😊 Convolutional Neural Network (CNN)

A **Convolutional Neural Network (CNN)** is a type of neural network specifically designed to process and analyze visual data, like images. Think of it as a "smart camera" that learns to recognize patterns and objects in images, such as faces, animals, or even handwritten digits.

Understanding CNNs:

Imagine you are looking at a photo, trying to figure out if it shows a cat or a dog. At first glance, you might notice individual features like ears, fur, or eyes. Your brain processes these details and compares them to what it already knows about cats and dogs to make a decision. Similarly, a CNN breaks an image down into small parts, identifies patterns like edges, textures, or shapes, and then combines these patterns to understand what the image is.

How CNNs Work:

CNNs work by breaking images down into smaller parts and using a process called **convolution** to learn patterns. Here's how it happens:

1. **Convolution Layer (Pattern Detection)**:
    - Imagine looking at small patches of an image, say, a 3x3 section of pixels. A CNN uses a filter (a small matrix of numbers) that scans over the image and checks for patterns, like edges or lines. This is like a magnifying glass moving over the image, identifying important details.
    - The result is a new simplified version of the image, called a **feature map**, that highlights areas where certain patterns were detected.

Example: Think of it like a stencil that highlights certain features of a picture, like the outline of a face, without looking at every detail of the whole picture.

2. **Pooling Layer (Simplification)**:
    - After detecting patterns, the CNN simplifies the data by taking the most important information. It reduces the size of the feature map by selecting the most relevant details (this is called **max pooling**).
    - Pooling helps the network focus on the big picture, so it can recognize patterns more effectively without being bogged down by minor details.

Example: Imagine taking a very detailed drawing of a cat and shrinking it. Even though you lose some details, the drawing still clearly looks like a cat. Pooling helps in this way—by simplifying the image without losing key information.

3. **Fully Connected Layer (Decision-Making)**:
    - After the image has been simplified and patterns have been detected, the CNN passes the information to the **fully connected layer**. This part of the network works like a traditional neural network, where every neuron is connected to all the neurons in the previous layer.

- The fully connected layer looks at all the learned patterns and makes the final decision (for example, "this is a cat" or "this is a dog").

Example: After breaking down an image into patterns like eyes, ears, and fur, the fully connected layer combines all these features to decide if it's looking at a cat or a dog.

Key Features of CNNs:

1. **Filters (Kernels)**:
   - These are small grids of numbers (like 3x3 or 5x5) that slide over the image to detect specific patterns like edges, corners, or textures.
2. **Strides**:
   - The stride is the step size that the filter moves as it slides over the image. If the stride is 1, the filter moves pixel by pixel. A larger stride skips more pixels and reduces the size of the feature map.
3. **Padding**:
   - Sometimes, to make sure the filter can scan the entire image, extra layers of pixels are added around the borders. This process is called padding and ensures that the output size is preserved.

Example of CNN in Action:

Task: Recognizing a Cat in a Photo

1. **Input Image**: A CNN starts with a photo of a cat.
2. **Convolution Layer**: The network looks for simple patterns like edges of the cat's ears or whiskers by applying filters.
3. **Pooling Layer**: It simplifies these patterns, focusing on the most important details, like the shape of the face or the texture of the fur.
4. **More Convolution + Pooling**: As the network goes deeper, it starts to combine smaller patterns into more complex ones, like recognizing the cat's face or body.
5. **Fully Connected Layer**: Finally, all the detected patterns are combined, and the network makes the decision: "This is a cat."

Why CNNs are Powerful:

- **Automatic Feature Detection**: Unlike traditional methods where features (like edges or textures) had to be manually defined, CNNs automatically learn the best features from the data.
- **Spatial Hierarchies**: CNNs understand spatial hierarchies of features. In early layers, they detect simple features (like edges); in later layers, they detect more complex features (like shapes or objects).

- **Works Well for Visual Data**: CNNs are especially powerful for image and video recognition tasks, such as facial recognition, object detection, and even medical image analysis.

