COURSE 3

Deep Learning with Keras & Tensorflow

INDEX

Module 1
Advanced Keras Functionalities

M1 - Section 1
Custom Layers

📌 Creating Custom Layers in Keras

Custom layers provide a mechanism to define operations and behaviors in a neural network that go beyond what is offered by Keras's built-in layers.

They enable the implementation of **task-specific logic**, **novel research ideas**, and **optimized computational flows**, making them indispensable for both experimentation and production-grade modeling.

🔹 Why Use Custom Layers?

Although standard layers such as Dense, Conv2D, or LSTM serve a wide range of applications, there are situations where default implementations are not enough.

Custom layers are essential in the following scenarios:

- **Implementing novel research**: When designing new algorithms or techniques that are not yet part of the Keras API.
- **Task-specific behavior**: When an application requires a specific mathematical operation or architectural element that isn't supported natively.
- **Optimizing for data or constraints**: Tailoring layers to better handle specific types of data distributions, memory constraints, or latency-sensitive applications.
- **Encapsulation of logic**: Wrapping complex or repeated logic into a layer promotes code readability, modularity, and reuse across different models.

By creating custom layers, it becomes possible to extend Keras and build models that are **more fine-tuned**, **highly adaptable**, and **research-ready**.

🔹 How to Create a Custom Layer in Keras

A custom layer in Keras is implemented by subclassing *tensorflow.keras.layers.Layer* and overriding three main methods:

1. __init__(self, ...)

- Initializes the layer's attributes and configuration.
- Typically used to define constants or input parameters (e.g., number of units, activation function type).

2. build(self, input_shape)

- Creates and initializes the layer's trainable weights.
- Called automatically the first time the layer is used (during the first forward pass).
- Should define self.add_weight(...) for custom weight variables.

3. call(self, inputs)

- Defines the forward computation logic.
- Implements the actual transformation applied to the inputs using TensorFlow operations.
- Can include activation functions, mathematical operations, or sub-layer calls.

```
from tensorflow.keras.layers
import Layer
class MyCustomLayer(Layer):
       def __init__(self, units=32, **kwargs):
        super(MyCustomLayer, self).__init__(**kwargs)
        self.units = units
       def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], self.units),
                                 initializer='random_normal',
       trainable=True)
        self.b = self.add_weight(shape=(self.units,),
     initializer='zeros',
     trainable=True)
     def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b
import tensorflow as tf

print(tf.executing_eagerly())  # Returns True in TensorFlow 2.x
# Simple tensor operation
a = tf.constant([1, 2, 3])
b = tf.constant([4, 5, 6])
result = tf.add(a, b)
print(result)  # Outputs: [5 7 9]
```

🔗 Using a Custom Layer in a Model

Once defined, a custom layer integrates seamlessly into a Keras model. For example, in a simple *Sequential* model, the custom dense layer can be used like any standard layer:

```
from tensorflow.keras.models import Sequential
model = Sequential([
    CustomDenseLayer(64),
    CustomDenseLayer(10)
])
model.compile(optimizer='adam', loss='categorical_crossentropy')
```

This ease of integration makes it practical to test and prototype novel components without modifying the broader model logic.

☑Takeaways

☑ Custom layers expand the expressive power of Keras beyond built-in layers.

☑ Useful for tailoring models to domain-specific needs or implementing operations based on new research.

☑ The three essential methods to override *are* ***__init__***, ***build***, and ***call***, which control initialization, weight setup, and forward logic respectively.

☑ Once implemented, a custom layer behaves like any other layer in the Keras ecosystem and can be added to both Sequential and Functional models.

☑ Experimenting with custom layers deepens understanding of neural network internals and supports innovation in model design.

📌Overview of TensorFlow 2.x

TensorFlow is an open-source machine learning platform developed by **Google**.

It has grown into one of the most widely used frameworks for building and deploying both **machine learning (ML)** and **deep learning** models.

Its popularity stems from its ability to support the full model lifecycle — from **training and prototyping**, all the way to **deployment on servers, mobile phones, embedded devices, and web environments**.

🔑Key Features of TensorFlow 2.x

TensorFlow 2.x introduces several important features that make it a more powerful, intuitive, and user-friendly framework. These features include:

Its popularity stems from its ability to support the full model lifecycle — from **training and prototyping**, all the way to **deployment on servers, mobile phones, embedded devices, and web environments**.

⚙️Eager Execution

With eager execution, **operations are executed immediately as they are called**, rather than being added to a static computational graph to be executed later.

It fundamentally changes how operations are evaluated and makes the framework far more accessible and flexible for development and debugging.

Key Benefits:

☑ **Improved Debugging**
Since each operation is executed immediately, developers receive **real-time feedback**. This allows easier identification of logic errors and makes it possible to **inspect intermediate values** during development.

☑ **Simplified Code**
The absence of a need to manually build and run static computation graphs means that code becomes **easier to read and write**, and more consistent with standard Python programming.

☑ **Interactive Development**
The immediacy of operation execution makes TensorFlow suitable for **interactive environments**, such as Jupyter notebooks or command-line experiments. This supports **exploratory workflows** common in research and model prototyping.

Eager execution is particularly beneficial for:

- Beginners in machine learning, due to its intuitive programming model.
- Advanced use cases requiring **dynamic model behaviors**, such as models whose architecture may depend on runtime inputs or branching conditions.

⚙ High-Level API

TensorFlow 2.x integrates **Keras** as its official **high-level API**.

This integration marks a significant usability improvement, as Keras simplifies the development of neural networks with a clear and concise interface.

Keras provides a **user-friendly abstraction layer** for building and training deep learning models.

Key Benefits:

☑ **User-Friendly Syntax**
Keras enables defining models in just a few lines of code, making deep learning more accessible without requiring low-level control logic.

☑ **Modular and Composable Design**
Keras is built to be **modular**, allowing developers to assemble models by combining layers, loss functions, optimizers, and other building blocks with ease.

☑ **Extensive Documentation and Examples**

The integration comes with a large body of well-maintained documentation and **pre-built examples**, which significantly **reduces the learning curve** for newcomers and improves reproducibility in development.

The integration of Keras and TensorFlow 2.X streamlined to the workflow and makes the model building and training more accessible.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Define a simple feedforward neural network
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),
    Dense(10, activation='softmax')
])
# Compile the model
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

⚙ Deployment Versatility

TensorFlow 2.x supports deployment of machine learning models on **a wide variety of hardware and software environments**.

🔹 Platform Flexibility

- Models can be trained on **servers or cloud platforms** using **CPUs, GPUs, or TPUs**.
- The trained models can then be **deployed across different targets**, such as:
    - Mobile phones (Android/iOS)
    - Web browsers
    - Embedded or low-power devices

This cross-platform support enhances the **reusability and portability** of TensorFlow models.

⚙ Deployment Versatility

TensorFlow 2.x enables seamless scaling of training and inference workloads across hardware accelerators:

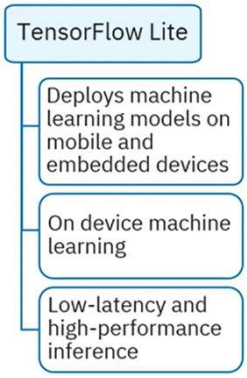
- Supports **CPUs, GPUs, and TPUs**.
- Allows distribution of workloads for **large-scale training** using modern compute infrastructure.
- Optimization strategies are integrated into the TensorFlow backend, allowing models to **leverage hardware acceleration** with minimal configuration.

TensorFlow Ecosystem

TensorFlow 2.x includes a rich set of complementary libraries that expand its core functionality and enable specialized workflows.

These components support tasks such as deployment, monitoring, visualization, and reusability.

TensorFlow Lite (TFLite)

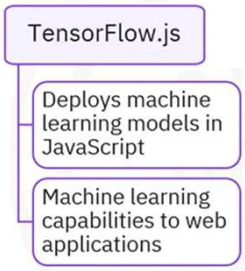


A **lightweight version** of TensorFlow for **mobile and embedded devices**.

Enables **on-device machine learning**, which reduces latency, improves responsiveness, and avoids the need for constant internet connectivity.

Ideal for applications that require **real-time inference**, such as augmented reality, mobile diagnostics, or sensor-based input.

TensorFlow.js (TFJS)

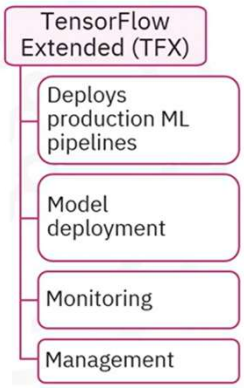


A **JavaScript library** that allows training and deployment of models in **web environments**.

Compatible with both **web browsers** and **Node.js**.

Empowers developers to bring machine learning directly to the **frontend**, making it possible to run models **client-side** without the need for a server backend.

TensorFlow Extended (TFX)



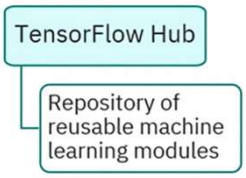
An **end-to-end platform** for deploying **production machine learning pipelines**.

TFX includes tools for:

- Data validation and transformation
- Model serving and deployment
- Monitoring model performance and drift

Ensures **reliability and consistency** of models in production environments where versioning, testing, and reproducibility are crucial.

TensorFlow Hub

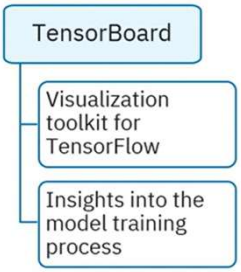


A **repository of reusable machine learning modules**.

Developers can download and plug in **pre-trained components** (like embeddings, vision models, and classifiers) into their own workflows.

Useful for **accelerating development**, improving reproducibility, and reducing training costs when leveraging transfer learning.

TensorBoard



TensorFlow's official **visualization toolkit**.

Provides tools to inspect:

- Training progress
- Model graphs
- Weight distributions
- Scalar metrics (e.g., loss, accuracy)

Essential for understanding how the model learns, and for diagnosing issues during training and evaluation.

☑Takeaways

☑ TensorFlow 2.x is a **powerful and flexible** platform for developing and deploying machine learning models.

☑ **Eager execution** enables immediate operation execution, which improves user experience, simplifies debugging, and supports interactive development.

☑ The **integration of Keras as the high-level API** streamlines model building, making it more intuitive through concise and modular code structures.

☑ TensorFlow 2.x includes a **rich ecosystem** of tools and libraries — such as TensorFlow Lite, TensorFlow.js, TFX, TensorFlow Hub, and TensorBoard — that support the complete machine learning lifecycle from experimentation to production.

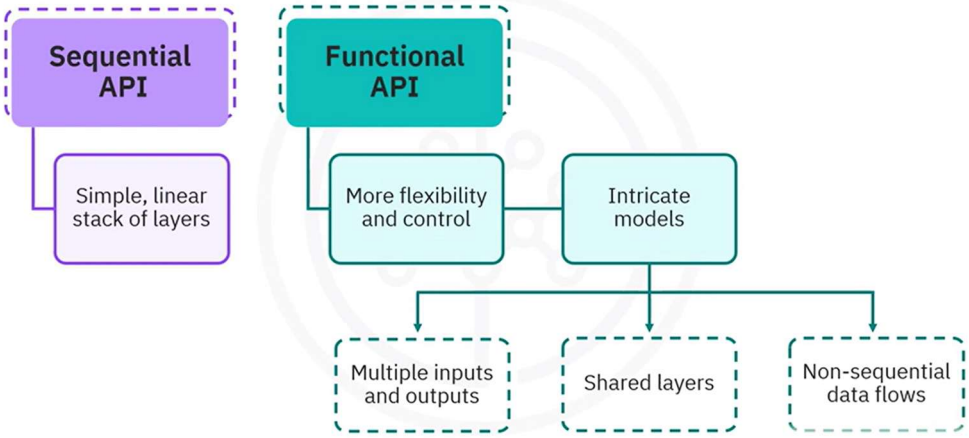
☑ Understanding the **features and capabilities of TensorFlow 2.x** allows for more effective model building and deployment across research, prototyping, and production environments.

M1 - Section 2
Advanced Keras Functional API

📌 Advanced Keras Functional API

The Functional API in Keras provides the architectural flexibility required for building complex deep learning models that go beyond the linear constraints of the Sequential API. While **Sequential API** is ideal for simple, linear layer stacks, the **Functional API** allows for **non-linear, multi-branch, and multi-input/output architectures.**

🔹 Functional API Capabilities



- **Multi-Input & Multi-Output Architectures**: Supports models that can simultaneously process multiple inputs (e.g., different sensor data or feature types) and generate multiple outputs (e.g., multi-task learning or multi-label classification).
- **Shared Layers**: Enables reusing the same layer instance multiple times across a model, which is especially useful in architectures where certain layers share parameters.
- **Non-Sequential Graph Structures**: Permits creation of models with non-linear flows, skip connections, residual links, and parallel branches.
- **Explicit Model Definition**: Inputs, transformations, and outputs are defined as explicit, connectable nodes (tensors), making the model structure transparent and easy to debug.

🔹 Sequential API vs Functional API

🔧 **Sequential API example**:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Create a Sequential model
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),
    Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

Layers are added in a linear stack. The first dense layer is added with relu activation, and the output layer is added with softmax activation for classification.

This approach is suitable for straightforward tasks like single-input feedforward classification. Layers are stacked in order, with no flexibility for branching or multiple outputs.

🔧 **Functional API example**:

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense

# Define the input
inputs = Input(shape=(784,))

# Define layers
x = Dense(64, activation='relu')(inputs)
outputs = Dense(10, activation='softmax')(x)

# Create the model
model = Model(inputs=inputs, outputs=outputs)

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

In the functional API example, you first define an input layer. Connect the first dense layer with relu activation to the input layer. The output layer has the softmax activation. Then combine the input and output layers into a model.

This definition allows for inspecting and manipulating each step explicitly, enabling the extension to more sophisticated configurations (e.g., dual-branch inputs or skip connections).

🔧 **Functional API Example: Multi-Input Model**:

```
from tensorflow.keras.layers import concatenate
# Define two sets of inputs
inputA = Input(shape=(64,))
inputB = Input(shape=(128,))
# The first branch operates on the first input
x = Dense(8, activation='relu')(inputA)
x = Dense(4, activation='relu')(x)
# The second branch operates on the second input
y = Dense(16, activation='relu')(inputB)
y = Dense(4, activation='relu')(y)
# Combine the outputs of the two branches
combined = concatenate([x, y])
# Apply a dense layer and then a regression prediction on the combined
outputs
z = Dense(2, activation='relu')(combined)
z = Dense(1, activation='linear')(z)
# The model will accept the inputs of the two branches and output a single
value
model = Model(inputs=[inputA, inputB], outputs=z)
# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')
```

🔹 Functional API Advantages

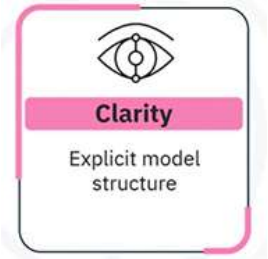
☑ **Flexibility:**

Supports **arbitrary model topologies.** What enables creation of **multi-input/multi-output** models, **skip connections**, **branched logic**, and **shared layers**.

☑ **Clarity:**

Each model component — inputs, intermediate layers, and outputs — is **explicitly defined and connected**, making the architecture more transparent.

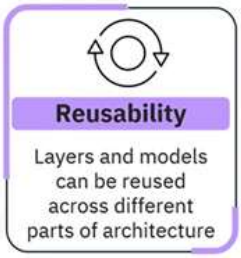
Easier to **visualize**, **understand**, and **debug** than the Sequential API.



☑ **Reusability:**

Layers can be **reused** across different parts of a model without redefining them.

Shared layer instances retain their learned weights across the reused paths.



☑Takeaways

☑ The Functional API is indispensable when building deep learning models that require structural flexibility and fine-grained architectural control.

☑ It allows creation of **multi-input/output**, **layer-sharing**, and **non-linear** models, which are otherwise impossible with Sequential.

☑ Its transparency makes it easier to **debug**, **visualize**, and **adapt** models in research and industrial workflows.

☑ For modern applications involving hybrid data sources or multiple prediction targets, the Functional API is the standard approach.

📌Keras Functional API and Subclassing API

The **Functional API** in Keras allows the creation of flexible and complex neural network architectures beyond the limitations of the Sequential API. It supports non-linear topologies, multiple inputs and outputs, and shared layers.

In parallel, the **Model Subclassing API** provides the highest level of flexibility by allowing users to define custom models by extending the base Keras Model class.

These two approaches are foundational for solving advanced problems in research and development.

🔹Functional API: Capabilities and Architecture

While Sequential is limited to stacking layers linearly, the Functional API allows defining models as **graphs of layers**, where the **connections** between layers are explicitly constructed. This is especially beneficial for building models that:

- Require **multiple inputs** and/or **multiple outputs**
- Use **shared layers** (e.g., identical weight layers applied to different inputs)
- Involve **non-linear data flows** (e.g., branching, merging, or skip connections)

This flexibility is crucial in **research** and in solving **complex problems** that require customized model behavior

⚙️ Syntax and Basic Workflow

In the Functional API:

- You begin by defining one or more **Input layers**, specifying the shape of the input data.
- Each **Dense layer** or transformation is then defined and explicitly connected to the previous layer's output.
- The model is completed by combining the input(s) and output(s) into a Model.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
# Define the input
inputs = Input(shape=(784,))
# Define a dense layer
x = Dense(64, activation='relu')(inputs)
# Define the output layer
outputs = Dense(10, activation='softmax')(x)

# Create the model
model = Model(inputs=inputs, outputs=outputs)
```

Each **Dense layer** has a parameter unit, which refers to the **dimensionality of the output space**. Activation functions such as relu (for hidden layers) and softmax (for multiclass classification outputs) are used to introduce non-linearities and enable learning complex mappings.

⚙️ Advanced Use Cases of Functional API

🔗 1. Multiple Inputs and Outputs

One of the key advantages of the Functional API is the ability to define models with **multiple input layers and multiple output layers**. This is particularly valuable in scenarios like **multitask learning**, where different outputs need to be predicted from shared and specialized parts of the network.

- Two separate input layers are defined, each with its own shape.
- Each input is processed through its own set of layers (independent branches).
- The outputs of these branches can be **merged** (e.g., concatenated), and additional layers can be applied.
- Finally, the network splits again to generate **multiple outputs**, each possibly corresponding to a different task.

This architecture supports learning **shared representations** while still making distinct predictions per task

```
from tensorflow.keras.layers import concatenate
# Define two sets of inputs
inputA = Input(shape=(64,))
inputB = Input(shape=(128,))
# The first branch operates on the first input
x = Dense(8, activation='relu')(inputA)
x = Dense(4, activation='relu')(x)
x = Model(inputs=inputA, outputs=x)
```

```
# The second branch operates on the second input
y = Dense(16, activation='relu')(inputB)
y = Dense(4, activation='relu')(y)
y = Model(inputs=inputB, outputs=y)
# Combine the output of the two branches
combined = concatenate([x.output, y.output])
# Apply a FC layer and then a regression prediction on the
combined outputs
z = Dense(2, activation='relu')(combined)
z = Dense(1, activation='linear')(z)
# The model will accept the inputs of the two branches and
then output a single value
model = Model(inputs=[x.input, y.input], outputs=z)
```

🔗 2. Shared Layers

Another powerful feature of the Functional API is the ability to **share layers across inputs**. This means a single layer (and its weights) can be applied to different inputs, which is essential in applications like **Siamese networks**.

In a Siamese network:

- A **shared Dense layer** is defined once.
- Two different input layers are processed using this shared layer (i.e., the same layer instance is applied to both).
- The outputs from these shared transformations are then compared or further processed.

This approach is used when two different inputs need to be processed **identically** to compare their embeddings or extract similarities.

```
from tensorflow.keras.layers import Lambda
# Define the input layer
input = Input(shape=(28, 28, 1))
# Define a shared convolutional base
conv_base = Dense(64, activation='relu')
# Process the input through the shared layer
processed_1 = conv_base(input)
processed_2 = conv_base(input)
# Create a model using the shared layer
model = Model(inputs=input, outputs=[processed_1, processed_2])
```

⚙️ Full Example: Combining All Features

In a more complex setup:

- The model has **two branches**, each starting from a different input.
- Each branch passes through **convolutional layers**, followed by **pooling** and **flattening** layers.
- The resulting outputs are **concatenated** into a merged representation.
- This merged vector is then passed through additional Dense layers.
- A final **single output** is produced (e.g., binary or multiclass classification).

This architecture demonstrates how multiple inputs, branching paths, and merging layers can all be implemented seamlessly using the Functional API.

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.activations import relu, linear
# First input model
inputA = Input(shape=(32, 32, 1))
x = Conv2D(32, (3, 3), activation=relu)(inputA)
x = MaxPooling2D((2, 2))(x)
x = Flatten()(x)
x = Model(inputs=inputA, outputs=x)
# Second input model
inputB = Input(shape=(32, 32, 1))
y = Conv2D(32, (3, 3), activation=relu)(inputB)
y = MaxPooling2D((2, 2))(y)
y = Flatten()(y)
y = Model(inputs=inputB, outputs=y)
# Combine the output of the two branches
combined = concatenate([x.output, y.output])
# Apply a FC layer and then a regression prediction on the
combined outputs
z = Dense(64, activation=relu)(combined)
z = Dense(1, activation=linear)(z)
# The model will accept the inputs of the two branches and
then output a single value
model = Model(inputs=[x.input, y.input], outputs=z)
```

🔹 Model Subclassing API: Custom and Dynamic Models

Beyond the Functional API, Keras also supports a **Model Subclassing API**, which offers the **most control and flexibility**.

Instead of defining a model by stacking or connecting layers declaratively, a model is defined by **subclassing tf.keras.Model** and **manually implementing the architecture** in code.

🧱 Structure of Subclassed Models

- Layers are **defined in the __init__() method**, like attributes of a class.
- The forward pass is written explicitly in the **call() method**, where the order and logic of applying each layer is controlled programmatically.

This model definition style is **imperative**, allowing the forward pass to contain conditional logic, loops, or dynamic behaviors — which is not possible in static graph models like Sequential or Functional API.

```
import tensorflow as tf
# Define your model by subclassing
class MyModel(tf.keras.Model):
    def __init__(self):
         super(MyModel, self).__init__()
        # Define layers
        self.dense1 = tf.keras.layers.Dense(64, activation='relu')
        self.dense2 = tf.keras.layers.Dense(10,
                        activation='softmax')
   def call(self, inputs):
        # Forward pass
        x = self.dense1(inputs)
        return self.dense2(x)
# Instantiate the model
model = MyModel()
# Define loss function and optimizer
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()
```

When to Use Subclassing API

Subclassing is particularly useful in advanced or unconventional scenarios:

- **Dynamic Architectures**: When the model's structure must change during runtime, based on input data or conditional flows. This is common in **reinforcement learning** or **adaptive models**.
- **Custom Training Loops**: When full control over training behavior is needed — such as tracking custom metrics, defining loss manually, or implementing specific update steps.
- **Research and Prototyping**: Useful when designing **novel layer types** or **experimental architectures** that don't fit neatly into Keras's standard layer types.

⚙️ Training with *tf.GradientTape*

Models built via subclassing are often trained using **custom training loops** with tf.GradientTape instead of the standard .fit() method.

The training loop involves:

- Running the forward pass manually
- Calculating loss
- Tracking gradients with tf.GradientTape
- Applying gradients using an optimizer

This method provides **finer control** over every training step.

```
# Define number of epochs
epochs = 5
# Create a dummy training dataset
(train_images, train_labels), _ = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(-1, 28*28).astype("float32") / 255
# Flatten and normalize
train_labels = train_labels.astype("int32")
# Create a tf.data dataset for batching
train_dataset = tf.data.Dataset.from_tensor_slices((train_images,
train_labels)).batch(32)
# Custom training loop
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    for x_batch, y_batch in train_dataset:
        with tf.GradientTape() as tape:
            predictions = model(x_batch, training=True)
            loss = loss_fn(y_batch, predictions)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients,
model.trainable_variables))
    print(f"Epoch {epoch+1}, Loss: {loss.numpy():.4f}")
```

Benefits of Subclassing API

Enables building models with **dynamic graphs**, where the structure can change on-the-fly during training.

Unlike the Functional and Sequential APIs (which define **static graphs**), subclassed models can adapt their architecture conditionally.

Useful for advanced domains such as:

- Reinforcement learning agents
- Attention-based models with adaptive masking
- Research models with evolving internal states or dynamic flow logic

☑Takeaways

☑ The **Functional API** allows building flexible, reusable, and modular networks with complex topologies, including multi-input/output models and shared-layer architectures.

☑ **Shared layers** and **non-linear data flows** enable design patterns like Siamese networks, multitask models, and hierarchical pipelines.

