# LIBRARIES

**TensorFlow**: TensorFlow is an open-source library developed by Google for numerical computation and machine learning.

**Keras**: Keras is an API for building and training deep learning models. It's integrated within TensorFlow.

**TensorFlow and Keras**:

* TensorFlow provides the backend for performing the computations required for deep learning models.
* Keras is used to create and manage the structure of neural networks in an intuitive and user-friendly way.

· **ImageDataGenerator**: This is a class in Keras used for real-time data augmentation. It allows you to generate batches of tensor image data with real-time data augmentation.

It applies random transformations like rotation, zooming, flipping, etc., to the training images, which helps in increasing the diversity of the training data and improves the model's generalization ability.

# **Layers**

* **Conv2D**: A 2D convolutional layer, useful for processing image data.
* **MaxPooling2D**: A layer that performs max pooling operation on 2D data, reducing its dimensionality and helping to reduce overfitting.
* **Flatten**: Flattens the input, transforming a multi-dimensional input into a single-dimensional vector.
* **Dense**: A fully connected neural network layer.
* **Dropout**: A layer that randomly sets a fraction of input units to 0 at each update during training time, which helps prevent overfitting.

1. · **NumPy**: A library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

2.  **Conv2D Layer**: This layer applies a convolution operation to the input image, passing over different regions of the image with a set of filters. It is commonly used to extract features like edges, textures, and shapes from images.

3. **MaxPooling2D Layer**:

* This layer performs down-sampling by dividing the input into rectangular regions and taking the maximum value from each region. It reduces the spatial dimensions of the input and helps in reducing the computational load.

4. **Flatten Layer**:

* This layer converts the multi-dimensional output from the convolutional and pooling layers into a single-dimensional vector, which can then be fed into fully connected layers.

5. **Dense Layer**:

* This layer is a fully connected layer where each neuron receives input from all the neurons in the previous layer. It is typically used at the end of the network for classification tasks.

6. **Dropout Layer**:

* This layer randomly sets a fraction of the input units to zero at each update during training time, which helps prevent overfitting by ensuring that the model does not become too dependent on any single neuron.

# CNN FOR IMAGE CLASSIFICATION - COMPLETE FLOW

**Input Layer**:

* This is where the input images are fed into the network. Typically, images are represented as multi-dimensional arrays (e.g., height x width x channels).

**Convolutional Layers (Conv2D)**:

* These layers apply a set of convolutional filters to the input image to extract features such as edges, textures, and patterns. Each filter produces a feature map.

**Activation Function**:

* Often applied right after a convolutional layer. A common activation function is ReLU (Rectified Linear Unit), which introduces non-linearity to the model.

**Pooling Layers (MaxPooling2D)**:

* These layers reduce the spatial dimensions (height and width) of the feature maps generated by the convolutional layers. Max pooling is a common pooling operation that takes the maximum value in each patch of the feature map.

**Flatten Layer**:

* After several convolutional and pooling layers, the multi-dimensional feature maps are flattened into a single-dimensional vector. This vector will be used as the input for the fully connected layers.

**Fully Connected Layers (Dense)**:

* These layers are traditional neural network layers where each neuron is connected to every neuron in the previous layer. These layers perform the final classification based on the features extracted by the convolutional and pooling layers.

**Dropout Layer (Dropout)**:

* Often used after fully connected layers to randomly set a fraction of the input units to zero at each update during training time. This helps prevent overfitting by ensuring the model does not rely too heavily on any particular neurons.

**Output Layer (Dense)**:

* The final fully connected layer typically uses a softmax activation function (for multi-class classification) or a sigmoid activation function (for binary classification). This layer outputs the probabilities of each class.

