CHAPTER 1

INTRODUCTION TO INTELLIGENCE

1.1 DEFINITION:

Intelligence is the ability to learn,understand and apply knowledge and skills.It involves reasoning,problem solving and adapting to new situations.
Intelligence includes various cognitive abilities:

1. Learning

- The ability to acquire, retain, and apply knowledge or skills over time.

2. Reasoning

- The capacity to think logically, analyze information, and make informed decisions.

3. Problem-Solving

- The ability to identify challenges and develop effective solutions.

4. Adaptability

- The flexibility to adjust behavior and thinking in response to changing environments or new information.

5. Creativity

- The ability to generate novel and useful ideas, solutions, or approaches.

6. Emotional Intelligence

- The capacity to recognize, understand, and manage one's own emotions, as well as empathize with and influence others.

7. Abstract Thinking

- The ability to understand complex concepts and relationships that go beyond concrete experiences.

8. Social Intelligence

- The skill to interact effectively with others, navigate social situations, and build meaningful relationships.

1.2 FIRST STEP TOWARDS AI BACK IN 1940s:

Wartime Contributions (1939–1945): Code Breaking during the second world war.

- Context: During World War II, Turing worked for the British government at the center for Allied code breaking efforts.
- Primary Achievement:
    - Turing developed techniques to crack the German Enigma machine, a sophisticated encryption device used by Nazi forces to encode military communications.
    - He played a critical role in designing the Bombe machine, an electromechanical device that significantly automated the decryption of Enigma messages.
- Significance:
    - His work in codebreaking is credited with shortening the war by several years, saving millions of lives.
    - This effort showcased his ability to design machines capable of solving complex, logical problems, a precursor to modern computing.

What actually is the Turing test ?

The Turing Test is a measure proposed by a British mathematician and computer scientist Alan Turing in 1950 to determine a machine's ability to exhibit intelligent behavior indistinguishable from that of a human.

It was introduced in his seminal paper titled **"Computing Machinery and Intelligence"** published in the journal *Mind*.

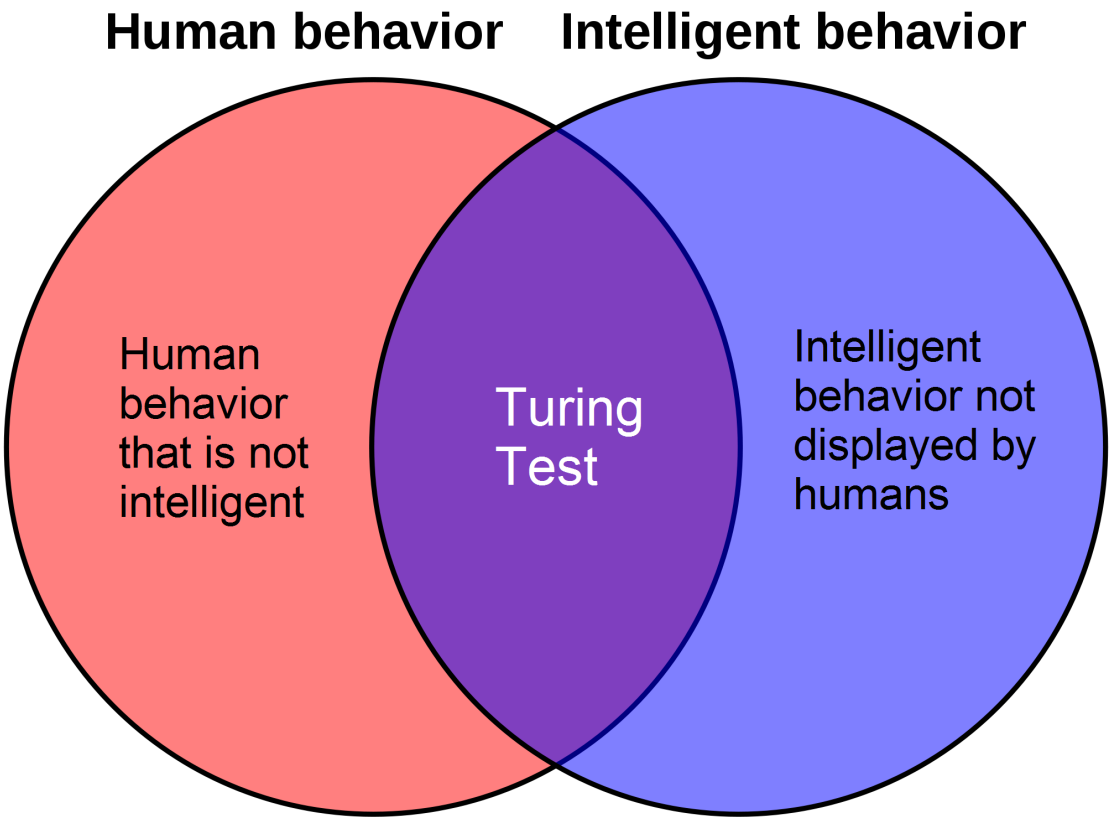
The test involves a human evaluator engaging in natural language conversations with a machine and another human through a text-based interface.

The evaluator doesn't know which participant is the machine and which is the human.

- If the evaluator **cannot reliably distinguish** the machine from the human based on their responses, the machine is said to have passed the Turing Test.
- The test focuses on **behavior** rather than internal thought processes, meaning it doesn't require the machine to "think" like a human, only to behave like one in conversation.

Key Elements of the Turing Test

1. Human Interaction: The machine must communicate with humans naturally, often via text.
2. Indistinguishability: The evaluator must not distinguish the machine's responses from a human's.
3. No Task Restriction: The conversation can cover any topic.



1.3 WHAT IS ARTIFICIAL INTELLIGENCE?

Artificial Intelligence (AI) refers to the field of computer science focused on creating systems or machines that can perform tasks typically requiring human intelligence. These tasks include learning, reasoning, problem-solving, understanding natural language, perception, and decision-making.

Categories of artificial intelligence

#Types of AI Based on Capability

1. Narrow AI (Weak AI)
    - Definition: AI systems designed to perform a single specific task with proficiency.
    - Examples:
        - Virtual assistants (e.g., Siri, Alexa).
        - Recommendation systems (e.g., Netflix, Amazon).
        - Image recognition software.
    - Characteristics:
        - Task-specific.

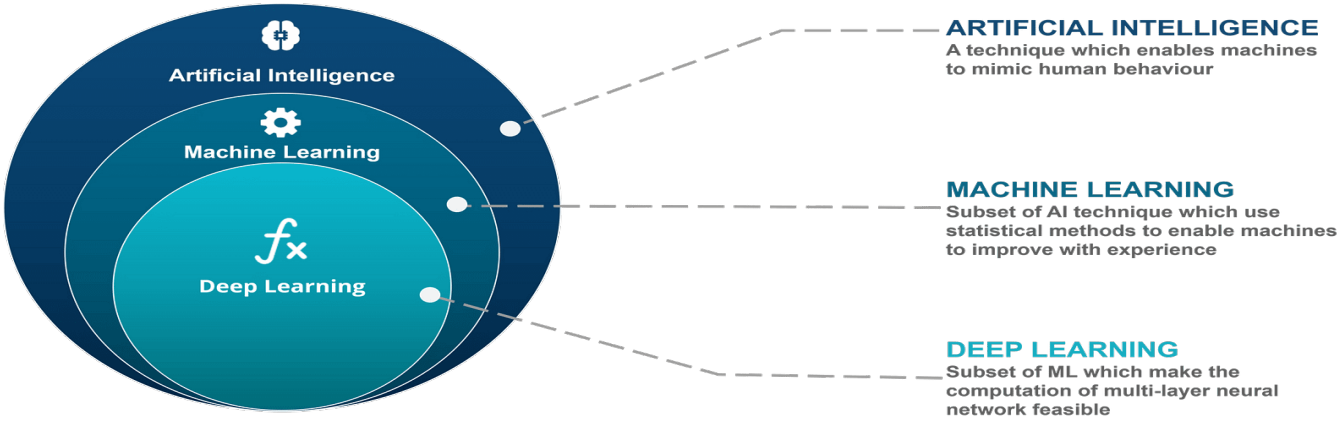
        - Lacks generalization and human-like adaptability.
2. General AI (Strong AI)
    - Definition: Hypothetical AI that possesses the ability to perform any intellectual task a human can do.
    - Examples: No real-world implementations yet.
    - Characteristics:
        - Can think, reason, and learn like a human.
        - Able to apply knowledge across diverse domains.
3. Superintelligent AI
    - Definition: A theoretical AI that surpasses human intelligence in virtually all fields, including creativity, decision-making, and problem-solving.
    - Examples: Only a concept for now; often discussed in speculative or futuristic contexts.
    - Characteristics:
        - Far exceeds human capabilities.
        - Potential to revolutionize or pose risks to humanity (e.g., ethical concerns).

#Types of AI Based on Functionality

1. Reactive Machines
    - Definition: AI systems that react to current scenarios without relying on past experiences or memory.
    - Examples:
        - IBM's Deep Blue, the chess-playing AI.
    - Characteristics:
        - No memory or learning capability.
        - Performs tasks based on real-time input.
2. Limited Memory
    - Definition: AI systems that can use past data to inform current decisions and improve performance.
    - Examples:
        - Autonomous vehicles using sensor data for navigation.
    - Characteristics:
        - Retains short-term memory for specific tasks.
        - Most modern AI systems fall into this category.
3. Theory of Mind
    - Definition: AI capable of understanding emotions, beliefs, and intentions, and interacting socially.
    - Examples: Still in research phases, with developments in emotionally intelligent systems.
    - Characteristics:
        - Can interpret human emotions and respond accordingly.
        - Crucial for advanced human-AI interaction.

4. Self-Aware AI
    - Definition: Theoretical AI with self-consciousness and awareness of its own existence.
    - Examples: Does not currently exist; considered a distant goal of AI research.
    - Characteristics:
        - Possesses self-awareness and subjective understanding.

RELATIONSHIP BETWEEN AI VS ML VS DL:

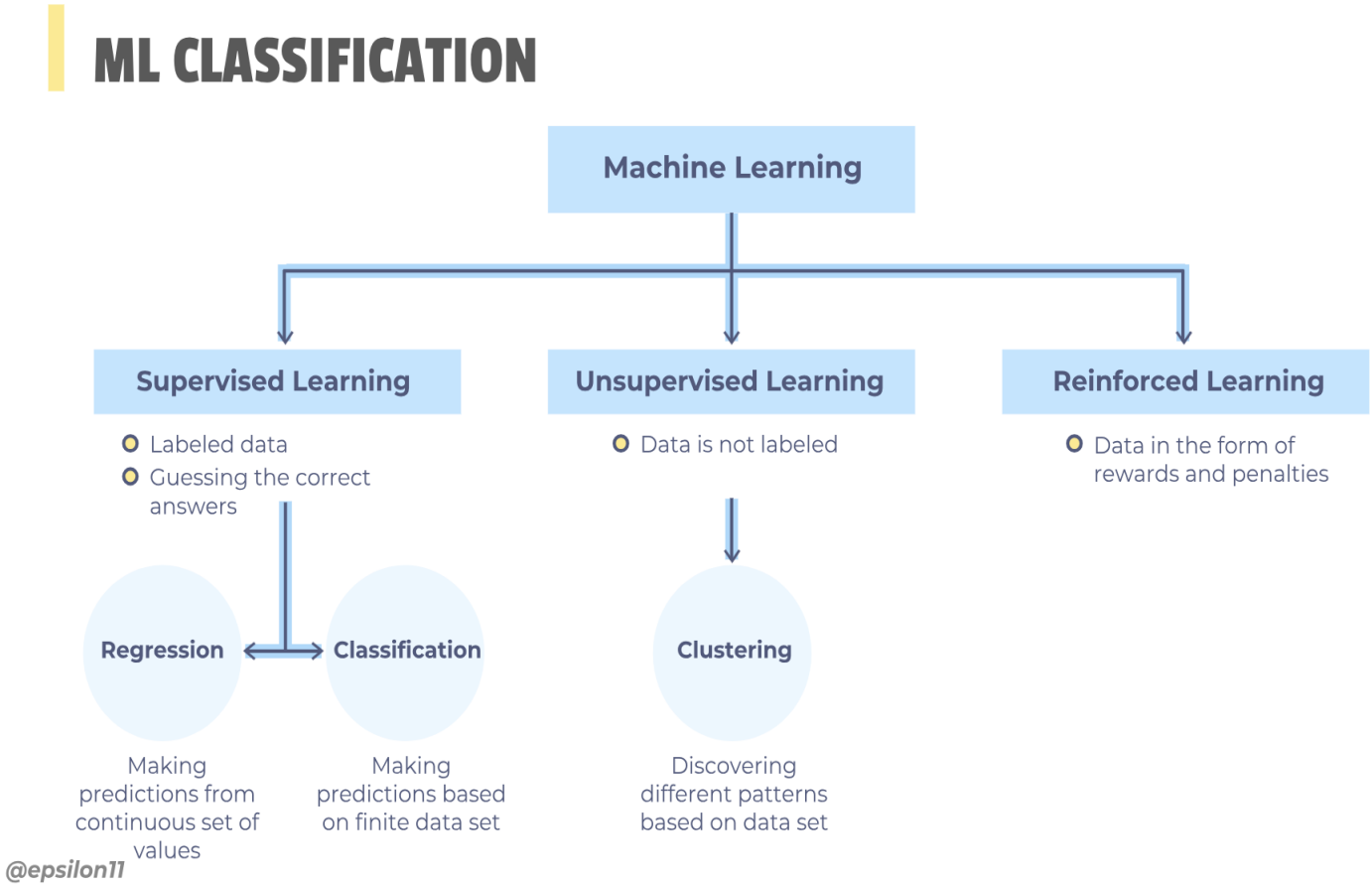


CHAPTER 2
INTRODUCTION TO MACHINE LEARNING

2.1 MACHINE LEARNING

Machine Learning (ML) is a branch of artificial intelligence (AI) that focuses on building systems capable of learning and improving their performance automatically from data without being explicitly programmed for every specific task.

2.2 TYPES OF MACHINE LEARNING

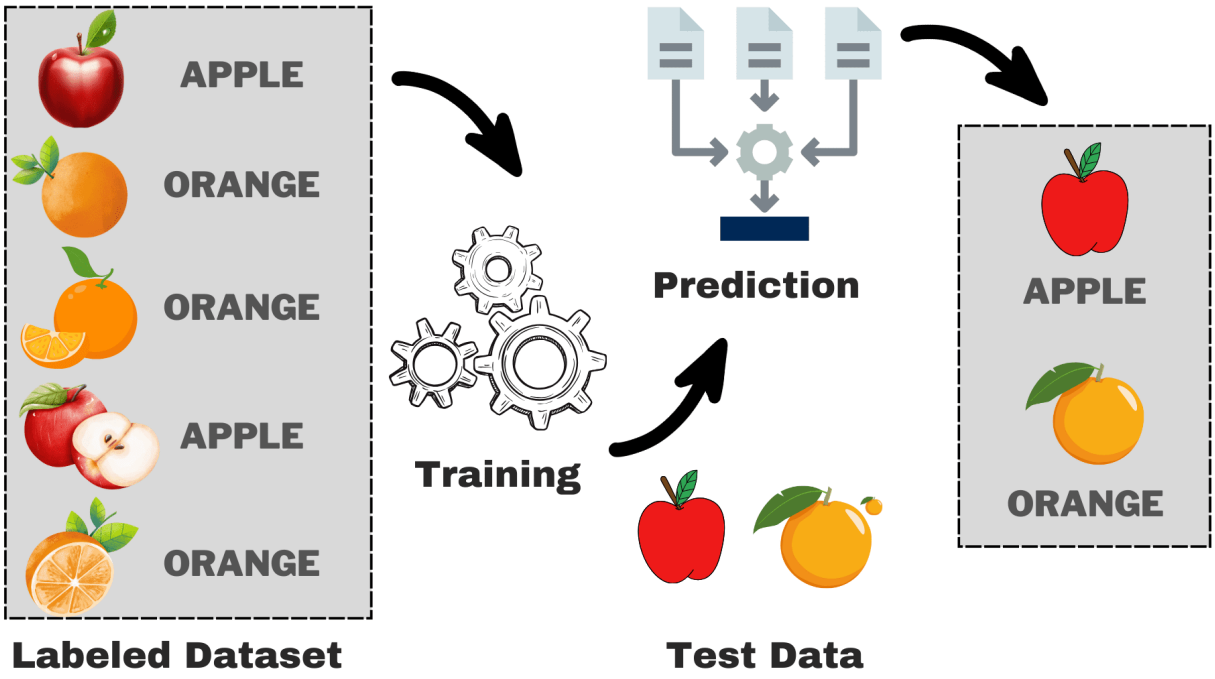


2.2.1.SUPERVISED LEARNING:

Supervised learning is defined as when a model gets trained on a “Labelled dataset”. Labelled datasets have both input and output parameters. Supervised learning algorithms learn to map points between inputs and correct outputs. It has both training and validation datasets labelled.

Let’s understand it with the help of an example.