☑ The **Subclassing API** offers the ultimate freedom, enabling dynamic computation graphs, arbitrary control flows, and custom training logic.

☑ **Custom training loops** via `tf.GradientTape` allow training models in ways that are not supported by `.fit()`, giving total control over loss computation, updates, and behavior.

Module 2
Advanced CNNs in Keras

M2 - Section 1
Advanced CNNs and Data Augmentation

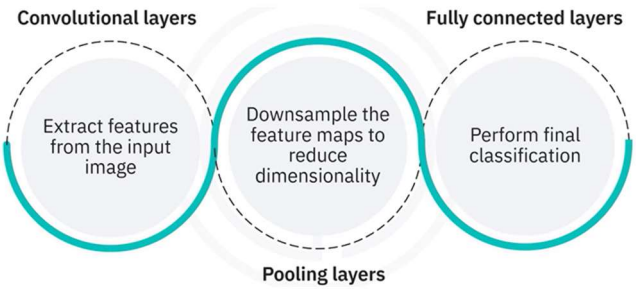
📌 Advance CNNs in keras

This section focuses on implementing **advanced convolutional neural network architectures** in Keras to improve performance on complex visual tasks. It builds on basic CNN knowledge and introduces architectural innovations used in state-of-the-art models like **VGG** and **ResNet**..

🔹 Understanding CNN Architecture

Convolutional Neural Networks (CNNs) mimic the human visual system by processing images through multiple layers:

- **Convolutional Layers**: When designing new algorithms or techniques that are not yet part of the Keras API.
- **Pooling Layers**: When an application requires a specific mathematical operation or architectural element that isn't supported natively.
- **Fully Connected Layers**: Tailoring layers to better handle specific types of data distributions, memory constraints, or latency-sensitive applications.



The structure below demonstrates a standard CNN stack:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
# Create a Sequential model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    MaxPooling2D((2,  2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Summary of the model
model.summary()
```

🔹 Advanced CNN Architecture

While the basic CNN is a powerful tool, complex tasks demand deeper and more efficient architectures. Popular advanced architectures include:

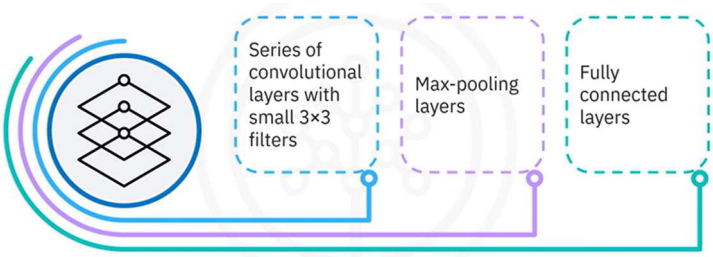
⚙️ VGG-Like Architecture

The **VGG architecture** is known for its **depth and simplicity**, relying on small 3×3 convolutional filters and stacking them in increasing depth.

It consists of a series of convolutional layers with small three by three filters, followed by Max-pooling layers and fully connected layers.

VGG Principles:

- Use of small filters (3×3)
- Deep stacking of layers
- MaxPooling after each convolutional block



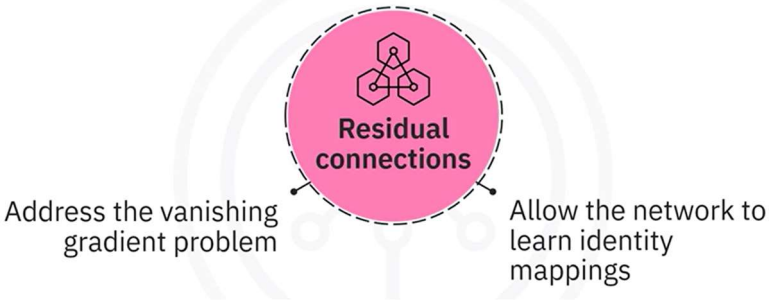
Implementing a VGG-Like Architecture in Keras:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
# Create a VGG-like Sequential model
model = Sequential([
    Conv2D(64, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(256, (3, 3), activation='relu'),
    Conv2D(256, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(512, activation='relu'),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
# Summary of the model
model.summary()
```

⚙️ ResNet-Like Architecture

ResNet (Residual Network) introduces **residual connections** to allow deeper network training by solving the **vanishing gradient problem**.



💡 Key Concepts:

- **Residual Blocks**: Learn identity mappings using shortcut connections.
- **Improves gradient flow**, enabling very deep networks to converge.

Here's an example of implementing a ResNet-Like architecture in Keras:

The residual block function defines a residual block with two convolutional layers and a shortcut connection.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, BatchNormalization,
Activation, Add, Flatten, Dense

def residual_block(x, filters, kernel_size=3, stride=1):
    shortcut = x
    x = Conv2D(filters, kernel_size, strides=stride, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = Conv2D(filters, kernel_size, strides=1, padding='same')( x)
    x = BatchNormalization()(x)
    x = Add()([x, shortcut])
    x = Activation('relu')(x)
    return x
```

```
input = Input(shape=(64, 64, 3))
x = Conv2D(64, (7, 7), strides=2, padding='same')(input)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = residual_block(x, 64)
x = residual_block(x, 64)
x = Flatten()(x)
outputs = Dense(10, activation='softmax')(x)
# Create the model
model = Model(inputs=input, outputs=outputs)

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Summary of the model
model.summary()
```

☑ Takeaways

☑ CNNs consist of convolutional, pooling, and fully connected layers that together extract and interpret visual features.

☑ **Basic CNNs** are foundational, but advanced tasks benefit from deeper and more specialized architectures.

☑ **VGG** networks use small filters with deep stacks for hierarchical feature learning.

☑ **ResNet** networks introduce **residual connections** to overcome training difficulties in deep networks.

☑ Keras enables flexible implementation of both basic and advanced CNNs with clean and modular APIs.

📌 Data Augmentation Techniques in Keras

Data augmentation is a **crucial technique** for training **robust and generalizable models** in computer vision. By artificially increasing the diversity of training data through image transformations, models learn to better identify invariant patterns and avoid overfitting.

🎯 **Purpose of Data Augmentation:**

- Introduces **variation** to training inputs without requiring new data.
- Helps models **generalize** to unseen examples by simulating real-world conditions.
- Reduces the risk of **overfitting** by exposing the model to multiple versions of the same image.

🔹 Basic Augmentation Techniques

Keras provides the ***ImageDataGenerator*** class to apply a variety of common image augmentations.

☑ **Available Transformations:**

- **Rotation**: Rotates image by a random angle.
- **Width Shift / Height Shift**: Randomly moves the image along horizontal or vertical axis.
- **Shear**: Applies geometric distortion (shearing).
- **Zoom**: Random zoom-in or zoom-out.
- **Horizontal Flip**: Flips the image left-to-right.
- **Rescaling**: Normalizes pixel values.

Let's see an example of how to apply common augmentations like rotation, width shift, height shift and horizontal flip

```
# Create an instance of ImageDataGenerator with augmentation options
datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Load a sample image and reshape it
from tensorflow.keras.preprocessing
import image
img = image.load_img('sample.jpg')
x = image.img_to_array(img)
x = x.reshape((1,) + x.shape)

# Generate batches of augmented images
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
```

Normalization-Based Augmentation

Keras also allows **feature-wise and sample-wise normalization**, which standardizes input images either globally across the dataset or locally for each image.

- **Options:**
  - **featurewise_center:** Set global dataset mean to 0
  - **featurewise_std_normalization:** Normalize dataset to std = 1
  - **samplewise_center:** Set each image's mean to 0
  - **samplewise_std_normalization:** Normalize each image to std = 1

Let's see an example of using feature-wise and sample-wise normalization.

```
# Create an instance of ImageDataGenerator with advanced options
datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    samplewise_center=True,
    samplewise_std_normalization=True
)
# Compute the mean and standard deviation on a dataset of images
datagen.fit(training_images)
```

```
# Generate batches of normalized images
i = 0
for batch in datagen.flow(training_images, batch_size=32):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break
plt.show()
```

🔹 Custom Augmentation Functions

Keras allows developers to define **custom augmentation logic** using the ***preprocessing_function*** parameter.

This gives complete control over how each image is transformed.

Custom functions can implement domain-specific distortions, simulate hardware artifacts, or apply stochastic noise patterns for robustness.

```
import numpy as np
def add_random_noise(image):
    noise = np.random.normal(0, 0.1, image.shape)
    return image + noise
# Create an instance of ImageDataGenerator with custom augmentation
datagen = ImageDataGenerator(preprocessing_function=add_random_noise)
# Generate batches of augmented images
i = 0
for batch in datagen.flow(training_images, batch_size=32):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break
plt.show()
```

☑ Takeaways

☑ Data augmentation improves **model generalization** by simulating diverse visual conditions.

☑ **ImageDataGenerator** supports essential transformations like rotation, translation, shear, and flip.

☑ **Normalization techniques** (feature-wise or sample-wise) standardize image inputs to stabilize training.

☑ **Custom augmentation functions** unlock flexibility and allow full control over data preprocessing.

☑ These augmentation techniques are critical for **computer vision tasks** involving limited data or high variability.

M2 - Section 2
Transfer Learning and Image Processing

📌 Transfer Learning in Keras

🔹 What Is Transfer Learning?

Transfer learning is a technique where a model developed for one task is reused as the starting point for a model on a second task. It's especially powerful when you have limited training data for the new task.

- It works similarly to how humans apply known skills to new but related challenges.
  Example: Knowing how to play the piano helps you learn the organ more quickly.

- In deep learning, it means **reusing a model pre-trained on a large dataset** (like ImageNet), and fine-tuning it on a smaller, task-specific dataset.

Involves taking a **pre-trained model**, such as VGG16 trained on ImageNet, and adapting it to a new dataset.

The idea is to reuse the **feature extraction layers** (like convolutional layers) while customizing the final layers for the new task.

Enables **faster convergence**, **higher accuracy**, and **training on smaller datasets**.

Why use Transfer Learning?

- **Reduced Training Time**:
  The model already learned general visual features (edges, textures, shapes), so training starts from a strong baseline.

- **Improved Performance**:
  Pre-trained models, optimized on large datasets, tend to generalize better on smaller ones.

- **Works Well with Small Datasets**:
  Even with limited data, transfer learning achieves high accuracy because the model retains useful pre-learned features.

These advantages make it suitable for domains like **medical imaging** or **natural language processing**, where labeled data is scarce or expensive

Implementing Transfer Learning with Keras (Using VGG-16)

1. Import Required Modules

```
!pip install --upgrade tensorflow numpy Pillow

import sys
import os
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from PIL import Image
import numpy as np

# Increase recursion limit (temporary solution)
sys.setrecursionlimit(10000)
```

If you are working in environments with low default recursion limits (like Google Colab), increase the recursion depth to avoid runtime errors during model loading or training.

2. (Optional) Generate Sample Training Data

```
# Ensure the dataset directory exists and generate sample data if needed
def generate_sample_data():
    os.makedirs('/content/training_data/class_1', exist_ok=True)
    os.makedirs('/content/training_data/class_2', exist_ok=True)

    for i in range(10):
        img = Image.fromarray(np.random.randint(0, 255, (224, 224, 3),
dtype=np.uint8))
        img.save(f'/content/training_data/class_1/img_{i}.jpg')
        img.save(f'/content/training_data/class_2/img_{i}.jpg')

# Generate sample data (uncomment if you need to generate data)
# generate_sample_data()
```

This was done to demonstrate transfer learning, create a simple image classification dataset:

- Two directories (*class_1/* and *class_2/*) are created under *training_data/*
- Randomly generated images are saved in each class directory to simulate a binary classification scenario

3. Load the Pre-trained VGG-16 Model

```
# Load the VGG16 model pre-trained on ImageNet
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=(224, 224, 3))
```

Import VGG16 from tensorflow.keras.applications:

- ***weights='imagenet'*** means it uses pre-trained weights. The model is pre-trained on ImageNet, which includes millions of images across thousands of categories.
- ***include_top=False*** argument excludes the original classifier layers, so you can add your own custom output layers.
- ***input_shape=(224, 224, 3)*** defines the shape of input images the model expects — 224×224 pixels and 3 color channels (RGB).

4. Freeze the Pre-trained Base Layers

```
# Freeze the base model layers
for layer in base_model.layers:
    layer.trainable = False
```

To use the model as a **feature extractor**:

- You **iterate over all layers** in the base model. For each one, set layer.trainable = False.

This prevents the base model's weights from being updated during training, preserving the general image features it learned from ImageNet.

5. Create a New Model with Custom Classifier

```
# Create a new model and add the base model and new layers
model = Sequential([
    base_model,
    Flatten(),
    Dense(256, activation='relu'),
    Dense(1, activation='sigmoid')  # Change to the number of classes you
have
])

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

Using the Keras Sequential model, we stack:

- The **frozen base model** (VGG16 convolutional layers)
- **Flatten() layer** to transform the convolutional output into a 1D array
- **Dense(256, activation='relu') layer**: a fully connected layer to learn new patterns
- **Final Dense(1, activation='sigmoid') layer**:
  This is the binary classification output (e.g., cat vs. dog)

The model is compiled with:

- **optimizer='adam'**: A fast, adaptive optimizer
- **loss='binary_crossentropy'**: Since this is a binary classification task
- **metrics=['accuracy']**: To track classification performance during training

6. **Prepare Data**

```
# Load and preprocess the dataset
train_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
    'training_data',
    target_size=(224, 224),
    batch_size=32,
    class_mode='binary'
)
```

You create an instance of ImageDataGenerator with rescale=1./255, which:

- Normalizes pixel values to a range between 0 and 1.

Using flow_from_directory():

- You specify the training data path (**'training_data'**)
- **target_size=(224, 224):** All input images are resized to match the model's input shape
- **batch_size=32:** Images are processed in batches of 32
- **class_mode='binary':** Automatically assigns binary labels based on folder names

  📂 for example, folder class_1/ will be one class, and class_2/ will be the other.

7. **Train the Model**

```
# Train the model
model.fit(train_generator, epochs=10)
```

The model is trained using the **fit()** method, and uses **train_generator** as input. In this case the model is trained with *10 epochs* (it goes through the entire dataset 10 times)

🔹 Fine-Tuning the Pre-trained model layers

Instead of using the base model strictly as a feature extractor, you can **fine-tune some of its layers.**

```
# Unfreeze the top layers of the base model
for layer in base_model.layers[-4:]:
    layer.trainable = True

# Compile the model again
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Train the model again
model.fit(train_generator, epochs=10)
```

- Set **layer.trainable = True** for the **top 4 layers** of the base model
- Re-compile the model after unfreezing
- Train the model again with the same fit() call and dataset

This step lets the model adapt high-level pre-trained features to your specific task, improving performance even further.

☑Takeaways

☑ Transfer learning **reduces training time**, boosts **performance**, and is especially helpful when working with **limited data**. It enables you to reuse knowledge from one task (e.g., ImageNet classification) for another.

☑ Pre-trained models like **VGG16** can be used as fixed feature extractors or fine-tuned for specific use cases.

☑ Keras makes it easy to import, freeze, and adapt pre-trained models to new tasks.

☑ Transfer learning is especially useful for domains where labeled data is scarce, such as:

- Medical imaging
- Natural language processing
- Object detection and classification

📌 Using Pre-Trained Models

Pre-trained models are deep convolutional networks trained on large-scale datasets such as **ImageNet**. These models capture low-level and high-level visual features like edges, textures, shapes, and object components through their convolutional layers.

Instead of training a new model from scratch, these pre-trained models can be reused for new tasks in two main ways:

1. **As fixed feature extractors**, without updating the weights.
2. **With fine-tuning**, to adjust some of the layers to fit the new task better.

Both approaches are widely used in real-world deep learning pipelines where labeled data is limited or compute resources are constrained.

🔹 Using Pre-trained Models as Feature Extractors

A pre-trained model can be used as a **static feature extractor**, where:

- The convolutional base processes new input images to generate **feature maps**.
- These feature maps are passed to new, task-specific **fully connected layers**.
- The **pre-trained weights are not updated** — the base is frozen.

This technique is effective when:

- The new dataset is small or similar in nature to the original dataset.
- The computational budget is limited.
- The goal is fast deployment with minimal training time.

⚡ Advantages of Fixed Feature Extraction:

- **No additional training required**
  The convolutional base processes new data without weight updates.
- **Efficient use of learned representations**
  The pre-trained layers offer **rich hierarchical features** that can be reused directly.
- **Ideal for limited resources**
  Reduces memory and compute demands — suitable for laptops or edge devices.
- **Effective even with small datasets**
  Particularly useful in domains like **medical imaging** or **satellite image classification**, where labeled data is expensive to collect.

🧠 Example of Pre-trained model used as Feature Extractor

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam

# Load the VGG16 model pre-trained on ImageNet
base_model = VGG16(weights='imagenet'
, include_top=False, input_shape=(224, 224, 3))

# Freeze all layers initially
for layer in base_model.layers:
    layer.trainable = False

# Create a new model and add the base model and new layers
model = Sequential([
    base_model,
    Flatten(),
    Dense(256, activation='relu'),
    Dense(1, activation='sigmoid')

# Change to the number of classes you have
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001),
loss='binary_crossentropy', metrics=['accuracy'])

# Load and preprocess the dataset
train_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
    '/content/sample_data',
    target_size=(224, 224),
    batch_size=32,
    class_mode='binary'
)

# Train the model with frozen layers
model.fit(train_generator, epochs=10)
```

🛠️ Implementation Workflow:

- Load a pre-trained model (e.g., VGG16) with **include_top=False**.
- Freeze the base layers by setting **layer.trainable = False**.
- Add custom layers for the new classification task.
- Compile the model with an appropriate optimizer and loss.
- Load the dataset using **ImageDataGenerator** with rescaling.
- Train only the newly added top layers.

🔹 Fine-Tuning Pre-trained Models

While using a pre-trained model as a fixed feature extractor is effective, **fine-tuning** allows for additional performance improvements by:

- **Unfreezing part of the convolutional base**, typically the upper layers.
- **Jointly training** the newly added dense layers along with the unfrozen base layers.

This process updates weights in the upper convolutional layers to better **adapt the learned representations** to the new dataset.

⚡ When to use Fine-Tuning:

Fine-tuning is especially useful when:

- The **new dataset differs** significantly from the dataset used to train the original model.
- There's **enough data to support partial retraining**, but not enough for full model training.
- You want to **preserve general features** from the base model while adapting to task-specific details.

⚡ Advantages of Fine-Tuning:

- **Better alignment with new data**
  Fine-tuning lets the model learn more nuanced features relevant to the new dataset.
- **Improved performance**
  Especially when the new task is **similar but not identical** to the original.
- **Efficient transfer learning**
  Instead of training a deep network from scratch, you build on **already-learned knowledge**.
- **Supports low-data scenarios**
  Fine-tuning enables strong results even when **full supervised training isn't feasible**.

🧠 Example of Pre-trained model used as Feature Extractor

ℹ️ **NOTE**: When applying fine-tuning its always better **to initially train the Neural Network with all pre-trained model layers freeze**, so we have a base model to compare with.

```
# Gradually unfreeze layers and fine-tune
for layer in base_model.layers[-4:]:  # Unfreeze the last 4 layers
    layer.trainable = True

# Compile the model again with a lower learning rate for fine-tuning
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='binary_crossentropy', metrics=['accuracy'])

# Fine-tune the model
model.fit(train_generator, epochs=10)
```

🛠️ Implementation Workflow:

- **Use a pre-trained model** with the top layers removed.
- **Freeze** all layers initially to train only the new classifier layers.
- **Unfreeze a subset of the top layers** in the base model (e.g., last 4 layers).
- **Recompile the model** to include the updated trainable configuration.
- **Train again**, this time jointly updating the unfrozen base layers and the classifier.

☑Takeaways

☑ Pre-trained models can be used as fixed feature extractors for tasks like classification, clustering, and visualization.

☑ Feature extraction requires no additional training and works well with limited data and compute.

☑ Fine-tuning selectively updates upper layers of the pre-trained model to better adapt to new datasets.

☑ Transfer learning enables high performance even when the new dataset differs from the original training data.

☑ Using pre-trained models accelerates development, improves accuracy, and reduces the need for large labeled datasets.

📌Image Processing with TensorFlow

TensorFlow offers a powerful suite of tools and APIs for performing **image processing tasks**, from basic transformations to model-ready pipelines. These capabilities are essential for building scalable and efficient computer vision applications.

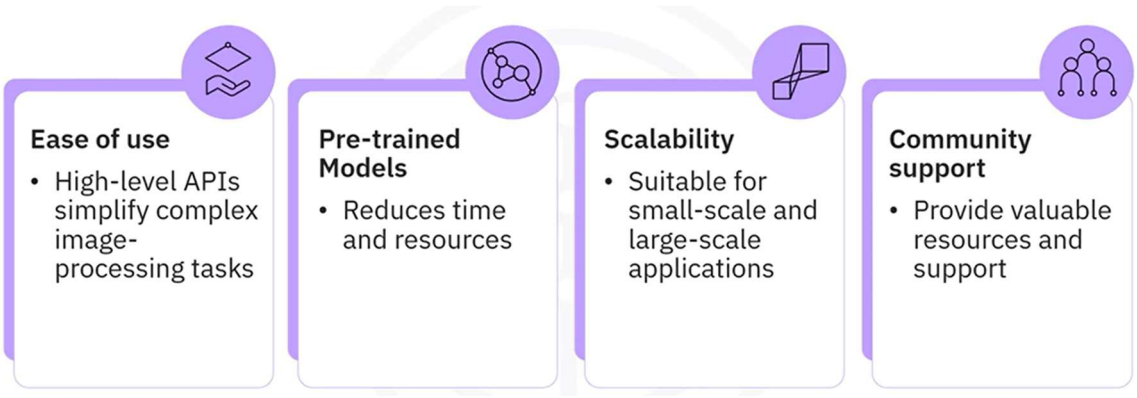
📘 What Is Image Processing?

Image processing refers to the practice of **transforming or analyzing images** to extract useful information or to prepare data for model training. It is a foundational step in computer vision pipelines.

Image processing is used across a wide range of fields, including:

- **Medical Imaging** – Analyzing scans for diagnostics (e.g., X-rays, MRIs)
- **Autonomous Vehicles** – Object detection and environment mapping
- **Facial Recognition** – Identifying or verifying individuals from images
- **Robotic Vision** – Interpreting environments for interaction

⚙️ Why Use TensorFlow for Image Processing?



Basic Image Processing Tasks in TensorFlow

The process for preparing images with TensorFlow includes a series of preprocessing steps. These steps are crucial for ensuring that the input data is in the correct format for inference or training.

1. Image Loading and Resizing

- Load an image from disk
- Resize the image to the dimensions expected by the model (e.g., 224 × 224 pixels)

2. Convert to Array

- Convert the image into a NumPy array, which is the format TensorFlow models operate on.

3. Add Batch Dimension

- Models expect input with a batch dimension, which is necessary for model predictions.
- Expand the image shape to match: [batch_size, height, width, channels].

This is required even when working with a **single image**, as TensorFlow models are batch-oriented.

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator,
img_to_array, load_img

# Load and preprocess the image
img = load_img('/content/path_to_image.jpg', target_size=(224, 224))
img_array = img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Add batch dimension
```

```
# Display the original image
import matplotlib.pyplot as plt
plt.imshow(img)
plt.show()
```

🔹 Image Augmentation in TensorFlow

Data augmentation is the process of applying random transformations to training images in order to improve a model's generalization and robustness.

These transformations simulate variations in the training data and help prevent overfitting.

✨ **Common Augmentations Include:**

- **Rotation** – Randomly rotate the image
- **Translation (Shift)** – Move image left, right, up, or down
- **Shear** – Apply shearing transformation
- **Zoom** – Randomly zoom in/out
- **Horizontal Flip** – Flip image horizontally

📦 **Workflow:**

- Configure augmentation parameters
- Use data generators to apply augmentations during training
- Augmented images are produced in batches to feed into models

This setup ensures the model sees a **diverse range of image variants** across epochs, enhancing its ability to generalize.

☑Takeaways

☑ TensorFlow provides robust tools for **loading, preprocessing, and augmenting images**.

☑ It supports both **basic transformations** (e.g., resizing, batching) and **advanced augmentations** (e.g., rotation, zoom, flip).

☑ These capabilities make it easier to train robust models with better generalization on unseen data.

☑ TensorFlow's ecosystem provides **scalability**, **pre-trained models**, and **developer support**, making it ideal for modern image processing workflows.

📌 Tips for Transfer Learning Implementation

Transfer learning is a powerful technique that enables the use of pre-trained models on new tasks, significantly saving time and computational resources. Here are key tips for implementing transfer learning effectively:

1. **Choose the right pre-trained model:** Select a model trained on a dataset similar to your target task to enhance performance. Popular models like VGG16, ResNet, or InceptionV3 are particularly effective for image-related tasks. Ensure that the architecture aligns with your specific problem requirements.

