Cat Vs Dog Image Classification Project

Dataset: dogs vs cats

- Size= 1GB
- 10,000 Images of cats & dogs
- Link: https://www.kaggle.com/datasets/salader/dogs-vs-cats

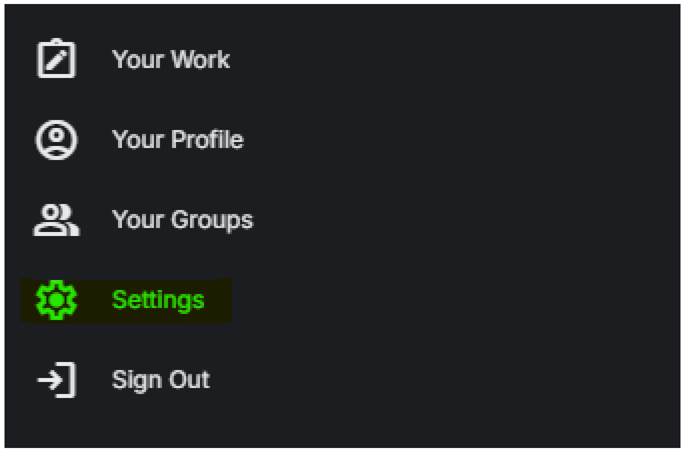
Take this dataset on Google Colab without locally downloading it:

Colab NB Link → https://colab.research.google.com/drive/1-RCfpCvN-ZFFDvdWzj2BnFpsR5zBX1WP#scrollTo=ecwzbtGIlDye

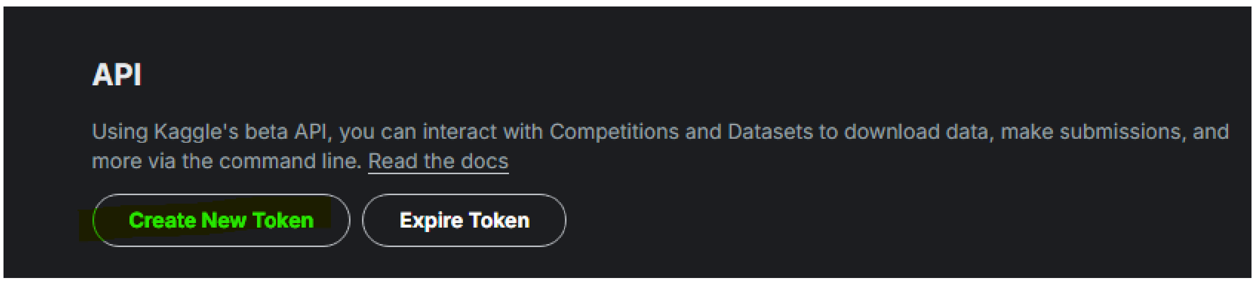
Step 1:

- Download the API .json file

- ***Settings***



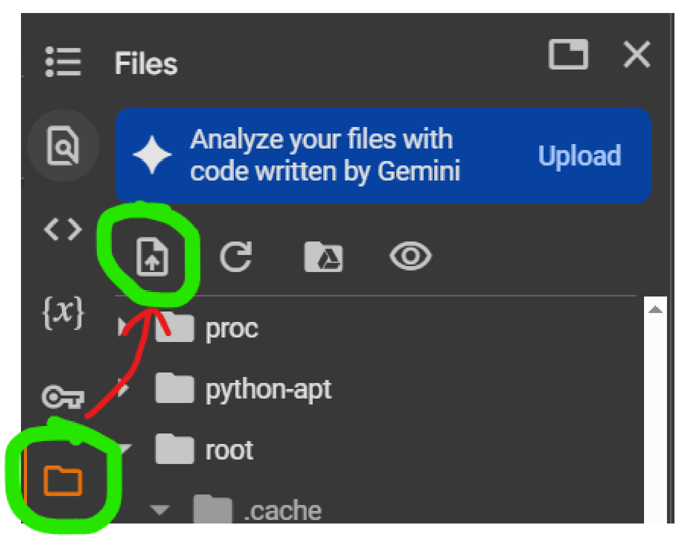
- ***API → Create new token***



- It will download a json file

Step 2:

