expected value. The penalty is logarithmic in nature yielding a large score for large differences close to 1 and small score for small differences tending to 0.

Additionally, after the model compilation, I defined two callbacks for the purpose of monitoring the validation accuracy, stopping training early if necessary, and adjusting the learning rate if the model's performance plateaus. These callbacks are the EarlyStopping and the ReduceLROnPlateau.

EarlyStopping is a callback function used to stop the training process if a monitored metric stops improving. The purpose of early stopping is to prevent overfitting and save computational resources by stopping training when further training does not lead to significant improvement.

ReduceLROnPlateau is a callback function that reduces the learning rate when a monitored metric plateaus (stops improving). The purpose of reducing the learning rate is to fine-tune the model's performance by allowing it to make smaller adjustments when progress slows down, potentially improving convergence and finding better optima.

The learning rate regulates the weights of our neural network concerning the loss gradient. It indicates how often the neural network refreshes the notions it has learned.

Model Training

For training I need to create the training and validation generators. To do this, I used ImageDataGenerator, a utility in Keras that allows you to generate augmented data by applying various transformations and modifications to the input images during training. It helps to increase the diversity of the training data and improve the model's ability to generalize.

As transformations, I applied rotations, scaling, shearing, zooming, flipping, and shifting. These transformations and modifications help introduce variations in the training data, making the model more robust and capable of generalizing to different instances of the same object.

Once the transformations are defined, I create the train generator which is an iterator that generates batches of augmented image data during training. It takes the input images, applies the specified transformations, and returns them in batches along with their corresponding labels.

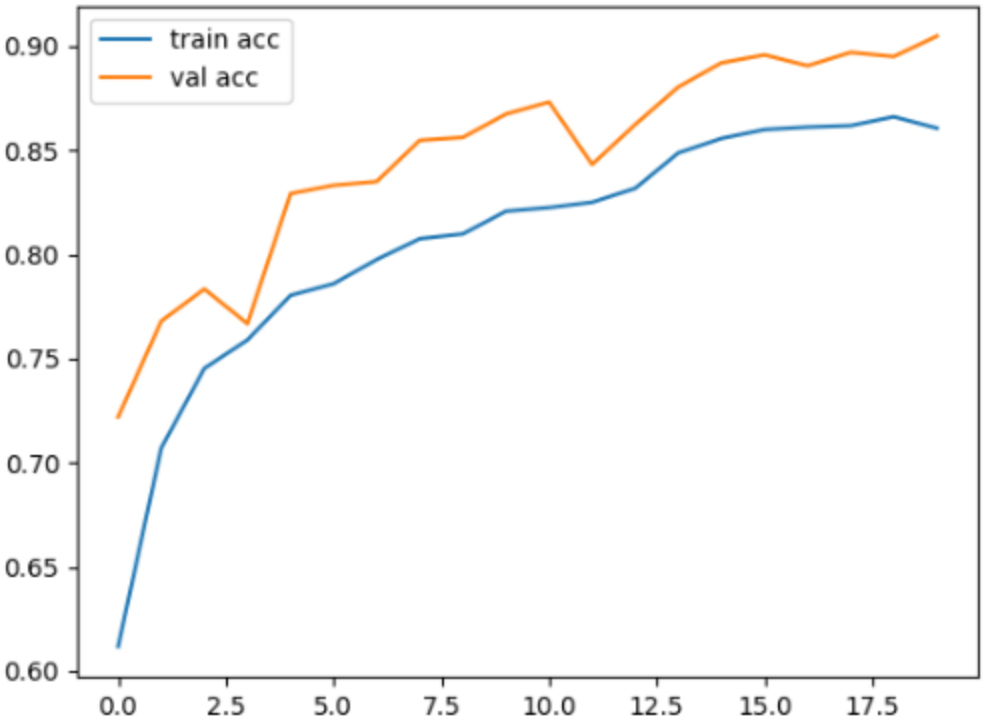
This process is repeated for the validation generator, however the only transformation added to the data in the validation is a scaling. This is because the validation data should reflect the real-world distribution of the data as closely as possible and scaling is a normalization step used to ensure that the input data has a consistent scale.

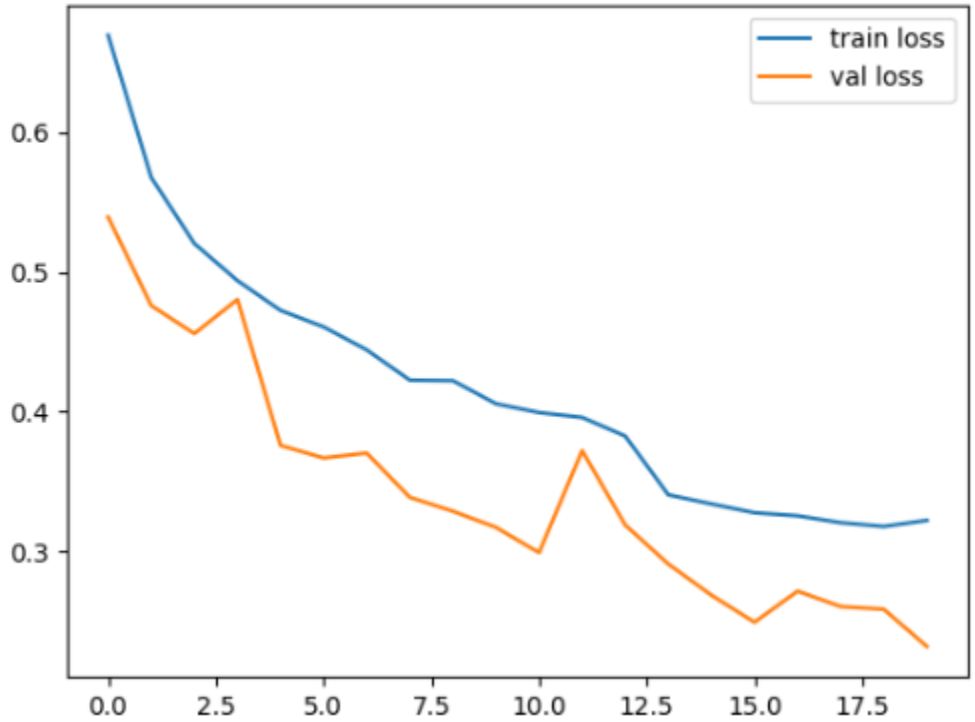
Once the generators are created, I can train the model using the fit() method. The method takes the training data generator as input and iteratively performs training epochs, evaluating the model's performance on the validation data at the end of each epoch. The training process is guided by the specified number of epochs and callbacks.

Model evaluation

To evaluate the performance of the model, I created another generator, a test generator using the images from the test_set. The methods creating the generator are the same as the ones used for creating the validation generator.

To see the performance of the model on training and validation datasets,I used accuracy and loss graphs to visualize it.

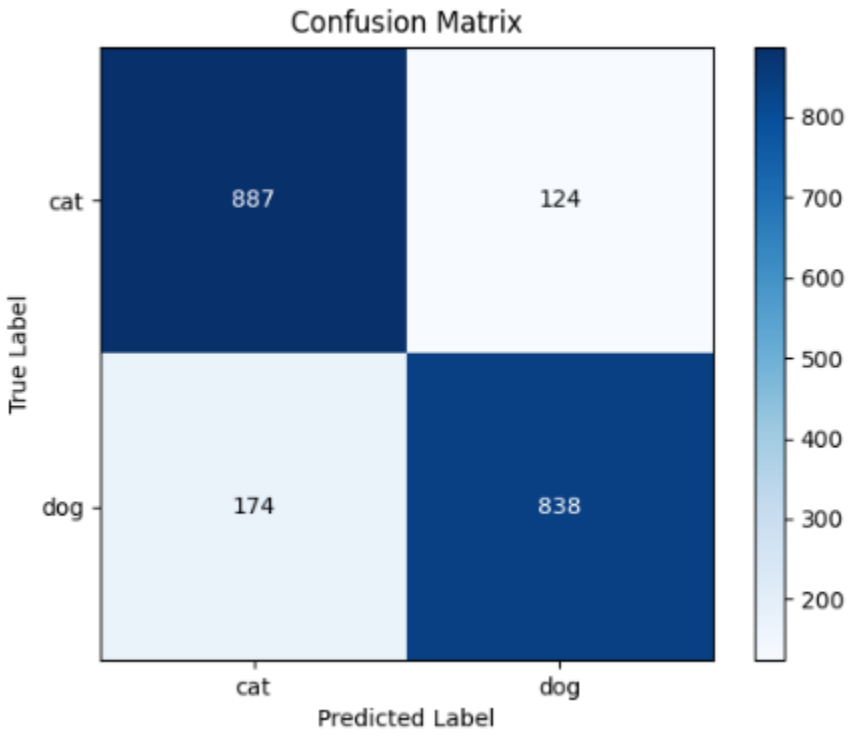

train acc
val acc
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


train loss
val loss
0.6
0.5
0.4
0.3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

