UNIT- 5

TensorFlow is an open-source machine-learning framework developed by Google. It is written in Python, making it accessible and easy to understand. It is designed to build and train machine learning (ML) and deep learning models.

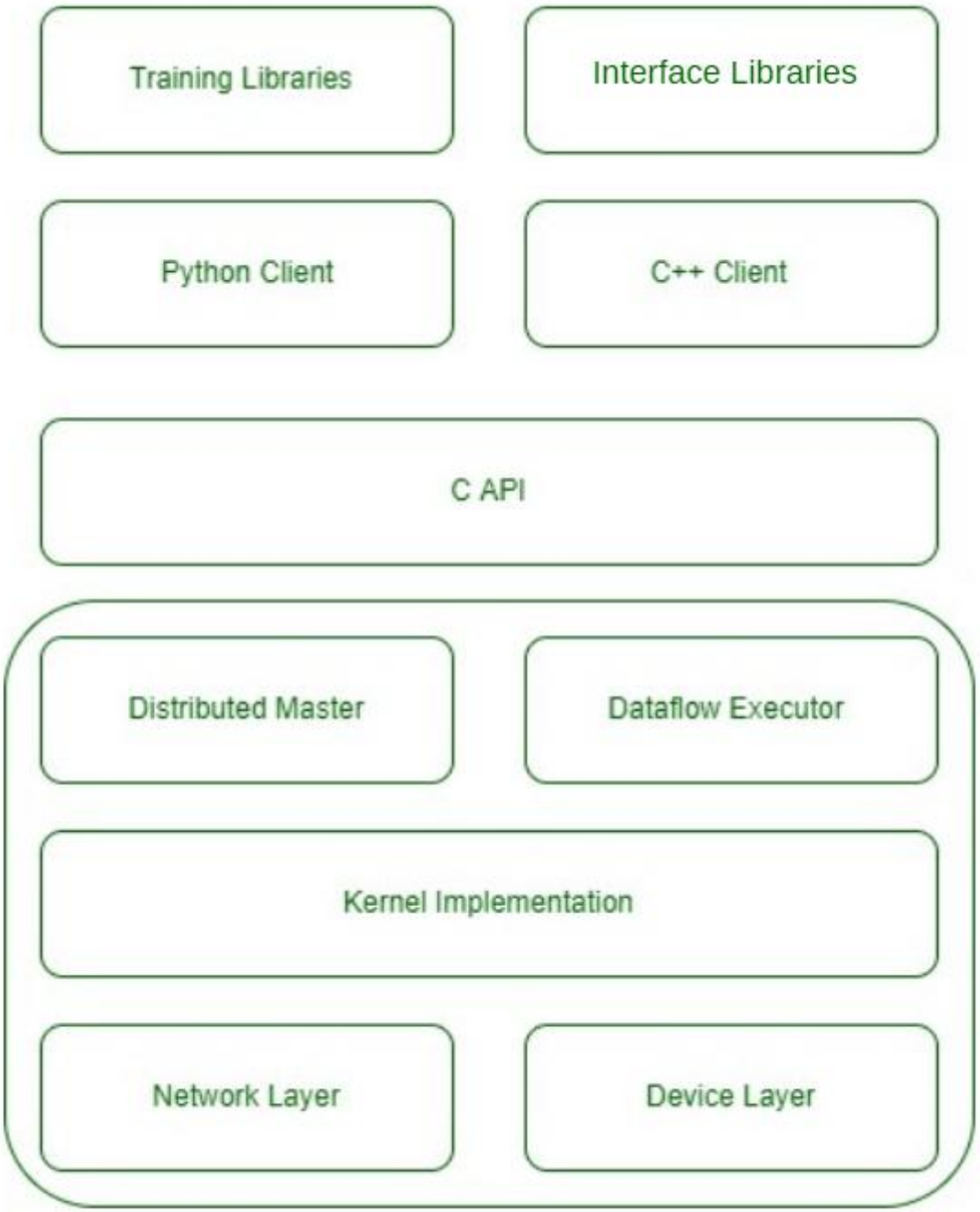
- It is highly scalable for both research and production.
- It supports CPUs, GPUs, and TPUs for faster computation.
- TensorFlow provides built-in tools for visualization and debugging.
- It works seamlessly with other AI frameworks and libraries.

Programming model and Basic concepts:

Each computation in TensorFlow describes a directed graph that's composed of nodes and edges where nodes are operations/ functions and edges are input and output overflows and those functions.

- Inputs/ Outputs in TensorFlow are called Tensor. Tensor is nothing but a multi-dimensional array for which the underpinning element type is specified at the graph construction time.
- The client program that uses TensorFlow generates data- flow graphs using endorsed programming languages (C or Python).
- An operation is a function that has a name and represents an abstract computation. An operation can have attributes that should be generated and handed at the time of graph construction.
- One frequent application of attributes is to fabricate operation polymorphic.
- The kernel is the implementation of an operation on a specific device.
- The client program interacts with TensorFlow system inflow by creating sessions. The session interface has extended styles to generate a computation graph and its reinforcement run() method which computes output of individual node in prosecution graph by providing needed inputs.
- In utmost of the machine learning tasks computation graphs implemented multifold times and utmost of the ordinary tensors don't survive after unattached accomplishment that's why TensorFlow has variables.
- Variable is a special kind of operation that returns a handle to a variable Tensor which can survive during multiple prosecutions of the graph.
- Trainable Parameters like weights, biases are reposited in Tensors held in variables.

High-level Architecture



Architecture of TensorFlow

- The first layer of TensorFlow consists of the device layer and the network layer. The device layer contains the implementation to communicate to the various devices like GPU, CPU, TPU in the operating system where TensorFlow will run. Whereas the network layer has implementations to communicate with different machines using different networking protocols in the Distributable Trainable setting.
- The second layer of TensorFlow contains kernel implementations for applications mostly used in machine learning.

- The third layer of TensorFlow consists of distributed master and dataflow executors. Distributed Master has the ability to distribute workloads to different devices on the system. Whereas data flow executor performs the data flow graph optimally.
- The next layer exposes all the functionalities in the form of API which is implemented in C language. C language is chosen because it is fast, reliable, and can run on any operating system.
- The fifth layer provides support for Python and C++ clients.
- And the last layer of TensorFlow contains training and inference libraries implemented in python and C++.

Advanced TensorFlow Features

Advanced TensorFlow features allow developers to create more complex and powerful machine-learning models. Some of these features include:

- **Custom Operations**: TensorFlow allows developers to define their own custom operations, which can be used in the computation graph just like built-in operations. This allows developers to add new functionality or optimize existing functionality for specific use cases.
- **Custom Gradients**: TensorFlow also allows developers to define custom gradients for their custom operations. This can be useful for implementing complex or non-standard backpropagation algorithms.
- **TensorFlow Eager Execution**: TensorFlow Eager Execution is an alternative execution mode that allows developers to run TensorFlow operations immediately, without building a computation graph first. This can make development and debugging easier, and can also be useful for interactive sessions and research.
- **TensorFlow Keras**: TensorFlow Keras is a high-level API that provides a convenient and consistent interface for building and training machine learning models using TensorFlow. It is built on top of TensorFlow and is compatible with both TensorFlow Eager Execution and the standard TensorFlow session-based execution.
- **TensorFlow Probability**: TensorFlow Probability is a library for probabilistic reasoning and statistical analysis in TensorFlow. It provides a wide range of tools for probabilistic modeling, including distributions, bijectors, and more.