Example: Consider a scenario where you have to build an image classifier to differentiate between cats and dogs. If you feed the datasets of dogs and cats labelled images to the algorithm, the machine will learn to classify between a dog or a cat from these labeled images. When we input new dog or cat images that it has never seen before, it will use the learned algorithms and predict whether it is a dog or a cat. This is how supervised learning works, and this is particularly an image classification.



There are two main categories of supervised learning:

1. Classification
2. Regression

Classification

Definition: Classification is a task where the goal is to predict a discrete label or category for an input based on its features.

Key Characteristics:

- Outputs are categorical or discrete values.
- Answers questions like "Which category does this belong to?" or "Is this yes or no?"

Examples:

1. Spam Detection:
    - Input: An email (text, metadata).
    - Output: "Spam" or "Not Spam" (binary classification).
2. Image Recognition:
    - Input: An image of a fruit.
    - Output: "Apple," "Banana," or "Orange" (multi-class classification).

3. Medical Diagnosis:
    - Input: Patient's test results.
    - Output: "Disease Present" or "No Disease."

Regression

Definition: Regression is a task where the goal is to predict a continuous numerical value based on input features.

Key Characteristics:

- Outputs are continuous or real-valued numbers.
- Answer questions like "How much?" or "What is the value?"

Examples:

1. House Price Prediction:
    - Input: Features like size, location, number of rooms.
    - Output: A predicted price (e.g., $350,000).
2. Weather Forecasting:
    - Input: Historical weather data.
    - Output: Predicted temperature (e.g., 25.6°C).
3. Stock Price Prediction:
    - Input: Historical stock prices, trading volume.
    - Output: Predicted future stock price.

Applications of Supervised Learning

Supervised learning is used in a wide variety of applications, including:

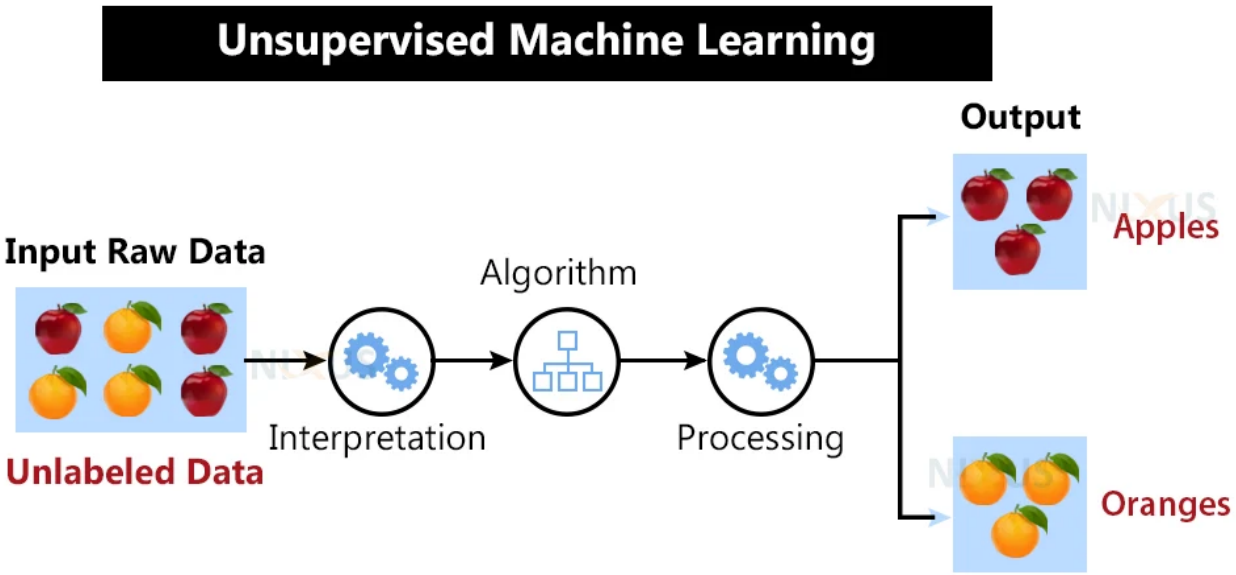
- Image classification: Identify objects, faces, and other features in images.
- Natural language processing: Extract information from text, such as sentiment, entities, and relationships.
- Speech recognition: Convert spoken language into text.
- Recommendation systems: Make personalized recommendations to users.
- Predictive analytics: Predict outcomes, such as sales, customer churn, and stock prices.
- Medical diagnosis: Detects diseases and other medical conditions.
- Fraud detection: Identify fraudulent transactions.
- Autonomous vehicles: Recognize and respond to objects in the environment.
- Email spam detection: Classify emails as spam or not spam.

2.2.2 . UNSUPERVISED LEARNING:

Unsupervised learning is a type of machine learning technique in which an algorithm discovers patterns and relationships using unlabeled data. Unlike supervised learning, unsupervised learning doesn't involve providing the algorithm with labeled target outputs. The primary goal of Unsupervised learning is often to discover hidden patterns, similarities, or clusters within the data, which can then be used for various purposes, such as data exploration, visualization, dimensionality reduction, and more.

Let's understand it with the help of an example.

Example: Consider that you have a dataset that contains information about the purchases you made from



the shop. Through clustering, the algorithm can group the same purchasing behavior among you and other customers, which reveals potential customers without predefined labels. This type of information can help businesses get target customers as well as identify outliers.

Main category of unsupervised learning-Clustering

Clustering is an unsupervised learning technique used to divide a dataset into groups (called clusters) such that data points within the same cluster are more similar to each other than to those

in other clusters. It helps uncover hidden patterns or structures in data without using predefined labels.

Imagine a box full of mixed marbles of different colors: red, blue, and green. Without any prior knowledge about the marbles, your task is to sort them into groups based on their colors.

- Input: A collection of mixed marbles.
- Process: Group marbles based on their color similarity (you might visually notice clusters of red, blue, and green marbles forming).
- Output: Three clusters—one for red, one for blue, and one for green marbles.

Here, clustering identifies groups (clusters) of similar items (marbles) based on their shared characteristic (color), even though no labels (e.g., "Red," "Blue," "Green") are provided initially.

Applications of Unsupervised Learning
Here are some common applications of unsupervised learning:

- Clustering: Group similar data points into clusters.
- Anomaly detection: Identify outliers or anomalies in data.
- Dimensionality reduction: Reduce the dimensionality of data while preserving its essential information.
- Recommendation systems: Suggest products, movies, or content to users based on their historical behavior or preferences.

2.2.3. REINFORCEMENT LEARNING:

Reinforcement Learning is a type of machine learning where an agent learns to make decisions by interacting with an environment to maximize a reward. Unlike supervised learning, RL doesn't rely on labeled data; instead, the agent learns through trial and error using feedback in the form of rewards or penalties.

How Reinforcement Learning Works

1. The agent starts in an initial state.
2. It chooses an action based on its policy.
3. The environment transitions to a new state and provides a reward.
4. The agent updates its policy to maximize cumulative rewards over time.

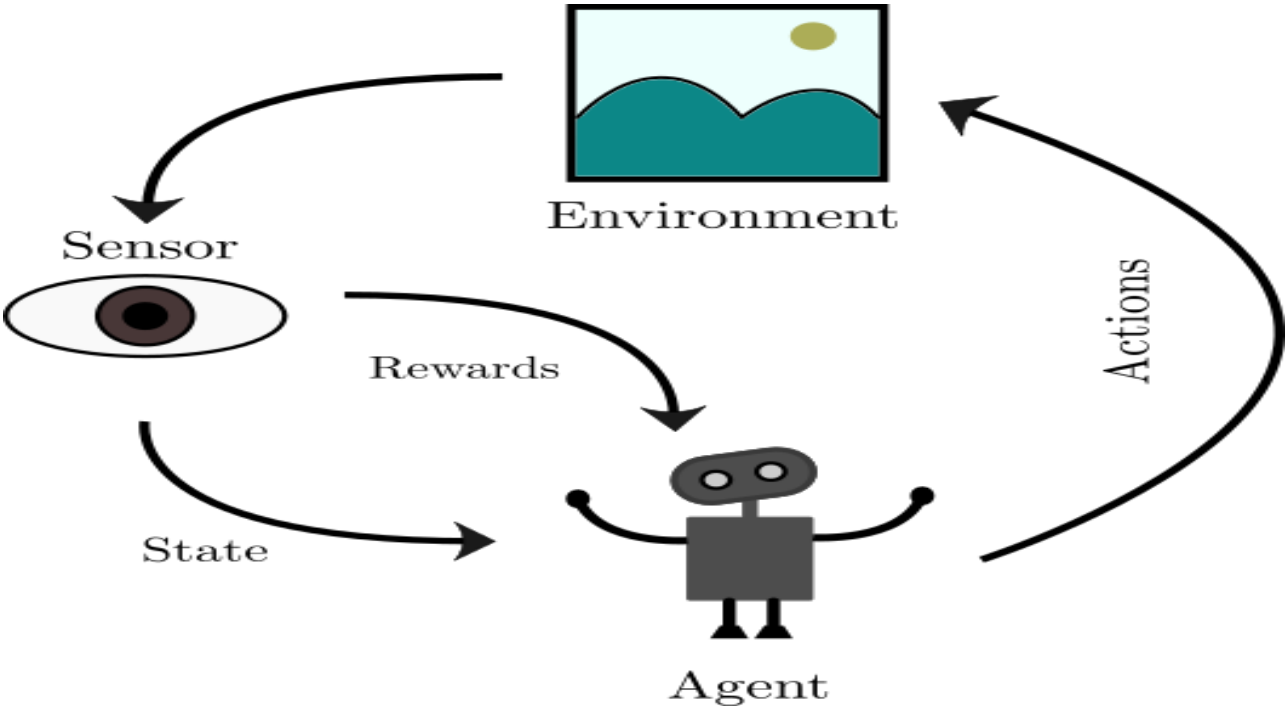
This cycle repeats until the agent learns an optimal strategy.

Example: Teaching a Robot to Walk

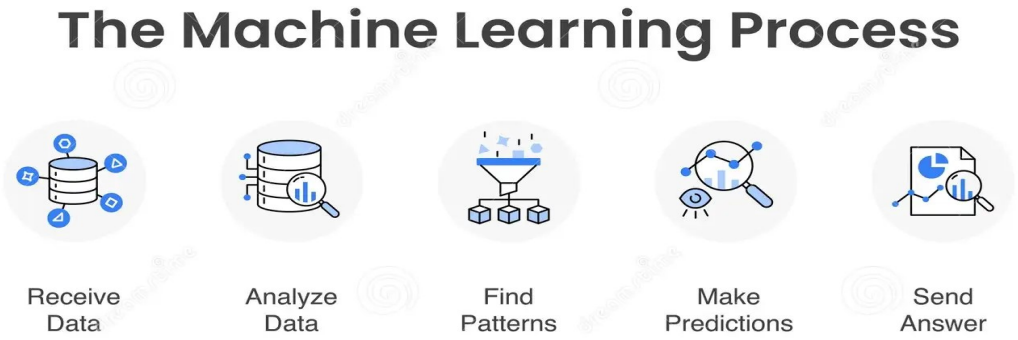
1. Environment: A simulated room.

2. Agent: The robot.
3. State: The robot's current position and posture.
4. Action: Moving its legs (forward, backward, lift, etc.).
5. Reward:
    - +1: For walking a step without falling.
    - -1: For falling.

Initially, the robot may fall often, receiving penalties. Over time, it learns which leg movements keep it stable and help it walk further.



WORKFLOW OF MACHINE LEARNING MODEL:



CHAPTER 3
DEEP LEARNING AND CNN'S BASICS

3.1 DEEP LEARNING

Deep Learning is a subset of machine learning that uses neural networks with many layers (hence "deep") to automatically extract and learn complex patterns from data. It is particularly effective for tasks involving unstructured data, such as images, audio, and text.Deep learning comes into existence when we have to work with large unstructured data because when training with simple machine learning algorithms the accuracy of the model decreases due to large amounts of data.

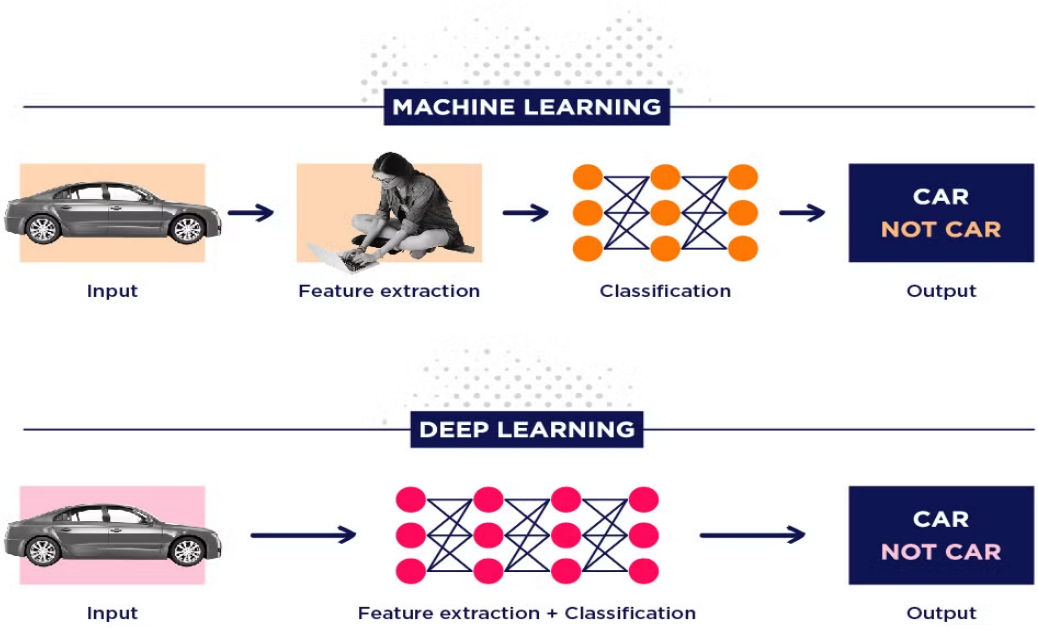
In this automatically the deep learning architecture will extract features and there is no need to define it explicitly by the programmer.

Steps in Deep Learning

1. Define the Problem:
    - Identify the task (e.g., image classification, text generation).
    - Specify the input and output.
2. Collect and Preprocess Data:
    - Gather a dataset relevant to the problem.
    - Preprocess data:
        - Normalize numerical data (e.g., scaling pixel values to [0, 1]).
        - Augment data (e.g., rotate or flip images for more variety).
        - Tokenize and pad sequences for text data.
3. Choose a Deep Learning Framework:
    - Use libraries such as TensorFlow, PyTorch, or Keras to build and train models.
4. Design the Neural Network:
    - Decide the architecture:
        - Number of layers (e.g., convolutional, recurrent, dense).
        - Number of neurons in each layer.
        - Activation functions (e.g., ReLU, sigmoid, softmax).
    - Select a loss function (e.g., cross-entropy, mean squared error).
5. Split Data:

    - Split the dataset into training, validation, and test sets.
6. Train the Model:
    - Feed training data into the model in batches.
    - Optimize the model using an optimizer (e.g., Adam, SGD).
    - Adjust weights and biases to minimize the loss.
7. Validate the Model:
    - Evaluate performance on the validation set.
    - Tune hyperparameters (e.g., learning rate, batch size).
8. Test the Model:
    - Use the test set to measure final accuracy, precision, recall, etc.

MACHINE LEARNING VS DEEP LEARNING:



ITERATORS
Source: www.iteratorshq.com

Example:

Image classification is a task where the goal is to assign a label (class) to an input image. In **brain tumor classification**, the objective is to classify medical images (e.g., MRI scans) into categories, such as:

1. **No Tumor**

2. **Benign Tumor**
3. **Malignant Tumor**

This is a critical application in healthcare to assist radiologists and doctors in diagnosing brain tumors quickly and accurately.

Steps for Brain Tumor Classification

1. Define the Problem

- Objective: Classify MRI images into one of three categories: No Tumor, Benign Tumor, or Malignant Tumor.
- Input: MRI scans (images).
- Output: Predicted class label for each image.

2. Collect and Preprocess Data

- Data Collection:
    1. Obtain MRI images of brain scans from medical datasets (e.g., Kaggle, TCIA, or hospital archives).
    2. Ensure data includes diverse cases and imaging modalities.
- Data Preprocessing:
    3. Resize all images to a uniform size (e.g., 224x224 pixels) for consistent input to the neural network.
    4. Normalize pixel values to a range of [0, 1] by dividing by 255.
    5. Augment the dataset to improve model robustness:
        - Rotate images.
        - Flip horizontally/vertically.
        - Add slight noise or adjust brightness.
    6. Label Encoding:
        - Assign numerical labels to classes: 0 (No Tumor), 1 (Benign), 2 (Malignant).

3. Split the Dataset

- Training Set: 70% of the images for training the model.
- Validation Set: 15% of the images for tuning hyperparameters.
- Test Set: 15% of the images for final evaluation.