Real-Life Examples of CNNs:

1. **Self-driving cars**: CNNs help the car's system recognize traffic signs, pedestrians, and obstacles by analyzing the video feed in real-time.
2. **Medical Imaging**: CNNs are used to analyze X-rays or MRI scans to detect anomalies, like tumors or fractures, with high accuracy.
3. **Social Media**: When platforms like Facebook or Instagram automatically tag people in photos, they use CNNs to identify faces.

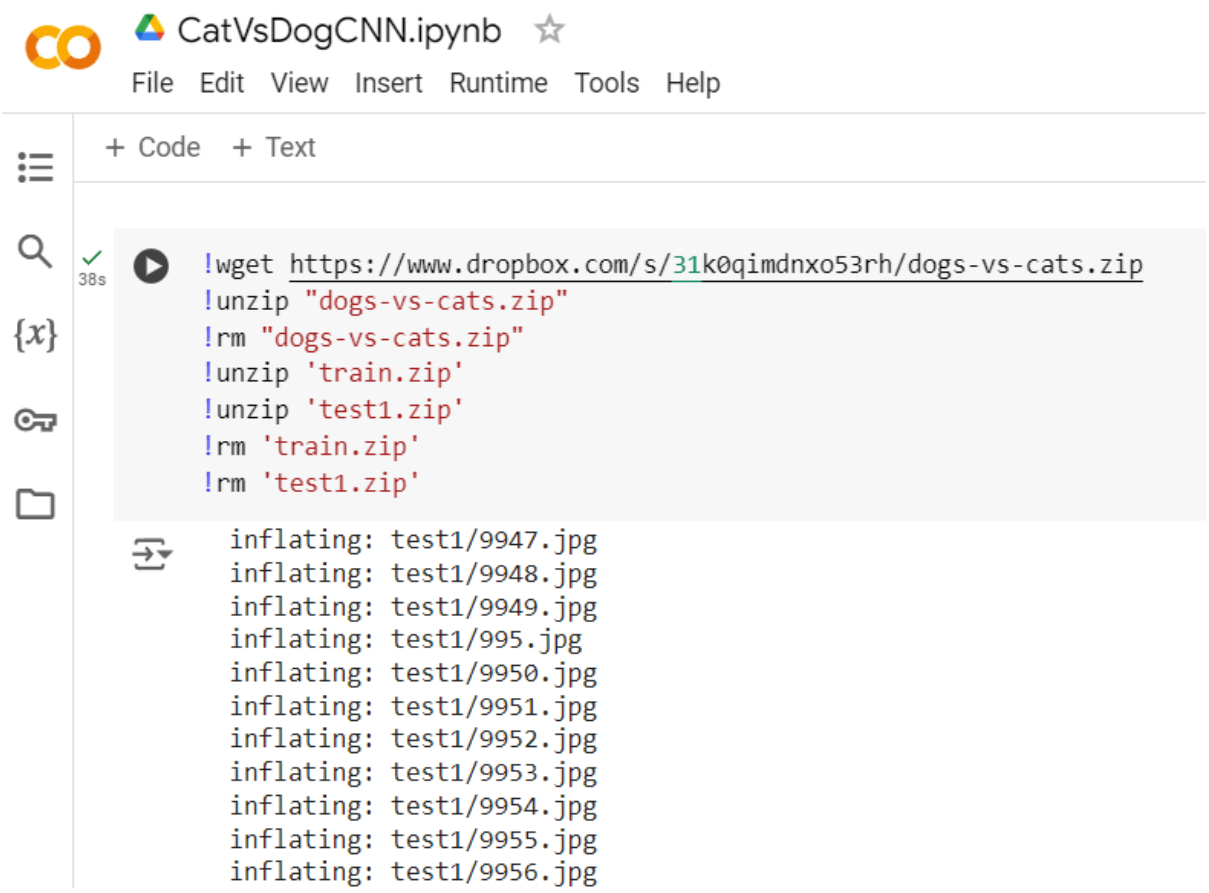
Summary

- A **Convolutional Neural Network (CNN)** is like a detective that breaks down an image into smaller parts, looks for patterns, and then makes a decision based on those patterns.
- CNNs are especially good at tasks involving visual data, such as recognizing objects in photos.
- They consist of layers like **convolution**, which detects patterns, and **pooling**, which simplifies data, followed by a decision-making layer.

