This notebook is by F. Chollet and is included in his book.

```
# import os
# os.environ['LD_LIBRARY_PATH'] = 
'/workspaces/artificial_intelligence/.venv/lib/python3.11/site-
packages/tensorrt_libs'

import tensorflow as tf
import keras
gpus = tf.config.list_physical_devices('GPU')
for gpu in gpus:
    print("Name:", gpu.name, "  Type:", gpu.device_type)
```

Using convnets with small datasets

This notebook contains the code sample found in Chapter 5, Section 2 of Deep Learning with Python. Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

Training a convnet from scratch on a small dataset

Having to train an image classification model using only very little data is a common situation, which you likely encounter yourself in practice if you ever do computer vision in a professional context.

Having "few" samples can mean anywhere from a few hundreds to a few tens of thousands of images. As a practical example, we will focus on classifying images as "dogs" or "cats", in a dataset containing 4000 pictures of cats and dogs (2000 cats, 2000 dogs). We will use 2000 pictures for training, 1000 for validation, and finally 1000 for testing.

In this section, we will review one basic strategy to tackle this problem: training a new model from scratch on what little data we have. We will start by naively training a small convnet on our 2000 training samples, without any regularization, to set a baseline for what can be achieved. This will get us to a classification accuracy of 71%. At that point, our main issue will be overfitting. Then we will introduce *data augmentation*, a powerful technique for mitigating overfitting in computer vision. By leveraging data augmentation, we will improve our network to reach an accuracy of 82%.

In the next section, we will review two more essential techniques for applying deep learning to small datasets: *doing feature extraction with a pre-trained network* (this will get us to an accuracy of 90% to 93%), and *fine-tuning a pre-trained network* (this will get us to our final accuracy of 95%). Together, these three strategies -- training a small model from scratch, doing feature extracting using a pre-trained model, and fine-tuning a pre-trained model -- will constitute your future toolbox for tackling the problem of doing computer vision with small datasets.

The relevance of deep learning for small-data problems

You will sometimes hear that deep learning only works when lots of data is available. This is in part a valid point: one fundamental characteristic of deep learning is that it is able to find interesting features in the training data on its own, without any need for manual feature engineering, and this can only be achieved when lots of training examples are available. This is especially true for problems where the input samples are very high-dimensional, like images.

However, what constitutes "lots" of samples is relative -- relative to the size and depth of the network you are trying to train, for starters. It isn't possible to train a convnet to solve a complex problem with just a few tens of samples, but a few hundreds can potentially suffice if the model is small and well-regularized and if the task is simple. Because convnets learn local, translation-invariant features, they are very data-efficient on perceptual problems. Training a convnet from scratch on a very small image dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering. You will see this in action in this section.

But what's more, deep learning models are by nature highly repurposable: you can take, say, an image classification or speech-to-text model trained on a large-scale dataset then reuse it on a significantly different problem with only minor changes. Specifically, in the case of computer vision, many pre-trained models (usually trained on the ImageNet dataset) are now publicly available for download and can be used to bootstrap powerful vision models out of very little data. That's what we will do in the next section.

For now, let's get started by getting our hands on the data.

Downloading the data

The cats vs. dogs dataset that we will use isn't packaged with Keras. It was made available by Kaggle.com as part of a computer vision competition in late 2013, back when convnets weren't quite mainstream. You can download the original dataset at: `https://www.kaggle.com/c/dogs-vs-cats/data` (you will need to create a Kaggle account if you don't already have one -- don't worry, the process is painless).

The pictures are medium-resolution color JPEGs. They look like this:

Unsurprisingly, the cats vs. dogs Kaggle competition in 2013 was won by entrants who used convnets. The best entries could achieve up to 95% accuracy. In our own example, we will get fairly close to this accuracy (in the next section), even though we will be training our models on less than 10% of the data that was available to the competitors. This original dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543MB large (compressed). After downloading and uncompressing it, we will create a new dataset containing three subsets: a training set with 1000 samples of each class, a validation set with 500 samples of each class, and finally a test set with 500 samples of each class.

Here are a few lines of code to do this:

```
import os, shutil

# Unzip file
!mkdir -p dogscats/subset
!unzip -o -q dogs-vs-cats-subset.zip -d dogscats

base_dir = 'dogscats/subset'
train_dir = os.path.join(base_dir, 'train')
train_cats_dir = os.path.join(base_dir, 'train', 'cats')
train_dogs_dir = os.path.join(base_dir, 'train', 'dogs')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')
```

So we have indeed 2000 training images, and then 1000 validation images and 1000 test images. In each split, there is the same number of samples from each class: this is a balanced binary classification problem, which means that classification accuracy will be an appropriate measure of success.

Building our network

We've already built a small convnet for MNIST in the previous example, so you should be familiar with them. We will reuse the same general structure: our convnet will be a stack of alternated `Conv2D` (with `relu` activation) and `MaxPooling2D` layers.

However, since we are dealing with bigger images and a more complex problem, we will make our network accordingly larger: it will have one more `Conv2D` + `MaxPooling2D` stage. This serves both to augment the capacity of the network, and to further reduce the size of the feature maps, so that they aren't overly large when we reach the `Flatten` layer. Here, since we start from inputs of size 150x150 (a somewhat arbitrary choice), we end up with feature maps of size 7x7 right before the `Flatten` layer.

Note that the depth of the feature maps is progressively increasing in the network (from 32 to 128), while the size of the feature maps is decreasing (from 148x148 to 7x7). This is a pattern that you will see in almost all convnets.

Since we are attacking a binary classification problem, we are ending the network with a single unit (a `Dense` layer of size 1) and a `sigmoid` activation. This unit will encode the probability that the network is looking at one class or the other.

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Let's take a look at how the dimensions of the feature maps change with every successive layer:

```
model.summary()

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 148, 148, 32)      896

 max_pooling2d (MaxPooling2  (None, 74, 74, 32)        0
 D)
```

```
 conv2d_1 (Conv2D)           (None, 72, 72, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 36, 36, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 34, 34, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 17, 17, 128)       0
 g2D)

 conv2d_3 (Conv2D)           (None, 15, 15, 128)       147584

 max_pooling2d_3 (MaxPoolin  (None, 7, 7, 128)         0
 g2D)

 flatten (Flatten)           (None, 6272)              0

 dense (Dense)               (None, 512)               3211776

 dense_1 (Dense)             (None, 1)                 513

=================================================================
Total params: 3453121 (13.17 MB)
Trainable params: 3453121 (13.17 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

For our compilation step, we'll go with the `RMSprop` optimizer as usual. Since we ended our network with a single sigmoid unit, we will use binary crossentropy as our loss (as a reminder, check out the table in Chapter 4, section 5 for a cheatsheet on what loss function to use in various situations).

```
from keras import optimizers

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])

WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.
```

Data preprocessing

As you already know by now, data should be formatted into appropriately pre-processed floating point tensors before being fed into our network. Currently, our data sits on a drive as JPEG files, so the steps for getting it into our network are roughly:

- Read the picture files.
- Decode the JPEG content to RBG grids of pixels.
- Convert these into floating point tensors.
- Rescale the pixel values (between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values).

It may seem a bit daunting, but thankfully Keras has utilities to take care of these steps automatically. Keras has a module with image processing helper tools, located at `keras.preprocessing.image`. In particular, it contains the class `ImageDataGenerator` which allows to quickly set up Python generators that can automatically turn image files on disk into batches of pre-processed tensors. This is what we will use here.

```
from keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

Let's take a look at the output of one of these generators: it yields batches of 150x150 RGB images (shape `(20, 150, 150, 3)`) and binary labels (shape `(20,)`). 20 is the number of samples in each batch (the batch size). Note that the generator yields these batches indefinitely: it just loops endlessly over the images present in the target folder. For this reason, we need to `break` the iteration loop at some point.

```
for data_batch, labels_batch in train_generator:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break

data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

Let's fit our model to the data using the generator. We do it using the `fit_generator` method, the equivalent of `fit` for data generators like ours. It expects as first argument a Python generator that will yield batches of inputs and targets indefinitely, like ours does. Because the data is being generated endlessly, the generator needs to know example how many samples to draw from the generator before declaring an epoch over. This is the role of the `steps_per_epoch` argument: after having drawn `steps_per_epoch` batches from the generator, i.e. after having run for `steps_per_epoch` gradient descent steps, the fitting process will go to the next epoch. In our case, batches are 20-sample large, so it will take 100 batches until we see our target of 2000 samples.

When using `fit_generator`, one may pass a `validation_data` argument, much like with the `fit` method. Importantly, this argument is allowed to be a data generator itself, but it could be a tuple of Numpy arrays as well. If you pass a generator as `validation_data`, then this generator is expected to yield batches of validation data endlessly, and thus you should also specify the `validation_steps` argument, which tells the process how many batches to draw from the validation generator for evaluation.

```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=50)
```

```
Epoch 1/30
```

```
<ipython-input-16-a7acfc8093a4>:1: UserWarning: `Model.fit_generator`
is deprecated and will be removed in a future version. Please use
`Model.fit`, which supports generators.
  history = model.fit_generator(
```

```
100/100 [==============================] - 18s 62ms/step - loss:
0.7091 - acc: 0.5290 - val_loss: 0.6919 - val_acc: 0.5710
Epoch 2/30
100/100 [==============================] - 5s 50ms/step - loss: 0.6950
- acc: 0.5350 - val_loss: 0.6740 - val_acc: 0.5860
Epoch 3/30
100/100 [==============================] - 6s 58ms/step - loss: 0.6723
- acc: 0.5985 - val_loss: 0.6669 - val_acc: 0.5780
Epoch 4/30
100/100 [==============================] - 5s 51ms/step - loss: 0.6470
- acc: 0.6175 - val_loss: 0.6373 - val_acc: 0.6250
Epoch 5/30
100/100 [==============================] - 6s 61ms/step - loss: 0.6056
- acc: 0.6675 - val_loss: 0.5872 - val_acc: 0.6960
Epoch 6/30
100/100 [==============================] - 5s 52ms/step - loss: 0.5828
- acc: 0.6975 - val_loss: 0.5726 - val_acc: 0.6740
Epoch 7/30
100/100 [==============================] - 5s 50ms/step - loss: 0.5401
```

```
- acc: 0.7250 - val_loss: 0.6396 - val_acc: 0.6850
Epoch 8/30
100/100 [==============================] - 6s 60ms/step - loss: 0.4978
- acc: 0.7550 - val_loss: 0.5530 - val_acc: 0.7210
Epoch 9/30
100/100 [==============================] - 5s 50ms/step - loss: 0.4639
- acc: 0.7780 - val_loss: 1.0896 - val_acc: 0.5850
Epoch 10/30
100/100 [==============================] - 6s 61ms/step - loss: 0.4093
- acc: 0.8085 - val_loss: 0.4899 - val_acc: 0.7670
Epoch 11/30
100/100 [==============================] - 5s 51ms/step - loss: 0.3463
- acc: 0.8415 - val_loss: 0.4705 - val_acc: 0.8040
Epoch 12/30
100/100 [==============================] - 6s 58ms/step - loss: 0.2752
- acc: 0.8800 - val_loss: 0.5017 - val_acc: 0.7930
Epoch 13/30
100/100 [==============================] - 5s 49ms/step - loss: 0.2186
- acc: 0.9130 - val_loss: 0.5790 - val_acc: 0.8040
Epoch 14/30
100/100 [==============================] - 5s 53ms/step - loss: 0.1629
- acc: 0.9390 - val_loss: 0.6387 - val_acc: 0.8250
Epoch 15/30
100/100 [==============================] - 5s 49ms/step - loss: 0.1491
- acc: 0.9475 - val_loss: 0.5681 - val_acc: 0.8280
Epoch 16/30
100/100 [==============================] - 6s 61ms/step - loss: 0.0866
- acc: 0.9720 - val_loss: 0.9153 - val_acc: 0.8410
Epoch 17/30
100/100 [==============================] - 5s 50ms/step - loss: 0.1008
- acc: 0.9710 - val_loss: 0.8755 - val_acc: 0.8470
Epoch 18/30
100/100 [==============================] - 6s 59ms/step - loss: 0.0674
- acc: 0.9790 - val_loss: 1.0278 - val_acc: 0.8370
Epoch 19/30
100/100 [==============================] - 5s 51ms/step - loss: 0.0682
- acc: 0.9755 - val_loss: 1.0038 - val_acc: 0.8520
Epoch 20/30
100/100 [==============================] - 5s 52ms/step - loss: 0.0388
- acc: 0.9885 - val_loss: 1.0790 - val_acc: 0.8470
Epoch 21/30
100/100 [==============================] - 5s 50ms/step - loss: 0.0722
- acc: 0.9770 - val_loss: 1.4979 - val_acc: 0.8230
Epoch 22/30
100/100 [==============================] - 6s 60ms/step - loss: 0.0667
- acc: 0.9860 - val_loss: 1.2899 - val_acc: 0.8360
Epoch 23/30
100/100 [==============================] - 5s 49ms/step - loss: 0.0490
- acc: 0.9855 - val_loss: 1.0768 - val_acc: 0.8410
```

```
Epoch 24/30
100/100 [==============================] - 6s 58ms/step - loss: 0.0780
- acc: 0.9815 - val_loss: 1.2480 - val_acc: 0.8500
Epoch 25/30
100/100 [==============================] - 5s 50ms/step - loss: 0.0287
- acc: 0.9930 - val_loss: 1.2922 - val_acc: 0.8310
Epoch 26/30
100/100 [==============================] - 6s 62ms/step - loss: 0.0411
- acc: 0.9880 - val_loss: 1.4473 - val_acc: 0.8360
Epoch 27/30
100/100 [==============================] - 5s 50ms/step - loss: 0.0334
- acc: 0.9895 - val_loss: 1.4845 - val_acc: 0.8600
Epoch 28/30
100/100 [==============================] - 6s 61ms/step - loss: 0.0550
- acc: 0.9895 - val_loss: 1.2972 - val_acc: 0.8510
Epoch 29/30
100/100 [==============================] - 5s 50ms/step - loss: 0.0348
- acc: 0.9925 - val_loss: 1.3508 - val_acc: 0.8520
Epoch 30/30
100/100 [==============================] - 6s 59ms/step - loss: 0.0240
- acc: 0.9945 - val_loss: 1.9262 - val_acc: 0.8280
```

It is good practice to always save your models after training:

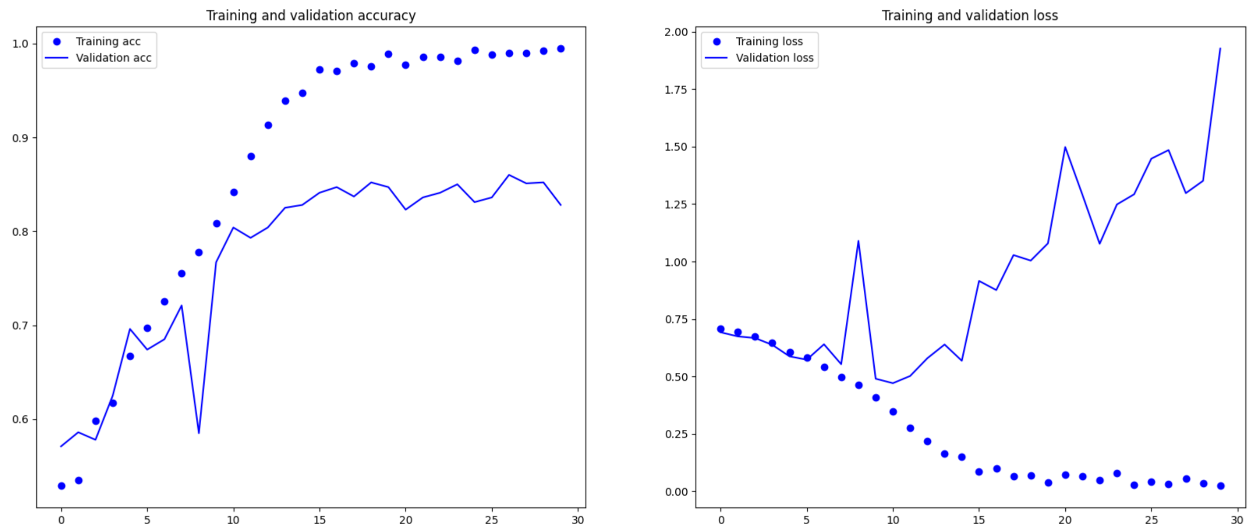
```
model.save('cats_and_dogs_small_1.h5')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/
training.py:3000: UserWarning: You are saving your model as an HDF5
file via `model.save()`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')`.
  saving_api.save_model(
```

Let's plot the loss and accuracy of the model over the training and validation data during training:

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.figure(figsize=(20,8))

plt.subplot(1,2,1)
plt.plot(epochs, acc, 'bo', label='Training acc')
```

```
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

# plt.figure()
plt.subplot(1,2,2)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```



These plots are characteristic of overfitting. Our training accuracy increases linearly over time, until it reaches nearly 100%, while our validation accuracy stalls at 70-72%. Our validation loss reaches its minimum after only five epochs then stalls, while the training loss keeps decreasing linearly until it reaches nearly 0.

Because we only have relatively few training samples (2000), overfitting is going to be our number one concern. You already know about a number of techniques that can help mitigate overfitting, such as dropout and weight decay (L2 regularization). We are now going to introduce a new one, specific to computer vision, and used almost universally when processing images with deep learning models: *data augmentation*.

Plot the ROC curves

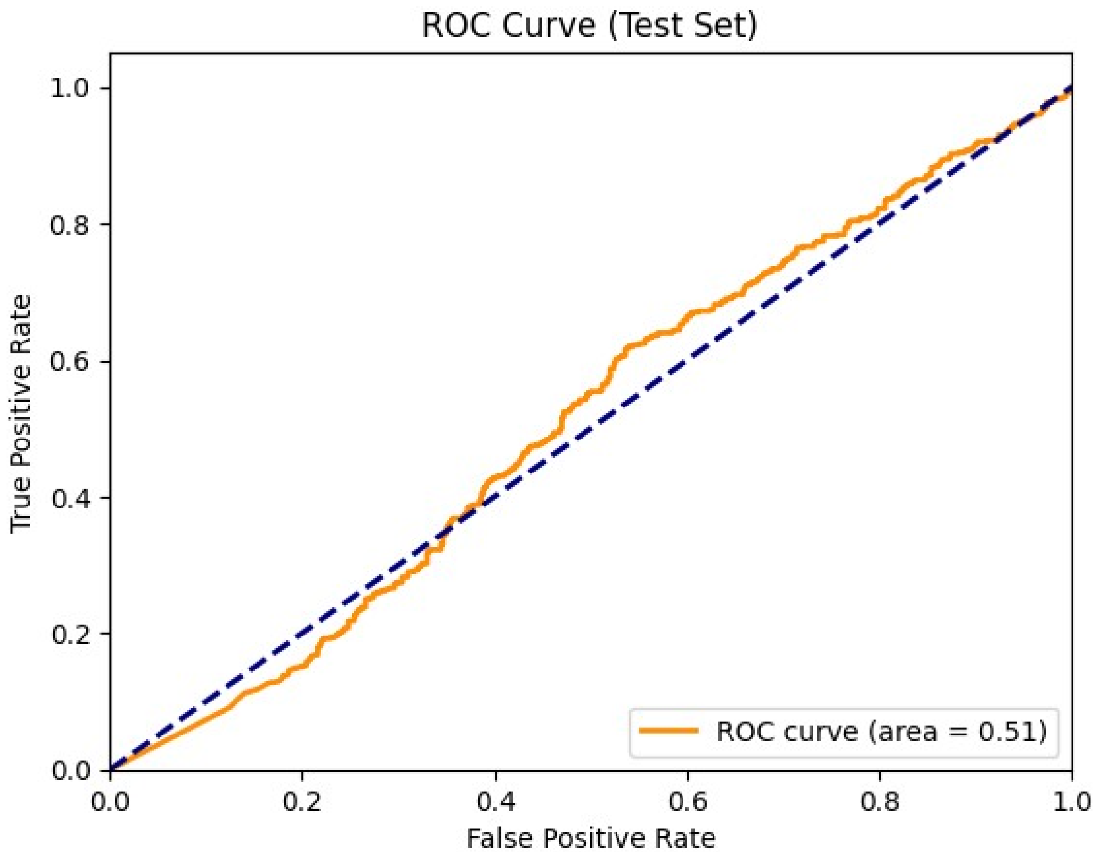
```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')
```

```
Found 1000 images belonging to 2 classes.
```

```
def find_labels_and_probability(img_gen):
  y_true = img_gen.classes

  # Get predicted probabilities (y_score)
  y_pred = model.predict(img_gen)

  return y_true, y_pred

# train_y_true, train_probs =
find_labels_and_probability(train_generator)
# val_y_true, val_probs =
find_labels_and_probability(validation_generator)
test_y_true, test_probs = find_labels_and_probability(test_generator)
```

```
32/32 [==============================] - 2s 50ms/step
```

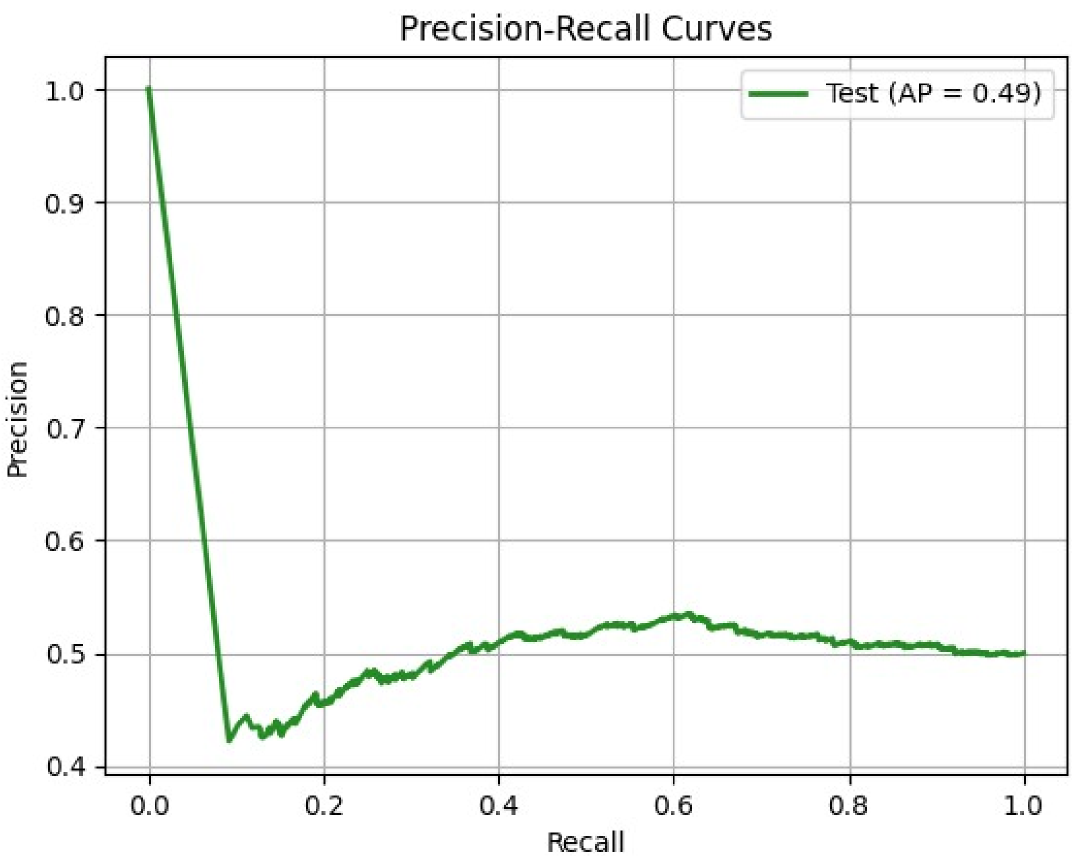
```
# Function to plot ROC curve
def plot_roc_curve(y_true, y_score, title):
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()

# Plot ROC curves for training, validation, and test sets
# plot_roc_curve(train_y_true, train_probs, title='ROC Curve (Training
Set)')
# plot_roc_curve(val_y_true, val_probs, title='ROC Curve (Validation
Set)')
plot_roc_curve(test_y_true, test_probs, title='ROC Curve (Test Set)')
```



Plot the Recall-Precision curves

```
from sklearn.metrics import precision_recall_curve,
average_precision_score

# Compute precision and recall for each dataset
# train_precision, train_recall, threshold =
precision_recall_curve(train_y_true, train_probs)
# val_precision, val_recall, threshold =
precision_recall_curve(val_y_true, val_probs)
test_precision, test_recall, threshold =
precision_recall_curve(test_y_true, test_probs)

# Compute average precision (AUC-PR) for each dataset
# train_average_precision = average_precision_score(train_y_true,
train_probs)
# val_average_precision = average_precision_score(val_y_true,
val_probs)
test_average_precision = average_precision_score(test_y_true,
test_probs)

# Plot Precision-Recall curves for all datasets
plt.figure()
# plt.plot(train_recall, train_precision, color='darkorange', lw=2,
```

```
label=f'Training (AP = {train_average_precision:.2f})')
# plt.plot(val_recall, val_precision, color='royalblue', lw=2,
label=f'Validation (AP = {val_average_precision:.2f})')
plt.plot(test_recall, test_precision, color='forestgreen', lw=2,
label=f'Test (AP = {test_average_precision:.2f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



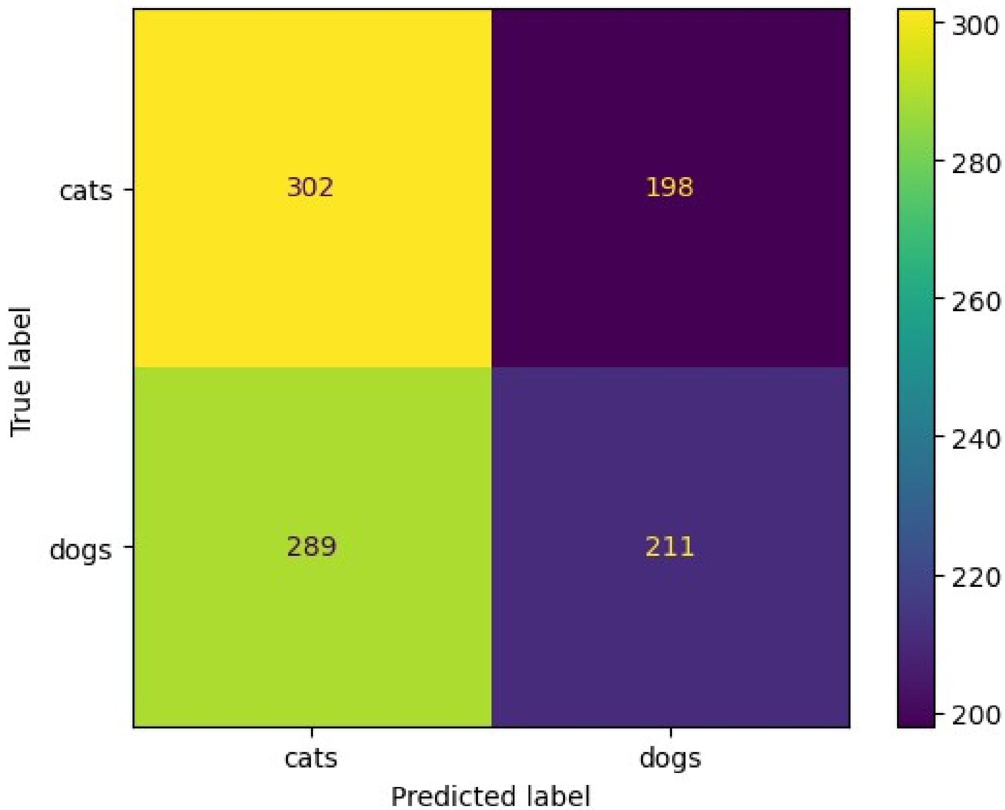
Confusion matrix for 50% threshold

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def find_labels_and_probability(y_true, y_probs):
  threshold = 0.5
  # Convert probabilities to binary classes using the 50% threshold
  y_pred = (y_probs >= threshold).astype(int)  # Predicted classes (0
or 1)

