# Module 5

**Advanced Keras Techniques**

**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.

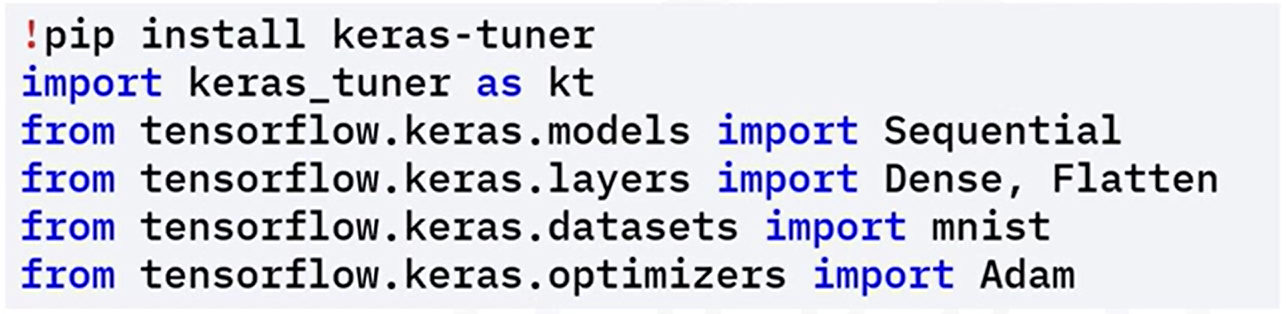
These algorithms allow for efficient and automated exploration of possible configurations.

### 🔹 Hyperparameter search implementation.

ℹ️ **Setting Up the Environment:**

This step prepares the environment for model definition and tuning.

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



ℹ️ **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.



**🔧** **Implementation:**

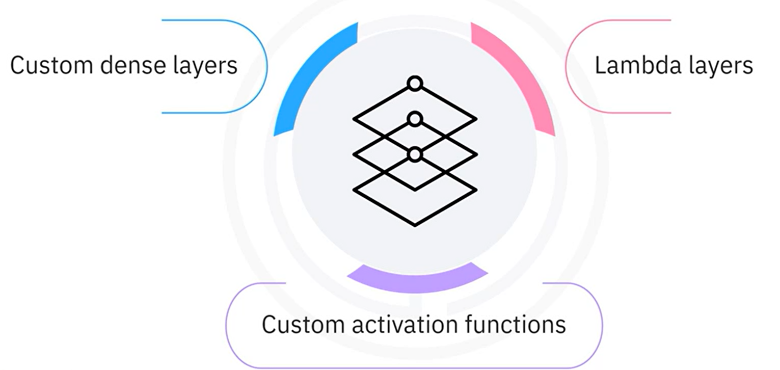
* Start by defining a simple neural network model.
* Set up the optimizer and the loss function.
* Use a training loop that iterates over the dataset for a specified number of epochs.
* Within each training step, a gradient tape is used to record operations for automatic differentiation.
* Gradients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

