

Keras and Data Retrieval in TensorFlow 2

This chapter transitions from the low-level TensorFlow API to a higher-level approach for building deep learning models. Previously, models were constructed using fundamental components such as `tf.Variable`, `tf.Tensor`, and `tf.Operation`, along with core mathematical operations like matrix multiplication and convolution. While these low-level tools are powerful, implementing full neural networks directly with them often leads to repetitive, time-consuming, and difficult-to-maintain code. To address this issue, TensorFlow provides **Keras**, a high-level submodule designed to abstract these repetitive building blocks and simplify model development.

Keras is integrated directly into TensorFlow and offers a more intuitive interface for creating machine learning models. By hiding low-level implementation details, it allows developers to focus on model design rather than boilerplate operations. This chapter emphasizes that Keras provides multiple APIs, each suited to different levels of model complexity, enabling developers to choose the most appropriate abstraction for their specific problem.

In addition to model construction, the chapter highlights another essential aspect of machine learning workflows: **data retrieval and preparation**. Before training a model, data must often be loaded from disk or the web, then cleaned and processed. TensorFlow offers dedicated tools such as the `tf.data` API and the `tensorflow-datasets` library to streamline these tasks, making it easier to read, transform, and feed data efficiently into models.

3.1 Keras Model-Building APIs

To motivate the choice of Keras APIs, the chapter introduces a practical scenario: developing a flower species classifier during a hackathon. The goal is to compare multiple multilayer perceptron models trained on a flower measurement dataset. These models vary in complexity, ranging from a simple baseline model that uses raw features, to more advanced variants that incorporate principal components or unconventional hidden-layer computations. Because these models differ structurally, selecting the appropriate Keras API becomes crucial for rapid and effective development.

Keras originated as a standalone high-level library capable of running on multiple backends, such as TensorFlow and Theano. Its primary objective was to simplify neural network construction by abstracting low-level operations behind a clean and expressive interface. Since TensorFlow 1.4, Keras has been fully integrated into TensorFlow and can be accessed via `tensorflow.keras`.

Keras provides **three main model-building APIs**, each offering a different balance between simplicity and flexibility. The **Sequential API** is the most straightforward option, allowing developers to stack layers linearly from a single input to a single output. While easy to use, it is

highly restrictive and unsuitable for more complex architectures. The **Functional API** requires more effort but offers significantly greater flexibility, supporting models with multiple inputs, multiple outputs, and non-linear layer connections. The **Sub-classing API** is the most advanced and flexible approach, where models or layers are defined as custom Python classes using low-level TensorFlow operations, providing maximum control at the cost of increased complexity.

The chapter concludes this section by noting that these APIs will be explored in greater depth later, while **Figure 3.1** provides a high-level comparison of the Sequential, Functional, and Sub-classing APIs to illustrate their key differences.