I used the evaluate function which returns the computed loss value and the evaluation metrics as specified during model compilation, in my case that is the accuracy. This output provides a quantitative measure of how well the model performs on the given dataset.

```
Found 20000 images belonging to 2 classes.
Found 5000 images belonging to 2 classes.
Found 2023 images belonging to 2 classes.
64/64 [==============================] - 4s 54ms/step
Test loss: 0.20230890810489655
Test accuracy: 0.928324282169342
```

To better see the accuracy, I also used a confusion matrix on the test images. For this I noticed that when using the test_generator to get the predictions, the matrix never matched the accuracy given by the evaluate method. As such, I decided to create a list of images that was to be predicted.



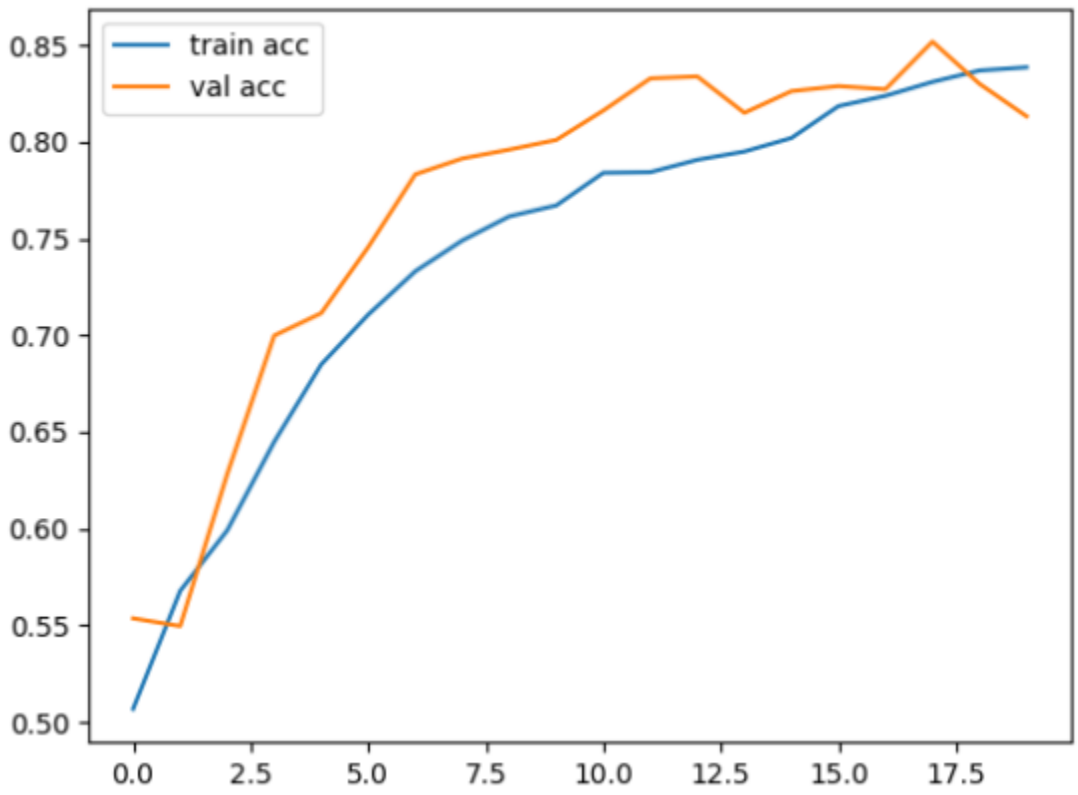
Additionally, I added a user interface functionality inside the Run Terminal, such that a user can add images to a specified path and then enter the name of the image in the Terminal. The program will return a prediction of whether the image is a cat or a dog.

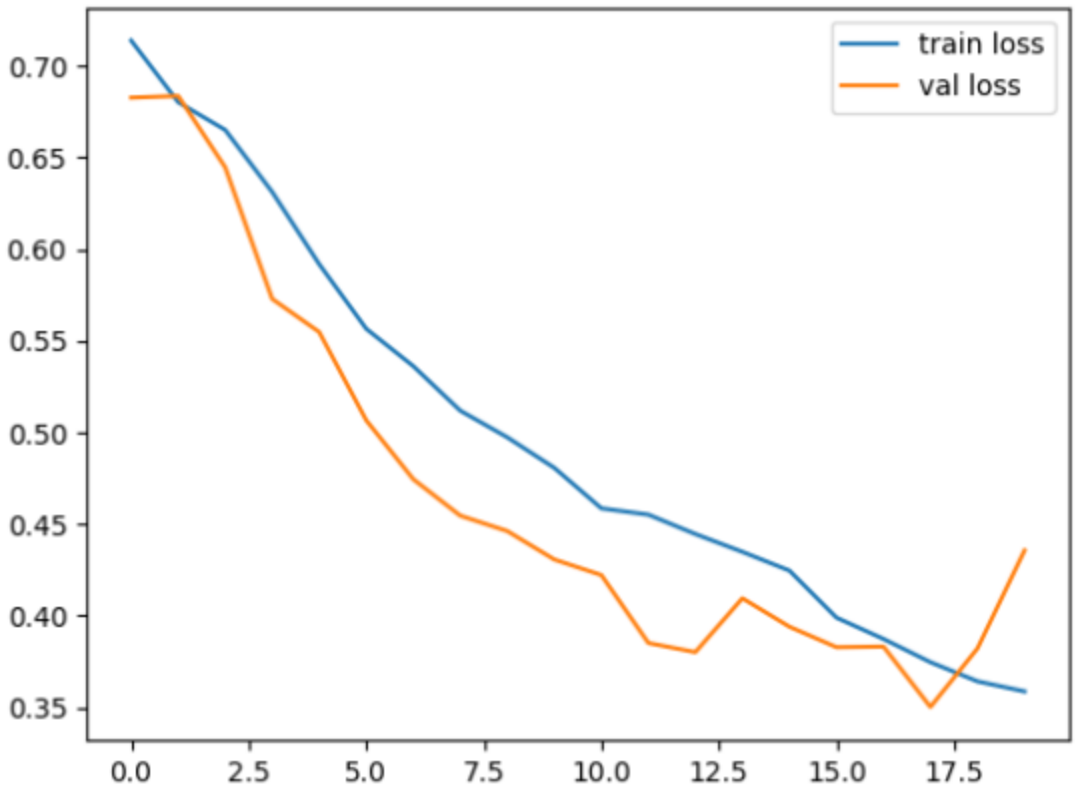
```
To test an image you can either add it to the folder test, or chose an existing one from the same folder.

To evaluate an image, write its name. To stop write 'end'.

13
1/1 [==============================] - 0s 76ms/step
0.36291152
Yor image is a cat
```

For this test the image imputed was this one.