2. **Freeze early layers:** In the initial training stages, freeze the early layers of the pre-trained model to preserve their learned features. This approach is beneficial when working with small datasets or datasets that closely resemble the original dataset the model was trained on.

3. **Fine-tune later layers:** As training progresses, gradually unfreeze the deeper layers and fine-tune them. These layers capture task-specific features, and fine-tuning allows the model to adapt better to the nuances of your new dataset.

4. **Adjust learning rates:** Use a lower learning rate for fine-tuning to prevent catastrophic forgetting of the pre-trained knowledge. High learning rates during this phase can disrupt the learned features and degrade model performance.

5. **Use data augmentation:** Implement data augmentation techniques, particularly for image tasks, to increase variability within the dataset. This practice helps prevent overfitting and enhances the model's ability to generalize.

6. **Consider domain adaptation:** If there is a significant disparity between the domain of the pre-trained model and your target task, consider applying domain adaptation techniques. These methods can help align the source and target datasets, improving the model's performance.

By following these tips, you can optimize your use of transfer learning, enhancing your model's performance with minimal additional effort.

M2 - Section 3
Introduction to Transpose Convolution

📌 Transpose Convolution

Transpose convolution — also known as **deconvolution** — is a critical operation used in deep learning for **increasing spatial resolution** within image-based neural networks. It serves as the **inverse of the convolution operation**, and is frequently used for **image generation**, **super-resolution**, and **semantic segmentation**.

🔹 What Is Transpose Convolution?

In standard convolution, a kernel slides over the input feature map to **reduce spatial dimensions** and extract meaningful features.

While this down-sampling is essential for recognition tasks, some applications require the opposite — **up-sampling** the input to produce **higher-resolution outputs**.

Transpose convolution addresses this need by:

- **Expanding** the spatial dimensions of the input
- Performing an **inverse convolution operation**
- Producing an **up-sampled feature map** that preserves structural characteristics of the original input

Common Applications:

Transpose convolutions are essential in models and systems that require **spatial reconstruction** or **pixel-level predictions**:

- **Generative Adversarial Networks (GANs)**
  Used to generate high-resolution images from latent vectors.
- **Image Super-Resolution**
  Enhances the resolution of low-quality images.
- **Semantic Segmentation**
  Used to up-sample feature maps for **pixel-wise classification**, often producing detailed masks.

🔹 How Transpose Convolution Works

The process involves two key steps:

1. **Zero Insertion**
   Zeros are inserted between elements of the input feature map to **expand the spatial dimensions**.

2. **Convolution Over Expanded Input**
   A standard convolution is then applied over the zero-inserted feature map using a defined kernel. This process fills in the gaps and generates a **larger output** with learned patterns.

This approach increases the resolution while retaining learned representations from lower-resolution input.

🔹 Implementing Transpose Convolution in Keras

To implement a transpose convolution layer in Keras, follow a simple model construction pattern:

```
import os
import logging
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2DTranspose

# Define the input layer
input_layer = Input(shape=(28, 28, 1))

# Add a transpose convolution layer
transpose_conv_layer = Conv2DTranspose(filters=32, kernel_size=(3, 3),
strides=(2, 2), padding='same', activation='relu')(input_layer)

# Define the output layer
output_layer = Conv2DTranspose(filters=1, kernel_size=(3, 3),
activation='sigmoid', padding='same')(transpose_conv_layer)

# Create the model
model = Model(inputs=input_layer, outputs=output_layer)

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error',
metrics=['accuracy'])

# Display the model summary
model.summary()
```

⚠️ Best Practices and Pitfalls

Checkerboard Artifacts

A known issue with transpose convolution is the appearance of **checkerboard artifacts**, which arise due to uneven overlap of kernel applications during up-sampling.

These artifacts can affect visual quality and prediction accuracy.

Mitigation Strategy

To reduce these effects, apply the following approach:

1. Use **UpSampling2D** to increase spatial dimensions smoothly (e.g., by a factor of 2)
2. Follow it with a **standard Conv2D layer** to refine the output

This two-step method often yields better visual quality and reduces unintended noise patterns.

Let's see an example where we reduce the artifacts and improve the quality of the up-sampled images.

In this example, up-sampling 2D performs up-sampling by a factor of two, and applies a convolution layer to refine the up-sampled output.

```
import os
import logging
import tensorflow as tf
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Input, UpSampling2D, Conv2D

# Define the input layer
input_layer = Input(shape=(28, 28, 1))

# Define a model with up-sampling followed by convolution to avoid
checkerboard artifacts
x = UpSampling2D(size=(2, 2))(input_layer)
output_layer = Conv2D(filters=64, kernel_size=(3, 3), padding='same')(x)

# Create the model
model = Model(inputs=input_layer, outputs=output_layer)

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error',
metrics=['accuracy'])

# Display the model summary
model.summary()
```

Involves taking a **pre-trained model**, such as VGG16 trained on ImageNet, and adapting it to a new dataset.

The idea is to reuse the **feature extraction layers** (like convolutional layers) while customizing the final layers for the new task.

Enables **faster convergence**, **higher accuracy**, and **training on smaller datasets**.

Why use Transfer Learning?

- **Reduced Training Time**:
  The model already learned general visual features (edges, textures, shapes), so training starts from a strong baseline.
- **Improved Performance**:
  Pre-trained models, optimized on large datasets, tend to generalize better on smaller ones.
- **Works Well with Small Datasets**:
  Even with limited data, transfer learning achieves high accuracy because the model retains useful pre-learned features.

These advantages make it suitable for domains like **medical imaging** or **natural language processing**, where labeled data is scarce or expensive

☑ Takeaways

☑ Transpose convolution is used to **up-sample** feature maps in tasks such as image generation, super-resolution, and segmentation.

☑ It works by **inserting zeros** into the input feature map and applying convolution to increase spatial dimensions.

☑ Keras provides native support for this through layers like Conv2DTranspose.

☑ **Checkerboard artifacts** are a common issue, and can be mitigated by combining **UpSampling2D** with a **Conv2D** layer.

☑ Transpose convolution enables models to perform **pixel-level reconstruction** and generate higher-resolution outputs, which are essential for many image processing applications.

Module 3

Transformers in Keras

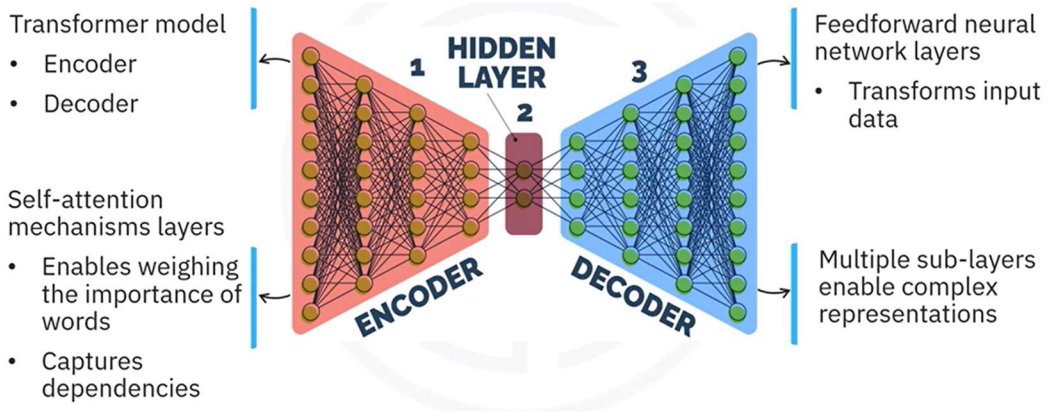
M3 – Section 1
Introduction to Transformers in Keras

📌 Introduction to Transformers in Keras

Transformers have revolutionized deep learning, particularly in **natural language processing (NLP)**, and are now applied to a broader range of domains such as **image processing**, **time series prediction**, and other sequence modeling tasks.

First introduced by **Vaswani et al.** in the seminal paper *"Attention is All You Need"*, transformers utilize **self-attention mechanisms** to process data in parallel, overcoming the sequential limitations of earlier models like RNNs.

🔹 Transformer Architecture Overview



The transformer model consists of **two main components**:

Encoder	Decoder
Responsible for processing the input sequence. Contains multiple layers, each with: • Self-attention mechanism • Feed-forward neural network • Residual connections and layer normalization.	Generates de output sequence. Contains multiple layers, each with: • Self-attention mechanism • Cross-attention mechanism (attending to encoder output) • Feed-forward neural network • Residual connections and layer normalization

Transformers components explanation:

- **Feed-Forward Neural Network**

  The encoder and decoder apply a **feed-forward neural network** in each layer after the attention mechanisms.

  - The feed-forward network processes each position independently.
  - It consists of two fully connected layers.
  - It applies an activation function between the layers to introduce non-linearity.

- **Residual Connections and Layer Normalization**

  - The model applies **residual connections** after self-attention and feed-forward sub-layers to help stabilize training.
  - It applies **layer normalization** after each residual connection to normalize the layer outputs and maintain stable gradients.

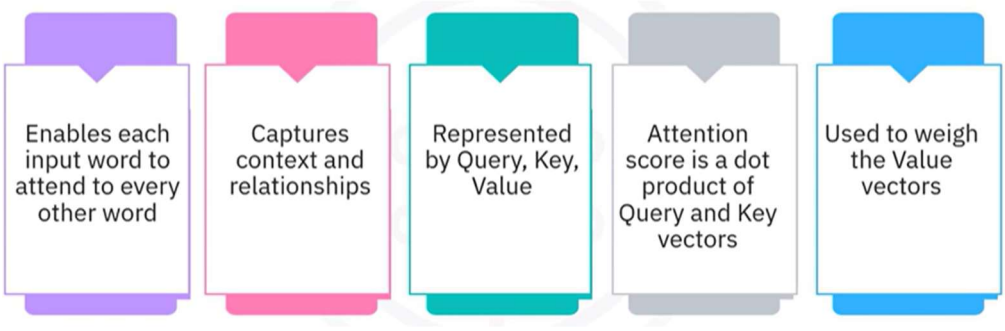
- **Positional Encoding**

  The model uses **positional encoding** to inject position information into the input embeddings.

  - It applies positional encoding immediately after embedding the input sequence.
  - This allows the model to understand the order of the tokens while processing the sequence in parallel

🧠 Key Components Explained

⚡ Self-Attention Mechanism



Self-attention allows the model to focus on **different words within the same sequence** when encoding a particular word, enabling the capture of **long-range dependencies** regardless of their position in the sequence.

Each token is represented by three vectors:

- **Query (Q)**
- **Key (K)**
- **Value (V)**

The attention score is calculated as the **dot product of the Query and Key vectors**, followed by a **softmax activation** to generate attention weights. These weights are applied to the **Value vectors**, resulting in context-aware outputs.

This mechanism ensures that each word attends to **every other word in the sequence**, enriching contextual understanding.

Example of Self-Attention Calculation:

```
import tensorflow as tf
from tensorflow.keras.layers import Layer
class SelfAttention(Layer):
  def __init__(self, d_model):
    super(SelfAttention, self).__init__()
    self.d_model = d_model
    self.query_dense = tf.keras.layers.Dense(d_model)
    self.key_dense = tf.keras.layers.Dense(d_model)
    self.value_dense = tf.keras.layers.Dense(d_model)

  def call(self, inputs):
    q = self.query_dense(inputs)
    k = self.key_dense(inputs)
    v = self.value_dense(inputs)

    attention_weights = tf.nn.softmax(tf.matmul(q, k, transpose_b=True) /
tf.math.sqrt(tf.cast(self.d_model, tf.float32)), axis=-1)
    output = tf.matmul(attention_weights, v)

    return output
# Example usage
inputs = tf.random.uniform((1, 60, 512)) # Batch size of 1, sequence
length of 60, and model dimension of 512
self_attention = SelfAttention(d_model=512)
output = self_attention(inputs)
print(output.shape) # Should print (1, 60, 512)
```

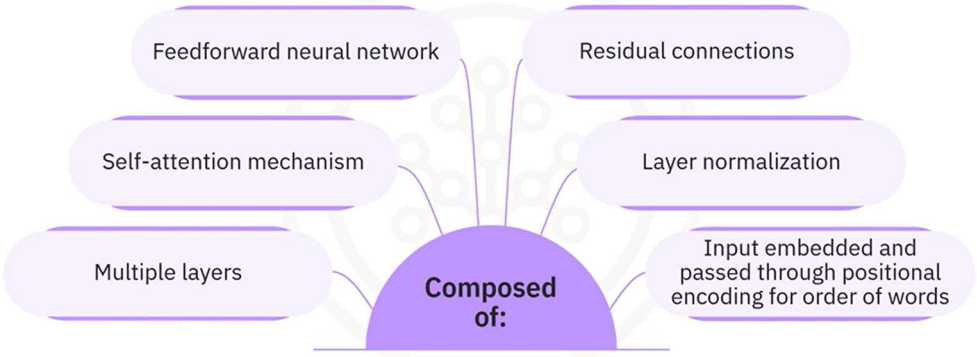
In this code example, the self-attention class defines the self-attention mechanism.

The **init method** initializes the dense layers for query, key and value projections.

The **call method** calculates the attention scores, applies softmax normalization, and computes the output by applying the attention weights to the value vectors.

The **tf.nn.softmax parameter** applies the Softmax function to the attention scores to get the attention weights.

Transformer Encoder



The **transformer encoder** is composed of multiple layers, each containing:

- **Self-attention mechanism**: Enables the encoder to model relationships between all tokens in the input sequence.
- **Feed-forward neural network**: Further transforms the representations after self-attention.
- **Residual connections and layer normalization**: Applied after both self-attention and feed-forward sublayers to stabilize and accelerate training.

The input to the encoder is first embedded and then passed through positional encoding to add information about the position of words in the sequence. This helps the model understand the order of words.

Each encoder layer learns progressively more complex representations of the input sequence, capturing both local and global relationships.

Example of Transformer Encoder:

```
class TransformerEncoder(Layer):
  def __init__(self, d_model, num_heads, dff, rate=0.1):
    super(TransformerEncoder, self). __init__()

    self.mha = tf.keras.layers.MultiHeadAttention(num_heads=num_heads,
key_dim=d_model)
    self.ffn = tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),
      tf.keras.layers.Dense(d_model)
    ])

    self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.dropout1 = tf.keras.layers.Dropout(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)
```

```
  def call(self, x, training, mask):
    attn_output = self.mha(x, x, x, attention_mask=mask) # Self attention
    attn_output = self.dropout1(attn_output, training=training)
    out1 = self.layernorm1(x + attn_output) # Residual connection and
normalization

    ffn_output = self.ffn(out1) # Feed forward network
    ffn_output = self.dropout2(ffn_output, training=training)
    out2 = self.layernorm2(out1 + ffn_output)
# Residual connection and normalization
    return out2

# Example usage
encoder = TransformerEncoder(d_model=512, num_heads=8, dff=2048)
x = tf.random.uniform((1, 60, 512))
```

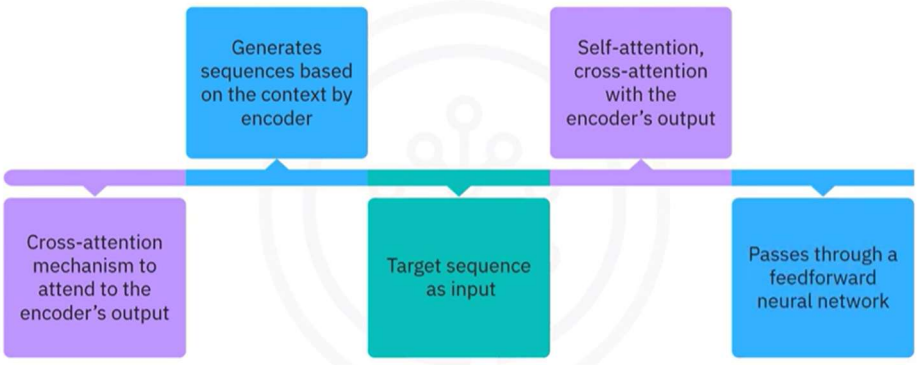
In the example, the transformer encoder class defines the transformer encoder.

The **init method** initializes the multi head attention, feed forward network, layer normalization, and dropout layers.

The **multi head attention** method applies multi head attention to the input.

The **call method** applies self-attention, residual connection, normalization, feed forward network, and other residual connection and normalization.

Transformer Decoder



The transformer decoder is similar to the encoder, but with an additional cross attention mechanism to attend to the encoders output.

- The decoder takes the target sequence as input.
- The input passes through an embedding layer and positional encoding.
- The decoder applies **self-attention** to the target sequence, followed by a **residual connection and layer normalization**.
- It then applies **cross-attention** where the decoder attends to the encoder's output.
- This allows the decoder to use context from the encoder while generating output tokens.

- After cross-attention, the result is passed through another **residual connection and layer normalization**, followed by a **feed-forward neural network**, and another **residual connection and layer normalization**.

This design allows the decoder to generate sequences based on the context provided by the encoder output and the previously generated tokens.

Example of Transformer Decoder:

```
class TransformerDecoder(Layer):
  def __init__ (self, d_model, num_heads, dff, rate=0.1):
    super(TransformerDecoder, self).__init__()

    self.mha1 = tf.keras.layers.MultiHeadAttention(num_heads=num_heads,
key_dim=d_model)
    self.mha2 = tf.keras.layers.MultiHeadAttention(num_heads=num_heads,
key_dim=d_model)
    self.ffn = tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),
      tf.keras.layers.Dense(d_model)
    ])

    self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.dropout1 = tf.keras.layers.Dropout(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)
    self.dropout3 = tf.keras.layers.Dropout(rate)

  def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
    attn1 = self.mha1(x, x, x, attention_mask=look_ahead_mask) # Self
attention
    attn1 = self.dropout1(attn1, training=training)
    out1 = self.layernorm1(x + attn1) # Residual connection and
normalization

    attn2 = self.mha2(out1, enc_output, enc_output,
attention_mask=padding_mask) # Cross attention
    attn2 = self.dropout2(attn2, training=training)
    out2 = self.layernorm2(out1 + attn2) # Residual connection and
normalization
    ffn_output = self.ffn(out2) # Feed forward network
    ffn_output = self.dropout3(ffn_output, training=training)
    out3 = self.layernorm3(out2 + ffn_output) # Residual connection and
normalization
    return out3
```

The **init** method initializes the multi head attention, feed forward network, layer normalization, and drop out layers.

The **multi head attention** method applies multi head attention to the input and encoder output.

The **call** method applies self-attention, cross attention, residual connection, normalization, feed forward network, and another residual connection and normalization.

☑Takeaways

☑ Transformers are composed of an encoder and a decoder, both built from layers that include self-attention mechanisms and feed-forward neural networks.

☑ Self-attention allows each word to attend to every other word in the input, enabling the model to capture relationships and dependencies across the entire sequence.

☑ The encoder processes the input by embedding the sequence, applying positional encoding, and passing it through layers that perform self-attention, feed-forward operations, residual connections, and layer normalization.

☑ The decoder processes the target sequence by applying self-attention, cross-attention with the encoder's output, and feed-forward networks, all followed by residual connections and layer normalization.

☑ Understanding and implementing transformers enables the development of powerful models for natural language processing, image processing, and time series prediction.

📌Building Transformers for Sequential Data

Transformers are designed to efficiently handle sequential data such as time series, text, and audio. These types of data require models that can process sequences where each element depends on the elements that came before it.

Traditional models such as **Recurrent Neural Networks (RNNs)** and **Long Short-Term Memory (LSTM)** networks have been widely used for sequence modeling. However, these models often struggle with long-term dependencies and lack efficient parallelism, which limits their scalability and performance in complex tasks.

Transformers overcome these limitations by introducing a **self-attention mechanism**, which enables the model to attend to all positions in the sequence simultaneously. This approach improves the model’s ability to capture long-range dependencies and supports efficient parallel computation during training.

Characteristics of Sequential Data

Sequential data consists of data points arranged in a specific order, where each element depends on the preceding elements. Is defined by two key characteristics:

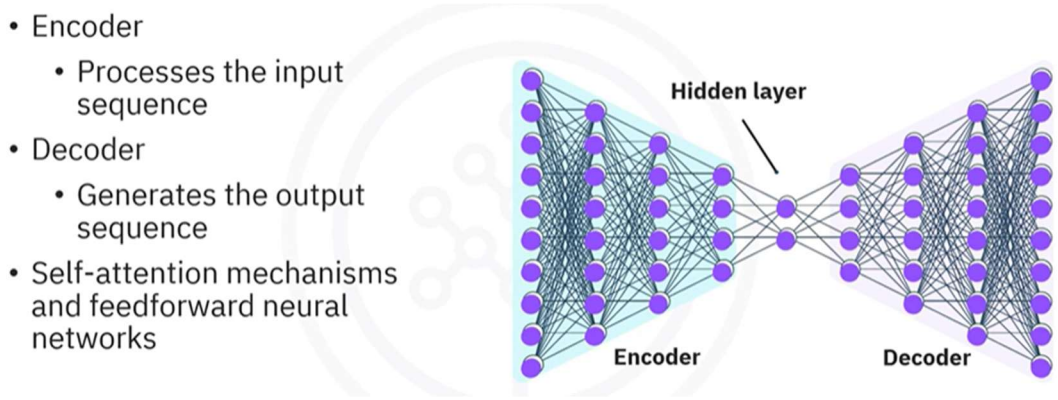
1. **Order**: The position of each element in the sequence affects its meaning.
2. **Dependency**: Each element is influenced by previous elements.

Examples of sequential data include:

- Natural language sentences, where the interpretation of each word depends on its surrounding context.
- Time series datasets, where each data point depends on historical values.

Models handling this data must preserve these properties while learning meaningful representations.

Transformer Architecture for Sequential Modeling



A standard transformer model for sequential data is composed of two core components:

- **Encoder**: Processes the input sequence.
- **Decoder**: Generates the output sequence based on the encoder's processed information.

Both encoder and decoder components use:

- **Multi-head self-attention mechanisms** to process all tokens in parallel and capture relationships across the sequence.
- **Feed-forward neural networks** to transform attention outputs and introduce non-linearity.
- **Residual connections and layer normalization** to maintain stable gradient flow and accelerate training.

⚙️ Core Transformer Components in Keras

🧩 Multi-Head Self-Attention

The **multi-head self-attention mechanism** allows the model to project the input sequence into multiple subspaces and perform attention in each subspace separately. This enhances the model's capacity to represent different aspects of the sequence.

The key components in the implementation include:

- **attention method:**
  Computes the attention scores and applies them to the value vectors.
- **split_heads method:**
  Divides the input into multiple heads for parallel attention computation.
- **call method:**
  Applies self-attention to the input and combines the results from all heads into a single tensor.

In the example the **multi-head self-attention class** defines the multi-head self-attention mechanism.

```
import tensorflow as tf
from tensorflow.keras.layers import Layer, Dense, LayerNormalization, Dropout
class MultiHeadSelfAttention(Layer):
    def __init__(self, embed_dim, num_heads=8):
        super(MultiHeadSelfAttention, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.projection_dim = embed_dim // num_heads
        self.query_dense = Dense(embed_dim)
        self.key_dense = Dense(embed_dim)
        self.value_dense = Dense(embed_dim)
        self.combine_heads = Dense(embed_dim)

    def attention(self, query, key, value):
            score = tf.matmul(query, key, transpose_b=True)
            dim_key = tf.cast(tf.shape(key)[-1], tf.float32)
            scaled_score = score / tf.math.sqrt(dim_key)
            weights = tf.nn.softmax(scaled_score, axis=-1)
            output = tf.matmul(weights, value)
            return output, weights

    def split_heads(self, x, batch_size):
        x = tf.reshape(x, (batch_size, -1, self.num_heads,
            self.projection_dim))
        return tf.transpose(x, perm=[0, 2, 1, 3])
```

```
    def call(self, inputs):
        batch_size = tf.shape(inputs)[0]
        query = self.query_dense(inputs)
        key = self.key_dense(inputs)
        value = self.value_dense(inputs)
        query = self.split_heads(query, batch_size)
        key = self.split_heads(key, batch_size)
        value = self.split_heads(value, batch_size)
        attention, _ = self.attention(query, key, value)
        attention = tf.transpose(attention, perm=[0, 2, 1, 3])
        concat_attention = tf.reshape(attention, (batch_size, -1,
self.embed_dim))
        output = self.combine_heads(concat_attention)
        return output
```

🧩 Transformer Block

The transformer block integrates the multi-head self-attention with a feed-forward network.