4. Design the Neural Network

For image classification, a Convolutional Neural Network (CNN) is ideal because CNNs are excellent at capturing spatial patterns in images.

5. Train the Model

- Loss Function: Use categorical cross-entropy to measure the difference between predicted and true labels.
- Optimizer: Use Adam for faster convergence.
- Batch Size: Train in small batches (e.g., 32 images at a time).
- Epochs: Train the model over several iterations (e.g., 20–50 epochs).

6. Validate the Model

- Evaluate the model on the validation set after each epoch to monitor overfitting and adjust hyperparameters like:
    - Learning rate.
    - Number of layers or neurons.
    - Regularization parameters (e.g., dropout rate).

7. Test the Model

- Measure performance on the test set using metrics:
    - Accuracy: Overall correctness of predictions.
    - Precision: Focuses on correctly identifying true positives.

3.2 NEURAL NETWORK :

A neural network in deep learning is a computational model inspired by the way biological neural networks in the human brain process information. It is a type of machine learning model that learns patterns from data and makes predictions or decisions based on those patterns.

Components of a neural network:

1. NEURONS:

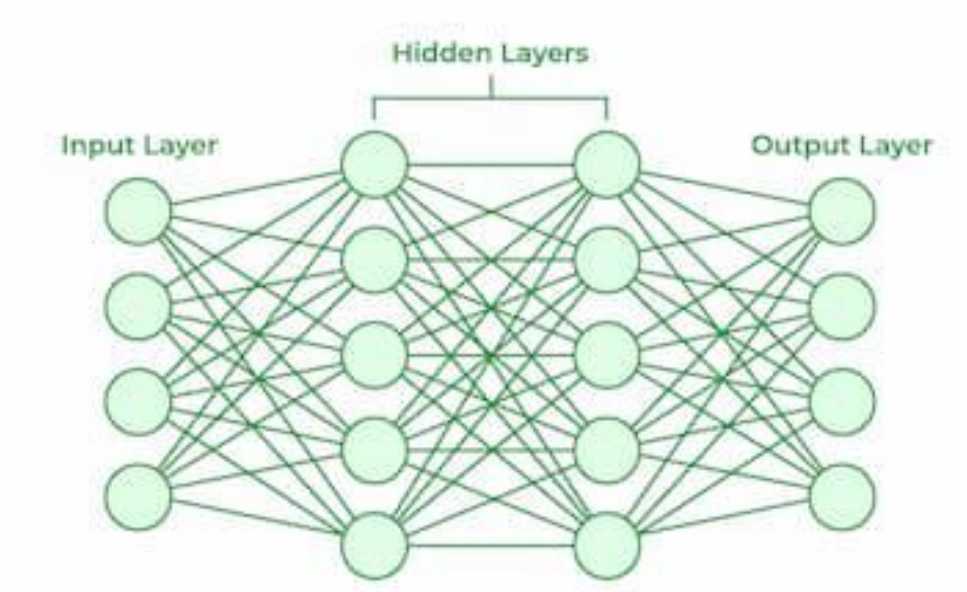
   These are the basic units of a neural network.Each neuron receives inputs, processes them, and passes the result to the next layer.

2. LAYERS:

   Input Layer: Receives the raw data features.

   Hidden Layers: Intermediate layers where data transformations and computations occur.

   Output Layer: Produces the final result (e.g., classification, regression value).



3. WEIGHTS AND BIASES:

   Weights are parameters that scale the input values, determining their importance.

   Biases are additional parameters added to the weighted input to shift the output.

4. ACTIVATION FUNCTIONS:

An activation function is a mathematical function applied to the output of a neuron. It introduces non-linearity into the model, allowing the network to learn and represent complex patterns in the data. Without this nonlinearity feature, a neural network would behave like a linear regression model, no matter how many layers it has.

Examples: ReLU, Sigmoid, Tanh, Softmax.

Working of a neural network:

Forward Propagation

When data is input into the network, it passes through the network in the forward direction, from the input layer through the hidden layers to the output layer. This process is known as forward propagation. Here's what happens during this phase:

1. Linear Transformation: Each neuron in a layer receives inputs, which are multiplied by the weights associated with the connections. These products are summed together, and a

bias is added to the sum. This can be represented mathematically as: $z=w_1x_1+w_2x_2+\ldots+w_nx_n+b$

2. Activation: The result of the linear transformation (denoted as z is then passed through an activation function. The activation function is crucial because it introduces non-linearity into the system, enabling the network to learn more complex patterns. Popular activation functions include ReLU, sigmoid, and tanh.

Backpropagation

After forward propagation, the network evaluates its performance using a loss function, which measures the difference between the actual output and the predicted output. The goal of training is to minimize this loss. This is where backpropagation comes into play:

1. Loss Calculation: The network calculates the loss, which provides a measure of error in the predictions. The loss function could vary; common choices are mean squared error for regression tasks or cross-entropy loss for classification.
2. Gradient Calculation: The network computes the gradients of the loss function with respect to each weight and bias in the network. This involves applying the chain rule of calculus to find out how much each part of the output error can be attributed to each weight and bias.
3. Weight Update: Once the gradients are calculated, the weights and biases are updated using an optimization algorithm like stochastic gradient descent (SGD). The weights are adjusted in the opposite direction of the gradient to minimize the loss. The size of the step taken in each update is determined by the learning rate.

Iteration

This process of forward propagation, loss calculation, backpropagation, and weight update is repeated for many iterations over the dataset. Over time, this iterative process reduces the loss, and the network's predictions become more accurate.

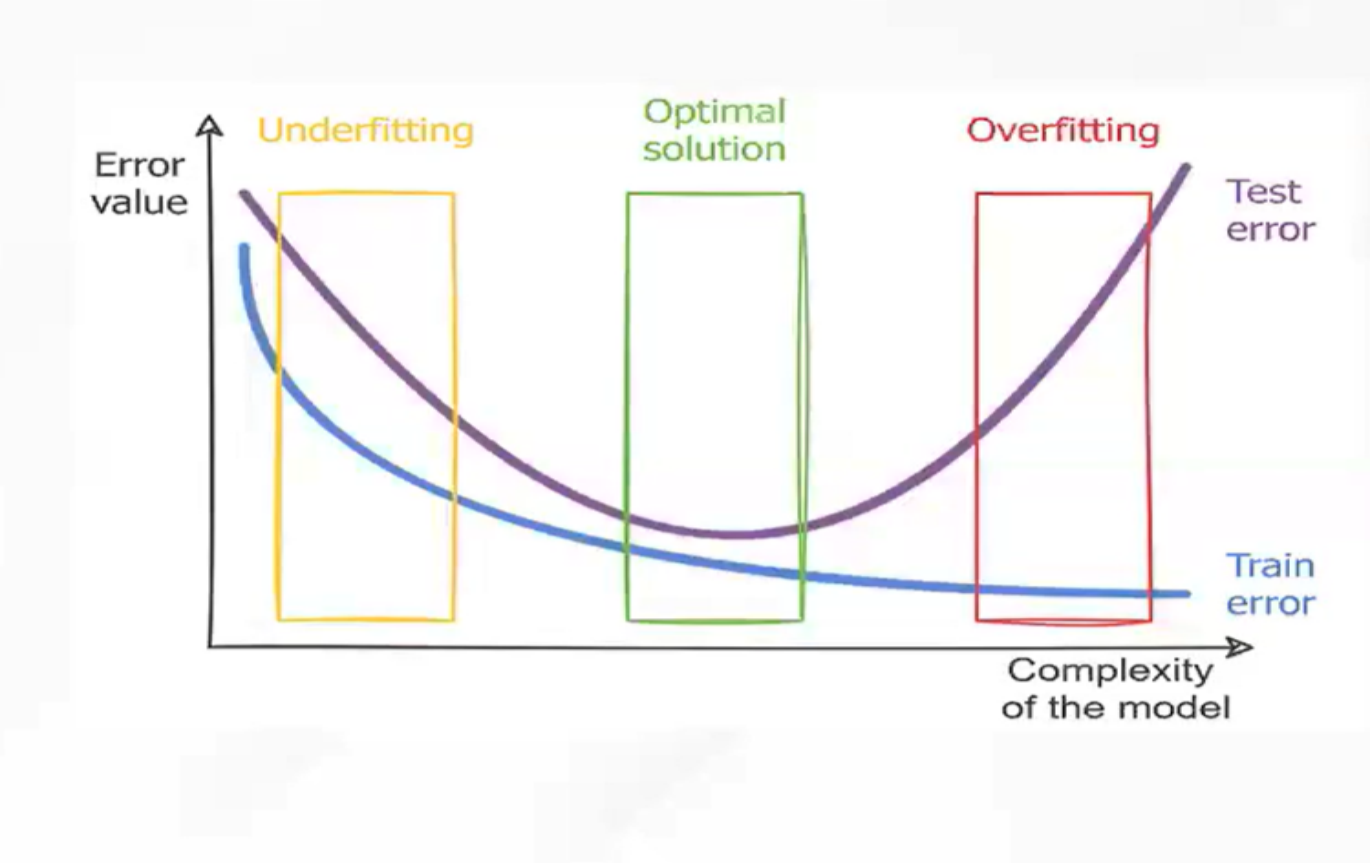
Through these steps, neural networks can adapt their parameters to better approximate the relationships in the data, thereby improving their performance on tasks such as classification, regression, or any other predictive modeling.

OVERFITTING AND UNDERFITTING:

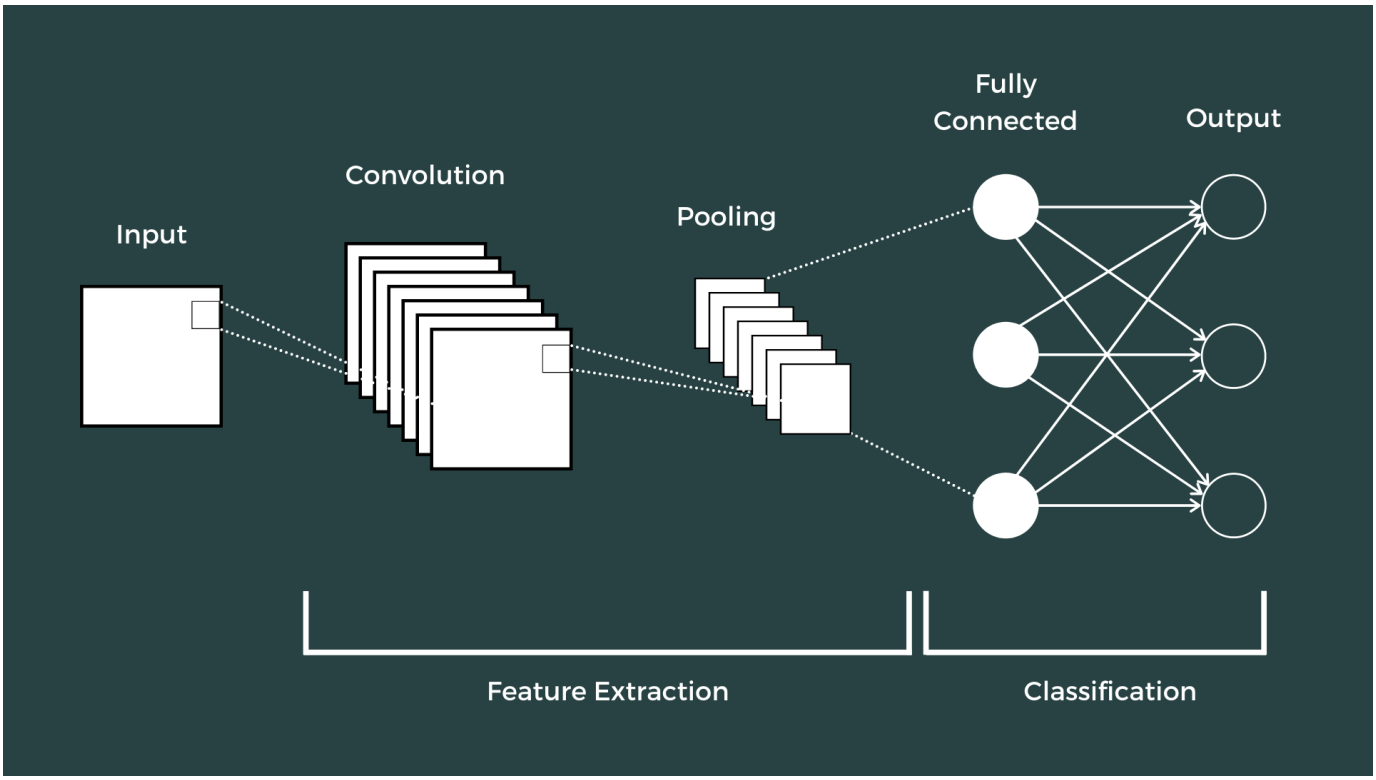
1. Overfitting: refers to a scenario where a machine learning model can not generalize or fit well on unseen dataset. A clear sign of overfitting is that its error on testing and validation dataset is much greater than the error on testing dataset.
2. Underfitting: that can neither model the training dataset and nor generalize to new dataset

DROPOUT:

Dropout is a regularization technique used in neural networks to prevent overfitting. It works by randomly "dropping out" (deactivating) subsets of neurons during training, which forces the network to learn more robust and generalizable features.



3.3 CONVOLUTIONAL NEURAL NETWORK



A Convolutional Neural Network (CNN) is a type of deep learning model primarily used for analyzing visual data such as images, video frames, or any data with a grid-like structure. CNNs are particularly well-suited for tasks like image classification, object detection, and segmentation due to their ability to automatically learn hierarchical features from raw image data.

The key advantage of CNNs over traditional neural networks is that they are designed to capture spatial hierarchies in images by exploiting the local relationships between pixels.

A CNN consists of several types of layers, each with its own function. Here's a breakdown of the main layers in a typical CNN:

1. Input Layer

- The input layer receives the raw image data. Images are usually represented as matrices of pixel values (e.g., 256x256 for a color image). Each pixel has values for the three-color channels: red, green, and blue (RGB).

2. Convolutional Layer

- The convolutional layer is the core building block of a CNN and performs the convolution operation, which is the process of applying filters (also called kernels) to the input image to detect patterns, edges, textures, and other local features.

- Filters: These are small matrices (e.g., 3x3 or 5x5) that slide over the input image, performing a dot product at each location. The result of the convolution is a feature map, which highlights the presence of specific features in the image.
- The idea is to learn different filters during training that can detect various features like edges, corners, and textures.

3. Activation Function (ReLU)

- After the convolution operation, the ReLU (Rectified Linear Unit) activation function is typically applied. ReLU introduces non-linearity by setting all negative values to zero and leaving positive values unchanged.
- This allows the network to learn more complex features and patterns beyond simple linear transformations.

4. Pooling (Subsampling) Layer

- The pooling layer is used to reduce the spatial dimensions (width and height) of the feature maps, helping decrease the number of parameters and computations, which reduces overfitting.
- Max Pooling is the most common technique, which selects the maximum value from a specific window (e.g., 2x2) of the feature map. This helps in retaining the most important features while reducing the dimensionality.

5. Fully Connected (FC) Layer

- After several convolutional and pooling layers, the feature maps are flattened (converted into a one-dimensional vector) and passed through one or more fully connected layers.
- These layers are traditional dense layers, where every neuron is connected to every neuron in the previous layer. The fully connected layers help in classifying the extracted features into the final output classes (e.g., "cat" or "dog" in an image classification task).

6. Output Layer

- The final layer is typically a SoftMax layer for multi-class classification or a sigmoid layer for binary classification. It outputs a probability distribution over the classes, indicating which class the image belongs to.

How CNNs Work:

1. Feature Learning:

    a. During training, CNNs learn filters that can automatically detect low-level features such as edges or textures in the initial layers and more complex features (like shapes or parts of objects) in deeper layers.
2. End-to-End Learning:
    a. CNNs can be trained end-to-end on labeled image datasets (e.g., ImageNet). The backpropagation algorithm adjusts the weights of filters based on the error between the predicted and true labels.
3. Hierarchical Feature Extraction:
    a. Each convolutional layer learns more abstract features by building on the features detected by previous layers. For example, while the first layer might learn simple edges, subsequent layers can detect patterns like corners, textures, or even higher-level structures like faces or animals.

IMPLEMENTATION IN KAGGLE

[1]:
```
import numpy as np
import pandas as pd
import json
import os
import tensorflow as tf
from tqdm import tqdm
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img, img_to_array
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import Sequence
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, Activation, Dropout,
from tensorflow.keras.layers import Embedding, LSTM, add, Concatenate, Reshape, concatenate, Bidirectio
from tensorflow.keras.applications import VGG16, ResNet50, DenseNet201
```

[2]:
```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
import warnings
import matplotlib.pyplot as plt
import seaborn as sns
from textwrap import wrap

plt.rcParams['font.size'] = 12
sns.set_style("dark")
warnings.filterwarnings('ignore')
```

+ Code + Markdown

1. **Libraries and Setup**

   This section of the code is responsible for importing the necessary libraries and setting up the environment for building an image captioning model using deep learning. The imports include various modules for data handling, model building, image processing, training, and visualization.