  # Create the confusion matrix
```

```
    conf_matrix = confusion_matrix(y_true, y_pred)

    return conf_matrix

# val_conf_matrix = find_labels_and_probability(val_y_true, val_probs)
test_conf_matrix = find_labels_and_probability(test_y_true,
test_probs)

# Print the confusion matrix
print("Confusion Matrix Test Data (Threshold = 50%):")
print(test_conf_matrix)

# Display the confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=test_conf_matrix,
display_labels=test_generator.class_indices)
disp.plot(cmap='viridis', values_format='d')

# print("Confusion Matrix Validation Data (Threshold = 50%):")
# print(val_conf_matrix)
```

```
Confusion Matrix Test Data (Threshold = 50%):
[[302 198]
 [289 211]]
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7917480259f0>
```



Using data augmentation

Overfitting is caused by having too few samples to learn from, rendering us unable to train a model able to generalize to new data. Given infinite data, our model would be exposed to every possible aspect of the data distribution at hand: we would never overfit. Data augmentation takes the approach of generating more training data from existing training samples, by "augmenting" the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, our model would never see the exact same picture twice. This helps the model get exposed to more aspects of the data and generalize better.

In Keras, this can be done by configuring a number of random transformations to be performed on the images read by our `ImageDataGenerator` instance. Let's get started with an example:

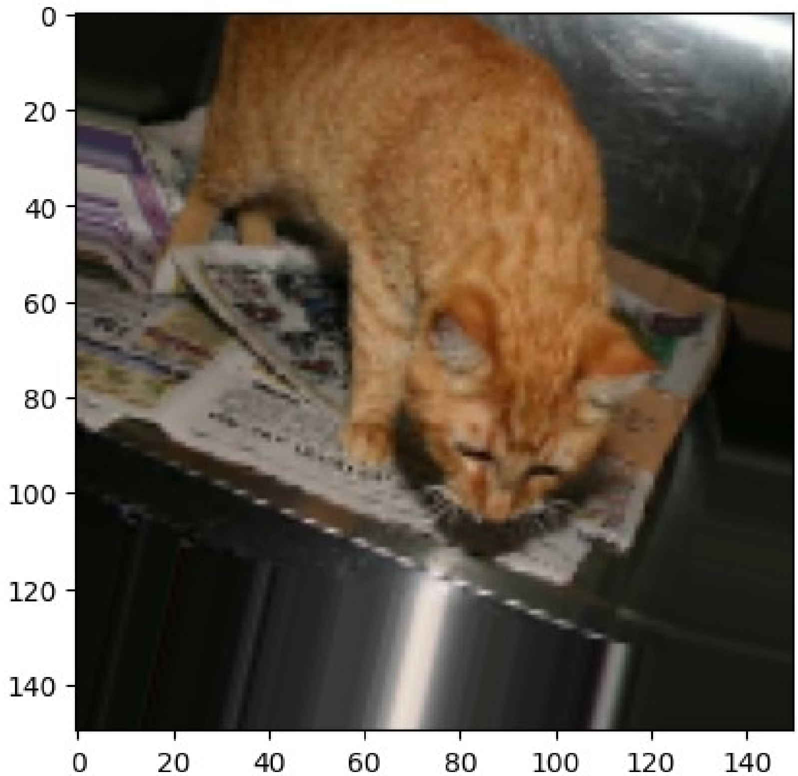
```
datagen = ImageDataGenerator(
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

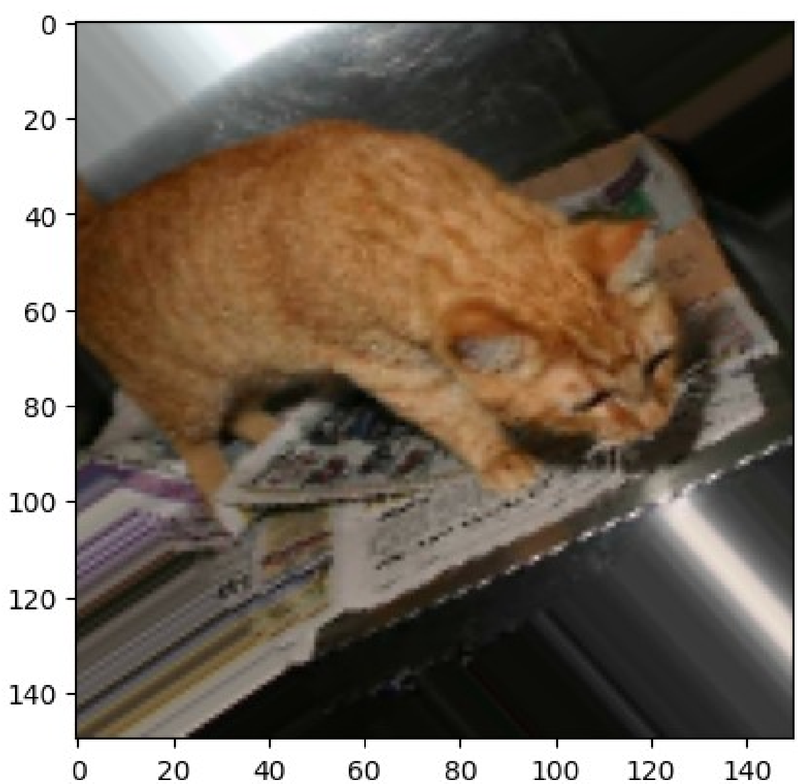
These are just a few of the options available (for more, see the Keras documentation). Let's quickly go over what we just wrote:

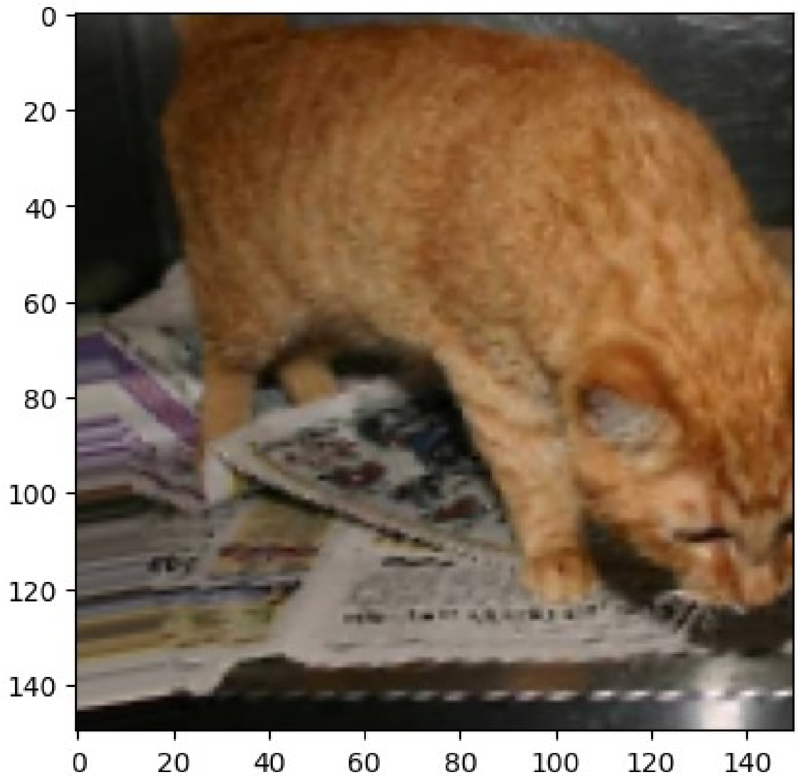
- `rotation_range` is a value in degrees (0-180), a range within which to randomly rotate pictures.
- `width_shift` and `height_shift` are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- `shear_range` is for randomly applying shearing transformations.
- `zoom_range` is for randomly zooming inside pictures.
- `horizontal_flip` is for randomly flipping half of the images horizontally -- relevant when there are no assumptions of horizontal asymmetry (e.g. real-world pictures).
- `fill_mode` is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

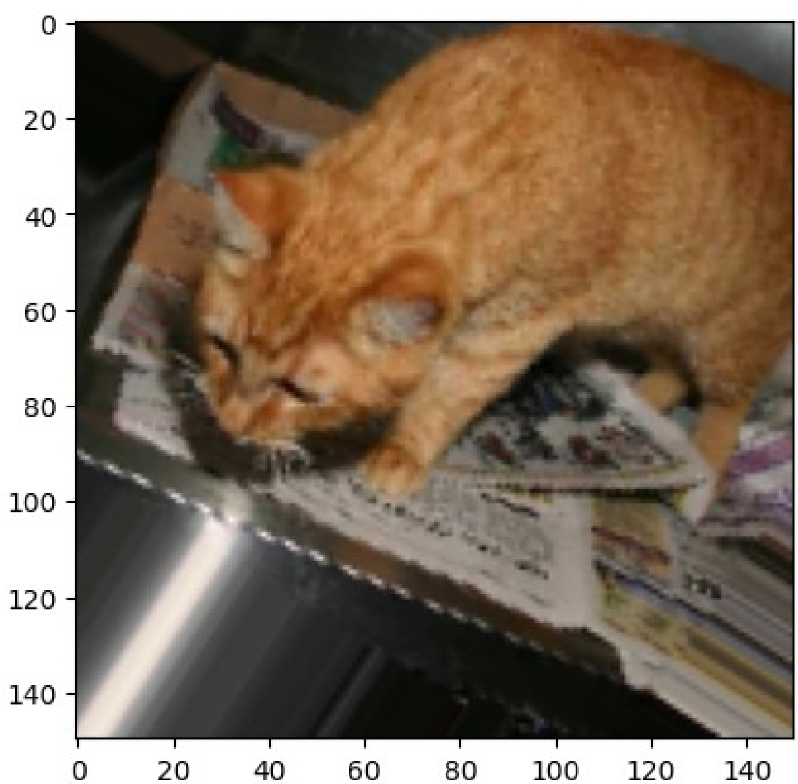
Let's take a look at our augmented images:

```
# This is module with image preprocessing utilities
import keras.utils as image

fnames = [os.path.join(train_cats_dir, fname) for fname in
os.listdir(train_cats_dir)]

# We pick one image to "augment"
img_path = fnames[3]

# Read the image and resize it
img = image.load_img(img_path, target_size=(150, 150))

# Convert it to a Numpy array with shape (150, 150, 3)
x = image.img_to_array(img)

# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

# The .flow() command below generates batches of randomly transformed
images.
# It will loop indefinitely, so we need to `break` the loop at some
point!
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```


0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140


0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140


0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

If we train a new network using this data augmentation configuration, our network will never see twice the same input. However, the inputs that it sees are still heavily intercorrelated, since they come from a small number of original images -- we cannot produce new information, we can only remix existing information. As such, this might not be quite enough to completely get rid of overfitting. To further fight overfitting, we will also add a Dropout layer to our model, right before the densely-connected classifier:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.
```

Let's train our network using data augmentation and dropout:

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,)

# Note that the validation data should not be augmented!
validation_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=32,
```

```
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation_generator = validation_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

history = model.fit(
      train_generator,
      steps_per_epoch=2000//train_generator.batch_size,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=1000//validation_generator.batch_size)
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/100
62/62 [==============================] - 19s 248ms/step - loss: 0.6998
- acc: 0.4924 - val_loss: 0.6924 - val_acc: 0.5192
Epoch 2/100
62/62 [==============================] - 16s 261ms/step - loss: 0.6962
- acc: 0.5391 - val_loss: 0.6895 - val_acc: 0.5081
Epoch 3/100
62/62 [==============================] - 16s 260ms/step - loss: 0.6809
- acc: 0.5772 - val_loss: 0.6757 - val_acc: 0.5554
Epoch 4/100
62/62 [==============================] - 16s 260ms/step - loss: 0.6880
- acc: 0.5777 - val_loss: 0.6665 - val_acc: 0.5998
Epoch 5/100
62/62 [==============================] - 30s 485ms/step - loss: 0.6656
- acc: 0.5899 - val_loss: 0.6563 - val_acc: 0.5897
Epoch 6/100
62/62 [==============================] - 15s 249ms/step - loss: 0.6540
- acc: 0.6098 - val_loss: 0.7147 - val_acc: 0.5433
Epoch 7/100
62/62 [==============================] - 16s 259ms/step - loss: 0.6606
- acc: 0.6098 - val_loss: 0.6527 - val_acc: 0.6149
Epoch 8/100
62/62 [==============================] - 16s 260ms/step - loss: 0.6534
- acc: 0.6265 - val_loss: 0.6471 - val_acc: 0.5988
Epoch 9/100
62/62 [==============================] - 21s 334ms/step - loss: 0.6441
- acc: 0.6418 - val_loss: 0.6556 - val_acc: 0.5988
Epoch 10/100
62/62 [==============================] - 16s 260ms/step - loss: 0.6449
- acc: 0.6336 - val_loss: 0.6326 - val_acc: 0.6462
Epoch 11/100
62/62 [==============================] - 16s 260ms/step - loss: 0.6340
```

```
- acc: 0.6484 - val_loss: 0.6131 - val_acc: 0.6552
Epoch 12/100
62/62 [==============================] - 23s 370ms/step - loss: 0.6201
- acc: 0.6540 - val_loss: 0.6371 - val_acc: 0.6361
Epoch 13/100
62/62 [==============================] - 18s 287ms/step - loss: 0.6037
- acc: 0.6748 - val_loss: 0.5997 - val_acc: 0.6724
Epoch 14/100
62/62 [==============================] - 15s 245ms/step - loss: 0.6115
- acc: 0.6682 - val_loss: 0.6227 - val_acc: 0.6542
Epoch 15/100
62/62 [==============================] - 19s 312ms/step - loss: 0.6078
- acc: 0.6575 - val_loss: 0.5861 - val_acc: 0.6905
Epoch 16/100
62/62 [==============================] - 16s 259ms/step - loss: 0.6078
- acc: 0.6682 - val_loss: 0.6085 - val_acc: 0.6855
Epoch 17/100
62/62 [==============================] - 16s 258ms/step - loss: 0.6207
- acc: 0.6575 - val_loss: 0.5763 - val_acc: 0.6915
Epoch 18/100
62/62 [==============================] - 15s 241ms/step - loss: 0.5812
- acc: 0.6870 - val_loss: 0.6016 - val_acc: 0.6653
Epoch 19/100
62/62 [==============================] - 16s 261ms/step - loss: 0.5830
- acc: 0.6789 - val_loss: 0.5731 - val_acc: 0.7157
Epoch 20/100
62/62 [==============================] - 16s 252ms/step - loss: 0.5834
- acc: 0.7078 - val_loss: 0.5968 - val_acc: 0.6835
Epoch 21/100
62/62 [==============================] - 20s 329ms/step - loss: 0.5843
- acc: 0.6890 - val_loss: 0.6033 - val_acc: 0.6784
Epoch 22/100
62/62 [==============================] - 16s 257ms/step - loss: 0.5853
- acc: 0.6926 - val_loss: 0.6072 - val_acc: 0.6784
Epoch 23/100
62/62 [==============================] - 15s 243ms/step - loss: 0.5744
- acc: 0.6977 - val_loss: 0.5536 - val_acc: 0.7188
Epoch 24/100
62/62 [==============================] - 15s 245ms/step - loss: 0.5657
- acc: 0.7007 - val_loss: 0.6416 - val_acc: 0.6492
Epoch 25/100
62/62 [==============================] - 16s 257ms/step - loss: 0.5642
- acc: 0.7109 - val_loss: 0.5648 - val_acc: 0.6946
Epoch 26/100
62/62 [==============================] - 17s 271ms/step - loss: 0.5694
- acc: 0.7012 - val_loss: 0.5498 - val_acc: 0.7278
Epoch 27/100
62/62 [==============================] - 16s 252ms/step - loss: 0.5691
- acc: 0.7058 - val_loss: 0.5478 - val_acc: 0.7268
```

```
Epoch 28/100
62/62 [==============================] - 15s 246ms/step - loss: 0.5534
- acc: 0.7114 - val_loss: 0.5817 - val_acc: 0.7046
Epoch 29/100
62/62 [==============================] - 16s 258ms/step - loss: 0.5499
- acc: 0.7251 - val_loss: 0.5248 - val_acc: 0.7359
Epoch 30/100
62/62 [==============================] - 15s 246ms/step - loss: 0.5536
- acc: 0.7073 - val_loss: 0.5351 - val_acc: 0.7319
Epoch 31/100
62/62 [==============================] - 15s 242ms/step - loss: 0.5497
- acc: 0.7170 - val_loss: 0.5065 - val_acc: 0.7661
Epoch 32/100
62/62 [==============================] - 15s 243ms/step - loss: 0.5493
- acc: 0.7109 - val_loss: 0.5293 - val_acc: 0.7329
Epoch 33/100
62/62 [==============================] - 16s 261ms/step - loss: 0.5381
- acc: 0.7317 - val_loss: 0.5857 - val_acc: 0.7198
Epoch 34/100
62/62 [==============================] - 22s 357ms/step - loss: 0.5361
- acc: 0.7368 - val_loss: 0.5880 - val_acc: 0.7349
Epoch 35/100
62/62 [==============================] - 16s 260ms/step - loss: 0.5169
- acc: 0.7353 - val_loss: 0.5308 - val_acc: 0.7480
Epoch 36/100
62/62 [==============================] - 16s 257ms/step - loss: 0.5629
- acc: 0.7241 - val_loss: 0.5130 - val_acc: 0.7409
Epoch 37/100
62/62 [==============================] - 16s 258ms/step - loss: 0.5251
- acc: 0.7373 - val_loss: 0.5239 - val_acc: 0.7450
Epoch 38/100
62/62 [==============================] - 17s 267ms/step - loss: 0.5330
- acc: 0.7332 - val_loss: 0.5529 - val_acc: 0.7248
Epoch 39/100
62/62 [==============================] - 32s 507ms/step - loss: 0.5157
- acc: 0.7464 - val_loss: 0.5117 - val_acc: 0.7520
Epoch 40/100
62/62 [==============================] - 26s 426ms/step - loss: 0.5374
- acc: 0.7215 - val_loss: 0.5040 - val_acc: 0.7601
Epoch 41/100
62/62 [==============================] - 26s 421ms/step - loss: 0.5255
- acc: 0.7449 - val_loss: 0.5059 - val_acc: 0.7560
Epoch 42/100
62/62 [==============================] - 31s 496ms/step - loss: 0.5334
- acc: 0.7292 - val_loss: 0.4942 - val_acc: 0.7631
Epoch 43/100
62/62 [==============================] - 15s 240ms/step - loss: 0.5178
- acc: 0.7358 - val_loss: 0.4801 - val_acc: 0.7712
Epoch 44/100
```

```
62/62 [==============================] - 16s 257ms/step - loss: 0.5170
- acc: 0.7464 - val_loss: 0.5281 - val_acc: 0.7500
Epoch 45/100
62/62 [==============================] - 15s 245ms/step - loss: 0.5146
- acc: 0.7393 - val_loss: 0.5060 - val_acc: 0.7591
Epoch 46/100
62/62 [==============================] - 16s 258ms/step - loss: 0.5105
- acc: 0.7424 - val_loss: 0.4974 - val_acc: 0.7631
Epoch 47/100
62/62 [==============================] - 16s 258ms/step - loss: 0.5098
- acc: 0.7490 - val_loss: 0.5020 - val_acc: 0.7853
Epoch 48/100
62/62 [==============================] - 16s 261ms/step - loss: 0.5037
- acc: 0.7571 - val_loss: 0.5309 - val_acc: 0.7571
Epoch 49/100
62/62 [==============================] - 16s 266ms/step - loss: 0.4891
- acc: 0.7632 - val_loss: 0.4789 - val_acc: 0.7792
Epoch 50/100
62/62 [==============================] - 16s 264ms/step - loss: 0.4963
- acc: 0.7647 - val_loss: 0.5153 - val_acc: 0.7853
Epoch 51/100
62/62 [==============================] - 15s 244ms/step - loss: 0.5080
- acc: 0.7617 - val_loss: 0.4663 - val_acc: 0.7853
Epoch 52/100
62/62 [==============================] - 16s 253ms/step - loss: 0.4998
- acc: 0.7597 - val_loss: 0.4643 - val_acc: 0.7823
Epoch 53/100
62/62 [==============================] - 15s 247ms/step - loss: 0.4784
- acc: 0.7551 - val_loss: 0.4826 - val_acc: 0.7843
Epoch 54/100
62/62 [==============================] - 15s 250ms/step - loss: 0.4822
- acc: 0.7663 - val_loss: 0.4851 - val_acc: 0.7772
Epoch 55/100
62/62 [==============================] - 17s 266ms/step - loss: 0.4883
- acc: 0.7729 - val_loss: 0.5065 - val_acc: 0.7530
Epoch 56/100
62/62 [==============================] - 15s 240ms/step - loss: 0.4677
- acc: 0.7713 - val_loss: 0.4678 - val_acc: 0.7903
Epoch 57/100
62/62 [==============================] - 15s 240ms/step - loss: 0.4720
- acc: 0.7576 - val_loss: 0.4648 - val_acc: 0.7863
Epoch 58/100
62/62 [==============================] - 16s 253ms/step - loss: 0.4812
- acc: 0.7661 - val_loss: 1.7104 - val_acc: 0.5746
Epoch 59/100
62/62 [==============================] - 16s 258ms/step - loss: 0.5006
- acc: 0.7673 - val_loss: 0.5322 - val_acc: 0.7671
Epoch 60/100
62/62 [==============================] - 15s 246ms/step - loss: 0.4639
```

```
- acc: 0.7825 - val_loss: 0.4515 - val_acc: 0.8054
Epoch 61/100
62/62 [==============================] - 17s 271ms/step - loss: 0.4545
- acc: 0.7846 - val_loss: 0.5061 - val_acc: 0.7661
Epoch 62/100
62/62 [==============================] - 15s 242ms/step - loss: 0.4582
- acc: 0.7815 - val_loss: 0.4418 - val_acc: 0.8125
Epoch 63/100
62/62 [==============================] - 15s 241ms/step - loss: 0.4602
- acc: 0.7790 - val_loss: 0.4597 - val_acc: 0.8115
Epoch 64/100
62/62 [==============================] - 16s 256ms/step - loss: 0.4608
- acc: 0.7957 - val_loss: 0.5512 - val_acc: 0.7329
Epoch 65/100
62/62 [==============================] - 15s 242ms/step - loss: 0.4640
- acc: 0.7922 - val_loss: 0.5032 - val_acc: 0.7923
Epoch 66/100
62/62 [==============================] - 16s 257ms/step - loss: 0.4630
- acc: 0.7785 - val_loss: 0.4497 - val_acc: 0.8044
Epoch 67/100
62/62 [==============================] - 15s 240ms/step - loss: 0.4625
- acc: 0.7856 - val_loss: 0.4950 - val_acc: 0.7802
Epoch 68/100
62/62 [==============================] - 16s 260ms/step - loss: 0.4656
- acc: 0.7983 - val_loss: 0.4578 - val_acc: 0.7954
Epoch 69/100
62/62 [==============================] - 16s 255ms/step - loss: 0.4438
- acc: 0.7891 - val_loss: 0.4412 - val_acc: 0.8054
Epoch 70/100
62/62 [==============================] - 15s 240ms/step - loss: 0.4662
- acc: 0.7881 - val_loss: 0.5337 - val_acc: 0.7792
Epoch 71/100
62/62 [==============================] - 15s 242ms/step - loss: 0.4420
- acc: 0.7876 - val_loss: 0.4125 - val_acc: 0.8306
Epoch 72/100
62/62 [==============================] - 16s 257ms/step - loss: 0.4578
- acc: 0.7825 - val_loss: 0.4154 - val_acc: 0.8266
Epoch 73/100
62/62 [==============================] - 15s 245ms/step - loss: 0.4396
- acc: 0.7881 - val_loss: 0.4096 - val_acc: 0.8367
Epoch 74/100
62/62 [==============================] - 15s 247ms/step - loss: 0.4528
- acc: 0.7942 - val_loss: 0.4441 - val_acc: 0.8054
Epoch 75/100
62/62 [==============================] - 15s 241ms/step - loss: 0.4486
- acc: 0.7973 - val_loss: 0.4678 - val_acc: 0.8075
Epoch 76/100
62/62 [==============================] - 15s 239ms/step - loss: 0.4508
- acc: 0.7815 - val_loss: 0.4106 - val_acc: 0.8165
```

```
Epoch 77/100
62/62 [==============================] - 15s 240ms/step - loss: 0.4406
- acc: 0.7957 - val_loss: 0.4294 - val_acc: 0.8206
Epoch 78/100
62/62 [==============================] - 16s 257ms/step - loss: 0.4230
- acc: 0.8079 - val_loss: 0.4793 - val_acc: 0.8034
Epoch 79/100
62/62 [==============================] - 16s 256ms/step - loss: 0.4435
- acc: 0.7978 - val_loss: 0.4113 - val_acc: 0.8175
Epoch 80/100
62/62 [==============================] - 16s 263ms/step - loss: 0.4339
- acc: 0.7927 - val_loss: 0.3989 - val_acc: 0.8236
Epoch 81/100
62/62 [==============================] - 16s 259ms/step - loss: 0.4146
- acc: 0.8084 - val_loss: 0.4254 - val_acc: 0.8175
Epoch 82/100
62/62 [==============================] - 15s 249ms/step - loss: 0.4160
- acc: 0.8039 - val_loss: 0.4161 - val_acc: 0.8165
Epoch 83/100
62/62 [==============================] - 15s 246ms/step - loss: 0.4110
- acc: 0.8105 - val_loss: 0.3970 - val_acc: 0.8317
Epoch 84/100
62/62 [==============================] - 16s 260ms/step - loss: 0.4131
- acc: 0.7978 - val_loss: 0.4618 - val_acc: 0.8196
Epoch 85/100
62/62 [==============================] - 15s 242ms/step - loss: 0.4335
- acc: 0.8039 - val_loss: 0.3908 - val_acc: 0.8256
Epoch 86/100
62/62 [==============================] - 16s 258ms/step - loss: 0.3913
- acc: 0.8262 - val_loss: 0.3922 - val_acc: 0.8387
Epoch 87/100
62/62 [==============================] - 16s 258ms/step - loss: 0.3927
- acc: 0.8201 - val_loss: 0.5115 - val_acc: 0.8145
Epoch 88/100
62/62 [==============================] - 17s 271ms/step - loss: 0.4092
- acc: 0.8176 - val_loss: 0.4436 - val_acc: 0.8135
Epoch 89/100
62/62 [==============================] - 16s 258ms/step - loss: 0.4000
- acc: 0.8242 - val_loss: 0.4316 - val_acc: 0.8155
Epoch 90/100
62/62 [==============================] - 16s 259ms/step - loss: 0.4080
- acc: 0.8206 - val_loss: 0.3912 - val_acc: 0.8417
Epoch 91/100
62/62 [==============================] - 16s 251ms/step - loss: 0.3968
- acc: 0.8145 - val_loss: 0.3726 - val_acc: 0.8357
Epoch 92/100
62/62 [==============================] - 20s 327ms/step - loss: 0.4027
- acc: 0.8191 - val_loss: 0.4301 - val_acc: 0.8317
Epoch 93/100
```