# Flow of a typical convolutional neural network (CNN) for image classification:

The CNN begins with Conv2D layers that apply filters to extract features from input images, with each layer typically followed by ReLU activation to introduce non-linearity. After each Conv2D layer, MaxPooling2D layers are applied to downsample the output, reducing spatial dimensions(width and height) and aiding in controlling overfitting by focusing on the most important features. Once the feature extraction and downsampling phases are complete, the Flatten layer reshapes the 2D output from the last pooling layer into a 1D vector. This flattened vector is then fed into Dense layers, which are fully connected layers where each neuron connects to every neuron in the previous layer, often using ReLU activation to further capture complex patterns in the data. Dropout layers are intermittently inserted into the dense layers during training, randomly setting a fraction of input units to zero to prevent overfitting and improve generalization. This structured approach of feature extraction, downsampling, flattening, and dense classification layers, combined with dropout regularization, forms a robust framework for training CNNs on image classification tasks.

# CHANNELS IN TENSORS

When working with image data in deep learning frameworks like TensorFlow and Keras, images are typically represented as multi-dimensional arrays (tensors). The dimensions of these arrays include:

1. **Batch Size**: Number of images in a batch.
2. **Height**: Height of the images.
3. **Width**: Width of the images.
4. **Channels**: Number of channels in each image (e.g., 1 for grayscale, 3 for RGB).

### **Example: Tensor Shape**

* For a batch of RGB images:
  + Shape: (batch\_size, height, width, 3)

### **Example: Channels in Convolutional Layers**

In convolutional layers (Conv2D), the number of channels corresponds to the depth of the input tensor and the number of filters applied. For example:

* If the input image has a shape of (64, 64, 3), it means the image is 64x64 pixels with 3 channels (RGB).
* Applying a Conv2D layer with 32 filters will produce an output tensor with a shape of (64, 64, 32), where 32 is the number of filters (new channels).

NOTE : Non-linearity is added to neural networks through activation functions to allow the model to learn complex patterns and relationships in the data. Without non-linearity, the model would be limited to learning only linear transformations, which are insufficient for most real-world tasks

# USING 3\*3 FILTERS:

The size of 3x3 for filters (also called kernels) in a Conv2D layer is a design choice that has been found to work well in many convolutional neural networks (CNNs). Let's delve into why a 3x3 filter size is commonly used:

### **Reasons for Using 3x3 Filters**

1. **Receptive Field**:
   * A 3x3 filter has a small receptive field, meaning it looks at a small region of the input image at a time. This allows the network to capture fine details and local patterns, such as edges and textures.
   * When multiple 3x3 layers are stacked, the receptive field increases, allowing the network to capture larger patterns and more complex features in a hierarchical manner.
2. **Efficiency**:
   * 3x3 filters are computationally efficient. They require fewer parameters and less computation compared to larger filters, which helps in reducing the model's complexity and the risk of overfitting.
   * Smaller filters mean less computational cost during both the forward pass (inference) and the backward pass (training).
3. **Flexibility**:
   * By stacking multiple 3x3 convolutional layers, the network can achieve the same effect as using larger filters, but with more non-linearity and flexibility.
   * For instance, two consecutive 3x3 convolutions have an effective receptive field of 5x5, and three consecutive 3x3 convolutions have an effective receptive field of 7x7.
4. **Empirical Success**:
   * The use of 3x3 filters has been empirically validated by successful architectures like VGGNet, ResNet, and others. These architectures have demonstrated state-of-the-art performance in various image recognition tasks.

### **Comparison with Larger Filters**

* **Larger Filters (e.g., 5x5, 7x7)**:
  + They cover a larger area of the input image in a single convolution operation.
  + They increase the number of parameters and computational cost.
  + They might capture more complex features in a single layer but at the expense of flexibility and increased risk of overfitting.
* **Smaller Filters (e.g., 1x1)**:
  + 1x1 convolutions are also used in modern architectures (e.g., Inception modules). They are mainly used for dimensionality reduction, combining features across channels, and adding non-linearity without affecting the spatial dimensions.

### **Example Calculation**

Here's an example to illustrate how stacking multiple 3x3 layers increases the receptive field:

* **Single 3x3 Layer**:
  + Receptive field: 3x3
* **Two Consecutive 3x3 Layers**:
  + Effective receptive field: 5x5
* **Three Consecutive 3x3 Layers**:
  + Effective receptive field: 7x7

# ARCHITECTURES FOR IMAGE CLASSIFICATION - WELL KNOWN

### **1. LeNet-5**

* **Developed by**: Yann LeCun et al.
* **Year**: 1998
* **Key Features**: One of the earliest convolutional neural networks designed for handwritten digit recognition (MNIST dataset).
* **Architecture Highlights**: Consists of two sets of convolutional and pooling layers, followed by three fully connected layers.