- Load the file in Google Colab



Step 3:

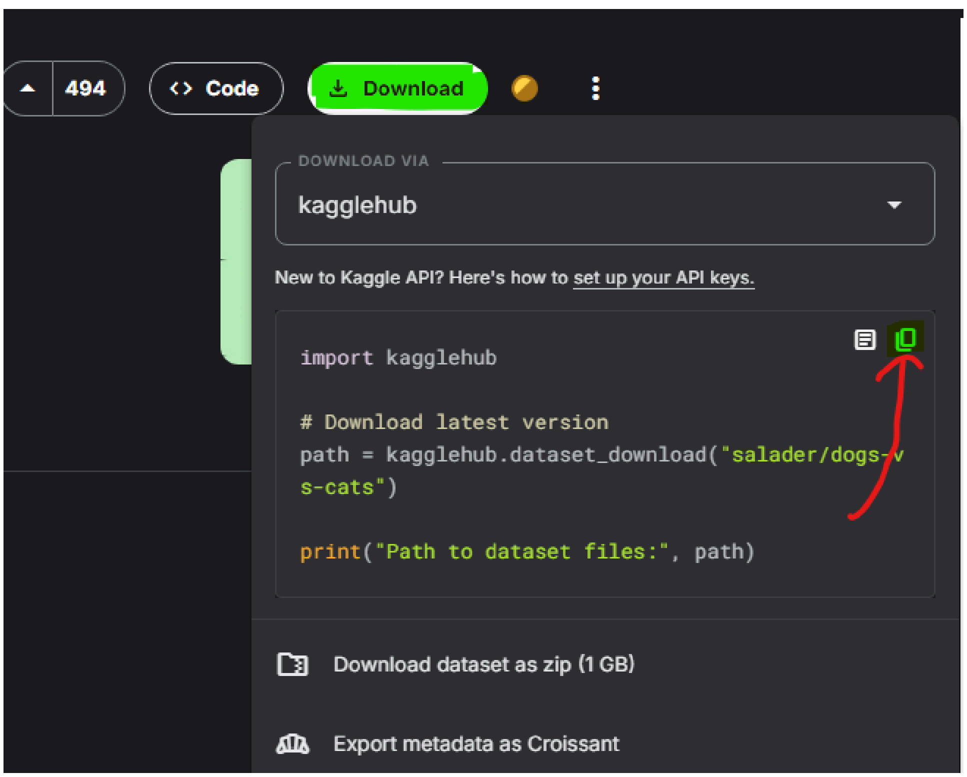
- **Run the following command:**

```
!mkdir -p ~/.kaggle
!mv kaggle.json ~/.kaggle/
```

- `-p`: A flag for `mkdir` that:
  - Creates parent directories if they don't exist.
  - Prevents an error if the directory already exists (unlike mkdir without `-p`, which throws an error like `mkdir: cannot create directory '/root/.kaggle'`: File exists).

Step 4:

- Copy the download path from kaggle



Step 5:

- Paste it in colab notebook
- It will download the dataset

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("salader/dogs-vs-cats")

print("Path to dataset files:", path)
```

<u>Path:</u> The dataset files are located at ***/root/.cache/kagglehub/datasets/salader/dogs-vs-cats/versions/***1 inColab virtual machine (VM).

```
/root/.cache/kagglehub/datasets/salader/dogs-vs-cats/versions/1/
    ├── cats/
    │   ├── cat1.jpg
    │   ├── cat2.jpg
    │   └── ...
    ├── dogs/
    │   ├── dog1.jpg
    │   ├── dog2.jpg
    │   └── ...
    └── test/
        ├── test1.jpg
        ├── test2.jpg
        └── ...
```

Step 6:

- Define path to the dataset directory :

```
# Path to the dataset
data_dir = '/root/.cache/kagglehub/datasets/salader/dogs-vs-cats/versions/1'
```

Define test & training data paths

```
import os

train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')

print("Train data path:", train_dir)
print("Test data path:", test_dir)
```

```
Train data path: /root/.cache/kagglehub/datasets/salader/dogs-vs-cats/versions/1/train
Test data path: /root/.cache/kagglehub/datasets/salader/dogs-vs-cats/versions/1/test
```

Use Generators

- It loads data in batches
- it's useful to process large amount of data

Use → `image_dataset_from_directory` function