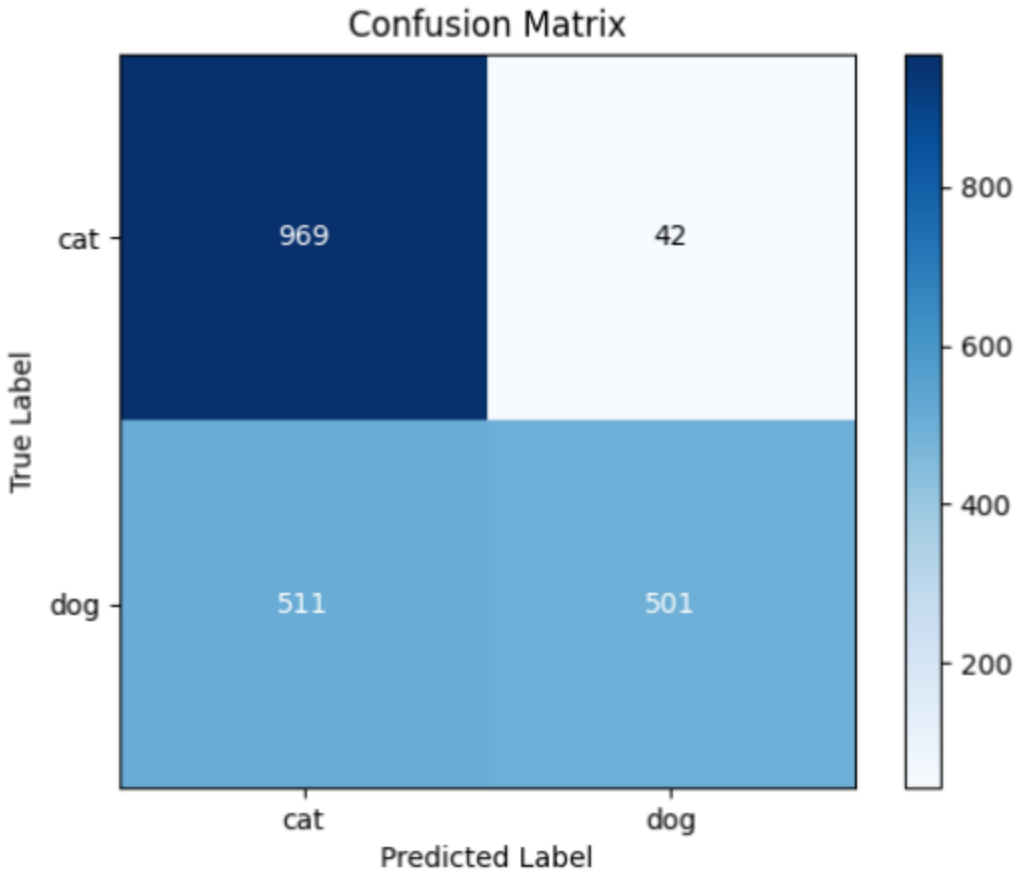
The Results

To begin, one of the first models I tried that had decent results had 3 convolutions: one with 32 layers, one with 64 layers and one with 128 layers. It also had no Dropout and no Batch normalization. The image size was 128 x 128, it had an additional Dense layer with 512 neurons and activation relu. The optimizer used was ‘adam’. The number of epochs was 20.

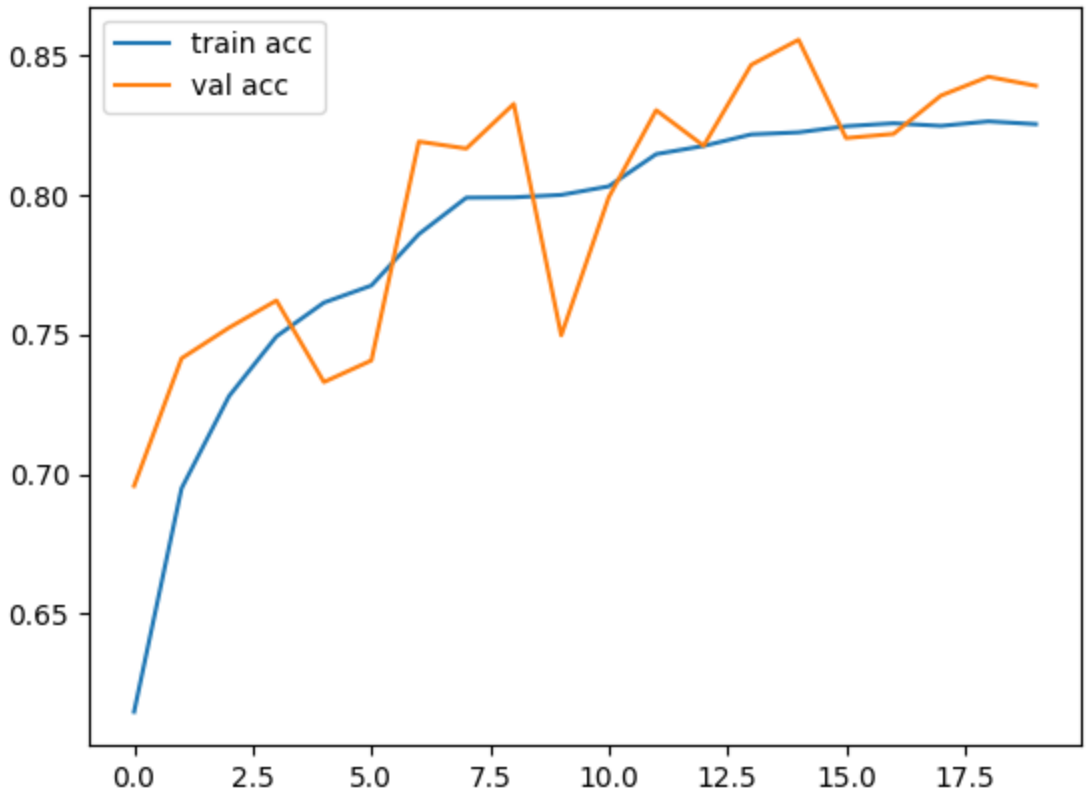

train acc
val acc
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

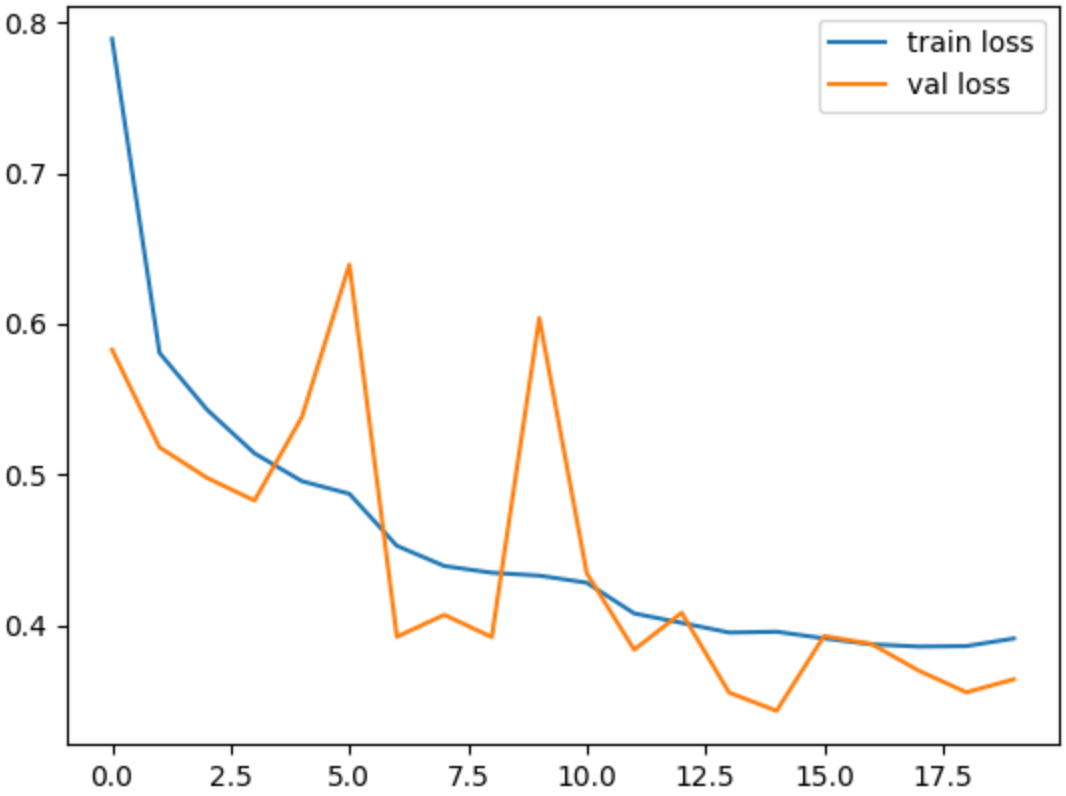



```
_________________________________________________________________
Found 20000 images belonging to 2 classes.
Found 5000 images belonging to 2 classes.
Found 2023 images belonging to 2 classes.
64/64 [==============================] - 4s 56ms/step
Test loss: 0.40608617663383484
Test accuracy: 0.8304498195648193
```