### **2. AlexNet**

* **Developed by**: Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton
* **Year**: 2012
* **Key Features**: Won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 by a significant margin.
* **Architecture Highlights**: Deep CNN with 8 layers (5 convolutional and 3 fully connected), ReLU activations, and dropout for regularization.

### **3. VGGNet**

* **Developed by**: Karen Simonyan and Andrew Zisserman
* **Year**: 2014
* **Key Features**: Known for its simplicity and use of small (3x3) convolution filters.
* **Architecture Highlights**: Deep CNN with 16 or 19 layers (VGG16, VGG19), stacked convolutional layers, and three fully connected layers at the end.

### **4. GoogLeNet (Inception v1)**

* **Developed by**: Christian Szegedy et al.
* **Year**: 2014
* **Key Features**: Introduced the Inception module, which uses filters of multiple sizes to capture different features.
* **Architecture Highlights**: 22 layers deep, extensive use of 1x1 convolutions to reduce dimensionality, and multiple Inception modules.

### **5. ResNet (Residual Networks)**

* **Developed by**: Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun
* **Year**: 2015
* **Key Features**: Introduced the concept of residual learning, allowing much deeper networks without suffering from vanishing gradients.
* **Architecture Highlights**: Networks with up to 152 layers, identity shortcut connections that bypass one or more layers.

### **6. Inception v3**

* **Developed by**: Christian Szegedy et al.
* **Year**: 2015
* **Key Features**: Improved version of GoogLeNet with refined Inception modules and factorized convolutions.
* **Architecture Highlights**: Efficient grid size reduction, deeper network, and use of label smoothing.

### **7. DenseNet (Densely Connected Convolutional Networks)**

* **Developed by**: Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger
* **Year**: 2017
* **Key Features**: Each layer is connected to every other layer in a feed-forward manner, promoting feature reuse.
* **Architecture Highlights**: Dense blocks with direct connections from any layer to all subsequent layers.

### **8. MobileNet**

* **Developed by**: Andrew G. Howard et al.
* **Year**: 2017
* **Key Features**: Designed for mobile and embedded vision applications, with a focus on model size and speed.
* **Architecture Highlights**: Uses depthwise separable convolutions to reduce the number of parameters and computations.

### **9. EfficientNet**

* **Developed by**: Mingxing Tan and Quoc V. Le
* **Year**: 2019
* **Key Features**: Balances network depth, width, and resolution using a compound scaling method.
* **Architecture Highlights**: Achieves better performance with fewer parameters by scaling all dimensions of depth, width, and resolution uniformly.

### **10. Vision Transformers (ViT)**

* **Developed by**: Alexey Dosovitskiy et al.
* **Year**: 2020
* **Key Features**: Applies transformer models, originally developed for NLP tasks, to image recognition by treating images as sequences of patches.
* **Architecture Highlights**: Uses self-attention mechanisms to capture global dependencies, achieving competitive performance with fewer inductive biases compared to CNNs.

# HYPERPARAMETERS

## **Outliers :** Remove outliers by setting a threshold

## **Batch size :** determines the number of samples that will be propagated through the network at one time.

**Memory Constraints**: Smaller batch sizes use less memory, which is important if you are training on hardware with limited resources (e.g., GPUs with limited VRAM).