```
62/62 [==============================] - 15s 243ms/step - loss: 0.3995
- acc: 0.8216 - val_loss: 0.4046 - val_acc: 0.8377
Epoch 94/100
62/62 [==============================] - 15s 244ms/step - loss: 0.3944
- acc: 0.8272 - val_loss: 0.3691 - val_acc: 0.8488
Epoch 95/100
62/62 [==============================] - 15s 241ms/step - loss: 0.4102
- acc: 0.8196 - val_loss: 0.3592 - val_acc: 0.8498
Epoch 96/100
62/62 [==============================] - 22s 364ms/step - loss: 0.3847
- acc: 0.8298 - val_loss: 0.4010 - val_acc: 0.8488
Epoch 97/100
62/62 [==============================] - 15s 243ms/step - loss: 0.3998
- acc: 0.8262 - val_loss: 0.3674 - val_acc: 0.8357
Epoch 98/100
62/62 [==============================] - 16s 258ms/step - loss: 0.3889
- acc: 0.8288 - val_loss: 0.4280 - val_acc: 0.8327
Epoch 99/100
62/62 [==============================] - 15s 243ms/step - loss: 0.3714
- acc: 0.8379 - val_loss: 0.4775 - val_acc: 0.7853
Epoch 100/100
62/62 [==============================] - 15s 241ms/step - loss: 0.3661
- acc: 0.8338 - val_loss: 0.4617 - val_acc: 0.8075
```

Let's save our model -- we will be using it in the section on convnet visualization.

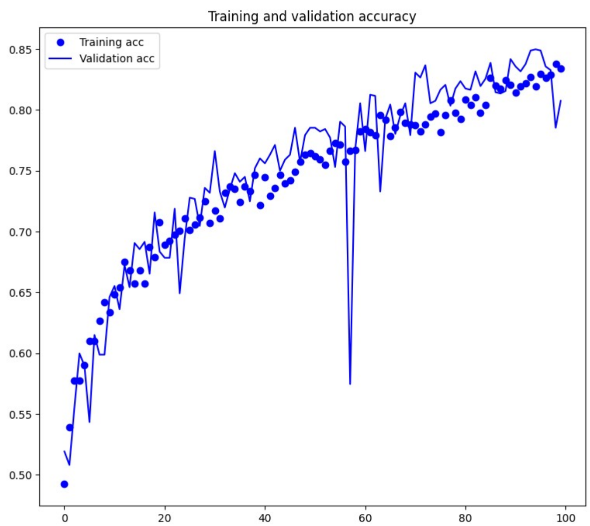
```
model.save('cats_and_dogs_small_2.h5')
```

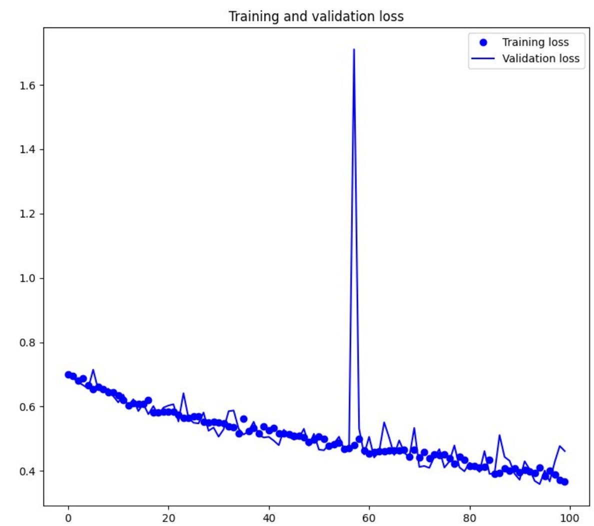
```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/
training.py:3000: UserWarning: You are saving your model as an HDF5
file via `model.save()`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')`.
  saving_api.save_model(
```

Let's plot our results again:

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.figure(figsize= (20,8))

plt.subplot(1,2,1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
```

```
plt.title('Training and validation accuracy')
plt.legend()

# plt.figure()

plt.subplot(1,2,2)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```





Thanks to data augmentation and dropout, we are no longer overfitting: the training curves are rather closely tracking the validation curves. We are now able to reach an accuracy of 82%, a 15% relative improvement over the non-regularized model.

By leveraging regularization techniques even further and by tuning the network's parameters (such as the number of filters per convolution layer, or the number of layers in the network), we may be able to get an even better accuracy, likely up to 86-87%. However, it would prove very difficult to go any higher just by training our own convnet from scratch, simply because we have so little data to work with. As a next step to improve our accuracy on this problem, we will have to leverage a pre-trained model, which will be the focus of the next two sections.

Plot the ROC curves

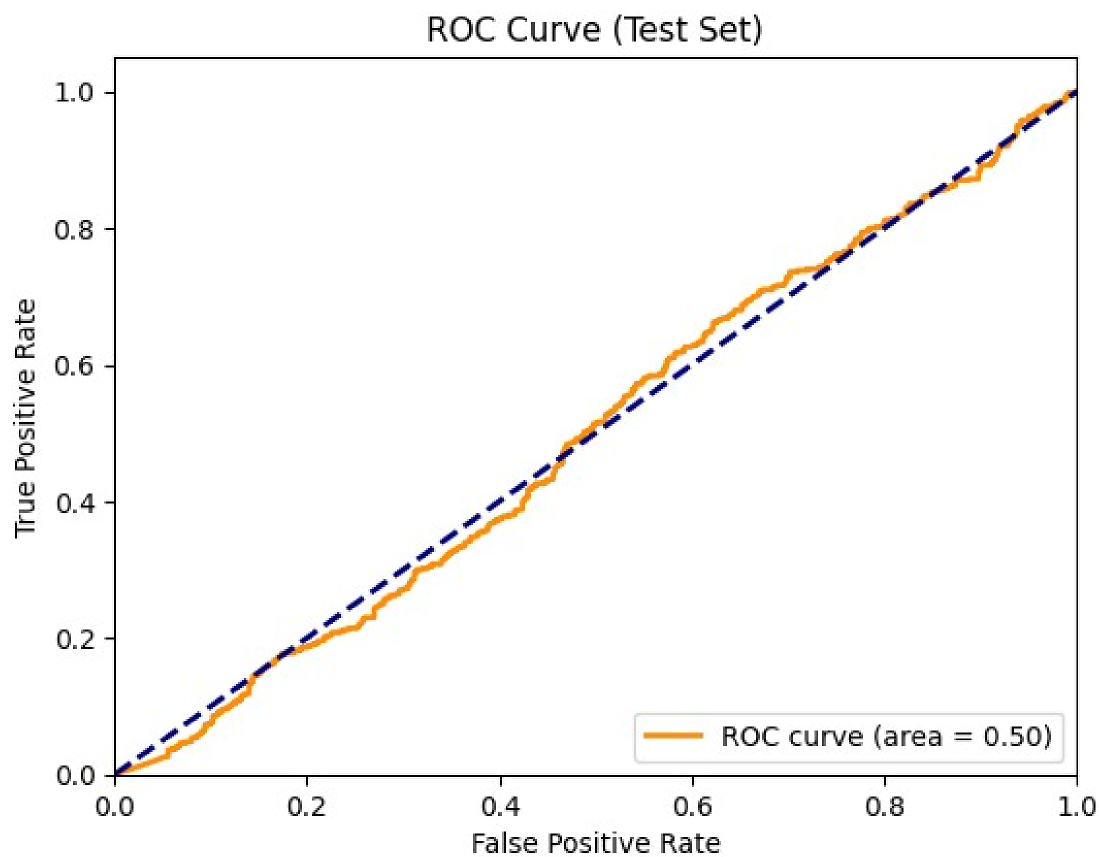
```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')
```

```
Found 1000 images belonging to 2 classes.
```

```python
def find_labels_and_probability(img_gen):
  y_true = img_gen.classes

  # Get predicted probabilities (y_score)
  y_pred = model.predict(img_gen)

  return y_true, y_pred

# train_y_true, train_probs =
find_labels_and_probability(train_generator)
# val_y_true, val_probs =
find_labels_and_probability(validation_generator)
test_y_true, test_probs = find_labels_and_probability(test_generator)
```

```
32/32 [==============================] - 2s 48ms/step
```

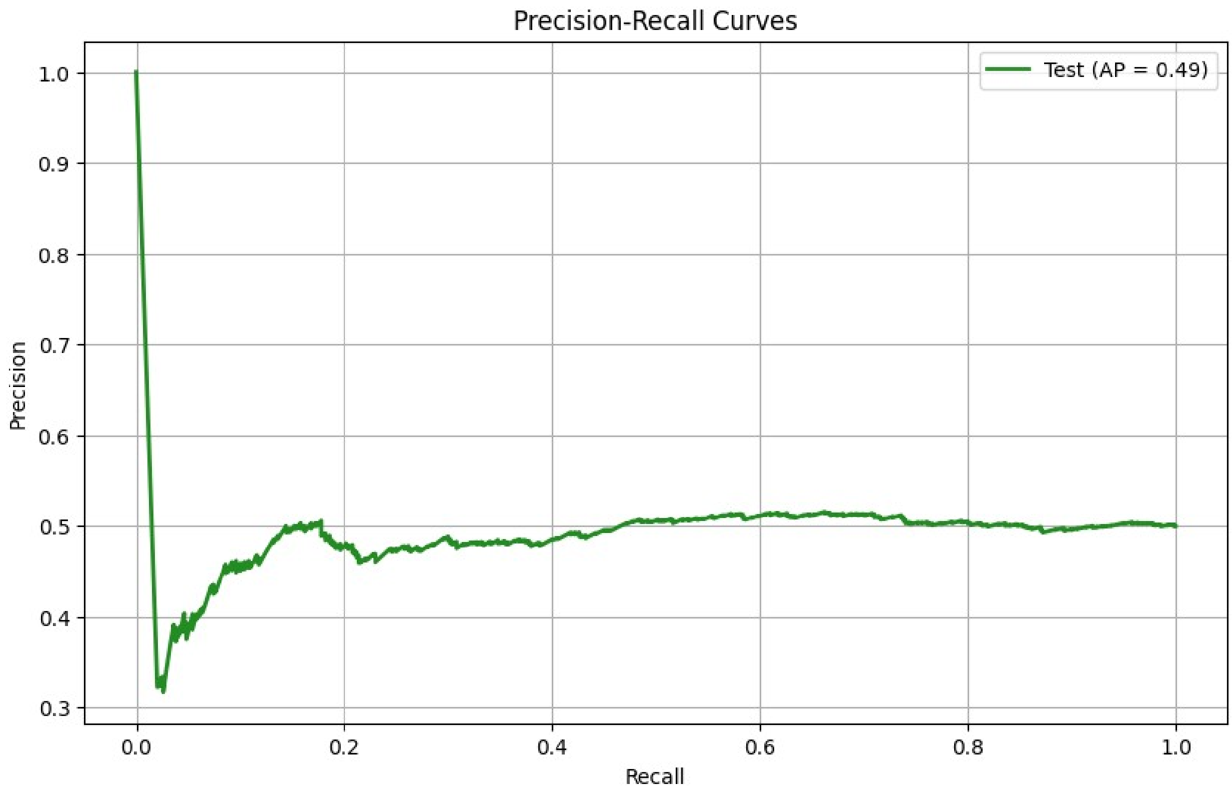
```python
# Function to plot ROC curve
def plot_roc_curve(y_true, y_score, title):
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()

# Plot ROC curves for training, validation, and test sets
# plot_roc_curve(train_y_true, train_probs, title='ROC Curve (Training
Set)')
# plot_roc_curve(val_y_true, val_probs, title='ROC Curve (Validation
Set)')
plot_roc_curve(test_y_true, test_probs, title='ROC Curve (Test Set)')
```



Plot the Recall-Precision curves

```
from sklearn.metrics import precision_recall_curve,
average_precision_score

# Compute precision and recall for each dataset
# train_precision, train_recall, threshold =
precision_recall_curve(train_y_true, train_probs)
# val_precision, val_recall, threshold =
precision_recall_curve(val_y_true, val_probs)
test_precision, test_recall, threshold =
precision_recall_curve(test_y_true, test_probs)

# Compute average precision (AUC-PR) for each dataset
# train_average_precision = average_precision_score(train_y_true,
train_probs)
# val_average_precision = average_precision_score(val_y_true,
val_probs)
test_average_precision = average_precision_score(test_y_true,
test_probs)

# Plot Precision-Recall curves for all datasets
plt.figure(figsize=(10, 6))
# plt.plot(train_recall, train_precision, color='darkorange', lw=2,
```

```
label=f'Training (AP = {train_average_precision:.2f})')
# plt.plot(val_recall, val_precision, color='royalblue', lw=2,
label=f'Validation (AP = {val_average_precision:.2f})')
plt.plot(test_recall, test_precision, color='forestgreen', lw=2,
label=f'Test (AP = {test_average_precision:.2f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```

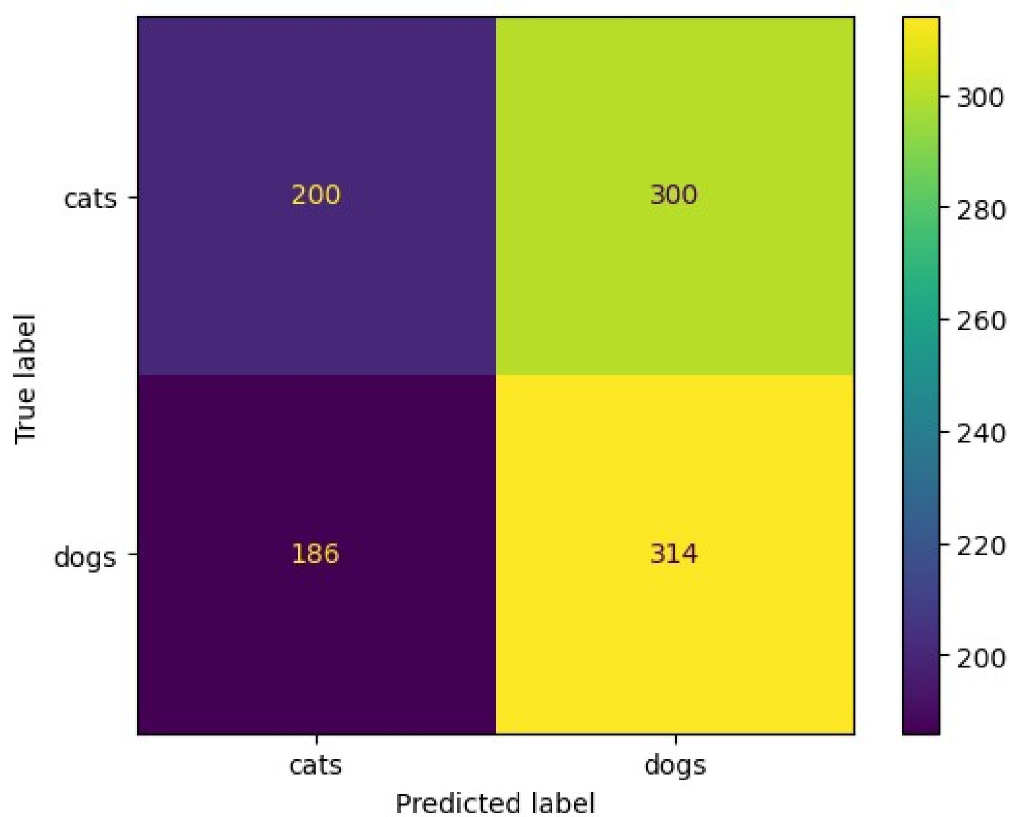


Confusion matrix for 50% threshold

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def find_labels_and_probability(y_true, y_probs):
  threshold = 0.5
  # Convert probabilities to binary classes using the 50% threshold
  y_pred = (y_probs >= threshold).astype(int)  # Predicted classes (0
or 1)

  # Create the confusion matrix
  conf_matrix = confusion_matrix(y_true, y_pred)
```

```python
    return conf_matrix

# val_conf_matrix = find_labels_and_probability(val_y_true, val_probs)
test_conf_matrix = find_labels_and_probability(test_y_true,
test_probs)

# Print the confusion matrix
print("Confusion Matrix Test Data (Threshold = 50%):")
print(test_conf_matrix)

# Display the confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=test_conf_matrix,
display_labels=test_generator.class_indices)
disp.plot(cmap='viridis', values_format='d')

# print("Confusion Matrix Validation Data (Threshold = 50%):")
# print(val_conf_matrix)
```

```
Confusion Matrix Test Data (Threshold = 50%):
[[200 300]
 [186 314]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7917283d72e0>
```



Optuna to find the best hyperparameters

```
!pip install --quiet optuna
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/404.2 kB ? eta -:--:--
━━━━━━━━━━━━━━━━╸━━━━━━━━━━━━━━━━━━━━━━━ 163.8/404.2 kB 4.9 MB/s eta
0:00:01 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 404.2/404.2 kB 7.3
MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 226.0/226.0 kB 24.9 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 78.7/78.7 kB 10.0 MB/s eta
0:00:00
```

```
import optuna
```

tune model with hyperparameters

```
def build_cnn_dog_cat_model(learning_rate, num_filters,
                              dropout_rate, batch_size):
  model = keras.Sequential()

  model.add(layers.Conv2D(num_filters, (3, 3), activation='relu',
input_shape=(150, 150, 3)))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(num_filters * 2, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(num_filters * 4, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(num_filters * 4, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Flatten())
  model.add(layers.Dropout(dropout_rate))
  model.add(layers.Dense(512, activation='relu'))
  model.add(layers.Dense(1, activation='sigmoid'))

  # if optimizer_name == 'rmsprop':
  #           optimizer = optimizers.RMSprop(lr=learning_rate)
  # elif optimizer_name == 'adam':
  #         optimizer = optimizers.Adam(learning_rate=learning_rate)
  # elif optimizer_name == 'sgd':
  #         optimizer = optimizers.SGD(learning_rate=learning_rate,
nesterov=True)
  # else:
  #     raise ValueError("Invalid optimizer name")

  model.compile(loss='binary_crossentropy',
                optimizer=optimizers.RMSprop(lr=learning_rate),
                metrics=['acc'])

  # model.compile(loss='binary_crossentropy',
```

```
  #                 optimizer=optimizer,
  #                 metrics=['acc'])

  return model
```

define the objective function

```
def objective(trial):
  # Define hyperparameters to optimize
  # optimizer_name = trial.suggest_categorical('optimizer',
['rmsprop', 'adam', 'sgd' ])
  learning_rate = trial.suggest_loguniform('learning_rate', 1e-5, 1e-
1)
  num_filters = trial.suggest_int('num_filters', 32, 256)
  dropout_rate = trial.suggest_uniform('dropout_rate', 0.0, 0.5)
  batch_size = trial.suggest_categorical('batch_size', [16, 32, 64])
  rotation_range = trial.suggest_int('rotation_range', 0, 180)
  width_shift_range = trial.suggest_float('width_shift_range', 0.0,
1.0)
  height_shift_range = trial.suggest_float('height_shift_range', 0.0,
1.0)
  shear_range = trial.suggest_float('shear_range', 0.0, 1.0)
  zoom_range = trial.suggest_float('zoom_range', 0.0, 1.0)
  horizontal_flip = trial.suggest_categorical('horizontal_flip',
[True, False])

  train_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=rotation_range,
        width_shift_range=width_shift_range,
        height_shift_range=height_shift_range,
        shear_range=shear_range,
        zoom_range=zoom_range,
        horizontal_flip=horizontal_flip,
  )

  train_generator = train_datagen.flow_from_directory(
          # This is the target directory
          train_dir,
          # All images will be resized to 150x150
          target_size=(150, 150),
          batch_size=batch_size,
          # Since we use binary_crossentropy loss, we need binary
labels
          class_mode='binary')

  validation_generator = validation_datagen.flow_from_directory(
          validation_dir,
          target_size=(150, 150),
          batch_size=batch_size,
```

```
          class_mode='binary')

  model_optuna = build_cnn_dog_cat_model(learning_rate, num_filters,
dropout_rate, batch_size)

  history = model_optuna.fit(
      train_generator,
      steps_per_epoch=2000//train_generator.batch_size,
      epochs=20,
      validation_data=validation_generator,
      validation_steps=1000//validation_generator.batch_size)

  # Evaluate the model on the validation set
  val_loss, val_acc = model.evaluate(validation_generator,
steps=len(validation_generator))

  # Return the validation accuracy as the objective to optimize
  return val_acc

# Create an Optuna study
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=3)  # You can adjust the number of
trials

# Get the best hyperparameters
best_params = study.best_params
print("Best Hyperparameters:", best_params)
```

```
[I 2023-10-09 04:00:37,356] A new study created in memory with name:
no-name-d87da641-e89b-4d4f-a8f5-8471973e6cf2
```

```
Found 2000 images belonging to 2 classes.
```

```
<ipython-input-51-2d5146648bdb>:4: FutureWarning: suggest_loguniform
has been deprecated in v3.0.0. This feature will be removed in v6.0.0.
See https://github.com/optuna/optuna/releases/tag/v3.0.0. Use
suggest_float(..., log=True) instead.
  learning_rate = trial.suggest_loguniform('learning_rate', 1e-5, 1e-
1)
<ipython-input-51-2d5146648bdb>:6: FutureWarning: suggest_uniform has
been deprecated in v3.0.0. This feature will be removed in v6.0.0. See
https://github.com/optuna/optuna/releases/tag/v3.0.0. Use
suggest_float instead.
  dropout_rate = trial.suggest_uniform('dropout_rate', 0.0, 0.5)
```

```
Found 1000 images belonging to 2 classes.
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.
```

```
Epoch 1/20
125/125 [==============================] - 43s 249ms/step - loss:
0.7035 - acc: 0.4980 - val_loss: 0.6931 - val_acc: 0.5020
Epoch 2/20
125/125 [==============================] - 20s 163ms/step - loss:
0.6939 - acc: 0.5085 - val_loss: 0.6923 - val_acc: 0.5101
Epoch 3/20
125/125 [==============================] - 18s 145ms/step - loss:
0.6930 - acc: 0.5400 - val_loss: 0.6916 - val_acc: 0.5010
Epoch 4/20
125/125 [==============================] - 19s 155ms/step - loss:
0.6954 - acc: 0.5345 - val_loss: 0.6840 - val_acc: 0.5625
Epoch 5/20
125/125 [==============================] - 19s 147ms/step - loss:
0.6947 - acc: 0.5500 - val_loss: 0.6816 - val_acc: 0.5655
Epoch 6/20
125/125 [==============================] - 20s 158ms/step - loss:
0.6882 - acc: 0.5695 - val_loss: 0.6861 - val_acc: 0.5585
Epoch 7/20
125/125 [==============================] - 19s 151ms/step - loss:
0.6885 - acc: 0.5615 - val_loss: 0.6758 - val_acc: 0.5696
Epoch 8/20
125/125 [==============================] - 20s 163ms/step - loss:
0.6813 - acc: 0.5675 - val_loss: 0.7115 - val_acc: 0.5524
Epoch 9/20
125/125 [==============================] - 18s 145ms/step - loss:
0.6849 - acc: 0.5715 - val_loss: 0.6970 - val_acc: 0.5302
Epoch 10/20
125/125 [==============================] - 20s 164ms/step - loss:
0.6834 - acc: 0.5680 - val_loss: 0.6683 - val_acc: 0.5756
Epoch 11/20
125/125 [==============================] - 18s 147ms/step - loss:
0.6757 - acc: 0.5900 - val_loss: 0.6739 - val_acc: 0.5514
Epoch 12/20
125/125 [==============================] - 32s 253ms/step - loss:
0.6780 - acc: 0.5685 - val_loss: 0.6658 - val_acc: 0.5706
Epoch 13/20
125/125 [==============================] - 22s 175ms/step - loss:
0.6708 - acc: 0.5775 - val_loss: 0.6628 - val_acc: 0.5817
Epoch 14/20
125/125 [==============================] - 29s 232ms/step - loss:
0.6706 - acc: 0.5750 - val_loss: 0.6636 - val_acc: 0.5877
Epoch 15/20
125/125 [==============================] - 22s 178ms/step - loss:
0.6614 - acc: 0.6080 - val_loss: 0.6662 - val_acc: 0.5766
Epoch 16/20
125/125 [==============================] - 23s 181ms/step - loss:
0.6708 - acc: 0.5905 - val_loss: 0.6650 - val_acc: 0.5827
Epoch 17/20
125/125 [==============================] - 26s 212ms/step - loss:
```

```
0.6651 - acc: 0.5875 - val_loss: 0.6642 - val_acc: 0.5958
Epoch 18/20
125/125 [==============================] - 19s 150ms/step - loss:
0.6674 - acc: 0.5720 - val_loss: 0.6617 - val_acc: 0.5675
Epoch 19/20
125/125 [==============================] - 32s 258ms/step - loss:
0.6652 - acc: 0.5905 - val_loss: 0.6674 - val_acc: 0.5776
Epoch 20/20
125/125 [==============================] - 25s 195ms/step - loss:
0.6709 - acc: 0.5950 - val_loss: 0.6583 - val_acc: 0.6018
63/63 [==============================] - 2s 25ms/step - loss: 0.4627 -
acc: 0.8070

[I 2023-10-09 04:09:48,733] Trial 0 finished with value:
0.8069999814033508 and parameters: {'learning_rate':
0.00013019564762250 21, 'num_filters': 122, 'dropout_rate':
0.40332556704914646, 'batch_size': 16, 'rotation_range': 164,
'width_shift_range': 0.4711457168062444, 'height_shift_range':
0.5650464421060059, 'shear_range': 0.7091916436488243, 'zoom_range':
0.626790424124882, 'horizontal_flip': False}. Best is trial 0 with
value: 0.8069999814033508.

Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.

WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.

Epoch 1/20
31/31 [==============================] - 31s 635ms/step - loss: 0.6974
- acc: 0.5026 - val_loss: 0.6926 - val_acc: 0.5052
Epoch 2/20
31/31 [==============================] - 29s 884ms/step - loss: 0.6936
- acc: 0.5181 - val_loss: 0.6945 - val_acc: 0.4938
Epoch 3/20
31/31 [==============================] - 23s 704ms/step - loss: 0.6952
- acc: 0.4969 - val_loss: 0.6929 - val_acc: 0.5063
Epoch 4/20
31/31 [==============================] - 19s 610ms/step - loss: 0.6934
- acc: 0.4979 - val_loss: 0.6927 - val_acc: 0.5063
Epoch 5/20
31/31 [==============================] - 19s 606ms/step - loss: 0.6938
- acc: 0.4824 - val_loss: 0.6917 - val_acc: 0.5312
Epoch 6/20
31/31 [==============================] - 26s 834ms/step - loss: 0.6920
- acc: 0.5217 - val_loss: 0.6926 - val_acc: 0.5031
Epoch 7/20
31/31 [==============================] - 21s 657ms/step - loss: 0.6958
- acc: 0.5486 - val_loss: 0.6866 - val_acc: 0.5813
```

```
Epoch 8/20
31/31 [==============================] - 21s 656ms/step - loss: 0.6886
- acc: 0.5604 - val_loss: 0.6873 - val_acc: 0.5573
Epoch 9/20
31/31 [==============================] - 17s 557ms/step - loss: 0.6801
- acc: 0.5790 - val_loss: 0.7881 - val_acc: 0.5010
Epoch 10/20
31/31 [==============================] - 26s 853ms/step - loss: 0.6787
- acc: 0.5723 - val_loss: 0.8278 - val_acc: 0.4979
Epoch 11/20
31/31 [==============================] - 23s 741ms/step - loss: 0.6782
- acc: 0.5558 - val_loss: 0.6753 - val_acc: 0.5531
Epoch 12/20
31/31 [==============================] - 18s 579ms/step - loss: 0.6795
- acc: 0.5661 - val_loss: 0.8324 - val_acc: 0.5094
Epoch 13/20
31/31 [==============================] - 19s 606ms/step - loss: 0.7029
- acc: 0.5573 - val_loss: 0.6700 - val_acc: 0.5948
Epoch 14/20
31/31 [==============================] - 20s 655ms/step - loss: 0.6735
- acc: 0.5919 - val_loss: 0.6687 - val_acc: 0.5667
Epoch 15/20
31/31 [==============================] - 26s 821ms/step - loss: 0.6727
- acc: 0.5847 - val_loss: 0.6627 - val_acc: 0.6021
Epoch 16/20
31/31 [==============================] - 21s 664ms/step - loss: 0.6668
- acc: 0.5770 - val_loss: 0.6737 - val_acc: 0.5708
Epoch 17/20
31/31 [==============================] - 18s 566ms/step - loss: 0.6750
- acc: 0.5976 - val_loss: 0.6734 - val_acc: 0.5562
Epoch 18/20
31/31 [==============================] - 18s 584ms/step - loss: 0.6674
- acc: 0.5749 - val_loss: 0.7187 - val_acc: 0.5417
Epoch 19/20
31/31 [==============================] - 18s 560ms/step - loss: 0.6729
- acc: 0.5992 - val_loss: 0.6589 - val_acc: 0.6146
Epoch 20/20
31/31 [==============================] - 18s 583ms/step - loss: 0.6632
- acc: 0.5978 - val_loss: 0.6513 - val_acc: 0.6229
16/16 [==============================] - 2s 108ms/step - loss: 0.4627
- acc: 0.8070

[I 2023-10-09 04:18:38,887] Trial 1 finished with value:
0.8069999814033508 and parameters: {'learning_rate':
0.005526225298297534, 'num_filters': 112, 'dropout_rate':
0.16366027603344935, 'batch_size': 64, 'rotation_range': 3,
'width_shift_range': 0.8135198800366643, 'height_shift_range':
0.40614169317451276, 'shear_range': 0.8646171783728398, 'zoom_range':
0.37871191758122347, 'horizontal_flip': False}. Best is trial 0 with
value: 0.8069999814033508.
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.

WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.

Epoch 1/20
125/125 [==============================] - 23s 154ms/step - loss:
0.7042 - acc: 0.4965 - val_loss: 0.6931 - val_acc: 0.5000
Epoch 2/20
125/125 [==============================] - 19s 154ms/step - loss:
0.6933 - acc: 0.5010 - val_loss: 0.6942 - val_acc: 0.4960
Epoch 3/20
125/125 [==============================] - 20s 156ms/step - loss:
0.6925 - acc: 0.5390 - val_loss: 0.6850 - val_acc: 0.5444
Epoch 4/20
125/125 [==============================] - 19s 155ms/step - loss:
0.6904 - acc: 0.5505 - val_loss: 0.6900 - val_acc: 0.5192
Epoch 5/20
125/125 [==============================] - 20s 157ms/step - loss:
0.6910 - acc: 0.5760 - val_loss: 0.6781 - val_acc: 0.5544
Epoch 6/20
125/125 [==============================] - 19s 150ms/step - loss:
0.6838 - acc: 0.5715 - val_loss: 0.6667 - val_acc: 0.5726
Epoch 7/20
125/125 [==============================] - 20s 156ms/step - loss:
0.6771 - acc: 0.5810 - val_loss: 0.7212 - val_acc: 0.5060
Epoch 8/20
125/125 [==============================] - 19s 149ms/step - loss:
0.6718 - acc: 0.5765 - val_loss: 0.6647 - val_acc: 0.5766
Epoch 9/20
125/125 [==============================] - 20s 157ms/step - loss:
0.6731 - acc: 0.5775 - val_loss: 0.6665 - val_acc: 0.5766
Epoch 10/20
125/125 [==============================] - 19s 148ms/step - loss:
0.6692 - acc: 0.5900 - val_loss: 0.6702 - val_acc: 0.5716
Epoch 11/20
125/125 [==============================] - 19s 151ms/step - loss:
0.6784 - acc: 0.5905 - val_loss: 0.6791 - val_acc: 0.5786
Epoch 12/20
125/125 [==============================] - 19s 149ms/step - loss:
0.6696 - acc: 0.6010 - val_loss: 0.6650 - val_acc: 0.5857
Epoch 13/20
125/125 [==============================] - 19s 154ms/step - loss:
0.6634 - acc: 0.5890 - val_loss: 0.7088 - val_acc: 0.5282
Epoch 14/20
125/125 [==============================] - 19s 148ms/step - loss:
0.6645 - acc: 0.6020 - val_loss: 0.6726 - val_acc: 0.5696
Epoch 15/20
```

```
125/125 [==============================] - 19s 154ms/step - loss: 0.6588 - acc: 0.5960 - val_loss: 0.6536 - val_acc: 0.6190
Epoch 16/20
125/125 [==============================] - 23s 184ms/step - loss: 0.6740 - acc: 0.5820 - val_loss: 0.7264 - val_acc: 0.5565
Epoch 17/20
125/125 [==============================] - 36s 290ms/step - loss: 0.6553 - acc: 0.6055 - val_loss: 0.6441 - val_acc: 0.6200
Epoch 18/20
125/125 [==============================] - 19s 152ms/step - loss: 0.6604 - acc: 0.6115 - val_loss: 0.6448 - val_acc: 0.6109
Epoch 19/20
125/125 [==============================] - 19s 148ms/step - loss: 0.6584 - acc: 0.6160 - val_loss: 0.6982 - val_acc: 0.5665
Epoch 20/20
125/125 [==============================] - 19s 154ms/step - loss: 0.6534 - acc: 0.6130 - val_loss: 0.6524 - val_acc: 0.6079
63/63 [==============================] - 2s 29ms/step - loss: 0.4627 - acc: 0.8070

[I 2023-10-09 04:25:54,182] Trial 2 finished with value: 0.8069999814033508 and parameters: {'learning_rate': 0.0011184113733061457, 'num_filters': 114, 'dropout_rate': 0.02419759609792682, 'batch_size': 16, 'rotation_range': 23, 'width_shift_range': 0.16926657797401556, 'height_shift_range': 0.9648470210299581, 'shear_range': 0.0019223956238895168, 'zoom_range': 0.10601085343362138, 'horizontal_flip': True}. Best is trial 0 with value: 0.8069999814033508.

Best Hyperparameters: {'learning_rate': 0.0001301956476225021, 'num_filters': 122, 'dropout_rate': 0.40332556704914646, 'batch_size': 16, 'rotation_range': 164, 'width_shift_range': 0.4711457168062444, 'height_shift_range': 0.5650464421060059, 'shear_range': 0.7091916436488243, 'zoom_range': 0.626790424124882, 'horizontal_flip': False}
```

```python
# Get the best trial and hyperparameters
best_trial = study.best_trial
# best_optimizer = best_trial.params['optimizer']
best_learning_rate = best_trial.params['learning_rate']
best_num_filters = best_trial.params['num_filters']
best_dropout_rate = best_trial.params['dropout_rate']
best_batch_size = best_trial.params['batch_size']

best_rotation_range = best_trial.params['rotation_range']
best_width_shift_range = best_trial.params['width_shift_range']
best_height_shift_range = best_trial.params['height_shift_range']
best_shear_range = best_trial.params['shear_range']
best_zoom_range = best_trial.params['zoom_range']
best_horizontal_flip = best_trial.params['horizontal_flip']
```

```python
# Define the best hyperparameters here
best_params = study.best_params
# print("Best Hyperparameters:", best_params)
print("Best Hyperparameters:")
for key, value in best_params.items():
    print(f"{key}: {value}")
```

```
Best Hyperparameters: {'learning_rate': 0.00013019564762250 21,
'num_filters': 122, 'dropout_rate': 0.40332556704914646, 'batch_size':
16, 'rotation_range': 164, 'width_shift_range': 0.4711457168062444,
'height_shift_range': 0.5650464421060059, 'shear_range':
0.7091916436488243, 'zoom_range': 0.626790424124882,
'horizontal_flip': False}
```

Build the Best model

```python
best_model = build_cnn_dog_cat_model(best_learning_rate,
best_num_filters, best_dropout_rate, best_batch_size)
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.
```

```python
model.save('cats_and_dogs_best_tuned_model_1.h5')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/
training.py:3000: UserWarning: You are saving your model as an HDF5
file via `model.save()`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')`.
  saving_api.save_model(
```

Data Augmentation with the best hyperparameters

```python
train_datagen_tuned = ImageDataGenerator(
    rescale=1./255,
    rotation_range=best_rotation_range,
    width_shift_range=best_width_shift_range,
    height_shift_range=best_height_shift_range,
    shear_range=best_shear_range,
    zoom_range=best_zoom_range,
    horizontal_flip=best_horizontal_flip)

# Note that the validation data should not be augmented!
validation_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator_tuned = train_datagen_tuned.flow_from_directory(
        # This is the target directory
```

```
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=32,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation_generator = validation_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

Train the new model

```
history = best_model.fit(
      train_generator_tuned,
      steps_per_epoch=2000//train_generator_tuned.batch_size,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=1000//validation_generator.batch_size)

Epoch 1/100
62/62 [==============================] - 18s 288ms/step - loss: 0.5962
- acc: 0.6895 - val_loss: 0.5813 - val_acc: 0.6925
Epoch 2/100
62/62 [==============================] - 18s 294ms/step - loss: 0.6073
- acc: 0.6753 - val_loss: 0.5925 - val_acc: 0.6905
Epoch 3/100
62/62 [==============================] - 29s 473ms/step - loss: 0.6067
- acc: 0.6712 - val_loss: 0.6338 - val_acc: 0.6593
Epoch 4/100
62/62 [==============================] - 18s 290ms/step - loss: 0.5956
- acc: 0.6900 - val_loss: 0.6174 - val_acc: 0.6663
Epoch 5/100
62/62 [==============================] - 18s 292ms/step - loss: 0.6100
- acc: 0.6758 - val_loss: 0.5732 - val_acc: 0.6986
Epoch 6/100
62/62 [==============================] - 18s 281ms/step - loss: 0.5973
- acc: 0.6712 - val_loss: 0.5848 - val_acc: 0.6794
```

```
Epoch 7/100
62/62 [==============================] - 19s 298ms/step - loss: 0.5981
- acc: 0.6860 - val_loss: 0.5961 - val_acc: 0.6724
Epoch 8/100
62/62 [==============================] - 19s 305ms/step - loss: 0.6131
- acc: 0.6626 - val_loss: 0.5992 - val_acc: 0.6804
Epoch 9/100
62/62 [==============================] - 18s 287ms/step - loss: 0.5988
- acc: 0.6677 - val_loss: 0.6201 - val_acc: 0.6986
Epoch 10/100
62/62 [==============================] - 26s 416ms/step - loss: 0.6218
- acc: 0.6646 - val_loss: 0.5805 - val_acc: 0.6865
Epoch 11/100
62/62 [==============================] - 25s 406ms/step - loss: 0.5925
- acc: 0.6834 - val_loss: 0.5713 - val_acc: 0.7167
Epoch 12/100
62/62 [==============================] - 23s 370ms/step - loss: 0.6016
- acc: 0.6829 - val_loss: 0.6306 - val_acc: 0.6462
Epoch 13/100
62/62 [==============================] - 23s 357ms/step - loss: 0.5933
- acc: 0.6839 - val_loss: 0.6340 - val_acc: 0.6623
Epoch 14/100
62/62 [==============================] - 30s 480ms/step - loss: 0.5941
- acc: 0.6926 - val_loss: 0.5823 - val_acc: 0.6925
Epoch 15/100
62/62 [==============================] - 23s 368ms/step - loss: 0.5990
- acc: 0.6916 - val_loss: 0.6437 - val_acc: 0.6774
Epoch 16/100
62/62 [==============================] - 18s 287ms/step - loss: 0.6075
- acc: 0.6626 - val_loss: 0.5922 - val_acc: 0.6794
Epoch 17/100
62/62 [==============================] - 18s 282ms/step - loss: 0.6001
- acc: 0.6794 - val_loss: 0.5512 - val_acc: 0.7208
Epoch 18/100
62/62 [==============================] - 20s 319ms/step - loss: 0.5981
- acc: 0.6870 - val_loss: 0.6067 - val_acc: 0.6804
Epoch 19/100
62/62 [==============================] - 17s 280ms/step - loss: 0.5968
- acc: 0.6743 - val_loss: 0.5772 - val_acc: 0.6935
Epoch 20/100
62/62 [==============================] - 17s 280ms/step - loss: 0.5898
- acc: 0.6890 - val_loss: 0.5702 - val_acc: 0.6925
Epoch 21/100
62/62 [==============================] - 19s 300ms/step - loss: 0.5877
- acc: 0.6987 - val_loss: 0.5782 - val_acc: 0.7147
Epoch 22/100
62/62 [==============================] - 18s 286ms/step - loss: 0.5999
- acc: 0.6784 - val_loss: 0.5783 - val_acc: 0.6946
Epoch 23/100
```

```
62/62 [==============================] - 19s 299ms/step - loss: 0.5953
- acc: 0.6921 - val_loss: 0.5724 - val_acc: 0.7006
Epoch 24/100
62/62 [==============================] - 18s 294ms/step - loss: 0.5989
- acc: 0.6717 - val_loss: 0.5823 - val_acc: 0.7067
Epoch 25/100
62/62 [==============================] - 32s 526ms/step - loss: 0.5910
- acc: 0.6961 - val_loss: 0.5741 - val_acc: 0.7117
Epoch 26/100
62/62 [==============================] - 19s 307ms/step - loss: 0.6004
- acc: 0.6677 - val_loss: 0.5660 - val_acc: 0.7077
Epoch 27/100
62/62 [==============================] - 26s 414ms/step - loss: 0.5929
- acc: 0.6829 - val_loss: 0.5732 - val_acc: 0.6815
Epoch 28/100
62/62 [==============================] - 18s 282ms/step - loss: 0.5842
- acc: 0.6890 - val_loss: 0.5776 - val_acc: 0.6875
Epoch 29/100
62/62 [==============================] - 25s 399ms/step - loss: 0.5892
- acc: 0.7007 - val_loss: 0.6628 - val_acc: 0.6562
Epoch 30/100
62/62 [==============================] - 26s 424ms/step - loss: 0.6015
- acc: 0.6829 - val_loss: 0.6011 - val_acc: 0.6694
Epoch 31/100
62/62 [==============================] - 24s 390ms/step - loss: 0.5858
- acc: 0.6875 - val_loss: 0.6538 - val_acc: 0.6260
Epoch 32/100
62/62 [==============================] - 20s 312ms/step - loss: 0.5995
- acc: 0.6824 - val_loss: 0.5539 - val_acc: 0.7167
Epoch 33/100
62/62 [==============================] - 18s 293ms/step - loss: 0.6031
- acc: 0.6814 - val_loss: 0.5950 - val_acc: 0.6754
Epoch 34/100
62/62 [==============================] - 20s 321ms/step - loss: 0.5874
- acc: 0.7022 - val_loss: 0.6778 - val_acc: 0.6562
Epoch 35/100
62/62 [==============================] - 18s 283ms/step - loss: 0.5965
- acc: 0.6860 - val_loss: 0.6341 - val_acc: 0.6643
Epoch 36/100
62/62 [==============================] - 22s 354ms/step - loss: 0.5964
- acc: 0.6905 - val_loss: 0.7660 - val_acc: 0.6361
Epoch 37/100
62/62 [==============================] - 21s 343ms/step - loss: 0.5988
- acc: 0.6845 - val_loss: 0.5764 - val_acc: 0.6996
Epoch 38/100
62/62 [==============================] - 28s 457ms/step - loss: 0.5980
- acc: 0.6875 - val_loss: 0.6568 - val_acc: 0.6321
Epoch 39/100
62/62 [==============================] - 24s 391ms/step - loss: 0.5926
```

```
- acc: 0.6900 - val_loss: 0.5660 - val_acc: 0.7026
Epoch 40/100
62/62 [==============================] - 24s 389ms/step - loss: 0.5840
- acc: 0.6956 - val_loss: 0.5834 - val_acc: 0.6784
Epoch 41/100
62/62 [==============================] - 19s 298ms/step - loss: 0.5771
- acc: 0.7129 - val_loss: 0.5605 - val_acc: 0.7218
Epoch 42/100
62/62 [==============================] - 18s 293ms/step - loss: 0.5798
- acc: 0.7027 - val_loss: 0.5587 - val_acc: 0.7026
Epoch 43/100
62/62 [==============================] - 25s 408ms/step - loss: 0.6013
- acc: 0.6951 - val_loss: 0.5633 - val_acc: 0.7006
Epoch 44/100
62/62 [==============================] - 17s 281ms/step - loss: 0.5846
- acc: 0.6895 - val_loss: 0.5564 - val_acc: 0.7056
Epoch 45/100
62/62 [==============================] - 19s 298ms/step - loss: 0.5923
- acc: 0.6677 - val_loss: 0.5612 - val_acc: 0.7157
Epoch 46/100
62/62 [==============================] - 19s 299ms/step - loss: 0.5857
- acc: 0.6961 - val_loss: 0.5413 - val_acc: 0.7157
Epoch 47/100
62/62 [==============================] - 31s 495ms/step - loss: 0.5863
- acc: 0.6935 - val_loss: 0.5539 - val_acc: 0.7127
Epoch 48/100
62/62 [==============================] - 24s 384ms/step - loss: 0.6061
- acc: 0.6728 - val_loss: 0.5747 - val_acc: 0.6946
Epoch 49/100
62/62 [==============================] - 19s 289ms/step - loss: 0.5864
- acc: 0.6946 - val_loss: 0.5728 - val_acc: 0.7097
Epoch 50/100
62/62 [==============================] - 19s 299ms/step - loss: 0.5968
- acc: 0.6702 - val_loss: 0.5889 - val_acc: 0.6794
Epoch 51/100
62/62 [==============================] - 22s 360ms/step - loss: 0.6068
- acc: 0.6895 - val_loss: 0.5590 - val_acc: 0.7198
Epoch 52/100
62/62 [==============================] - 18s 282ms/step - loss: 0.6113
- acc: 0.6885 - val_loss: 0.6854 - val_acc: 0.5877
Epoch 53/100
62/62 [==============================] - 18s 288ms/step - loss: 0.5964
- acc: 0.6829 - val_loss: 0.5356 - val_acc: 0.7228
Epoch 54/100
62/62 [==============================] - 19s 309ms/step - loss: 0.6039
- acc: 0.6895 - val_loss: 0.5420 - val_acc: 0.7329
Epoch 55/100
62/62 [==============================] - 19s 300ms/step - loss: 0.5884
- acc: 0.6839 - val_loss: 0.5965 - val_acc: 0.6804
```

```
Epoch 56/100
62/62 [==============================] - 18s 284ms/step - loss: 0.5753
- acc: 0.7068 - val_loss: 0.6905 - val_acc: 0.6603
Epoch 57/100
62/62 [==============================] - 19s 298ms/step - loss: 0.5820
- acc: 0.7022 - val_loss: 0.6354 - val_acc: 0.6421
Epoch 58/100
62/62 [==============================] - 18s 293ms/step - loss: 0.5823
- acc: 0.7017 - val_loss: 0.5809 - val_acc: 0.7036
Epoch 59/100
62/62 [==============================] - 18s 296ms/step - loss: 0.5794
- acc: 0.7012 - val_loss: 0.6588 - val_acc: 0.6643
Epoch 60/100
62/62 [==============================] - 18s 290ms/step - loss: 0.5978
- acc: 0.6921 - val_loss: 0.5970 - val_acc: 0.6754
Epoch 61/100
62/62 [==============================] - 19s 308ms/step - loss: 0.6062
- acc: 0.6885 - val_loss: 0.6103 - val_acc: 0.6764
Epoch 62/100
62/62 [==============================] - 18s 285ms/step - loss: 0.5849
- acc: 0.6992 - val_loss: 0.6012 - val_acc: 0.6704
Epoch 63/100
62/62 [==============================] - 18s 286ms/step - loss: 0.5988
- acc: 0.6860 - val_loss: 0.5772 - val_acc: 0.6935
Epoch 64/100
62/62 [==============================] - 18s 294ms/step - loss: 0.5906
- acc: 0.6987 - val_loss: 0.6718 - val_acc: 0.6865
Epoch 65/100
62/62 [==============================] - 18s 291ms/step - loss: 0.5918
- acc: 0.6916 - val_loss: 0.5505 - val_acc: 0.7379
Epoch 66/100
62/62 [==============================] - 19s 299ms/step - loss: 0.5865
- acc: 0.6738 - val_loss: 0.5678 - val_acc: 0.6996
Epoch 67/100
62/62 [==============================] - 18s 289ms/step - loss: 0.5906
- acc: 0.6905 - val_loss: 0.5472 - val_acc: 0.7087
Epoch 68/100
62/62 [==============================] - 19s 307ms/step - loss: 0.6000
- acc: 0.6809 - val_loss: 0.5600 - val_acc: 0.7218
Epoch 69/100
62/62 [==============================] - 18s 289ms/step - loss: 0.5905
- acc: 0.6900 - val_loss: 0.6024 - val_acc: 0.6895
Epoch 70/100
62/62 [==============================] - 19s 299ms/step - loss: 0.5746
- acc: 0.6880 - val_loss: 0.6068 - val_acc: 0.7026
Epoch 71/100
62/62 [==============================] - 18s 286ms/step - loss: 0.5850
- acc: 0.7033 - val_loss: 0.6070 - val_acc: 0.6935
Epoch 72/100
```

```
62/62 [==============================] - 22s 349ms/step - loss: 0.5948
- acc: 0.6855 - val_loss: 0.5530 - val_acc: 0.7026
Epoch 73/100
62/62 [==============================] - 18s 287ms/step - loss: 0.5890
- acc: 0.6982 - val_loss: 0.6193 - val_acc: 0.6865
Epoch 74/100
62/62 [==============================] - 19s 302ms/step - loss: 0.5848
- acc: 0.7033 - val_loss: 0.6439 - val_acc: 0.6583
Epoch 75/100
62/62 [==============================] - 18s 285ms/step - loss: 0.5830
- acc: 0.6916 - val_loss: 0.5703 - val_acc: 0.6925
Epoch 76/100
62/62 [==============================] - 18s 297ms/step - loss: 0.5797
- acc: 0.6931 - val_loss: 0.5415 - val_acc: 0.7319
Epoch 77/100
62/62 [==============================] - 18s 285ms/step - loss: 0.5837
- acc: 0.7033 - val_loss: 0.5478 - val_acc: 0.7097
Epoch 78/100
62/62 [==============================] - 18s 284ms/step - loss: 0.5845
- acc: 0.6870 - val_loss: 0.5792 - val_acc: 0.7127
Epoch 79/100
62/62 [==============================] - 20s 329ms/step - loss: 0.5994
- acc: 0.6845 - val_loss: 0.6552 - val_acc: 0.6008
Epoch 80/100
62/62 [==============================] - 18s 287ms/step - loss: 0.5974
- acc: 0.6804 - val_loss: 0.5716 - val_acc: 0.6734
Epoch 81/100
62/62 [==============================] - 19s 300ms/step - loss: 0.5737
- acc: 0.6972 - val_loss: 0.5385 - val_acc: 0.7349
Epoch 82/100
62/62 [==============================] - 18s 284ms/step - loss: 0.5882
- acc: 0.6916 - val_loss: 0.5450 - val_acc: 0.7157
Epoch 83/100
62/62 [==============================] - 19s 303ms/step - loss: 0.6000
- acc: 0.6921 - val_loss: 0.5509 - val_acc: 0.7097
Epoch 84/100
62/62 [==============================] - 18s 284ms/step - loss: 0.5850
- acc: 0.6977 - val_loss: 0.5660 - val_acc: 0.7278
Epoch 85/100
62/62 [==============================] - 19s 303ms/step - loss: 0.5932
- acc: 0.6911 - val_loss: 0.6039 - val_acc: 0.6603
Epoch 86/100
62/62 [==============================] - 18s 282ms/step - loss: 0.6049
- acc: 0.6885 - val_loss: 0.5413 - val_acc: 0.7238
Epoch 87/100
62/62 [==============================] - 18s 297ms/step - loss: 0.5635
- acc: 0.7104 - val_loss: 0.5524 - val_acc: 0.7137
Epoch 88/100
62/62 [==============================] - 18s 293ms/step - loss: 0.5722
```

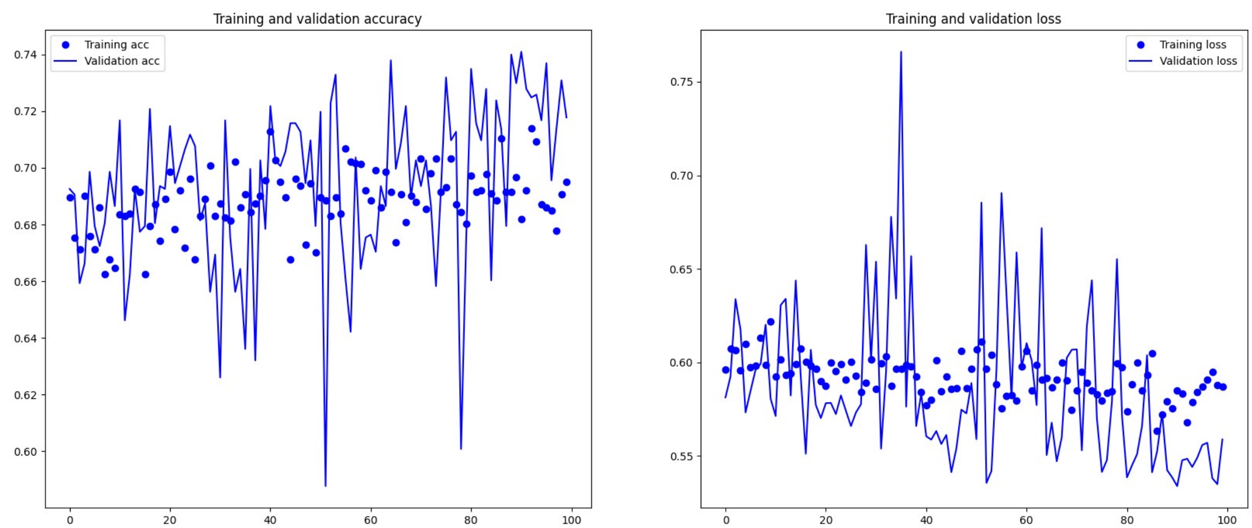
```
- acc: 0.6916 - val_loss: 0.5727 - val_acc: 0.6794
Epoch 89/100
62/62 [==============================] - 18s 283ms/step - loss: 0.5793
- acc: 0.6916 - val_loss: 0.5424 - val_acc: 0.7399
Epoch 90/100
62/62 [==============================] - 19s 304ms/step - loss: 0.5754
- acc: 0.6966 - val_loss: 0.5385 - val_acc: 0.7298
Epoch 91/100
62/62 [==============================] - 18s 284ms/step - loss: 0.5849
- acc: 0.6819 - val_loss: 0.5339 - val_acc: 0.7409
Epoch 92/100
62/62 [==============================] - 23s 366ms/step - loss: 0.5835
- acc: 0.6921 - val_loss: 0.5477 - val_acc: 0.7278
Epoch 93/100
62/62 [==============================] - 22s 348ms/step - loss: 0.5678
- acc: 0.7139 - val_loss: 0.5485 - val_acc: 0.7248
Epoch 94/100
62/62 [==============================] - 24s 389ms/step - loss: 0.5789
- acc: 0.7093 - val_loss: 0.5441 - val_acc: 0.7258
Epoch 95/100
62/62 [==============================] - 19s 301ms/step - loss: 0.5840
- acc: 0.6870 - val_loss: 0.5491 - val_acc: 0.7167
Epoch 96/100
62/62 [==============================] - 18s 282ms/step - loss: 0.5870
- acc: 0.6860 - val_loss: 0.5559 - val_acc: 0.7369
Epoch 97/100
62/62 [==============================] - 18s 295ms/step - loss: 0.5906
- acc: 0.6850 - val_loss: 0.5570 - val_acc: 0.6956
Epoch 98/100
62/62 [==============================] - 18s 282ms/step - loss: 0.5950
- acc: 0.6778 - val_loss: 0.5381 - val_acc: 0.7137
Epoch 99/100
62/62 [==============================] - 18s 288ms/step - loss: 0.5878
- acc: 0.6905 - val_loss: 0.5349 - val_acc: 0.7308
Epoch 100/100
62/62 [==============================] - 18s 283ms/step - loss: 0.5871
- acc: 0.6951 - val_loss: 0.5588 - val_acc: 0.7177
```