- **TensorFlow Model Optimization**: TensorFlow Model Optimization toolkit is a suite of tools for optimizing machine learning models for deployment. The toolkit includes techniques for quantization, pruning, and more.

These features can help developers create more powerful and flexible machine-learning models, and can also make the development process more efficient.

TensorFlow Examples and Use Cases

TensorFlow is a powerful tool for a wide range of machine learning tasks, and it has been used in many real-world applications. Some examples of TensorFlow use cases include:

- **Image Classification**: TensorFlow has been used for image classification tasks, such as identifying objects in images or classifying images into different categories. Convolutional Neural Networks (CNNs) are commonly used in TensorFlow for image classification tasks.
- **Natural Language Processing (NLP**): TensorFlow has been used for a wide range of NLP tasks, such as language translation, text summarization, and sentiment analysis. Recurrent Neural Networks (RNNs) and Transformers are commonly used in TensorFlow for NLP tasks.
- **Time-series Analysis:** TensorFlow can be used to analyze time-series data, such as stock prices, weather data, or sensor data. Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks are commonly used in TensorFlow for time-series analysis.
- **Generative Models**: TensorFlow has been used to train generative models, such as Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs), which can be used for tasks such as image generation, style transfer, and more.
- Reinforcement Learning: TensorFlow can be used to train Reinforcement Learning(RL) agents, which can be used for tasks such as game playing, robotics, and more.
- **Anomaly Detection**: TensorFlow can be used to train models for anomaly detection, identifying patterns or events in data that deviate from normal behavior, such as in network security or manufacturing.

Difference between TensorFlow and Keras

Both Tensorflow and Keras are famous machine learning modules used in the field of data science. In this article, we will look at the advantages, disadvantages and the difference between these libraries.

TensorFlow

TensorFlow is an open-source platform for machine learning and a symbolic math library that is used for machine learning applications.

Advantages of TensorFlow:

Tensor flow has a better graph representation for a given data rather than any other top platform out there.

- Tensor flow has the advantage that it does support and uses many backend software like GUI and ASIC.
- When it comes to community support tensor flow has the best.
- Tensor flow also helps in debugging the sub-part of the graphs.
- Tensor flow has shown a better performance when compared with other platforms.
- Easy to extend as it gives freedom to add custom blocks to build on new ideas.

Disadvantages of TensorFlow:

- Tensor flow not specifically designed for the Windows operating systems but it is designed for other OS like Linux but tensor flow can be installed in windows with the help of a python package installer(pip).
- The speed of the tensor flow is less when it is compared to other platforms of the same type.
- For a better understanding of tensor flow, the user must have the fundamentals of calculus.
- Tensor flow does not support OpenCL.

Keras

It is an Open Source Neural Network library that runs on top of Theano or Tensorflow. It is designed to be fast and easy for the user to use. It is a useful library to construct any deep learning algorithm of whatever choice we want.

Advantages of Keras:

- Keras is the best platform out there to work on neural network models.
- The API that Keras has a user-friendly where a beginner can easily understand.
- Keras has the advantage that it can choose any libraries which support it for its backend support.
- Keras provides various pre-trained models which help the user in further improving the models the user is designing.
- When it comes to community support Keras has the best like stack overflow.

Disadvantages of Keras:

- The major drawback of Keras is it is a low-level application programming interface.

- Few of the pre-trained models that the Keras has been not much supportive when it comes to designing of some models.
- The errors given by the Keras library were not much helpful for the user.

Difference between TensorFlow and Keras:

S.No	TensorFlow	Keras
1.	Tensorhigh-performanceFlow is written in C++, CUDA, Python.	Keras is written in Python.
2.	TensorFlow is used for large datasets and high performance models.	Keras is usually used for small datasets.
3.	TensorFlow is a framework that offers both high and low-level APIs.	Keras is a high-Level API.
4.	TensorFlow is used for high-performance models.	Keras is used for low-performance models.
5.	In TensorFlow performing debugging leads to complexities.	In Keras framework, there is only minimal requirement for debugging the simple networks.
6.	TensorFlow has a complex architecture and not easy to use.	Keras has a simple architecture and easy to use.
7.	TensorFlow was developed by the Google Brain team.	Keras was developed by François Chollet while he was working on the part of the research effort of project ONEIROS.

Keras

Keras high-level neural networks APIs that provide easy and efficient design and training of deep learning models. It is built on top of powerful frameworks like TensorFlow, making it both highly flexible and accessible. Keras has a simple and user-friendly interface, making it ideal for both beginners and experts in deep learning.

DEEP LEARNING FRAMEWORKS

Deep learning frameworks are the backbone of AI development, offering pre-built modules, optimization libraries and deployment tools that make building complex neural networks much faster and easier. Choosing the right Deep Learning Frameworks can help us make quick prototype and a scalable production system.

1. PyTorch

PyTorch is one of the most widely used deep learning frameworks today known for its flexibility, dynamic computation graph and large community support. Initially developed by Facebook's AI Research (FAIR) lab, it has become the go-to choice for both researchers and industry leaders working on computer vision, NLP and large language models.

- Open-source, Python-first framework with C++ backend.
- Hugely popular in research and prototyping.
- Supported by tech giants like Meta, Tesla and Microsoft.
- Integrates well with Hugging Face for LLMs.

Working:

- Uses dynamic computation graphs (define-by-run).
- Tensors enable GPU acceleration.
- Autograd module performs automatic differentiation.
- Supports distributed training across GPUs and nodes.

Applications:

- Training and fine-tuning LLMs like GPT, BERT, etc.
- Computer vision tasks (object detection, classification).
- Speech recognition and NLP assistants.
- Reinforcement learning and robotics.

2. TensorFlow

TensorFlow developed by Google Brain is a highly scalable and production-ready

framework. With TensorFlow 2.x, Keras is now the default high-level API making it easier for developers to build, train and deploy deep learning models for real-world applications at scale.

- Open-source with strong enterprise adoption.
- Supports TensorFlow Lite (mobile/edge) and TensorFlow Serving (deployment).
- Backed by Google Cloud AI ecosystem.
- Focused on production pipelines and scalability.

Working:

- Works with static computational graphs.
- Keras API provides abstraction for easy prototyping.
- Supports GPUs, TPUs and distributed training.
- TensorFlow Extended (TFX) for full ML pipelines.

Applications:

- Mobile and edge AI (TensorFlow Lite).
- Large-scale production ML pipelines.
- Healthcare diagnostics using image analysis.
- Fraud detection and financial forecasting.

3. Keras

Keras is an open-source deep learning library that provides a user-friendly, high-level API for building and training neural networks. It is widely adopted in academia and industry due to its simplicity and flexibility and it runs on top of backends like TensorFlow.

- Incubated as a project by François Chollet in 2015.
- Designed for ease of use, rapid prototyping and modularity.
- Supports multiple backends (primarily TensorFlow, but earlier also Theano and CNTK).
- Provides simple APIs for training, evaluation and deployment.