**Stochastic Gradient Descent**: Smaller batch sizes often lead to better generalization and can help the model to escape local minima in the loss landscape due to the noisier gradient estimates.

| **Scenario** | **num\_classes** | **batch\_size** | **epochs** | **input\_shape** | **learning\_rate** |
| --- | --- | --- | --- | --- | --- |
| **Small Dataset, Simple Task** | 3 | 16-32 | 10-20 | (64, 64, 3) | 0.001 |
| **Small Dataset, Complex Task** | 3-5 | 8-16 | 20-50 | (128, 128, 3) | 0.0001 |
| **Medium Dataset, Simple Task** | 3-10 | 32-64 | 20-30 | (128, 128, 3) | 0.001 |
| **Medium Dataset, Complex Task** | 3-20 | 16-32 | 30-50 | (224, 224, 3) | 0.0001 |
| **Large Dataset, Simple Task** | 3-50 | 64-128 | 10-30 | (224, 224, 3) | 0.001 |
| **Large Dataset, Complex Task** | 3-100 | 32-64 | 30-100 | (224, 224, 3) | 0.0001 |
| **Transfer Learning** | 3-1000+ | 16-32 | 5-20 | (224, 224, 3) | 0.00001-0.0001 |
| **Resource-Constrained Environment** | 3-10 | 8-16 | 10-30 | (64, 64, 3) | 0.001 |
| **High-Performance Computing (HPC)** | 3-1000+ | 64-256 | 20-100 | (224, 224, 3) | 0.0001-0.001 |

## **High Epochs for Complex Tasks/Large Datasets**: Needed to fully learn and fine-tune intricate patterns over extensive data, ensuring the model generalizes well.

## **Low Epochs for Simple Tasks/Small Datasets**: Prevents overfitting, saves computational resources, and achieves sufficient convergence for less complex patterns.

## **Early Stopping**: Regardless of the task complexity or dataset size, using early stopping can help determine the optimal number of epochs. This technique monitors the model's performance on a validation set and stops training when performance no longer improves, thereby avoiding overfitting.

## **Learning Rate**: When using a high number of epochs, it is often coupled with a lower learning rate. This allows the model to make fine adjustments to the weights, leading to better convergence. For fewer epochs, a higher learning rate might be used to ensure the model converges quickly.

## Complex Task with Large Dataset

* **Reason for High Epochs**:
  + **Complex Patterns**: Complex tasks often involve learning intricate patterns and relationships within the data. More epochs allow the model to better learn and fine-tune these patterns.
  + **Large Dataset**: With a larger dataset, the model has more information to learn from. It typically requires more epochs to fully train on the extensive amount of data to generalize well.
  + **Overfitting Risk Management**: Although more epochs increase the risk of overfitting, large datasets tend to mitigate this risk by providing diverse examples, which helps the model generalize better.

## Simple Task with Small Dataset

* **Reason for Low Epochs**:
  + **Simplicity of Patterns**: Simple tasks involve less complex patterns and relationships, which can be learned quickly. Fewer epochs are often sufficient for the model to converge and perform well.
  + **Overfitting Risk**: Small datasets increase the risk of overfitting because the model can easily memorize the training data. Using fewer epochs helps prevent the model from overfitting to the small dataset.
  + **Resource Efficiency**: Training for fewer epochs is more resource-efficient, saving time and computational power when the task and data do not require extensive training.

## Simple Task with Large Dataset

* **Lower Epochs**:
  + **Simplicity of Patterns**: Simple tasks involve straightforward patterns that the model can learn quickly. The abundance of data in a large dataset allows the model to see enough examples of these simple patterns within fewer epochs.
  + **Faster Convergence**: With simple tasks, the model can converge faster because the relationships in the data are less complex. The large dataset provides sufficient exposure to these patterns, making prolonged training unnecessary.
  + **Risk of Overfitting**: Training for too many epochs on a simple task, even with a large dataset, can lead to overfitting. The model might start learning noise or irrelevant details instead of the underlying simple patterns.
  + **Resource Efficiency**: Fewer epochs mean reduced training time and computational resources, which is practical when the model has already achieved good performance.

## Complex Task with Small Dataset

* **Fewer Epochs**:
  + **Limited Data**: A small dataset lacks the diversity and quantity of examples needed for the model to generalize well.
  + **Overfitting Risk**: Training for too many epochs on a small dataset can lead to overfitting, where the model memorizes the training data instead of learning generalizable patterns.
  + **Model Complexity**: Despite the task's complexity, the limited amount of data restricts the model's ability to fully learn intricate patterns. Thus, fewer epochs are often sufficient to prevent overfitting and achieve optimal performance.