The call method executes the following steps:

1. Applies multi-head self-attention to the input.
2. Adds a residual connection and layer normalization.
3. Passes the result through a feed-forward network.
4. Adds another residual connection and layer normalization.

This structure enables the model to refine sequence representations layer by layer.

```
import tensorflow as tf
from tensorflow.keras.layers import Layer, Dense, LayerNormalization, Dropout

class TransformerBlock(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = MultiHeadSelfAttention(embed_dim, num_heads)
        self.ffn = tf.keras.Sequential([
            Dense(ff_dim, activation="relu"),
            Dense(embed_dim),
        ])
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)

    def call(self, inputs, training):
        attn_output = self.att(inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)
```

```
# Example usage
embed_dim = 128
num_heads = 8
ff_dim = 512
sequence_length = 100
transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
inputs = tf.random.uniform((1, sequence_length, embed_dim))
outputs = transformer_block(inputs)
print(outputs.shape)  # Should print (1, 100, 128)
```

🧩 Encoder Layer

Each encoder layer contains:

- A **multi-head self-attention mechanism** that allows each token to attend to all other tokens in the input.
- A **feed-forward neural network** that further transforms the attention outputs.
- **Residual connections** and **layer normalization** applied after each sub-layer.

The implementation of the encoder layer in Keras follows this sequence:

1. Receives the input sequence.
2. Applies multi-head self-attention.
3. Adds a residual connection and applies layer normalization.
4. Applies a feed-forward network.
5. Adds another residual connection and applies layer normalization.

Multiple encoder layers are stacked to deepen the model and increase its representation power.

```
import tensorflow as tf
from tensorflow.keras.layers import Layer, Dense, LayerNormalization, Dropout

class EncoderLayer(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(EncoderLayer, self).__init__()
        self.att = MultiHeadSelfAttention(embed_dim, num_heads)
        self.ffn = tf.keras.Sequential([
            Dense(ff_dim, activation="relu"),
            Dense(embed_dim),
        ])
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)

    def call(self, inputs, training):
            attn_output = self.att(inputs)
            attn_output = self.dropout1(attn_output, training=training)
            out1 = self.layernorm1(inputs + attn_output)
            ffn_output = self.ffn(out1)
            ffn_output = self.dropout2(ffn_output, training=training)
            return self.layernorm2(out1 + ffn_output)
```

```
# Example usage
encoder_layer = EncoderLayer(embed_dim, num_heads, ff_dim)
outputs = encoder_layer(inputs)
print(outputs.shape)  # Should print (1, 100, 128)
```

☑Takeaways

☑ Sequential data requires models that account for order and dependency between elements.

☑ Traditional models like RNNs and LSTMs struggle with long-range dependencies and parallel training limitations.

☑ Transformers address these challenges by using self-attention to process all tokens in a sequence simultaneously.

☑ A standard transformer includes an encoder and a decoder, both composed of multi-head self-attention layers, feed-forward networks, residual connections, and layer normalization.

☑ Implementing transformer models in Keras involves defining multi-head attention, transformer blocks, and encoder layers in a modular and structured way.

M3 – Section 2
Advanced Transformers and Sequential Data using TensorFlow

📌Advanced Transformer Applications

Transformers, originally developed for natural language processing, have evolved into a general-purpose architecture suitable for a wide variety of domains.

Their ability to model sequences using self-attention mechanisms makes them highly effective in computer vision, speech recognition, and reinforcement learning.

📸Transformers in Computer Vision

Transformers can be applied to images by treating them as sequences of patches.

Vision Transformers (ViTs) divide an image into smaller patches and treat each patch similarly to a word in a sentence. This enables the use of self-attention mechanisms on image data.

Key Steps:

1. Divide the input image into fixed-size patches.
2. Pass patches through a **patch embedding** layer to project them into vector representations.
3. Use a **transformer block** consisting of:
    - Multi-head self-attention mechanism
    - Feed-forward network
    - Residual connections and layer normalization
4. Stack multiple transformer blocks to form the **Vision Transformer model**.

Components:

🔧 **TransformerBlock:**
Applies self-attention and feed-forward layers with residual and normalization steps.

```
import tensorflow as tf
from tensorflow.keras.layers import Layer, Dense, LayerNormalization, Dropout
# Define the TransformerBlock class
class TransformerBlock(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = tf.keras.layers.MultiHeadAttention(num_heads=num_heads,
        key_dim=embed_dim)
        self.ffn = tf.keras.Sequential([
        Dense(ff_dim, activation="relu"),
        Dense(embed_dim),
        ])
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)

    def call(self, inputs, training, mask=None):
        attn_output = self.att(inputs, inputs, inputs, attention_mask=mask)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)
```

🔧**PatchEmbedding:**
Defines how image patches are converted into embedded vectors.

```
# Define the PatchEmbedding layer
class PatchEmbedding(Layer):
    def __init__(self, num_patches, embedding_dim):
        super(PatchEmbedding, self).__init__()
        self.num_patches = num_patches
        self.embedding_dim = embedding_dim
        self.projection = Dense(embedding_dim)

    def call(self, patches):
        return self.projection(patches)
```

🔧**VisionTransformer:**
Combines patch embedding and stacked transformer blocks for image classification.

```
# Define the VisionTransformer model
class VisionTransformer(Model):
    def __init__(self, num_patches, embedding_dim, num_heads, ff_dim,
num_layers, num_classes):
        super(VisionTransformer, self).__init__()
        self.patch_embed = PatchEmbedding(num_patches, embedding_dim)
        self.transformer_layers = [
           TransformerBlock(
              embedding_dim,
              num_heads,
              ff_dim
              )for _ in range(num_layers)
           ]
        self.flatten = Flatten()
        self.dense = Dense(num_classes, activation='softmax')

    def call(self, images, training = False ):
        patches = self.extract_patches(images)
        x = self.patch_embed(patches)
        for transformer_layer in self.transformer_layers:
            x = transformer_layer(x, training=training)
        x = self.flatten(x)
        return self.dense(x)

    def extract_patches(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, 16, 16, 1],
            strides=[1, 16, 16, 1],
            rates=[1, 1, 1, 1],
            padding='VALID'
        )
        patches = tf.reshape(patches, [batch_size, -1, 16*16*3])
        return patches
```

⚙️The example usage demonstrates how to create and use the vision transformer model for image classification.

```
# Example usage
num_patches = 196  # Assuming 14x14 patches
embedding_dim = 128
num_heads = 4
ff_dim = 512
num_layers = 6
num_classes = 10  # For CIFAR-10 dataset

vit = VisionTransformer(num_patches, embedding_dim, num_heads, ff_dim,
num_layers, num_classes)
images = tf.random.uniform((32, 224, 224, 3))  # Batch of 32 images of size
224x224
output = vit(images)
print(output.shape)  # Should print (32, 10)
```

📢Transformers in Speech Recognition

Transformers can process speech by first converting audio signals into **spectrograms**, which are 2D representations of sound over time. These spectrograms are then treated as sequences, allowing transformers to model the temporal structure of speech.

Key Steps:

1. Convert audio signals into spectrograms.
2. Define a **SpeechTransformer** model that includes:
   - **Convolutional layers** to capture local patterns
   - **Transformer blocks** to model long-term dependencies
3. Apply self-attention and feed-forward networks in each transformer layer.
4. Use residual connections and layer normalization for stable training.

This approach has led to models like **Wav2Vec** and **SpeechTransformer**, which perform competitively in speech-to-text tasks.

Components:

🔧 **TransformerBlock:**
Implements self-attention and feed-forward sub-layers. Includes residual connections and normalization.

```
import tensorflow as tf
from tensorflow.keras.layers import Layer, Dense, LayerNormalization, Dropout,
Flatten
from tensorflow.keras.models import Model

# Define the TransformerBlock within the same cell
class TransformerBlock(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = tf.keras.layers.MultiHeadAttention(num_heads=num_heads,
key_dim=embed_dim)
        self.ffn = tf.keras.Sequential([
            Dense(ff_dim, activation="relu"),
            Dense(embed_dim),
        ])
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)

    def call(self, inputs, training, mask=None):
        attn_output = self.att(inputs, inputs, inputs, attention_mask=mask)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)
```

🔧 **PatchEmbedding:**
Defines a layer that embeds self-projected patches.
Although patch embedding is typically for vision, it's conceptually similar in audio processing, where segments of the spectrogram may be projected before attention layers.

```
   # Define the PatchEmbedding layer
class PatchEmbedding(Layer):
    def __init__(self, num_patches, embedding_dim):
        super(PatchEmbedding, self).__init__()
        self.num_patches = num_patches
        self.embedding_dim = embedding_dim
        self.projection = Dense(embedding_dim)

    def call(self, patches):
        return self.projection(patches)
```

🔧 **SpeechTransformer:**
Defines the overall architecture using convolutional and transformer layers to process sequential speech data.

```
# Define the SpeechTransformer model
class SpeechTransformer(Model):
    def __init__(self, num_mel_bins, embedding_dim, num_heads, ff_dim,
num_layers, num_classes):
        super(SpeechTransformer, self).__init__()
        self.conv1 = Conv1D(filters=embedding_dim, kernel_size=3,
strides=1, padding='same', activation='relu')
        self.batch_norm = BatchNormalization()
        self.reshape = Reshape((-1, embedding_dim))
        self.transformer_layers = [TransformerBlock(embedding_dim,
num_heads, ff_dim) for _ in range(num_layers)]
        self.flatten = Flatten()
        self.dense = Dense(num_classes, activation='softmax')

    def call(self, spectrograms):
        x = self.conv1(spectrograms)
        x = self.batch_norm(x)
        x = self.reshape(x)
        for transformer_layer in self.transformer_layers:
            x = transformer_layer(x)
        x = self.flatten(x)
        return self.dense(x)
```

⚙️ The example usage demonstrates how to create and use speech transformer model for speech recognition.

```
# Example usage
num_mel_bins = 80
embedding_dim = 128
num_heads = 4
ff_dim = 512
num_layers = 6
num_classes = 30  # Example for phoneme classification

# Initialize SpeechTransformer model
st = SpeechTransformer(num_mel_bins, embedding_dim, num_heads, ff_dim,
num_layers, num_classes)

# Generate example spectrograms
spectrograms = tf.random.uniform((32, 100, num_mel_bins))  # Batch of 32
spectrograms with 100 time frames

# Get model predictions
output = st(spectrograms, training=True)
print(output.shape)  # Should print (32, 30) for batch size 32 and 30
classes
```

🎮 Transformers in Reinforcement Learning

Transformers are also used in **reinforcement learning** to model sequences of states, actions, and rewards. This allows them to capture complex dependencies in long trajectories and learn patterns from past experience.

The **Decision Transformer** is a transformer-based model designed to predict the next action based on historical context, rather than using value-based or policy-based objectives. It processes entire trajectories as input sequences and learns directly from return-conditioned sequences.

Key Steps:

1. Embed the sequence of past states, actions, and returns.
2. Use a transformer block to model the dependencies across time steps.
3. Output the predicted action based on context from the full trajectory.

Components:

🔧 **TransformerBlock:**
Defines the building block of the model, including self-attention and feed-forward sub-layers.
The call() method handles the computation of attention outputs and their transformation through the feed-forward layer, followed by residual and normalization layers.

```
from tensorflow.keras.layers import Dense, TimeDistributed
# Define the TransformerBlock class
class TransformerBlock(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = tf.keras.layers.MultiHeadAttention(num_heads=num_heads,
key_dim=embed_dim)
        self.ffn = tf.keras.Sequential([
            Dense(ff_dim, activation="relu"),
            Dense(embed_dim),
        ])
        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.dropout2 = tf.keras.layers.Dropout(rate)

    def call(self, inputs, training, mask=None):
        attn_output = self.att(inputs, inputs, attention_mask=mask)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)

        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)

        return self.layernorm2(out1 + ffn_output)
```

🔧 **DecisionTransformer:**

Implements the full transformer model tailored for RL.

The call() method applies embeddings for sequences of returns, states, and actions, then feeds them through transformer layers to generate action predictions.

Applies embedding logic and transformer computations to process entire trajectories and produce predictions.

```
# Define the DecisionTransformer model
class DecisionTransformer(tf.keras.Model):
    def __init__(self, state_dim, action_dim, embedding_dim, num_heads,
ff_dim, num_layers):
        super(DecisionTransformer, self).__init__()
        self.state_embed = Dense(embedding_dim, activation='relu')
        self.action_embed = Dense(embedding_dim, activation='relu')
        self.transformer_layers = [TransformerBlock(embedding_dim,
num_heads, ff_dim) for _ in range(num_layers)]
        self.dense = TimeDistributed(Dense(action_dim))
```

```
    def call(self, states, actions):
        state_embeddings = self.state_embed(states)
        action_embeddings = self.action_embed(actions)
        x = state_embeddings + action_embeddings
        for transformer_layer in self.transformer_layers:
            x = transformer_layer(x)
        return self.dense(x)
```

This architecture allows transformers to operate directly on RL trajectories, offering a unified modeling approach for decision making across sequential environments. By conditioning on past rewards and actions, decision transformers enable **policy learning without traditional reinforcement learning objectives**.

⚙️The example usage demonstrates how to instantiate the **DecisionTransformer** model and use it for predicting the next action in an environment using historical state-action-reward data..

```
# Example usage
state_dim = 20
action_dim = 5
embedding_dim = 128
num_heads = 4
ff_dim = 512
num_layers = 6

# Initialize DecisionTransformer model
dt = DecisionTransformer(state_dim, action_dim, embedding_dim, num_heads,
ff_dim, num_layers)

# Generate example states and actions
states = tf.random.uniform((32, 100, state_dim))
# Batch of 32 sequences of 100 states
actions = tf.random.uniform((32, 100, action_dim))
 # Batch of 32 sequences of 100 actions

# Get model predictions
output = dt(states, actions, training=True)

print(output.shape)
# Should print (32, 100, 5) for batch size 32, sequence length 100, and
action dimension 5
```

☑Takeaways

☑ Transformers are not limited to text; they are highly adaptable and effective in image, audio, and control-based tasks.

☑ Vision Transformers use self-attention on image patches for classification.

☑ Speech Transformers process spectrograms to model audio data sequentially.

☑ Decision Transformers use transformer architecture to model trajectories for decision making in reinforcement learning.

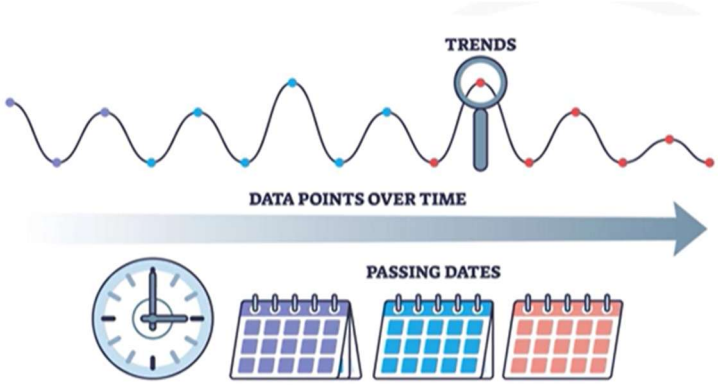
☑ Each application uses the same core principles: self-attention, feed-forward networks, residual connections, and normalization — but adapts the input structure to the specific domain.

📌 Transformers for Time Series Prediction

Transformers can be applied to time series forecasting tasks by modeling sequences of numerical data collected at regular intervals.

These models offer an efficient and scalable alternative to traditional forecasting methods.

🔹 Time Series Forecasting Overview



Time series data is a sequence of values recorded over time — for example, stock prices, weather measurements, or sensor readings.
Forecasting involves predicting the next value(s) in the sequence based on historical data.

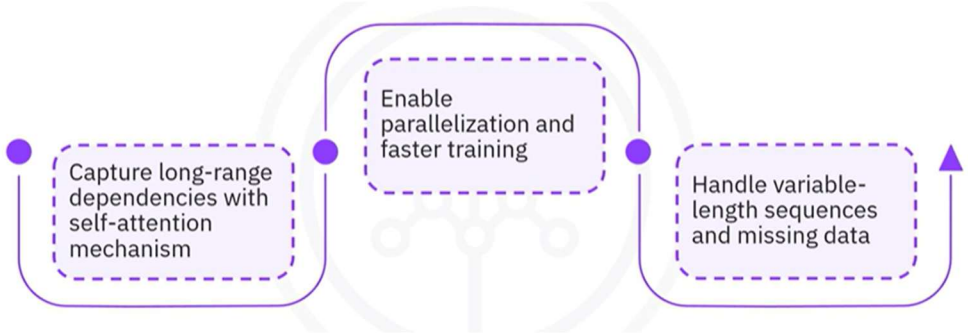
Traditional models include:

- **ARIMA** (AutoRegressive Integrated Moving Average)
- **RNNs** (Recurrent Neural Networks)
- **LSTMs** (Long Short-Term Memory networks)

While effective, these models have limitations in capturing long-term dependencies and parallelizing computation.

Transformers address these limitations using **self-attention**, which enables the model to consider the entire sequence at once and efficiently capture dependencies over long time ranges.

🔹 Key Advantages of Transformers for Time Series

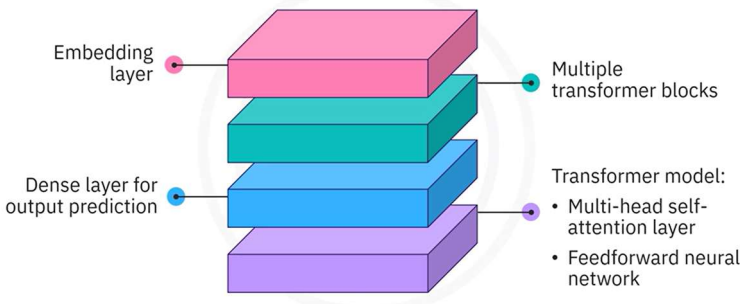


- **Self-attention mechanism** models long-range relationships more effectively than recurrent models.
- **Parallel computation** speeds up training by processing entire sequences simultaneously.
- **Flexible sequence handling** allows working with variable-length inputs and incomplete data.

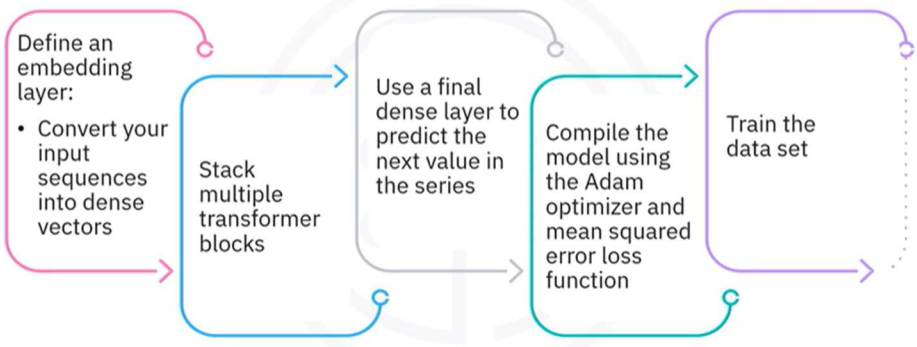
These strengths make transformers especially useful in time series forecasting applications where dependencies across many time steps matter.

🔹 Building a Transformer for Time Series with Keras

To build a time series forecasting model using Keras, follow this step-by-step structure:



🔁 Key Steps:



1. Normalize the dataset and convert it into input sequences and next-step labels.
2. Define an embedding layer to transform input sequences into dense vectors.
3. Stack multiple transformer blocks, each containing self-attention and feed-forward layers.
4. Add a final dense layer to output the predicted next value in the sequence.
5. Compile the model using the Adam optimizer and mean squared error loss.
6. Train the model using the prepared time series data.

🧩 Preparing Time Series Data

1. Normalize the data using a **MinMaxScaler** to map values between 0 and 1.
2. Split the normalized series into input sequences and corresponding labels.
   - Each input contains a fixed number of past time steps.
   - The label is the next value in the sequence.
3. Print shapes and check for missing values to validate the preparation}

```
# Load the data set
data = pd.read_csv('/content/stock_prices.csv')
data = data[['Close']].values
# Normalize the data
scaler = MinMaxScaler(feature_range=(0, 1))
data = scaler.fit_transform(data)
# Prepare the data for training
def create_dataset(data, time_step=1):
    X, Y = [], []
    for i in range(len(data) - time_step - 1):
        a = data[i:(i + time_step), 0]  # slice a sequence from data
        X.append(a)  # append the sequence to X
        Y.append(data[i + time_step, 0])  # append the next value to Y
    return np.array(X), np.array(Y)

time_step = 60
X, Y = create_dataset(data, time_step)
```

Model Components

TransformerBlock:

Defines a transformer layer that includes, Multi-head self-attention layer, Feed-forward network, Residual connections and layer normalization.

The call() method applies these operations in the correct order.

```
import tensorflow as tf
from tensorflow.keras.layers import Layer, Dense, LayerNormalization,
Dropout

class TransformerBlock(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = tf.keras.layers.MultiHeadAttention(num_heads=num_heads,
key_dim=embed_dim)
        self.ffn = tf.keras.Sequential([
            Dense(ff_dim, activation="relu"),
            Dense(embed_dim),
        ])
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)


    def call(self, inputs, training, mask=None):
        attn_output = self.att(inputs, inputs, inputs, attention_mask=mask)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)
```

Embedding Layer:

Converts input sequences (time steps) into dense vectors that can be processed by the transformer.

Dense Layer:

Produces the final prediction — the next value in the time series.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Flatten, Embedding
# Define the Transformer model
input_shape = (X.shape[1], X.shape[2])
inputs = Input(shape=input_shape)
# Embedding layer
x = Dense(128)(inputs)
# Transformer blocks
for _ in range(4):
    x = TransformerBlock(embed_dim=128, num_heads=4, ff_dim=512)(x)
# Output layer
x = Flatten()(x)
outputs = Dense(1)(x)
# Create the model
model = Model(inputs, outputs)
```

🧩 Training and Evaluation

After preparing the data and defining the model, train and evaluate the forecasting model:

1. Train the model using the input sequences and next-step labels.
2. Use the model to predict future values.
3. Inverse transform the predictions back to the original scale.
4. Plot both the actual values and predicted values to visualize performance.

```
# Compile the model
model.compile(optimizer='adam', loss='mse')
# Train the model
model.fit(X, Y, epochs=20, batch_size=32)
# Make predictions
predictions = model.predict(X)
# Inverse transform the predictions to get the original scale
predictions = scaler.inverse_transform(predictions)
# Plot the predictions
plt.plot(scaler.inverse_transform(data), label='True Data')
plt.plot(np.arange(time_step, time_step + len(predictions)), predictions,
label='Predictions')
plt.xlabel('Time')
plt.ylabel('Stock Prices')
plt.legend()
plt.show()
```

☑Takeaways

☑ Transformers can be applied to time series forecasting by modeling sequences of past values.

☑ Self-attention enables transformers to capture long-range dependencies more effectively than RNNs or LSTMs.

☑ The model architecture includes:

- An **embedding layer** for input transformation
- One or more **transformer blocks** to process the sequence
- A **dense output layer** for prediction

☑ Keras provides a flexible framework for building, training, and evaluating such models end-to-end.

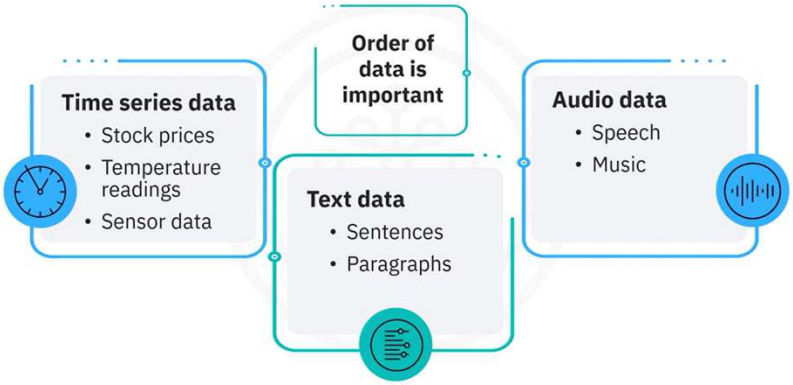
📌 TensorFlow for Sequential Data

TensorFlow provides a rich set of tools for building and training models on **sequential data**.

This includes time series, text, and audio — all of which require preserving the order of data points and capturing dependencies across time or position.

🔹 Understanding Sequential Data

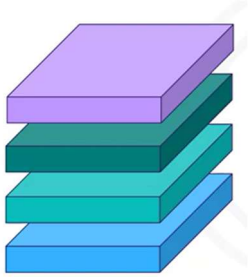
Sequential data is defined by the **order and structure of its elements**. Each point depends on those that come before it.



Modeling this type of data requires architectures that can **understand and learn from sequences**, recognizing both **short-term** and **long-term dependencies**.

🔹 TensorFlow Tools for Sequence Modeling

TensorFlow includes several layers and components specifically designed for sequential data:



- **RNN (Recurrent Neural Network)** layers
- **LSTM (Long Short-Term Memory)** layers
- **GRU (Gated Recurrent Unit)** layers
- **Conv1D (1D Convolution)** layers for sequential pattern detection
- **TextVectorization** for transforming text into numeric tensors

These components allow TensorFlow to be used in a wide range of sequence-based tasks including:

- **Time series forecasting**
- **Natural language processing**
- **Speech recognition**

🔹 Building Sequence Models with TensorFlow

```
# Generate some example sequential data
import numpy as np
# Create a simple sine wave dataset
def create_sine_wave_dataset(seq_length=100):
    x = np.linspace(0, 50, seq_length)
    y = np.sin(x)
    return y
data = create_sine_wave_dataset()
time_steps = np.arange(len(data))
```

🛠️ RNN for Time Series Prediction

1. Create input sequences and matching labels.
2. Build a model using:
   - A **SimpleRNN** layer
   - A **Dense** layer for output prediction
3. Compile and train the model.
4. Make predictions and plot the results to evaluate performance.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
# Prepare the dataset
def prepare_data(data, time_steps, time_window):
    X, Y = [], []
    for i in range(len(data) - time_window):
        X.append(data[i:i + time_window])
        Y.append(data[i + time_window])
    return np.array(X), np.array(Y)
time_window = 10
X, Y = prepare_data(data, time_steps, time_window)

# Reshape the data to match RNN input shape
X = X.reshape((X.shape[0], X.shape[1], 1))
# Build the RNN model
model = Sequential([
    SimpleRNN(50, activation='relu', input_shape=(time_window, 1)),
    Dense(1)
])
# Compile the model
model.compile(optimizer='adam', loss='mse')
# Train the model
model.fit(X, Y, epochs=20, batch_size=16)
# Make predictions
predictions = model.predict(X)
```

```
# Plot the results
import matplotlib.pyplot as plt
plt.plot(time_steps, data, label='True Data')
plt.plot(time_steps[time_window:], predictions, label='Predictions')
plt.xlabel('Time Steps')
plt.ylabel('Value')
plt.legend()
plt.show()
```

🛠️ LSTM for Long-Term Dependency Modeling

LSTMs are a type of RNN that are capable of learning long-term dependencies, making them suitable for sequential data with long-term patterns.