Plot loss and accuracy of the model

over the training and validation data during training:

```python
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
```

```
epochs = range(len(acc))

plt.figure(figsize=(20,8))

plt.subplot(1,2,1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

# plt.figure()
plt.subplot(1,2,2)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

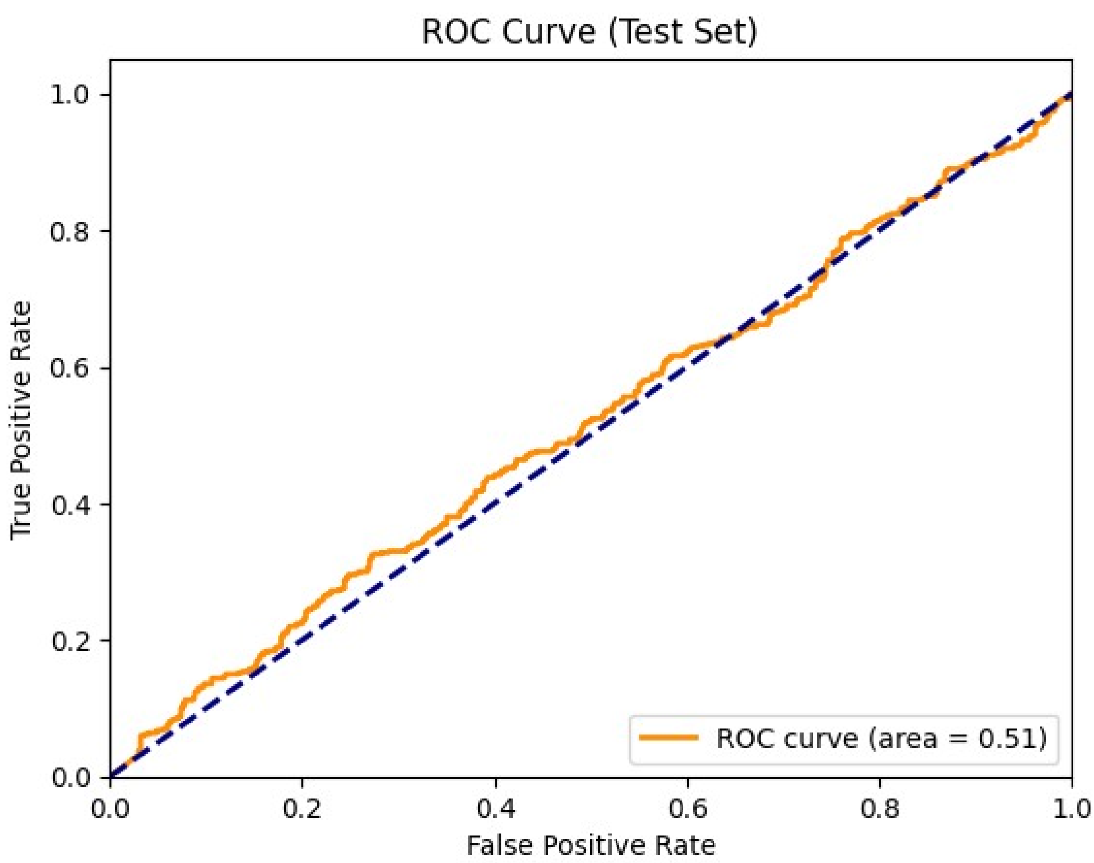


Plot the ROC curves

```
def find_labels_and_probability(img_gen):
  y_true = img_gen.classes

  # Get predicted probabilities (y_score)
  y_pred = model.predict(img_gen)

  return y_true, y_pred

test_y_true, test_probs = find_labels_and_probability(test_generator)
```

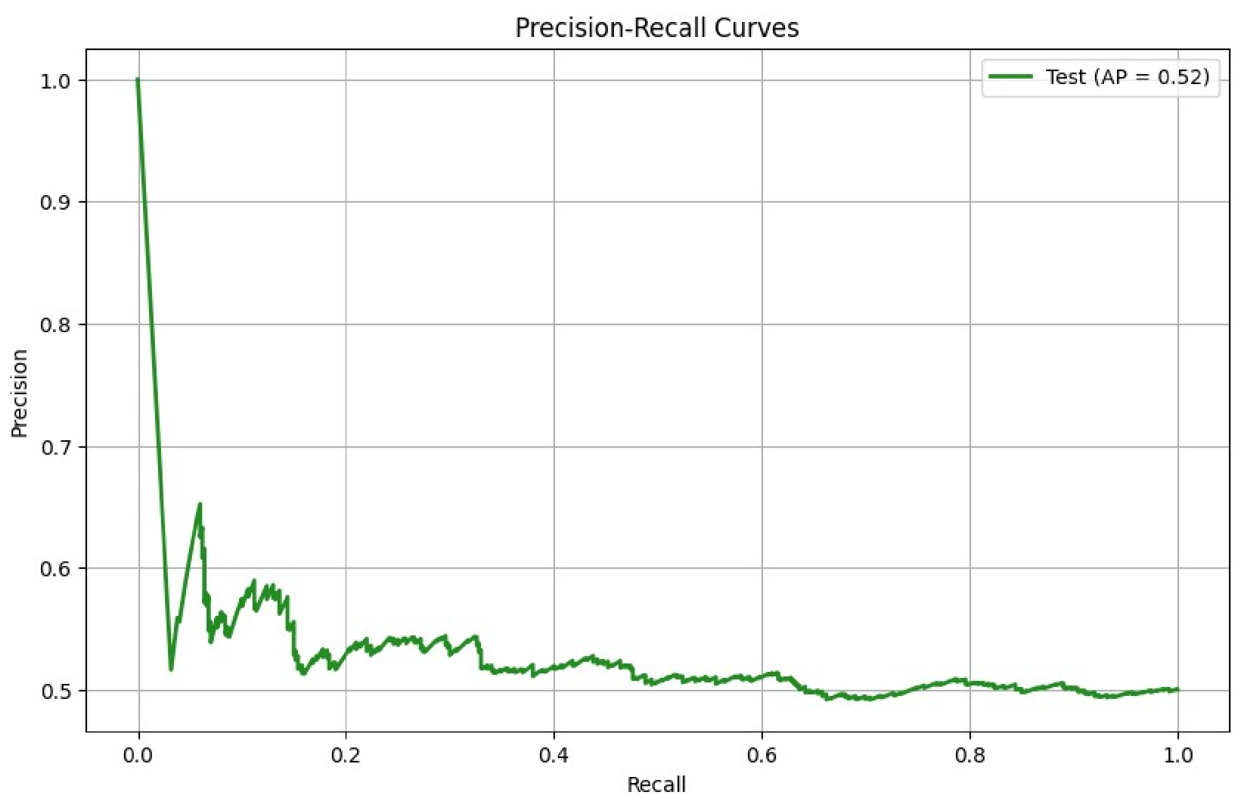
```
32/32 [==============================] - 2s 49ms/step
```

```python
# Function to plot ROC curve
def plot_roc_curve(y_true, y_score, title):
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()

# Plot ROC curves for training, validation, and test sets
plot_roc_curve(test_y_true, test_probs, title='ROC Curve (Test Set)')
```



Plot the Recall-Precision curves

```python
# Compute precision and recall for each dataset
test_precision, test_recall, threshold =
precision_recall_curve(test_y_true, test_probs)

# Compute average precision (AUC-PR) for each dataset
test_average_precision = average_precision_score(test_y_true,
test_probs)

# Plot Precision-Recall curves for all datasets
plt.figure(figsize=(10, 6))
# plt.plot(train_recall, train_precision, color='darkorange', lw=2,
label=f'Training (AP = {train_average_precision:.2f})')
# plt.plot(val_recall, val_precision, color='royalblue', lw=2,
label=f'Validation (AP = {val_average_precision:.2f})')
plt.plot(test_recall, test_precision, color='forestgreen', lw=2,
label=f'Test (AP = {test_average_precision:.2f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```

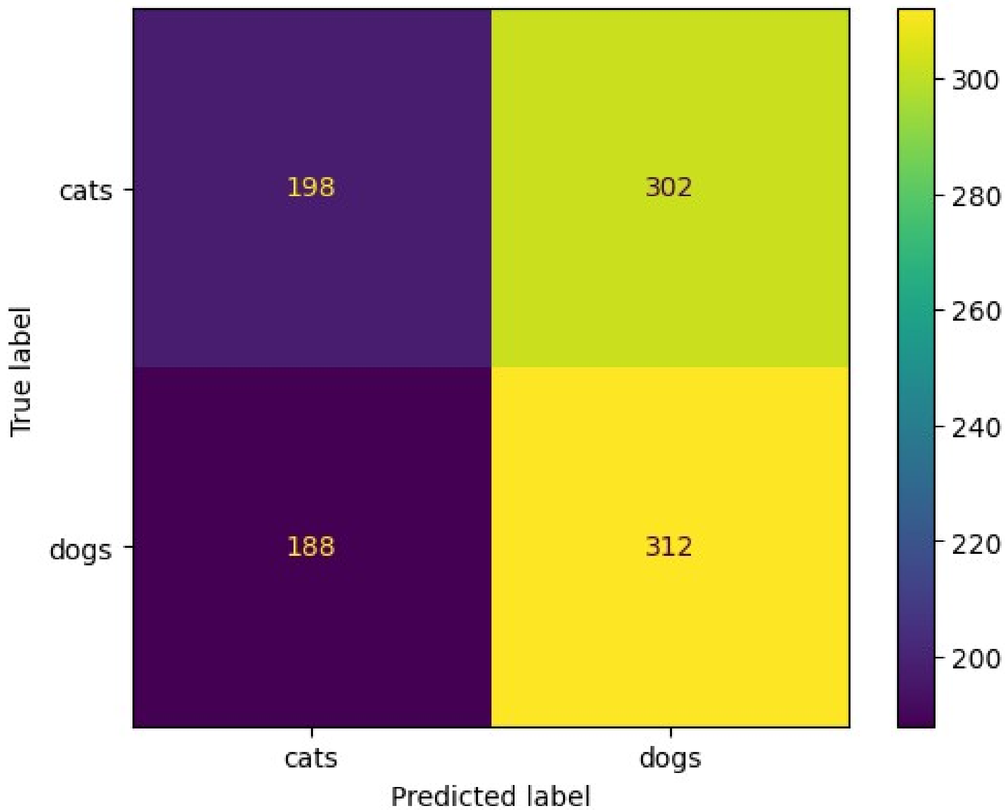


Confusion matrix for 50% threshold

```
def find_labels_and_probability(y_true, y_probs):
  threshold = 0.5
  # Convert probabilities to binary classes using the 50% threshold
  y_pred = (y_probs >= threshold).astype(int)  # Predicted classes (0 
or 1)

  # Create the confusion matrix
  conf_matrix = confusion_matrix(y_true, y_pred)

  return conf_matrix

test_conf_matrix = find_labels_and_probability(test_y_true, 
test_probs)

# Print the confusion matrix
print("Confusion Matrix Test Data (Threshold = 50%):")
print(test_conf_matrix)

# Display the confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=test_conf_matrix, 
display_labels=test_generator.class_indices)
disp.plot(cmap='viridis', values_format='d')
```

```
Confusion Matrix Test Data (Threshold = 50%):
[[198 302]
 [188 312]]
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 
0x791708771f30>
```



Data Preprocessing

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
from keras import models, layers, optimizers

# Data Augmentation -  tuned with hyperparameters
# train_datagen_tuned = ImageDataGenerator(
#     rescale=1./255,
#     rotation_range=best_rotation_range,
#     width_shift_range=best_width_shift_range,
#     height_shift_range=best_height_shift_range,
#     shear_range=best_shear_range,
#     zoom_range=best_zoom_range,
#     horizontal_flip=best_horizontal_flip)

baseline_datagen = ImageDataGenerator(
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
```

```
      horizontal_flip=True,
      fill_mode='nearest')

# Note that the validation & test data should not be augmented!
validation_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)
```

10% of the rare class (cats) and 100% of the common class (dogs).

```
# Unzip file
!mkdir -p dogscats/data
!unzip -o -q dogs-vs-cats-10-per-rare.zip -d dogscats

base_dir10 = 'dogscats/data'
train_dir10 = os.path.join(base_dir10, 'train')
train_cats_dir10 = os.path.join(base_dir10, 'train', 'cats')
train_dogs_dir10 = os.path.join(base_dir10, 'train', 'dogs')
validation_dir10 = os.path.join(base_dir10, 'validation')
test_dir10 = os.path.join(base_dir10, 'test')
```

Baseline Data Augmentation

```
best_batch_size = 32

train10_generator = baseline_datagen.flow_from_directory(
        # This is the target directory
        train_dir10,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=best_batch_size,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation10_generator = validation_datagen.flow_from_directory(
        validation_dir10,
        target_size=(150, 150),
        batch_size=best_batch_size,
        class_mode='binary')

test10_generator = test_datagen.flow_from_directory(
        test_dir10,
        target_size=(150, 150),
        batch_size=best_batch_size,
        class_mode='binary')
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

1% of the rare class (cats) and 100% of the common class (dogs).

```
# shutil.rmtree('/content/dogscats/data1')

# Unzip file
!mkdir -p dogscats/data1
!unzip -o -q dogs-vs-cats-1-per-rare.zip -d dogscats

base_dir1 = 'dogscats/data1'
train_dir1 = os.path.join(base_dir1, 'train')
train_cats_dir1 = os.path.join(base_dir1, 'train', 'cats')
train_dogs_dir1 = os.path.join(base_dir1, 'train', 'dogs')
validation_dir1 = os.path.join(base_dir1, 'validation')
test_dir1 = os.path.join(base_dir1, 'test')
```

Baseline Data Augmentation

```
best_batch_size=32

train1_generator = baseline_datagen.flow_from_directory(
        # This is the target directory
        train_dir1,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=best_batch_size,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation1_generator = validation_datagen.flow_from_directory(
        validation_dir1,
        target_size=(150, 150),
        batch_size=best_batch_size,
        class_mode='binary')

test1_generator = test_datagen.flow_from_directory(
        test_dir1,
        target_size=(150, 150),
        batch_size=best_batch_size,
        class_mode='binary')
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

Re-train the model

```
from keras import layers, losses, optimizers, metrics
from sklearn.utils import class_weight

def build_cnn_model_new_dataset():
  model = models.Sequential()
  model.add(layers.Conv2D(32, (3, 3), activation='relu',
                          input_shape=(150, 150, 3)))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(128, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(128, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Flatten())
  model.add(layers.Dropout(0.5))
  model.add(layers.Dense(512, activation='relu'))
  model.add(layers.Dense(1, activation='sigmoid'))

  model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])

  return model
```

10% rare class datasets and evaluate

Retrain and Revaluate

```
# model10 = build_cnn_dog_cat_model(learning_rate, num_filters,
dropout_rate, batch_size)
# model = model.load("/content/cats_and_dogs_best_tuned_model_1.h5")
# model = model.load("cats_and_dogs_best_tuned_model_1.h5")
model10 = build_cnn_model_new_dataset()
# model10 = build_cnn_dog_cat_model(best_learning_rate,
best_num_filters,
#                                  best_dropout_rate,
best_batch_size)

## Possibly the above might be to load the Baseline model instead of
the optuna tuned

history10 = model10.fit(
      train10_generator,
      steps_per_epoch=2000//train10_generator.batch_size,
      epochs=20,
```

```
      validation_data=validation10_generator,
      validation_steps=1000//validation10_generator.batch_size)
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.
```

```
Epoch 1/20
62/62 [==============================] - 70s 1s/step - loss: 5.6279 -
acc: 0.8765 - val_loss: 0.6890 - val_acc: 0.8992
Epoch 2/20
62/62 [==============================] - 69s 1s/step - loss: 0.4011 -
acc: 0.9014 - val_loss: 0.6781 - val_acc: 0.8992
Epoch 3/20
62/62 [==============================] - 68s 1s/step - loss: 0.4493 -
acc: 0.9004 - val_loss: 0.6689 - val_acc: 0.8992
Epoch 4/20
62/62 [==============================] - 68s 1s/step - loss: 0.3600 -
acc: 0.8994 - val_loss: 0.6619 - val_acc: 0.8992
Epoch 5/20
62/62 [==============================] - 68s 1s/step - loss: 0.3825 -
acc: 0.8999 - val_loss: 0.6599 - val_acc: 0.9002
Epoch 6/20
62/62 [==============================] - 68s 1s/step - loss: 0.3790 -
acc: 0.8994 - val_loss: 0.6487 - val_acc: 0.9002
Epoch 7/20
62/62 [==============================] - 69s 1s/step - loss: 0.3598 -
acc: 0.9009 - val_loss: 0.6378 - val_acc: 0.8992
Epoch 8/20
62/62 [==============================] - 80s 1s/step - loss: 0.3468 -
acc: 0.8994 - val_loss: 0.6284 - val_acc: 0.9012
Epoch 9/20
62/62 [==============================] - 70s 1s/step - loss: 0.3602 -
acc: 0.8989 - val_loss: 0.6266 - val_acc: 0.8992
Epoch 10/20
62/62 [==============================] - 70s 1s/step - loss: 0.3376 -
acc: 0.9009 - val_loss: 0.6159 - val_acc: 0.9012
Epoch 11/20
62/62 [==============================] - 70s 1s/step - loss: 0.3456 -
acc: 0.8999 - val_loss: 0.6214 - val_acc: 0.8992
Epoch 12/20
62/62 [==============================] - 68s 1s/step - loss: 0.3325 -
acc: 0.8999 - val_loss: 0.6253 - val_acc: 0.8992
Epoch 13/20
62/62 [==============================] - 69s 1s/step - loss: 0.3556 -
acc: 0.8999 - val_loss: 0.6288 - val_acc: 0.9002
Epoch 14/20
62/62 [==============================] - 70s 1s/step - loss: 0.3434 -
acc: 0.8999 - val_loss: 0.6239 - val_acc: 0.9002
Epoch 15/20
```

```
62/62 [==============================] - 70s 1s/step - loss: 0.3602 -
acc: 0.8989 - val_loss: 0.6203 - val_acc: 0.8992
Epoch 16/20
62/62 [==============================] - 70s 1s/step - loss: 0.3419 -
acc: 0.8999 - val_loss: 0.6165 - val_acc: 0.8992
Epoch 17/20
62/62 [==============================] - 70s 1s/step - loss: 0.3499 -
acc: 0.8994 - val_loss: 0.6184 - val_acc: 0.9002
Epoch 18/20
62/62 [==============================] - 71s 1s/step - loss: 0.3252 -
acc: 0.8999 - val_loss: 0.6200 - val_acc: 0.8992
Epoch 19/20
62/62 [==============================] - 69s 1s/step - loss: 0.3444 -
acc: 0.8984 - val_loss: 0.6146 - val_acc: 0.9002
Epoch 20/20
62/62 [==============================] - 69s 1s/step - loss: 0.3299 -
acc: 0.8994 - val_loss: 0.6133 - val_acc: 0.9002

----------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
<ipython-input-19-d5e7030dd8d4> in <cell line: 18>()
     16
     17 # Evaluate the model on the validation set
---> 18 val_loss, val_acc = model.evaluate(validation10_generator,
steps=len(validation10_generator))

NameError: name 'model' is not defined
```

```
# Evaluate the model on the validation set
val_loss, val_acc = model10.evaluate(validation10_generator,
steps=len(validation10_generator))
```

```
32/32 [==============================] - 10s 296ms/step - loss: 0.6134
- acc: 0.9000
```

Accuracy on validation data is obtained 90% with baseline

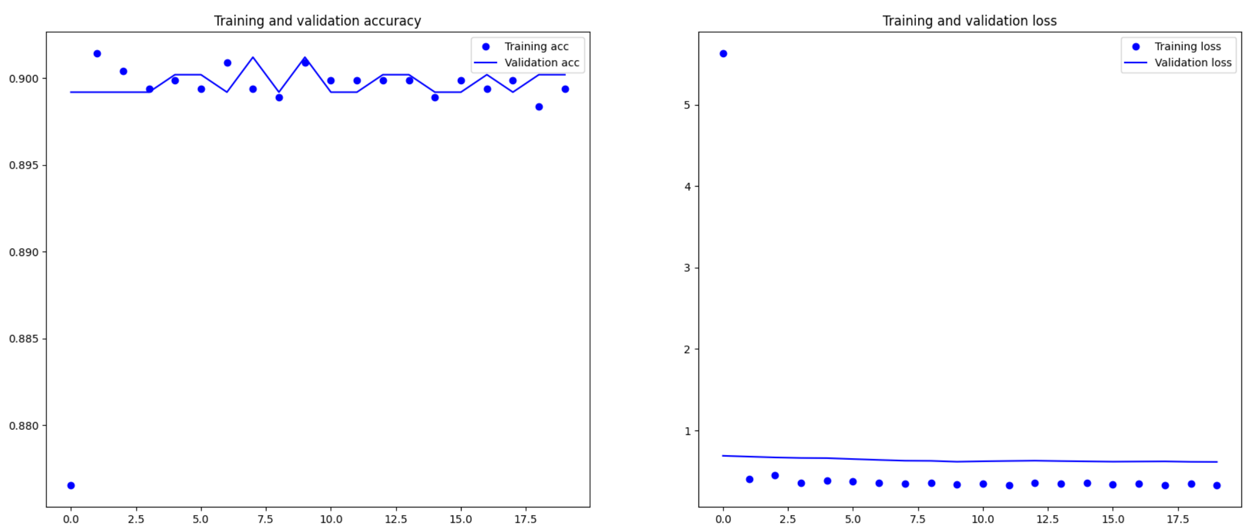
```
# model.save('cats_and_dogs_small10_1.h5')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/
training.py:3000: UserWarning: You are saving your model as an HDF5
file via `model.save()`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')`.
  saving_api.save_model(
```

Plot the loss and accuracy

of the model over the training and validation data during training:

```
import matplotlib.pyplot as plt

acc = history10.history['acc']
val_acc = history10.history['val_acc']
loss = history10.history['loss']
val_loss = history10.history['val_loss']

epochs = range(len(acc))

plt.figure(figsize=(20,8))

plt.subplot(1,2,1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

# plt.figure()
plt.subplot(1,2,2)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```



These plots are characteristic of overfitting. Our training accuracy increases linearly over time, until it reaches nearly 100%, while our validation accuracy stalls at 70-72%. Our validation loss reaches its minimum after only five epochs then stalls, while the training loss keeps decreasing linearly until it reaches nearly 0.

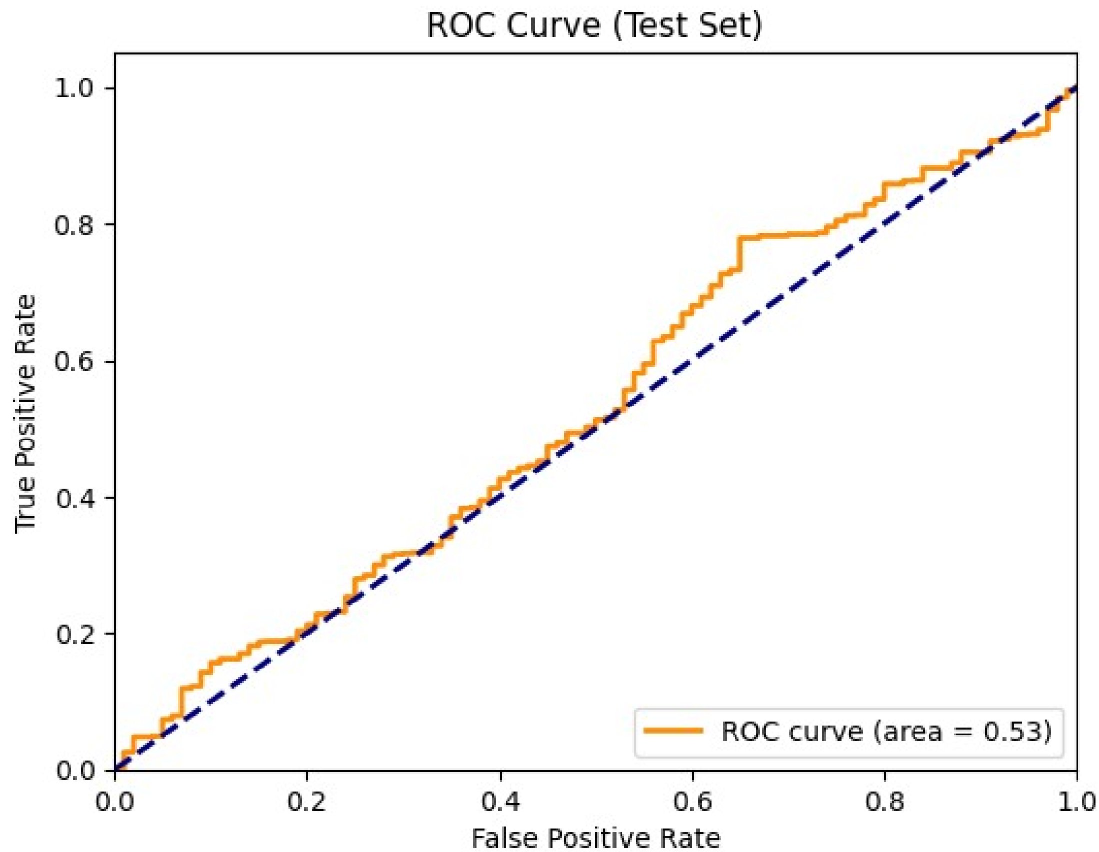
Because we only have relatively few training samples (2000), overfitting is going to be our number one concern. You already know about a number of techniques that can help mitigate overfitting, such as dropout and weight decay (L2 regularization). We are now going to introduce a new one, specific to computer vision, and used almost universally when processing images with deep learning models: *data augmentation*.