## **Optimizer**: Adam is a popular optimizer that adapts the learning rate during training based on how the gradients are changing. It combines techniques like momentum and adaptive learning rates for efficient training.

| **Optimizer** | **Typical Application** |
| --- | --- |
| **SGD (Stochastic Gradient Descent)** | Basic optimizer for training neural networks. |
| **Adam (Adaptive Moment Estimation)** | Generally well-suited for most tasks due to adaptive learning rates. |
| **RMSprop (Root Mean Square Propagation)** | Similar to Adam but with different moving average decay. |
| **Adagrad (Adaptive Gradient Algorithm)** | Well-suited for sparse data. |
| **Adadelta** | Extension of Adagrad that adapts learning rates over time. |
| **Adamax** | Variant of Adam based on infinity norm. |
| **Nadam (Nesterov-accelerated Adam)** | Adam with Nesterov momentum. |

## **Loss Function**: Categorical crossentropy is suitable when the target (y\_true) is in categorical format (one-hot encoded). It calculates the difference between the predicted probabilities (y\_pred) and the actual labels (y\_true).

## **Metrics**: Accuracy is a straightforward metric that measures the percentage of correctly classified examples out of the total examples.

## **sparse data** refers to datasets where most of the values are zeros or empty. Sparse data sets often have many missing values or very few non-zero values compared to the total number of possible values.

## **Learning rate** : 0.0001 -> for better convergence

# Augmenting image data during training using train\_datagen

**rescale=1.0/255**:

* This parameter scales pixel values of the images. Rescaling to a range of 0 to 1 is common practice in image processing and helps the model converge faster during training.

**rotation\_range=30**:

* Specifies the range (in degrees) for random rotations applied to images. In this case, images can be rotated randomly between -30 degrees and +30 degrees. This helps the model generalize better by exposing it to variations in orientation that may occur in real-world scenarios.

**width\_shift\_range=0.2** and **height\_shift\_range=0.2**:

* These parameters specify the maximum proportion of the image's width and height by which the image can be randomly shifted horizontally or vertically. A value of 0.2 means the image can be shifted up to 20% of its width or height. This augmentation technique helps the model learn translation invariance and improves robustness.

**shear\_range=0.2**:

* Controls the intensity of shearing transformation, where the image is distorted along an axis. A shear range of 0.2 means the shear angle will vary up to 20 degrees, enhancing the model's ability to recognize objects from different perspectives.

**zoom\_range=0.2**:

* Specifies the range for random zooming applied to images. A value of 0.2 allows the image to be zoomed in or out by up to 20%. This augmentation simulates different scales at which objects might appear in the real world and improves the model's robustness.

**horizontal\_flip=True**:

* Enables random horizontal flipping of images. This augmentation is useful for tasks where horizontal orientation does not affect the interpretation of the image, such as recognizing objects or scenes.

**fill\_mode='nearest'**:

* Determines how pixels are filled in when an image is transformed (e.g., rotated or shifted) and new pixels are introduced. 'nearest' fills new pixels with the nearest pixel value from the original image, which helps maintain the integrity of the image structure.

THEN WE USE test\_datagen

**rescale=1.0/255**:

* Similar to the train\_datagen, rescale scales the pixel values of images. In this case, each pixel value in the images will be divided by 255. This operation ensures that pixel values are in the range [0, 1], which is typical for neural network models using activation functions like sigmoid or tanh.

**Training Data**: Your training images are stored in a directory structure:

* /train/cat/cat1.jpg
* /train/cat/cat2.jpg
* /train/dog/dog1.jpg
* /train/dog/dog2.jpg
* ...

Images are categorized into subdirectories (cat, dog, etc.), each representing a class.

During training (train\_generator), augmented images are fed into the model, whereas during testing (test\_generator), only rescaled images are used to evaluate the model's performance.