Using the same sine wave dataset generated before, the steps for LSTM modeling are as follows:

1. Replace the **SimpleRNN** layer with an **LSTM** layer.
2. Keep the rest of the model structure the same.
3. Compile and train the model.
4. Generate predictions and compare them with the original data to see improvements in learning longer patterns.

```
from tensorflow.keras.layers import LSTM
# Build the LSTM model
lstm_model = Sequential([
    LSTM(50, activation='relu', input_shape=(time_window, 1)),
    Dense(1)
])
# Compile the model
lstm_model.compile(optimizer='adam', loss='mse')
# Train the model
lstm_model.fit(X, Y, epochs=20, batch_size=16)
# Make predictions
lstm_predictions = lstm_model.predict(X)
# Plot the results
plt.plot(time_steps, data, label='True Data')
plt.plot(time_steps[time_window:], lstm_predictions, label='LSTM
Predictions')
plt.xlabel('Time Steps')
plt.ylabel('Value')
plt.legend()
plt.show()
```

🛠️ Handling text data with TensorFlow

Text data requires specific pre-processing steps such as tokenization and padding.

TensorFlow's text vectorization layer helps in converting text data into numerical format suitable for model training.

1. Create a **TextVectorization** layer.
   - Tokenizes text
   - Pads sequences to ensure consistent input shape
2. Adapt the vectorizer to the dataset.
3. Transform the text into numerical sequences suitable for model input.

```
from tensorflow.keras.layers import TextVectorization
# Sample text data
texts = [
    "Hello, how are you?",
    "I am fine, thank you.",
    "How about you?",
    "I am good too."
]
# Define the TextVectorization layer
vectorizer = TextVectorization(output_mode='int', max_tokens=100,
output_sequence_length=10)

# Adapt the vectorizer to the text data
vectorizer.adapt(texts)
# Vectorize the text data
text_vectorized = vectorizer(texts)
print("Vectorized text data:\n", text_vectorized.numpy())
```

☑Takeaways

☑ Sequential data depends on the order of data points, making it important to use models that can retain and learn from temporal structure.

☑ TensorFlow supports a variety of layers for sequence modeling including RNNs, LSTMs, GRUs, and Conv1D.

☑ Time series data can be modeled using simple RNN or LSTM layers to predict future values based on past inputs.

☑ Text data requires preprocessing through tokenization and padding before being passed to a model. TensorFlow's TextVectorization layer simplifies this step.

☑ TensorFlow provides all necessary components to build, train, and evaluate models on structured sequential data.

Module 4

Unsupervised Learning and Generative Models in Keras

M4 – Section 1
Unsupervised Learning, Autoencoders, and Difussion Models

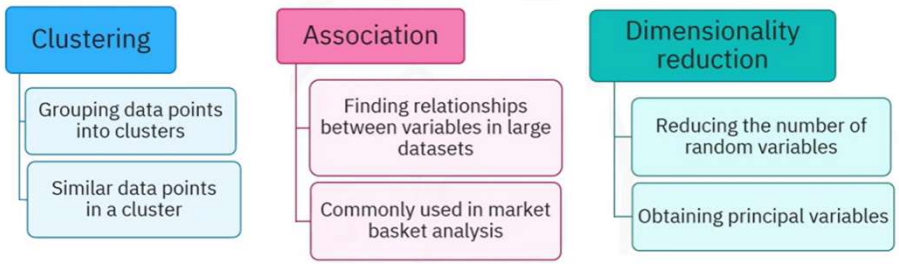
📌 Intro to Unsupervised Learning in Keras

Unsupervised learning is a machine learning approach where models are trained to identify patterns, structures, or relationships in data **without labeled outputs or target variables**.

The primary goal is to **discover meaningful insights** from data without predefined categories or supervision.

🔹 Categories of Unsupervised Learning

Unsupervised learning techniques fall into three broad categories:



🔧 Clustering:

Clustering methods group data points based on similarity.

The model forms clusters such that items in the same cluster are more similar to each other than to those in different clusters.

Examples:

- K-Means Clustering
- Hierarchical Clustering

🔧 Association:

Association rule learning identifies patterns, correlations, and relationships between variables in large datasets.

Often used in **market basket analysis** to detect co-occurring products.

Examples:

- Apriori Algorithm
- Eclat Algorithm

🔧 **Dimensionality Reduction:**

Dimensionality reduction techniques reduce the number of variables in a dataset while preserving its essential structure.

Helps simplify data and remove redundancy.

Examples:

- Principal Component Analysis (PCA)
- T-distributed Stochastic Neighbor Embedding (t-SNE)

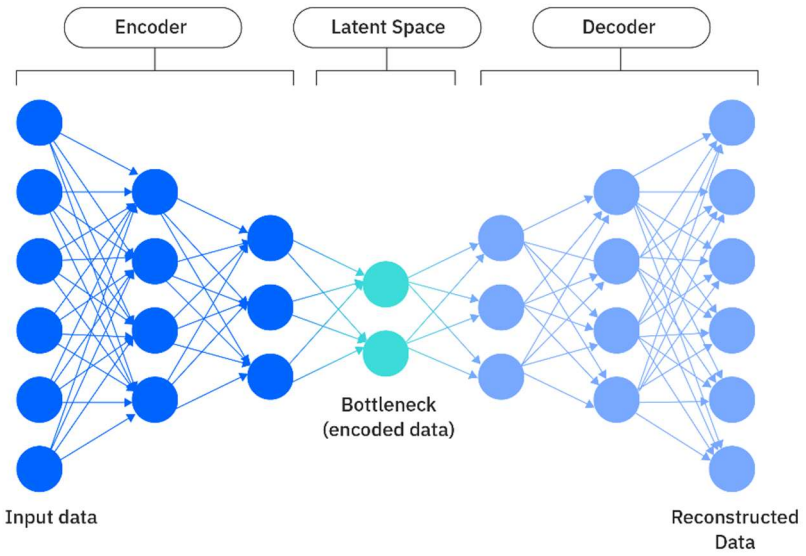
🔹 Autoencoders

Autoencoders are neural networks designed for **learning efficient data representations**, often used for dimensionality reduction and feature learning.

The autoencoder is trained to **minimize the difference** between the input and reconstructed output, forcing the network to learn meaningful representations of the data.

Key Components:

- **Encoder**: Compresses input data into a lower-dimensional **latent space**.
- **Decoder**: Reconstructs the original input from the latent representation.



Simple implementation of an autoencoder in Keras:

In this code, an autoencoder for a data set with 784 features is defined.

The encoder compresses the input into 64 features, and the decoder reconstructs the original 784 features.

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
# Define the encoder
input_layer = Input(shape=(784,))
encoded = Dense(64, activation='relu')(input_layer)
# Define the decoder
decoded = Dense(784, activation='sigmoid')(encoded)
# Combine the encoder and decoder into an autoencoder model
autoencoder = Model(input_layer, decoded)
# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
# Summary of the model
autoencoder.summary()
```

To train the autoencoder, you use the same data for both input and output.

For example, you can use the Modified National Institute of Standards and Technology, MNIST data set as shown in this code.

This code trains the autoencoder to learn efficient representations of the MNIST digits.

```
# Load the MNIST dataset
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()

# Normalize the data
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), 784))
x_test = x_test.reshape((len(x_test), 784))

# Train the autoencoder
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

🔍 Difference Between Autoencoders and Transformers

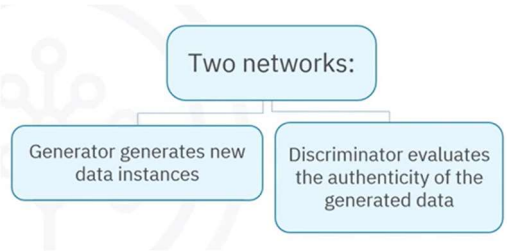
Feature	Autoencoders	Transformers
Purpose	Learn compact, efficient representations of input data for reconstruction, denoising, or compression.	Model dependencies across sequences for tasks like translation, classification, generation, or forecasting.
Input/Output Structure	Input and output are usually identical (or closely related); used for reconstructing the input.	Input and output may differ (e.g., source and target languages, time series to predictions).
Architecture	Composed of two parts: an encoder that compresses input into latent space and a decoder that reconstructs the original input.	Uses self-attention mechanisms in encoder-decoder or encoder-only stacks to process entire sequences.
Attention Mechanism	Typically absent (uses dense or convolutional layers); focuses on representation learning.	Central to the architecture; enables global context modeling across entire input.
Use Cases	Dimensionality reduction, anomaly detection, denoising, pretraining.	NLP (e.g., BERT, GPT), vision (ViT), time series forecasting, sequence-to-sequence tasks.
Training Objective	Minimize the difference between input and reconstruction.	Minimize prediction loss (e.g., cross-entropy, MSE) for output targets.
Purpose	Learn compact, efficient representations of input data for reconstruction, denoising, or compression.	Model dependencies across sequences for tasks like translation, classification, generation, or forecasting.
Input/Output Structure	Input and output are usually identical (or closely related); used for reconstructing the input.	Input and output may differ (e.g., source and target languages, time series to predictions).
Architecture	Composed of two parts: an encoder that compresses input into latent space and a decoder that reconstructs the original input.	Uses self-attention mechanisms in encoder-decoder or encoder-only stacks to process entire sequences.
Attention Mechanism	Typically absent (uses dense or convolutional layers); focuses on representation learning.	Central to the architecture; enables global context modeling across entire input.
Use Cases	Dimensionality reduction, anomaly detection, denoising, pretraining.	NLP (e.g., BERT, GPT), vision (ViT), time series forecasting, sequence-to-sequence tasks.
Training Objective	Minimize the difference between input and reconstruction.	Minimize prediction loss (e.g., cross-entropy, MSE) for output targets.

Generative Adversarial Networks (GANs)

GANs are generative models introduced by **Ian Goodfellow in 2014**, consisting of two competing networks:

Generator:

- Takes a random noise vector (e.g., 100 dimensions) as input.
- Produces synthetic data (e.g., a 784-dimensional image).



Discriminator:

- Receives real or generated data.
- Classifies input as either **real** or **fake**.

The **generator tries to fool the discriminator** while the discriminator tries to distinguish between real and fake data. This adversarial process leads to the generator producing increasingly realistic data.

Simple implementation of GAN in Keras:

In this code, simple GAN is defined, where the generator takes a 100-dimensional noise vector as input, and produces a 784-dimensional image.

The discriminator evaluates whether the image is real or generated.

```
from tensorflow.keras.layers import LeakyReLU
import numpy as np

# Define the generator model
def build_generator():
    model = tf.keras.Sequential()
    model.add(Dense(128, input_dim=100))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(784, activation='tanh'))
    return model

# Define the discriminator model
def build_discriminator():
    model = tf.keras.Sequential()
    model.add(Dense(128, input_shape=(784,)))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(1, activation='sigmoid'))
    return model
# Build and compile the discriminator
discriminator = build_discriminator()
discriminator.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Build the generator
generator = build_generator()
```

```
# Create the GAN by combining the generator and discriminator
discriminator.trainable = False
gan_input = Input(shape=(100,))
generated_image = generator(gan_input)
gan_output = discriminator(generated_image)
gan = Model(gan_input, gan_output)

# Compile the GAN
gan.compile(optimizer='adam', loss='binary_crossentropy')
```

Training a GAN involves training the discriminator and generator alternately.

Here's a simplified version of the training loop.

This loop trains the discriminator to distinguish real images from generated ones and trains the generator to produce realistic images.

```
def train_gan(gan, generator, discriminator, x_train, epochs=400,
batch_size=128):
    # Loop through epochs
    for epoch in range(epochs):
        # Generate random noise as input for the generator
        noise = np.random.normal(0, 1, (batch_size, 100))
        generated_images = generator.predict(noise)

        # Get a random set of real images
        idx = np.random.randint(0, x_train.shape[0], batch_size)
        real_images = x_train[idx]

        # Labels for real and fake images
        real_labels = np.ones((batch_size, 1))
        fake_labels = np.zeros((batch_size, 1))

        # Train the discriminator on real and fake images separately
        d_loss_real = discriminator.train_on_batch(real_images, real_labels)
        d_loss_fake = discriminator.train_on_batch(generated_images, fake_labels)

        # Calculate the average loss for the discriminator
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Generate new noise and train the generator through the GAN model (note:
we train the generator via the GAN model, where the discriminator's weights are
frozen)
        noise = np.random.normal(0, 1, (batch_size, 100))
        g_loss = gan.train_on_batch(noise, real_labels, return_dict=True)

        # Print the progress every 10 epochs
        if epoch % 10 == 0:
            print(f"Epoch {epoch} - Discriminator Loss: {d_loss[0]}, Generator
Loss: {g_loss['loss']}")

    return d_loss, g_loss
```

☑ Takeaways

☑ Unsupervised learning is used to discover patterns in data without labeled outputs or predefined targets.

☑ The three main categories of unsupervised learning are, **Clustering**, **Association**, **Dimensionality Reduction**.

☑ Autoencoders are neural networks that compress input data into a latent representation and then reconstruct it, learning efficient features in the process.

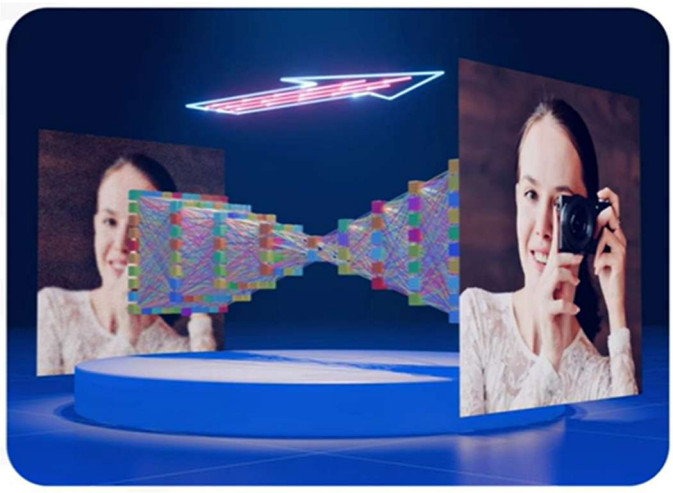
☑ Generative Adversarial Networks (GANs) consist of a generator that produces synthetic data and a discriminator that evaluates its authenticity. They are trained through adversarial competition to generate increasingly realistic outputs.

☑ Both autoencoders and GANs are powerful tools for representation learning and data generation, foundational in unsupervised deep learning with Keras.

📌 Building Autoencoders in Keras

Autoencoders are a specialized type of neural network used in **unsupervised learning**. They are commonly applied to tasks such as **dimensionality reduction**, **data denoising**, and **feature extraction**.

🔹 What is an Autoencoder?



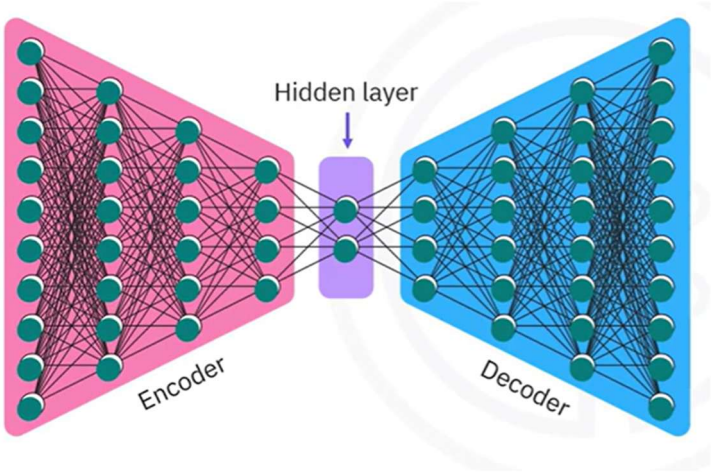
An **autoencoder** is a type of artificial neural network designed to learn **efficient representations** of input data, typically for the purpose of compressing and reconstructing the data.

It is composed of three main parts:

Encoder:
Compresses the input data into a lower-dimensional latent representation.

Bottleneck:
The compressed intermediate representation where the most important features are preserved.

Decoder:
Reconstructs the original input from the compressed representation.



The autoencoder is trained by **minimizing the difference between the input and the reconstructed output**, forcing it to learn meaningful, compressed representations of the data

Type of Autoencoders

Autoencoders come in several forms, each adapted for different purposes:

Basic Autoencoders:
Feature a simple structure with one hidden layer in both the encoder and the decoder.

Variational Autoencoders (VAEs):
Introduce probabilistic components and are designed for **generating new data samples**.

Convolutional Autoencoders:
Use **convolutional layers** and are especially effective for processing **image data**.

Building a Basic Autoencoder with Keras

The implementation is demonstrated using the **MNIST dataset**.

Structure:

1. **Define the input layer** With 784 neurons (28 × 28 image flattened).
2. **Construct the encoder**: Reduce the dimensionality from 784 to 64.
3. **Create the bottleneck**: Compress the 64-dimensional data further to 32 dimensions.
4. **Construct the decoder**: Expand the data from 32 back to 784 dimensions to match the original input.
5. **Compile the model**.

```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
# Encoder
input_layer = Input(shape=(784,))
encoded = Dense(64, activation='relu')(input_layer)

# Bottleneck
bottleneck = Dense(32, activation='relu')(encoded)

# Decoder
decoded = Dense(64, activation='relu')(bottleneck)
output_layer = Dense(784, activation='sigmoid')(decoded)

# Autoencoder model
autoencoder = Model(input_layer, output_layer)

# Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Summary of the model
autoencoder.summary()
```

Data preparation and Training:

1. **Load and preprocess the MNIST dataset**:
   - Normalize the pixel values.
   - Flatten the images into single 784-length vectors.
2. **Train the autoencoder**
   - Using the same dataset for both inputs and targets, teaches the model to **reconstruct the input data**.
   - Emphasizes learning internal data structure without needing labels.

```
from tensorflow.keras.datasets import mnist
import numpy as np
# Load and preprocess the dataset
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# Train the model
autoencoder.fit(x_train, x_train, epochs=50, batch_size=256, shuffle=True,
validation_data=(x_test, x_test))
```

🔹 Fine-Tuning the Autoencoder

You can **fine-tune** a trained autoencoder by **unfreezing specific layers**, to allow parts of the network to be retrained on new or slightly different data.

This helps in adapting the autoencoder to new data or improving its performance

Steps to Fine-Tune:

1. **Unfreeze the last layers** of the autoencoder.
2. **Recompile the model** to update training configuration.
3. **Retrain the model** for a few additional epochs. This improves reconstruction accuracy or adapts the autoencoder to new input domains.

```
# Unfreeze the top layers of the encoder
for layer in autoencoder.layers[-4:]:
  layer.trainable = True

# Compile the model again
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the model again
autoencoder.fit(x_train, x_train, epochs=10, batch_size=256, shuffle=True,
validation_data=(x_test, x_test))
```

🧠 Key Concepts

Autoencoders work by compressing and reconstructing input data using an encoder–decoder structure.

They are trained by minimizing the reconstruction error between inputs and outputs.

The bottleneck forces the network to learn **core features** of the data.

Keras provides a flexible way to build and train these models using Functional API.

☑ Takeaways

☑ Autoencoders are effective tools for learning compressed representations of data in an unsupervised setting.

☑ The architecture includes an encoder, bottleneck, and decoder, which together compress and reconstruct input data.

☑ Keras makes it easy to implement and train autoencoders, and models can be fine-tuned to improve performance or adapt to new datasets.

☑ Different types of autoencoders serve specialized tasks, such as generation (VAEs) or image processing (convolutional autoencoders).

📌 Diffusion Models

Diffusion models are a type of generative model designed to create high-quality synthetic data.

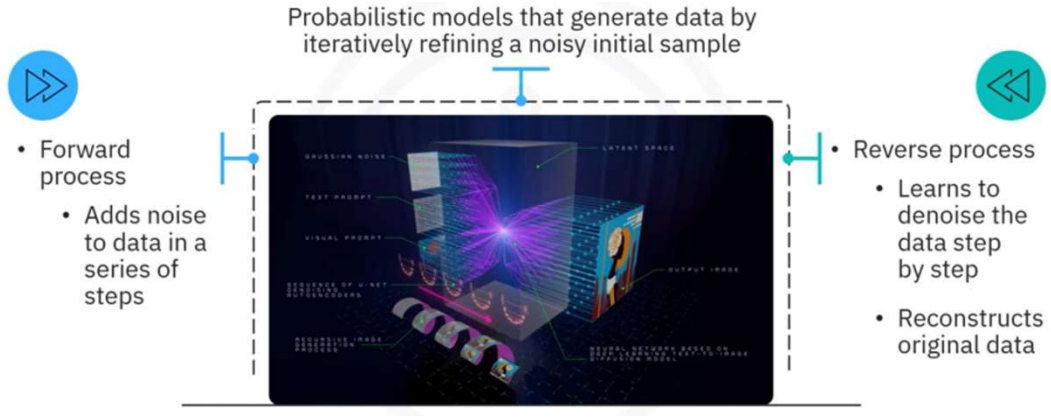
These models have gained popularity due to their effectiveness in tasks such as image generation and enhancement.

🔹 What are diffusion models?

Diffusion models are **probabilistic neural networks** that generate data by **iteratively refining a noisy input**, such as random noise, until it becomes a coherent data sample.

Diffusion models generate data through a two-step:

- **Forward Process:** Adds noise to the original data over a series of time steps, converting structured input into random noise. The process mimics how data deteriorates gradually over time.
- **Reverse Process:** Trains the model to remove noise step by step, the model learns to recover the original sample by reversing the forward process, eventually transforming noise into a meaningful data instance.



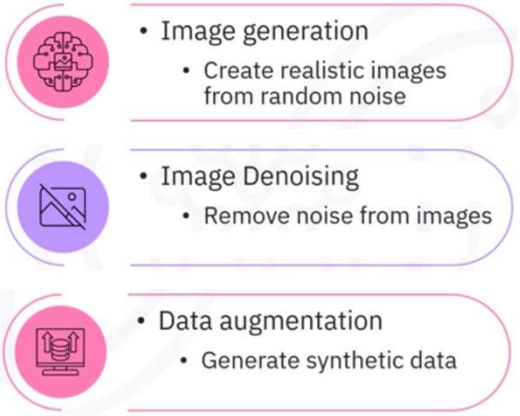
This approach mirrors the physical diffusion process, where particles move from areas of high concentration to low concentration. By learning how to reverse this noise process, the model can create new, coherent data from pure noise.