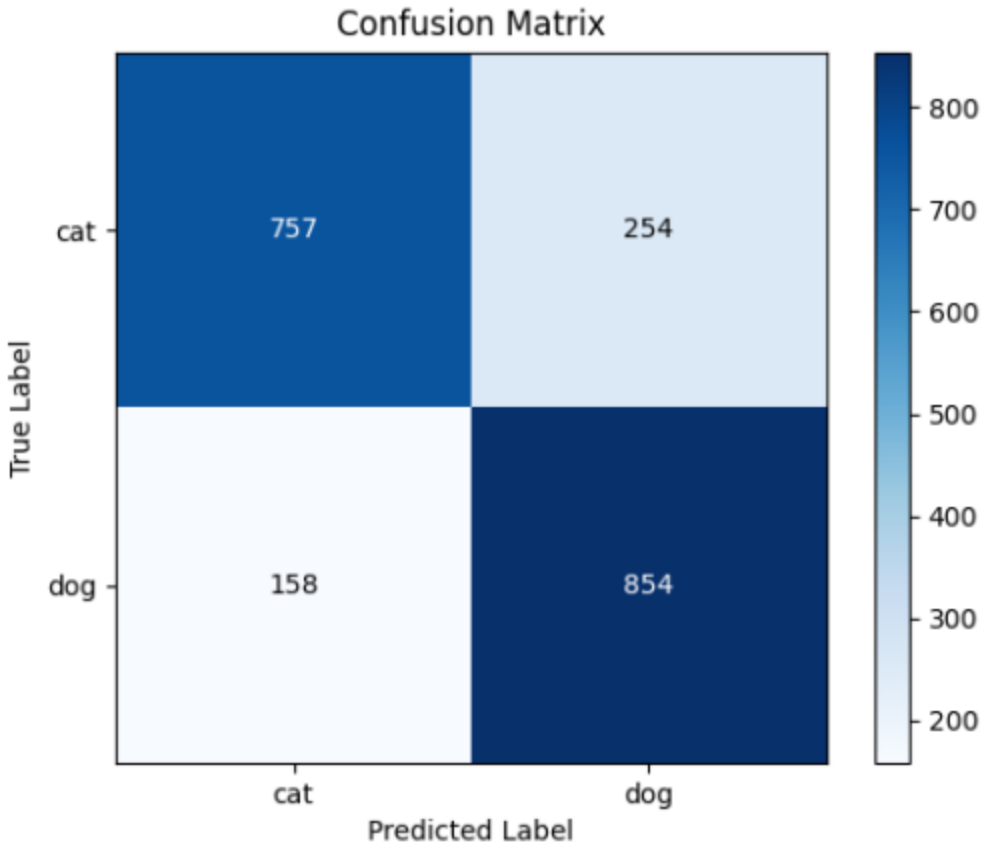


Another model I tried, I changed the optimizer to rmsprop, which I kept for the rest of the models, kept only two convolution layers, one with 32 and one with 64 and added Dropout at the end before the last Dense. The image size here was 64 x64. The number of epochs was 20.

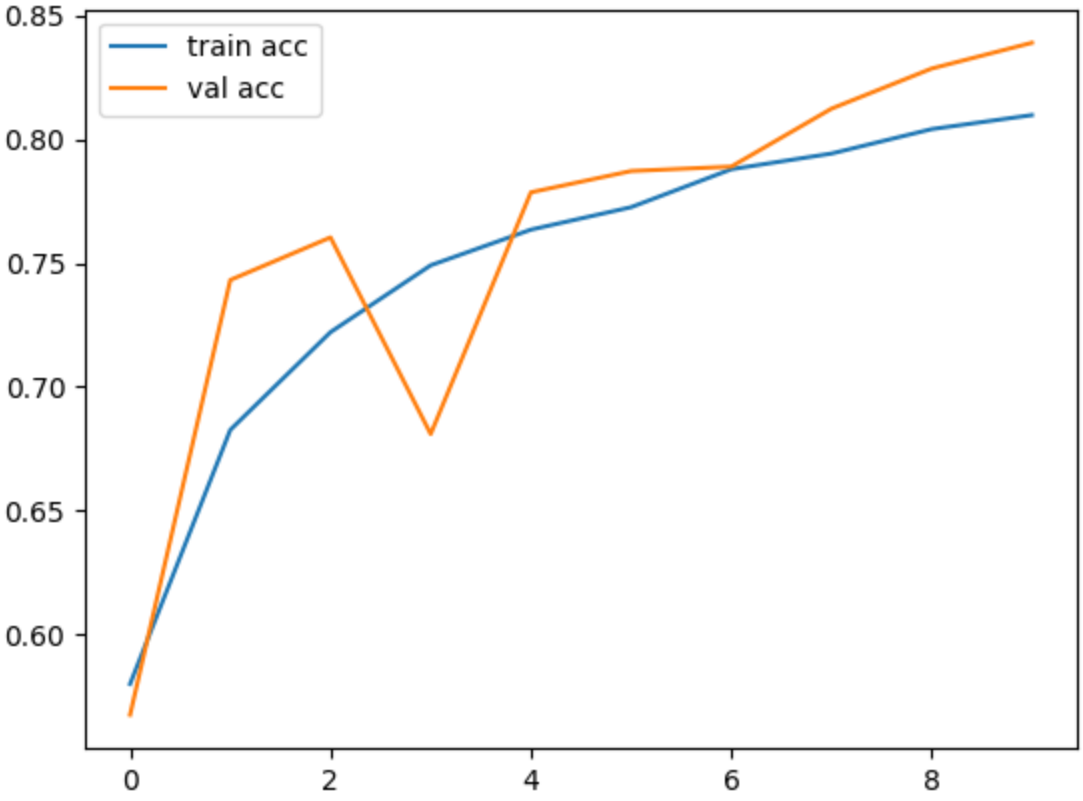

train acc
val acc
0.85
0.80
0.75
0.70
0.65
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

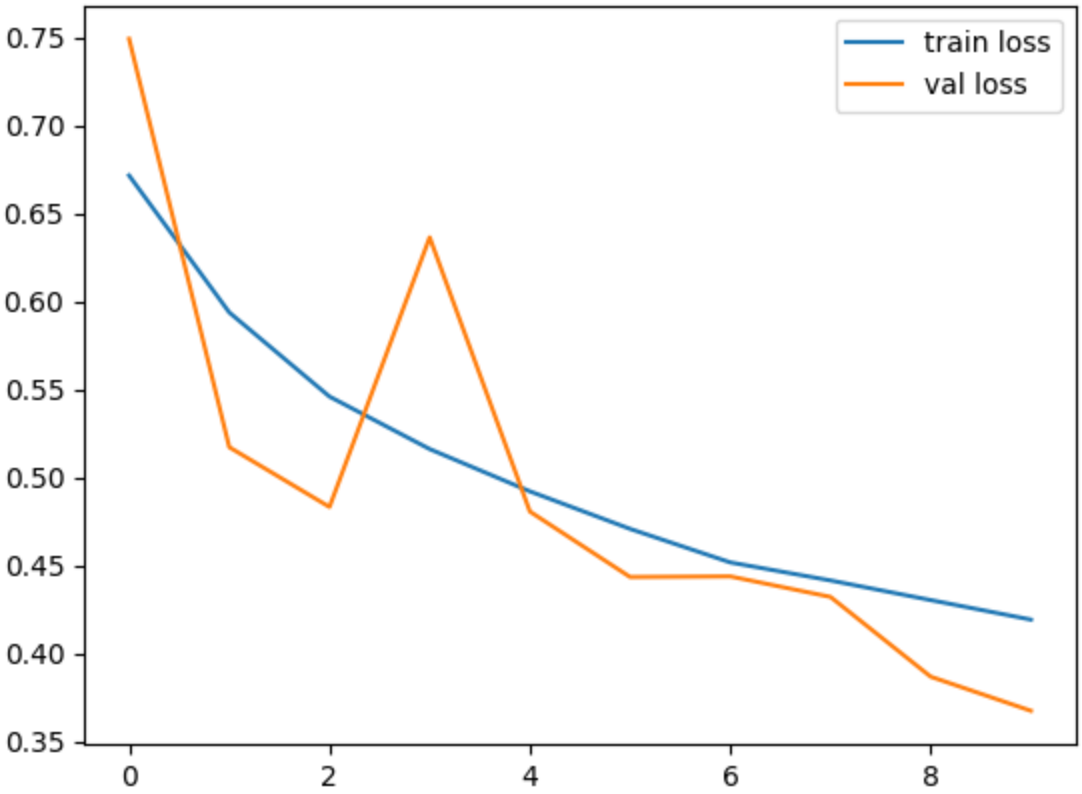



```
----------------------------------------------------------------
Found 20000 images belonging to 2 classes.
Found 5000 images belonging to 2 classes.
Found 2023 images belonging to 2 classes.
64/64 [==============================] - 1s 18ms/step
Test loss: 0.33991244435310364
Test accuracy: 0.8512110710144043
```