```
keras.utils.image_dataset_from_directory(
    directory,
    labels="inferred",
    label_mode="int",
    class_names=None,
    color_mode="rgb",
    batch_size=32,
    image_size=(256, 256),
```

```
    shuffle=True,
    seed=None,
    validation_split=None,
    subset=None,
    interpolation="bilinear",
    follow_links=False,
    crop_to_aspect_ratio=False,
    pad_to_aspect_ratio=False,
    data_format=None,
    verbose=True,
)
```

Training Data:

```
from types import LambdaType
train_ds= keras.utils.image_dataset_from_directory(train_dir)
```

- We kept everything as default
- `label_mode="int"` will assign 0 for cat, 1 for dog

What it does exactly:

- Scans the folder at `train_dir`
- Automatically **reads images**, **resizes them**, and **labels them** based on folder names
- Returns a ready-to-train dataset of `(image, label)` pairs

✅ What you get:

A dataset where:

- Each item is `(image_tensor, label)`
- Images are resized to `(256, 256)` by default

- Labels are integers starting from 0 (e.g., class1 → 0, class2 → 1)

Testing/Validation data:

```
validation_ds = keras.utils.image_dataset_from_directory(test_dir)
```

```
Found 20000 files belonging to 2 classes.
Found 5000 files belonging to 2 classes.
```

Normalize the data:

```
def normalize_img(image, label):
  """Normalizes images: `uint8` -> `float32`."""
  return tf.cast(image, tf.float32) / 255., label

train_ds = train_ds.map(normalize_img)
validation_ds= valiation_ds.map(normalize_img)
```

`tf.cast(x, dtype)`

- `x` : The input tensor
- `dtype` : The new data type you want to convert to (like `tf.float32` , `tf.int32` , etc.)

Why divide by `255.` ?

- The images are usually loaded as `uint8` (i.e., integers from 0 to 255).
- `tf.cast(..., tf.float32)` converts them into **floating point numbers** so that we can perform decimal math on them.
- Dividing by 255 **scales the values to the range [0, 1]**.

What is `label` ?

- `label` is the **target or class** associated with the image.

- In **dog vs cat classification**:
  - An image of a dog might have label `0`
  - An image of a cat might have label `1`

Create CNN Model

- 3 Conv layers → 32, 64, 128

```
model =Sequential()

model.add(Conv2D(32, (3,3), activation='relu', input_shape=(256,256,3)))
model.add(MaxPooling2D())

model.add(Conv2D(64, (3,3), activation='relu'))
model.add(MaxPooling2D())

model.add(Conv2D(128, (3,3), activation='relu'))
model.add(MaxPooling2D())

model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))

model.add(Dense(1, activation='sigmoid'))
```

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 254, 254, 32)	896
max_pooling2d (MaxPooling2D)	(None, 127, 127, 32)	0
conv2d_1 (Conv2D)	(None, 125, 125, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 62, 62, 64)	0
conv2d_2 (Conv2D)	(None, 60, 60, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 30, 30, 128)	0
flatten (Flatten)	(None, 115200)	0
dense (Dense)	(None, 128)	14,745,728
dense_1 (Dense)	(None, 36)	4,644
dense_2 (Dense)	(None, 1)	37

```
Total params: 14,843,657 (56.62 MB)
Trainable params: 14,843,657 (56.62 MB)
Non-trainable params: 0 (0.00 B)
```

Compile:

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Fit:

```
history= model.fit(train_ds, epochs=10, validation_data=validation_ds)
```

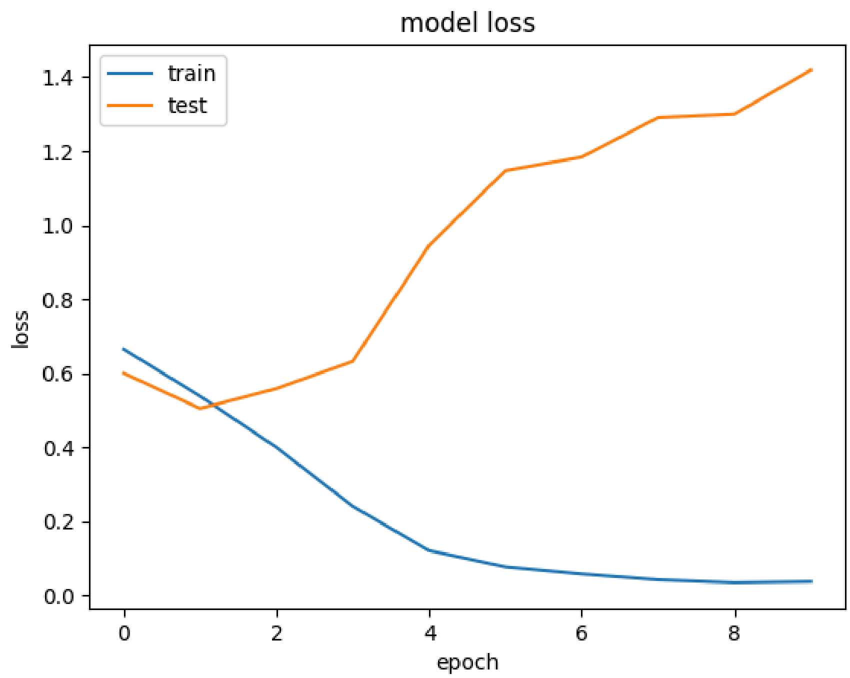
```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 134s 199ms/step - accuracy: 0.5451 - loss: 0.6962 - val_accuracy: 0.6830 - val_loss: 0.599
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.7035 - loss: 0.5702 - val_accuracy: 0.7598 - val_loss: 0.5047
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.7958 - loss: 0.4383 - val_accuracy: 0.7630 - val_loss: 0.5585
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 80s 75ms/step - accuracy: 0.8798 - loss: 0.2908 - val_accuracy: 0.7684 - val_loss: 0.6321
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 44s 70ms/step - accuracy: 0.9406 - loss: 0.1459 - val_accuracy: 0.7650 - val_loss: 0.9451
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 44s 71ms/step - accuracy: 0.9691 - loss: 0.0896 - val_accuracy: 0.7668 - val_loss: 1.1471
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 85s 75ms/step - accuracy: 0.9775 - loss: 0.0639 - val_accuracy: 0.7754 - val_loss: 1.1838
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 75ms/step - accuracy: 0.9854 - loss: 0.0451 - val_accuracy: 0.7554 - val_loss: 1.2904
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 47s 75ms/step - accuracy: 0.9886 - loss: 0.0363 - val_accuracy: 0.7570 - val_loss: 1.2997
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 79s 70ms/step - accuracy: 0.9864 - loss: 0.0390 - val_accuracy: 0.7592 - val_loss: 1.4185
```

👆 Took 12 Mins with GPU

Plot graphs:

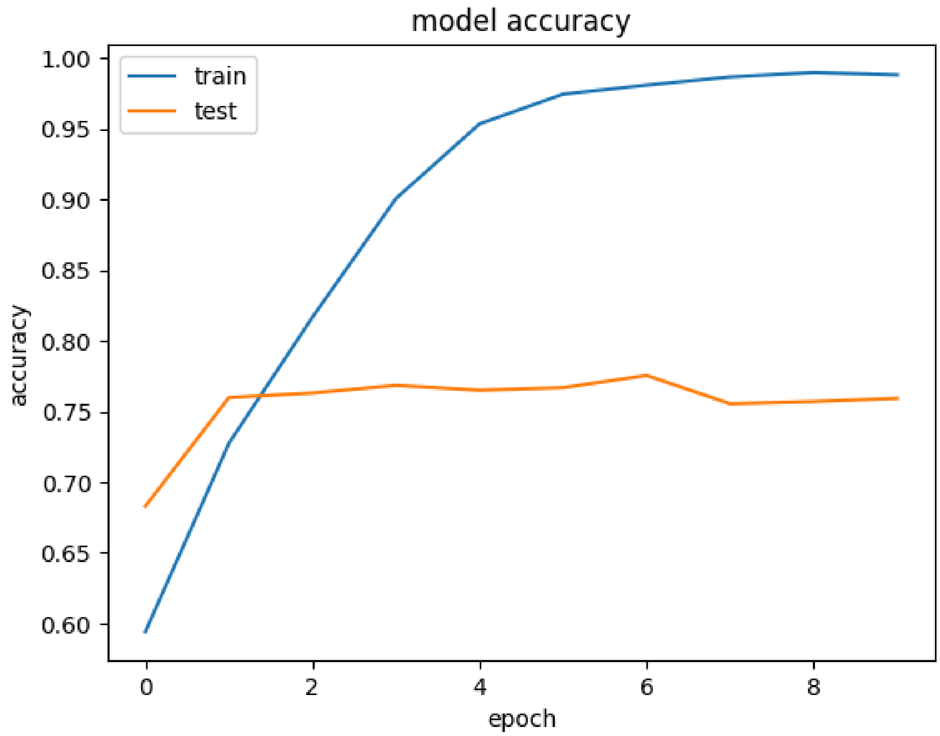
```
import matplotlib.pyplot as plt