Working:

- Offers a modular architecture with building blocks such as layers, models, optimizers and loss functions.
- Supports both sequential and functional APIs for model building.
- Handles GPU acceleration through TensorFlow backend.
- Includes pre-trained models and transfer learning support for quick experimentation.

Applications:

- Computer vision (image classification, object detection, image segmentation).

- Natural language processing (sentiment analysis, text classification, sequence modeling).
- Healthcare (disease detection from medical images, drug discovery).
- Finance (fraud detection, algorithmic trading).
- Academic research and AI prototyping.

4. JAX

JAX is Google's high-performance deep learning and scientific computing framework that combines NumPy-like syntax with automatic differentiation and hardware acceleration. Together with Flax and Haiku libraries, it's increasingly popular in AI research and large-scale model training.

- NumPy-compatible API for researchers.
- Strong support for TPU/GPU acceleration.
- Focus on speed and efficiency.
- Growing use in cutting-edge AI labs.

Working:

- Uses just-in-time (JIT) compilation for speed.
- Vectorization (vmap) and parallelization (pmap) simplify scaling.
- Autograd for automatic differentiation.
- Functional programming style for reproducibility.

Applications:

- Training massive models (vision transformers, LLMs).
- Scientific simulations and optimization problems.
- Reinforcement learning research.
- Generative AI experiments (diffusion models, LLM pretraining).

5. Hugging Face Transformers

The Hugging Face Transformers library has become the cornerstone for working with pre-trained models in natural language processing and beyond. It provides thousands of models for text, vision and audio, along with tools for fine-tuning, deployment and integration.

- Focused on transformer architectures (BERT, GPT, LLaMA, etc.).

- Large model hub with community and enterprise support.
- Easy APIs for inference, fine-tuning and deployment.
- Works with PyTorch, TensorFlow and JAX backends.

Working:

- Provides pre-trained models ready for fine-tuning.
- Tokenizers handle text preprocessing.
- Model hub allows easy sharing and versioning.
- Integrates with Accelerate for distributed training.

Applications:

- Chatbots and conversational AI.
- Sentiment analysis, summarization, translation.
- Multimodal AI (text + image).
- Fine-tuning large foundation models for enterprise use.

6. DeepSpeed

DeepSpeed developed by Microsoft, is a deep learning optimization library that makes training and inference of very large models (billions of parameters) efficient and cost-effective. It's widely used for scaling large language models in cloud environments.

- Specializes in model parallelism and memory optimization.
- Powers models like GPT-NeoX and BLOOM.
- Reduces hardware costs with 3D parallelism.
- Strong integration with PyTorch.

Working:

- Zero Redundancy Optimizer (ZeRO) reduces memory use.
- Pipeline and tensor parallelism for large models.
- Offloading to CPU/NVMe to save GPU memory.
- Optimized inference engine for deployment.

Applications:

- Training trillion-parameter language models.
- Serving efficient inference at scale.
- Enterprise cloud AI deployments.
- Research on scaling AI beyond single GPU limits.

7. OpenVINO

OpenVINO (Open Visual Inference and Neural Network Optimization) is Intel's toolkit designed to optimize and deploy deep learning models for high-performance inference across edge devices, CPUs, GPUs and VPUs.

- Optimized for Intel hardware acceleration.

- Focus on inference rather than training.
- Supports model compression and quantization.
- Works with models from TensorFlow, PyTorch, ONNX.

Working:

- Converts trained models into optimized IR (Intermediate Representation).
- Applies quantization, pruning and layer fusion.
- Deploys across CPUs, integrated GPUs and FPGAs.
- Provides APIs for computer vision and edge applications.

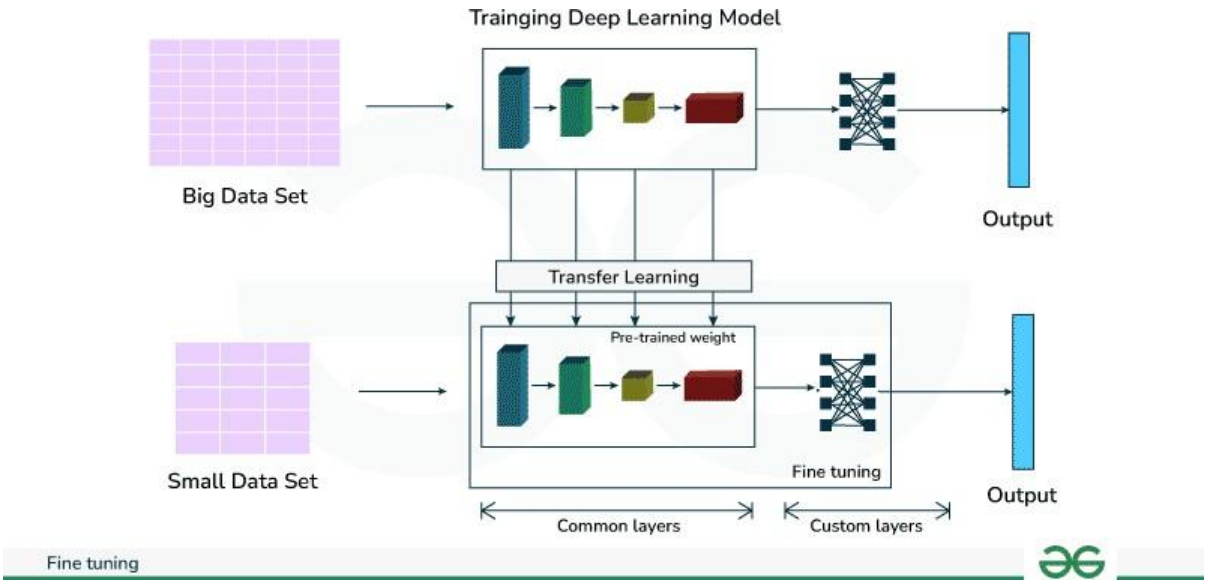
Applications:

- Real-time video analytics.
- AI at the edge (smart cameras, IoT).
- Industrial automation and robotics.
- Healthcare imaging solutions.

TUNNING THE LAYERS

Tuning layers in deep learning primarily refers to fine-tuning, a transfer learning technique where you adapt a pre-trained model to a new task by selectively updating its weights. This involves freezing the early layers that capture general features and unfreezing or adding new, task-specific layers at the end to learn specialized features for your problem. You also configure and tune hyperparameters like learning rate and batch size, which control how the layers learn.

Fine-tuning allows a pre-trained model to adapt to a new task. This approach uses the knowledge gained from training a model on a large dataset and applying it to a smaller, domain specific dataset. Fine-tuning involves adjusting the weights of the model's layers or updating certain parts of the model to improve its performance on the new task.



Fine-tuning is used in **transfer learning** where a is model trained on one similar task and is reused for another task often with minimal changes. The underlying assumption is that the model has already learned useful features in the original task that can be transferred and adapted to the new task hence reducing the need for training a model from scratch.

How Fine-Tuning Works?