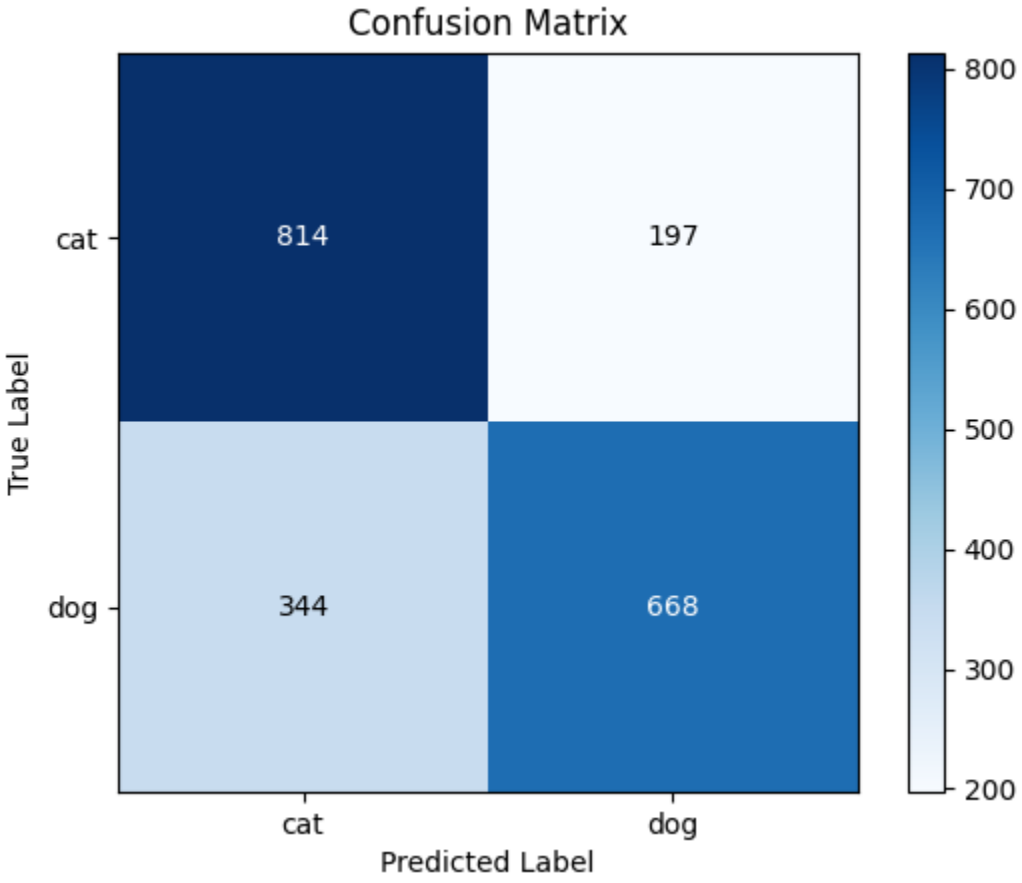


Another model had no Dropout, no Batch normalization, had a second Dense layer with 256 neurons other than the last one. Image size was 64 x 64. Number of epochs was 10.