In short, CNNs are the go-to tool in deep learning for tasks involving images, making them incredibly important for modern technologies like self-driving cars, facial recognition, and even medical diagnostics.

😄 To begin with the Lab:

1. For this lab we are going to use Google Colab instead of Jupyter Notebook because we are going to load a data from zip file which is uploaded on Dropbox.
2. In this code, a command is used to download a ZIP file containing a dataset called "dogs vs. cats" from Dropbox. After downloading, the ZIP file is unzipped to extract its contents. Then, the original ZIP file is deleted to free up space.
3. Next, two additional ZIP files, train.zip and test1.zip, are unzipped to extract their contents as well. Finally, these two ZIP files are removed to clean up after the extraction



4. In this code, several libraries are imported for data manipulation and visualization. NumPy and Pandas are used for numerical and data handling tasks, while Matplotlib and Seaborn are used for creating visualizations.
5. The TensorFlow Keras module is imported for image processing tasks, including loading images and augmenting datasets with the ImageDataGenerator. Additionally, Scikit-learn is used for splitting the dataset into training and testing subsets.
6. Several constants are defined for image processing: the width and height of the images are set to 128 pixels, and the images are expected to have three color channels (RGB). The size of the images is specified as a tuple, which will be useful for preparing the data for training a model.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.utils import to_categorical,load_img
from sklearn.model_selection import train_test_split
import os
```

```
[3] Image_width = 128
    Image_height = 128
    Image_size = (Image_width , Image_height)
    Image_channel = 3

    Image_rgb_size = (Image_width ,Image_height , 3 )
```

7. In this code, the filenames of all files in the '/content/train' directory are listed using os.listdir(). An empty list called categories is created to hold the categories of the images.
8. The code then loops through each filename, extracting the category by splitting the filename at the period and taking the first part (which typically represents the label, like "dog" or "cat"). Each category is appended to the categories list.
9. Finally, a Pandas DataFrame is created with two columns: one for the filenames and one for the corresponding categories. The data.head() function displays the first few rows of this DataFrame, providing a quick overview of the filenames and their associated categories.

```
[4] filenames = os.listdir('/content/train')
    categories = []
    for filename in filenames:
        category = filename.split('.')[0]
        categories.append(category)
    data = pd.DataFrame({'filename' : filenames , 'category' : categories})
    data.head()
```

	filename	category
0	dog.1755.jpg	dog
1	cat.2150.jpg	cat
2	dog.10119.jpg	dog
3	dog.5534.jpg	dog
4	dog.20.jpg	dog

Next steps: Generate code with data | View recommended plots | New interactive sheet

10. In this code, the value_counts() method is used on the 'category' column of the DataFrame to count how many images belong to each category (e.g., dogs and cats). This gives an overview of the distribution of the dataset.
11. Next, the code randomly selects one filename from the list of filenames using random.choice(). It then loads the corresponding image using load_img() and displays it with plt.imshow(). Finally, plt.show() is called to render the image on the screen.

```
data['category'].value_counts()
```

category	count
dog	12500
cat	12500

dtype: int64

```
import random
sample = random.choice(filenames)
image = load_img(f'train/{sample}')
plt.imshow(image)
plt.show()
```

12. In this code, the dataset is split into training and validation sets using train_test_split() from Scikit-learn. The test_size parameter is set to 0.2, meaning 20% of the data will be allocated for validation, while 80% will be used for training. The random_state parameter ensures that the split is reproducible.
13. After splitting, the indices of both DataFrames are reset using reset_index(drop=True) to create clean, sequential indices.
14. The shapes of the training and validation DataFrames are displayed, indicating how many samples are in each set. Finally, train_df.head() is used to show the first few rows of the training DataFrame for a quick preview of its contents.