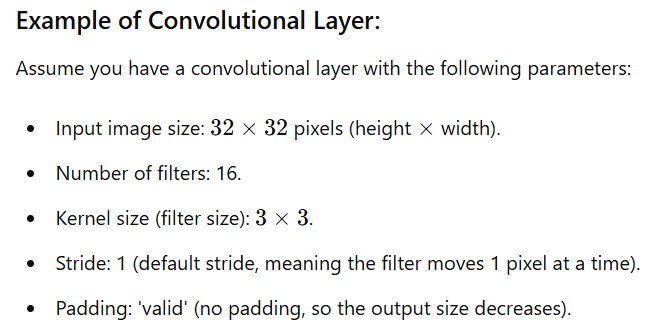
**Input Dimensions**: Neural networks expect input images to have consistent dimensions (height, width, and channels). For example, if your model's input shape is (height, width, channels), all input images must be resized to match this shape. This consistency ensures that each image is processed uniformly by the network.

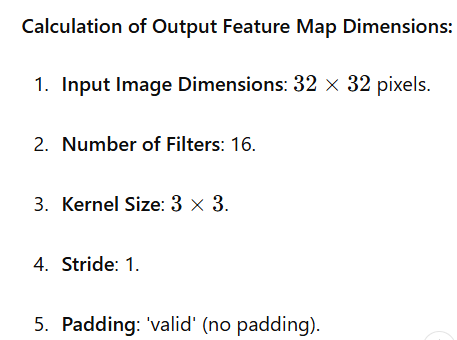
**Output Dimensions**: In the context of neural networks, especially convolutional neural networks (CNNs), the term "output dimensions" typically refers to the spatial dimensions of the feature maps produced by each layer in the network. For example, after passing through a convolutional layer, the output feature map has dimensions determined by the number of filters used and the stride of the convolution operation.

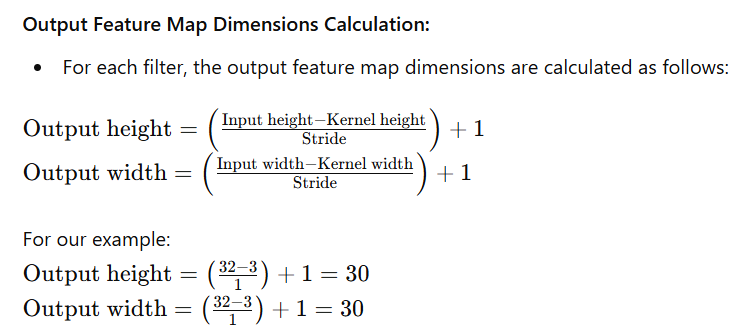
**Resizing Images**: When you specify target\_size in flow\_from\_directory, it ensures that all images loaded from the directory are resized to the specified dimensions (height, width) before being fed into the network. This resizing step preprocesses the data to conform to the input requirements of your neural network model.

**Model Input and Output**: Neural networks have an input layer that specifies the expected input shape (height, width, channels). The output of the network refers to the final predictions or outputs produced by the model after processing the input data through multiple layers. These outputs are typically tensors of various dimensions depending on the task (e.g., classification, segmentation).

# Dimensions of feature maps are determined by the number of filters and the stride of the convolution operation in a convolutional neural network (CNN):







Therefore, the output feature map dimensions after passing through this convolutional layer would be 30×3030 \times 3030×30 pixels for each of the 16 filters.

## Interpretation:

* Each filter in the convolutional layer convolves across the input image, applying a 3×33 \times 33×3 kernel with a stride of 1.
* The result is 16 feature maps (one for each filter), each with dimensions 30×3030 \times 3030×30.
* These feature maps represent the learned features extracted from the input image through the convolution operation.

# MODEL TRAINING TESTING AND SAVING

**model.fit()**: This method is used to train the model. It takes several parameters:

* train\_generator: This is the data generator object (train\_generator) that provides batches of training data during training. It should be configured to yield batches of input data (X) and corresponding target data (y) for training.
* epochs: Number of epochs (iterations over the entire training dataset) to train the model.
* verbose: Verbosity mode. verbose=1 prints progress bar and details for each epoch during training, while verbose=0 or verbose=2 may print less or no progress information respectively.
* validation\_data: This parameter specifies the data generator (test\_generator) used for validation during training. It should provide batches of validation data for evaluating the model after each epoch.