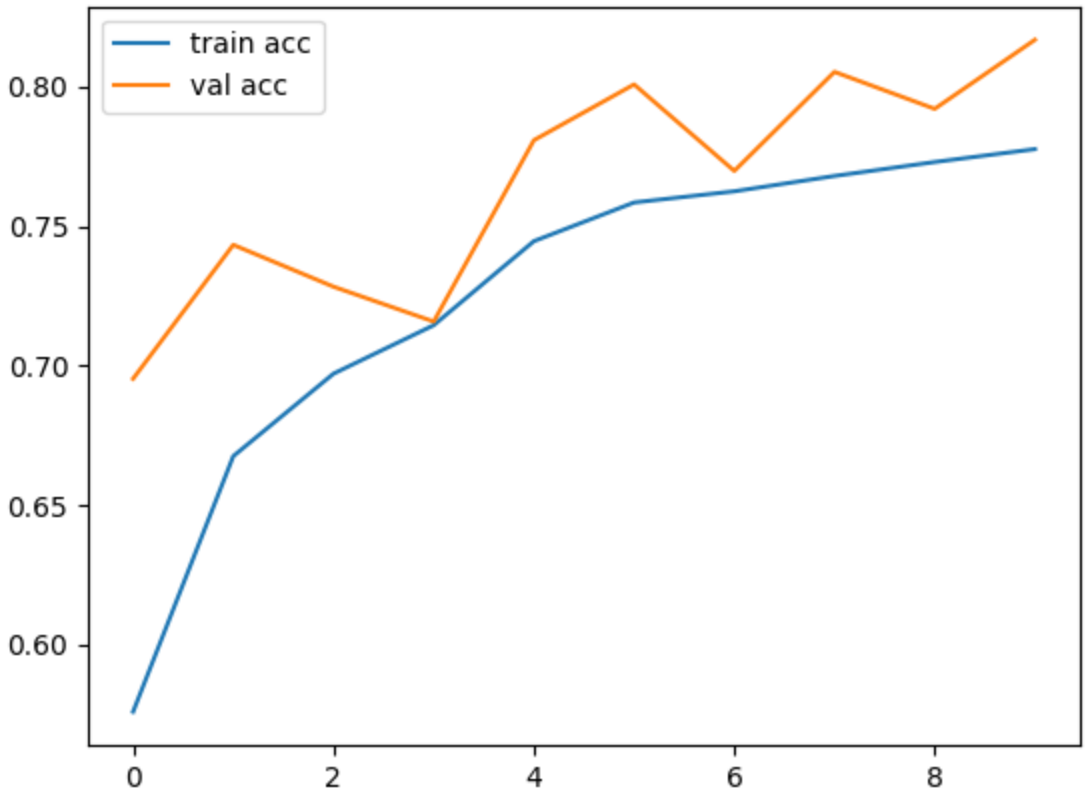

train acc
val acc
0.85
0.80
0.75
0.70
0.65
0.60
0
2
4
6
8

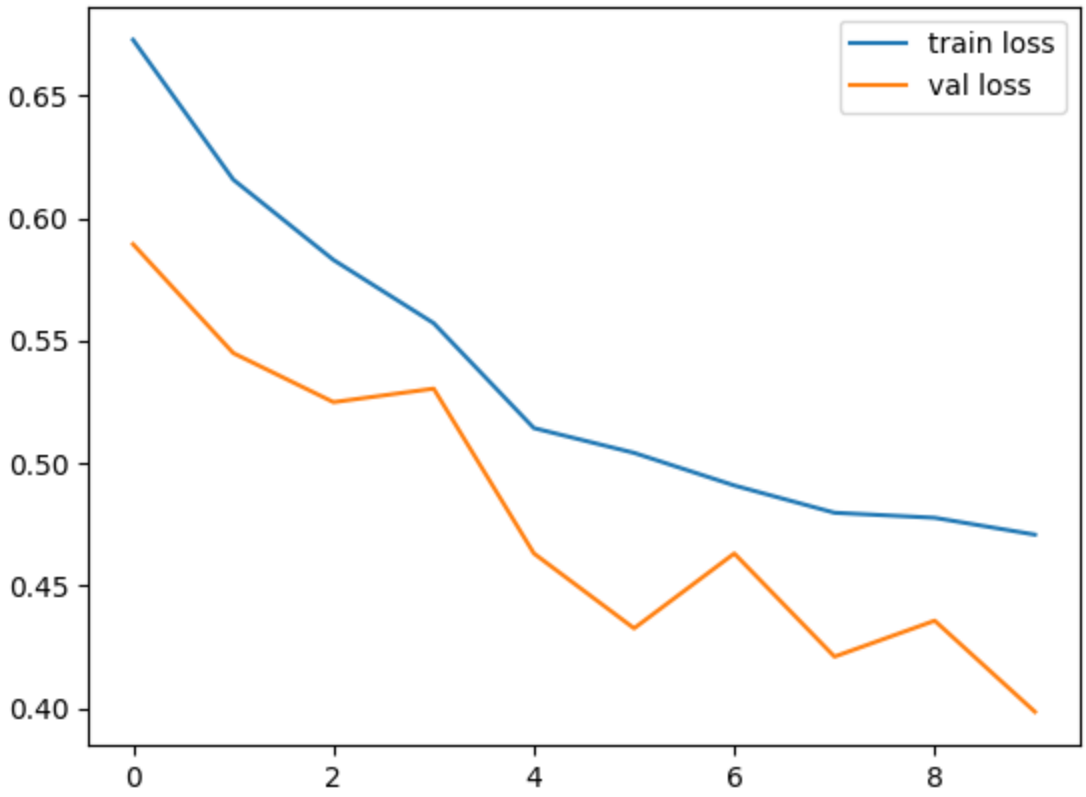

train loss
val loss
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0
2
4
6
8

```
----------------------------------------------------------------
Found 20000 images belonging to 2 classes.
Found 5000 images belonging to 2 classes.
Found 2023 images belonging to 2 classes.
64/64 [==============================] - 1s 12ms/step
Test loss: 0.3415605425834656
Test accuracy: 0.8452792763710022
```

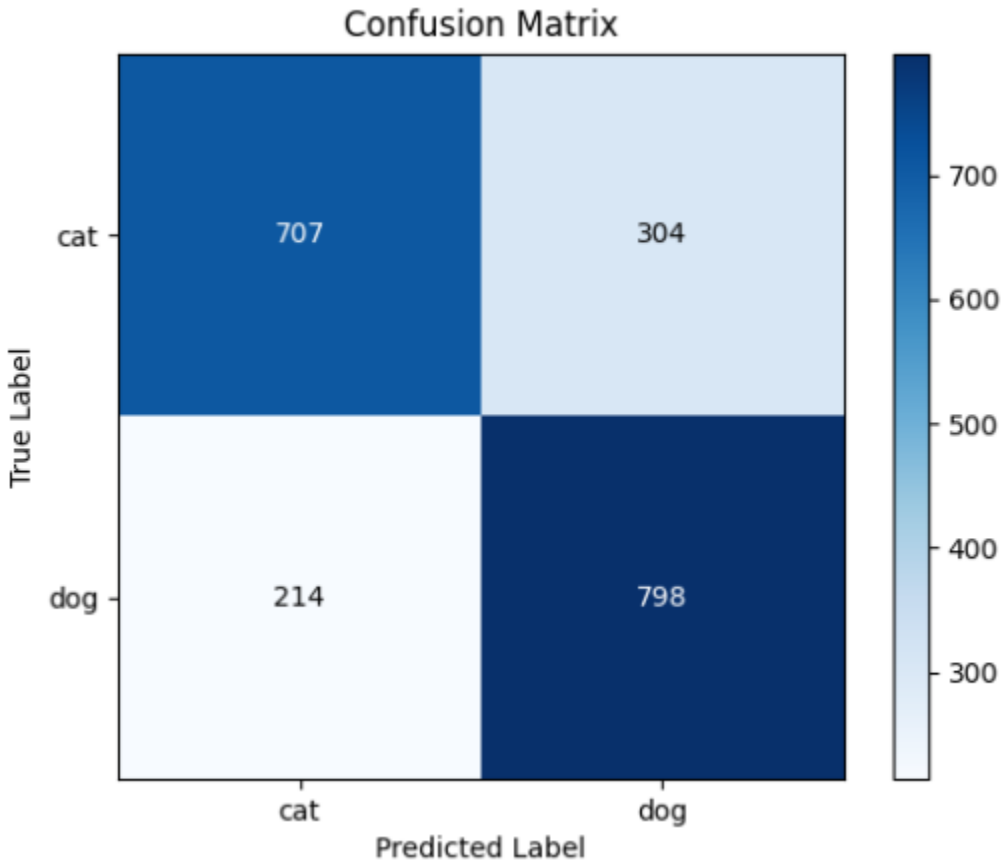


This model had a Dropout of 0.25 after each max pooling, had no batch normalization and had only the last dense layer. The image size was 64 x 64 and the number of epochs was 10.