```
[7] train_df , val_df = train_test_split(data , test_size = 0.2 , random_state = 42)
    train_df = train_df.reset_index(drop = True)
    val_df = val_df.reset_index(drop = True)
```

```
[8] train_df.shape , val_df.shape
```

```
((20000, 2), (5000, 2))
```

```
[9] train_df.head()
```

	filename	category
0	cat.9282.jpg	cat
1	cat.5665.jpg	cat
2	cat.885.jpg	cat
3	dog.10049.jpg	dog
4	dog.4533.jpg	dog

Next steps: Generate code with train_df | View recommended plots | New interactive sheet

15. In this code, several parameters are set for training a neural network. The batch_size is defined as 32, meaning that the model will process 32 images at a time. The epochs variable is set to 7, indicating the number of times the model will go through the entire training dataset. The total number of training and validation samples is retrieved from the shapes of the respective DataFrames.
16. An instance of ImageDataGenerator is created for data augmentation during training. This generator applies various transformations to the images, such as random rotations, rescaling pixel values, shear transformations, zooming, horizontal flips, and shifts in width and height. These augmentations help improve the model's robustness by providing diverse training examples.
17. Finally, the flow_from_dataframe method is used to create a data generator (train_generator) that pulls images from the training DataFrame (train_df). It specifies the path to the images, which column contains the filenames, which column contains the categories, the target size for resizing images, the classification mode (categorical for multi-class classification), and the batch size. This generator will yield batches of augmented images during training.

```
[10] batch_size  = 32
     epochs = 7
     total_train = train_df.shape[0]
     total_validate = val_df.shape[0]
```

```
[11] train_dategen = ImageDataGenerator(rotation_range = 15 ,
                                        rescale = 1.0/255 ,
                                        shear_range = 0.1,
                                        zoom_range = 0.2 ,
                                        horizontal_flip = True ,
                                        width_shift_range = 0.1 ,
                                        height_shift_range = 0.1
                                        )
```

```
[12] train_generator = train_dategen.flow_from_dataframe(
         train_df,
         "/content/train",
         x_col='filename',
         y_col='category',
         target_size=Image_size,
         class_mode='categorical',
         batch_size=batch_size
     )
```

```
Found 20000 validated image filenames belonging to 2 classes.
```

```
[16] validation_datagen = ImageDataGenerator(rescale = 1./255)
     val_generator = validation_datagen.flow_from_dataframe(
         val_df,
         "/content/train",
         x_col='filename',
         y_col='category',
         target_size=Image_size,
         class_mode='categorical',
         batch_size=batch_size
     )
```

```
Found 5000 validated image filenames belonging to 2 classes.
```

18. In this code, a convolutional neural network (CNN) is built using Keras. Here's a breakdown of the model architecture:
    a. **Model Initialization**: A sequential model is created, which allows layers to be stacked linearly.
    b. **Convolutional Layers**:

- Three convolutional layers are added in sequence, each with 10 filters and a kernel sizeof 3×33 \times 33×3. These layers learn features from the input images.
- The input shape is specified for the first layer, indicating the images will be 128×128128 \times 128128×128 pixels with 3 color channels (RGB).

c. **MaxPooling Layers**: After every three convolutional layers, a MaxPooling layer is added to downsample the feature maps, reducing their spatial dimensions while retaining important features.

d. **Flattening**: After the last pooling layer, the output is flattened into a one-dimensional array, preparing it for the fully connected (dense) layers.

e. **Dense Layers**:
   - A dense layer with 20 neurons and a ReLU activation function is added, which introduces non-linearity.
   - Finally, a dense output layer with 2 neurons and a softmax activation function is added, suitable for multi-class classification (in this case, distinguishing between two categories: dogs and cats).

f. **Model Summary**: The model.summary() function displays the architecture of the model, including the number of parameters in each layer and the total number of parameters in the model.

This architecture is suitable for image classification tasks, leveraging convolutional layers to extract features and dense layers to make predictions.