   Now lets split them and see what each line means and does for our code:

   1. Data Handling and Utilities

- **NumPy** (*np*) is used for efficient numerical computations and handling of multi-dimensional arrays.

- **Pandas** (*pd*) is used for loading and manipulating structured data (e.g., CSV or JSON).

- **json** handles reading and writing JSON files (often used in datasets like MS-COCO).

- **os** is used for interacting with the operating system, e.g., navigating file paths.

2. TensorFlow and Keras (Deep Learning Framework)

These imports bring in all necessary components from TensorFlow's Keras API:

a. Image loading and augmentation via *ImageDataGenerator*, *load_img*, *img_to_array*.

b. Text preprocessing via *Tokenizer* and *pad_sequences*, required for processing captions.

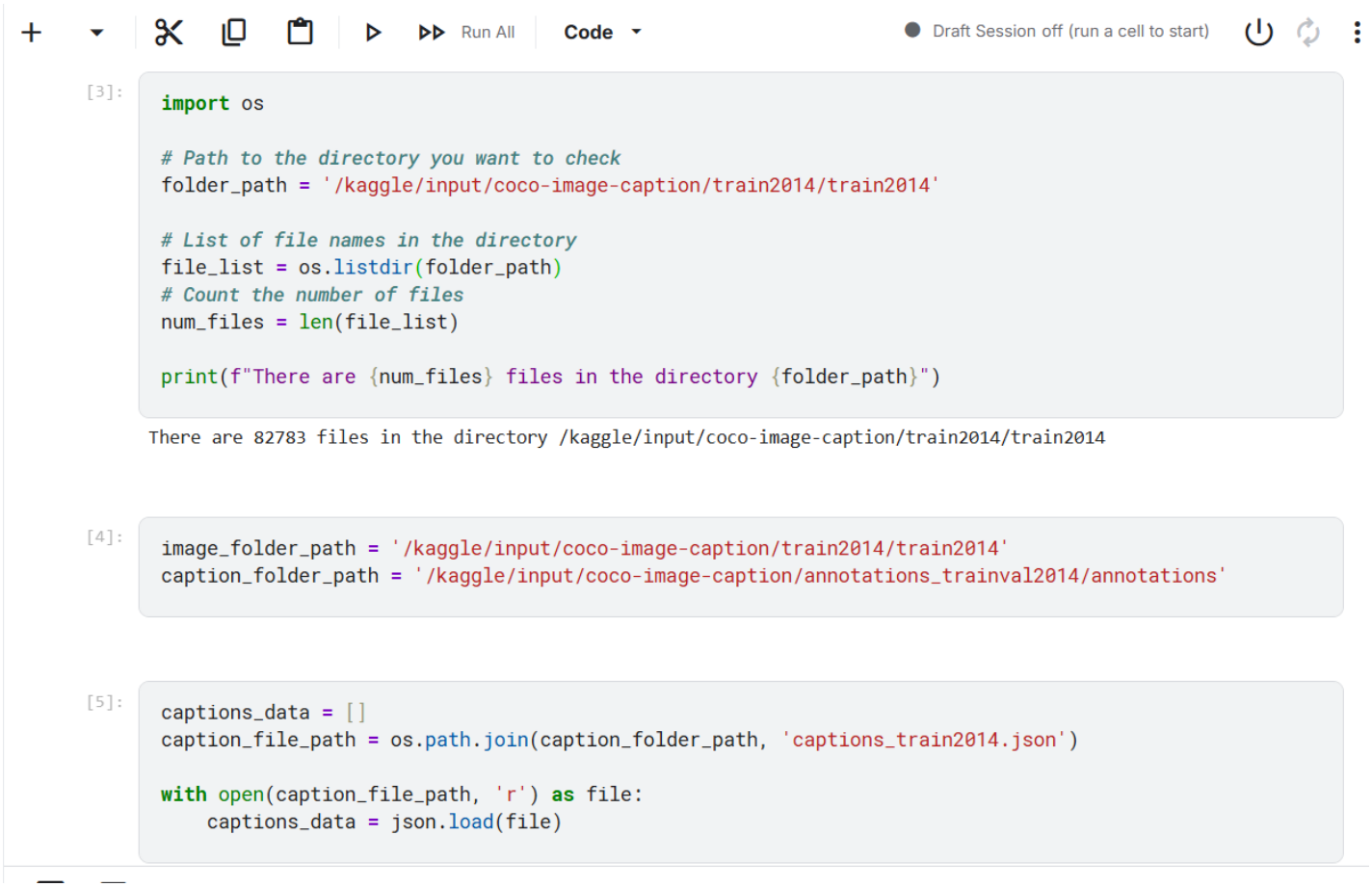
c. Sequence and to_categorical help in building custom data generators and encoding output.

d. Modeling tools like *Sequential*, *Model*, and various layers (e.g., CNN, LSTM, Dense).

e. Pre-trained models (VGG16, ResNet50, DenseNet201) are imported to use as image feature extractors.

f. Optimizers like *Adam* help improve model training.

g. Callbacks like *ModelCheckpoint*, *EarlyStopping*, and *ReduceLROnPlateau* are used for efficient training and preventing overfitting.



2. Dataset Loading and Preparation

In this above figure, we perform the initial loading of the COCO image dataset and its corresponding caption annotations. The COCO (Common Objects in Context) dataset is a widely used benchmark for image captioning tasks and contains thousands of images with multiple human-annotated captions.

Mainly it consists of three steps:

1. Accessing image folder
2. Setting path for images and captions
3. Loading the caption annotations form the JSON file

[6]:

```
file_names = []
image_ids = []
captions = []

annotations = captions_data['annotations']
images=captions_data['images']

# Create a dictionary to map image_id to file_name
image_id_to_filename = {image['id']: image['file_name'] for image in images}

# Initialize empty lists to store the data
image_ids = []
captions = []
file_names = []

# Loop through the list of annotations
for annotation in annotations:
    image_id = annotation['image_id']
    if image_id in image_id_to_filename:
        file_name = image_id_to_filename[image_id]
        image_ids.append(image_id)
        captions.append(annotation['caption'])
        file_names.append(file_name)

# Create a pandas DataFrame using the extracted data
data = {
    'image_id': image_ids,
     'image': file_names,
    'caption': captions
```

[7]:
```
data
```

[7]:

	image_id	image	caption
0	318556	COCO_train2014_000000318556.jpg	A very clean and well decorated empty bathroom
1	116100	COCO_train2014_000000116100.jpg	A panoramic view of a kitchen and all of its a...
2	318556	COCO_train2014_000000318556.jpg	A blue and white bathroom with butterfly theme...
3	116100	COCO_train2014_000000116100.jpg	A panoramic photo of a kitchen and dining room
4	379340	COCO_train2014_000000379340.jpg	A graffiti-ed stop sign across the street from...
...	...	...	...
414108	133071	COCO_train2014_000000133071.jpg	a slice of bread is covered with a sour cream ...
414109	410182	COCO_train2014_000000410182.jpg	A long plate hold some fries with some sliders...
414110	180285	COCO_train2014_000000180285.jpg	Two women sit and pose with stuffed animals.
414111	133071	COCO_train2014_000000133071.jpg	White Plate with a lot of guacamole and an ext...
414112	133071	COCO_train2014_000000133071.jpg	A dinner plate has a lemon wedge garnishment.

414113 rows × 3 columns

Now we have a dataframe named data where three columns are present : image_id, image and caption .

```python
import os
import matplotlib.pyplot as plt
from tensorflow.keras.utils import load_img, img_to_array
from textwrap import wrap

# 1. Read image with full filename
def read_coco_image(filename, img_size=224, base_path='/kaggle/input/coco-image-caption/train2014/train2
    path = os.path.join(base_path, filename)
    if not os.path.exists(path):
        print(f"[Missing File] {path}")
        return None
    img = load_img(path, color_mode='rgb', target_size=(img_size, img_size))
    img = img_to_array(img) / 255.0
    return img
```

```python
# 2. Display a batch of images with captions
def display_images(df, img_size=224, base_path='/kaggle/input/coco-image-caption/train2014/train2014'):
    df = df.reset_index(drop=True)
    plt.figure(figsize=(20, 20))
    n = 0
    shown = 0
    for i in range(len(df)):
        image = read_coco_image(df.image[i], img_size=img_size, base_path=base_path)
        if image is None:
            continue
        n += 1
        plt.subplot(5, 5, n)
        plt.subplots_adjust(hspace=0.7, wspace=0.3)
        plt.imshow(image)
        plt.title("\n".join(wrap(df.caption[i], 20)))
        plt.axis("off")
        shown += 1
        if shown == 15:
            break
    plt.show()
```

+ Code + Markdown

[9]:
```python
display_images(data.sample(15))
```

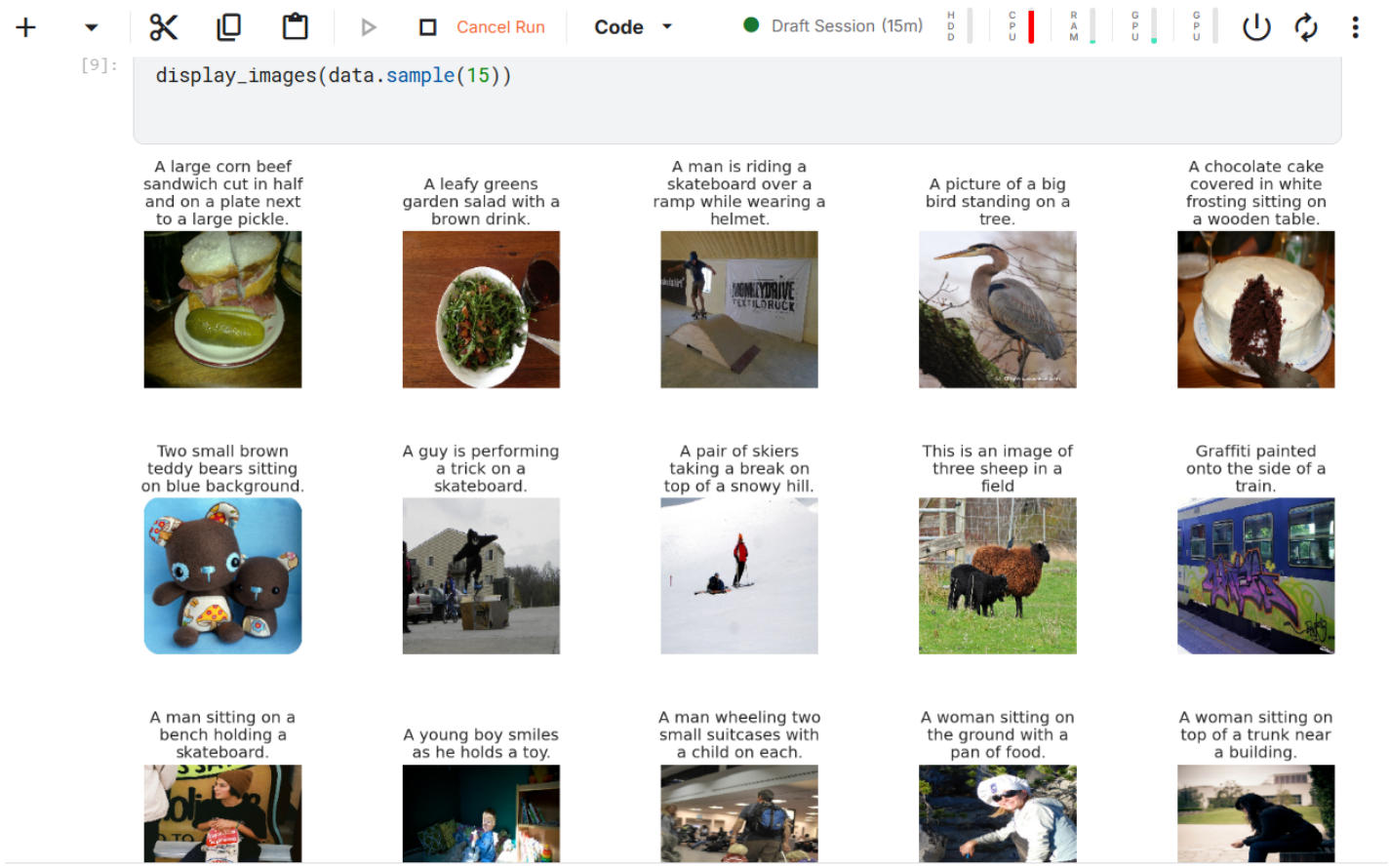
3. Image Loading and Visualization

This section presents the utility functions developed for reading images from the COCO dataset and visualizing a batch of images alongside their corresponding captions. Visualizing data samples helps verify the integrity of the dataset and gain insights into the image-caption pairs before model training.

Here we have created two functions:

1. Function to read a coco image: To read a single image file from the dataset, resize it, and normalize pixel values.
2. Function to display a batch of images with caption: To visualize up to 15 images from a DataFrame along with their respective captions.

The *read_coco_image* function standardizes images to a fixed size and normalizes pixel values for model compatibility. The *display_images* function helps in exploratory data analysis by displaying a batch of images with their captions, enabling manual inspection of data quality and caption relevance.



4. Text preprocessing

Text preprocessing is a crucial step in Natural Language Processing (NLP) that involves cleaning and transforming raw text data into a suitable format for analysis or modeling. This process helps improve data quality, enhance model performance, and reduce computational complexity.

There are various techniques in Natural language processing for text preprocessing which includes:

1. Text Cleaning

We'll convert the text to lowercase, remove punctuation, numbers, special characters, and HTML tags

```
import re

text = "Hello!!! This is an NLP example, with numbers like 123 and symbols #@%$."
text = text.lower()
text = re.sub(r'\d+', '', text)
text = re.sub(r'[^\w\s]', '', text)
text = re.sub(r'\s+', ' ', text).strip()
print(text)
```

```
hello this is an nlp example with numbers like and symbols
```

2. Tokenisation

Splitting the cleaned text into tokens (words).

```
import nltk
from nltk.tokenize import word_tokenize

# Download punkt tokenizer if not already present
nltk.download('punkt_tab')

# Your example text
text = "Hello! This is an NLP example with numbers like 123 and symbols #@%$."

# Correct tokenization
tokens = word_tokenize(text)

print(tokens)
```

```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
['Hello', '!', 'This', 'is', 'an', 'NLP', 'example', 'with', 'numbers', 'like', '123', 'and', 'symbols', '#', '@', '%', '$', '.']
```

Here we have used nltk(Natural language Toolkit) for the process of tokenisation.

3. Stop words removal

Removing common stop words from the tokens.

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('stopwords')

text = "This is a simple example showing how to remove stop words from text using NLTK."
words = word_tokenize(text)
stop_words = set(stopwords.words('english'))
filtered_words = [word for word in words if word.lower() not in stop_words]
print("Original Words:", words)
print("Filtered Words (without stop words):", filtered_words)
```

```
Original Words: ['This', 'is', 'a', 'simple', 'example', 'showing', 'how', 'to', 'remove', 'stop', 'words', 'from', 'text', 'using', 'NLTK', '.']
Filtered Words (without stop words): ['simple', 'example', 'showing', 'remove', 'stop', 'words', 'text', 'using', 'NLTK', '.']
```

4. Stemming and Lemmatization

Both stemming and lemmatization are techniques used to reduce words to their root form (i.e., normalization).

Stemming: Stemming cuts off prefixes or suffixes to get the root word.

```
[10] from nltk.stem import PorterStemmer

     stemmer = PorterStemmer()

     words = ["running", "runs", "ran", "easily", "fairly"]
     stems = [stemmer.stem(word) for word in words]
     print(stems)
```

```
['run', 'run', 'ran', 'easili', 'fairli']
```

Lemmatization: Lemmatization uses vocabulary and grammar rules to return the base or dictionary form (lemma) of a word.

