# 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.

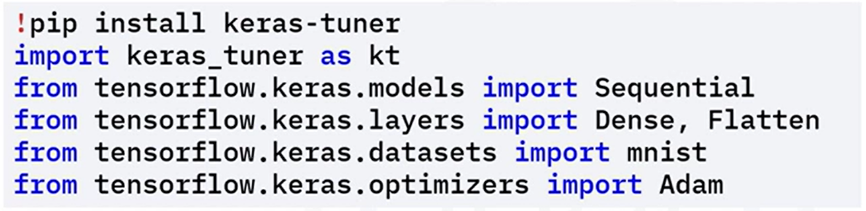
It supports three major search strategies:

* **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.

🔧**Setting Up the Environment:**

This step prepares the environment for model definition and tuning.

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

🔧 **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.

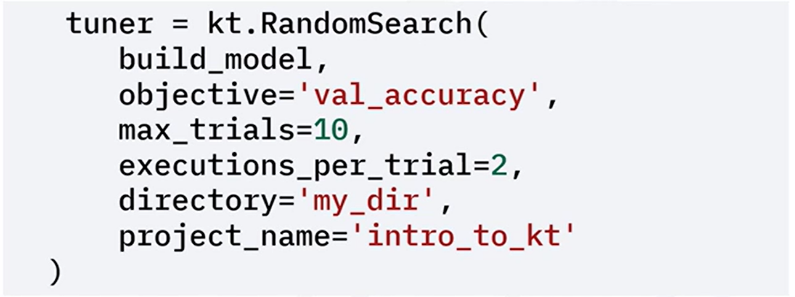


🔧 **Configuring the Tuner:**

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:

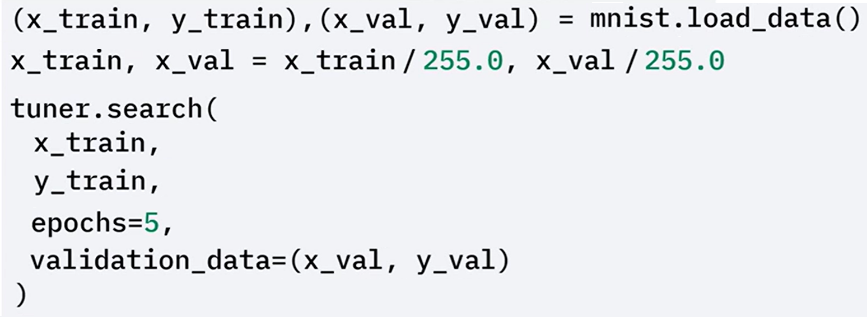
* A search strategy (e.g., RandomSearch).
* The model-building function.
* An optimization objective (e.g., validation accuracy).
* The number of trials (number of different hyperparameters configuration to try) and executions per trial (number of times to run each configuration)
* Directory and project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

🔧 **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.

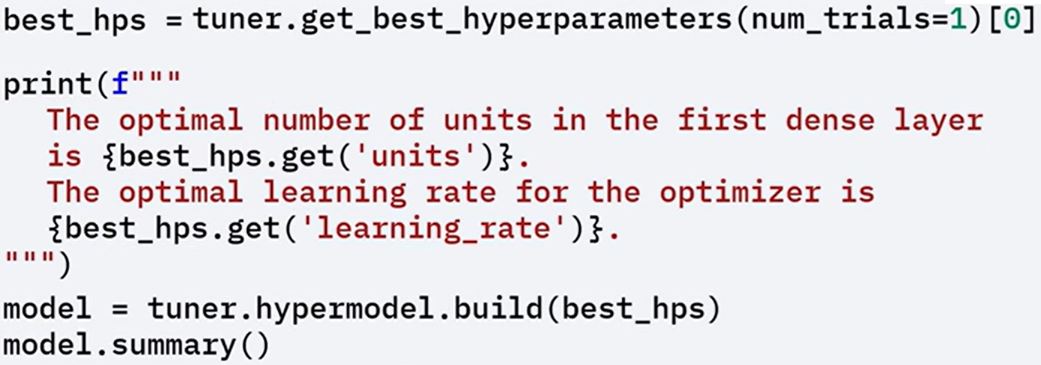


🔧 **Retrieving and Using the Best Hyperparameters:**

After tuning:

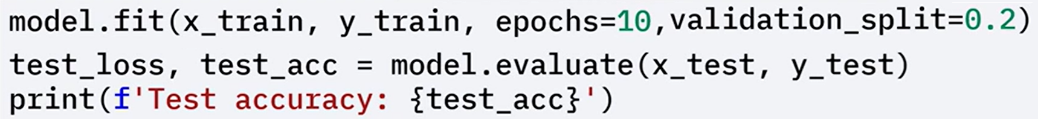
* Use ***get\_best\_hyperparameters()*** to retrieve the optimal values.
* Build a model using these settings.
* Review the model summary for architecture verification.

This results in a final model customized to perform optimally under the chosen hyperparameters.



**🔧** **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.



### ✅ 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.



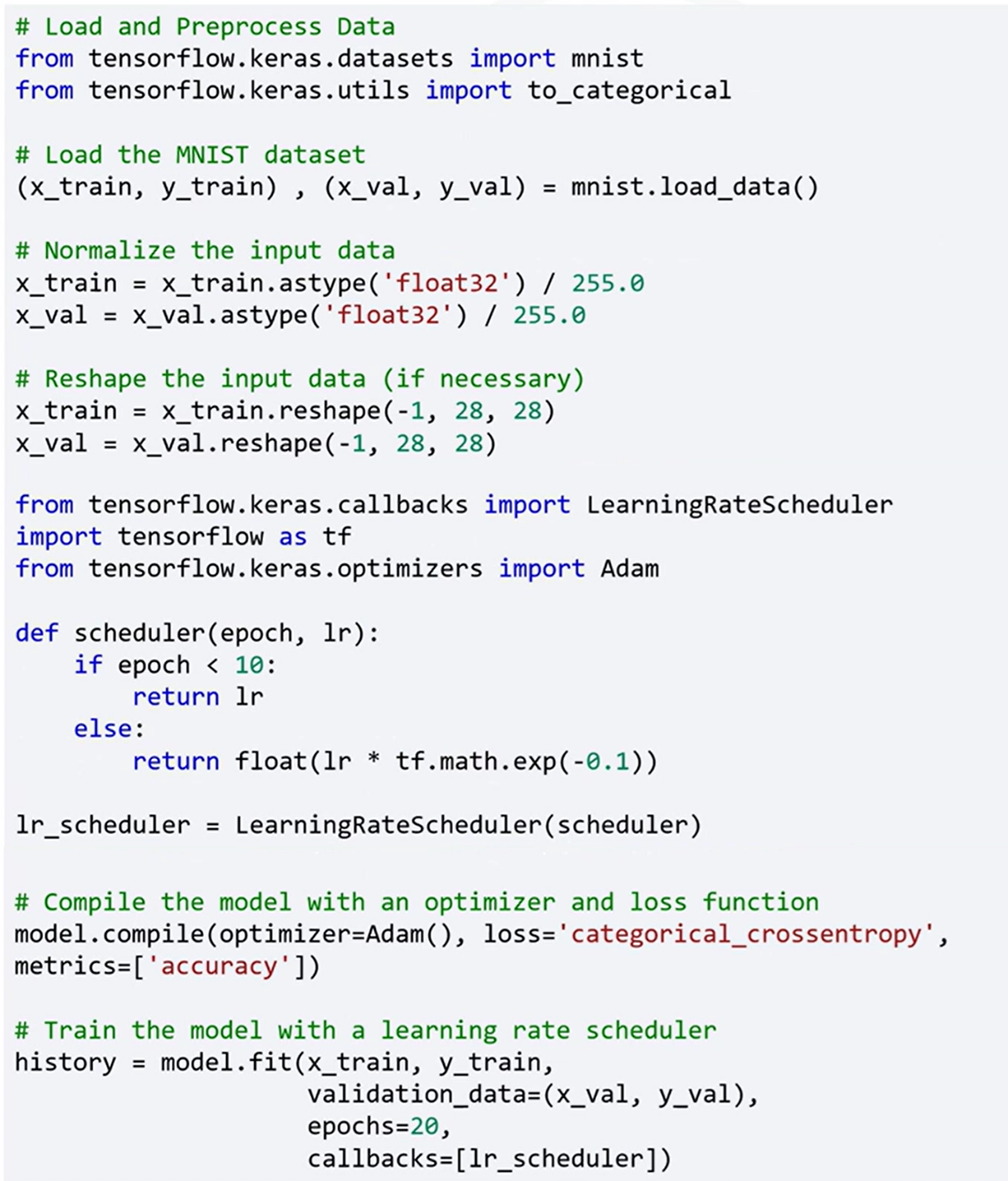
### 🔹 Learning Rate Scheduling

Learning rate scheduling helps dynamically adjust the learning rate during training.

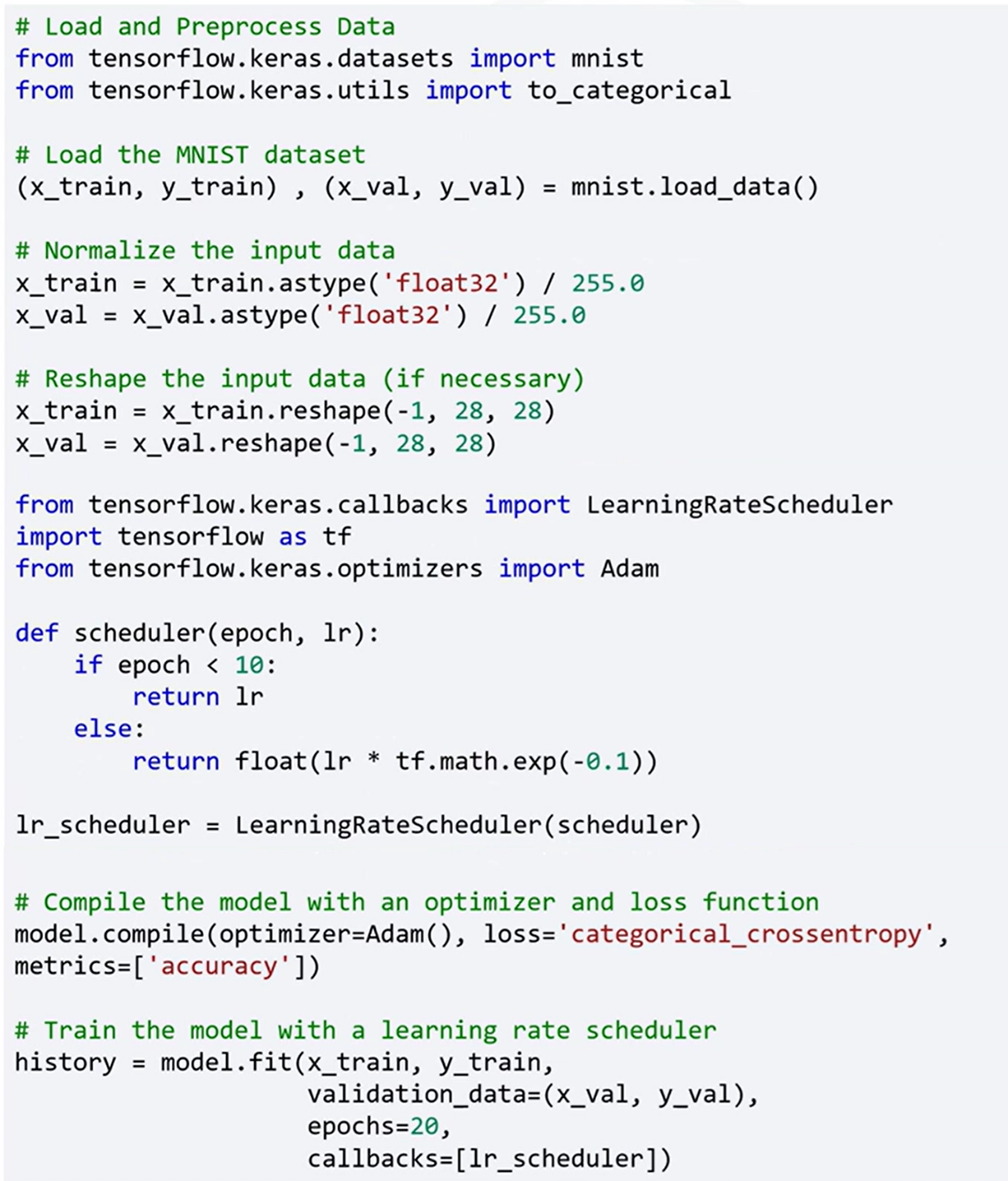
This improves training efficiency by allowing the model to converge faster and fine-tune more precisely in later epochs.