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



### 🔹 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:

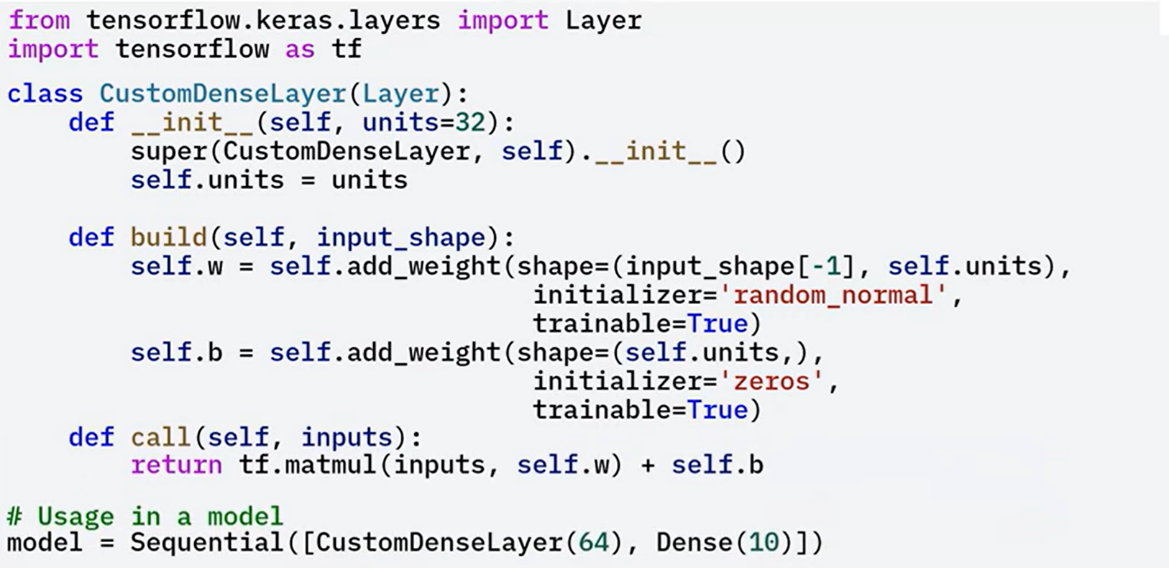
* **Custom Dense Layers**, dense layers created by subclassing the layer class (example provided below).
* **Lambda Layers**, allow definition of simple custom computations inline within the model, typically used for non-trainable logic like reshaping or mathematical operations

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.

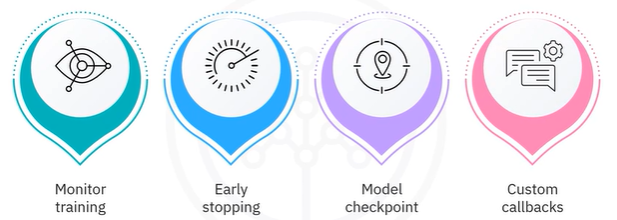


### 🔹 Advanced Callback Functions

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:**

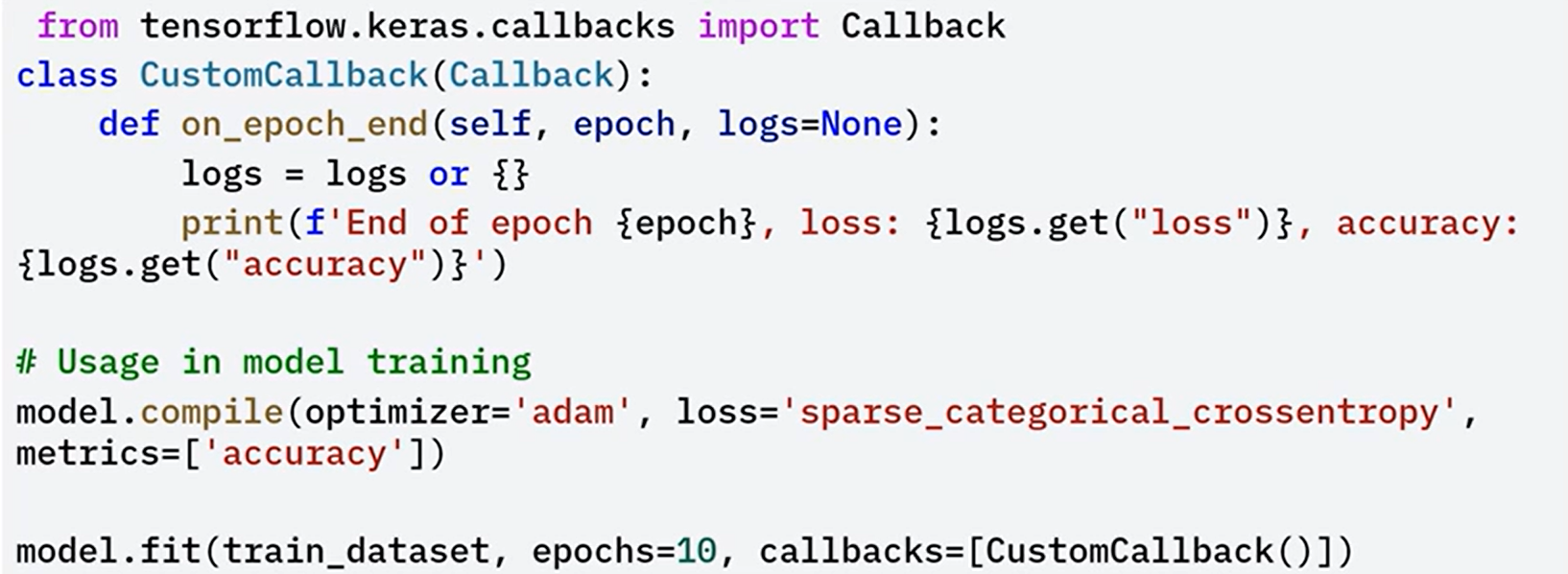
* Monitor training metrics in real-time.
* Implement early stopping when validation loss stops improving.
* Save model checkpoints during training.
* Implement Custom callbacks