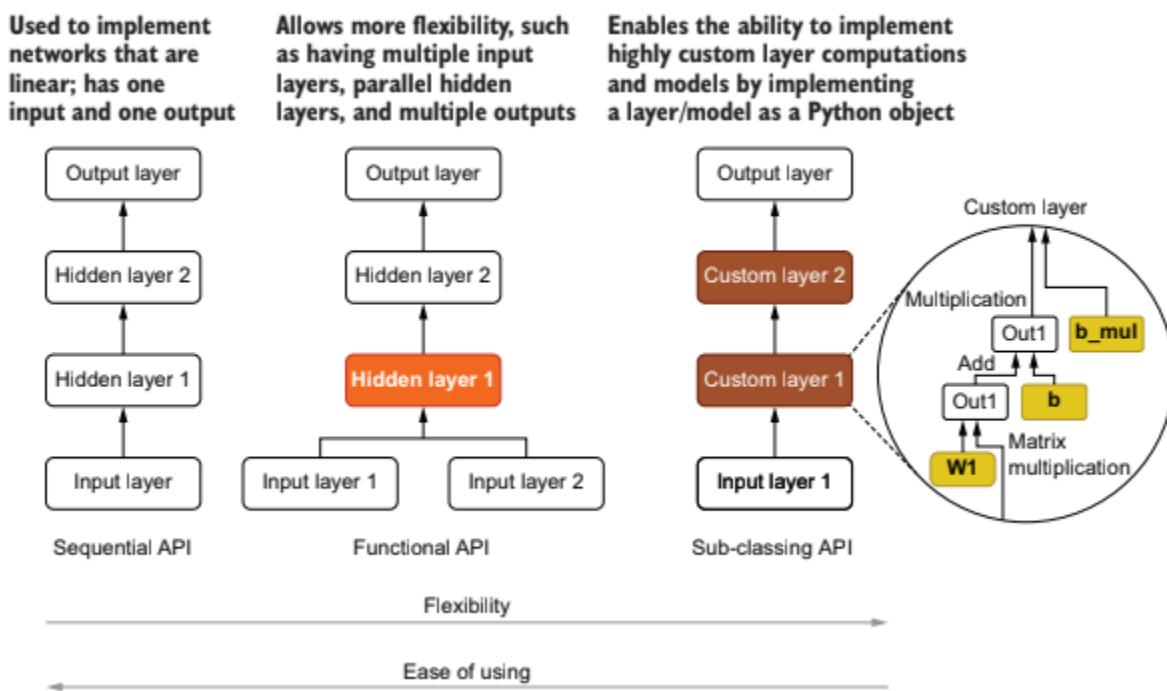


Figure 3.1 Sequential, functional, and sub-classing APIs in comparison

3.1.1 Introducing the Data Set

This section introduces the **Iris dataset**, a widely used benchmark dataset in machine learning that contains measurements of Iris flowers. Each data instance records four numerical features—**sepal length**, **sepal width**, **petal length**, and **petal width**—and is associated with one of three flower species: *Iris-setosa*, *Iris-versicolor*, or *Iris-virginica*. As a result, each input sample has four features and belongs to one of three possible classes, making the dataset suitable for multiclass classification experiments.

The workflow begins by preparing the development environment and opening the provided Jupyter notebook for this chapter. Several libraries are required: requests for downloading the dataset from the web, pandas for data manipulation, and tensorflow for later model-related operations. The dataset is downloaded directly from the UCI Machine Learning Repository and

saved locally as a file, after which it is loaded into a pandas DataFrame using the `read_csv()` function. At this stage, the data is organized into rows and columns, where each row corresponds to a flower sample and each column represents either a feature or the class label.

To improve readability and usability, the DataFrame columns are renamed using meaningful feature names derived from the dataset documentation. Additionally, the original string-based class labels are mapped to integer values—0, 1, and 2—representing the three Iris species. This transformation simplifies subsequent processing and model training. After these adjustments, the dataset consists of numerical feature columns and a numerical class label column.

Before training any model, the data undergoes further preprocessing. The dataset is first **shuffled**, ensuring that samples from different classes are mixed rather than grouped by species. This step is crucial because training on ordered data can bias the learning process, while shuffled data helps each training batch contain a representative mix of classes. The features are then separated into an input matrix x , and the labels are separated into y . The feature values are **centered by subtracting the mean of each feature**, a common preprocessing technique that often improves model convergence and performance.

Finally, the class labels are transformed using **one-hot encoding**, converting each class index into a binary vector of length three. For example, class 0 becomes [1, 0, 0], class 1 becomes [0, 1, 0], and class 2 becomes [0, 0, 1]. This encoding is especially important for training neural networks with categorical outputs, as it allows the model to represent class probabilities more effectively.

3.1.2 The Sequential API

This section demonstrates how to implement **Model A**, a baseline neural network that predicts Iris flower species directly from the provided features. Because the model has a simple, linear structure—one input and one output with layers stacked in order—the **Keras Sequential API** is the most appropriate choice. This API is the easiest to use but is limited to strictly sequential architectures.