```
from tensorflow.keras.layers import GlobalAveragePooling2D
from tensorflow.keras.models import Model
from  tensorflow.keras.models import Sequential
from  tensorflow.keras.layers import Conv2D , MaxPooling2D , Dropout , Flatten , Dense , Activation , BatchNormalization

model = Sequential()
model.add(Conv2D(filters=10, kernel_size=(3,3), strides=(1, 1),padding='valid', input_shape=(128,128,3)))
model.add(Conv2D(filters=10, kernel_size=(3,3), strides=(1, 1),padding='valid'))
model.add(Conv2D(filters=10, kernel_size=(3,3), strides=(1, 1),padding='valid'))
model.add(MaxPooling2D())

model.add(Conv2D(filters=10, kernel_size=(3,3), strides=(1, 1),padding='valid'))
model.add(Conv2D(filters=10, kernel_size=(3,3), strides=(1, 1),padding='valid'))
model.add(Conv2D(filters=10, kernel_size=(3,3), strides=(1, 1),padding='valid'))
model.add(MaxPooling2D())

model.add(Flatten())
model.add(Dense(20,activation='relu'))
model.add(Dense(2,activation='softmax'))
model.summary()
```

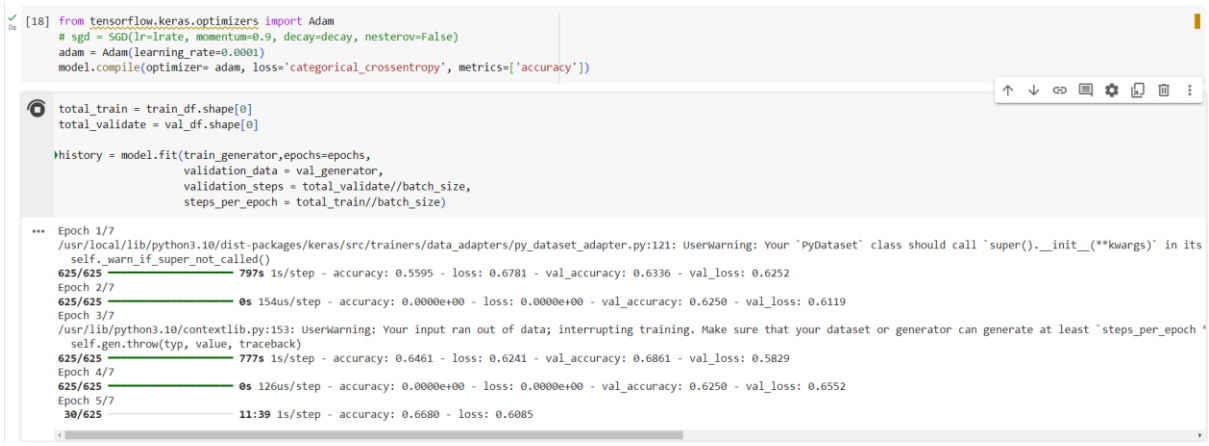
```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 126, 126, 10)	280
conv2d_7 (Conv2D)	(None, 124, 124, 10)	910
conv2d_8 (Conv2D)	(None, 122, 122, 10)	910
max_pooling2d_2 (MaxPooling2D)	(None, 61, 61, 10)	0
conv2d_9 (Conv2D)	(None, 59, 59, 10)	910
conv2d_10 (Conv2D)	(None, 57, 57, 10)	910
conv2d_11 (Conv2D)	(None, 55, 55, 10)	910
max_pooling2d_3 (MaxPooling2D)	(None, 27, 27, 10)	0
flatten_1 (Flatten)	(None, 7290)	0
dense_2 (Dense)	(None, 20)	145,820
dense_3 (Dense)	(None, 2)	42

Total params: 150,692 (588.64 KB)
Trainable params: 150,692 (588.64 KB)
Non-trainable params: 0 (0.00 B)

19. In this code, the Adam optimizer is imported and initialized with a learning rate of 0.0001. The model is then compiled using this optimizer, specifying the loss function as categorical crossentropy (suitable for multi-class classification) and the metric to track as accuracy.
20. Next, the total number of training and validation samples is retrieved from the shapes of the respective DataFrames.
    The model is trained using the fit() method, which takes several parameters:
    - **train_generator**: The data generator for the training set.
    - **epochs**: The number of times to iterate over the entire training dataset.
    - **validation_data**: The generator for the validation set.
    - **validation_steps**: The number of steps to validate after each epoch, calculated based on the number of validation samples and batch size.
    - **steps_per_epoch**: The number of steps to take for each epoch during training, also based on the total number of training samples and batch size.
21. The training process will run for the specified number of epochs, during which the model learns from the training data and is evaluated on the validation data. The training history, which includes metrics like loss and accuracy over the epochs, is stored in the history variable.