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.



### 🔹 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:**



### ✅ 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.

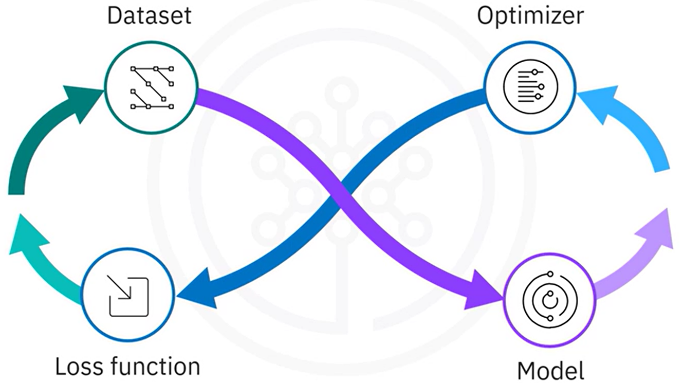
✅ **TensorFlow Model Optimization Toolkit** provides pruning, quantization, and clustering strategies to streamline and deploy efficient models.

## 📌 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.

### 🔹 Model Optimization

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.

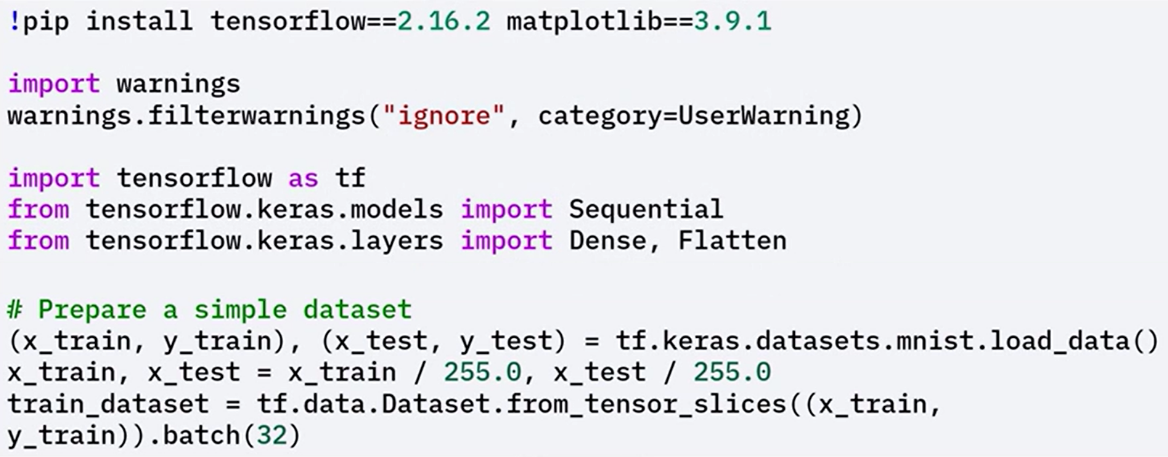
### 🔹 Implementing custom training loop

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.