Plot the ROC curves

```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

def find_labels_and_probability(img_gen):
  y_true = img_gen.classes

  # Get predicted probabilities (y_score)
  y_pred = model10.predict(img_gen)

  return y_true, y_pred

test_y_true, test_probs =
find_labels_and_probability(test10_generator)
```

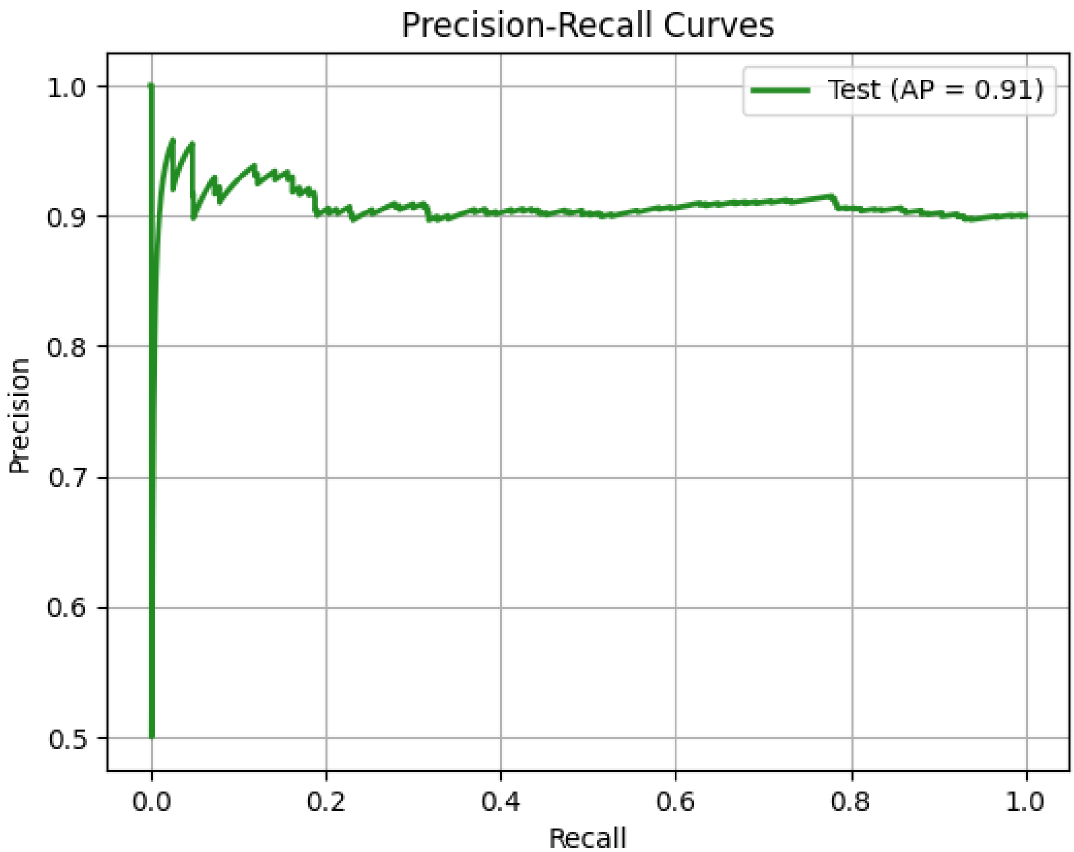
```
32/32 [==============================] - 9s 290ms/step
```

```
# Function to plot ROC curve
def plot_roc_curve(y_true, y_score, title):
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()

# Plot ROC curves for test sets
plot_roc_curve(test_y_true, test_probs, title='ROC Curve (Test Set)')
```



Plot the Recall-Precision curves

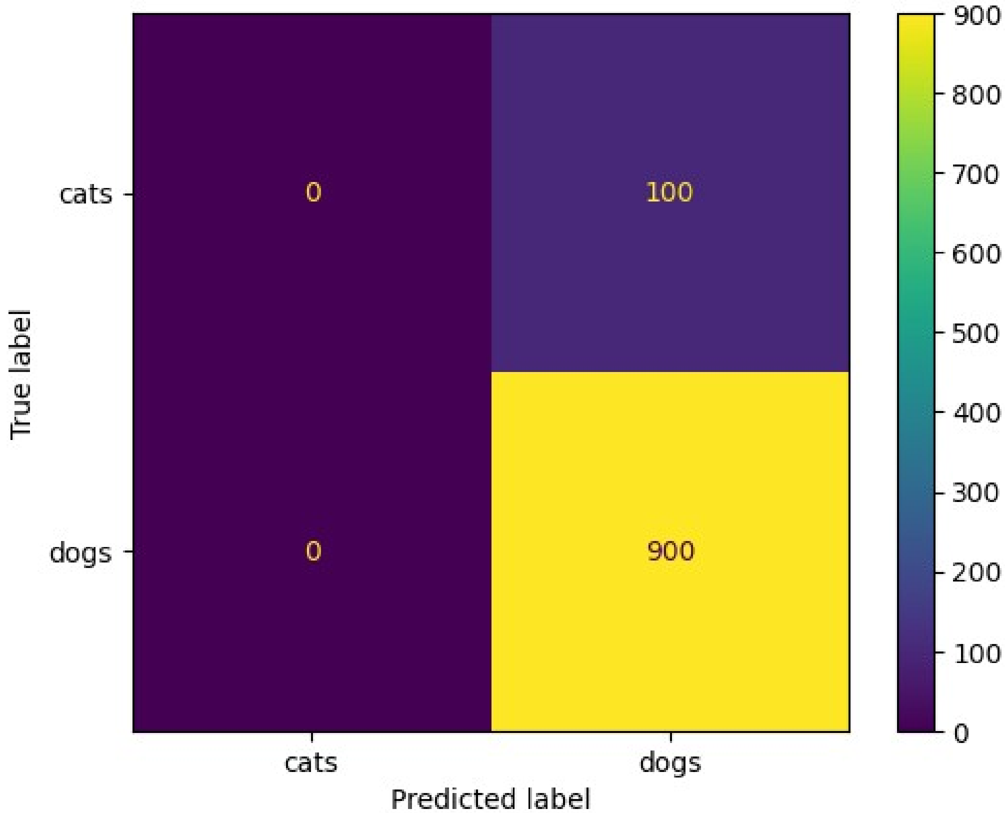
```
from sklearn.metrics import precision_recall_curve,
average_precision_score

# Compute precision and recall for each dataset
test_precision, test_recall, threshold =
precision_recall_curve(test_y_true, test_probs)

# Compute average precision (AUC-PR) for each dataset
test_average_precision = average_precision_score(test_y_true,
test_probs)

# Plot Precision-Recall curves for all datasets
# plt.figure(figsize=(10, 6))
plt.figure()
plt.plot(test_recall, test_precision, color='forestgreen', lw=2,
label=f'Test (AP = {test_average_precision:.2f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



Confusion matrix for 50% threshold

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def find_labels_and_probability(y_true, y_probs):
  threshold = 0.5
  # Convert probabilities to binary classes using the 50% threshold
  y_pred = (y_probs >= threshold).astype(int)  # Predicted classes (0
or 1)

  # Create the confusion matrix
  conf_matrix = confusion_matrix(y_true, y_pred)

  return conf_matrix

# val_conf_matrix = find_labels_and_probability(val_y_true, val_probs)
test_conf_matrix = find_labels_and_probability(test_y_true,
test_probs)

# Print the confusion matrix
print("Confusion Matrix Test Data (Threshold = 50%):")
print(test_conf_matrix)

# Display the confusion matrix as a heatmap
```

```
disp = ConfusionMatrixDisplay(confusion_matrix=test_conf_matrix,
display_labels=test10_generator.class_indices)
disp.plot(cmap='viridis', values_format='d')

# print("Confusion Matrix Validation Data (Threshold = 50%):")
# print(val_conf_matrix)

Confusion Matrix Test Data (Threshold = 50%):
[[  0 100]
 [  0 900]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7e6bb468c070>
```



Comment on what is the performance of the 10% rare class

1% rare class datasets and evaluate

Retrain and reevaluate

```
# model1 = build_cnn_dog_cat_model(learning_rate, num_filters,
dropout_rate, batch_size)
# model1 = model.load("/content/cats_and_dogs_best_tuned_model_1.h5")
# model1 = build_cnn_CSL_model(best_learning_rate, best_num_filters,
```

```
#                                 best_dropout_rate, best_batch_size,
class_weights)
model1 = build_cnn_model_new_dataset()

history1 = model1.fit(
      train1_generator,
      steps_per_epoch=2000//train1_generator.batch_size,
      epochs=20,
      validation_data=validation1_generator,
      validation_steps=1000//validation1_generator.batch_size)

# Evaluate the model on the validation set
val_loss, val_acc = model1.evaluate(validation1_generator,
steps=len(validation1_generator))
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.

Epoch 1/20
62/62 [==============================] - 84s 1s/step - loss: 3.3400 -
acc: 0.9690 - val_loss: 0.6860 - val_acc: 0.9899
Epoch 2/20
62/62 [==============================] - 71s 1s/step - loss: 0.1354 -
acc: 0.9903 - val_loss: 0.6615 - val_acc: 0.9899
Epoch 3/20
62/62 [==============================] - 72s 1s/step - loss: 0.1977 -
acc: 0.9898 - val_loss: 0.6524 - val_acc: 0.9899
Epoch 4/20
62/62 [==============================] - 70s 1s/step - loss: 0.1130 -
acc: 0.9898 - val_loss: 0.6553 - val_acc: 0.9899
Epoch 5/20
62/62 [==============================] - 70s 1s/step - loss: 0.1147 -
acc: 0.9898 - val_loss: 0.6198 - val_acc: 0.9909
Epoch 6/20
62/62 [==============================] - 71s 1s/step - loss: 0.0933 -
acc: 0.9898 - val_loss: 0.5864 - val_acc: 0.9899
Epoch 7/20
62/62 [==============================] - 69s 1s/step - loss: 0.0871 -
acc: 0.9903 - val_loss: 0.6048 - val_acc: 0.9899
Epoch 8/20
62/62 [==============================] - 69s 1s/step - loss: 0.0912 -
acc: 0.9903 - val_loss: 0.5467 - val_acc: 0.9899
Epoch 9/20
62/62 [==============================] - 71s 1s/step - loss: 0.1132 -
acc: 0.9898 - val_loss: 0.6107 - val_acc: 0.9909
Epoch 10/20
62/62 [==============================] - 70s 1s/step - loss: 0.0862 -
acc: 0.9898 - val_loss: 0.5531 - val_acc: 0.9899
Epoch 11/20
```

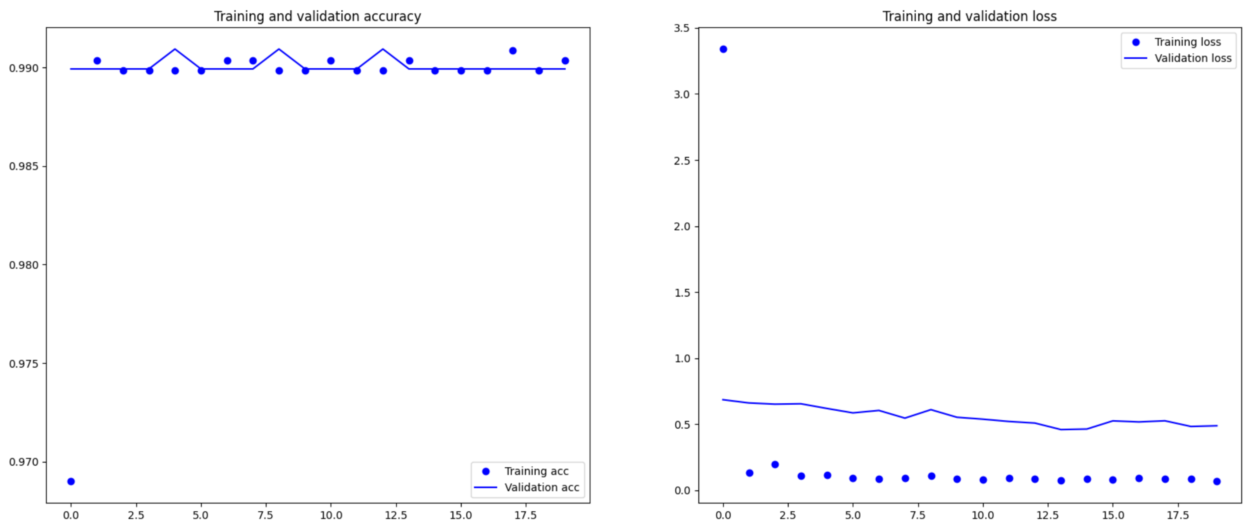
```
62/62 [==============================] - 69s 1s/step - loss: 0.0814 -
acc: 0.9903 - val_loss: 0.5387 - val_acc: 0.9899
Epoch 12/20
62/62 [==============================] - 71s 1s/step - loss: 0.0930 -
acc: 0.9898 - val_loss: 0.5213 - val_acc: 0.9899
Epoch 13/20
62/62 [==============================] - 71s 1s/step - loss: 0.0871 -
acc: 0.9898 - val_loss: 0.5093 - val_acc: 0.9909
Epoch 14/20
62/62 [==============================] - 71s 1s/step - loss: 0.0762 -
acc: 0.9903 - val_loss: 0.4601 - val_acc: 0.9899
Epoch 15/20
62/62 [==============================] - 69s 1s/step - loss: 0.0873 -
acc: 0.9898 - val_loss: 0.4641 - val_acc: 0.9899
Epoch 16/20
62/62 [==============================] - 71s 1s/step - loss: 0.0810 -
acc: 0.9898 - val_loss: 0.5260 - val_acc: 0.9899
Epoch 17/20
62/62 [==============================] - 69s 1s/step - loss: 0.0923 -
acc: 0.9898 - val_loss: 0.5178 - val_acc: 0.9899
Epoch 18/20
62/62 [==============================] - 70s 1s/step - loss: 0.0862 -
acc: 0.9909 - val_loss: 0.5265 - val_acc: 0.9899
Epoch 19/20
62/62 [==============================] - 69s 1s/step - loss: 0.0898 -
acc: 0.9898 - val_loss: 0.4836 - val_acc: 0.9899
Epoch 20/20
62/62 [==============================] - 69s 1s/step - loss: 0.0682 -
acc: 0.9903 - val_loss: 0.4888 - val_acc: 0.9899
32/32 [==============================] - 9s 270ms/step - loss: 0.4888
- acc: 0.9900
```

Plot the loss and accuracy

of the model over the training and validation data during training:

```
import matplotlib.pyplot as plt

acc = history1.history['acc']
val_acc = history1.history['val_acc']
loss = history1.history['loss']
val_loss = history1.history['val_loss']

epochs = range(len(acc))

plt.figure(figsize=(20,8))

plt.subplot(1,2,1)
plt.plot(epochs, acc, 'bo', label='Training acc')
```

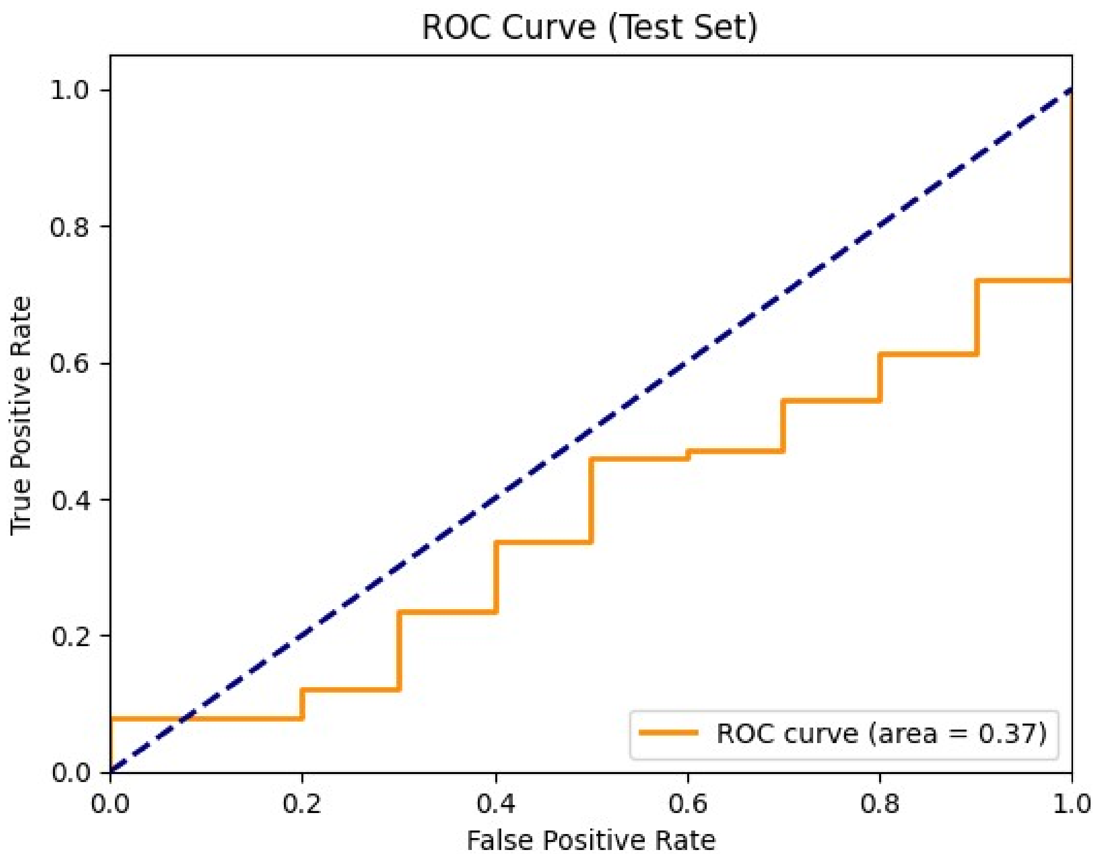
```
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

# plt.figure()
plt.subplot(1,2,2)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```



Plot the ROC curves

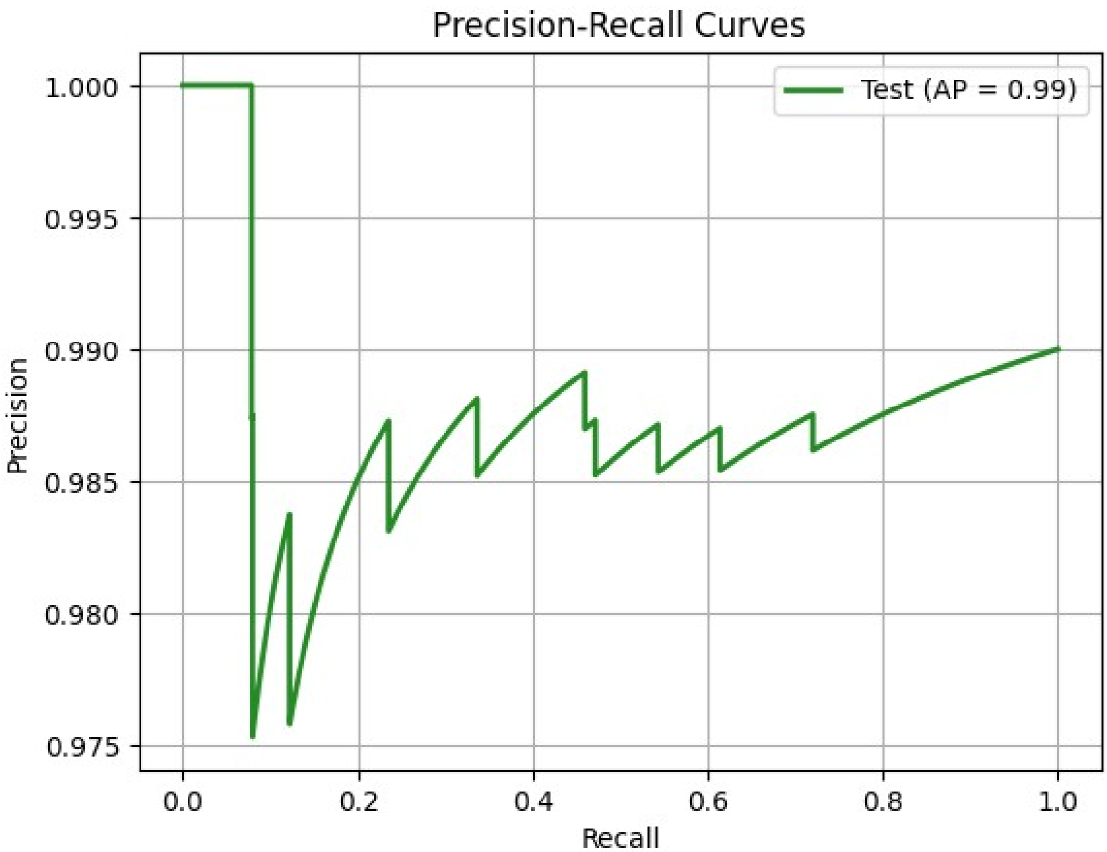
```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

def find_labels_and_probability(img_gen):
  y_true = img_gen.classes

  # Get predicted probabilities (y_score)
  y_pred = model1.predict(img_gen)

  return y_true, y_pred

test_y_true, test_probs = find_labels_and_probability(test1_generator)

32/32 [==============================] - 9s 268ms/step

# Function to plot ROC curve
def plot_roc_curve(y_true, y_score, title):
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)
```

```
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()

# Plot ROC curves for training, validation, and test sets
# plot_roc_curve(train_y_true, train_probs, title='ROC Curve (Training
Set)')
# plot_roc_curve(val_y_true, val_probs, title='ROC Curve (Validation
Set)')
plot_roc_curve(test_y_true, test_probs, title='ROC Curve (Test Set)')
```



Plot the Recall-Precision curves

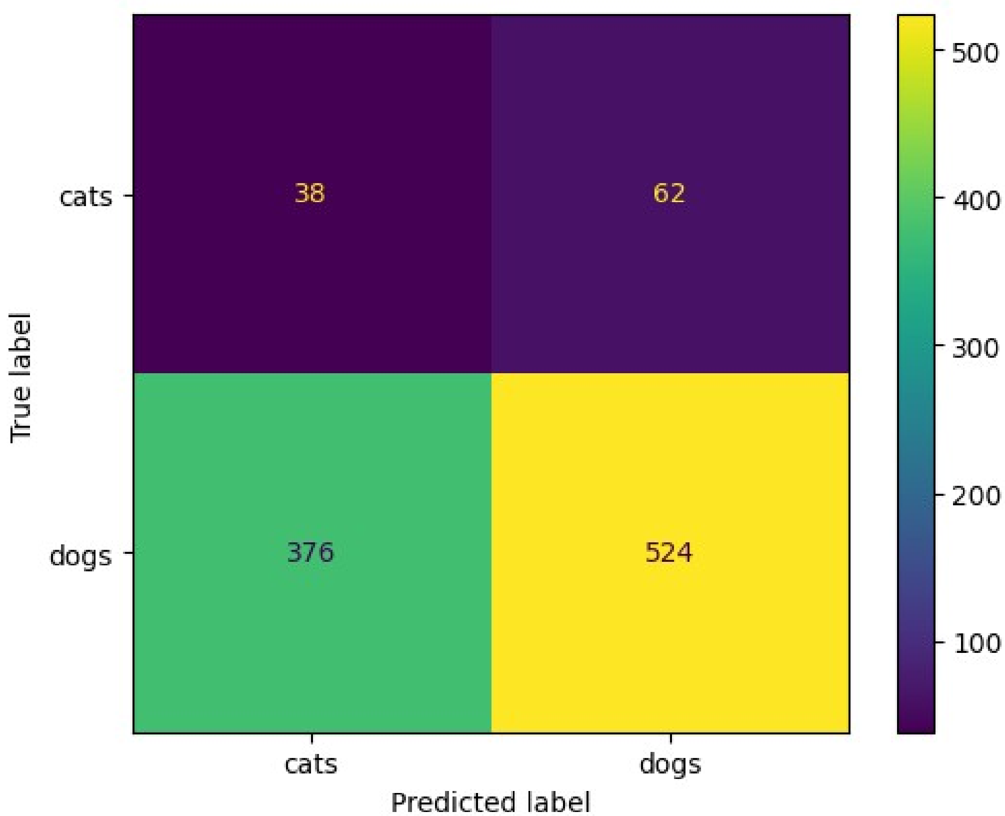
```
from sklearn.metrics import precision_recall_curve,
average_precision_score

# Compute precision and recall for each dataset
# train_precision, train_recall, threshold =
precision_recall_curve(train_y_true, train_probs)
# val_precision, val_recall, threshold =
precision_recall_curve(val_y_true, val_probs)
test_precision, test_recall, threshold =
precision_recall_curve(test_y_true, test_probs)

# Compute average precision (AUC-PR) for each dataset
# train_average_precision = average_precision_score(train_y_true,
train_probs)
# val_average_precision = average_precision_score(val_y_true,
val_probs)
test_average_precision = average_precision_score(test_y_true,
test_probs)

# Plot Precision-Recall curves for all datasets
# plt.figure(figsize=(10, 6))
# plt.plot(train_recall, train_precision, color='darkorange', lw=2,
label=f'Training (AP = {train_average_precision:.2f})')
# plt.plot(val_recall, val_precision, color='royalblue', lw=2,
label=f'Validation (AP = {val_average_precision:.2f})')
plt.figure()
plt.plot(test_recall, test_precision, color='forestgreen', lw=2,
label=f'Test (AP = {test_average_precision:.2f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



Confusion matrix for 50% threshold

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def find_labels_and_probability(y_true, y_probs):
  threshold = 0.99
  # Convert probabilities to binary classes using the 50% threshold
  y_pred = (y_probs >= threshold).astype(int)  # Predicted classes (0
or 1)

  # Create the confusion matrix
  conf_matrix = confusion_matrix(y_true, y_pred)

  return conf_matrix

# val_conf_matrix = find_labels_and_probability(val_y_true, val_probs)
test_conf_matrix = find_labels_and_probability(test_y_true,
test_probs)

# Print the confusion matrix
print("Confusion Matrix Test Data (Threshold = 50%):")
print(test_conf_matrix)

# Display the confusion matrix as a heatmap
```

```
disp = ConfusionMatrixDisplay(confusion_matrix=test_conf_matrix,
display_labels=test1_generator.class_indices)
disp.plot(cmap='viridis', values_format='d')

# print("Confusion Matrix Validation Data (Threshold = 50%):")
# print(val_conf_matrix)

Confusion Matrix Test Data (Threshold = 50%):
[[ 38  62]
 [376 524]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7e6bb49cfd60>
```



Comment on what is the performance of the 1% rare class

Improve

[NOTE]: As discussed the following model is taking too much time so , training on the previous simpler model for faster results

```
# def build_cnn_CSL_model(learning_rate, num_filters,
#                              dropout_rate, batch_size,
class_weights):
#   model = keras.Sequential()

#   model.add(layers.Conv2D(num_filters, (3, 3), activation='relu',
input_shape=(150, 150, 3)))
#   model.add(layers.MaxPooling2D((2, 2)))
#   model.add(layers.Conv2D(num_filters * 2, (3, 3),
activation='relu'))
#   model.add(layers.MaxPooling2D((2, 2)))
#   model.add(layers.Conv2D(num_filters * 4, (3, 3),
activation='relu'))
#   model.add(layers.MaxPooling2D((2, 2)))
#   model.add(layers.Conv2D(num_filters * 4, (3, 3),
activation='relu'))
#   model.add(layers.MaxPooling2D((2, 2)))
#   model.add(layers.Flatten())
#   model.add(layers.Dropout(dropout_rate))
#   model.add(layers.Dense(512, activation='relu'))
#   model.add(layers.Dense(1, activation='sigmoid'))

#   # Define your custom loss function with class-sensitive re-
weighting
#   def class_sensitive_loss(class_weights):
#       def loss(y_true, y_pred):
#           weighted_loss =
tf.keras.losses.binary_crossentropy(test_y_true, test_y_pred,
#
from_logits=False) * class_weights
#           return tf.reduce_mean(weighted_loss)

#       return loss

#   # Compile the model with your custom loss function
#   model.compile(loss=class_sensitive_loss(class_weights),
#                 optimizer=optimizers.RMSprop(lr=learning_rate),
#                 metrics=['acc'])

#   return model
```

10% rare class dataset

Using DataAugmentation

Tweaking the augmentation values further and adding more settings to strike a balance between the 2 datasets

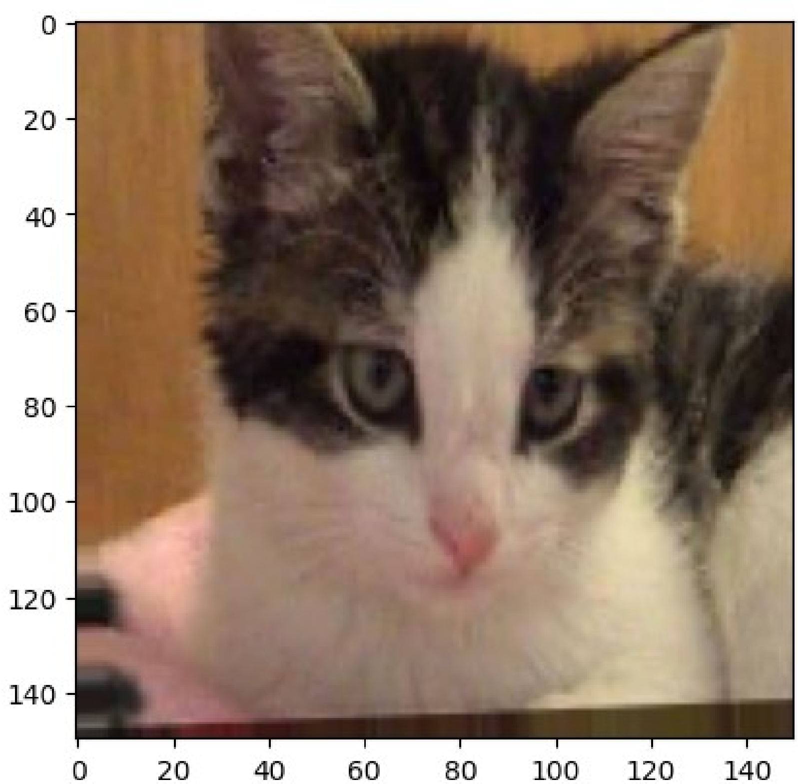
```
moderate_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=20,  # Reduce rotation range
    shear_range=0.1,  # Reduce shear range
    horizontal_flip=True,
    fill_mode='nearest',
    brightness_range=[0.8, 1.2],  # Adjust brightness levels
moderately
    channel_shift_range=0.1,  # Apply slight color channel shifts
    # contrast_stretching_range=[0.9, 1.1],  # Adjust contrast
moderately
    vertical_flip=True,  # Flip vertically
    height_shift_range=0.05,  # Reduce height shift range further
    width_shift_range=0.05,  # Reduce width shift range further
    zoom_range=[0.95, 1.05],  # Reduce zoom range further
)

train2_generator = moderate_datagen.flow_from_directory(
        # This is the target directory
        train_dir10,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=30,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation2_generator = validation_datagen.flow_from_directory(
        validation_dir10,
        target_size=(150, 150),
        batch_size=30,
        class_mode='binary')

test2_generator = test_datagen.flow_from_directory(
        test_dir10,
        target_size=(150, 150),
        batch_size=30,
        class_mode='binary')
```

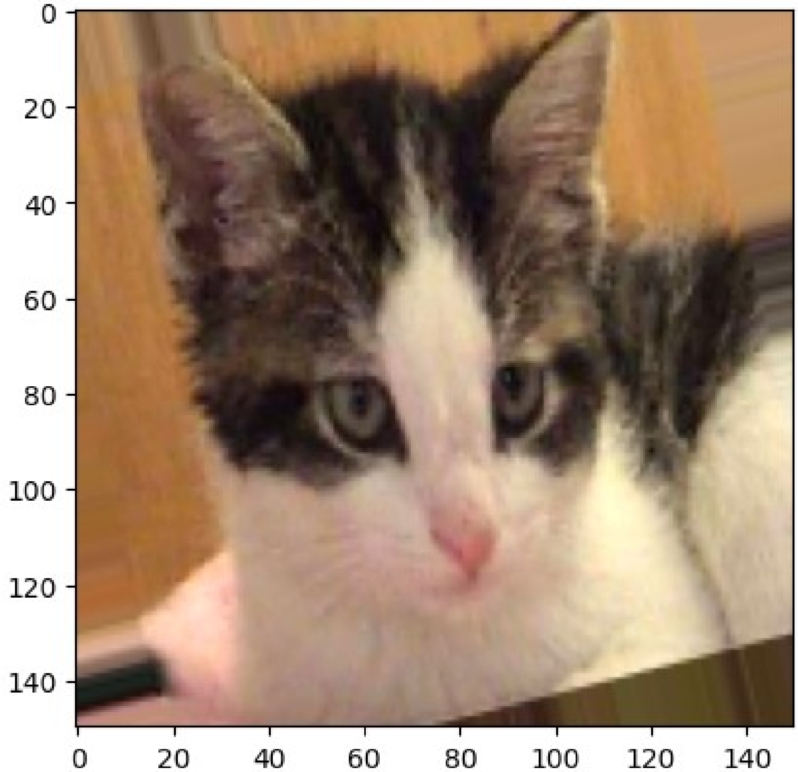
```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