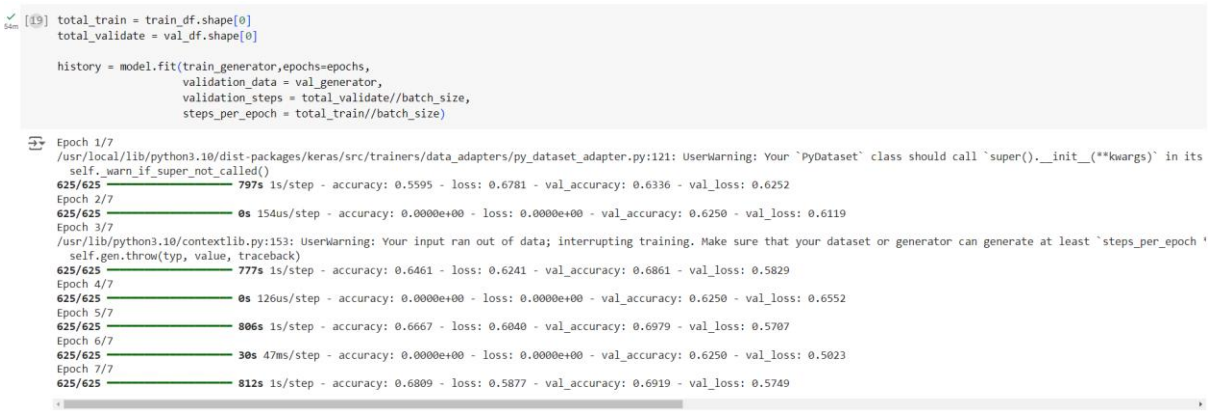


```
[18] from tensorflow.keras.optimizers import Adam
     # sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
     adam = Adam(learning_rate=0.0001)
     model.compile(optimizer= adam, loss='categorical_crossentropy', metrics=['accuracy'])

     total_train = train_df.shape[0]
     total_validate = val_df.shape[0]

     history = model.fit(train_generator,epochs=epochs,
                         validation_data = val_generator,
                         validation_steps = total_validate//batch_size,
                         steps_per_epoch = total_train//batch_size)

     Epoch 1/7
     /usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its
       self._warn_if_super_not_called()
     625/625 ━━━━━━━━━━━━━━━━━━━━ 797s 1s/step - accuracy: 0.5595 - loss: 0.6781 - val_accuracy: 0.6336 - val_loss: 0.6252
     Epoch 2/7
     625/625 ━━━━━━━━━━━━━━━━━━━━ 0s 154us/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.6250 - val_loss: 0.6119
     Epoch 3/7
     /usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch `
       self.gen.throw(typ, value, traceback)
     625/625 ━━━━━━━━━━━━━━━━━━━━ 777s 1s/step - accuracy: 0.6461 - loss: 0.6241 - val_accuracy: 0.6861 - val_loss: 0.5829
     Epoch 4/7
     625/625 ━━━━━━━━━━━━━━━━━━━━ 0s 126us/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.6250 - val_loss: 0.6552
     Epoch 5/7
      30/625 ━━━━━━━━━━━━━━━━━━━━ 11:39 1s/step - accuracy: 0.6680 - loss: 0.6085
```



```
[19] total_train = train_df.shape[0]
     total_validate = val_df.shape[0]

     history = model.fit(train_generator,epochs=epochs,
                         validation_data = val_generator,
                         validation_steps = total_validate//batch_size,
                         steps_per_epoch = total_train//batch_size)

     Epoch 1/7
     /usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its
       self._warn_if_super_not_called()
     625/625 ━━━━━━━━━━━━━━━━━━━━ 797s 1s/step - accuracy: 0.5595 - loss: 0.6781 - val_accuracy: 0.6336 - val_loss: 0.6252
     Epoch 2/7
     625/625 ━━━━━━━━━━━━━━━━━━━━ 0s 154us/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.6250 - val_loss: 0.6119
     Epoch 3/7
     /usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch `
       self.gen.throw(typ, value, traceback)
     625/625 ━━━━━━━━━━━━━━━━━━━━ 777s 1s/step - accuracy: 0.6461 - loss: 0.6241 - val_accuracy: 0.6861 - val_loss: 0.5829
     Epoch 4/7
     625/625 ━━━━━━━━━━━━━━━━━━━━ 0s 126us/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.6250 - val_loss: 0.6552
     Epoch 5/7
     625/625 ━━━━━━━━━━━━━━━━━━━━ 806s 1s/step - accuracy: 0.6667 - loss: 0.6040 - val_accuracy: 0.6979 - val_loss: 0.5707
     Epoch 6/7
     625/625 ━━━━━━━━━━━━━━━━━━━━ 30s 47ms/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.6250 - val_loss: 0.5023
     Epoch 7/7
     625/625 ━━━━━━━━━━━━━━━━━━━━ 812s 1s/step - accuracy: 0.6809 - loss: 0.5877 - val_accuracy: 0.6919 - val_loss: 0.5749
```