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



🔴OVERFITTING👆

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



Reduce Overfitting

- Dropout
- Batch Normalization

```
# prompt: add dropout & batch normalization

model =Sequential()

model.add(Conv2D(32, (3,3), activation='relu', input_shape=(256,256,3)))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D(64, (3,3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())
```

```
model.add(Conv2D(128, (3,3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.1)) # Increased dropout for dense layers

model.add(Dense(64, activation='relu'))
model.add(Dropout(0.1))

model.add(Dense(1, activation='sigmoid'))
model.summary()
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 254, 254, 32)	896
batch_normalization_12 (BatchNormalization)	(None, 254, 254, 32)	128
max_pooling2d_12 (MaxPooling2D)	(None, 127, 127, 32)	0
conv2d_13 (Conv2D)	(None, 125, 125, 64)	18,496
batch_normalization_13 (BatchNormalization)	(None, 125, 125, 64)	256
max_pooling2d_13 (MaxPooling2D)	(None, 62, 62, 64)	0
conv2d_14 (Conv2D)	(None, 60, 60, 128)	73,856
batch_normalization_14 (BatchNormalization)	(None, 60, 60, 128)	512
max_pooling2d_14 (MaxPooling2D)	(None, 30, 30, 128)	0
flatten_4 (Flatten)	(None, 115200)	0
dense_12 (Dense)	(None, 128)	14,745,728
dropout_6 (Dropout)	(None, 128)	0
dense_13 (Dense)	(None, 64)	8,256
dropout_7 (Dropout)	(None, 64)	0
dense_14 (Dense)	(None, 1)	65

Total params: 14,848,193 (56.64 MB)
Trainable params: 14,847,745 (56.64 MB)
Non-trainable params: 448 (1.75 KB)

Compile:

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Fit:

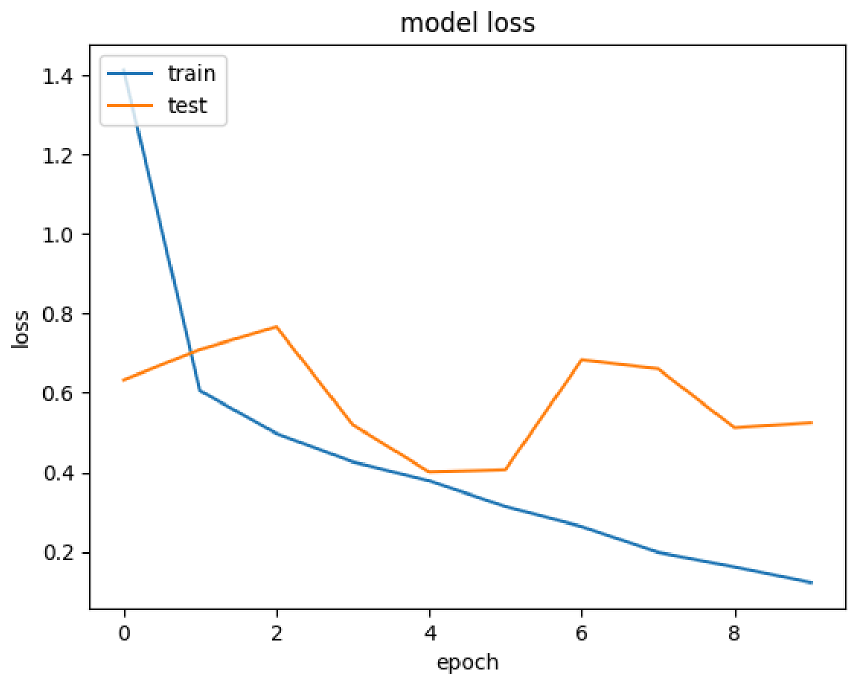
```
history= model.fit(train_ds, epochs=10, validation_data=validation_ds)
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 60s 85ms/step - accuracy: 0.5482 - loss: 2.7477 - val_accuracy: 0.6386 - val_loss: 0.6323
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 79s 85ms/step - accuracy: 0.6632 - loss: 0.6212 - val_accuracy: 0.6236 - val_loss: 0.7090
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.7414 - loss: 0.5260 - val_accuracy: 0.6598 - val_loss: 0.7659
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.7872 - loss: 0.4508 - val_accuracy: 0.7548 - val_loss: 0.5195
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 85s 83ms/step - accuracy: 0.8264 - loss: 0.3975 - val_accuracy: 0.8146 - val_loss: 0.4011
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 79s 79ms/step - accuracy: 0.8513 - loss: 0.3389 - val_accuracy: 0.8128 - val_loss: 0.4067
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 84s 82ms/step - accuracy: 0.8835 - loss: 0.2750 - val_accuracy: 0.7654 - val_loss: 0.6829
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 82ms/step - accuracy: 0.9050 - loss: 0.2149 - val_accuracy: 0.7906 - val_loss: 0.6601
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.9323 - loss: 0.1674 - val_accuracy: 0.8076 - val_loss: 0.5120
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 51s 82ms/step - accuracy: 0.9477 - loss: 0.1300 - val_accuracy: 0.8304 - val_loss: 0.5242
```

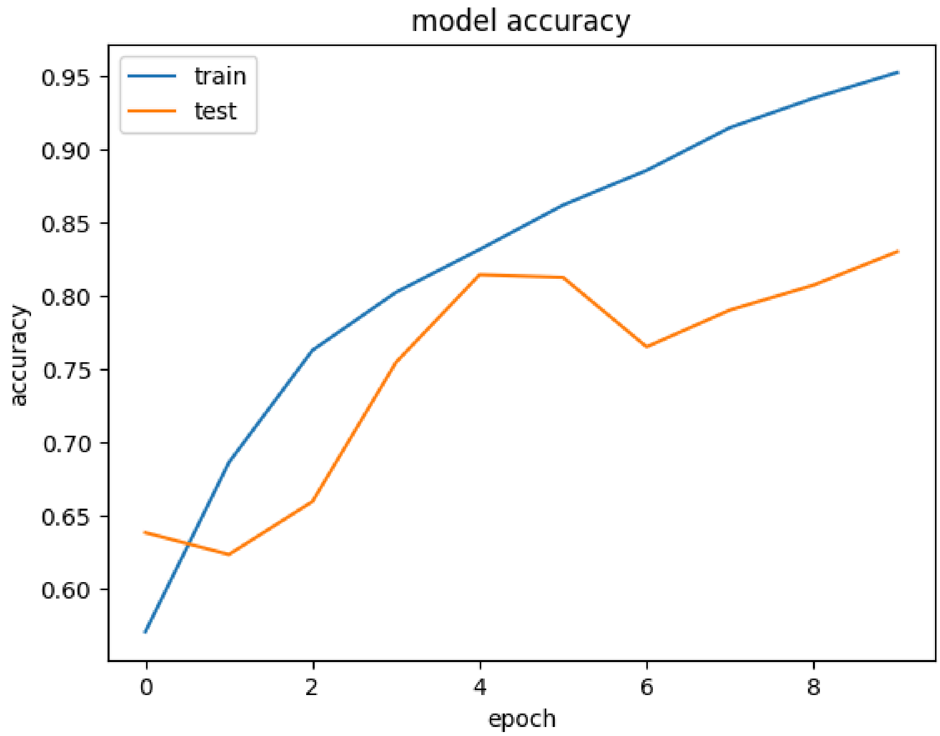
Plot Graphs:

```
import matplotlib.pyplot as plt

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```




```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



Prediction on unseen data

- Upload image on colab

```
import cv2
```

```
test_img = cv2.imread(r"/kaggle/images.jfif")
plt.imshow(test_img)
```


```
test_img.shape

(175, 289, 3)
```

Resize:

```
test_img= cv2.resize(test_img, (256,256))
```

Reshape:

```
test_input = test_img.reshape((1,256,256,3))
```

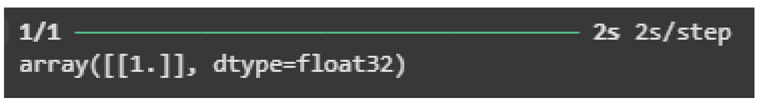
`1` → Batch size

`3` → 3 Layers (RGB)

`256,256` → Size of the image

Predict:

```
model.predict(test_input)
```

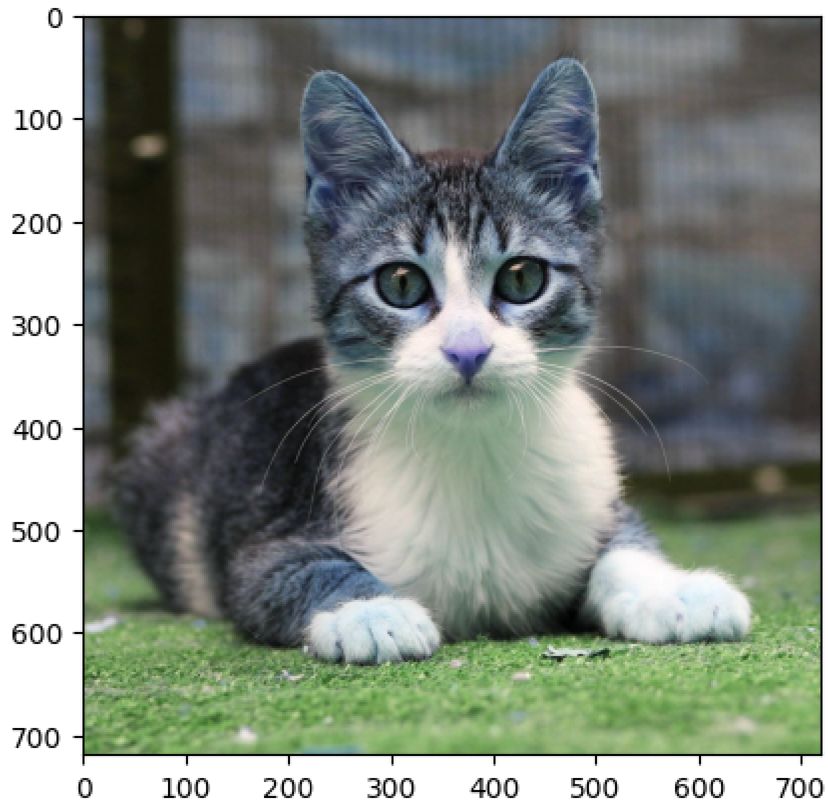


0= Cat

1 = Dog

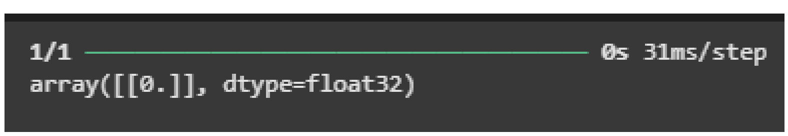
Prediction 2:

```
test_img2 = cv2.imread('/kaggle/cut cat serhio 02-1813×1811-720×719.jpg')
plt.imshow(test_img2)
```




```
test_img2= cv2.resize(test_img2, (256,256))
test_input2= test_img2.reshape((1,256,256,3))
```

```
model.predict(test_input2)
```



```
1/1 ──────────────────────────── 0s 31ms/step
array([[0.]], dtype=float32)
```

0 = Cat