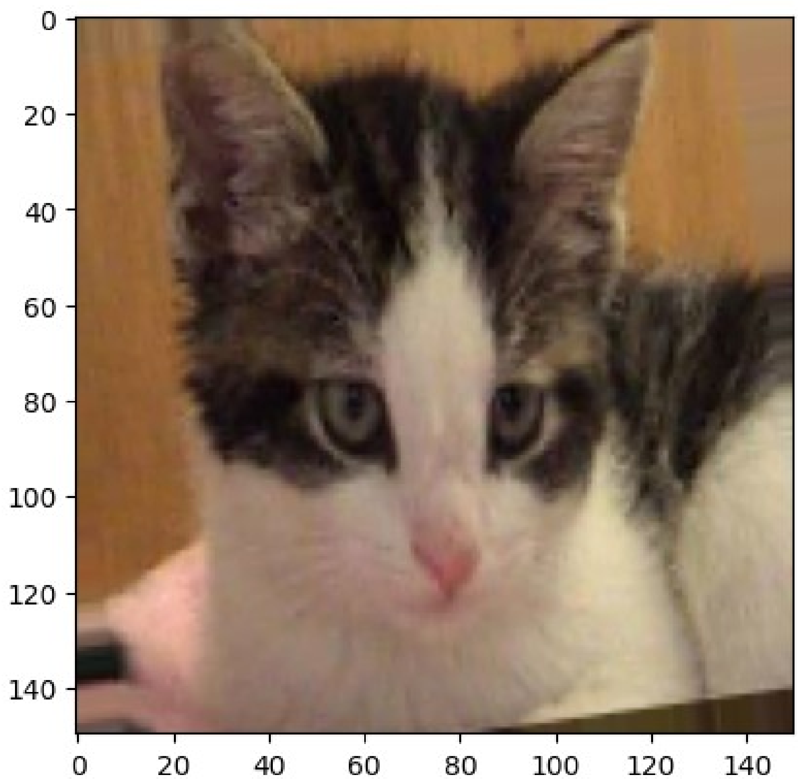
look at our augmented images

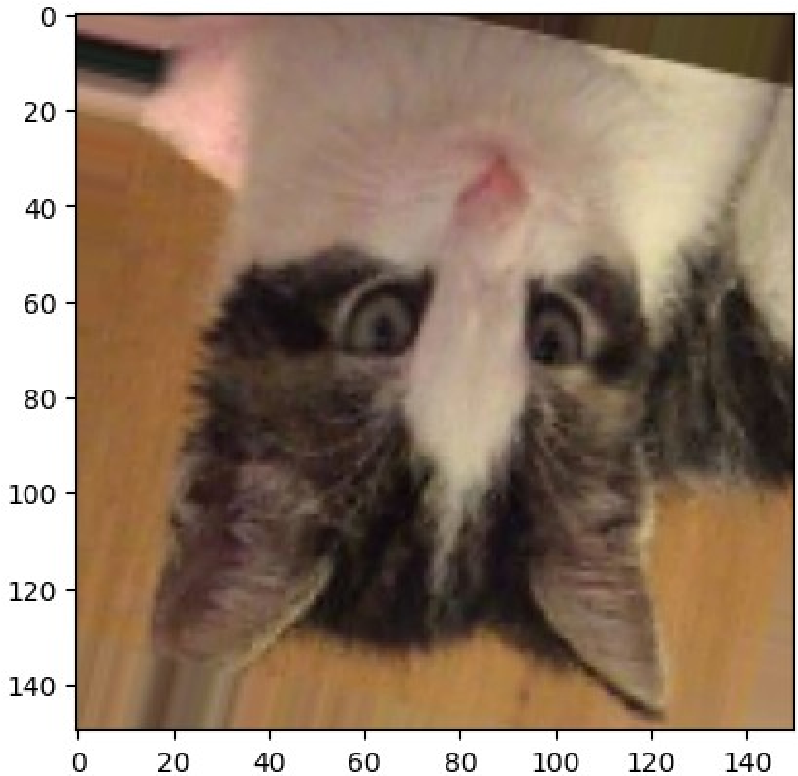
```
# This is module with image preprocessing utilities
import keras.utils as image

fnames = [os.path.join(train_cats_dir10, fname) for fname in
os.listdir(train_cats_dir10)]
```

```
# We pick one image to "augment"
img_path = fnames[3]

# Read the image and resize it
img = image.load_img(img_path, target_size=(150, 150))

# Convert it to a Numpy array with shape (150, 150, 3)
x = image.img_to_array(img)

# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

# The .flow() command below generates batches of randomly transformed
images.
# It will loop indefinitely, so we need to `break` the loop at some
point!
i = 0
for batch in moderate_datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```




0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140



Using Class-sensitiveLearning

```
train_labels = train2_generator.labels

# Step 1: Calculate class weights
class_weights = class_weight.compute_class_weight('balanced',
classes=np.unique(train_labels), y=train_labels)

# Convert class weights to a dictionary for easy use in Keras
class_weights_dict = {i: weight for i, weight in
enumerate(class_weights)}

reverse_class_weights_dict = {cls: 1.0 / weight for cls, weight in
class_weights_dict.items()}

# Print or use the class weights as needed
print("Class Weights:", class_weights_dict)
print("reverse Class Weights:", reverse_class_weights_dict)
```

```
Class Weights: {0: 5.0, 1: 0.5555555555555556}
reverse Class Weights: {0: 0.2, 1: 1.7999999999999998}
```

Retrain the model

```
# Build and train your model on the balanced dataset with class-
sensitive learning
```

```
model = build_cnn_model_new_dataset()

# Train the model
history2 = model.fit(
      train2_generator,
      steps_per_epoch=2000//train2_generator.batch_size,
      epochs=20,
      validation_data=validation2_generator,
      validation_steps=1000//validation2_generator.batch_size,
      class_weight=reverse_class_weights_dict)

# Evaluate the model on the validation set
val_loss, val_acc = model.evaluate(validation2_generator,
steps=len(validation2_generator))
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.
```

```
Epoch 1/20
66/66 [==============================] - 74s 1s/step - loss: 0.1575 -
acc: 0.8858 - val_loss: 0.3690 - val_acc: 0.8990
Epoch 2/20
66/66 [==============================] - 72s 1s/step - loss: 0.1170 -
acc: 0.8995 - val_loss: 0.4404 - val_acc: 0.9000
Epoch 3/20
66/66 [==============================] - 71s 1s/step - loss: 0.1123 -
acc: 0.8990 - val_loss: 0.4140 - val_acc: 0.9000
Epoch 4/20
66/66 [==============================] - 71s 1s/step - loss: 0.1106 -
acc: 0.9005 - val_loss: 0.4748 - val_acc: 0.9000
Epoch 5/20
66/66 [==============================] - 71s 1s/step - loss: 0.1101 -
acc: 0.9005 - val_loss: 0.4582 - val_acc: 0.9020
Epoch 6/20
66/66 [==============================] - 72s 1s/step - loss: 0.1091 -
acc: 0.9000 - val_loss: 0.4232 - val_acc: 0.9000
Epoch 7/20
66/66 [==============================] - 72s 1s/step - loss: 0.1088 -
acc: 0.9005 - val_loss: 0.4682 - val_acc: 0.9010
Epoch 8/20
66/66 [==============================] - 73s 1s/step - loss: 0.1078 -
acc: 0.9000 - val_loss: 0.4293 - val_acc: 0.9000
Epoch 9/20
66/66 [==============================] - 72s 1s/step - loss: 0.1085 -
acc: 0.8990 - val_loss: 0.4752 - val_acc: 0.8990
Epoch 10/20
66/66 [==============================] - 71s 1s/step - loss: 0.1068 -
acc: 0.9000 - val_loss: 0.5445 - val_acc: 0.9010
Epoch 11/20
```

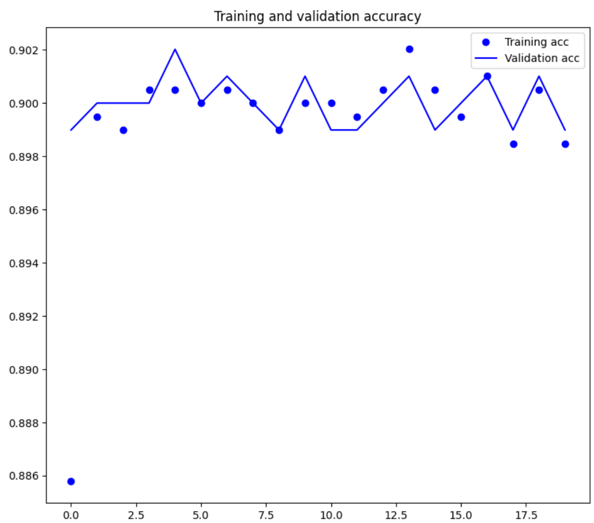
```
66/66 [==============================] - 69s 1s/step - loss: 0.1061 - acc: 0.9000 - val_loss: 0.4695 - val_acc: 0.8990
Epoch 12/20
66/66 [==============================] - 69s 1s/step - loss: 0.1063 - acc: 0.8995 - val_loss: 0.4293 - val_acc: 0.8990
Epoch 13/20
66/66 [==============================] - 69s 1s/step - loss: 0.1064 - acc: 0.9005 - val_loss: 0.4693 - val_acc: 0.9000
Epoch 14/20
66/66 [==============================] - 70s 1s/step - loss: 0.1032 - acc: 0.9020 - val_loss: 0.4433 - val_acc: 0.9010
Epoch 15/20
66/66 [==============================] - 69s 1s/step - loss: 0.1050 - acc: 0.9005 - val_loss: 0.4752 - val_acc: 0.8990
Epoch 16/20
66/66 [==============================] - 70s 1s/step - loss: 0.1056 - acc: 0.8995 - val_loss: 0.3857 - val_acc: 0.9000
Epoch 17/20
66/66 [==============================] - 76s 1s/step - loss: 0.1035 - acc: 0.9010 - val_loss: 0.4184 - val_acc: 0.9010
Epoch 18/20
66/66 [==============================] - 69s 1s/step - loss: 0.1066 - acc: 0.8985 - val_loss: 0.4389 - val_acc: 0.8990
Epoch 19/20
66/66 [==============================] - 69s 1s/step - loss: 0.1032 - acc: 0.9005 - val_loss: 0.4201 - val_acc: 0.9010
Epoch 20/20
66/66 [==============================] - 68s 1s/step - loss: 0.1060 - acc: 0.8985 - val_loss: 0.4324 - val_acc: 0.8990
34/34 [==============================] - 9s 253ms/step - loss: 0.4281 - acc: 0.9000
```

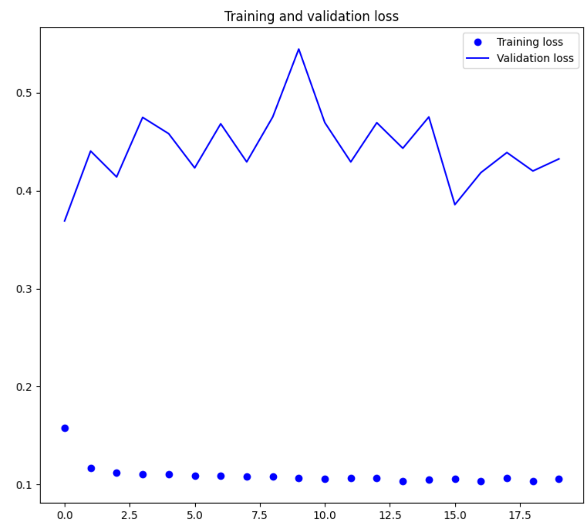
Plot the loss and accuracy

of the model over the training and validation data during training:

```python
import matplotlib.pyplot as plt

acc = history2.history['acc']
val_acc = history2.history['val_acc']
loss = history2.history['loss']
val_loss = history2.history['val_loss']

epochs = range(len(acc))

plt.figure(figsize=(20,8))

plt.subplot(1,2,1)
plt.plot(epochs, acc, 'bo', label='Training acc')
```

```
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

# plt.figure()
plt.subplot(1,2,2)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

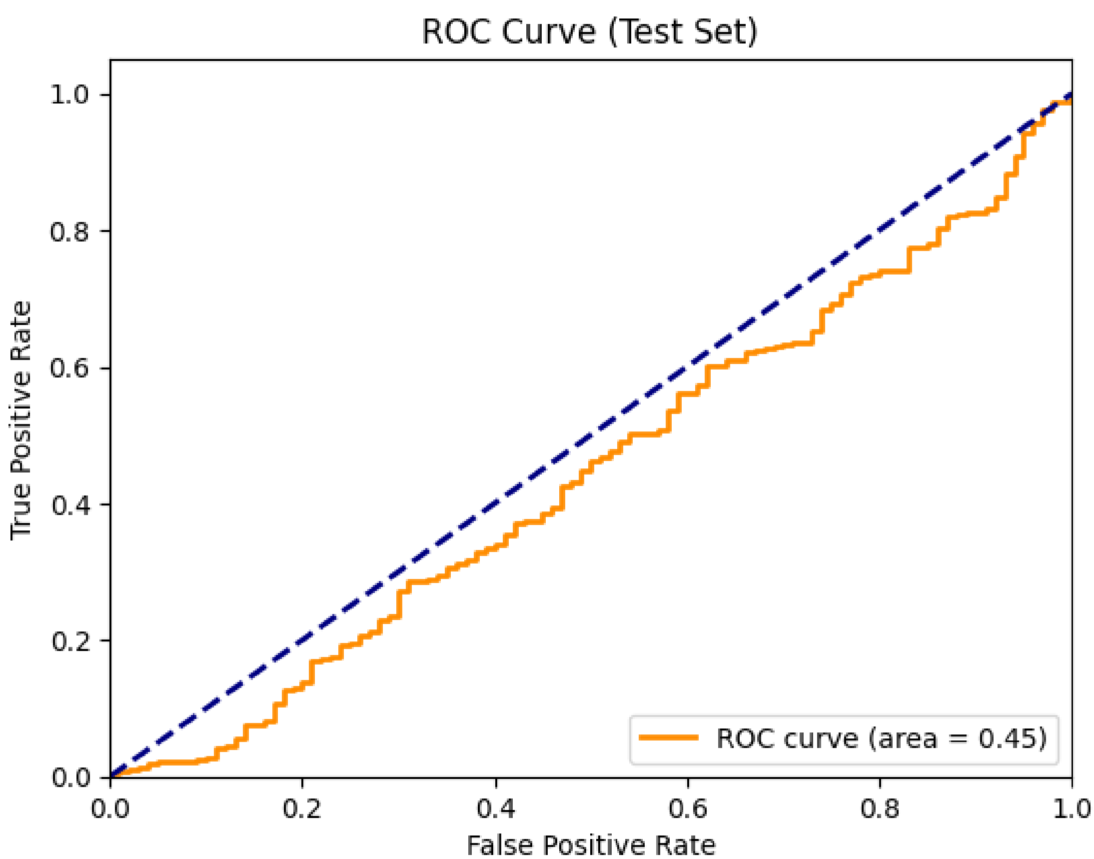




Plot the ROC curves

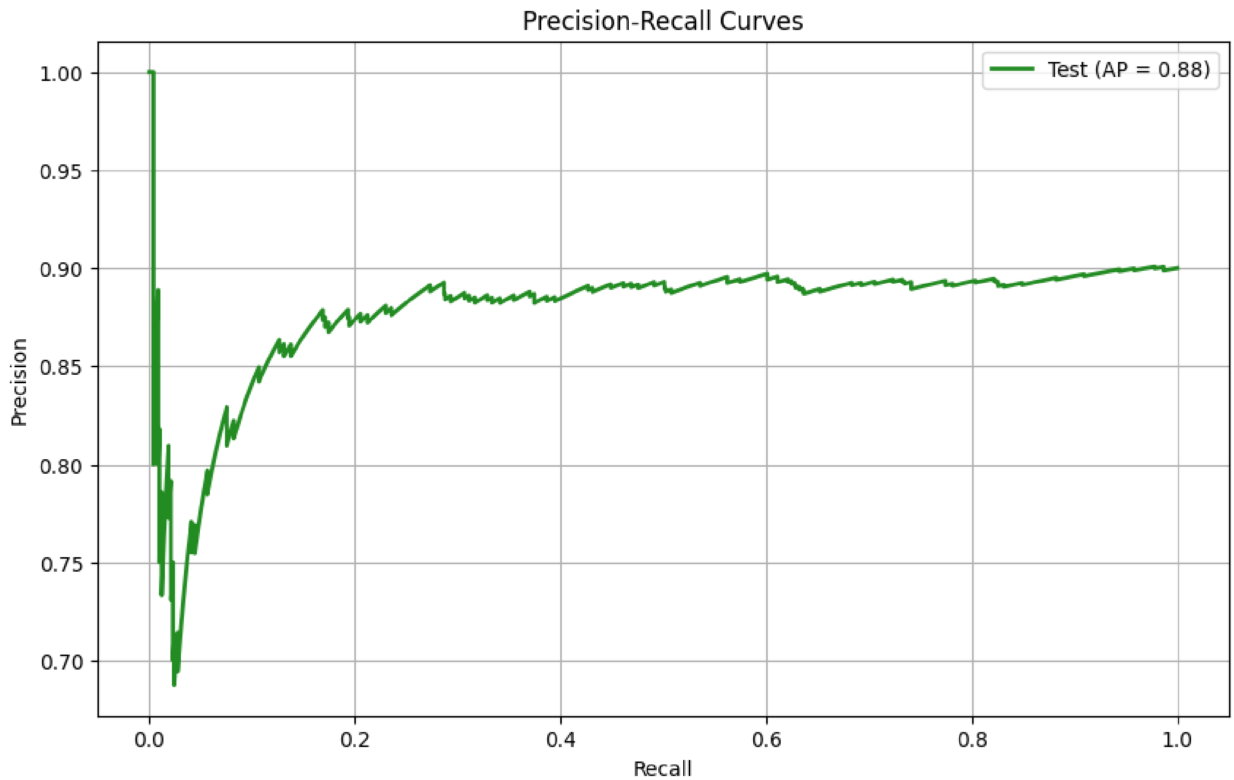
```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

def find_labels_and_probability(img_gen):
  y_true = img_gen.classes

  # Get predicted probabilities (y_score)
  y_pred = model.predict(img_gen)

  return y_true, y_pred

test_y_true, test_probs = find_labels_and_probability(test2_generator)

34/34 [==============================] - 9s 264ms/step

# Function to plot ROC curve
def plot_roc_curve(y_true, y_score, title):
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)
```

```
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()

# Plot ROC curves for training, validation, and test sets
# plot_roc_curve(train_y_true, train_probs, title='ROC Curve (Training
Set)')
# plot_roc_curve(val_y_true, val_probs, title='ROC Curve (Validation
Set)')
plot_roc_curve(test_y_true, test_probs, title='ROC Curve (Test Set)')
```



Plot the Recall-Precision curves

```
from sklearn.metrics import precision_recall_curve,
average_precision_score

# Compute precision and recall for each dataset
# train_precision, train_recall, threshold =
precision_recall_curve(train_y_true, train_probs)
# val_precision, val_recall, threshold =
precision_recall_curve(val_y_true, val_probs)
test_precision, test_recall, threshold =
precision_recall_curve(test_y_true, test_probs)

# Compute average precision (AUC-PR) for each dataset
# train_average_precision = average_precision_score(train_y_true,
train_probs)
# val_average_precision = average_precision_score(val_y_true,
val_probs)
test_average_precision = average_precision_score(test_y_true,
test_probs)

# Plot Precision-Recall curves for all datasets
plt.figure(figsize=(10, 6))
# plt.plot(train_recall, train_precision, color='darkorange', lw=2,
label=f'Training (AP = {train_average_precision:.2f})')
# plt.plot(val_recall, val_precision, color='royalblue', lw=2,
label=f'Validation (AP = {val_average_precision:.2f})')
plt.plot(test_recall, test_precision, color='forestgreen', lw=2,
label=f'Test (AP = {test_average_precision:.2f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```

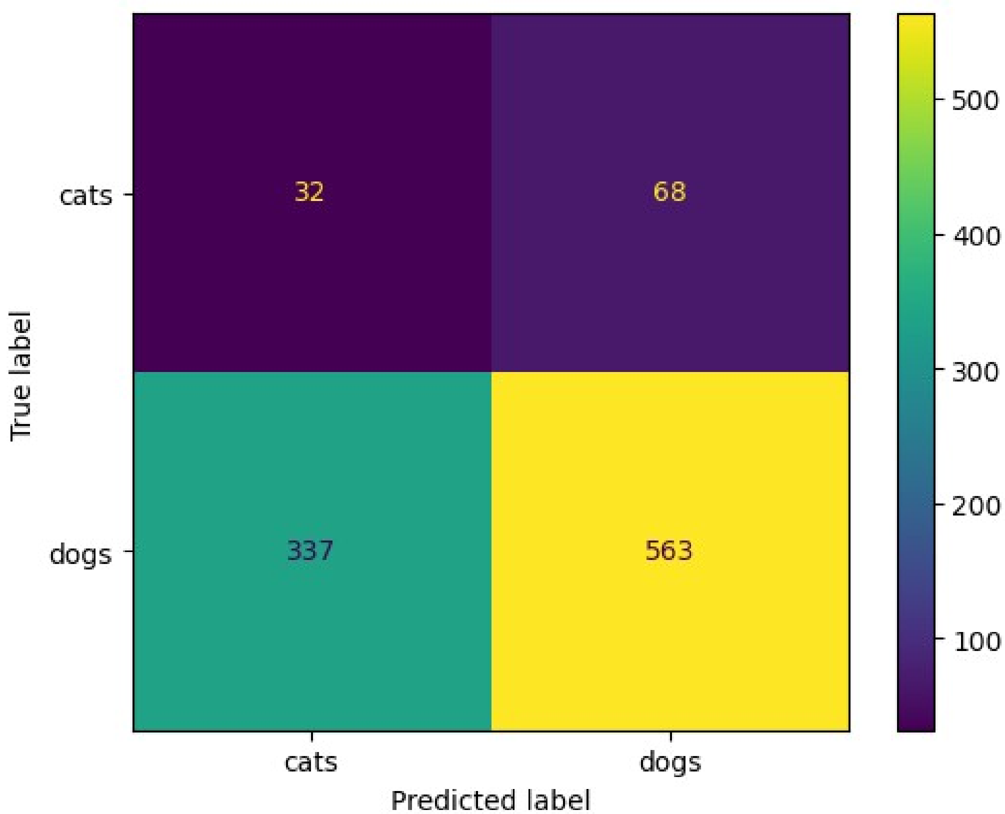


Confusion matrix for 50% threshold

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def find_labels_and_probability(y_true, y_probs):
  threshold = 0.98889
  # Convert probabilities to binary classes using the 50% threshold
  y_pred = (y_probs >= threshold).astype(int)  # Predicted classes (0
or 1)

  # Create the confusion matrix
  conf_matrix = confusion_matrix(y_true, y_pred)

  return conf_matrix

# val_conf_matrix = find_labels_and_probability(val_y_true, val_probs)
test_conf_matrix = find_labels_and_probability(test_y_true,
test_probs)

# Print the confusion matrix
print("Confusion Matrix Test Data (Threshold = 50%):")
print(test_conf_matrix)

# Display the confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=test_conf_matrix,
display_labels=test2_generator.class_indices)
```

```
disp.plot(cmap='viridis', values_format='d')

# print("Confusion Matrix Validation Data (Threshold = 50%):")
# print(val_conf_matrix)
```

```
Confusion Matrix Test Data (Threshold = 50%):
[[ 32  68]
 [337 563]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7e6ba5673fd0>
```



1% rare class dataset

Using DataAugmentation

Tweaking the augmentation values further and adding more settings to strike a balance between the 2 datasets

```
agrresive_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
```

```
    shear_range=0.2,
    # zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest',
    brightness_range=[0.5, 1.5],  # Adjust brightness levels
    channel_shift_range=0.2,  # Randomly shift color channels
    zoom_range=[0.8, 1.2],  # Randomly zoom in or out
    vertical_flip=True,  # Flip vertically
    featurewise_center=True,  # Apply mean centering
    featurewise_std_normalization=True,  # Apply standardization
    zca_whitening=True,  # Apply ZCA whitening
)
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/preprocessing/
image.py:1451: UserWarning: This ImageDataGenerator specifies
`zca_whitening` which overrides setting
of`featurewise_std_normalization`.
  warnings.warn(
```

```
train3_generator = agrresive_datagen.flow_from_directory(
        # This is the target directory
        train_dir1,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=30,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation3_generator = validation_datagen.flow_from_directory(
        validation_dir1,
        target_size=(150, 150),
        batch_size=30,
        class_mode='binary')

test3_generator = test_datagen.flow_from_directory(
        test_dir1,
        target_size=(150, 150),
        batch_size=30,
        class_mode='binary')
```