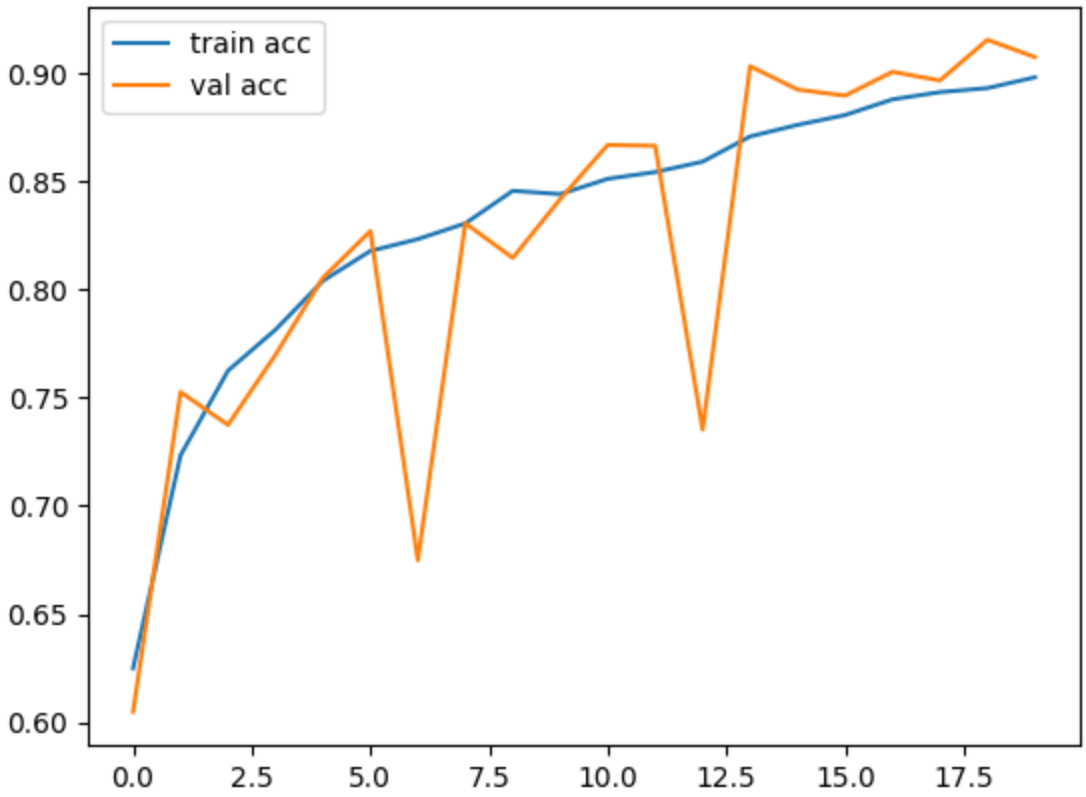

train acc
val acc
0.80
0.75
0.70
0.65
0.60
0
2
4
6
8

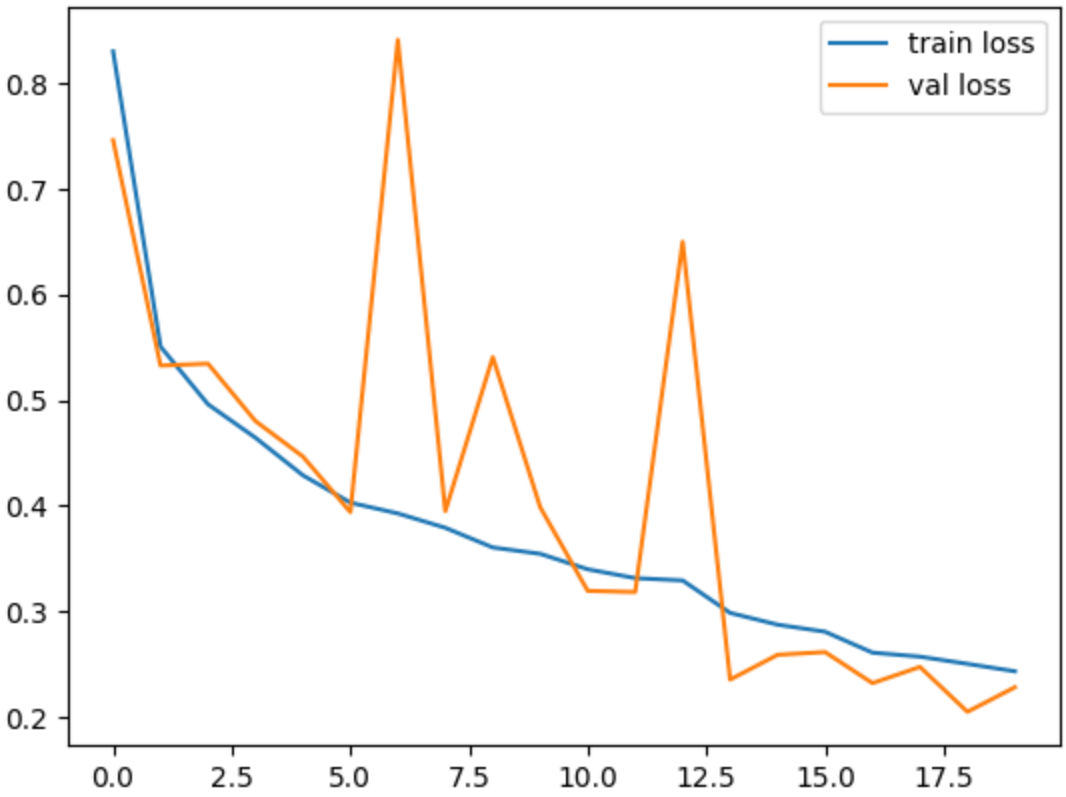

train loss
val loss
0.65
0.60
0.55
0.50
0.45
0.40
0
2
4
6
8

```
----------------------------------------------------------------
Found 20000 images belonging to 2 classes.
Found 5000 images belonging to 2 classes.
Found 2023 images belonging to 2 classes.
64/64 [==============================] - 1s 12ms/step
Test loss: 0.37374967336654663
Test accuracy: 0.8358873128890991
```

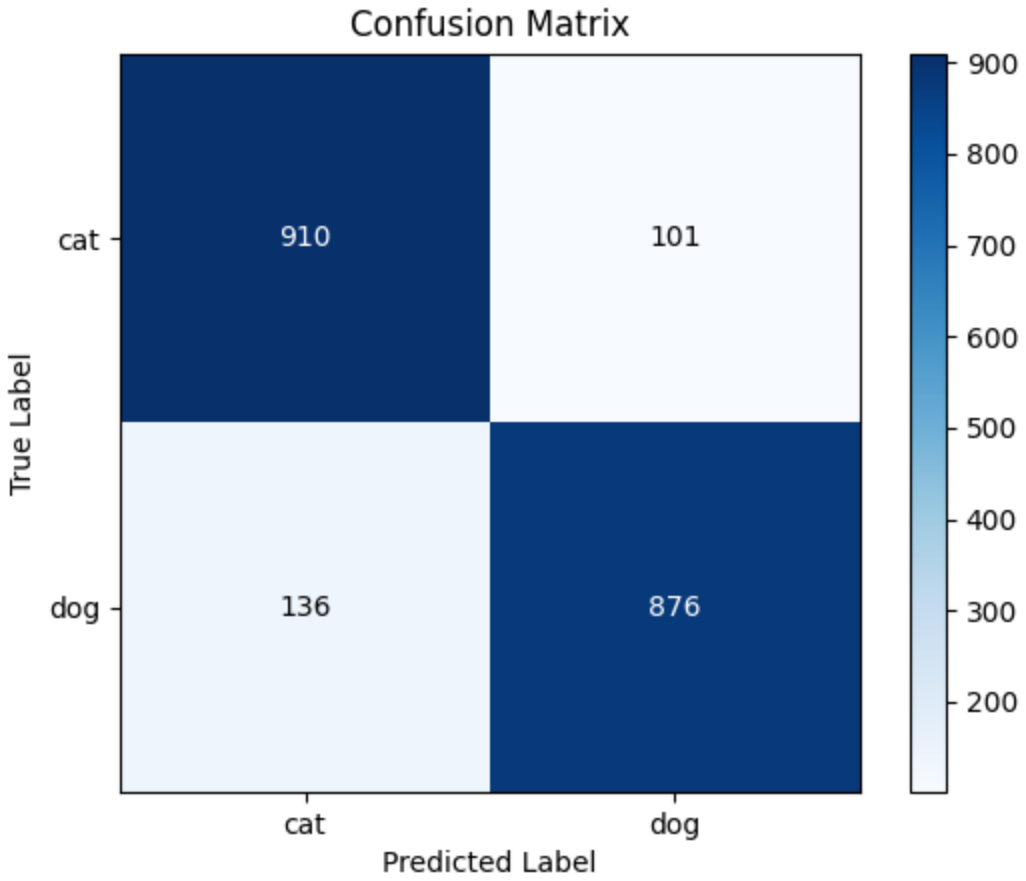


One of the better models had a dropout of 0.25 after every max pooling, a batch normalization after every convolution, an additional dense layer of 512 neurons, the image size was 128 x 128, and the number of epochs was 20.