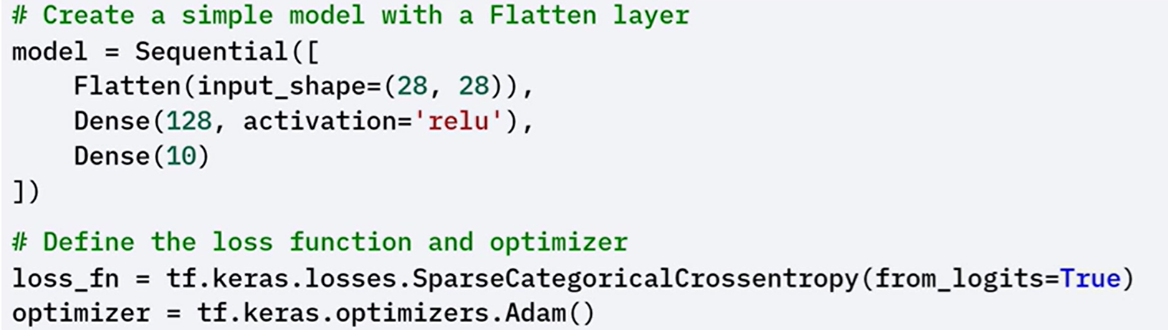


1. **Create model, define loss function and optimizer:**

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.



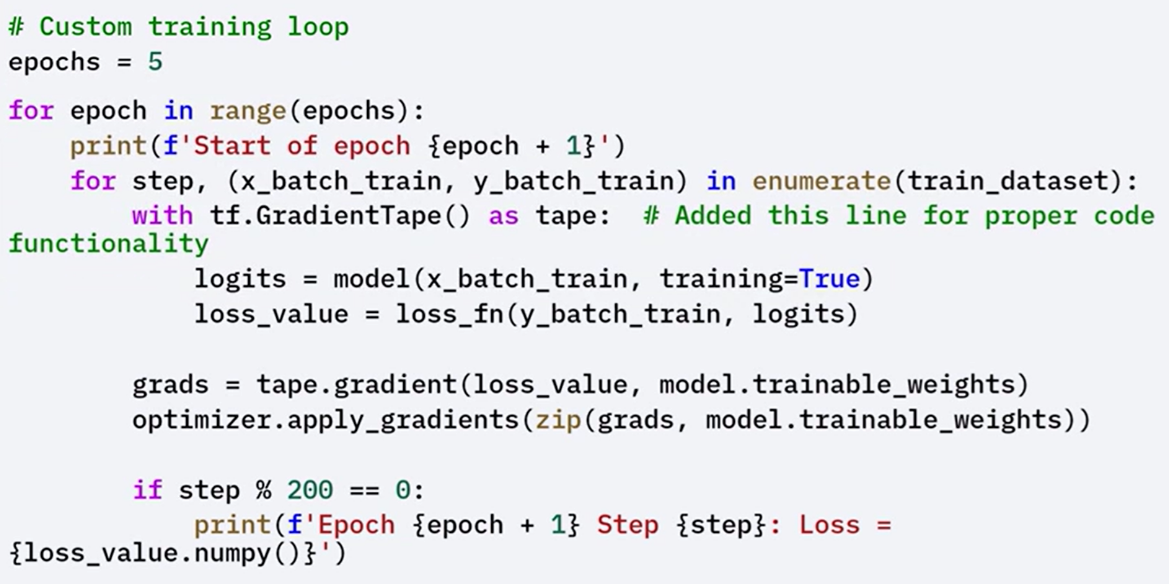
1. **Implementing custom training loop:**

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.



### 🔹 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.