22. In this code, a pre-trained ResNet50 model from Keras is used as the base for building a new image classification model. Here's a breakdown of the steps:
    a. **Load Pre-trained Model**: The ResNet50 model is loaded with weights pre-trained on the ImageNet dataset. The include_top=False parameter indicates that the top classification layers are not included, allowing for customization.
    b. **Global Average Pooling**: The output from the ResNet50 base is passed to a GlobalAveragePooling2D layer. This layer reduces the spatial dimensions of the feature maps to a single value per feature, creating a more compact representation.
    c. **Dropout Layer**: A Dropout layer with a rate of 0.7 is added. This layer randomly sets 70% of the input units to zero during training, which helps prevent overfitting.
    d. **Output Layer**: A dense layer with 2 neurons and a softmax activation function is added for classification into two categories (e.g., dogs and cats).
    e. **Model Creation**: A new model is created using the Keras Model class, specifying the inputs from the base model and the outputs from the final predictions layer.
    f. **Compile Model**: The model is compiled using the Adam optimizer with a learning rate of 0.0001, the categorical crossentropy loss function, and accuracy as the evaluation metric.
    g. This architecture leverages transfer learning, taking advantage of the features learned by ResNet50 to improve performance on the specific task of classifying images of dogs and cats.

Transfer Learning

```
[22] import tensorflow as tf
     base = tf.keras.applications.resnet50.ResNet50(weights = 'imagenet' , include_top = False ,
                                                    input_shape = Image_rgb_size)

     x = base.output
     x = GlobalAveragePooling2D()(x)
     x = Dropout(0.7)(x)
     predictions = Dense(2, activation= 'softmax')(x)
     model = Model(inputs = base.input, outputs = predictions)


     from tensorflow.keras.optimizers import Adam
     # sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
     adam = Adam(learning_rate=0.0001)
     model.compile(optimizer= adam, loss='categorical_crossentropy', metrics=['accuracy'])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

23. In this code, several training callbacks are defined:
    a. **EarlyStopping**: Stops training if validation accuracy doesn't improve for 10 epochs to prevent overfitting.
    b. **ReduceLROnPlateau**: Reduces the learning rate by half if validation accuracy doesn't improve for 2 epochs, with a minimum learning rate of 0.00001.
    c. **ModelCheckpoint**: Saves the model's weights when validation accuracy improves, using a naming format that includes the epoch number and accuracy.
    d. The callbacks are combined into a list for use during model training.

```
[25] from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint

     # Define callbacks
     earlystop = EarlyStopping(patience=10)
     learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy',
                                                 patience=2,
                                                 verbose=1,
                                                 factor=0.5,
                                                 min_lr=0.00001)
     mc = ModelCheckpoint(
         "resnet_v1_{epoch:02d}_{val_accuracy:.3f}.keras",  # Change the file extension to .keras
         monitor="val_accuracy",
         save_best_only=True,
         mode='max'
     )

     callbacks = [earlystop, learning_rate_reduction, mc]
```

24. In this code, the model is trained using the fit() method. The training data is provided by the train_generator, and the number of training iterations is specified by the epochs parameter. Validation data is supplied through the validation_data argument, allowing for evaluation during training.
25. The validation_steps parameter defines how many steps to take for validation based on the total number of validation samples. Similarly, steps_per_epoch sets the number of steps for each epoch, calculated from the total training samples. The callbacks argument includes previously defined callbacks to enhance the training process. As a result, the model begins learning from the training data while being evaluated on the validation data.