Applications:

Image Generation:
They can generate realistic images, producing coherent visual content after the reverse denoising process.

Image Denoising:
These models can clean noisy images to restore original images.

Data Augmentation:
They can synthesize new samples that expand existing datasets, useful for training deep learning models in scenarios with limited labeled data.



Building a Diffusion model in Keras

The implementation uses the MNIST dataset and is designed to learn how to denoise images corrupted by random noise.

1. Define model architecture

- A simple **Convolutional Neural Network (CNN)** is defined as the diffusion model.
- The CNN takes a noisy 28x28 image as input.
- The image is processed through several convolutional and dense layers.
- The output is a denoised image of the same size as the input.

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, Flatten, Dense,
Reshape, Conv2DTranspose
from tensorflow.keras.models import Model
import numpy as np
# Define the diffusion model architecture
input_layer = Input(shape=(28, 28, 1))
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_layer)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
x = Dense(784, activation='sigmoid')(x)
output_layer = Reshape((28, 28, 1))(x)

diffusion_model = Model(input_layer, output_layer)

# Compile the model
diffusion_model.compile(optimizer='adam', loss='binary_crossentropy')

# Summary of the model
diffusion_model.summary()
```

2. Data Preparation

- The **MNIST dataset** is loaded and preprocessed.
- Pixel values are normalized to the [0, 1] range.
- Random noise is added to the images to simulate the forward diffusion process.
- The noisy images are used as inputs, and the original images are used as targets for training.

```
from tensorflow.keras.datasets import mnist
# Load the dataset
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)
# Add noise to the images
noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0,
scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

3. Training and evaluating the Model

- The model is trained on the noisy input images with the goal of reconstructing the original clean images.
- This trains the model to learn the reverse denoising process from noise to structure.

```
# Train the model
diffusion_model.fit(x_train_noisy, x_train, epochs=50, batch_size=128,
shuffle=True, validation_data=(x_test_noisy, x_test))
```

- After training, the model is evaluated based on its ability to restore images:
  - The model makes predictions on noisy images.
  - The predictions are visualized alongside the original and noisy images.
  - This visual comparison demonstrates how effectively the model has learned to denoise inputs and reconstruct coherent outputs from corrupted data.

```
import matplotlib.pyplot as plt
# Predict the denoised images
denoised_images = diffusion_model.predict(x_test_noisy)
# Visualize the results
n = 10 # Number of digits to display
plt.figure(figsize=(20, 6))
for i in range(n):
  # Display original
  ax = plt.subplot(3, n, i + 1)
  plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
# Display noisy
  ax = plt.subplot(3, n, i + 1 + n)
  plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
# Display denoised
  ax = plt.subplot(3, n, i + 1 + 2*n)
  plt.imshow(denoised_images[i].reshape(28, 28), cmap='gray')
  ax.get_xaxis().set_visible(False)
```

🧪 Fine-Tuning

To enhance performance, the model can be fine-tuned:

- The last four layers of the trained diffusion model are unfrozen.
- The model is recompiled and retrained for additional epochs.
- Fine-tuning helps refine the model's parameters and can lead to improved denoising accuracy and better-quality image reconstruction.

```
# Unfreeze the top layers of the model
for layer in diffusion_model.layers[-4:]:
  layer.trainable = True

# Compile the model again
diffusion_model.compile(optimizer='adam', loss='binary_crossentropy')

# Train the model again
diffusion_model.fit(x_train_noisy, x_train, epochs=10, batch_size=128,
shuffle=True, validation_data=(x_test_noisy, x_test))
```

☑Takeaways

☑ Diffusion models are generative models that work by progressively adding noise to data (forward process) and then learning to reverse this process to reconstruct original data (reverse process).

☑ They are inspired by the physical diffusion process and can generate data from pure noise through learned denoising.

☑ These models are widely used for image generation, denoising, and data augmentation.

☑ Using Keras, a diffusion model can be implemented with a CNN architecture, trained on noisy image inputs and clean targets, and further fine-tuned for improved results.

M4 – Section 2
GANs and TensorFlow

📌 Generative Adversarial Networks (GANs)

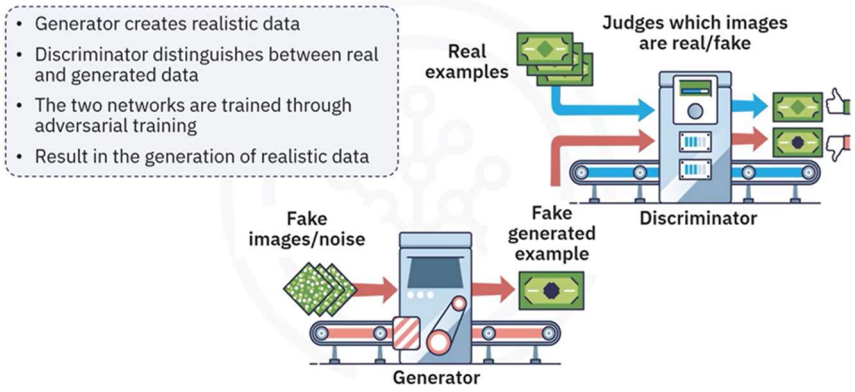
Generative Adversarial Networks (GANs) are a class of neural network architectures designed to generate synthetic data that closely resembles real-world data.

They have become widely known for their ability to create realistic content such as images, music, and text.

🔹 Concept and Architecture

A GAN consists of two primary components:

- **Generator**: Receives a random noise vector as input and generates synthetic data (e.g., images). Its objective is to produce data that mimics real data as closely as possible.
- **Discriminator**: Receives both real data and synthetic data generated by the generator. It attempts to classify each input as real or fake. Its objective is to accurately distinguish between genuine and generated data samples.



These two networks are trained simultaneously in an adversarial process where:

1. The **generator** is trained to fool the discriminator by creating realistic data.
2. The **discriminator** is trained to correctly identify real versus fake data.

This competition continues over multiple training cycles, pushing both networks to improve. Over time, the generator becomes so proficient that the discriminator can no longer reliably tell the difference between real and generated data.

🔧 **Applications of GANs:**

GANs have enabled significant advances across various domains due to their generative capabilities:

- **Image Generation:** Create high-quality images from random noise inputs.
- **Image-to-Image Translation:** Convert images from one domain to another (e.g., transforming sketches into realistic photos).
- **Text-to-Image Synthesis:** Generate images based on textual descriptions.
- **Data Augmentation:** Produce synthetic samples to enrich training datasets and reduce overfitting.

🔹 Building a Basic GAN in Keras

The process of building a GAN in Keras involves creating the two individual models (generator and discriminator) and then combining them into a complete GAN framework.

⚡ Building Process

1. Model Definitions

🔗 **Generator model:**

Accepts a 100-dimensional noise vector as input.

Outputs a synthetic image (e.g., 28x28 pixels).

The architecture includes dense layers that map noise to the structured image data.

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, LeakyReLU, BatchNormalization,
Reshape, Flatten, Input
from tensorflow.keras.models import Model, Sequential
# Define the generator model
def build_generator():
  model = Sequential()
  model.add(Dense(256, input_dim=100))
  model.add(LeakyReLU(alpha=0.2))
  model.add(BatchNormalization(momentum=0.8))
  model.add(Dense(512))
  model.add(LeakyReLU(alpha=0.2))
  model.add(BatchNormalization(momentum=0.8))
  model.add(Dense(1024))
  model.add(LeakyReLU(alpha=0.2))
  model.add(BatchNormalization(momentum=0.8))
  model.add(Dense(28 * 28 * 1, activation='tanh'))
  model.add(Reshape((28, 28, 1)))
  return model
```

🔗 **Discriminator model:**

Accepts an image as input (real or generated).

Outputs a single probability score indicating whether the input is real or fake.

The architecture includes dense layers for classification.

```
# Define the discriminator model
def build_discriminator():
  model = Sequential()
  model.add(Flatten(input_shape=(28, 28, 1)))
  model.add(Dense(512))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dense(256))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dense(1, activation='sigmoid'))
  return model
```

2. **Combining into a GAN:**

   A full GAN model is created by stacking the generator and discriminator.

   During GAN compilation:

   The discriminator is frozen (non-trainable) to ensure only the generator is updated when training the combined model.

   This setup ensures that the generator learns to improve its outputs based solely on the discriminator's feedback.

```
# Build and compile the discriminator
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Build the generator
generator = build_generator()

# Create the GAN by stacking the generator and the discriminator
gan_input = Input(shape=(100,))
generated_image = generator(gan_input)
discriminator.trainable = False
gan_output = discriminator(generated_image)
gan = Model(gan_input, gan_output)
gan.compile(loss='binary_crossentropy', optimizer='adam')
```

Training Process

Training a GAN involves alternating between updating the discriminator and the generator in each step of the training loop:

1. **Discriminator Training**:
   - A batch of real images is sampled from the dataset.
   - A batch of fake images is generated using the current generator.
   - The discriminator is trained to assign high confidence to real images and low confidence to generated ones.

2. **Generator Training**:
   - A new batch of noise vectors is passed through the generator.
   - These generated images are evaluated by the discriminator.
   - The generator is updated to improve its ability to fool the discriminator.

This adversarial cycle continues over many epochs, with the generator becoming increasingly adept at producing realistic images.

```
import numpy as np
from tensorflow.keras.datasets import mnist

# Load and preprocess the MNIST dataset
(x_train, _), (_, _) = mnist.load_data()
x_train = x_train / 127.5 - 1.
x_train = np.expand_dims(x_train, axis=-1)

# Training parameters
batch_size = 64
epochs = 10000
sample_interval = 1000

# Adversarial ground truths
real = np.ones((batch_size, 1))
fake = np.zeros((batch_size, 1))

# Training loop
for epoch in range(epochs):
  # Train the discriminator
  idx = np.random.randint(0, x_train.shape[0], batch_size)
  real_images = x_train[idx]
  noise = np.random.normal(0, 1, (batch_size, 100))
  generated_images = generator.predict(noise)
  d_loss_real = discriminator.train_on_batch(real_images, real)
  d_loss_fake = discriminator.train_on_batch(generated_images, fake)
  d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

# Train the generator
noise = np.random.normal(0, 1, (batch_size, 100))
g_loss = gan.train_on_batch(noise, real)

# Print the progress
if epoch % sample_interval == 0:
    print(f"{epoch} [D loss: {d_loss[0]}] [D accuracy: {100 * d_loss[1]}]
    [G loss: {g_loss}]")
```

🔹 Evaluating the GAN

After training a GAN, it is essential to evaluate the quality and effectiveness of the synthetic data produced by the generator. A well-rounded evaluation combines both **qualitative inspection** and **quantitative metrics** to assess visual realism, statistical similarity, and training dynamics.

🖼️ Qualitative Assessment: Visual Inspection

Throughout the training process, the quality of generated images can be evaluated visually.

Visual inspection is the most immediate way to assess the output of a GAN. It involves manually reviewing the generated images using a function like sample_images to display a grid of outputs. The following criteria should be evaluated:

- **Clarity**: The images should be sharp, with clean edges and distinct shapes. Blurry images suggest the generator has not fully captured the underlying data distribution.
- **Coherence**: The generated images should have structural integrity that reflects the real data. For example, MNIST outputs should resemble legible digits with correct stroke patterns and contours.
- **Diversity**: The outputs should vary across the dataset. Lack of variation can indicate **mode collapse**, where the generator produces nearly identical outputs regardless of the input noise vector.

Instructions for inspection:

1. Call the **sample_images** function after training to generate a 5x5 grid of synthetic images.
2. Review the grid visually for clarity, coherence, and diversity.
3. Use this process iteratively during training to detect degradation or improvement in generation quality.

```python
import matplotlib.pyplot as plt

def sample_images(epoch):
  noise = np.random.normal(0, 1, (25, 100))
  gen_images = generator.predict(noise)
  gen_images = 0.5 * gen_images + 0.5

  fig, axs = plt.subplots(5, 5, figsize=(10, 10))
  count = 0
  for i in range(5):
    for j in range(5):
      axs[i, j].imshow(gen_images[count, :, :, 0], cmap='gray')
      axs[i, j].axis('off')
      count += 1
  plt.show()

# Sample images at the end of training
sample_images(epochs)
```

Visual inspection provides fast, intuitive feedback but lacks objectivity. If quality issues are detected, consider tuning model architecture, learning rates, or batch size.

📏 Quantitative Assessment: Metric

While visual inspection offers immediate insight, it does not provide reproducible or statistically grounded metrics. Quantitative evaluations offer objective, measurable indicators of GAN performance:

- **Fréchet Inception Distance (FID)**: Measures the similarity between the distributions of real and generated images. It uses feature embeddings from a pre-trained network (e.g., Inception-V3) and computes the Fréchet distance between them. Lower scores indicate higher similarity and better performance.
- **Inception Score (IS)**: Evaluates both the clarity and diversity of generated images. Higher IS suggests that the images are recognizable and belong to distinct classes. IS is more suitable for complex datasets; it is less informative for datasets like MNIST.
- **Discriminator Accuracy**: During adversarial training, the ideal discriminator accuracy is around **50%**, indicating that the generator is producing images so realistic that the discriminator is uncertain. If accuracy is significantly higher or lower, it may suggest model imbalance or insufficient generator training.

In the provided code snippet, we calculate the discriminator accuracy on both the real and fake images.

```
# Calculate and print the discriminator accuracy on real vs. fake images
noise = np.random.normal(0, 1, (batch_size, 100))
generated_images = generator.predict(noise)

# Evaluate the discriminator on real images
real_images = x_train[np.random.randint(0, x_train.shape[0], batch_size)]
d_loss_real = discriminator.evaluate(real_images, np.ones((batch_size, 1)), verbose=0)

# Evaluate the discriminator on fake images
d_loss_fake = discriminator.evaluate(generated_images, np.zeros((batch_size, 1)), verbose=0)

print(f"Discriminator Accuracy on Real Images: {d_loss_real[1] * 100:.2f}%")
print(f"Discriminator Accuracy on Fake Images: {d_loss_fake[1] * 100:.2f}%")
```

If the discriminator's accuracy is around 50%, it implies the generator has learned to produce realistic images.

Higher or lower accuracy may suggest that either the generator is not producing convincing images, or the discriminator is overfitting.

🔁 Combining Qualitative and Quantitative Evaluation

For a robust GAN evaluation strategy, combine visual and metric-based assessments:

1. **Visual inspection**: Perform early and often to identify symptoms like blurry outputs, lack of structure, or mode collapse.
2. **Metric tracking**: Use FID or IS if working with more complex datasets. For MNIST or other simple datasets, discriminator accuracy provides meaningful feedback.
3. **Loss monitoring**: Plot generator and discriminator losses over time to assess convergence and stability. Irregular spikes or vanishing gradients indicate training imbalance.

☑Takeaways

☑ GANs are powerful generative models composed of a generator and discriminator trained in opposition.

☑ The generator creates synthetic data from random noise, while the discriminator classifies data as real or fake.

☑ Through adversarial training, both models improve iteratively, resulting in highly realistic data generation.

☑ GANs are widely used in image synthesis, translation, and data augmentation tasks.

☑ In Keras, a basic GAN can be built by defining both models, combining them, and implementing an alternating training loop with adversarial objectives.

📌TensorFlow for Unsupervised Learning

Unsupervised learning is a machine learning paradigm in which the model learns from data without any labeled responses or target outputs.

The objective is to uncover underlying patterns, groupings, or structures that are not explicitly provided.

TensorFlow offers robust tools to implement and scale unsupervised learning workflows across diverse applications.

Common unsupervised learning tasks include:

- **Clustering**: Grouping similar data points based on their feature similarity.
- **Dimensionality Reduction**: Compressing high-dimensional data into fewer dimensions while retaining key information.
- **Anomaly Detection**: Identifying data points that deviate significantly from the norm or distribution.

These techniques are widely applicable in domains such as customer segmentation, image compression, and fraud detection.

Building Clustering Models with TensorFlow

Clustering Process in TensorFlow:

1. **Data Preparation**:
   - Load the MNIST dataset.
   - Normalize pixel values to bring all features into a similar range.
   - Reshape the image data for compatibility with clustering models.
2. **K-Means Clustering**:
   - Apply the K-Means algorithm to the processed MNIST data.
   - Configure the algorithm to cluster the dataset into ten groups (representing the ten digits).
3. **Visualization**:
   - Display sample images from each cluster to visually assess the quality of clustering.
   - Groupings reveal how the algorithm identifies similar digit patterns.

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
# Load and preprocess the MNIST dataset
(x_train, _), (_, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_train = x_train.reshape(-1, 28 * 28)

# Apply K-means clustering
kmeans = KMeans(n_clusters=10)
kmeans.fit(x_train)
labels = kmeans.labels_

# Display a few clustered images
def display_cluster_images(x_train, labels):
    plt.figure(figsize=(10, 10))
    for i in range(10):
        idxs = np.where(labels == i)[0]
        for j in range(10):
            plt_idx = i * 10 + j + 1
            plt.subplot(10, 10, plt_idx)
            plt.imshow(x_train[idxs[j]].reshape(28, 28), cmap='gray')
            plt.axis('off')
    plt.show()

display_cluster_images(x_train, labels)
```

Dimensionality Reduction with Autoencoders

Dimensionality reduction is essential when dealing with high-dimensional datasets. Autoencoders are effective neural networks for this purpose. They compress input data into a lower-dimensional representation (bottleneck) and then reconstruct it to its original form.

Autoencoder Architecture in TensorFlow:

1. **Model Definition**:
   - Input Layer: Accepts the original image data (e.g., 784-pixel MNIST vectors).
   - Encoding Layer: Compresses the data to a smaller feature space.
   - Bottleneck Layer: Captures the most essential information in a compact representation.
   - Decoding Layer: Reconstructs the original data from the bottleneck.
   - Output Layer: Produces a reconstruction of the input image.
2. **Training**:
   - Train the autoencoder using the input data as both the input and the output.
   - Minimize the reconstruction loss to ensure the learned representation retains essential structure.
3. **Evaluation and Visualization**:
   - Extract the compressed features from the bottleneck layer after training.
   - Apply **t-distributed Stochastic Neighbor Embedding (t-SNE)** to project these high-dimensional features into two dimensions.
   - Plot the resulting 2D embeddings to visualize how the model separates and clusters the data in lower-dimensional space.

```
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
# Define the autoencoder model
input_layer = Input(shape=(784,))
encoded = Dense(64, activation='relu')(input_layer)
bottleneck = Dense(32, activation='relu')(encoded)
decoded = Dense(64, activation='relu')(bottleneck)
output_layer = Dense(784, activation='sigmoid')(decoded)
autoencoder = Model(input_layer, output_layer)
# Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
# Train the autoencoder
autoencoder.fit(x_train, x_train, epochs=50, batch_size=256, shuffle=True,
validation_split=0.2)
```

```
from sklearn.manifold import TSNE
# Get the compressed representations
encoder = Model(input_layer, bottleneck)
compressed_data = encoder.predict(x_train)

# Use t-SNE for 2D visualization
tsne = TSNE(n_components=2)
compressed_2d = tsne.fit_transform(compressed_data)

# Plot the compressed data
plt.scatter(compressed_2d[:, 0], compressed_2d[:, 1], c=labels, cmap='viridis')
plt.colorbar()
plt.show()
```

☑Takeaways

☑ TensorFlow provides scalable and flexible tools for implementing unsupervised learning models.

☑ Unsupervised learning reveals hidden structures in unlabeled data and supports key tasks such as clustering, dimensionality reduction, and anomaly detection.

☑ Clustering with TensorFlow (e.g., using K-Means) enables automatic grouping of similar instances from complex datasets.

☑ Autoencoders in TensorFlow effectively reduce data dimensionality by learning compressed representations that retain essential features.

☑ Visualizations such as t-SNE plots allow inspection of how well the autoencoder clusters similar data in lower dimensions.

Module 5
Advanced Keras Techniques

M5 – Section 1
Advanced Keras techniques and Custom Training Loops

📌 Advanced Keras Techniques

This topic introduces advanced techniques available in Keras that extend model development flexibility, customization, and performance optimization.

These methods are particularly useful when default training workflows are not sufficient for specialized tasks, or when fine-grained control over the process is needed.

Keras offers a variety of advanced techniques that can significantly enhance your model development process:

- Custom training loops for detailed control over training behavior
- Specialized layers to extend core functionality
- Callback functions to monitor and modify training in real-time

🔹 Custom Training Loops

Custom training loops allow you to tailor the training process to your specific needs.

While the Keras ***.fit()*** method is powerful and convenient, using a custom loop offers more control—especially for implementing complex training strategies or custom loss functions.

🔧 **Implementation:**

- Start by defining a simple neural network model.
- Set up the optimizer and the loss function.
- Use a training loop that iterates over the dataset for a specified number of epochs.
- Within each training step, a gradient tape is used to record operations for automatic differentiation.
- Gradients are calculated and applied to the model's trainable weights.

This approach allows greater flexibility compared to the built-in .fit() method.

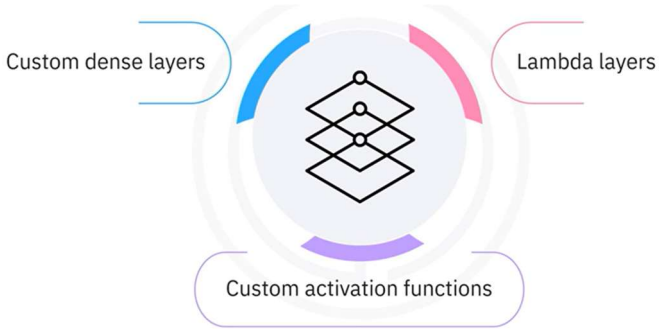
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Create a simple model
model = Sequential([Dense(64, activation='relu'), Dense(10)])
# Custom training loop
optimizer = tf.keras.optimizers.Adam()
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# Load or create your training dataset here
# Example:
x_train = tf.random.uniform((100, 10)) # Replace with your actual training data
y_train = tf.random.uniform((100,), maxval=10, dtype=tf.int64) # Replace with
your actual training labels
train_dataset = tf.data.Dataset.from_tensor_slices((x_train,
y_train)).batch(32)
for epoch in range(10):
    for x_batch, y_batch in train_dataset:
        with tf.GradientTape() as tape:
            logits = model(x_batch, training=True)
            loss = loss_fn(y_batch, logits)
        grads = tape.gradient(loss, model.trainable_weights)
        optimizer.apply_gradients(zip(grads, model.trainable_weights))
    print(f'Epoch {epoch + 1}, Loss: {loss.numpy()}')
```

🔹 Specialized Layers

Keras allows defining custom layers to extend built-in functionality, which is especially useful for implementing behaviors or custom activations that are not available in the standard library. Such as:

- **Custom Dense Layers**, dense layers created by subclassing the layer class (example provided below).
- **Lambda Layers**, allow definition of simple custom computations inline within the model, typically used for non-trainable logic like reshaping or mathematical operations



These tools are essential for building advanced neural networks with custom operations not supported by default.

🔧 **Custom Dense Layer implementation:**

A dense layer can be created by subclassing the Layer class.

- In the build() method, weights and biases are initialized.
- In the call() method, the forward computation is defined.
- This custom layer is then used like any other Keras layer inside a model.

```
from tensorflow.keras.layers import Layer
import tensorflow as tf

class CustomDenseLayer(Layer):
    def __init__(self, units=32):
        super(CustomDenseLayer, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], self.units),
                                 initializer='random_normal',
                                 trainable=True)
        self.b = self.add_weight(shape=(self.units,),
                                 initializer='zeros',
                                 trainable=True)
    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

# Usage in a model
model = Sequential([CustomDenseLayer(64), Dense(10)])
```

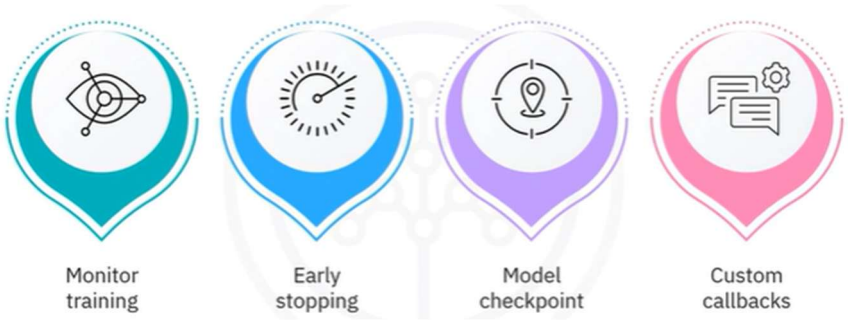
🔹 Advanced Callback Functions

Callback functions in Keras enable developers to inject custom behavior during the training lifecycle.

These behaviors can be defined at the start or end of epochs, after each batch, or in response to certain training conditions.

Built in capabilities:

- Monitor training metrics in real-time.
- Implement early stopping when validation loss stops improving.
- Save model checkpoints during training.
- Implement Custom callbacks



Callbacks are powerful tools for enhancing transparency, debugging, and controlling training behavior based on dynamic conditions.

🔧 **Custom Dense Layer implementation:**

In this implementation we are creating a custom callback that logs additional metrics during training:

- Subclass the Callback class.
- Override on_epoch_end() to display metrics like loss and accuracy at the end of each epoch.
- Use this callback during training by passing it to the .fit() method.