To implement this technique:

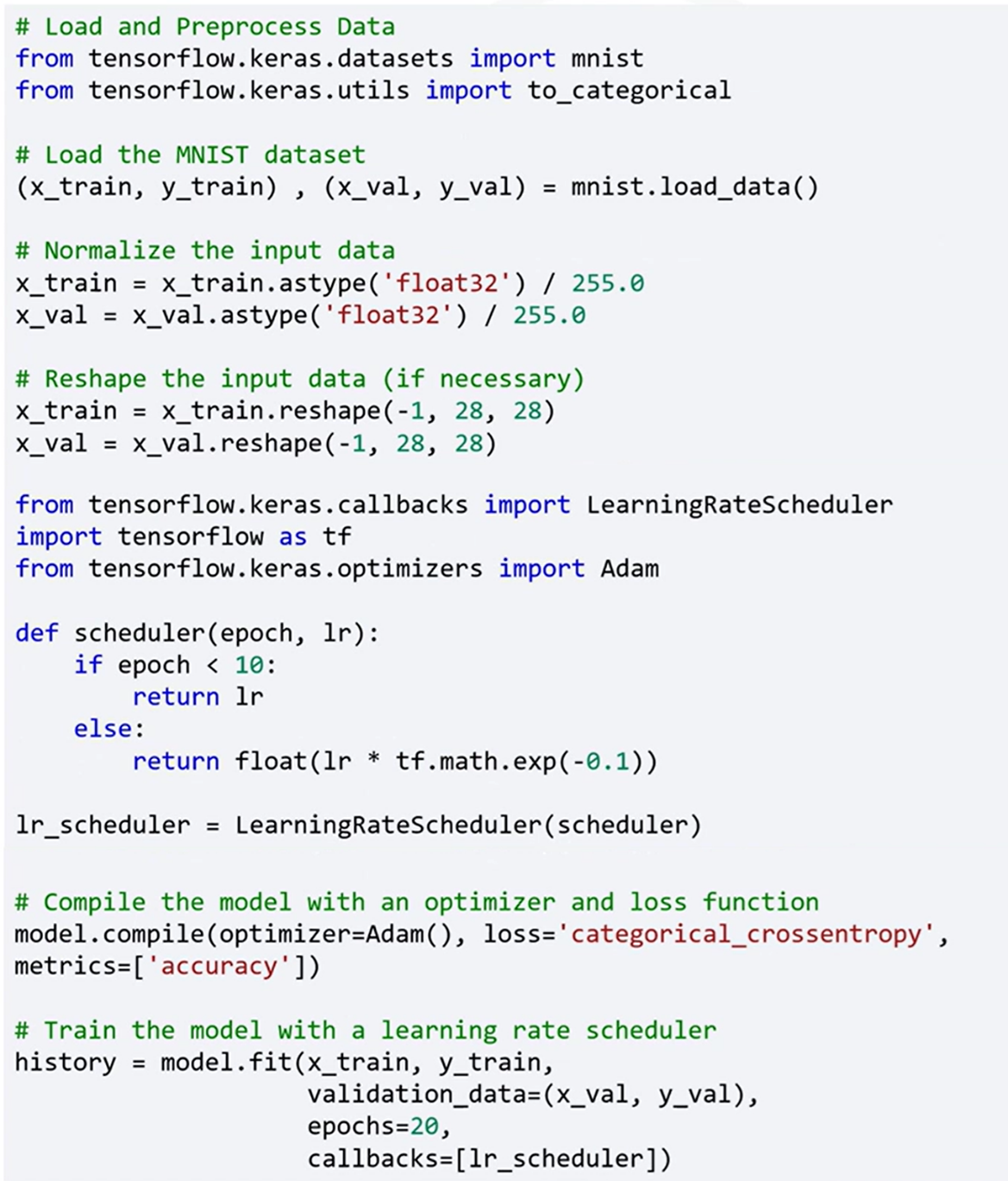
Load and preprocess the data.