Fine-tuning typically involves the following steps:

1. **Select a Pre-Trained Model**: The first step is to choose a pre-trained model that has been trained on a large, diverse dataset. Popular pre-trained models includes **BERT** for NLP tasks**, VGG** and **ResNet** for image classification tasks and **GPT** models for text generation.
2. **Freeze Initial Layers**: In fine-tuning we usually freeze the weights of the earlier layers in the model. These layers capture basic features like edges or textures in
3. **Update Later Layers**: The later layers of the model are the ones that specialize in more specific features. These layers are typically fine-tuned to adjust the weights based on the new task. For example if we're fine-tuning a language model for sentiment analysis then the final layers will adjust to classify the sentiment of the input text.
4. **Use a Smaller Learning Rate**: A key principle in fine-tuning is using a smaller learning rate compared to training from scratch. This ensures that the pre-trained weights are not drastically changed but are adjusted just enough to perform well on

the new task.

5. **Evaluate and Refine**: After training it's important to evaluate the performance of the fine-tuned model on the specific task. Based on the evaluation we may continue to adjust the learning rate, fine-tune more layers or apply other modifications until the model performs optimally.

Examples of Fine-Tuning

- **Image Classification**: A common use case for fine-tuning is in computer vision. A model like **ResNet** might be pre-trained on a large dataset like ImageNet. When we need to classify medical images we can fine-tune the model to focus on detecting relevant medical features such as tumors without retraining the model from scratch.
- **Natural Language Processing**: In NLP fine-tuning is done on models like **BERT** or **GPT**. For example if a model is trained on general text and needs to be used for a specific task like question-answering or sentiment analysis, fine-tuning helps adjust the model's knowledge to suit that particular application.

Advantages of Fine-Tuning

Fine-tuning is beneficial when:

- **We have a limited dataset**: Fine-tuning a pre-trained model helps achieve high performance even with small amounts of data.
- **We need domain-specific expertise**: Adapting a pre-trained model on specialized datasets (like legal or medical text) improves accuracy for specific fields. For example a model trained on general images can be fine-tuned for medical image analysis hence improving accuracy by focusing on relevant features.
- **We want to save training time:** Fine-tuning avoids training from scratch, making it a faster solution for time-sensitive projects.
- **Better Performance on Smaller Datasets**: Training models on smaller datasets can often result in overfitting as the model has too few examples to learn from. Fine-tuning allows the model to generalize better by using the knowledge from the larger pre-trained dataset.

Challenges of Fine-Tuning

- **Overfitting**: Even though fine-tuning reduces the risk of overfitting compared to training from scratch it can still occur if the new dataset is too small or lacks diversity.
- **Computational Constraints**: Fine-tuning may still require considerable computational resources especially when working with large models like GPT or BERT.
- **Selecting the Right Layers to Fine-Tune**: Deciding which layers to freeze and

which to train can be tricky. Some models may benefit from fine-tuning more layers while others might only require adjustments to the final layers.

DEEP LEARNING WORKFLOW

The deep learning workflow has seven primary components:

1. Acquiring data
2. Preprocessing
3. Splitting and balancing the dataset
4. Building and training the model
5. Evaluation
6. Hyperparameter tuning
7. Deploying our solution (For Industry)

Part 1: Acquiring Data

Publicly Available Datasets

The best source of data is often publicly available datasets. Sites like Kaggle host thousands of large, labeled data sources. Working with these curated datasets helps reduce the overhead of starting a deep learning project.

Existing Databases

In some cases, our organization may have a large dataset on hand. Often, these datasets are stored in a Relational Database Management System (RDMS). In this case, we can build our specific dataset using SQL queries.

Web scraping/APIs

Online news, social media posts, and search results represent rich streams of data, which we can leverage for our deep learning projects. do this via Web scraping: the extraction of data from websites. While scraping and collecting data, should keep in mind ethical considerations, including privacy and consent issues. There are many tools to web scrape in Python, including BeautifulSoup. Many sites, like Reddit and Twitter, have Python Application Programming Interfaces (APIs). use APIs to gather data from different applications.

Part 2: Preprocessing

Once built the dataset, need to preprocess it into useful features for our deep

learning models. At a high-level, three primary goals when preprocessing data for neural networks: 1) clean the data, 2) handle categorical features and text and 3) scale our real-valued features using normalization or standardization techniques. Preprocessing is also a fantastic opportunity to become more familiar with our data.

Cleaning data

datasets contain noisy examples, extra features, missing data and outliers. It is good practice to test for and remove outliers, remove unnecessary features, fill-in missing data, and filter out noisy examples.

Scaling features

initialize neural networks with small weights to stabilize training, the models will struggle when faced with input features that have large values. As a result, often scale real-valued features in two ways: normalize features so that they are between 0 and 1, and standardize them so they have a mean of zero and a variance of one.

Handling categorical data and text

Neural networks expect numbers as their inputs. This means need to convert all categorical data and text to real-valued numbers. - handle categorical variables by assigning each option its unique integer or converting them to one-hot encodings. - When working with strings of raw text, need to handle a few extra processing steps before encoding our words as integers. These steps include tokenizing our data (splitting our text into individual words/tokens), and padding our data (adding padding tokens to make all of our examples the same length).

Part 3: Splitting and Balancing the Dataset

split the data into two datasets: training and validation. Train the model on the training dataset and we evaluate it on the validation dataset. When splitting the dataset, there are two major considerations: the size of splits, and whether it will stratify the data. After split need to address imbalances in the training set. Scikit-learn provides the train_test_split function, which splits data into training and validation datasets and specifies the size of our validation data.

Handling Imbalanced Data

Imbalanced data, where some classes appear much more than others, pose a

challenge for deep learning models. If train neural networks on imbalanced data, resulting model will be heavily biased towards predicting those majority classes. This is especially problematic because usually we care much more about identifying instances of the minority classes (like rare cases of disease or credit fraud).

There are two main approaches to dealing with imbalanced training data undersampling and oversampling. These two approaches should be taken-up with utmost caution, and it's best to have a domain expert on hand to weigh in.

In undersampling, we balance data by throwing out examples from majority class.
In oversampling, duplicate instances of minority class so that they occur more often.
A popular alternative to traditional oversampling is called Synthetic Minority Oversampling TEchnique (SMOTE). The SMOTE algorithm creates synthetic examples that are similar to those in our minority class, and adds them to our dataset.

Part 4: Building and Training the Model

Once splitted the dataset, it's time to choose loss function, and layers.

For each layer, need to select a reasonable number of hidden units. There is no absolute science to choosing the right size for each layer, nor the number of layers — it all depends on your specific data and architecture.

It's good practice to start with a few layers (2-6).
Usually, create each layer with between 32 and 512 hidden units.
tend to decrease the size of hidden layers as we move upwards - through the model.
SGD and Adam optimizers first.
When setting an initial learning rate, a common practice is to default to 0.01.

Part 5: Evaluating Performance

Each time train the model, evaluate its performance on validation set. provide a validation set at training time, Keras handles this automatically. performance on the validation set gives us a sense for how our model will perform on new, unseen data.

When considering performance, it's important to choose the correct metric. If data set is heavily imbalanced, accuracy (and even AUC) will be less meaningful. In this case consider metrics like precision and recall. F1-score is another useful metric that

combines both precision and recall. A confusion matrix can help visualize what data-points are misclassified and what aren't.