```
 from tensorflow.keras.callbacks import Callback
class CustomCallback(Callback):
    def on_epoch_end(self, epoch, logs=None):
        logs = logs or {}
        print(f'End of epoch {epoch}, loss: {logs.get("loss")}, accuracy:
{logs.get("accuracy")}')

# Usage in model training
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

model.fit(train_dataset, epochs=10, callbacks=[CustomCallback()])
```

🔹 Model Optimization

Optimizing your models is crucial for achieving the best performance. TensorFlow provides tools for this, including:

- **Mixed Precision Training**: This technique speeds up training and reduces memory usage by using 16-bit floats (float16) where appropriate. It's activated by setting the global policy to mixed_float16. This allows TensorFlow to automatically use float16 in supported layers while maintaining accuracy.
- **TensorFlow Model Optimization Toolkit**: This toolkit includes advanced techniques such as pruning, quantization, and clustering to make models smaller and faster for deployment, especially in edge environments.

These optimization techniques help improve training efficiency and make your models production-ready.

🔧 **Mixed precision training implementation:**

```
from tensorflow.keras import mixed_precision

# Enable mixed precision
mixed_precision.set_global_policy('mixed_float16')

# Model definition
model = Sequential([Dense(64, activation='relu'), Dense(10)])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Training the model
model.fit(train_dataset, epochs=10)
```

☑Takeaways

☑ **Custom Training Loops** allow full control over the training steps using GradientTape, enabling fine-tuned loss handling and model updates beyond what *.fit()* provides.

☑ **Specialized Layers** can be built by subclassing *keras.layers.Layer*, giving flexibility to define unique architectures and operations not available in built-in layers.

☑ **Advanced Callback Functions** enhance training control by allowing event-based logic like metric logging, checkpointing, or training termination.

☑ **Mixed Precision Training** leverages lower-precision arithmetic to speed up training and reduce memory footprint, especially beneficial on supported hardware.

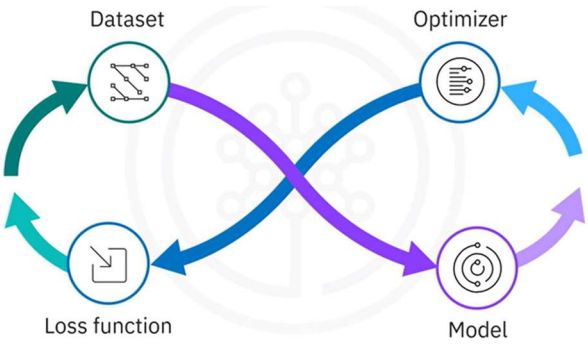
☑ **TensorFlow Model Optimization Toolkit** provides pruning, quantization, and clustering strategies to streamline and deploy efficient models.

📌Custom Training Loops in Keras

While the built-in *.fit()* method is powerful and convenient, custom loops allow the training workflow to be adapted for complex strategies, custom loss functions, and experimental model behaviors. This is especially useful for research, debugging, or integrating advanced logging and metrics.

🔹Basic Structure of a Custom Training Loop

A custom training loop in Keras is composed of the following core components:



- **Dataset**: The training data must be prepared and batched.
- **Model**: A simple neural network is defined.
- **Loss Function**: Measures how well the model's predicted outputs (logits) match the true labels. The choice of loss function affects how gradients are computed and how the model learns.
- **Optimizer**: Responsible for updating the model's weights to minimize the loss. Common optimizers include Adam or SGD.

These components are explicitly defined to allow manual control over the training process.

Implementing custom training loop

The implementation of a custom training loop follows a structured series of steps, that, gives full control over the training process and supports precise debugging and customization:

1. **Set up environment:**

   By importing the necessary libraries and preparing the dataset.

   In the example, the MNIST dataset is loaded, the pixel values are normalized to a [0, 1] range, and tf.data.Dataset is used to batch the data for efficient iteration.

```
!pip install tensorflow==2.16.2 matplotlib==3.9.1

import warnings
warnings.filterwarnings("ignore", category=UserWarning)

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten

# Prepare a simple dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
train_dataset = tf.data.Dataset.from_tensor_slices((x_train,
y_train)).batch(32)
```

2. **Create model, define loss function and optimizer:**

   A simple model is created, beginning with a Flatten layer to process image data.

   A loss function is defined, such as sparse categorical cross-entropy, to compute the error between predictions and true labels.

   An optimizer, like Adam, is initialized to manage the update of model weights.

```
# Create a simple model with a Flatten layer
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(10)
])

# Define the loss function and optimizer
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()
```

3. **Implementing custom training loop:**

   Iterates over the dataset for a specified number of epochs.

   In each epoch, the model processes every batch of input data.

   For each batch:

   - The inputs are passed through the model to compute predictions (logits).
   - The loss is calculated by comparing logits with the actual labels using the chosen loss function.
   - Gradients of the loss are computed with respect to the model's trainable weights.
   - The optimizer applies these gradients to update the model parameters and minimize the loss.

```
# Custom training loop
epochs = 5
for epoch in range(epochs):
    print(f'Start of epoch {epoch + 1}')
    for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):
        with tf.GradientTape() as tape:  # Added this line for proper code
functionality
            logits = model(x_batch_train, training=True)
            loss_value = loss_fn(y_batch_train, logits)

        grads = tape.gradient(loss_value, model.trainable_weights)
        optimizer.apply_gradients(zip(grads, model.trainable_weights))

        if step % 200 == 0:
            print(f'Epoch {epoch + 1} Step {step}: Loss =
{loss_value.numpy()}')
```

◆ Role of *tf.GradientTape*

To compute gradients and update model weights, the loop uses ***tf.GradientTape***.

tf.GradientTape is a TensorFlow API used to record operations for **automatic differentiation**, which is a technique to compute gradients required to optimize a model during training.

By using tf.GradientTape, you can watch the forward pass through the network, which allows TensorFlow to record all operations on the watched variable

Here's how it works:

- During the forward pass, tf.GradientTape tracks the operations executed on watched variables, typically the model's trainable weights.

- These recorded operations are then used to compute the gradient of the loss function with respect to each variable.
- The gradients are passed to the optimizer, which adjusts the weights accordingly.

This process ensures that the model learns effectively by updating weights in the direction that minimizes the loss.

Using GradientTape also enables flexibility in defining custom training logic, including support for non-standard operations or loss terms.

🔹 Benefits of Custom Training Loops

Custom training loops provide several key advantages over the default .fit() method:

- **Granular Control**: Every part of the training cycle can be explicitly controlled, from loss computation to weight updates.
- **Custom Loss and Metrics**: Allows defining and integrating non-standard loss functions and tracking custom evaluation metrics.
- **Advanced Logging and Monitoring**: Enables the collection of detailed metrics or internal state information at any point during training.
- **Research Flexibility**: Especially useful in research environments where experimental algorithms or training conditions need to be implemented.
- **Integration with Custom Components**: Supports the use of custom layers, activation functions, or external components that do not conform to standard Keras workflows.

☑Takeaways

☑ A custom training loop is composed of a dataset, model, loss function, and optimizer.

☑ The training procedure includes computing logits, calculating loss, and updating weights using computed gradients.

☑ tf.GradientTape enables automatic differentiation by recording operations on trainable variables.

☑ Custom loops provide:

- Support for custom loss functions and metrics.
- Integration with advanced logging.
- Flexibility for experimental training workflows.
- Compatibility with non-standard model components and custom operations.

M5 – Section 2
Hyperparameter and Model Optimization

📌 Hyperparameter Tuning with Keras Tuner

Hyperparameter tuning is a key stage in the machine learning pipeline, enabling the discovery of the most effective configuration for training deep learning models.

Keras Tuner provides an intuitive interface and supports multiple search strategies. The process includes setting up the tuner, defining a model with tunable hyperparameters, configuring the search, executing it, analyzing the results, and training the final model using the best configuration.

🔹 Understanding Hyperparameters

Hyperparameters are **variables set before training begins**. They govern how the model learns and significantly influence training outcomes. Examples include:

- ***Learning rate*:** Controls how quickly the model adapts.
- ***Batch size:*** Determines how many samples are processed before updating weights.
- ***Number of units:*** Defines the architecture depth or complexity of a layer.

These values are not learned during training, and selecting them effectively is critical for model performance

🔹 What is Keras Tuner

Keras Tuner is a **library** that automates the process of finding the best hyperparameters for a model.

It supports three major search strategies:

- **Random Search**: explores the space by randomly selecting values.
- **Hyperband**: adjusts training resources dynamically to focus on the most promising trials.
- **Bayesian Optimization**: uses past evaluations to inform the next choice of values.

These algorithms allow for efficient and automated exploration of possible configurations.

🔹 Hyperparameter search implementation.

🔧 Setting Up the Environment:

This step prepares the environment for model definition and tuning.

- Install Keras Tuner via ***pip.***
- Import required modules.

```
!pip install keras-tuner
import keras_tuner as kt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import Adam
```

🔧 Defining the Model with Hyperparameters:

A special model-building function is created to define how the model will change depending on different hyperparameter values:

- Use ***hp.Int()*** to define an integer range (e.g., number of units in a dense layer).
- Use ***hp.Float()*** for floating-point values (e.g., learning rate).
- Combine these with layers like Flatten and Dense to construct the model architecture dynamically.

```
def build_model(hp):
    model = Sequential([
              Flatten(input_shape=(28, 28)),
              Dense(
                units=hp.Int('units', min_value=32, max_value=512, step=32)
                activation='relu'
              ),
              Dense(10, activation='softmax')
    ])
    model.compile(
        optimizer=Adam(learning_rate=hp.Float(
                        'learning_rate',
                        min_value=1e-4,
                        max_value=1e-2,
                        sampling='LOG')
        ),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
```

🔧 **Configuring the Tuner:**

These settings define how tuning will proceed and how results are tracked.

The tuner will explore the hyperparameter space and find the best configurations based on model performance:

The tuner object is configured with:

- A search strategy (e.g., RandomSearch).
- The model-building function.
- An optimization objective (e.g., validation accuracy).
- The number of trials (number of different hyperparameters configuration to try) and executions per trial (number of times to run each configuration)
- Directory and project name for storing tuning results.

```
tuner = kt.RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=10,
    executions_per_trial=2,
    directory='my_dir',
    project_name='intro_to_kt'
)
```

🔧 **Running the Hyperparameter Search:**

With everything configured, use the search() method to begin tuning:

- Load and normalize the MNIST dataset.
- Supply training and validation sets.
- Specify the number of training epochs.

Keras Tuner runs multiple trials and tracks the best-performing configurations based on validation accuracy.

```
(x_train, y_train),(x_val, y_val) = mnist.load_data()
x_train, x_val = x_train / 255.0, x_val / 255.0
tuner.search(
  x_train,
  y_train,
  epochs=5,
  validation_data=(x_val, y_val)
)
```

🔧 **Retrieving and Using the Best Hyperparameters:**

After tuning:

- Use ***get_best_hyperparameters()*** to retrieve the optimal values.
- Build a model using these settings.
- Review the model summary for architecture verification.

This results in a final model customized to perform optimally under the chosen hyperparameters.

```
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"""
  The optimal number of units in the first dense layer
  is {best_hps.get('units')}.
  The optimal learning rate for the optimizer is
  {best_hps.get('learning_rate')}.
""")
model = tuner.hypermodel.build(best_hps)
model.summary()
```

🔧 **Training the Final Optimized Model:**

Train the optimized model on the full training set and evaluate it on the test set. This step confirms that the chosen configuration leads to effective and generalizable performance.

```
model.fit(x_train, y_train, epochs=10,validation_split=0.2)
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc}')
```

☑Takeaways

☑ Hyperparameters control how a model trains; examples include learning rate, batch size, and network size.

☑ Keras Tuner simplifies hyperparameter optimization using random search, hyperband, or Bayesian optimization.

☑ Models can be defined with tunable settings, tuned via automated search, and then trained with optimal configurations to improve performance.

📌 Model Optimization

Model optimization aims to improve the performance, efficiency, and scalability of deep learning models.

Optimized models can lead to:

- Faster training times.
- More efficient hardware usage.
- Higher accuracy.

🔹 Weight Initialization

Weight initialization significantly influences how well a neural network converges during training, and performs.

Proper initialization methods help avoid issues like vanishing or exploding gradients.

Two common initialization strategies are:

- **Xavier (Glorot) Initialization**: This method is effective for layers with tanh or sigmoid activations. It sets weights based on the number of input and output units to maintain signal variance across layers.
- **He Initialization**: This method is especially useful for layers using ReLU activation. It maintains a stable gradient flow by scaling the initial weights based on the number of input units, helping prevent the gradient from vanishing or exploding.

In the example, He initialization is applied using Keras to configure the weights for layers with ReLU activations, supporting smoother training.

```
from tensorflow.keras.initializers import HeNormal
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Flatten

# Define the model with 'He' initialization
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu', kernel_initializer=HeNormal()),
# He initialization applied here
    Dense(10, activation='softmax')
])
```

Learning Rate Scheduling

Learning rate scheduling helps dynamically adjust the learning rate during training.

This improves training efficiency by allowing the model to converge faster and fine-tune more precisely in later epochs.

To implement this technique:

Load and preprocess the data.

```
# Load and Preprocess Data
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset
(x_train, y_train) , (x_val, y_val) = mnist.load_data()

# Normalize the input data
x_train = x_train.astype('float32') / 255.0
x_val = x_val.astype('float32') / 255.0

# Reshape the input data (if necessary)
x_train = x_train.reshape(-1, 28, 28)
x_val = x_val.reshape(-1, 28, 28)
```

A learning rate scheduler function is defined to maintain a constant learning rate for the first ten epochs, then exponentially reduce it afterward. This allows the model to learn robust representations early and refine them as training progresses.

```
from tensorflow.keras.callbacks import LearningRateScheduler
import tensorflow as tf
from tensorflow.keras.optimizers import Adam

def scheduler(epoch, lr):
    if epoch < 10:
        return lr
    else:
        return float(lr * tf.math.exp(-0.1))

lr_scheduler = LearningRateScheduler(scheduler)
```

The model is then trained for 20 epochs using the prepared dataset and the learning rate scheduler. Validation data is used during training to monitor performance and generalization.

```
# Compile the model with an optimizer and loss function
model.compile(optimizer=Adam(), loss='categorical_crossentropy',
metrics=['accuracy'])

# Train the model with a learning rate scheduler
history = model.fit(x_train, y_train,
                    validation_data=(x_val, y_val),
                    epochs=20,
                    callbacks=[lr_scheduler])
```

Additional Optimization Techniques

The TensorFlow Model Optimization Toolkit (TF MOT) also introduces several additional model optimization strategies that support performance improvement, memory efficiency, and model deployment:

- **Batch Normalization:**
  Normalizes layer inputs by adjusting and scaling activations.
  - Helps accelerate training.
  - Improves convergence behavior.
  - Reduces sensitivity to the learning rate.

```
Example Code for Batch Normalization:
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, BatchNormalization

# Define a simple model with Batch Normalization
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    BatchNormalization(),
# Batch normalization layer
    Dense(10, activation='softmax')
])
```

- **Mixed Precision Training:**
  Uses both 16-bit and 32-bit floating-point data types.
  - Allows faster training on modern GPUs.
  - Reduces memory usage without sacrificing model accuracy.

```
#Example Code for Mixed Precision Training:
from tensorflow.keras import mixed_precision

# Enable mixed precision
policy = mixed_precision.Policy('mixed_float16')
mixed_precision.set_global_policy(policy)
```

- **Model Pruning:**
  Reduces the number of parameters by removing less significant connections or neurons from the model.
  - Maintains accuracy while improving efficiency.
  - Reduces computational complexity.

```
#Example Code for Model Pruning:
import tensorflow_model_optimization as tfmot

prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude
# Apply pruning to the model
pruning_params = {'pruning_schedule':
tfmot.sparsity.keras.PolynomialDecay(initial_sparsity=0.0,
final_sparsity=0.5,
begin_step=0,
end_step=2000)}
model_pruned = prune_low_magnitude(model, **pruning_params)
```

- **Quantization:**
  Reduces the precision of the numbers used to represent the model's weights and activations to lower-precision formats (e.g., from float32 to int8)
  - Reduces model size and speeds up inference.
  - Useful for deploying models on edge devices with limited hardware.

```
#Example Code for Quantization:
# Example code for Post-Training Quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
quantized_model = converter.convert()

# Save the quantized model
with open('quantized_model.tflite', 'wb') as f:
    f.write(quantized_model)
```

☑ Takeaways

☑ Model optimization improves model performance, efficiency, and scalability.

☑ Proper weight initialization (e.g., Xavier, He) is crucial for stable training.

☑ Learning rate scheduling adjusts learning rates dynamically for better convergence.

☑ Additional optimization techniques include batch normalization, mixed precision training, pruning, and quantization.

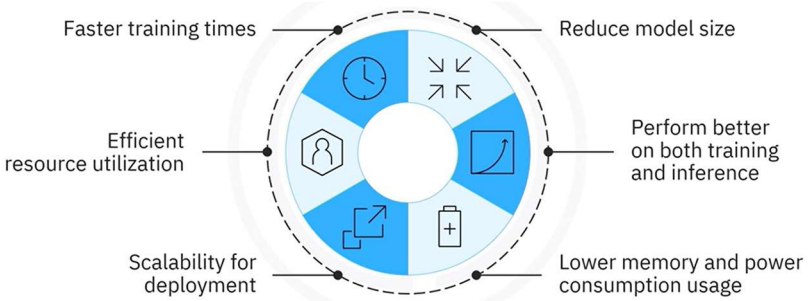
☑ These strategies help accelerate training, reduce memory consumption, and prepare models for efficient deployment.

📌 TensorFlow for Model Optimization

🔹 Optimization Benefits and Objectives

TensorFlow provides a set of tools and techniques aimed at optimizing deep learning models for efficiency in training and deployment.

These methods improve training speed, reduce memory usage, and make models more scalable and suitable for resource-constrained environments.



Optimized models offer better performance, consume less power, and are more efficient for real-world applications.

Model optimization improves:

- Training and inference speed.
- Resource utilization (GPU/CPU efficiency).
- Model size and memory consumption.
- Scalability and deployment feasibility.
- Power efficiency for edge and mobile devices.

TensorFlow facilitates these improvements through built-in support for multiple optimization techniques. These include:

- Mixed-precision training.
- Knowledge distillation.
- Post-training training optimization techniques (like pruning and quantization, seen on previous section).

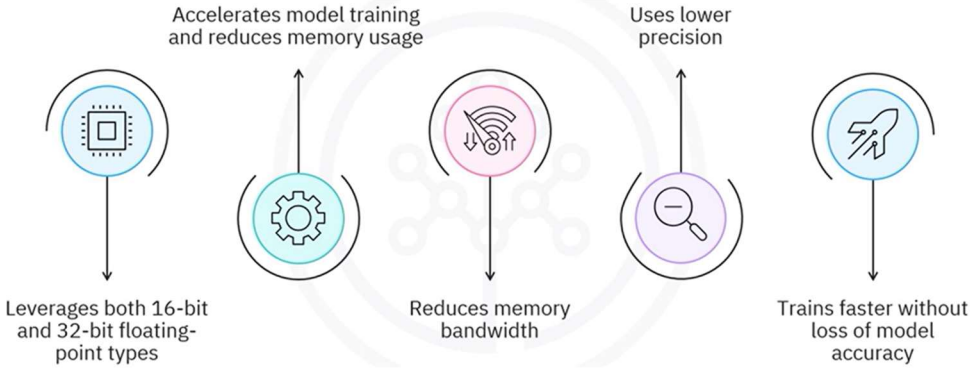
All these methods are integrated and supported by the **TensorFlow Model Optimization Toolkit**, which provides high-level APIs for enhancing model performance both during and after training.

🔹 Mixed-Precision Training

Mixed-precision training uses both 16-bit and 32-bit floating-point types to accelerate model training and memory efficiency.

This technique is especially useful on modern GPUs that support float16 operations.

It reduces memory bandwidth and uses lower precision where possible, allowing models to train faster on modern GPUs without significant loss of model accuracy.



🔧**Implementation:**

The global computation policy is set to "***mixed_float16***", allowing TensorFlow to automatically cast certain tensors to 16-bit precision.

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras import mixed_precision

# Enable mixed precision training
mixed_precision.set_global_policy('mixed_float16')
```

The MNIST dataset is loaded and preprocessed.

```
# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets. mnist.load_data()

# Normalize the data
x_train, x_test =
     x_train.astype('float32') / 255.0,
     x_test.astype('float32') / 255.0
```

A simple neural network model is defined.

```
# Define a simple model
model = models.Sequential([
    layers.Input(shape=(28, 28)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile the model with an optimizer and loss function
optimizer = optimizers.Adam(learning_rate=1e-3)
model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
```

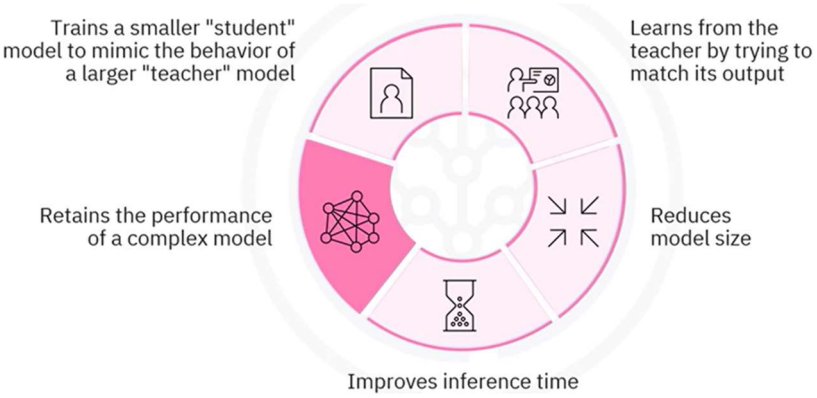
By reducing gradient sizes and using lower precision where appropriate, training becomes faster and less memory-intensive—without sacrificing accuracy.

🔹 Knowledge Distillation

Knowledge distillation **transfers the predictive power of a large, complex model** (teacher) **to a smaller**, lightweight model (student). This "student" model mimics the behavior of the larger "teacher" model.

This helps reduce model size and speed up inference, making deployment on limited-resource devices practical.

This technique maintains most of the original model's accuracy while significantly reducing its size and computational footprint.



How it works:

- A **teacher model** is trained to high accuracy using labeled data.
- A **student model** is trained to mimic the teacher's output logits, rather than learning directly from labels.
- A softened softmax (controlled by a temperature parameter) is applied to the teacher's outputs.
- The student learns using a **distillation loss**, which compares its predictions to the softened outputs of the teacher model.

🔧 Implementation:

The teacher model is trained to achieve high accuracy on the training data.

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
import numpy as np

# Teacher model (simpler model)
teacher_model = models.Sequential([
    layers.Input(shape=(28, 28)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
teacher_model.compile(optimizer=optimizers.Adam(),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# Assume the teacher model is already trained
# For demonstration, we'll skip training.
```

A simpler, student model is built.

```
# Student model (simpler model)
student_model = models.Sequential([
    layers.Input(shape=(28, 28)),
    layers.Flatten(),
    layers.Dense(32, activation='relu'),
    layers.Dense(10, activation='softmax')
])
student_model.compile(optimizer=optimizers.Adam(),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

The student model is then trained using the outputs, logits, of the teacher model instead of the original labels.

This is achieved using a softened version of the outputs, controlled by a temperature parameter in the softmax function.

```
# Distillation loss function using TensorFlow operations
def distillation_loss(teacher_logits, student_logits, temperature=3):
    teacher_probs = tf.nn.softmax(teacher_logits / temperature)
    student_probs = tf.nn.softmax(student_logits / temperature)
    return tf.reduce_mean(
             tf.keras.losses.categorical_crossentropy(
               teacher_probs,
               student_probs
             )
           )
```

Train the student model using knowledge distillation with smaller data and fewer epochs.

Predict teacher logits for the batch using ***tf.GradientTape*** method. Calculate distillation loss and apply gradients.