```
import nltk
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

words = ["running", "runs", "ran", "better"]
lemmas = [lemmatizer.lemmatize(word, pos='v') for word in words]
print(lemmas)
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
['run', 'run', 'run', 'better']
```

In our code we have used the following text preprocessing techniques:

[9]:

```python
import re

def text_preprocessing(data):
    # Convert to lowercase
    data['caption'] = data['caption'].apply(lambda x: x.lower())

    # Remove non-alphabetical characters
    data['caption'] = data['caption'].apply(lambda x: re.sub(r"[^a-z\s]", "", x))

    # Replace multiple spaces with single space
    data['caption'] = data['caption'].apply(lambda x: re.sub(r"\s+", " ", x).strip())

    # (Optional) Remove meaningless single-character words (but keep 'a' and 'i')
    data['caption'] = data['caption'].apply(
        lambda x: " ".join([word for word in x.split() if len(word) > 1 or word in ['a', 'i']])
    )

    # Add start and end tokens
    data['caption'] = data['caption'].apply(lambda x: f"startseq {x} endseq")

    return data
```

1. Punctuation Removal (optional) – Removes symbols like *.,!?"* etc., if they do not add semantic value.
2. Token Insertion – Adding special tokens:

- *"startseq"* at the beginning
- *"endseq"* at the end
  These are critical for training sequence models.

3. Tokenization – Splitting sentences into words (or subwords).
4. Vocabulary Creation – Build a word-index dictionary (*word_index*, *index_word*) from all training captions.
5. Padding – Ensures all input sequences are the same length

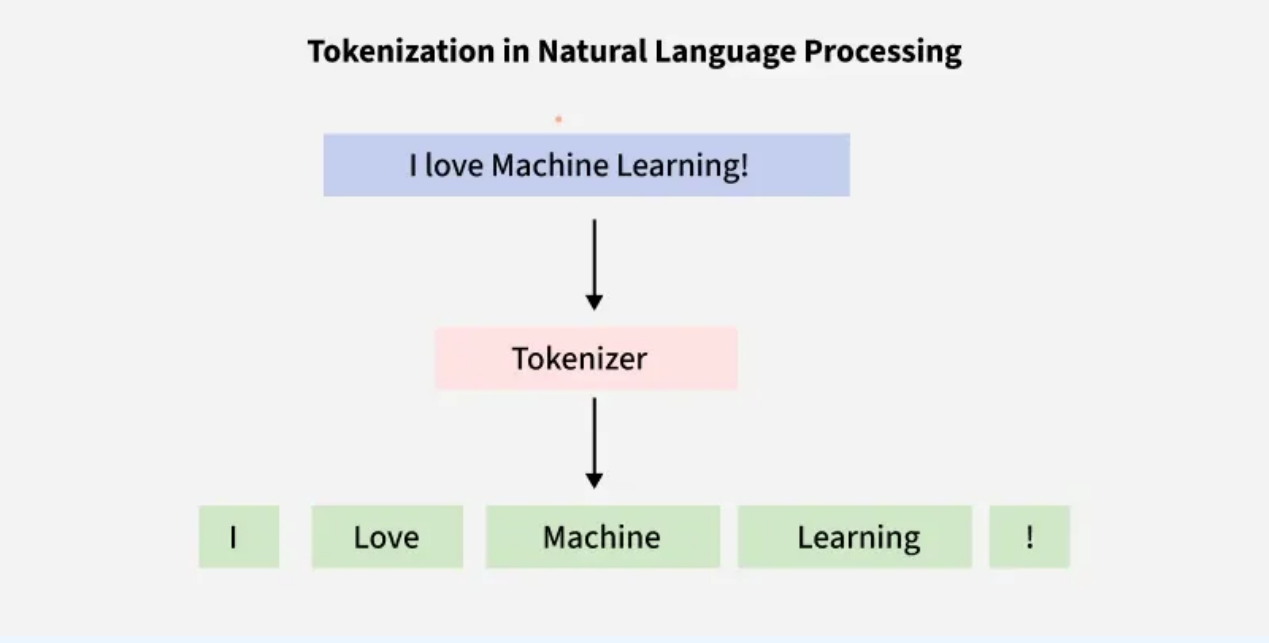
Example use case in our code :

[10]:
```
data = text_preprocessing(data)
captions = data['caption'].tolist()
print(captions[:10])
```

```
['startseq a very clean and well decorated empty bathroom endseq', 'startseq a panoramic view of a kitchen and al
l of its appliances endseq', 'startseq a blue and white bathroom with butterfly themed wall tiles endseq', 'start
seq a panoramic photo of a kitchen and dining room endseq', 'startseq a graffitied stop sign across the street fr
om a red car endseq', 'startseq a vandalized stop sign and a red beetle on the road endseq', 'startseq a bathroom
with a border of butterflies and blue paint on the walls above it endseq', 'startseq an angled view of a beautifu
lly decorated bathroom endseq', 'startseq the two people are walking down the beach endseq', 'startseq a sink and
a toilet inside a small bathroom endseq']
```

5. Tokenization:

Tokenization is a fundamental process in Natural Language Processing (NLP) that involves breaking down a piece of text into smaller units called tokens. These tokens can be words, subwords, or even characters, depending on the specific needs of the task at hand. Tokenization is typically the first step in the text preprocessing pipeline in NLP.



```python
from tensorflow.keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer()
tokenizer.fit_on_texts(data['caption'].tolist())

vocab_size = len(tokenizer.word_index) + 1

max_length = max(len(caption.split()) for caption in data['caption'])

images = data['image'].unique().tolist()
nimages = len(images)

split_index = round(0.85 * nimages)
train_images = images[:split_index]
val_images = images[split_index:]

train = data[data['image'].isin(train_images)].reset_index(drop=True)
test = data[data['image'].isin(val_images)].reset_index(drop=True)
```

This section of the code performs essential preprocessing steps for preparing text and image data used in the image captioning task. The operations include tokenizing the text captions, calculating vocabulary size and caption length, and splitting the dataset into training and validation sets.

This Step basically consists of:

1. Tokenization of Captions
2. Vocabulary Size Calculation
3. Maximum Caption Length
4. Image List and Count
5. Dataset Splitting (Train/Validation)
6. Creating Train and Validation DataFrames

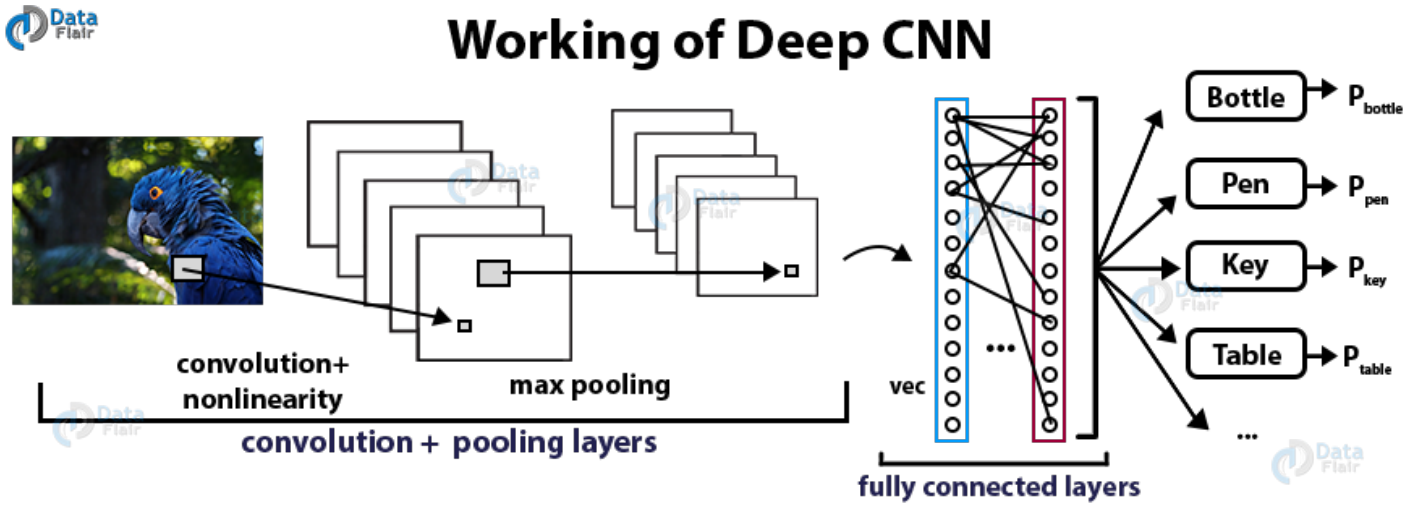
```python
sample_caption = train['caption'][1]
sequence = tokenizer.texts_to_sequences([sample_caption])[0]

print("Original Caption:", sample_caption)
print("Tokenized Sequence:", sequence)
```

```
Original Caption: startseq a panoramic view of a kitchen and all of its appliances endseq
Tokenized Sequence: [3, 2, 3974, 172, 6, 2, 62, 10, 317, 6, 114, 611, 4]
```

+ Code + Markdown

6. Feature extraction

Feature extraction using CNN (Convolutional Neural Network) is the process of using a CNN model to automatically identify and extract important patterns or features (such as edges, textures, shapes) from input images. These extracted features are then used for tasks like classification, captioning, segmentation, etc.



How CNN Works for Feature Extraction

Convolutional Neural Networks (CNNs) are deep learning models that automatically learn spatial hierarchies of features from images. They are extremely powerful for image tasks like classification, object detection, and segmentation.

CNN Layers

1. Input Image – e.g., 224×224×3 (RGB image).
2. Convolutional Layer – applies filters (kernels) to learn local features like edges, corners.
3. Activation Function – commonly ReLU, introduces non-linearity.
4. Pooling Layer – reduces spatial dimensions (e.g., max pooling), retaining dominant features.
5. Stacked Layers – deeper layers learn higher-level features (e.g., textures, object parts).

We have used EfficientNetB3 here. Let's understand what it is …

EfficientNet is a family of CNN'S that aims to achieve high performance with fewer computational resources compared to previous architectures. It was introduced by Mingxing Tan and Quoc V. Le from Google Research in their 2019 paper "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." The core idea behind EfficientNet is a new scaling method that uniformly scales all dimensions of depth, width, and resolution using a compound coefficient.

EfficientNet-B0 Architecture Overview
The EfficientNet-B0 network consists of:

1. **Stem:** Initial layer with a standard convolution followed by batch normalization and a ReLU6 activation. Convolution with 32 filters, kernel size 3x3, stride 2.
2. **Body:** Consists of a series of MBConv blocks with different configurations.Each block includes depthwise separable convolutions and squeeze-and-excitation layers.
3. **Head :**Includes a final convolutional block, followed by a global average pooling layer.A fully connected layer with a softmax activation function for classification.

Compound Scaling Method

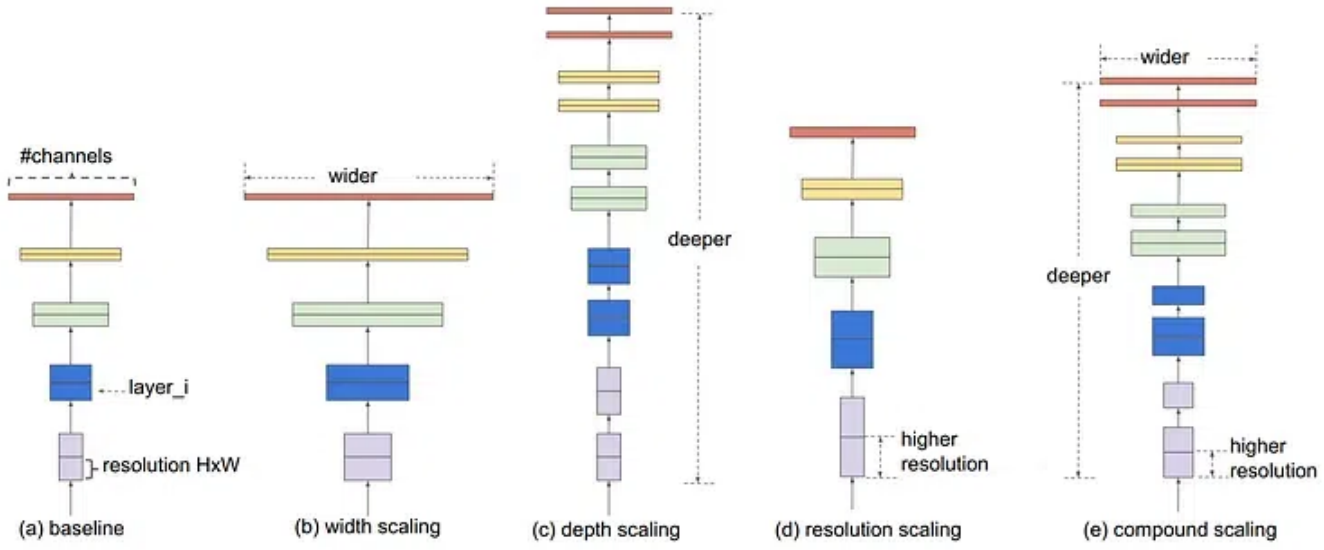
At the heart of EfficientNet lies a revolutionary compound scaling method, which orchestrates the simultaneous adjustment of network width, depth, and resolution using a set of fixed scaling coefficients. This approach ensures that the model adapts seamlessly to varying computational constraints while preserving its performance across different scales and tasks.

Compound Scaling:

The authors thoroughly investigated the effects that every scaling strategy has on the effectiveness and performance of the model before creating the compound scaling method. They came to the conclusion that, although scaling a single dimension can help improve model performance, the best way to increase model performance overall is to balance the scale in all three dimensions (width, depth, and image resolution) while taking the changeable available resources into consideration.

The below images show the different methods of scaling:

1. Baseline: The original network without scaling.
2. Width Scaling: Increasing the number of channels in each layer.
3. Depth Scaling: Increasing the number of layers.
4. Resolution Scaling: Increasing the input image resolution.
5. Compound Scaling: Simultaneously increasing width, depth, and resolution according to the compound scaling formula.



Different scaling methods vs. Compound scaling

This is achieved by uniformly scaling each dimension with a compound coefficient ϕ. The formula for scaling is:

$$\text{Width} \times \text{Depth}^2 \times \text{Resolution}^2 \approx \text{Constant}$$

```python
from tensorflow.keras.applications import EfficientNetB3
from tensorflow.keras.applications.efficientnet import preprocess_input
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.models import Model
import numpy as np
import os
from tqdm import tqdm
import pickle

image_path = '/kaggle/input/coco-image-caption/train2014/train2014'
output_path = '/kaggle/working/features_efficientnet.pkl'

base_model = EfficientNetB3(weights='imagenet', include_top=False, pooling='avg')
feature_extractor = Model(inputs=base_model.input, outputs=base_model.output)

img_size = 300
batch_size = 32

all_images = data['image'].unique().tolist()
features = {}
```

```python
for i in tqdm(range(0, len(all_images), batch_size), desc="Extracting features"):
    batch_images = all_images[i:i+batch_size]
    image_batch = []
    valid_filenames = []

    for image in batch_images:
        try:
            img_path = os.path.join(image_path, image)
            img = load_img(img_path, target_size=(img_size, img_size))
            img = img_to_array(img)
            img = preprocess_input(img)
            image_batch.append(img)
            valid_filenames.append(image)
        except:
            continue

    if image_batch:
        image_array = np.array(image_batch)
        feature_array = feature_extractor.predict(image_array, verbose=0)

        for j, fname in enumerate(valid_filenames):
            features[fname] = feature_array[j]

with open(output_path, 'wb') as f:
    pickle.dump(features, f)

print(f"✅ Features saved to {output_path}")
```

Output:

```
I0000 00:00:1749889544.092034      35 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:G
PU:0 with 13942 MB memory:  -> device: 0, name: Tesla T4, pci bus id: 0000:00:04.0, compute capability: 7.5
I0000 00:00:1749889544.092757      35 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:G
PU:1 with 13942 MB memory:  -> device: 1, name: Tesla T4, pci bus id: 0000:00:05.0, compute capability: 7.5
Downloading data from https://storage.googleapis.com/keras-applications/efficientnetb3_notop.h5
43941136/43941136 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Extracting features:   0%|          | 0/2587 [00:00<?, ?it/s]WARNING: All log messages before absl::InitializeLog
() is called are written to STDERR
I0000 00:00:1749889554.682045     107 service.cc:148] XLA service 0x7fb920004a20 initialized for platform CUDA (t
his does not guarantee that XLA will be used). Devices:
I0000 00:00:1749889554.682837     107 service.cc:156]   StreamExecutor device (0): Tesla T4, Compute Capability
7.5
I0000 00:00:1749889554.682858     107 service.cc:156]   StreamExecutor device (1): Tesla T4, Compute Capability
7.5
I0000 00:00:1749889555.855425     107 cuda_dnn.cc:529] Loaded cuDNN version 90300
I0000 00:00:1749889565.408601     107 device_compiler.h:188] Compiled cluster using XLA!  This line is logged at
most once for the lifetime of the process.
Extracting features: 100%|██████████| 2587/2587 [33:40<00:00,  1.28it/s]
✅ Features saved to /kaggle/working/features_efficientnet.pkl
```

7. Custom data generator:

This CustomDataGenerator class is a custom data generator used for training an image captioning model in Keras. It generates training samples on-the-fly, combining:

- image features extracted by a CNN (like EfficientNetB3),
- and corresponding partial caption sequences.