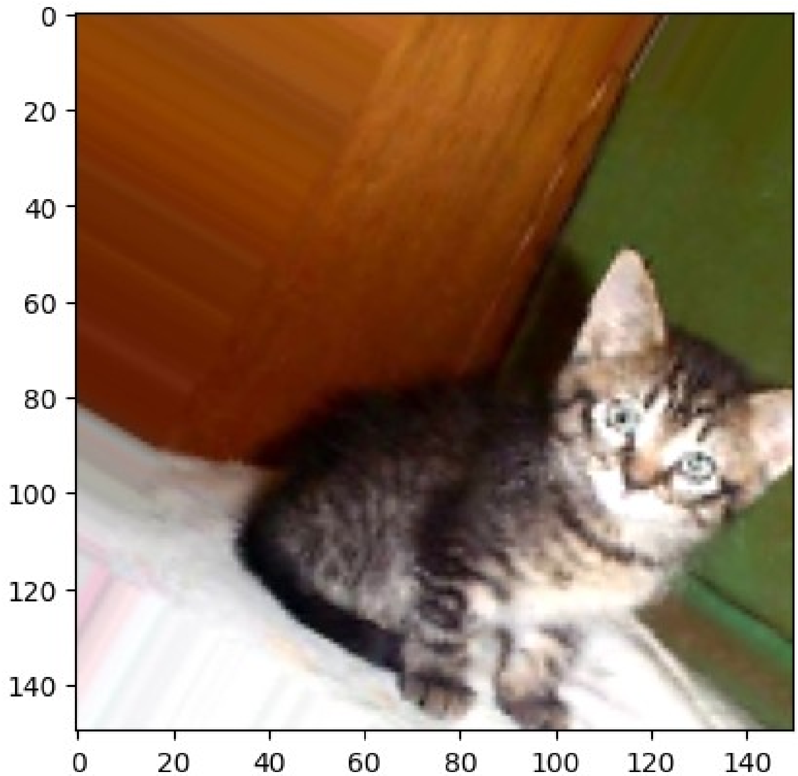
```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

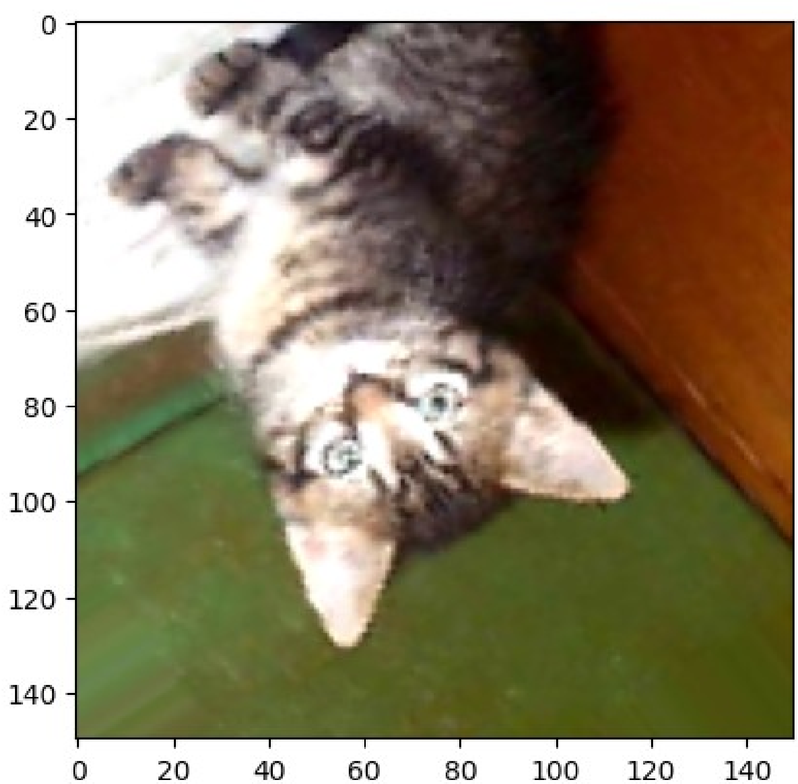
look at our augmented images

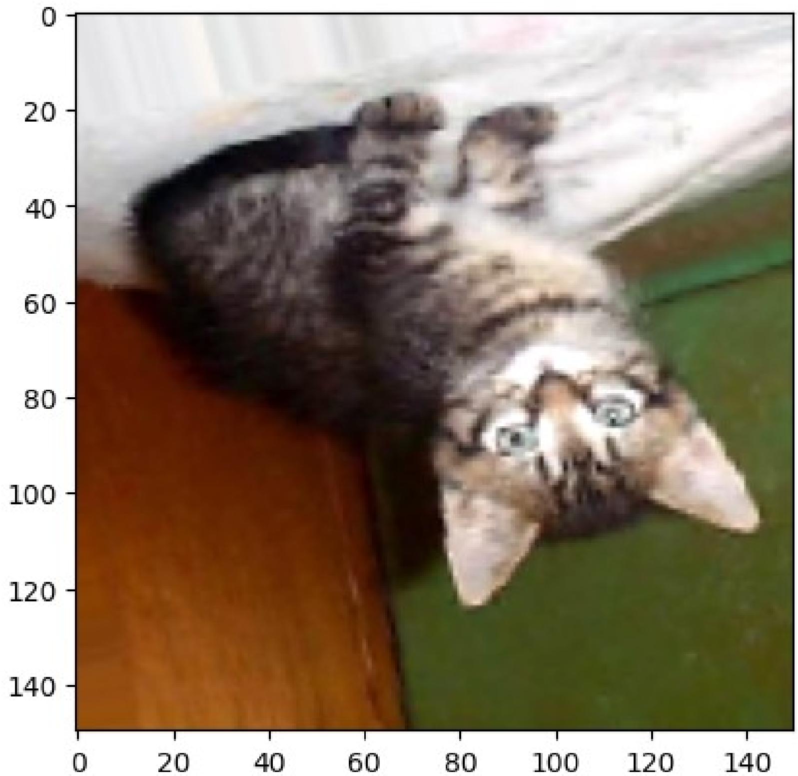
```
# This is module with image preprocessing utilities
import keras.utils as image

fnames = [os.path.join(train_cats_dir1, fname) for fname in
```

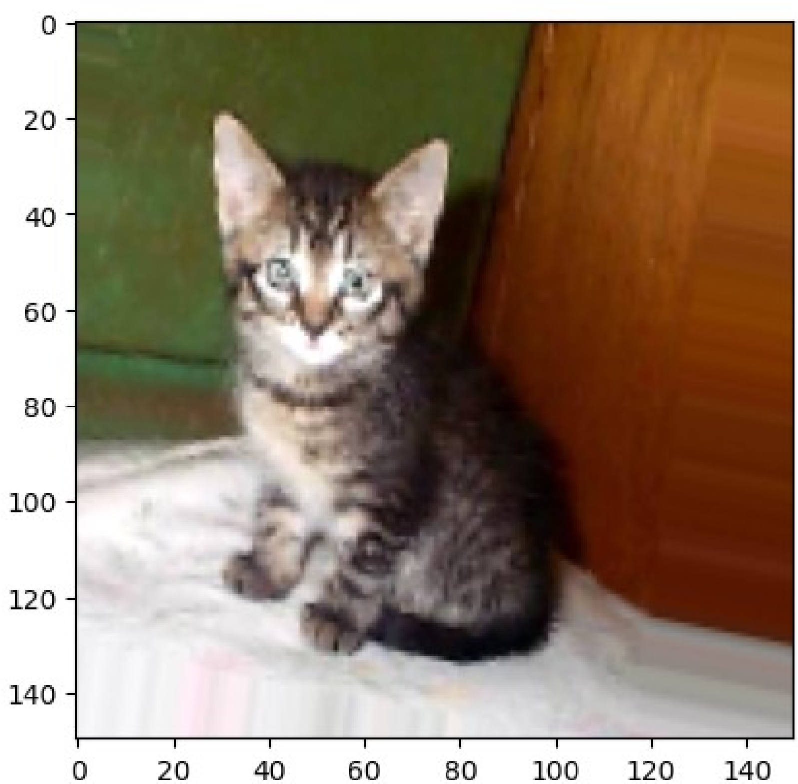
```
os.listdir(train_cats_dir1)]

# We pick one image to "augment"
img_path = fnames[3]

# Read the image and resize it
img = image.load_img(img_path, target_size=(150, 150))

# Convert it to a Numpy array with shape (150, 150, 3)
x = image.img_to_array(img)

# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

# The .flow() command below generates batches of randomly transformed
images.
# It will loop indefinitely, so we need to `break` the loop at some
point!
i = 0
for batch in agrresive_datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/preprocessing/
image.py:1861: UserWarning: This ImageDataGenerator specifies
`featurewise_center`, but it hasn't been fit on any training data. Fit
it first by calling `.fit(numpy_data)`.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/keras/src/preprocessing/image.
py:1884: UserWarning: This ImageDataGenerator specifies
`zca_whitening`, but it hasn't been fit on any training data. Fit it
first by calling `.fit(numpy_data)`.
  warnings.warn(
```






0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140


0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140

Using Class-sensitiveLearning

```
train_labels = train3_generator.labels

# Step 1: Calculate class weights
class_weights = class_weight.compute_class_weight('balanced',
classes=np.unique(train_labels), y=train_labels)

# Convert class weights to a dictionary for easy use in Keras
class_weights_dict = {i: weight for i, weight in
enumerate(class_weights)}

reverse_class_weights_dict = {cls: 1.0 / weight for cls, weight in
class_weights_dict.items()}

# Print or use the class weights as needed
print("Class Weights:", class_weights_dict)
print("reverse Class Weights:", reverse_class_weights_dict)

Class Weights: {0: 50.0, 1: 0.5050505050505051}
reverse Class Weights: {0: 0.02, 1: 1.98}
```

Retrain the model

```
# Build and train your model on the balanced dataset with class-
sensitive learning
model = build_cnn_model_new_dataset()

# Train the model
history3 = model.fit(
      train3_generator,
      steps_per_epoch=2000//train3_generator.batch_size,
      epochs=20,
      validation_data=validation3_generator,
      validation_steps=1000//validation3_generator.batch_size,
      class_weight=reverse_class_weights_dict)

# Evaluate the model on the validation set
val_loss, val_acc = model.evaluate(validation3_generator,
steps=len(validation3_generator))

WARNING:absl:`lr` is deprecated in Keras optimizer, please use
`learning_rate` or use the legacy optimizer,
e.g.,tf.keras.optimizers.legacy.RMSprop.
/usr/local/lib/python3.10/dist-packages/keras/src/preprocessing/image.
py:1861: UserWarning: This ImageDataGenerator specifies
`featurewise_center`, but it hasn't been fit on any training data. Fit
it first by calling `.fit(numpy_data)`.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/keras/src/preprocessing/image.
py:1884: UserWarning: This ImageDataGenerator specifies
```

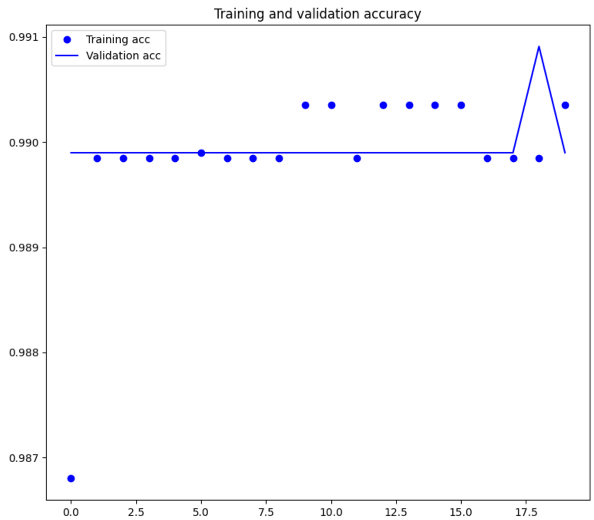
```
`zca_whitening`, but it hasn't been fit on any training data. Fit it
first by calling `.fit(numpy_data)`.
  warnings.warn(

Epoch 1/20
66/66 [==============================] - 72s 1s/step - loss: 0.0227 -
acc: 0.9868 - val_loss: 0.1562 - val_acc: 0.9899
Epoch 2/20
66/66 [==============================] - 73s 1s/step - loss: 0.0032 -
acc: 0.9898 - val_loss: 0.0881 - val_acc: 0.9899
Epoch 3/20
66/66 [==============================] - 71s 1s/step - loss: 0.0026 -
acc: 0.9898 - val_loss: 0.1148 - val_acc: 0.9899
Epoch 4/20
66/66 [==============================] - 76s 1s/step - loss: 0.0023 -
acc: 0.9898 - val_loss: 0.1047 - val_acc: 0.9899
Epoch 5/20
66/66 [==============================] - 73s 1s/step - loss: 0.0023 -
acc: 0.9898 - val_loss: 0.0961 - val_acc: 0.9899
Epoch 6/20
66/66 [==============================] - 74s 1s/step - loss: 0.0022 -
acc: 0.9899 - val_loss: 0.1046 - val_acc: 0.9899
Epoch 7/20
66/66 [==============================] - 71s 1s/step - loss: 0.0022 -
acc: 0.9898 - val_loss: 0.0992 - val_acc: 0.9899
Epoch 8/20
66/66 [==============================] - 71s 1s/step - loss: 0.0021 -
acc: 0.9898 - val_loss: 0.0840 - val_acc: 0.9899
Epoch 9/20
66/66 [==============================] - 72s 1s/step - loss: 0.0021 -
acc: 0.9898 - val_loss: 0.0975 - val_acc: 0.9899
Epoch 10/20
66/66 [==============================] - 72s 1s/step - loss: 0.0021 -
acc: 0.9904 - val_loss: 0.1038 - val_acc: 0.9899
Epoch 11/20
66/66 [==============================] - 73s 1s/step - loss: 0.0020 -
acc: 0.9904 - val_loss: 0.0979 - val_acc: 0.9899
Epoch 12/20
66/66 [==============================] - 71s 1s/step - loss: 0.0021 -
acc: 0.9898 - val_loss: 0.0968 - val_acc: 0.9899
Epoch 13/20
66/66 [==============================] - 73s 1s/step - loss: 0.0020 -
acc: 0.9904 - val_loss: 0.0910 - val_acc: 0.9899
Epoch 14/20
66/66 [==============================] - 71s 1s/step - loss: 0.0020 -
acc: 0.9904 - val_loss: 0.0924 - val_acc: 0.9899
Epoch 15/20
66/66 [==============================] - 71s 1s/step - loss: 0.0020 -
acc: 0.9904 - val_loss: 0.0955 - val_acc: 0.9899
Epoch 16/20
```

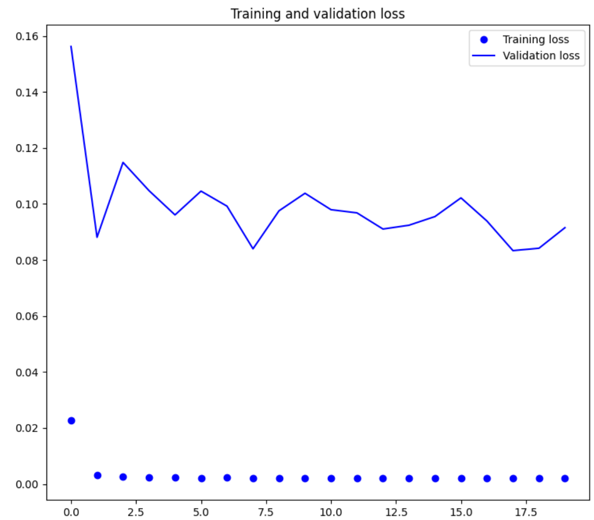
```
66/66 [==============================] - 72s 1s/step - loss: 0.0020 -
acc: 0.9904 - val_loss: 0.1022 - val_acc: 0.9899
Epoch 17/20
66/66 [==============================] - 73s 1s/step - loss: 0.0021 -
acc: 0.9898 - val_loss: 0.0939 - val_acc: 0.9899
Epoch 18/20
66/66 [==============================] - 72s 1s/step - loss: 0.0021 -
acc: 0.9898 - val_loss: 0.0833 - val_acc: 0.9899
Epoch 19/20
66/66 [==============================] - 89s 1s/step - loss: 0.0021 -
acc: 0.9898 - val_loss: 0.0842 - val_acc: 0.9909
Epoch 20/20
66/66 [==============================] - 80s 1s/step - loss: 0.0020 -
acc: 0.9904 - val_loss: 0.0915 - val_acc: 0.9899
34/34 [==============================] - 12s 340ms/step - loss: 0.0906
- acc: 0.9900
```

Plot the loss and accuracy

of the model over the training and validation data during training:

```python
import matplotlib.pyplot as plt

acc = history3.history['acc']
val_acc = history3.history['val_acc']
loss = history3.history['loss']
val_loss = history3.history['val_loss']

epochs = range(len(acc))

plt.figure(figsize=(20,8))

plt.subplot(1,2,1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.subplot(1,2,2)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```





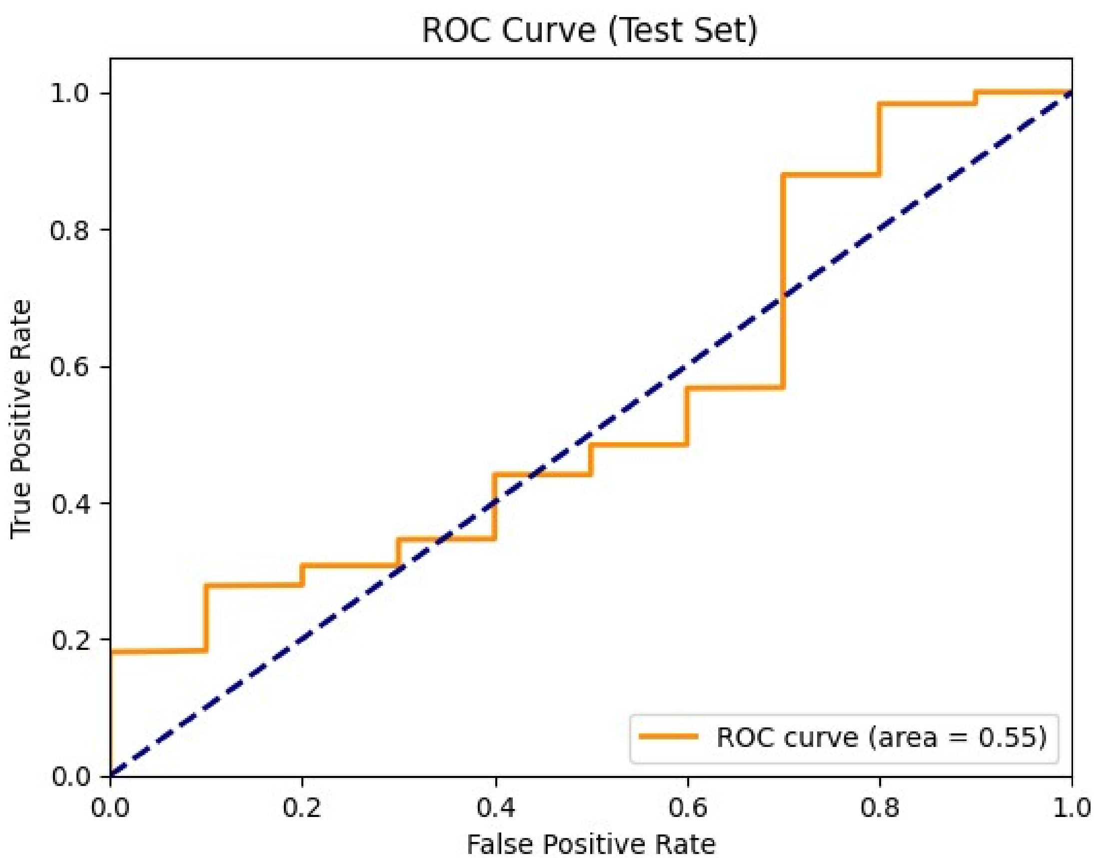
Plot the ROC curves

```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

def find_labels_and_probability(img_gen):
  y_true = img_gen.classes

  # Get predicted probabilities (y_score)
  y_pred = model.predict(img_gen)

  return y_true, y_pred

test_y_true, test_probs = find_labels_and_probability(test3_generator)
```

```
34/34 [==============================] - 9s 270ms/step
```

```
# Function to plot ROC curve
def plot_roc_curve(y_true, y_score, title):
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()

# Plot ROC curves for training, validation, and test sets
```

```
# plot_roc_curve(train_y_true, train_probs, title='ROC Curve (Training Set)')
# plot_roc_curve(val_y_true, val_probs, title='ROC Curve (Validation Set)')
plot_roc_curve(test_y_true, test_probs, title='ROC Curve (Test Set)')
```

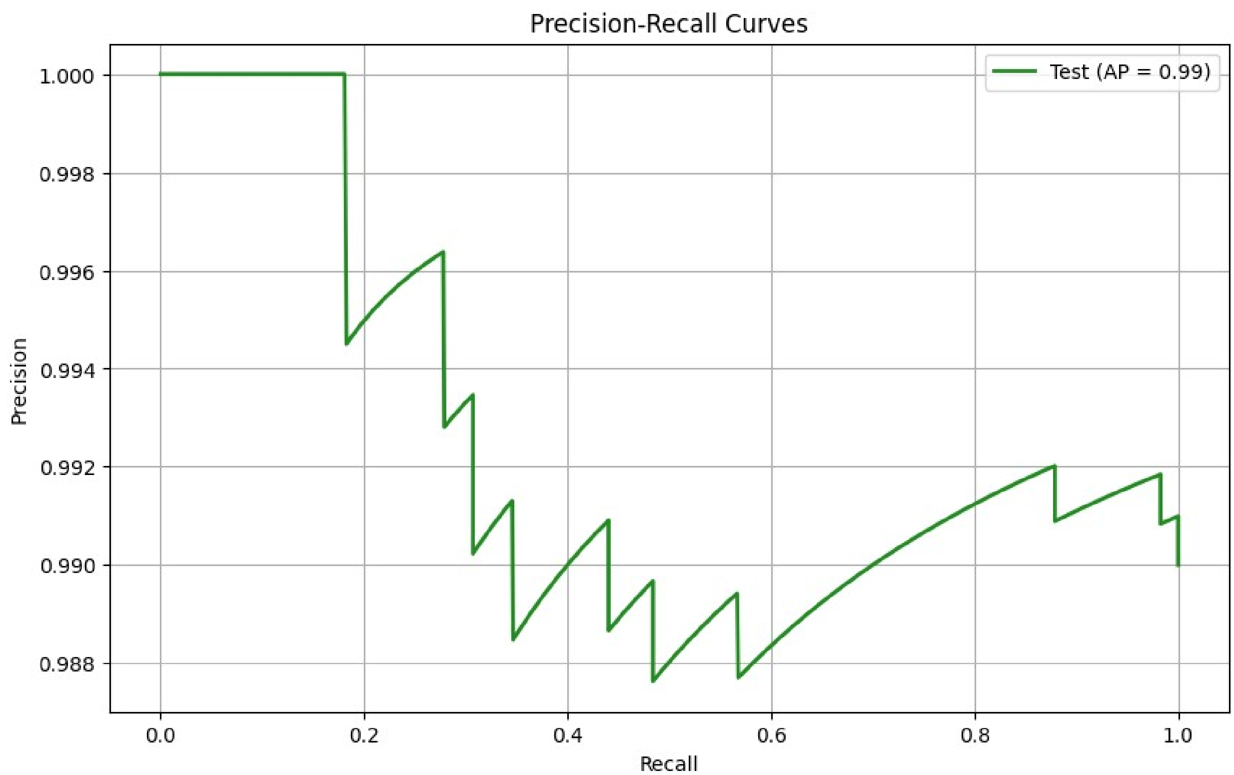


True positive rate is much higher than false positive, with the usage of hyper parameter tuned model at the start of the Improve section in this notebook, this can be further improved

Plot the Recall-Precision curves

```
from sklearn.metrics import precision_recall_curve, average_precision_score

# Compute precision and recall for each dataset
test_precision, test_recall, threshold = precision_recall_curve(test_y_true, test_probs)

# Compute average precision (AUC-PR) for each dataset
test_average_precision = average_precision_score(test_y_true, test_probs)
```

```
# Plot Precision-Recall curves for all datasets
plt.figure(figsize=(10, 6))
plt.plot(test_recall, test_precision, color='forestgreen', lw=2,
label=f'Test (AP = {test_average_precision:.2f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```

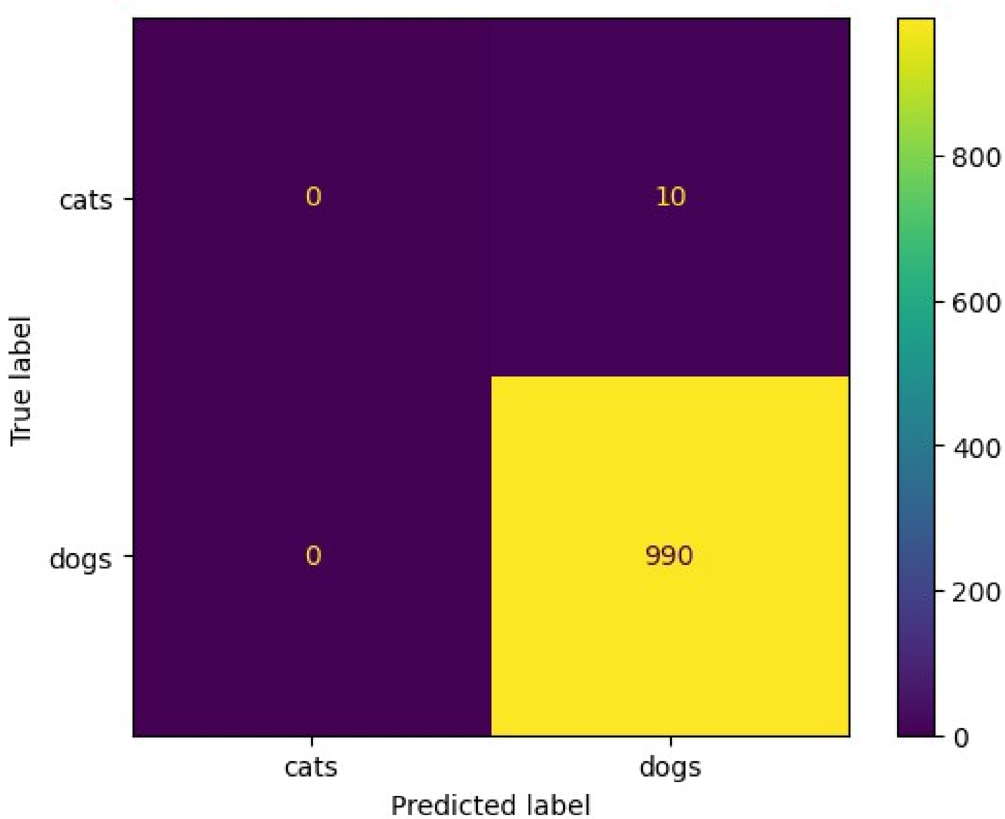


Confusion matrix for 50% threshold

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def find_labels_and_probability(y_true, y_probs):
  threshold = 0.5
  # Convert probabilities to binary classes using the 50% threshold
  y_pred = (y_probs >= threshold).astype(int)  # Predicted classes (0
or 1)

  # Create the confusion matrix
  conf_matrix = confusion_matrix(y_true, y_pred)

  return conf_matrix
```

```python
# val_conf_matrix = find_labels_and_probability(val_y_true, val_probs)
test_conf_matrix = find_labels_and_probability(test_y_true,
test_probs)

# Print the confusion matrix
print("Confusion Matrix Test Data (Threshold = 50%):")
print(test_conf_matrix)

# Display the confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=test_conf_matrix,
display_labels=test3_generator.class_indices)
disp.plot(cmap='viridis', values_format='d')

# print("Confusion Matrix Validation Data (Threshold = 50%):")
# print(val_conf_matrix)
```

```
Confusion Matrix Test Data (Threshold = 50%):
[[  0  10]
 [  0 990]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7e6b9e502f50>
```



Observation

Although using class imbalances was handled and the overfitting was managed by aggressively augmenting the data on the rare class and by providing tweaking the weightage and alloting more weightage to the rare class, however, it could have been better if we will further train the model with the Best Hyperparameters obtained from optuna tunining like the best learning rate, and number of filters in the augmented data etc along with regularization and custom loss on class weights.

Here, although we achieved ~98% accuracy, but there is overfitting that could have been lowered with the usage of the hyperparameter tuned model and further improving the accuracy for small amount of dataset.

The best_model that I plotted above in after using teh hyperparams had negligent overfitting as per the plot of training loss to validation loss. If I can use the same best hyperparameter tuned model, the Improved rare class events for 10% and 1% will have similar results and accuracy.

model1 = build_cnn_CSL_model(best_learning_rate, best_num_filters, best_dropout_rate, best_batch_size, class_weights) for the improve section.

Also if the hyperparameters prnted above could be used in data augmentation, that will also itigate the overfitting further