Use a smaller subset of the dataset and train the student model.

```
# Train the student model using knowledge distillation with smaller data
and fewer epochs
def train_student(student, teacher, x_train, y_train, batch_size=32,
epochs=2, temperature=3):
    for epoch in range(epochs):
        num_batches = len(x_train) // batch_size
        for batch in range(num_batches):
            x_batch = x_train[batch * batch_size: (batch + 1) * batch_size]
            y_batch = y_train[batch * batch_size: (batch + 1) * batch_size]

            # Predict teacher logits for the batch
            teacher_logits = teacher.predict(x_batch)

            with tf.GradientTape() as tape:
                # Predict student logits for the batch
                student_logits = student(x_batch)
                # Calculate distillation loss
                loss = distillation_loss(teacher_logits, student_logits,
                       temperature)

            # Apply gradients
            grads = tape.gradient(loss, student.trainable_variables)
            student.optimizer.apply_gradients(
               zip(grads,student.trainable_variables)
            )
        print(f"Epoch {epoch + 1} completed. Loss: {loss.numpy()}")
```

```
# Use a smaller subset of the MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train[:1000], x_test[:1000]  # Use only 1000 samples
for quick training
x_train, x_test = x_train.astype('float32') / 255.0,
x_test.astype('float32') / 255.0

# Train the student model
train_student(student_model, teacher_model, x_train, y_train,
batch_size=32, epochs=2)
```

☑ Takeaways

☑ TensorFlow provides built-in support for model optimization through training and post-training techniques.

☑ **Mixed-precision training** accelerates training and reduces memory usage using 16-bit precision on supported hardware.

☑ **Knowledge distillation** allows a small model to mimic a larger one, reducing size and inference time while preserving accuracy.

☑ **Post-training methods** like **pruning** and **quantization** further compress models and prepare them for efficient deployment.

☑ All these approaches are supported by the **TensorFlow Model Optimization Toolkit**, enabling scalable and resource-efficient deep learning development.

Module 6

Introduction to Reinforcement Learning with Keras

M6 – Section 1
Reinforcement Learning, Q-Learning, Q-Networks (DQNs)

📌 Introduction to Reinforcement Learning (RL)

🔹 Reinforcement Learning Overview

Reinforcement learning is a machine learning paradigm where an agent interacts with an environment by selecting actions to maximize rewards over time.

The interaction forms a continuous feedback loop:

- **Agent**: The entity that makes decisions.
  In games, it's the player; in web applications, it might be a program that places ads.
- **Environment**: The space in which the agent operates.
  In games, this could be the game board; for web applications, it's the webpage.
- **Actions**: The choices available to the agent at any given state. These alter the environment and trigger feedback.
- **Reward**: The signal returned by the environment to indicate the quality of the action taken.

An action taken by the agent impacts the environment.

The environment then returns a reward to guide the agent's learning. These rewards are often delayed and uncertain, requiring the agent to estimate them over time.

The process repeats dynamically, as actions continuously alter the environment, making the state and outcome fluid and non-static.

🔹 Real-World Examples of RL

Recent advances in deep learning have expanded RL applications:

- **DeepMind (2013)**: Created a system that learned to play Atari games and outperformed humans.
- **AlphaGo (2017)**: Beat the world champion in the complex board game Go.

While **RL** has proven successful in games, it **has high data and computational demands due to the vast number of possible states and actions**.

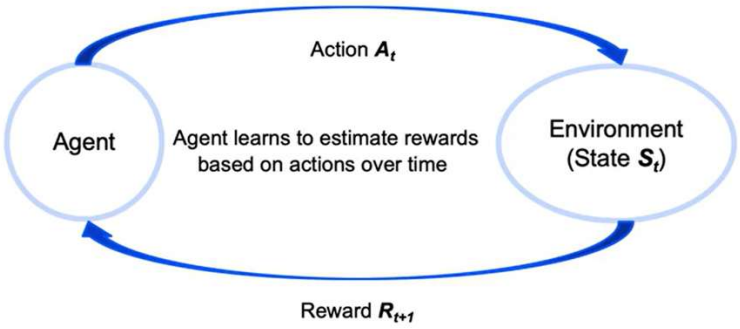
This challenge has slowed its broader adoption but hasn't stopped progress in practical business scenarios, including:

- **Recommendation Engines**: Reward defined by correct user predictions.
- **Marketing Optimization**: Rewards based on clicks or revenue.
- **Automated Bidding**: Rewards optimized around spending per item.

Policy and Rewards

In RL the agent takes some action, that action affects the **current environment**, and then **feedback** from that environment is passed back to the agent in terms of a **reward**.

- If the **feedback resulted in a positive result** in relation to our reward system, the agent's actions are reinforced.
- Vice-versa for **negative results**, if it ended up in a bad environment state, then the agent is **reinforced not to take those same steps.**



The **policy** is the strategy an agent uses to decide which actions to take given a particular state.

The agent's goal is to learn a policy that maximizes expected cumulative rewards over time.

Unlike supervised learning, rewards are not labeled and are often unpredictable or delayed.

Agents must balance **short-term and long-term rewards**, which introduces a critical trade-off in decision-making.

As a result, reinforcement learning problems are highly dynamic. Each action changes the state, which changes the nature of the problem the agent is trying to solve.

🔹 Reinforcement Learning in Python

The implementation of RL in Python often uses the **OpenAI Gym** library, which provides predefined environments for training and evaluation:

- ***gym.make()*:** Instantiates a simulation environment.
- ***render()*:** Visualizes the environment state.

```
# Import the Library
import gym

# Make an environment
env = gym.make("environment_type")

# Render (show) the environment
env.render()
```

These tools allow interaction with simulated environments such as games or virtual physical systems, making it easier to test reinforcement learning agents.

☑Takeaways

☑ Reinforcement learning enables agents to interact with environments by selecting actions that maximize rewards over time.

☑ Rewards are typically unknown in advance and must be estimated through repeated interaction with the environment.

☑ Reinforcement learning differs from traditional supervised learning because the feedback is not based on labeled outputs, but on delayed and sometimes uncertain rewards.

☑ Recent RL advances include DeepMind's Atari systems and AlphaGo, with growing business applications in recommendation systems and bidding strategies.

☑ Python implementation often uses the OpenAI Gym library to simulate and visualize RL environments.

📌Q-Learning with Keras

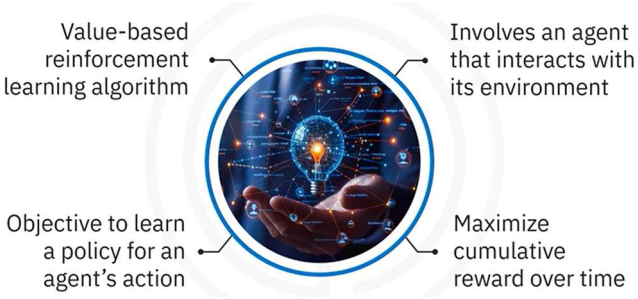
Q-learning is a foundational reinforcement learning algorithm that focuses on training agents to make sequential decisions by maximizing cumulative rewards.

🔹 Understanding Q-Learning

Reinforcement learning is defined as a powerful paradigm in machine learning focused on training agents to make **sequences of decisions** by maximizing a **cumulative reward**.

Q-learning is a **value-based** and **off-policy** reinforcement learning algorithm:

- **Off-policy:** It learns the optimal policy independently of the agent's current behavior
- **Value-based:** The agent learns to estimate the **value** of states or state-action pairs, which represent the expected cumulative reward. These values guide the agent in selecting actions that maximize long-term rewards.



Q-Learning objective:

Learn a policy that tells the agent what action to take under which state to **maximize cumulative future rewards**.

◆ Q-Value Function and Bellman Equation

Q-learning relies on the **Q-value function**, denoted as $\boldsymbol{Q(s,a)}$, which measures the **expected utility** of taking **action** "**a**" in **state** "**s**", and then following the **optimal policy** afterward.

The Q-values are updated **iteratively** using the **Bellman equation**, which incorporates:

- The **immediate reward**
- The **estimated future rewards**

The updated rule for Q-value is given by the **Bellman Equation**:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\,r + \gamma\, max_{a'}\, Q(s',a') - Q(s,a)\,]$$

- **s** → Current state
- **a** → Current action
- **r** → Reward received after taking action a
- **s'** → Resulting state from taking action a
- **a'** → Next action
- **α** (alpha) → **Learning rate**, controlling how much new information overrides old knowledge
- **γ** (gamma) → **Discount factor**, determining the importance of future rewards

Q-learning implementation steps

Q-learning implementation consists of **multiple steps** that are essential for allowing the agent to learn and perform effectively:

- **Initialize the environment and parameters:**

  Define the environment, using a platform like **OpenAI Gym**

  Initialize the **Q-table** with state-action pairs.

  Set hyperparameters:

  - Learning rate α
  - Discount factor γ
  - Exploration rate ε

- **Build the Q-network:**

  Storing Q-table is infeasible when the state space is large or continuous.

  To handle complex environments, we can use a **neural network** function approximator to learn the Q-value function – this approach is known as a **Deep Q-Network (DQN)**.

  Instead of a Q-table, a DQN uses a neural network (called the **Q-network**) to **predict Q(s,a)** for any given **state** "**s**" and **action** "**a**".

  This allows Q-learning to scale to high-dimensional or continuous state spaces by leveraging function approximation

  - Use Keras to construct a Q-Network that approximates the Q-value function.
  - The Q-network replaces the Q-table for large or continuous state spaces.

- **Train the Q-network:**

  Training the Q-network involves letting the agent interact with the environment repeatedly and updating the network's weights based on the Q-learning rule.

  We typically train over many **episodes**. In each episode, the agent starts in an initial state from the environment and then proceeds through a sequence of time-steps until the episode ends (for instance, when a termination condition like a pole falling or time limit is reached).

  Training loop implementation should be as follows:

  - The agent interacts with the environment.
  - Selects actions.
  - Receives rewards.
  - Transitions to new states.
  - Updates Q-values using the Bellman equation.

- **Evaluate the agent:**

  After training the Q-network, we need to **evaluate the agent** to see how well it has learned to solve the task.

  Evaluation is typically done by deploying the agent in the environment **without** any exploratory randomness. In this phase, the agent uses the learned Q-network to select the action with the highest Q-value at each state, reliably exploiting its training.

  Trained agent is tested in the environment, to assess:

  - Overall performance.
  - Ability to maximize rewards.

Q-learning implementation with Keras

- **Initialize the environment and parameters:**

  The environment which the agent will interact, is from **OpenAI Gym**, **CartPole**.

  CartPole is a classic control problem where the objective is to **balance a pole on a moving cart**.

  The setup includes:

  - Initilizing the Q-table with state-action pairs.
  - Defining important parameters that directly influence learning quality and overall agent performance:
    - Learning rate α
    - Discount factor γ
    - Exploration rate ε

  The exploration rate Epsilon is initialized to 1.0 and decays over time to shift the agent's behavior from exploration to exploitation.

  The state size and action size are determined based on the environment's observation and action spaces respectively.

  A Q-table is initialized with zeros, although it is not used directly in the neural network approach.

```python
import gym
import numpy as np

# Initialize the environment
env = gym.make('CartPole-v1')

# Set hyperparameters
alpha = 0.001 # Learning rate
gamma = 0.99  # Discount factor
epsilon = 1.0 # Exploration rate
epsilon_min = 0.01
epsilon_decay = 0.995
episodes = 100

# Initialize the Q-table
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
q_table = np.zeros((state_size, action_size))
```

  Since Q-Tables are impractical in environments with **large or continuous** state spaces, a **Q-network** is used to approximate the Q-value function.

- **Build the Q-network:**

  By using Keras, Q-Network can be build using a few dense layers:

  - The **Input layer size** should match the **state size**.
  - The **output layer size** should match the **action size** (number of possible actions), with **linear activation** function
  - **Hidden layers** can have any architecture, but **typically two or three hidden layers**, with **ReLu activation** function.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

def build_q_network(state_size, action_size):
  model = Sequential()
  model.add(Dense(24, input_dim=state_size, activation='relu'))
  model.add(Dense(24, activation='relu'))
  model.add(Dense(action_size, activation='linear'))


model.compile(loss='mse', optimizer=Adam(learning_rate=alpha))

  return model

# Build the Q-network
q_network = build_q_network(state_size, action_size)
```

- **Train the Q-network:**

  Training involves several steps:

  1. **Initialize the state**

     Reset the environment to get the initial state, agent interacts with that state for a given number of steps.

  2. **Select an action**

     Use an **epsilon-greedy policy** to balance exploration and exploitation:

     - **With probability ε**: choose a random action (exploration)
     - **With probability 1 – ε**: choose the action with the highest predicted Q-value (exploitation)

  3. **Take the action**

     Execute the selected action in the environment to receive the next state and reward

  4. **Update the Q-values**

     Use the **Bellman equation** to compute the target Q-value

     Train the Q-network to **minimize the difference** between the predicted and target Q-values

5. **Repeat**

   Continue the process until the agent reaches a **terminal state** or achieves the **goal**

   Over multiple episodes, **gradually reduce** the exploration rate ε to shift from **exploration** to **exploitation**.

```
episodes = 100
for episode in range(episodes):
    state, info = env.reset()
    state = np.reshape(state, [1, state_size])
    total_reward = 0

    for time in range(500):
        if np.random.rand() <= epsilon:
            action = np.random.choice(action_size)
        else:
            q_values = q_network.predict(state)
            action = np.argmax(q_values[0])

        next_state, reward, done, trunc, _ = env.step(action)
        next_state = np.reshape(next_state, [1, state_size])
        total_reward += reward

        if done: reward = -10

        q_target = q_network.predict(state)
        q_target[0][action] = reward + gamma *
                              np.amax(q_network.predict(next_state)[0])
        q_network.fit(state, q_target, epochs=1, verbose=0)
        state = next_state

        if done:
            print(f"Episode: {episode+1}/{episodes}, Score:
                  {total_reward}")
            break
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay
```

- **Evaluate Trained Agent:**

  After training is complete the agent is **evaluated** by letting it interact with the environment using the **learned policy**

  During evaluation, the agent chooses actions based on the **maximum Q-values,** exploiting the learned Q-values to maximize rewards.

  The agent's behavior is rendered, to measure performance based on the total rewards accumulated over several episodes.

  The evaluation confirms whether the agent successfully learned an **effective policy** for solving the task.

```
for episode in range(10):
  state, info = env.reset()
  state = np.reshape(state, [1, state_size])
  total_reward = 0

  for time in range(500):
    env.render()
    q_values = q_network.predict(state)
    action = np.argmax(q_values[0])
    next_state, reward, done, trunc, _ = env.step(action)
    next_state = np.reshape(next_state, [1, state_size])
    total_reward += reward
    state = next_state

    if done:
      print(f"Episode: {episode+1}, Score: {total_reward}")
      break

env.close()
```

☑Takeaways

☑ Reinforcement learning trains agents to make **sequential decisions** to maximize **cumulative reward**.

☑ Q-learning is a **value-based**, **off-policy** algorithm that estimates the Q-value function Q(s, a).

☑ The **Bellman equation** is used to iteratively update Q-values.

☑ For large state spaces, **Q-networks** implemented with **Keras** replace Q-tables.

☑ The **epsilon-greedy policy** balances exploration and exploitation during training.

☑ After training, the agent is evaluated based on **reward accumulation** over multiple episodes.

☑ The Q-network is trained using **Keras**, **Adam optimizer**, and **mean squared error loss** to approximate Q-values.

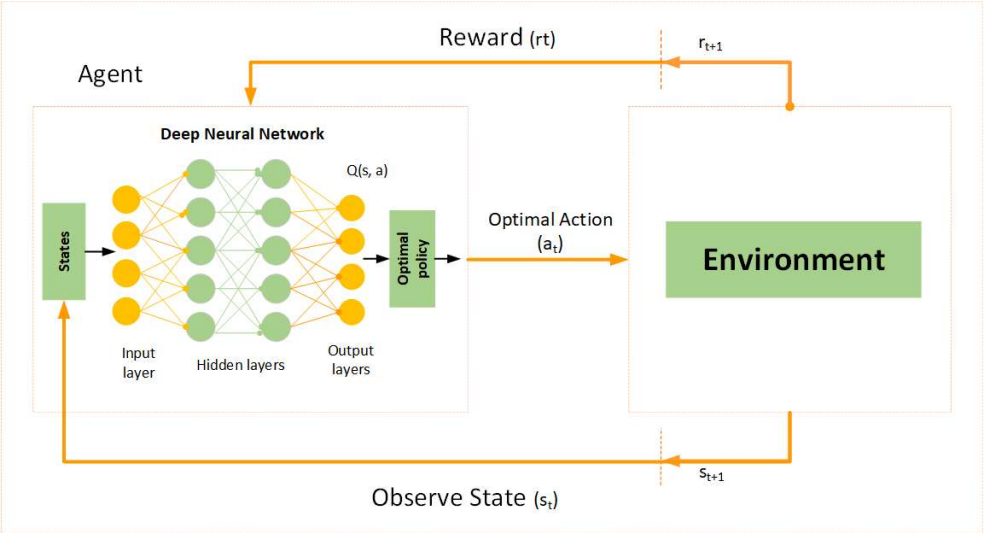
📌Deep Q-Networks (DQNs) with Keras

DQNs leverage deep neural networks to approximate the Q-value function and enable reinforcement learning to scale in complex environments.

The key innovations (experience replay and target networks) help stabilize training and improve performance.

🔹What are Deep Q-Networks

Deep Q-Networks (DQNs) extend the Q-learning algorithm by replacing the Q-table with a deep neural network that approximates the Q-value function.



Traditional Q-learning becomes impractical in environments with large or continuous state spaces due to the exponential growth of the Q-table. DQNs solve this problem by allowing the agent to estimate Q-values through a neural network, enabling scalability in more complex settings.

The success of DQNs lies in stabilizing training using two techniques:

- **Experience replay**
- **Target networks**.

🔹 Key Concepts of DQNs

🧠 Q-Value Function Approximation:

Instead of explicitly storing all state-action pairs, DQNs use a neural network to approximate the function ***Q(s,a)***, where "**s**" represents the state and "**a**" represents the action.

This generalizes Q-values across a broader range of states.

🧠 Experience Replay:

The agent stores its experiences in the form of (`state, action, reward, next_state`) in a replay buffer.

During training, it samples random minibatches from this buffer. This technique breaks the correlation between consecutive experiences and stabilizes learning.

🧠 Target Network:

A secondary neural network, the target network, is introduced to generate more stable Q-value targets.

Unlike the primary Q-network, the target network is updated less frequently, reducing oscillations and divergence during training.

DQNs implementation steps

The process of implementing DQNs follows the structure of Q-learning but introduces new components to improve stability and scalability.

Each step is critical to correctly build and train a deep reinforcement learning agent.

- **Initialize the environment and parameters:**

  Define the learning environment (por example, using OpenAI's Gym platform).

  Set all necessary hyperparameters for training, including learning rate, discount factor, exploration rate, and replay buffer size.

- **Build the Q-Network and Target Network:**

  Two neural networks are created using Keras: the **primary Q-network** and the **target network**.

  Both share the same architecture, but the target network's weights are updated less frequently to serve as a stable reference for computing Q-value target.

- **Implement Experience Replay:**

  A replay buffer is initialized to store agent experiences.

  During training, random minibatches are sampled from the buffer to update the Q-network. This reduces the impact of correlated experiences and improves learning efficiency.

- **Train the Q-Network:**

  The Bellman equation is applied to iteratively update Q-values, leveraging outputs from the target network as stable references.

  The primary Q-network is trained using gradients computed from the loss between predicted Q-values and target Q-values.

- **Evaluate the Agent:**

  After training, the agent interacts with the environment using the learned policy.

  Its performance is measured based on cumulative rewards across multiple episode

DQNs Code Implementation Workflow with Keras

This section outlines the practical implementation of DQNs using the CartPole environment and Keras, including network architecture, training loops, and evaluation setup.

- **Environment and Hyperparameters Initialization**:

  The **CartPole** environment is initialized using `gym.make`.

  Key hyperparameters such as learning rate, discount factor, exploration rate (ε), and batch size for experience replay are defined.

```python
import gym
import numpy as np
from collections import deque

# Initialize the environment
env = gym.make('CartPole-v1')

# Set hyperparameters
alpha = 0.001 # Learning rate
gamma = 0.99  # Discount factor
epsilon = 1.0 # Exploration rate
epsilon_min = 0.01
epsilon_decay = 0.995
episodes = 1000
batch_size = 64
memory_size = 2000

# Initialize replay buffer
memory = deque(maxlen=memory_size)

# Get state and action sizes
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
```

- **Replay Buffer Initialization:**

  A double-ended queue with a fixed size is created to store the agent's experiences (`state, action, reward, next_state, done`) over time.

  Deque, is a data structure that allows efficient insertion and deletion of elements from both its front, head, and back, tail.

- **Q-Network and Target Network Creation:**

  Two identical networks are created, **primary Q-Network** and **target network.** These networks are used to predict Q-values and compute targets for training.

  Both networks have the same architecture, with two hidden layers and ReLU activation.

  The target network is periodically updated to match the weights of the primary Q-network. This delay helps prevent instability during learning.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

def build_q_network(state_size, action_size):
  model = Sequential()
  model.add(Dense(24, input_dim=state_size, activation='relu'))
  model.add(Dense(24, activation='relu'))
  model.add(Dense(action_size, activation='linear'))

model.compile(loss='mse', optimizer=Adam(learning_rate=alpha))
  return model

# Build the Q-Network and target network
q_network = build_q_network(state_size, action_size)
target_network = build_q_network(state_size, action_size)
target_network.set_weights(q_network.get_weights())
```

- **Experience Storage and Sampling:**

  A **`remember()`** function stores (`state, action, reward, next_state, done`) **tuples** in the replay buffer.

  The **`replay()`** function samples minibatches randomly to break experience correlation between consecutives experiences and improve convergence.

```
def remember(state, action, reward, next_state, done):
  memory.append((state, action, reward, next_state, done))

def replay(batch_size):
  minibatch = np.random.choice(len(memory), batch_size, replace=False)
  for index in minibatch:
    state, action, reward, next_state, done = memory[index]
    target = q_network.predict(state)
    if done:
      target[0][action] = reward
    else:
      t = target_network.predict(next_state)
      target[0][action] = reward + gamma * np.amax(t)
    q_network.fit(state, target, epochs=1, verbose=0)
```

- **Training Loop and Policy:**

  The training loop runs for a defined number of episodes.

  At each step, the agent selects actions using an ε-greedy policy (random action with probability ε, or greedy action otherwise).

  After each episode, the ε value decays gradually to promote exploitation over exploration.

  For each sampled experience, the Bellman equation is applied to compute the target Q-value. The Q-network is trained to minimize the difference between predicted and target values.

Target networks weights are periodically updated to match the primary networks weights.

```
# Main training loop
for episode in range(episodes):
    state, info = env.reset()
    state = np.reshape(state, [1, state_size])
    total_reward = 0

    for time in range(500):
        if np.random.rand() <= epsilon:
            action = np.random.choice(action_size)
        else:
            q_values = q_network.predict(state, verbose=0)
            action = np.argmax(q_values[0])

        next_state, reward, done, truncated, info = env.step(action)
        next_state = np.reshape(next_state, [1, state_size])
        total_reward += reward

        if done:
            reward = -10
            remember(state, action, reward, next_state, done)
            break

        state = next_state

    if len(memory) > batch_size:
        replay(batch_size)

    if epsilon > epsilon_min:
            epsilon *= epsilon_decay

        # Update the target network weights every 10 episodes
        if episode % 10 == 0:
            target_network.set_weights(q_network.get_weights())

        print(f"Episode: {episode+1}/{episodes}, Score: {total_reward}")
```

- **Agent Evaluation:**

  After training, the agent is evaluated by running episodes with the learned policy.

  The environment is rendered for visual inspection, and total rewards are printed for each episode.

  During this phase, the agent focuses on exploiting the learned Q-values.

```
for episode in range(10):
    state, info = env.reset()
    state = np.reshape(state, [1, state_size])
    total_reward = 0

    for time in range(500):
        env.render()
        q_values = q_network.predict(state, verbose=0)
        action = np.argmax(q_values[0])
        next_state, reward, done, trunc, _ = env.step(action)
        next_state = np.reshape(next_state, [1, state_size])
        total_reward += reward
        state = next_state
        if done:
            print(f"Episode: {episode+1}, Score: {total_reward}")
            break
env.close()
```

☑Takeaways

☑ Deep Q-Networks (DQNs) extend Q-learning by using neural networks to approximate Q-values, enabling scalability to large or continuous state spaces.

☑ DQNs use experience replay and target networks to stabilize training and improve learning performance.

☑ The DQN implementation includes initializing the environment, defining networks, storing and sampling experience, training with the Bellman equation, synchronizing target networks, and evaluating the agent with the learned policy.