```python
from tensorflow.keras.utils import Sequence, to_categorical
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

class CustomDataGenerator(Sequence):
    def __init__(self, df, image_col, caption_col, batch_size, tokenizer,
                 vocab_size, max_length, features, shuffle=True):
        self.df = df.copy()
        self.image_col = image_col
        self.caption_col = caption_col
        self.batch_size = batch_size
        self.tokenizer = tokenizer
        self.vocab_size = vocab_size
        self.max_length = max_length
        self.features = features  # EfficientNet features (dict: image_id -> 1536-d np.array)
        self.shuffle = shuffle
        self.indexes = np.arange(len(self.df))
        self.on_epoch_end()

    def __len__(self):
        return int(np.floor(len(self.df) / self.batch_size))

    def on_epoch_end(self):
        if self.shuffle:
            np.random.shuffle(self.indexes)
```

```python
def __getitem__(self, index):
    batch_indexes = self.indexes[index * self.batch_size:(index + 1) * self.batch_size]
    batch = self.df.iloc[batch_indexes]

    X_img, X_seq, y = self.__data_generation(batch)
    return (X_img, X_seq), y
```

```
    def __data_generation(self, batch):
        X_img, X_seq, y = [], [], []

        for _, row in batch.iterrows():
            image_id = row[self.image_col]
            caption = row[self.caption_col]


            feature = self.features[image_id]

            seq = self.tokenizer.texts_to_sequences([caption])[0]

            for i in range(1, len(seq)):
                in_seq, out_seq = seq[:i], seq[i]
                in_seq_padded = pad_sequences([in_seq], maxlen=self.max_length)[0]
                out_seq_onehot = to_categorical([out_seq], num_classes=self.vocab_size)[0]

                X_img.append(feature)
                X_seq.append(in_seq_padded)
                y.append(out_seq_onehot)

        return np.array(X_img), np.array(X_seq), np.array(y)
```

```
train_generator = CustomDataGenerator(
    df=train,
    image_col='image',
    caption_col='caption',
    batch_size=64,
    tokenizer=tokenizer,
    vocab_size=vocab_size,
    max_length=max_length,
    features=features,
    shuffle=True
)

val_generator = CustomDataGenerator(
    df=test,
    image_col='image',
    caption_col='caption',
    batch_size=64,
    tokenizer=tokenizer,
    vocab_size=vocab_size,
    max_length=max_length,
    features=features,
    shuffle=False
)
```

8. Compiling Model

[17]:

```
from tensorflow.keras.layers import Input, Dense, Dropout, Embedding, LSTM, Add, Concatenate, Reshape
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

# Inputs
image_input = Input(shape=(1536,), name="image_input")  # EfficientNetB3 output
caption_input = Input(shape=(max_length,), name="caption_input")

# Image feature branch
img_dense = Dense(256, activation='relu')(image_input)
img_dense = Dropout(0.4)(img_dense)
img_reshaped = Reshape((1, 256))(img_dense)

# Caption branch
caption_embedding = Embedding(input_dim=vocab_size, output_dim=256)(caption_input)  # 👈 Removed mask_z
caption_lstm = LSTM(256, return_sequences=True)(caption_embedding)
caption_lstm = Dropout(0.3)(caption_lstm)

# Merge
merged = Concatenate(axis=1)([img_reshaped, caption_lstm])
merged = LSTM(256)(merged)
merged = Dropout(0.5)(merged)

# Skip connection
skip = Concatenate()([merged, img_dense])
dense1 = Dense(256, activation='relu')(skip)
dropout1 = Dropout(0.5)(dense1)
output = Dense(vocab_size, activation='softmax')(dropout1)
```

```
# Model
caption_model = Model(inputs=[image_input, caption_input], outputs=output)
caption_model.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=1e-4))

caption_model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
image_input (InputLayer)	(None, 1536)	0	-
caption_input (InputLayer)	(None, 51)	0	-
dense (Dense)	(None, 256)	393,472	image_input[0][0]
embedding (Embedding)	(None, 51, 256)	6,248,448	caption_input[0][0]
dropout (Dropout)	(None, 256)	0	dense[0][0]
lstm (LSTM)	(None, 51, 256)	525,312	embedding[0][0]
reshape (Reshape)	(None, 1, 256)	0	dropout[0][0]
dropout_1 (Dropout)	(None, 51, 256)	0	lstm[0][0]
concatenate (Concatenate)	(None, 52, 256)	0	reshape[0][0], dropout_1[0][0]
lstm_1 (LSTM)	(None, 256)	525,312	concatenate[0][0]

reshape (Reshape)	(None, 1, 256)	0	dropout[0][0]
dropout_1 (Dropout)	(None, 51, 256)	0	lstm[0][0]
concatenate (Concatenate)	(None, 52, 256)	0	reshape[0][0], dropout_1[0][0]
lstm_1 (LSTM)	(None, 256)	525,312	concatenate[0][0]
dropout_2 (Dropout)	(None, 256)	0	lstm_1[0][0]
concatenate_1 (Concatenate)	(None, 512)	0	dropout_2[0][0], dropout[0][0]
dense_1 (Dense)	(None, 256)	131,328	concatenate_1[0][0]
dropout_3 (Dropout)	(None, 256)	0	dense_1[0][0]
dense_2 (Dense)	(None, 24408)	6,272,856	dropout_3[0][0]

Total params: 14,096,728 (53.77 MB)

Trainable params: 14,096,728 (53.77 MB)

Non-trainable params: 0 (0.00 B)

[18]:

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau

checkpoint = ModelCheckpoint(
    "best_efficientnet_model.keras",   # <-- Renamed
    monitor="val_loss",
    mode="min",
    save_best_only=True,
    verbose=1
)

earlystopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True,
    verbose=1
)

lr_scheduler = ReduceLROnPlateau(
    monitor='val_loss',
    patience=3,
    factor=0.2,
    min_lr=1e-8,
    verbose=1
)

history = caption_model.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator,
```

```
Epoch 1/10
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 0s 262ms/step - loss: 5.0191
Epoch 1: val_loss improved from inf to 3.69239, saving model to best_efficientnet_model.keras
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 1687s 306ms/step - loss: 5.0190 - val_loss: 3.6924 - learning_rate: 1.0000e-04
Epoch 2/10
5213/5500 ━━━━━━━━━━━━━━━━━━━━ 1:11 248ms/step - loss: 3.5582
Epoch 2: val_loss improved from 3.69239 to 3.42323, saving model to best_efficientnet_model.keras
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 1605s 292ms/step - loss: 3.5553 - val_loss: 3.4232 - learning_rate: 1.0000e-04
Epoch 3/10
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 0s 250ms/step - loss: 3.3454
Epoch 3: val_loss improved from 3.42323 to 3.28532, saving model to best_efficientnet_model.keras
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 1613s 293ms/step - loss: 3.3454 - val_loss: 3.2853 - learning_rate: 1.0000e-04
Epoch 4/10
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 0s 250ms/step - loss: 3.2153
Epoch 4: val_loss improved from 3.28532 to 3.18520, saving model to best_efficientnet_model.keras
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 1618s 294ms/step - loss: 3.2153 - val_loss: 3.1852 - learning_rate: 1.0000e-04
Epoch 5/10
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 0s 249ms/step - loss: 3.1314
Epoch 5: val_loss improved from 3.18520 to 3.12762, saving model to best_efficientnet_model.keras
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 1608s 292ms/step - loss: 3.1314 - val_loss: 3.1276 - learning_rate: 1.0000e-04
Epoch 6/10
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 0s 249ms/step - loss: 3.0701
Epoch 6: val_loss improved from 3.12762 to 3.07950, saving model to best_efficientnet_model.keras
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 1611s 293ms/step - loss: 3.0701 - val_loss: 3.0795 - learning_rate: 1.0000e-04
Epoch 7/10
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 0s 248ms/step - loss: 3.0218
Epoch 7: val_loss improved from 3.07950 to 3.04218, saving model to best_efficientnet_model.keras
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 1605s 292ms/step - loss: 3.0218 - val_loss: 3.0422 - learning_rate: 1.0000e-04
Epoch 8/10
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 0s 248ms/step - loss: 2.9819
Epoch 8: val_loss improved from 3.04218 to 3.01303, saving model to best_efficientnet_model.keras
5500/5500 ━━━━━━━━━━━━━━━━━━━━ 1591s 289ms/step - loss: 2.9819 - val_loss: 3.0130 - learning_rate: 1.0000e-04
```

9. Plotting the loss

[20]:

```python
import matplotlib.pyplot as plt

def plot_training_loss(history):
    plt.figure(figsize=(20, 8))
    plt.plot(history.history['loss'], label='Training Loss', linewidth=2)
    plt.plot(history.history['val_loss'], label='Validation Loss', linewidth=2)
    plt.title('Model Training vs Validation Loss', fontsize=18)
    plt.xlabel('Epochs', fontsize=14)
    plt.ylabel('Loss', fontsize=14)
    plt.legend(loc='upper right', fontsize=12)
    plt.grid(True)
    plt.show()

# Call the function with your history object
plot_training_loss(history)
```

This graph provides us with the following points

1. **Consistent Learning**: Both training and validation loss steadily decrease over epochs, indicating effective learning and convergence.

2. **Good Generalization**: The small gap between training and validation loss suggests the model is not overfitting and generalizes well to unseen data.

3. **Improved Performance**: The model's loss decreased from ~4.3 to ~2.95 (training) and ~3.7 to ~3.0 (validation), showing significant performance improvement over 9 epochs.

10. Running the entire model as a function :

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.preprocessing.sequence import pad_sequences
import pickle
import os

def generate_caption_for_image(
    image_path,
    model_path="/kaggle/working/best_efficientnet_model.keras",
    tokenizer_path="/kaggle/working/tokenizer.pkl",
    feature_extractor_path="/kaggle/working/features_efficientnet.pkl",
    max_length=51,
    img_size=300
):
```

```
):
    # Load model
    caption_model = load_model(model_path)

    # Load tokenizer
    with open(tokenizer_path, "rb") as f:
        tokenizer = pickle.load(f)

    # Load feature extractor (EfficientNet features)
    with open(feature_extractor_path, "rb") as f:
        features = pickle.load(f)

    # Extract image name from path
    image_filename = os.path.basename(image_path)

    # If image is already in feature dict (precomputed)
    if image_filename in features:
        image_feature = features[image_filename]
    else:
        # Preprocess new image and extract feature using EfficientNetB3
        from tensorflow.keras.applications.efficientnet import EfficientNetB3, preprocess_input
        base_model = EfficientNetB3(weights='imagenet', include_top=False, pooling='avg')
        feature_extractor = tf.keras.Model(inputs=base_model.input, outputs=base_model.output)

        img = load_img(image_path, target_size=(img_size, img_size))
        img = img_to_array(img)
        img = preprocess_input(img)
        img = np.expand_dims(img, axis=0)
```

```
        image_feature = feature_extractor.predict(img, verbose=0)[0]

    # Start caption generation
    in_text = "startseq"
    for _ in range(max_length):
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        sequence = pad_sequences([sequence], maxlen=max_length)
        yhat = caption_model.predict([np.expand_dims(image_feature, axis=0), sequence], verbose=0)
        yhat_index = np.argmax(yhat)
        word = tokenizer.index_word.get(yhat_index, None)
        if word is None or word == "endseq":
            break
        in_text += " " + word

    caption = in_text.replace("startseq", "").strip().capitalize()

    # Show image with caption
    img_to_show = load_img(image_path)
    plt.figure(figsize=(8, 8))
    plt.imshow(img_to_show)
    plt.axis("off")
    plt.title(caption, fontsize=16, color='darkblue')
    plt.show()

    return caption
```

11. Results :

[28]:
```
generate_caption_for_image(
    "/kaggle/input/coco-image-caption/val2017/val2017/000000000885.jpg"

)
```

5. Image Segmentation

Image segmentation is a computer vision technique that involves dividing an image into meaningful parts (segments), usually by labeling each pixel according to what object or region it belongs to.

Type	Description	Example Use Case
Semantic Segmentation	Classifies each pixel into a class (e.g., dog, road), but doesn't distinguish between different instances of the same class.	Self-driving cars (road, lane, pedestrian)
Instance Segmentation	Like semantic segmentation, but also separates different objects of the same class (e.g., 3 dogs → dog1, dog2, dog3).	Medical imaging (separate tumors)
Panoptic Segmentation	Combines semantic + instance: all pixels get a class label, and countable objects get instance IDs too.	Advanced scene understanding

(a) Image

(b) Semantic segmentation

(c) Instance segmentation

(d) Panoptic segmentation

```
# Import all the libraries
import numpy as np
import cv2
import requests
import os
import imutils
from PIL import Image
from tqdm.notebook import tqdm
########3
from pycocotools import coco, cocoeval, _mask
from pycocotools import mask as maskUtils
from pycocotools.coco import COCO
import skimage.io as io
import random
#from tensorflow.keras.preprocessing.image import ImageDataGenerator

### For visualizing the outputs ###
import matplotlib.pyplot as plt
```

```
%matplotlib inline
from random import shuffle

from PIL import Image
import sys
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import *
from tensorflow.keras.optimizers import *
from tensorflow.keras.layers import *
from tensorflow.keras.metrics import *
```

I. Importing libraries

import NumPy, a library for numerical computations, and aliases it as `np`.
Importing OpenCV, a library for image processing and computer vision tasks.
Importing requests ,Allows you to make HTTP requests (e.g., to download images or data).
Importing OS , Provides functions to interact with the operating system (e.g., file paths, directories).
Importing `imutils`, a convenience library that simplifies common OpenCV functions (e.g., resizing, rotation).
Importing PIL , `Image` class for image handling.
Importing `tqdm` for creating progress bars in Jupyter Notebooks.
Importing pycocotools , Imports modules to work with the COCO dataset: annotations (`coco`), evaluation metrics (`cocoeval`), and mask handling (`_mask`).
Imports the `mask` module from `pycocotools` as `maskUtils`, useful for mask operations like encoding/decoding.
Imports the COCO class to load and parse COCO dataset annotations.
Import skimage.io ,Imports image input/output functions from skimage (scikit-image), usually for reading and saving images.
Importing Matplotlib.pyplot, used for plotting and visualizing data (e.g., images, graphs).
Importing Keras , used for plotting and visualizing data (e.g., images, graphs).
importing keras.layers , keras.optimizers and keras.metrics , Imports all optimizers (like Adam, SGD), layers (like Dense, Conv2D), and evaluation metrics (like accuracy, precision) used in deep learning models.

Loading Annotations from .json file using pycoco.COCO

```
coco_train = COCO(ANNOTATION_FILE_TRAIN)
catIds_train = coco_train.getCatIds() # Get all Categories ('horse','human' etc...)
imgIds_train = coco_train.getImgIds() # Get all image ID's (dict with path and annotations)
imgDict_train = coco_train.loadImgs(imgIds_train) # Func to load images from path
print(len(imgIds_train) , len(catIds_train))
```

```
loading annotations into memory...
Done (t=19.11s)
creating index...
index created!
118287 80
```

```python
# Create directories to save masks
os.makedirs(MASK_TRAIN_DIR, exist_ok=True)
os.makedirs(MASK_VAL_DIR, exist_ok=True)

# Load annotations
coco_train = COCO(os.path.join(ANNOTATIONS_PATH, 'instances_train2017.json'))
coco_val = COCO(os.path.join(ANNOTATIONS_PATH, 'instances_val2017.json'))

# Get category info and class mappings
cat_ids = coco_train.getCatIds()
cat_id_to_label = {cat_id: idx + 1 for idx, cat_id in enumerate(cat_ids)}  # label 0 is background

# Function to generate and save masks
def generate_multiclass_masks(coco, img_ids, save_dir):
    for img_id in tqdm(img_ids):
        anns = coco.loadAnns(coco.getAnnIds(imgIds=img_id, iscrowd=0))
        if len(anns) == 0:
            continue

        img_info = coco.loadImgs(img_id)[0]
        height, width = img_info['height'], img_info['width']
        mask = np.zeros((height, width), dtype=np.uint8)

        for ann in anns:
            cat_id = ann['category_id']
            label = cat_id_to_label[cat_id]
            ann_mask = coco.annToMask(ann)
            mask[ann_mask == 1] = label  # overwrite with class label

        # Save mask as PNG
        filename = img_info['file_name'].replace('.jpg', '.png')
        mask_path = os.path.join(save_dir, filename)
        Image.fromarray(mask, mode='L').save(mask_path)

# Generate masks
print("Generating training masks...")
generate_multiclass_masks(coco_train, coco_train.getImgIds(), MASK_TRAIN_DIR)

print("Generating validation masks...")
generate_multiclass_masks(coco_val, coco_val.getImgIds(), MASK_VAL_DIR)
```

This code is designed to generate **multi-class segmentation masks** from COCO dataset annotations, which are commonly used for training image segmentation models. It begins by creating directories to store training and validation masks. Then, it loads COCO-style JSON annotation files (`instances_train2017.json` and `instances_val2017.json`) using the `pycocotools` API. The script maps category IDs to numerical labels, assigning label `0` to the background and other labels to object classes.

The core function `generate_multiclass_masks` processes each image in the dataset by retrieving its annotations and creating a blank mask. For every object in the image, its binary mask is extracted and its pixels are assigned the corresponding class label. If objects overlap, later masks overwrite earlier ones. The final labeled mask is saved as a grayscale `.png` image, where each pixel value corresponds to a class ID. This entire process is run separately for both training and validation datasets, resulting in labeled masks suitable for training deep learning models like U-Net for semantic segmentation tasks.