```
[26] history = model.fit(
        train_generator ,
        epochs = epochs ,
        validation_data=val_generator,
        validation_steps=total_validate//batch_size,
        steps_per_epoch=total_train//batch_size,
        callbacks=callbacks
    )
```

```
Epoch 1/7
625/625 ━━━━━━━━━━━━━━━━━━━━ 6258s 10s/step - accuracy: 0.8703 - loss: 0.3418 - val_accuracy: 0.7568 - val_loss: 0.4824 - learning_rate: 1.0000e-04
Epoch 2/7
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch *
  self.gen.throw(typ, value, traceback)
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.8750 - val_loss: 0.4142 - learning_rate: 1.0000e-04
Epoch 3/7
 43/625 ━━━━━━━━━━━━━━━━━━━━ 1:29:45 9s/step - accuracy: 0.9617 - loss: 0.0918
```

26. In this code, the filenames of the images in the test directory are listed and stored in a DataFrame called test. The total number of test samples is calculated.
27. An ImageDataGenerator is created to preprocess the test images by rescaling pixel values. A test generator is then created to load images from the DataFrame, specifying the target size, batch size, and ensuring the images are not shuffled.
28. An example image is loaded and displayed to visualize it. A dictionary is created to map numerical predictions to labels (Cats and Dogs). The image is processed for prediction, and the model generates a prediction for it, which is printed as the predicted category.
29. Next, the number of steps required for prediction is calculated based on the total number of test samples and the batch size. The model predicts categories for all images in the test set using the test generator. The predicted categories are assigned to the test DataFrame, and the first few rows of the updated DataFrame are displayed, showing the filenames and their predicted categories.

```
[27] test_filenames = os.listdir('/content/test1')
     test = pd.DataFrame({'filename' : test_filenames})
     nb_samples = test.shape[0]
```

```
[35] test.head()
```

	filename
0	7828.jpg
1	5596.jpg
2	6766.jpg
3	5759.jpg
4	4298.jpg

Next steps: Generate code with test | View recommended plots | New interactive sheet

0s

```
test_gen = ImageDataGenerator(rescale=1./255)
test_generator = test_gen.flow_from_dataframe(
    test ,
    '/content/test1' ,
    x_col='filename',
    y_col=None,
    class_mode=None,
    target_size=Image_size,
    batch_size=batch_size,
    shuffle=False
    )
```

```
Found 12500 validated image filenames.
```

2s

```
img_id = '1366.jpg'
image = load_img(f'/content/test1/{img_id}')
plt.imshow(image)
```

```
<matplotlib.figure.AxesImage at 0x78c5379c12a0>
```

```
[30] from tensorflow.keras.applications.resnet import preprocess_input
```

```
[32] labels = {0:'Cats',1:'Dogs'}
```

```
img_id = '1366.jpg'
image = load_img(f'/content/test1/{img_id}')
plt.imshow(image)

path = f'/content/test1/{img_id}'
img = load_img(path, target_size=Image_size)
import numpy as np
x = np.array(img)
X = np.array([x])
X = preprocess_input(X)
pred = model.predict(X)
print(f"Predicton from model is {labels[pred[0].argmax()]}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 89ms/step
Predicton from model is Dogs
```

```python
import numpy as np

# Calculate the number of steps and convert it to an integer
steps = int(np.ceil(nb_samples / batch_size))

# Predict using the test generator
predict = model.predict(test_generator, steps=steps)

# Assign categories based on predictions
test['category'] = np.argmax(predict, axis=-1)

# Display the first few rows of the test dataframe
test.head()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121:
  self._warn_if_super_not_called()
94/391 ━━━━━━━━━━━━━━━━━━━━ 9:53 2s/step
```