Part 6: Tuning Hyperparameters

When training and evaluating our model, explore different learning rates, batch sizes, architectures, and regularization techniques. tune parameters, should watch the loss and metrics, and be on the lookout for clues as to why model is struggling.:

Unstable learning means that likely need to reduce the learning rate and or/increase batch size.

A disparity between performance on the training and evaluation sets means are overfitting, and should reduce the size of our model, or add regularization (like dropout).

Poor performance on both the training and the test set means that are underfitting, and may need a larger model or a different learning rate.

A common practice is to start with a smaller model and scale up hyperparameters until see training and validation performance diverge, which means have overfit to odata.

Critically, because neural network weights are randomly initialized scores will fluctuate, regardless of hyperparameters. One way to make accurate judgments is to run the same hyperparameter configuration multiple times, with different random seeds.

Confusion Matrix

Confusion matrix is a simple table used to measure how well a classification model is performing. It compares the predictions made by the model with the actual results and shows where the model was right or wrong. This helps you understand where the model is making mistakes so you can improve it. It breaks down the predictions into four categories:

- **True Positive (TP):** The model correctly predicted a positive outcome i.e the actual outcome was positive.
- **True Negative (TN):** The model correctly predicted a negative outcome i.e the actual outcome was negative.
- **False Positive (FP):** The model incorrectly predicted a positive outcome i.e the actual outcome was negative. It is also known as a Type I error.

- **False Negative (FN):** The model incorrectly predicted a negative outcome i.e the actual outcome was positive. It is also known as a Type II error.

Confusion Matrix

It also helps calculate key measures like **accuracy**, **precision** and **recall** which give a better idea of performance especially when the data is imbalanced.

Metrics based on Confusion Matrix Data

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

1. Accuracy

Accuracy shows how many predictions the model got right out of all the predictions. It gives idea of overall performance but it can be misleading when one class is more dominant over the other. For example a model that predicts the majority class correctly most of the time might have high accuracy but still fail to capture important details about other classes. It can be calculated using the below formula:

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

2. Precision

Precision focus on the quality of the model's positive predictions. It tells us how many of the "positive" predictions were actually correct. It is important in situations where false positives need to be minimized such as detecting spam emails or fraud. The formula of precision is:

$$\text{Precision} = \frac{TP}{TP+FP}$$

3. Recall

Recall measures how how good the model is at predicting positives. It shows the proportion of true positives detected out of all the actual positive instances. High recall is essential when missing positive cases has significant consequences like in medical tests.

$$\text{Recall} = \frac{TP}{TP+FN}$$

4. F1-Score

F1-score combines precision and recall into a single metric to balance their trade-off. It provides a better sense of a model's overall performance particularly for imbalanced datasets. It is helpful when both false positives and false negatives are important though it assumes precision and recall are equally important but in some situations one might matter more than the other.

$$\text{F1-Score} = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

5. **Specificity**

Specificity is another important metric in the evaluation of classification models particularly in binary classification. It measures the ability of a model to correctly identify negative instances. Specificity is also known as the True Negative Rate Formula is given by:

$$\text{Specificity} = \frac{TN}{TN+FP}$$

6. Type 1 and Type 2 error

Type 1 and Type 2 error are:

- **Type 1 error**: It occurs when the model incorrectly predicts a positive instance but the actual instance is negative. This is also known as a **false positive**. Type 1 Errors affect the **precision** of a model which measures the accuracy of positive predictions.

$$\text{Type 1 Error} = \frac{\text{FP}}{\text{FP+TN}}$$

- **Type 2 error**: This occurs when the model fails to predict a positive instance even though it is actually positive. This is also known as a **false negative**. Type 2 Errors impact the **recall** of a model which measures how well the model identifies all

actual positive cases.

$$\text{Type 2 Error} = \frac{FN}{TP+FN}$$

Example: A diagnostic test is used to detect a particular disease in patients.

- **Type 1 Error (False Positive):** This occurs when the test predicts a patient has the disease (positive result) but the patient is actually healthy (negative case).
- **Type 2 Error (False Negative):** This occurs when the test predicts the patient is healthy (negative result) but the patient actually has the disease (positive case).

Confusion Matrix For Binary Classification

A 2x2 Confusion matrix is shown below for the image recognition having a Dog image or Not Dog image:

	Predicted	**Predicted**
Actual	True Positive (TP)	False Negative (FN)
Actual	False Positive (FP)	True Negative (TN)

- **True Positive (TP):** It is the total counts having both predicted and actual values are Dog.
- **True Negative (TN):** It is the total counts having both predicted and actual values are Not Dog.
- **False Positive (FP):** It is the total counts having prediction is Dog while actually Not Dog.
- **False Negative (FN):** It is the total counts having prediction is Not Dog while actually, it is Dog.

Example: Confusion Matrix for Dog Image Recognition with Numbers

Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Actual	Dog	Dog	Dog	Not Dog	Dog	Not Dog	Dog	Dog	Not Dog	Not Dog
Predicted	Dog	Not Dog	Dog	Not Dog	Dog	Dog	Dog	Dog	Not Dog	Not Dog

Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Result	TP	FN	TP	TN	TP	FP	TP	TP	TN	TN

- Actual Dog Counts = 6
- Actual Not Dog Counts = 4
- True Positive Counts = 5
- False Positive Counts = 1
- True Negative Counts = 3
- False Negative Counts = 1

		Predicted	
		Dog	**Not Dog**
Actual	**Dog**	True Positive (TP =5)	False Negative (FN =1)
	Not Dog	False Positive (FP=1)	True Negative (TN=3)

The Role of a Confusion Matrix

To better comprehend the confusion matrix, you must understand the aim and why it is widely used.

When it comes to measuring a model's performance or anything in general, people focus on accuracy. However, being heavily reliant on the accuracy metric can lead to incorrect decisions. To understand this, we will go through the limitations of using accuracy as a standalone metric.

Limitations of accuracy as a standalone metric

However, using this metric as a standalone comes with limitations, such as:

- **Working with Imbalanced Data**: No data ever comes perfect, and the use of the accuracy metric should be evaluated on its predictive power. For example, working with

a dataset where one class outweighs another will cause the model to achieve a higher accuracy rate as it will predict the majority class.

- **Error Types**: Understanding and learning about your model's performance in a specific context will aid you in fine-tuning and improving its performance. For example, differentiating between the types of errors through a confusion matrix, such as FP and FN, will allow you to explore the model's limitations.

  Through these limitations, the confusion matrix, along with the variety of metrics, offers more detailed insight on how to improve a model's performance.

  The benefits of a confusion matrix

  As seen in the basic structure of a confusion matrix, the predictions are broken down into four categories: True Positive, True Negative, False Positive, and False Negative.