COCO Dataset Generator

```python
class COCODataGenerator(Sequence):
    def __init__(self,
                 image_dir,
                 mask_dir,
                 image_filenames,
                 batch_size=16,
                 image_size=(128, 128),
                 n_classes=81,               # 80 classes + background
                 shuffle=True,
                 augment_fn=None,
                 one_hot=True,
                 normalize=True,**kwargs):

        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.image_filenames = image_filenames
        self.batch_size = batch_size
        self.image_size = image_size
        self.n_classes = n_classes
        self.shuffle = shuffle
        self.augment_fn = augment_fn
        self.one_hot = one_hot
        self.normalize = normalize
        self.on_epoch_end()
        super().__init__(**kwargs)

    def __len__(self):
        return int(np.ceil(len(self.image_filenames) / self.batch_size))

    def on_epoch_end(self):
        if self.shuffle:
            random.shuffle(self.image_filenames)


    def __getitem__(self, idx):
        batch_filenames = self.image_filenames[idx * self.batch_size:(idx + 1) * self.batch_size]
        images = []
        masks = []

        for filename in batch_filenames:
            # Load image and mask
            image_path = os.path.join(self.image_dir, filename)
            mask_path = os.path.join(self.mask_dir, filename.replace(".jpg",".png"))

            img = cv2.imread(image_path)
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            # img = cv2.imread(image_path)[..., ::-1]  # Converts BGR to RGB
            mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)

            # Resize
            img = cv2.resize(img, self.image_size, interpolation=cv2.INTER_LINEAR)
            mask = cv2.resize(mask, self.image_size, interpolation=cv2.INTER_NEAREST)

            # Optional augmentation
            if self.augment_fn:
                augmented = self.augment_fn(image=img, mask=mask)
                img, mask = augmented['image'], augmented['mask']

            # Normalize image
            if self.normalize:
                img = img / 255.0

            # One-hot encode mask
            if self.one_hot:
                mask = to_categorical(mask, num_classes=self.n_classes)

            images.append(img)
            masks.append(mask)

        return np.array(images, dtype=np.float32), np.array(masks, dtype=np.float32)
```

This code defines a custom data generator class `COCODataGenerator`, designed to load images and their corresponding segmentation masks for training deep learning models (e.g., semantic segmentation). It inherits from `Sequence`, making it compatible with Keras training loops. The generator loads image-mask pairs in batches, applies optional data augmentation, normalizes the image pixel values, and optionally one-hot encodes the masks (converting pixel values into categorical class channels). It shuffles the dataset at the end of each epoch if specified. Inside the `__getitem__` method, it reads and resizes the images and masks using OpenCV, applies the augmentation function if provided, and returns NumPy arrays of images and masks, ready for training. This generator is particularly useful for efficient memory usage and dynamic preprocessing during training on large datasets like COCO.

```
train_mask_filenames = os.listdir(MASK_TRAIN_DIR)

# Derive image filenames from mask filenames
train_image_filenames = [f.replace('.png', '.jpg') for f in train_mask_filenames]
```

```
train_generator = COCODataGenerator(
    image_dir='/kaggle/input/coco-2017-dataset/coco2017/train2017',
    mask_dir=MASK_TRAIN_DIR,
    image_filenames=train_image_filenames,
    batch_size=32,
    image_size=(128, 128),
    n_classes=81,
    shuffle=True,
    one_hot=True,
    normalize=True
)
```

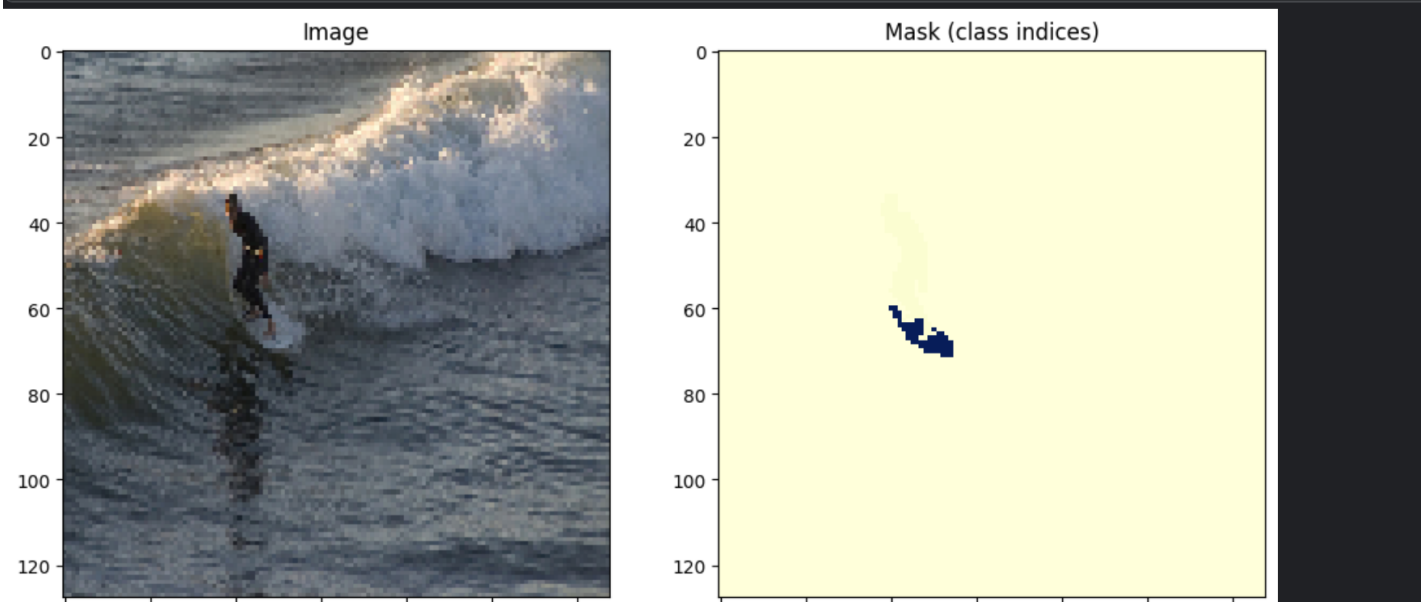
Using Custom dataset generator to get the training datasets.

In [26]:
```
x_train_batch, y_train_batch = train_generator[5]

print("Image batch shape:", x_train_batch.shape)  # (batch_size, height, width, 3)
print("Mask batch shape:", y_train_batch.shape)   # (batch_size, height, width, n_classes)
```

```
Image batch shape: (32, 128, 128, 3)
Mask batch shape: (32, 128, 128, 81)
```

Visualising the data batches generated by custom data generator and mask generator

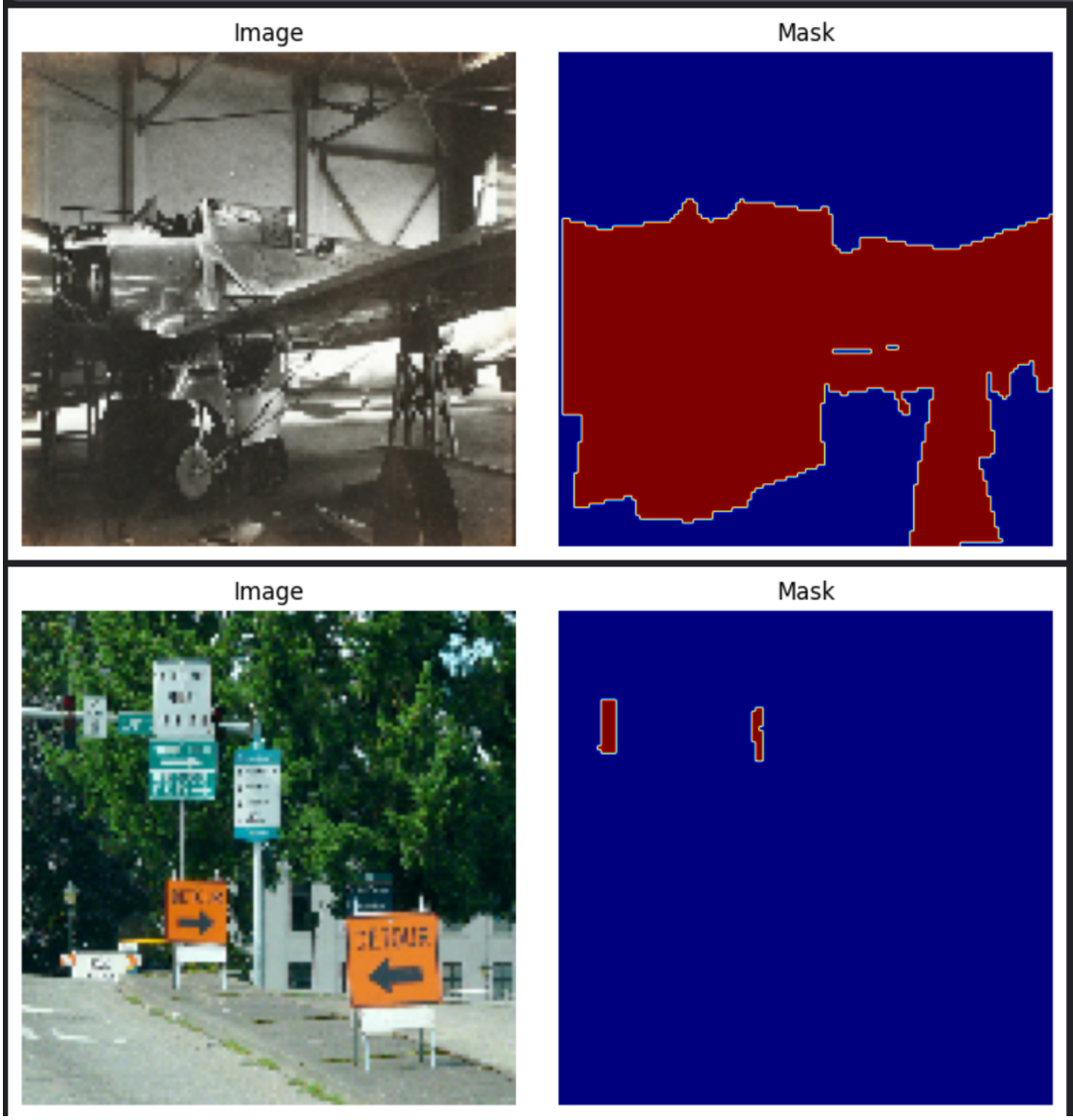
```
# Index into the batch, e.g., get the first image and mask from the first batch
x_img = x_val_batch[0]
y_mask = y_val_batch[0]

# If you want to visualize the original class labels from the one-hot mask
import numpy as np
y_mask_labels = np.argmax(y_mask, axis=-1)  # shape: (height, width)

# Or visualize with matplotlib
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Image")
plt.imshow(x_img)
plt.subplot(1, 2, 2)
plt.title("Mask (class indices)")
plt.imshow(y_mask_labels,cmap = "YlGnBu")
plt.show()
```




```
import matplotlib.pyplot as plt

def visualize_batch(generator, num_samples=2):
    x_batch, y_batch = generator[0]  # Get a batch
    for i in range(num_samples):
        img = x_batch[i]
        mask = y_batch[i]

        plt.figure(figsize=(8, 4))

        # Show image
        plt.subplot(1, 2, 1)
        plt.imshow(img)
        plt.title("Image")
        plt.axis("off")
```

```
        # If multiclass, use argmax
        plt.subplot(1, 2, 2)
        if mask.shape[-1] > 1:
            plt.imshow(mask.argmax(axis=-1), cmap="jet")
        else:
            plt.imshow(mask.squeeze(), cmap="gray")
        plt.title("Mask")
        plt.axis("off")

        plt.tight_layout()
        plt.show()
```

```
visualize_batch(train_generator, num_samples=2)
```



Creating the U_net Model

```
from tensorflow.keras.layers import *
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import tensorflow as tf

n_classes = 81  # For COCO 2017 (80 + 1 background)
image_size = 128

model_in = Input(shape=(image_size, image_size, 3))

# Downsampling path
conv1 = BatchNormalization()(model_in)
conv1 = Conv2D(64, 3, padding='same', activation='relu', kernel_initializer='he_normal')(conv1)
conv1 = Conv2D(64, 3, padding='same', kernel_initializer='he_normal')(conv1)
conv1 = BatchNormalization()(conv1)
conv1 = Activation('relu')(conv1)
down1 = MaxPooling2D()(conv1)

conv2 = Conv2D(128, 3, padding='same', activation='relu', kernel_initializer='he_normal')(down1)
conv2 = Conv2D(128, 3, padding='same', kernel_initializer='he_normal')(conv2)
conv2 = BatchNormalization()(conv2)
conv2 = Activation('relu')(conv2)
down2 = MaxPooling2D()(conv2)

conv3 = Conv2D(256, 3, padding='same', activation='relu', kernel_initializer='he_normal')(down2)
conv3 = Conv2D(256, 3, padding='same', kernel_initializer='he_normal')(conv3)
conv3 = BatchNormalization()(conv3)
conv3 = Activation('relu')(conv3)
down3 = MaxPooling2D()(conv3)

conv4 = Conv2D(512, 3, padding='same', activation='relu', kernel_initializer='he_normal')(down3)
conv4 = Conv2D(512, 3, padding='same', kernel_initializer='he_normal')(conv4)
conv4 = BatchNormalization()(conv4)
conv4 = Activation('relu')(conv4)
down4 = MaxPooling2D()(conv4)

conv5 = Conv2D(1024, 3, padding='same', activation='relu', kernel_initializer='he_normal')(down4)
conv5 = Conv2D(1024, 3, padding='same', activation='relu', kernel_initializer='he_normal')(conv5)

# Upsampling path
conv6 = BatchNormalization()(conv5)
conv6 = Conv2DTranspose(512, 2, strides=2, padding='same', kernel_initializer='he_normal')(conv6)
conv6 = Activation('relu')(conv6)
conv6 = concatenate([conv4, conv6])
conv6 = Conv2D(512, 3, padding='same', activation='relu', kernel_initializer='he_normal')(conv6)
conv6 = Conv2D(512, 3, padding='same', kernel_initializer='he_normal')(conv6)

conv7 = BatchNormalization()(conv6)
conv7 = Activation('relu')(conv7)
conv7 = Conv2DTranspose(256, 2, strides=2, padding='same', kernel_initializer='he_normal')(conv7)
conv7 = concatenate([conv3, conv7])
conv7 = Conv2D(256, 3, padding='same', activation='relu', kernel_initializer='he_normal')(conv7)
conv7 = Conv2D(256, 3, padding='same', kernel_initializer='he_normal')(conv7)

conv8 = BatchNormalization()(conv7)
conv8 = Activation('relu')(conv8)
conv8 = Conv2DTranspose(128, 2, strides=2, padding='same', kernel_initializer='he_normal')(conv8)
conv8 = concatenate([conv2, conv8])
conv8 = Conv2D(128, 3, padding='same', activation='relu', kernel_initializer='he_normal')(conv8)
conv8 = Conv2D(128, 3, padding='same', kernel_initializer='he_normal')(conv8)

conv9 = BatchNormalization()(conv8)
conv9 = Activation('relu')(conv9)
conv9 = Conv2DTranspose(64, 2, strides=2, padding='same', kernel_initializer='he_normal')(conv9)
conv9 = concatenate([conv1, conv9])
conv9 = Conv2D(64, 3, padding='same', activation='relu', kernel_initializer='he_normal')(conv9)
conv9 = Conv2D(64, 3, padding='same', activation='relu', kernel_initializer='he_normal')(conv9)

# Output layer
output = Conv2D(n_classes, 1, activation='softmax', padding='same')(conv9)

model = Model(inputs=model_in, outputs=output)
```

```
model.summary()
```

Model: "functional_2"

Layer (type)	Output Shape	Param #	Connected to
input_layer_2 (InputLayer)	(None, 128, 128, 3)	0	–
batch_normalization_22 (BatchNormalization)	(None, 128, 128, 3)	12	input_layer_2[0][0]
conv2d_38 (Conv2D)	(None, 128, 128, 64)	1,792	batch_normalization_2…
conv2d_39 (Conv2D)	(None, 128, 128, 64)	36,928	conv2d_38[0][0]
batch_normalization_23 (BatchNormalization)	(None, 128, 128, 64)	256	conv2d_39[0][0]
max_pooling2d_8 (MaxPooling2D)	(None, 64, 64, 64)	0	batch_normalization_2…
conv2d_40 (Conv2D)	(None, 64, 64, 128)	73,856	max_pooling2d_8[0][0]
conv2d_41 (Conv2D)	(None, 64, 64, 128)	147,584	conv2d_40[0][0]
batch_normalization_24 (BatchNormalization)	(None, 64, 64, 128)	512	conv2d_41[0][0]
max_pooling2d_9 (MaxPooling2D)	(None, 32, 32, 128)	0	batch_normalization_2…
conv2d_42 (Conv2D)	(None, 32, 32, 256)	295,168	max_pooling2d_9[0][0]
conv2d_43 (Conv2D)	(None, 32, 32, 256)	590,080	conv2d_42[0][0]
batch_normalization_25 (BatchNormalization)	(None, 32, 32, 256)	1,024	conv2d_43[0][0]
max_pooling2d_10 (MaxPooling2D)	(None, 16, 16, 256)	0	batch_normalization_2…
batch_normalization_26 (BatchNormalization)	(None, 16, 16, 256)	1,024	max_pooling2d_10[0][0]
conv2d_44 (Conv2D)	(None, 16, 16, 512)	1,180,160	batch_normalization_2…
conv2d_45 (Conv2D)	(None, 16, 16, 512)	2,359,808	conv2d_44[0][0]
batch_normalization_27 (BatchNormalization)	(None, 16, 16, 512)	2,048	conv2d_45[0][0]
max_pooling2d_11 (MaxPooling2D)	(None, 8, 8, 512)	0	batch_normalization_2…
batch_normalization_28 (BatchNormalization)	(None, 8, 8, 512)	2,048	max_pooling2d_11[0][0]
conv2d_46 (Conv2D)	(None, 8, 8, 1024)	4,719,616	batch_normalization_2…
conv2d_47 (Conv2D)	(None, 8, 8, 1024)	9,438,208	conv2d_46[0][0]
batch_normalization_29 (BatchNormalization)	(None, 8, 8, 1024)	4,096	conv2d_47[0][0]
conv2d_transpose_8 (Conv2DTranspose)	(None, 16, 16, 512)	2,097,664	batch_normalization_2…