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.



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.



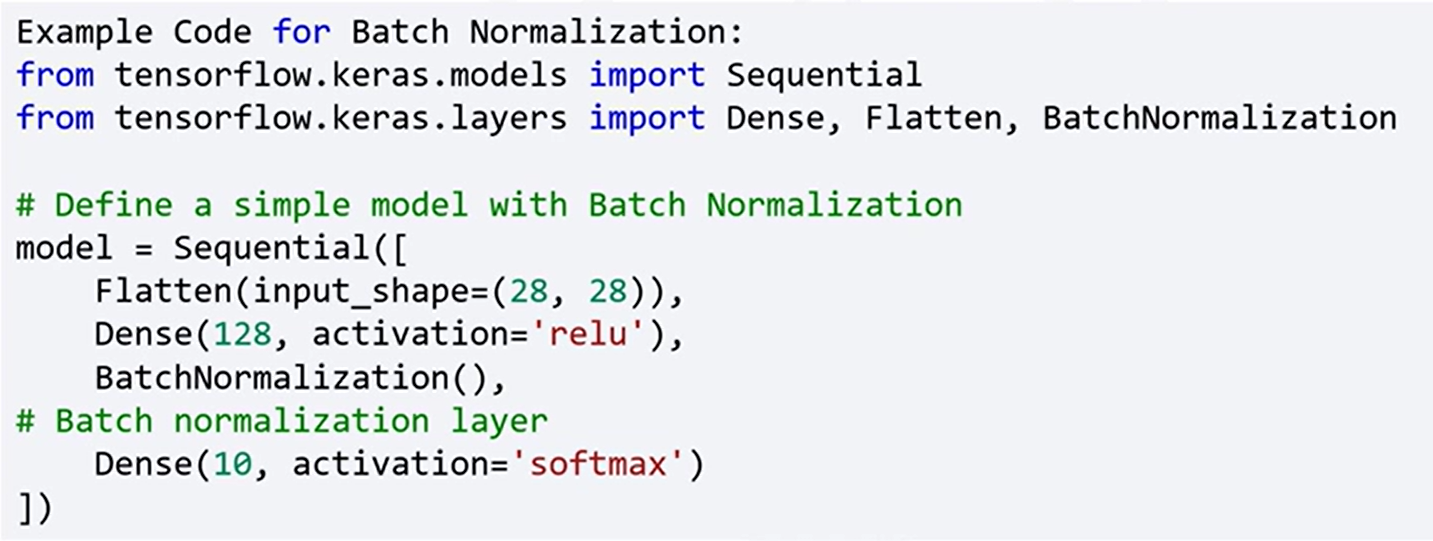
### 🔹 Additional Optimization Techniques

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.

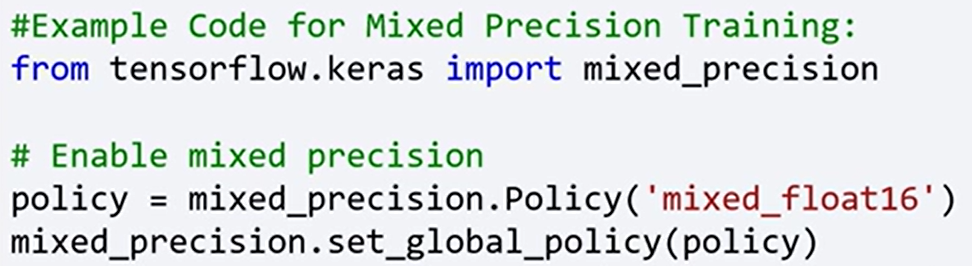
* Helps accelerate training.
* Improves convergence behavior.
* Reduces sensitivity to the learning rate.



🔸 **Mixed Precision Training:**

Uses both 16-bit and 32-bit floating-point data types.