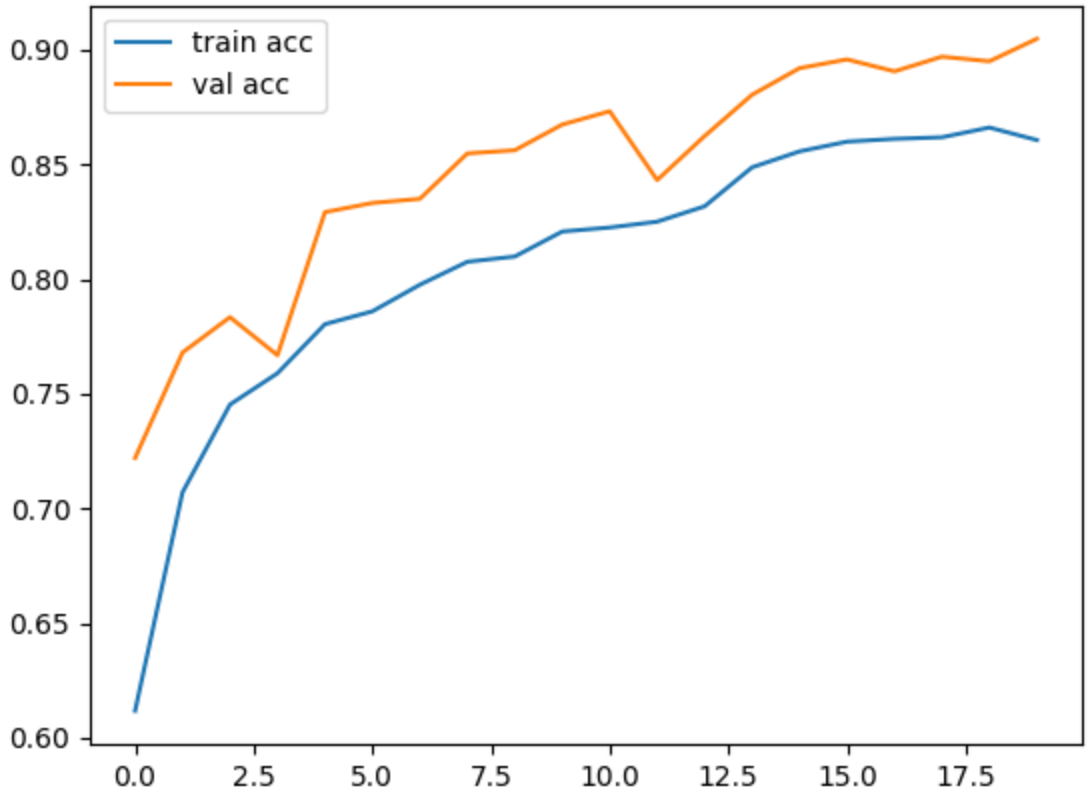

train acc
val acc
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

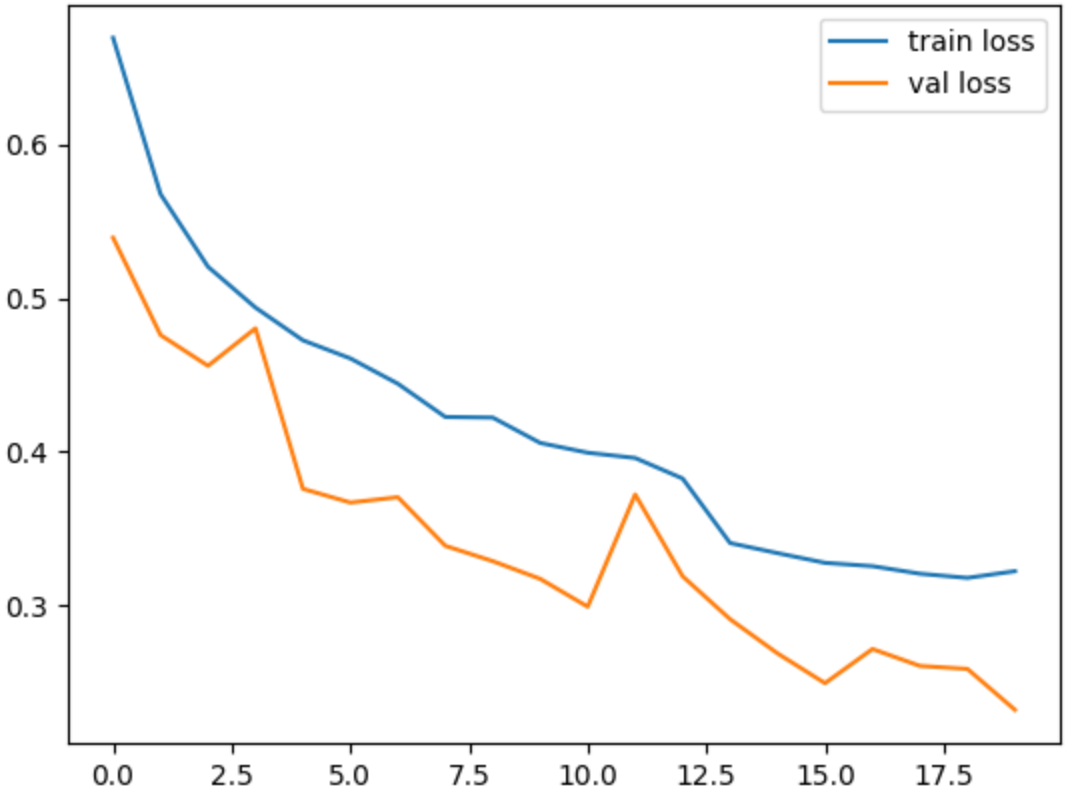



```
_________________________________________________________________
Found 20000 images belonging to 2 classes.
Found 5000 images belonging to 2 classes.
Found 2023 images belonging to 2 classes.
64/64 [==============================] - 6s 93ms/step
Test loss: 0.175918310880661
Test accuracy: 0.928324282169342
```

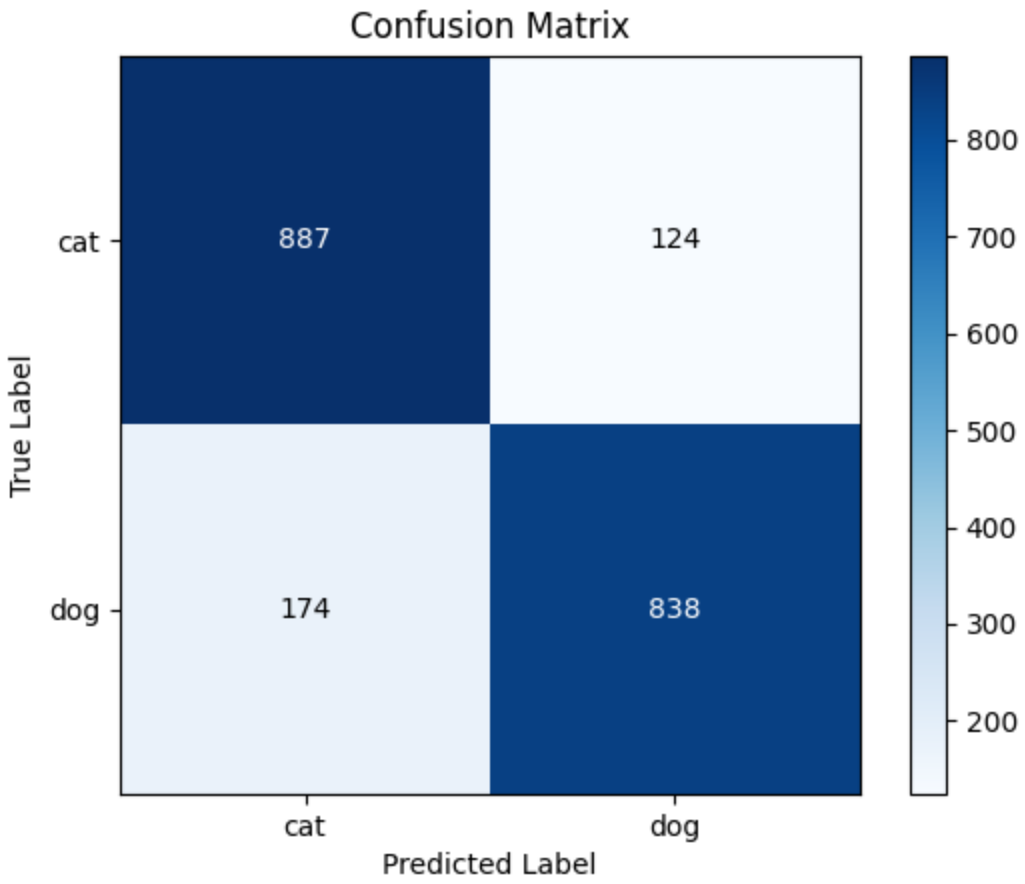


The model I ended up using had a dropout of 0.25 after every max pooling, no batch normalization, an additional dense layer of 512 neurons, the image size was 128 x 128, and the number of epochs was 20.


train acc
val acc
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5




```
----------------------------------------------------------------
Found 20000 images belonging to 2 classes.
Found 5000 images belonging to 2 classes.
Found 2023 images belonging to 2 classes.
64/64 [==============================] - 4s 67ms/step
Test loss: 0.20230898261070251
Test accuracy: 0.928324282169342
```


Confusion Matrix
887
124
174
838
cat
dog
True Label
cat
dog
Predicted Label
800
700
600
500
400
300
200