  This detailed breakdown offers valuable insight and solutions to improve a model's performance:

- **Solving Imbalanced Data**: As explored, using accuracy as a standalone metric has limitations when it comes to imbalanced data. Using other metrics, such as precision and recall, allows a more balanced view and accurate representation. For example, false positives and false negatives can lead to high consequences in sectors such as Finance.
- **Error Type Differentiator**: Understanding the different types of errors produced by the machine learning model provides knowledge of its limitations and areas of improvement.
- **Trade-Offs**: The trade-off between using different metrics in a Confusion Matrix is essential as they impact one another. For example, an increase in precision typically leads to a decrease in recall. This will guide you in improving the performance of the model using knowledge from impacted metric values.

Loss Functions in Deep Learning

A **loss function** is a mathematical way to measure how good or bad a model's predictions are compared to the actual results. It gives a single number that tells us how far off the predictions are. The smaller the number, the better the model is doing. Loss functions are used to train models. Loss functions are important because they:

1. **Guide Model Training:** During training, algorithms such as Gradient Descent use the loss function to adjust the model's parameters and try to reduce the error and improve the model's predictions.
2. **Measure Performance:** By finding the difference between predicted and actual values and it can be used for evaluating the model's performance.
3. **Affect learning behavior:** Different loss functions can make the model learn in different ways depending on what kind of mistakes they make.

There are many types of loss functions each suited for different tasks. Here are some common methods:

1. Regression Loss Functions

These are used when your model needs to **predict a continuous number** such as predicting the price of a product or age of a person. Popular regression loss functions are:

1. Mean Squared Error (MSE) Loss

Mean Squared Error (MSE) Loss is one of the most widely used loss functions for regression tasks. It calculates the average of the squared differences between the predicted values and the actual values. It is simple to understand and sensitive to outliers because the errors are squared which can affect the loss.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

2. Mean Absolute Error (MAE) Loss

Mean Absolute Error (MAE) Loss is another commonly used loss function for regression. It calculates the average of the absolute differences between the predicted values and the actual values. It is less sensitive to outliers compared to MSE. But it is not differentiable at zero which can cause issues for some optimization algorithms.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} \left| y_i - \widehat{y_i} \right|$$

2. Classification Loss Functions

Classification loss functions are used to evaluate how well a classification model's predictions match the actual class labels. There are different types of classification Loss functions:

1. Binary Cross-Entropy Loss (Log Loss)

Binary Cross-Entropy Loss is also known as Log Loss and is used for binary classification problems. It measures the performance of a classification model whose output is a probability value between 0 and 1.

$$\text{Binary Cross-Entropy} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

where:

- n is the number of data points
- y_i is the actual binary label (0 or 1)
- $\hat{y}_i$ is the predicted probability.

2. Categorical Cross-Entropy Loss

Categorical Cross-Entropy Loss is used for multiclass classification problems. It measures the performance of a classification model whose output is a probability distribution over multiple classes.

2. Categorical Cross-Entropy Loss

Categorical Cross-Entropy Loss is used for multiclass classification problems. It measures the performance of a classification model whose output is a probability distribution over multiple classes.

$$\text{Categorical Cross-Entropy} = -\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij}\log(\hat{y}_{ij})$$

where:

- n is the number of data points
- k is the number of classes,
- y_{ij} is the binary indicator (0 or 1) if class label j is the correct classification for data point i
- $\hat{y}_{ij}$ is the predicted probability for class j.

3. Sparse Categorical Cross-Entropy Loss

Sparse Categorical Cross-Entropy Loss is similar to Categorical Cross-Entropy Loss but is used when the target labels are integers instead of one-hot encoded vectors. It is efficient for large datasets with many classes.

$$\text{Sparse Categorical Cross-Entropy} = -\sum_{i=1}^{n}\log(\hat{y}_{i,y_i})$$

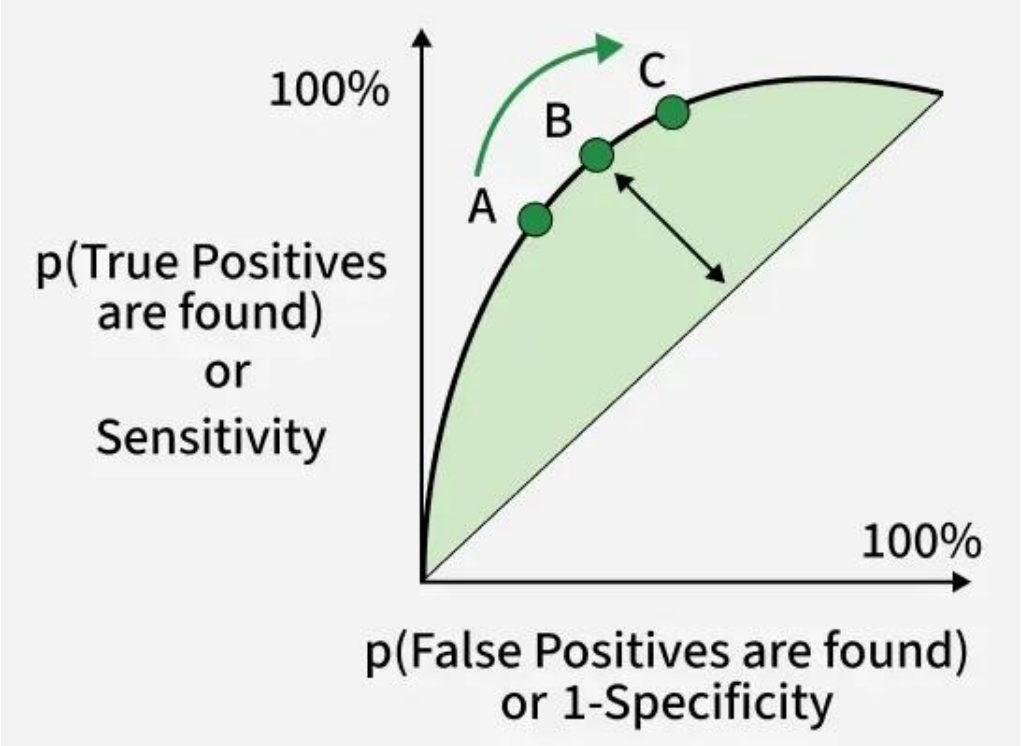
where y_i is the integer representing the correct class for data point i.

AUC ROC Curve

AUC-ROC curve is a graph used to check how well a binary classification model works. It helps us to understand how well the model separates the positive cases like people with a disease from the negative cases like people without the disease at different threshold level. It shows how good the model is at telling the difference between the two classes by plotting:

- **True Positive Rate (TPR):** how often the model correctly predicts the positive cases also known as **Sensitivity or Recall**.
- **False Positive Rate (FPR):** how often the model incorrectly predicts a negative case as positive.
- **Specificity:** measures the proportion of actual negatives that the model correctly identifies. It is calculated as 1 - FPR.

The higher the curve the better the model is at making correct predictions.