**model.save()**: This method saves the entire model to a specified file path (save\_path). It includes the model architecture, weights, and configuration. This saved model can later be loaded and used for inference or further training.

**save\_path**: Specifies the directory and file name where the model should be saved.

# LOADING THE MODEL

**Loading**: When keras.models.load\_model() is called with the path to the saved model file, it reads the model architecture, weights, and configuration from the file and reconstructs the exact model that was saved.

**Reconstructed Model**: The returned object (abc\_model in this case) is an instance of a Keras Model class. It has the same architecture (layers and their configurations) and weights (learned during training) as the model that was saved.

# MAKE PREDICTIONS

## COMPLETE FLOW OF IMAGE AUGMENTATION FOR PREDICTING :

1. setting the image path

2. setting the input shape and the number of classes

3. defining the categories of the classes

4. Loading and preprocessing the image using load\_img function of keras by taking only the spatial dimensions of the feature map

5. then converting the image to a numpy array and normalising the pixel values of the image to a range of [0,1] by dividing by 255

6. we pass the image to the model and make predictions using the .predict function of the model then

7. we will print the predictions by iterating through the multiple classes we defined earlier.

**Classes**:

* Cat
* Dog
* Bird

**Label Representation**: (ONE-HOT ENCODING)

* Each image of a cat is labeled [1, 0, 0].
* Each image of a dog is labeled [0, 1, 0].
* Each image of a bird is labeled [0, 0, 1].

# STEPS :

## Import the necessary libraries

Tensorflow, keras

ImageDataGenerator from Keras image preprocessing

Layers : Conv2D,MaxPooling2D,Flatten,Dense,Dropout from keras Layers

numpy

## For multiclass classification - define the classes

If we take classes in an image - saturated,hue,intensity

## Set the hyperparameters

batch\_size,

epochs,

input\_shape, (of an image in shape of (height,width,channel)

learning\_rate

## Defining a CNN model

-Conv2D layer with filters of size(3\*3) and an activation function such as “ReLU” for introducing non-linearity and to capture complex relationships in the data.

-Followed by a MaxPooling2D layer to reduce the spatial dimensions of the feature map developed as a result of an output of Conv2D layer and focus on the most important features

-Then converting the 2D output from previous pooling layer to 1D

To feed into the fully connected dense layer and then applying a

-Dropout layer to prevent overfitting and finally using softmax

Activation function in the output dense layer for multiclass classification.

-Hidden Layers - ReLU

-Output Layer - Softmax Activation

-For multiclass classification task.

## Compiling the model

-we use Adam optimiser from keras optimisers

With parameters as learning\_rate, loss as “categorical\_crossentropy”, and metrics as accuracy

## Data Augmentation and Preprocessing for Training

train\_datagen is used with parameters as

rescale, (within 0 to 1 so divide by 255)

rotation\_range,

width\_shift\_range,

height\_shift\_range,

horizontal\_flip,

fill\_mode,(if its nearest, value from the nearest pixel will be taken and filled)

Shear\_range

## Data Augmentation for Testing

In this only rescaling is applied

test\_datagen is used

## Load Training data

train\_generator is used with parameters as

flow\_from\_directory - path to be provided

target\_size (height,width) taken,

batch\_size,

Class\_mode

## Load testing data

test\_generator is used with parameters as

flow\_from\_directory - provide path

target\_size,

batch\_size,

class\_mode

## Train the model

Use model.fit with parameters as

train\_generator,

epochs,

verbose,

validation\_data(pass the test\_generator here)

## Save the trained model

save\_path, [in .keras format]

model.save(‘save\_path’)

## Make predictions on a sample image

image\_path,

load\_img using keras image preprocessing,

img\_to\_array(img) convert image to array using keras preprocessing of image,

expand\_dims of img,

and then normalise the image that is preprocess the image by div it by 255.