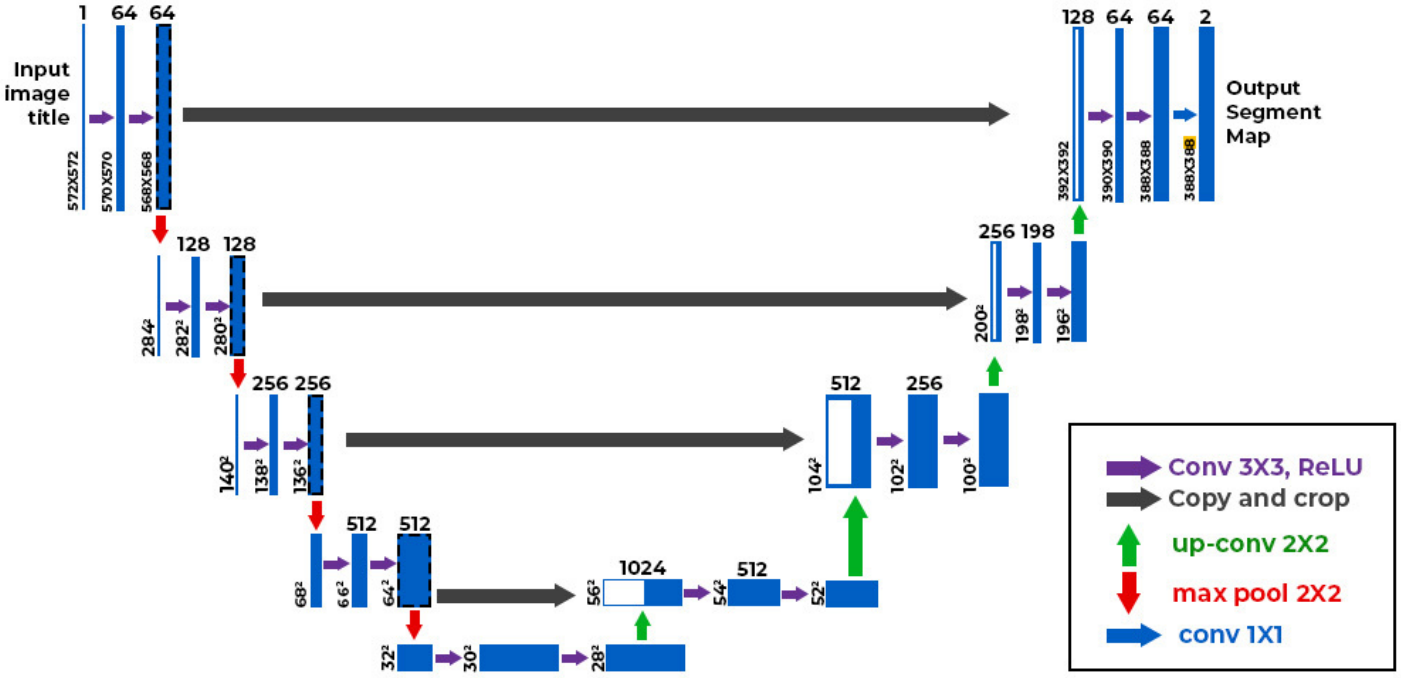
concatenate_8 (Concatenate)	(None, 16, 16, 1024)	0	batch_normalization_2… conv2d_transpose_8[0]…
conv2d_48 (Conv2D)	(None, 16, 16, 512)	4,719,104	concatenate_8[0][0]
conv2d_49 (Conv2D)	(None, 16, 16, 512)	2,359,808	conv2d_48[0][0]
batch_normalization_30 (BatchNormalization)	(None, 16, 16, 512)	2,048	conv2d_49[0][0]
conv2d_transpose_9 (Conv2DTranspose)	(None, 32, 32, 256)	524,544	batch_normalization_3…
concatenate_9 (Concatenate)	(None, 32, 32, 512)	0	batch_normalization_2… conv2d_transpose_9[0]…
conv2d_50 (Conv2D)	(None, 32, 32, 256)	1,179,904	concatenate_9[0][0]
conv2d_51 (Conv2D)	(None, 32, 32, 256)	590,080	conv2d_50[0][0]
batch_normalization_31 (BatchNormalization)	(None, 32, 32, 256)	1,024	conv2d_51[0][0]
conv2d_transpose_10 (Conv2DTranspose)	(None, 64, 64, 128)	131,200	batch_normalization_3…
concatenate_10 (Concatenate)	(None, 64, 64, 256)	0	batch_normalization_2… conv2d_transpose_10[0…
conv2d_52 (Conv2D)	(None, 64, 64, 128)	295,040	concatenate_10[0][0]
conv2d_53 (Conv2D)	(None, 64, 64, 128)	147,584	conv2d_52[0][0]
batch_normalization_32 (BatchNormalization)	(None, 64, 64, 128)	512	conv2d_53[0][0]
conv2d_transpose_11 (Conv2DTranspose)	(None, 128, 128, 128)	65,664	batch_normalization_3…
concatenate_11 (Concatenate)	(None, 128, 128, 192)	0	batch_normalization_2… conv2d_transpose_11[0…
conv2d_54 (Conv2D)	(None, 128, 128, 64)	110,656	concatenate_11[0][0]
conv2d_55 (Conv2D)	(None, 128, 128, 64)	36,928	conv2d_54[0][0]
conv2d_56 (Conv2D)	(None, 128, 128, 1)	65	conv2d_55[0][0]

Total params: 31,116,045 (118.70 MB)

Trainable params: 31,108,743 (118.67 MB)

Non-trainable params: 7,302 (28.52 KB)

U_NET Architecture MAP



Custom Metrics for calculating MeanIoU

```
class MeanIoUMetric(tf.keras.metrics.Metric):
    def __init__(self, num_classes, name='mean_iou', **kwargs):
        super(MeanIoUMetric, self).__init__(name=name, **kwargs)
        self.num_classes = num_classes
        self.iou = tf.keras.metrics.MeanIoU(num_classes=num_classes)

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_true = tf.argmax(y_true, axis=-1)
        y_pred = tf.argmax(y_pred, axis=-1)
        return self.iou.update_state(y_true, y_pred, sample_weight)

    def result(self):
        return self.iou.result()

    def reset_states(self):
        return self.iou.reset_states()
```

```python
model.compile(optimizer=AdamW(),
              loss='categorical_crossentropy',
              metrics=['accuracy', MeanIoUMetric(num_classes=n_classes)])
steps_per_epoch = int(np.ceil(len(train_image_filenames)/32))

# Load model with custom metric

from tensorflow.keras.callbacks import ModelCheckpoint



# Update callbacks if needed
checkpoint = ModelCheckpoint(
    filepath='/kaggle/working/unet_adamw_multi_class_12.keras',
    monitor='val_loss',
    verbose=1,
    save_best_only=True,
    save_weights_only=False
)

logdir = WORKING_DIR + "/logs/scalars/" + datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir)

# Train for more epochs (e.g., continue training)
history = model.fit(
    train_generator,
    validation_data=val_generator,
    steps_per_epoch=steps_per_epoch,
    initial_epoch = 10,
    epochs=12,  # ← Update this to current total desired epoch count
    callbacks=[tensorboard_callback, checkpoint],
    verbose=1
)
```

```
/usr/local/lib/python3.11/dist-packages/tensorflow/python/data/ops/structured_function.py:258: UserWarning: Even tho
ugh the `tf.config.experimental_run_functions_eagerly` option is set, this option does not apply to tf.data function
s. To force eager execution of tf.data functions, please use `tf.data.experimental.enable_debug_mode()`.
  warnings.warn(
Epoch 11/12
I0000 00:00:1750600295.822182      35 cuda_dnn.cc:529] Loaded cuDNN version 90300
3665/3665 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.8536 - loss: 0.5030 - mean_iou: 0.2980
/usr/local/lib/python3.11/dist-packages/tensorflow/python/data/ops/structured_function.py:258: UserWarning: Even tho
ugh the `tf.config.experimental_run_functions_eagerly` option is set, this option does not apply to tf.data function
s. To force eager execution of tf.data functions, please use `tf.data.experimental.enable_debug_mode()`.
  warnings.warn(

Epoch 11: val_loss improved from inf to 0.68315, saving model to /kaggle/working/unet_adamw_multi_class_12.keras
3665/3665 ━━━━━━━━━━━━━━━━━━━━ 4305s 1s/step - accuracy: 0.8536 - loss: 0.5030 - mean_iou: 0.2980 - val_accuracy: 0.
8183 - val_loss: 0.6832 - val_mean_iou: 0.2216
Epoch 12/12
3665/3665 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.8626 - loss: 0.4680 - mean_iou: 0.3227
Epoch 12: val_loss did not improve from 0.68315
3665/3665 ━━━━━━━━━━━━━━━━━━━━ 4228s 1s/step - accuracy: 0.8626 - loss: 0.4680 - mean_iou: 0.3227 - val_accuracy: 0.
8195 - val_loss: 0.7138 - val_mean_iou: 0.2274
```

Brief Explanation: ModelCheckpoint in TensorFlow/Keras

The ModelCheckpoint callback in TensorFlow/Keras is used to **automatically save the model during training**. In this code, it is configured to monitor the **validation loss (val_loss)** and save the model whenever it achieves a new best score.

- filepath='/kaggle/working/unet_adamw_multi_class_12.keras': Specifies where to save the model file.
- monitor='val_loss': Watches the validation loss after each epoch.

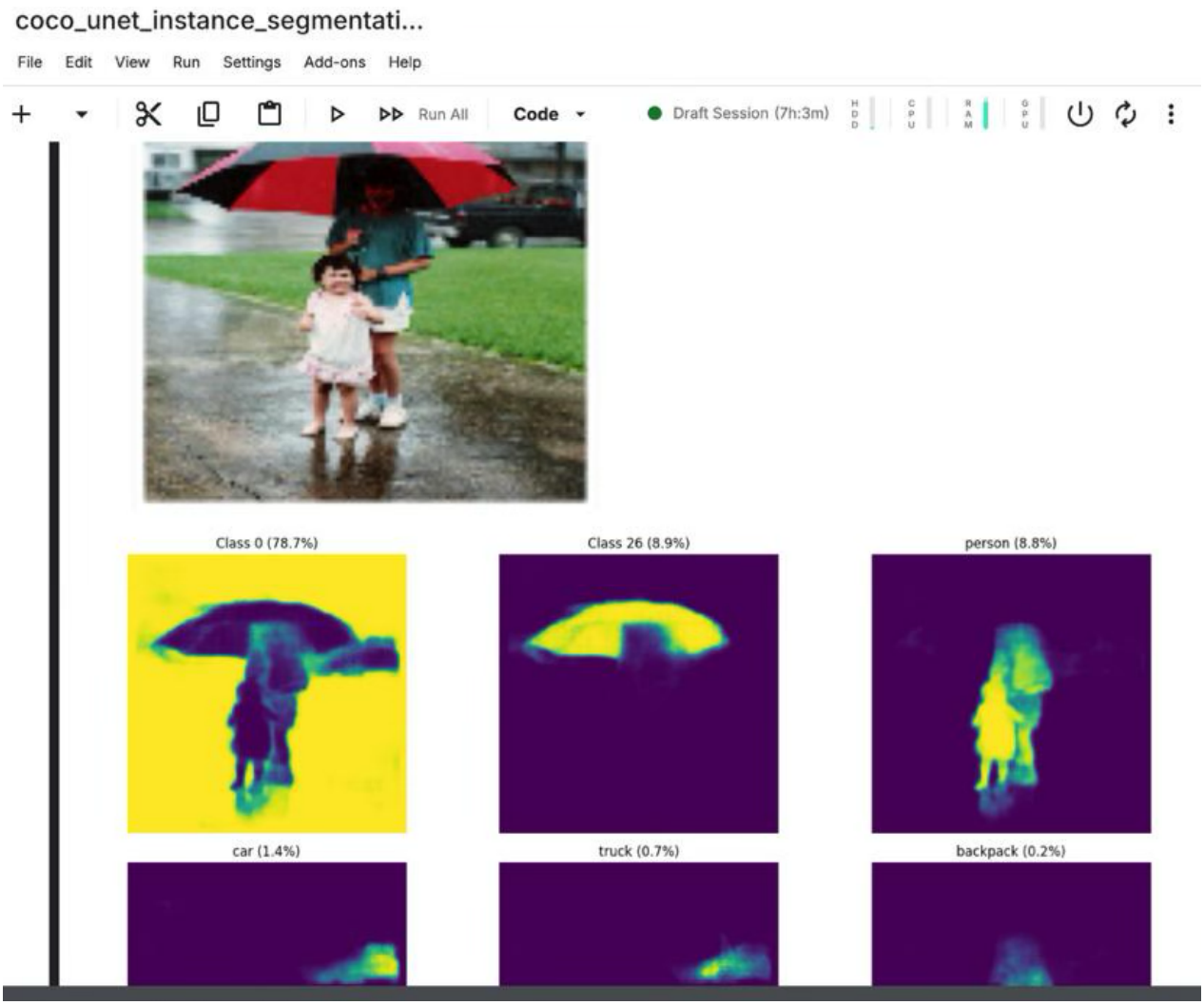
- save_best_only=True: Saves the model only when the val_loss improves (prevents overwriting with worse models).
- save_weights_only=False: Saves the **entire model**, not just weights.
- verbose=1: Prints messages when the model is saved.

This is crucial for preventing loss of the best model due to overfitting or training instability, allowing you to reload the most optimal version later for inference or fine-tuning.

Visualisation of Model's Prediction

```
model = tf.keras.models.load_model(
    "/kaggle/working/unet_adamw_multi_class_epoch_25.h5",custom_objects={"mean_iou":MeanIoUMetric(num_classes=n_classes)}
)
nu = np.random.randint(289)
x, y = val_generator[nu]
y_probas = np.stack([model(x, training=True) for _ in range(10)])
y_pred = y_probas.mean(axis=0)
# Original image

plt.imshow(x[0])
plt.title("Input Image")
plt.axis('off')

import numpy as np
import matplotlib.pyplot as plt

# y_pred shape: (1, H, W, C)
y_pred_probs = y_pred[0]  # shape: (H, W, C)

# 1. Total confidence per class
class_confidence = y_pred_probs.sum(axis=(0, 1))  # shape: (C,)
total_confidence = class_confidence.sum()

# 2. Normalize to get fraction of confidence per class
class_fraction = class_confidence / total_confidence

# 3. Get class indices above threshold
threshold = 0.04  # 4% (change as needed)
valid_indices = np.where(class_fraction >= threshold)[0]

print(f"Classes with ≥ {threshold:.0%} of total confidence:", valid_indices)

# Limit to max 6 for plotting
top_k = min(len(valid_indices), 6)
selected_class_ids = valid_indices[np.argsort(class_confidence[valid_indices])[::-1][:top_k]]

fig, axes = plt.subplots(2, 3, figsize=(15, 8))
axes = axes.flatten()
```

```python
for i, class_id in enumerate(selected_class_ids):
    axes[i].imshow(y_pred_probs[:, :, class_id], cmap="jet")

    try:
        cat_info = coco_val.loadCats(int(class_id))[0]
        cat_name = cat_info['name']
    except:
        cat_name = f"Class {class_id}"

    frac = class_fraction[class_id]
    axes[i].set_title(f"{cat_name} ({frac:.1%})")
    axes[i].axis("off")

# Turn off unused plots
for i in range(len(selected_class_ids), 6):
    axes[i].axis("off")

plt.tight_layout()
plt.show()
```



```
y_probas = np.stack([model(x , training = True) for sample in
range(300)])
result = y_probas.mean(axis =0)
```

Here , I Used Monte Carlo's Dropout method for predictions
Monte Carlo (MC) Dropout can be beneficial in CNNs even if dropout layers were not originally included during training. By introducing dropout layers during inference (or modifying the trained model to include them), MC Dropout allows the network to simulate a Bayesian approach, producing a distribution of predictions through multiple stochastic forward passes. This helps in quantifying predictive uncertainty, which is crucial in high-stakes tasks like medical imaging or autonomous driving. Even without dropout during training, adding it at inference can approximate model uncertainty, although its effectiveness is generally higher when dropout is present during both training and testing.