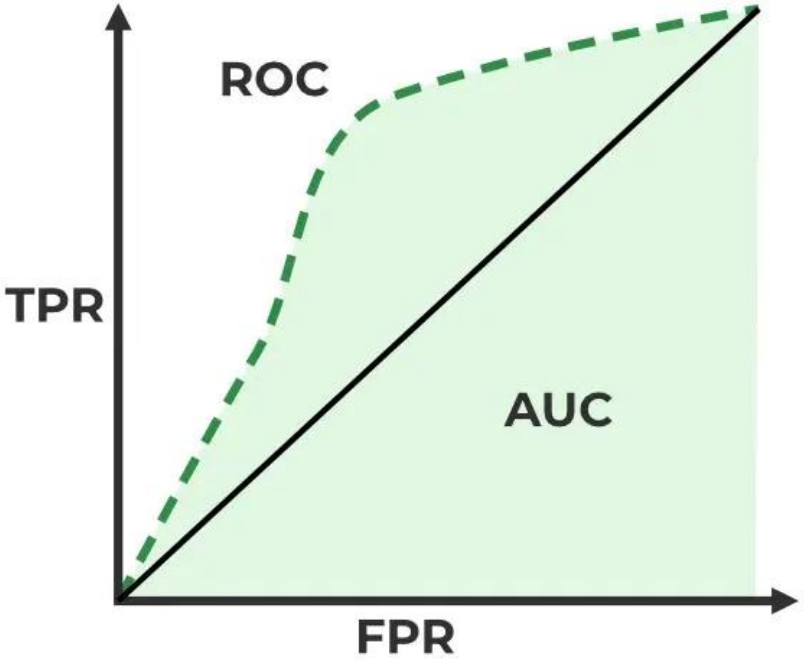
These terms are derived from the **confusion matrix** which provides the following values:

- **True Positive (TP)**: Correctly predicted positive instances
- **True Negative (TN)**: Correctly predicted negative instances
- **False Positive (FP)**: Incorrectly predicted as positive
- **False Negative (FN)**: Incorrectly predicted as negative
- **ROC Curve** : It plots TPR vs. FPR at different thresholds. It represents the trade-off between the sensitivity and specificity of a classifier.
- **AUC(Area Under the Curve)**: measures the area under the ROC curve. A higher AUC value indicates better model performance as it suggests a greater ability to distinguish between classes. An AUC value of 1.0 indicates perfect performance while 0.5 suggests it is random guessing.

How AUC-ROC Works

AUC-ROC curve helps us understand how well a classification model distinguishes between the two classes. Imagine we have 6 data points and out of these:

- **3 belong to the positive class:** Class 1 for people who have a disease.
- **3 belong to the negative class:** Class 0 for people who don't have disease.



Now the model will give each data point a predicted probability of belonging to Class 1. The AUC measures the model's ability to assign higher predicted probabilities to the positive class than to the negative class. Here's how it work:

1. **Randomly choose a pair**: Pick one data point from the positive class (Class 1) and one from the negative class (Class 0).
2. **Check if the positive point has a higher predicted probability**: If the model assigns a higher probability to the positive data point than to the negative one for correct ranking.
3. **Repeat for all pairs**: We do this for all possible pairs of positive and negative examples.

When to Use AUC-ROC

AUC-ROC is effective when:

- The dataset is balanced and the model needs to be evaluated across all thresholds.
- False positives and false negatives are of similar importance.

In cases of highly imbalanced datasets AUC-ROC might give overly optimistic results. In such cases the Precision-Recall Curve is more suitable focusing on the positive class.

Model Performance with AUC-ROC:

- **High AUC (close to 1)**: The model effectively distinguishes between positive and negative instances.
- **Low AUC (close to 0)**: The model struggles to differentiate between the two classes.
- **AUC around 0.5**: The model doesn't learn any meaningful patterns i.e it is doing random guessing.

In short AUC gives you an overall idea of how well your model is doing at sorting positives and negatives, without being affected by the threshold you set for classification. A higher AUC means your model is doing good.

Now the model will give each data point a predicted probability of belonging to Class 1. The AUC measures the model's ability to assign higher predicted probabilities to the positive class than to the negative class. Here's how it work:

1. **Randomly choose a pair**: Pick one data point from the positive class (Class 1) and one from the negative class (Class 0).
2. **Check if the positive point has a higher predicted probability**: If the model assigns a higher probability to the positive data point than to the negative one for correct ranking.
3. **Repeat for all pairs**: We do this for all possible pairs of positive and negative examples.

When to Use AUC-ROC

AUC-ROC is effective when:

- The dataset is balanced and the model needs to be evaluated across all thresholds.
- False positives and false negatives are of similar importance.

In cases of highly imbalanced datasets AUC-ROC might give overly optimistic results. In such cases the Precision-Recall Curve is more suitable focusing on the positive class.

Model Performance with AUC-ROC:

- **High AUC (close to 1)**: The model effectively distinguishes between positive and negative instances.
- **Low AUC (close to 0)**: The model struggles to differentiate between the two classes.
- **AUC around 0.5**: The model doesn't learn any meaningful patterns i.e it is doing random guessing.

In short AUC gives you an overall idea of how well your model is doing at sorting positives and negatives, without being affected by the threshold you set for classification. A higher AUC means your model is doing good.

Simple Deep Learning Implementation with the Iris Dataset

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from tensorflow.keras import layers, models

# 1. Load Iris dataset
iris = load_iris()
X = iris.data         # features (4 features)
y = iris.target       # labels (0, 1, 2)

# 2. Preprocess data
# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale features for better training
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# One-hot encode the labels
encoder = OneHotEncoder(sparse=False)
y_train = encoder.fit_transform(y_train.reshape(-1, 1))
y_test = encoder.transform(y_test.reshape(-1, 1))

# 3. Build the model
model = models.Sequential([
   layers.Dense(10, activation='relu', input_shape=(4,)),  # input layer with 4 features
   layers.Dense(10, activation='relu'),                # hidden layer
   layers.Dense(3, activation='softmax')                # output layer (3 classes)
])

# 4. Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
```

```
    metrics=['accuracy'])

# 5. Train the model
model.fit(X_train, y_train, epochs=50, batch_size=5, validation_split=0.2)

# 6. Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test accuracy: {accuracy:.2f}')
```

Keras Modules

Keras modules contains pre-defined classes, functions and variables which are useful for deep learning algorithm. Let us learn the modules provided by Keras in this chapter.

- **Initializers** – Provides a list of initializers function. We can learn it in details in Keras *layer chapter*. during model creation phase of machine learning.
- **Regularizers** – Provides a list of regularizers function. We can learn it in details in *Keras Layers* chapter.
- **Constraints** – Provides a list of constraints function. We can learn it in details in *Keras Layers* chapter.
- **Activations** – Provides a list of activator function. We can learn it in details in *Keras Layers* chapter.
- **Losses** – Provides a list of loss function. We can learn it in details in *Model Training* chapter.
- **Metrics** – Provides a list of metrics function. We can learn it in details in *Model Training* chapter.
- **Optimizers** – Provides a list of optimizer function. We can learn it in details in *Model Training* chapter.
- **Callback** – Provides a list of callback function. We can use it during the training process to print the intermediate data as well as to stop the training itself (**EarlyStopping** method) based on some condition.
- **Text processing** – Provides functions to convert text into NumPy array suitable for machine learning. We can use it in data preparation phase of machine learning.

- **Image processing** – Provides functions to convert images into NumPy array suitable for machine learning. We can use it in data preparation phase of machine learning.
- **Sequence processing** – Provides functions to generate time based data from the given input data. We can use it in data preparation phase of machine learning.
- **Backend** – Provides function of the backend library like *TensorFlow* and *Theano*.
- **Utilities** – Provides lot of utility function useful in deep learning.

1. Keras Activaton Module

- Provides **built-in activation functions** that introduce **non-linearity** in neural networks.
- Common activation functions:
    - relu (Rectified Linear Unit)
    - sigmoid
    - tanh
    - softmax
- Example:

```
from keras.activations import relu, sigmoid
```

2. Keras Constraints Module

- Provides functions to **impose constraints** on layer weights during training.
- Helps control the model complexity.
- Example:

```
from keras.constraints import max_norm
model.add(Dense(64, kernel_constraint=max_norm(2.)))
```

3. Keras Callbacks Module

- Functions executed **during training** to monitor, control, or log information.
- Common callbacks:

    - EarlyStopping – Stops training when validation performance stops improving
    - ModelCheckpoint – Saves the best model
- Example:

```
from keras.callbacks import EarlyStopping
```

4. Keras Backend Module

- Provides **low-level tensor operations** using backend libraries like **TensorFlow**.
- Useful for operations like matrix multiplication, variable initialization, and placeholder creation.
- Example:

```
from keras import backend as K

a = K.placeholder(shape=(4,2))
b = K.placeholder(shape=(2,3))
c = K.dot(a, b)  # matrix multiplication
```

5. Keras Optimizers Module

- Includes **optimization algorithms** for training neural networks.
- Common optimizers: SGD, Adam, RMSprop, Adagrad.
- Example:

```
from keras.optimizers import Adam
```

6. Keras Utils Module

- Contains **helper functions and utility classes**.
- Examples:
    - to_categorical() – Convert labels to one-hot encoding
    - plot_model() – Visualize the model architecture
    - HDF5Matrix() – Load large datasets efficiently

```
from keras.utils import to_categorical, plot_model
```

7. Keras Initializers Module

- Provides methods to **initialize layer weights**.
- Common initializers: RandomNormal, HeNormal, GlorotUniform.

```
from keras.initializers import RandomNormal
```

8. Keras Losses Module

- Contains **predefined loss functions** used to evaluate model performance.
- Examples: mean_squared_error, binary_crossentropy, categorical_crossentropy.

```
from keras import losses
model.compile(loss=losses.mean_squared_error, optimizer='sgd')
```

9. Keras Regularizers Module

- Adds **penalties** to prevent overfitting.
- Examples: l1, l2, l1_l2.

```
from keras import regularizers
```

10. Keras Models Module

- Provides classes to **build neural network models**:
    - Sequential – Linear stack of layers
    - Model – Functional API for complex architectures

```
from keras.models import Sequential, Model
```

11. Keras Metrics Module

- Provides functions to **evaluate model performance**.
- Examples: accuracy, Precision, Recall, AUC.

```
from keras import metrics
```

12. Keras Text Processing Module

- Functions for **converting text into numerical form** for NLP tasks.
- Example:

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

13. Keras Pre-processing Module

- Utilities for **data preprocessing**, e.g., scaling, normalization, and data augmentation.

14. Keras Layers Module

- Provides building blocks (layers) for neural networks:
  - Dense, Dropout, Flatten
  - Convolutional: Conv1D, Conv2D
  - Recurrent: LSTM, GRU
  - Normalization: BatchNormalization

```
from keras.layers import Dense, Conv2D, LSTM
```

15. Keras Image Processing Module

- Functions to **prepare images** for ML models.
- Example:

```
from keras.preprocessing.image import ImageDataGenerator
```

Summary Table

Module	Purpose
Activators	Activation functions
Constraints	Apply restrictions on weights
Callbacks	Functions during training
Backend	Low-level tensor operations
Optimizers	Optimization algorithms

Module	Purpose
Utils	Helper functions
Initializers	Weight initialization
Losses	Loss functions
Regularizers	Prevent overfitting
Models	Build neural networks
Metrics	Evaluation metrics
Text Processing	Tokenization and text conversion
Pre-processing	Data preparation
Layers	Model building blocks
Image Processing	Image loading and augmentation

Keras Backend?

- Keras is a **high-level deep learning API**.
- The **backend** is the **low-level engine** that actually performs tensor operations.
- Keras can use **TensorFlow**, **Theano**, or **CNTK** as its backend.
- Most modern Keras versions use **TensorFlow** as the default backend.

Purpose of the backend module (keras.backend or K):

- Provides **tensor-level operations** like:
  - Matrix multiplication
  - Reshaping
  - Transposing
  - Creating variables
- Helps make **Keras models portable** across different backend engines.

2. Importing Keras Backend

```
from keras import backend as K
```

- K is now an alias for backend functions.

3. Commonly Used Keras Backend Functions

Function	**Description**	**Example**
K.variable()	Create a backend variable	v = K.variable([[1,2],[3,4]])
K.constant()	Create a constant tensor	c = K.constant(5)
K.zeros()	Create tensor of zeros	z = K.zeros((2,3))
K.ones()	Create tensor of ones	o = K.ones((3,3))
K.zeros_like()	Tensor of zeros with same shape	z = K.zeros_like(v)
K.ones_like()	Tensor of ones with same shape	o = K.ones_like(v)
K.sum()	Sum elements of tensor	s = K.sum(v)
K.mean()	Mean of tensor elements	m = K.mean(v)
K.dot()	Dot product of tensors	d = K.dot(a,b)
K.transpose()	Transpose tensor	t = K.transpose(v)
K.reshape()	Reshape tensor	r = K.reshape(v,(4,1))
K.exp(), K.log(), K.sqrt()	Element-wise operations	K.exp(v)
K.eval()	Evaluate tensor to numpy array	val = K.eval(v)

4. Example: Using Keras Backend

```
from keras import backend as K
import numpy as np

# Create a variable
x = K.variable([[1, 2], [3, 4]])

# Perform operations
y = K.sum(x)           # Sum of all elements
z = K.dot(x, K.transpose(x))  # Dot product

# Evaluate tensors
print("Sum:", K.eval(y))
print("Dot product:\n", K.eval(z))
```

Output:

```
Sum: 10
```

Dot product:

[[5 11]

[11 25]]

APPLICATIONS OF KERAS

1. Image and Video Processing: Keras facilitates tasks like image classification, object detection and video analysis through convolutional neural networks (CNNs). This makes it ideal for applications from medical imaging diagnostics to automated manufacturing quality control.

2. Natural Language Processing (NLP): In NLP, Keras aids in building models for sentiment analysis and machine translation. Its support for sequential data processing is essential for systems capable of summarizing texts.

3. Time Series Forecasting: Keras models equipped with LSTM or GRU layers are perfect for predicting time series data, which is used in fields like finance for stock price predictions or meteorology for weather forecasting.

4. Autonomous Systems: Keras helps process real-time data from sensors in robotics and autonomous vehicles, It eases complex decision-making processes necessary for task performance without human input.

5. Game Development and Reinforcement Learning: Keras can be used in AI development for video games and simulations, employing reinforcement learning to create adaptable and engaging gameplay experiences.

Keras provides a streamlined and efficient interface for deep learning, making it easier to build and train neural networks. It supports efficient execution on both CPU and GPU.