model.predict(img) - for making predictions

Enumerate over the classes to predict the emotions

—----------------------------------------------

# RANDOM FOREST STEPS :

## Split the data into training and testing sets

X\_train,X\_test,y\_train,y\_test , with parameters:

X,y,test\_size,random\_state

## Initialise the Random Forest Classifier

randforest\_classifier by setting parameters as

n\_estimators

random\_state

## Train the classifier on the training data

randforest\_classifier.fit , parameters as

X\_train, y\_train

## Make predictions on testing data

y\_pred

Using .predict function of randforest\_classifier

Parameter as : X\_test

## Evaluate the performance of the classifier

accuracy using accuracy\_score(), parameters : y\_test,y\_pred

Conf\_matrix using confusion\_matrix(), parameters: y\_test,y\_pred

Classification\_rep using classification\_report(), parameters: y\_test,y\_pred

## Save model to the respective file path.

Then import pickle

model = randforest\_classifier

file\_path

## Use the trained model to Make predictions

randforest\_classifier.predict()

Then calculate probabilities if required.

# ML ALGORITHMS WITH DATASET SIZE

| **Dataset Size** | **Recommended Algorithms** |
| --- | --- |
| Small (<10,000 samples) | Decision Trees, k-Nearest Neighbors (k-NN), Naive Bayes |
| Medium (10,000 - 100,000 samples) | Random Forests, Support Vector Machines (SVM), Gradient Boosting Machines (GBM) |
| Large (>100,000 samples) | Stochastic Gradient Descent (SGD), Deep Learning (Neural Networks), Distributed Computing (e.g., Spark MLlib) |

# MOST COMMONLY USED VECTORIZERS

| **Vectorizer** | **Description** |
| --- | --- |
| **CountVectorizer** | Converts a collection of text documents to a matrix of token counts, where each column represents a unique word (token) and each cell represents the count of that word in the document. Commonly used for bag-of-words models. |
| **TF-IDF Vectorizer** | Transforms text into TF-IDF feature vectors, where each cell represents the importance of a word in distinguishing a document from others in the corpus. Useful for text classification and information retrieval tasks. |
| **HashingVectorizer** | Converts text documents to feature vectors using hash functions to limit the number of features and reduce memory usage. It's memory efficient and suitable for large datasets. |
| **Word Embeddings** | Techniques like Word2Vec, GloVe, and FastText create dense vector representations of words based on their contextual meanings in a corpus. Captures semantic relationships between words and is used for more advanced NLP tasks. |
| **Doc2Vec** | An extension of Word2Vec that learns vector representations for entire documents, capturing the semantic meaning of documents as a whole. Useful for document similarity and clustering. |

## TF-IDF VECTORIZER CONVERTS TEXTUAL DATA INTO NUMERICAL FORMAT

# FOR AUDIO CLASSIFICATION: (IN process…)

* Use librosa library
* Padding will be applied to reduce to the target length

**Sampling Rate (sampling\_rate)**:

* The sampling rate refers to the number of samples of audio carried per second, measured in Hertz (Hz). It determines how many times per second the audio signal is sampled or digitized.
* For example, a sampling rate of 22050 Hz means that 22050 samples of the audio signal are taken every second.
* Higher sampling rates capture more detail in the audio waveform but also result in larger file sizes and increased computational requirements.

**Max Input Length (max\_input\_length)**:

* This parameter often represents the maximum allowable length or duration of an audio input, typically measured in samples.
* In your example, max\_input\_length = 286155 indicates the maximum number of audio samples that can be processed or analyzed at once.
* This limit can be influenced by hardware constraints (e.g., memory limitations) or specific requirements of the audio processing or machine learning algorithm being used.
* It helps in setting boundaries for audio processing tasks such as feature extraction, model training, or real-time audio analysis.
* Features to be extracted are :

**MFCCs**: Capture spectral features of the audio signal, often used in speech and music analysis.

**Chroma Features**: Represent harmonic content, useful for music genre classification and chord recognition.

**Mel Spectrogram Features**: Provide a detailed frequency representation of the audio signal over the Mel scale, useful for general audio classification tasks.