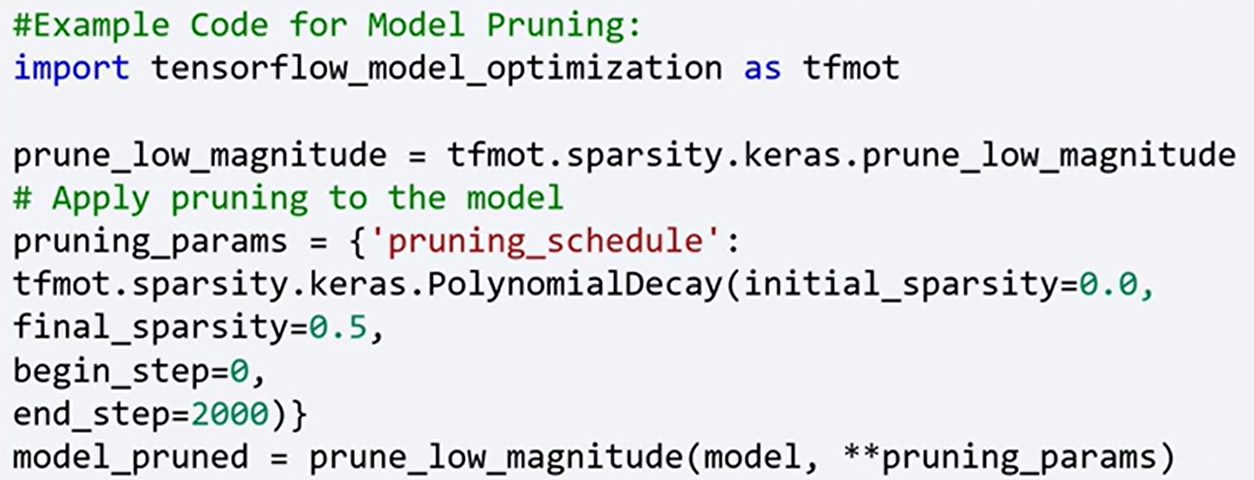
* Allows faster training on modern GPUs.
* Reduces memory usage without sacrificing model accuracy.



🔸 **Model Pruning:**

Reduces the number of parameters by removing less significant connections or neurons from the model.

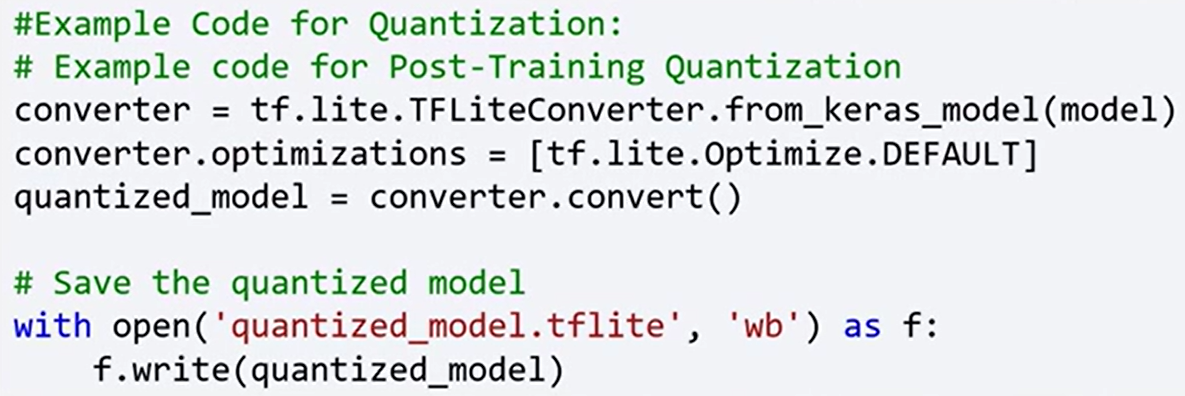
* Maintains accuracy while improving efficiency.
* Reduces computational complexity.



🔸 **Quantization:**

Reduces the precision of the numbers used to represent the model’s weights.

* Reduces model size and speeds up inference.
* Useful for deploying models on edge devices with limited hardware.



### ✅ 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