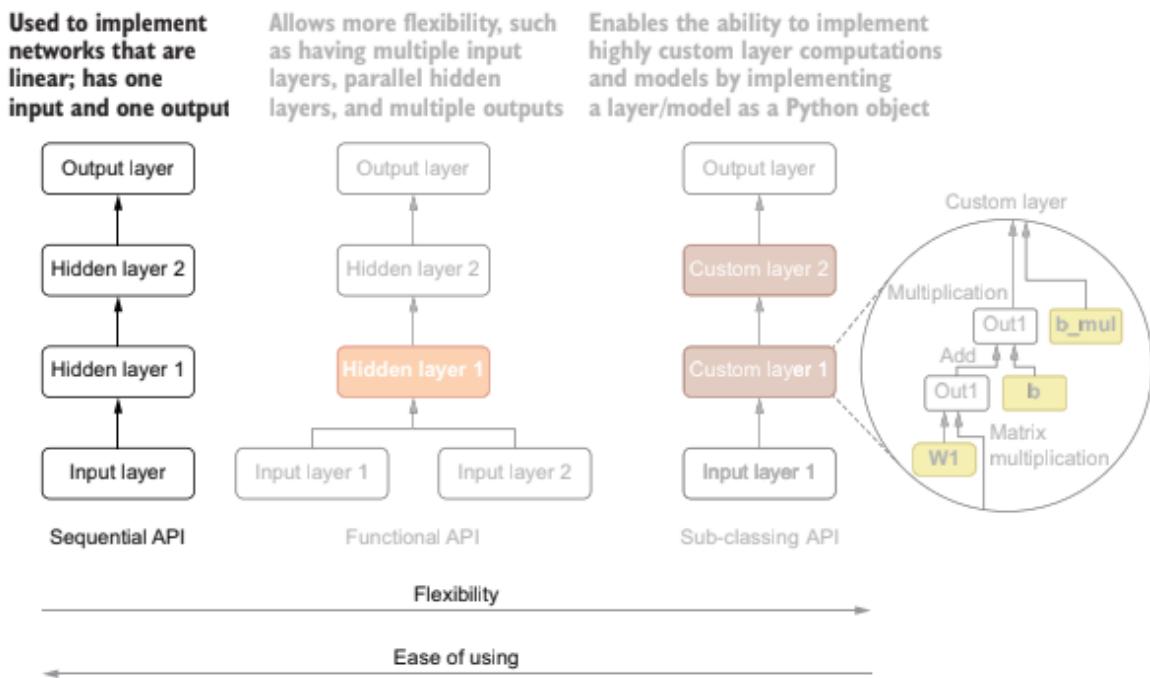


Figure 3.2 The Sequential API compared to other APIs (grayed out)

The model architecture consists of four main components: an input layer with four nodes corresponding to the four flower features, two hidden layers with 32 and 16 neurons respectively, and an output layer with three neurons representing the three flower classes. The number of neurons in each layer is a **hyperparameter**, chosen arbitrarily in this example, though in practice it should be optimized using systematic hyperparameter search methods.

Using the Sequential API, the model is defined by passing a list of layers to the Sequential object. Each layer is a Dense layer, which represents a fully connected neural network layer. A Dense layer performs the computation

$h = \text{activation}(xW + b)$, where W and b are trainable parameters. The first Dense layer explicitly specifies `input_shape=(4,)`, indicating that each input sample contains four features. Keras automatically adds an unspecified batch dimension (`None`), allowing the model to process any batch size during training or inference.

The hidden layers use the **ReLU (Rectified Linear Unit)** activation function, defined as $y = \max(0, x)$. ReLU is widely used in feed-forward networks due to its simplicity and effectiveness. The output layer uses the **softmax** activation function, which converts raw output scores into a valid probability distribution across the three classes. This allows the model to interpret outputs as class probabilities.

After defining the model, a critical step called **model compilation** is performed. Compilation specifies the loss function, optimizer, and evaluation metrics. For this multiclass classification problem, the model uses **categorical cross-entropy** as the loss function and the **Adam**

optimizer, which is known for strong performance across many tasks. Accuracy is included as a metric to monitor training progress.

The model structure and parameter counts can be inspected using `model.summary()`, which displays each layer, its output shape, and the number of trainable parameters. In this case, the model has three Dense layers and a total of 739 trainable parameters.

Training is performed using the `fit()` method, which takes the input features, one-hot encoded labels, batch size, and number of epochs. The batch size controls how many samples are processed at once, while epochs determine how many times the entire dataset is passed through the model. During training, the loss decreases and accuracy steadily improves, reaching approximately 74% after 25 epochs.

Although the training results indicate reasonable performance, the section emphasizes that **training accuracy alone is insufficient** to judge model quality. More robust evaluation techniques are required to assess generalization and reliability, which are discussed in later chapters. The section also introduces the concept of **reproducibility**, highlighting the importance of being able to consistently reproduce experimental results in machine learning research.

3.1.3 The Functional API

In this section, **Model B** is introduced, which extends the baseline approach by incorporating **principal components** as an additional set of input features. The motivation behind this design is that principal components, extracted using **Principal Component Analysis (PCA)**, may capture important variance in the data that is not explicitly represented in the original feature space. PCA is a dimensionality reduction technique that projects high-dimensional data into a lower-dimensional space while preserving as much variance as possible. In this model, the network is required to process **two different input feature sets simultaneously**, making it unsuitable for the Sequential API.

Because the Sequential API only supports linear architectures with a single input and a single output, it cannot be used for this scenario. Instead, the **Keras Functional API** is employed, as it supports models with **multiple inputs, parallel layers, and more complex topologies**. In Model B, the raw flower features and the PCA features are processed independently through parallel layers, merged using concatenation, and then passed through additional layers to produce the final classification output, as illustrated in Figure 3.3.

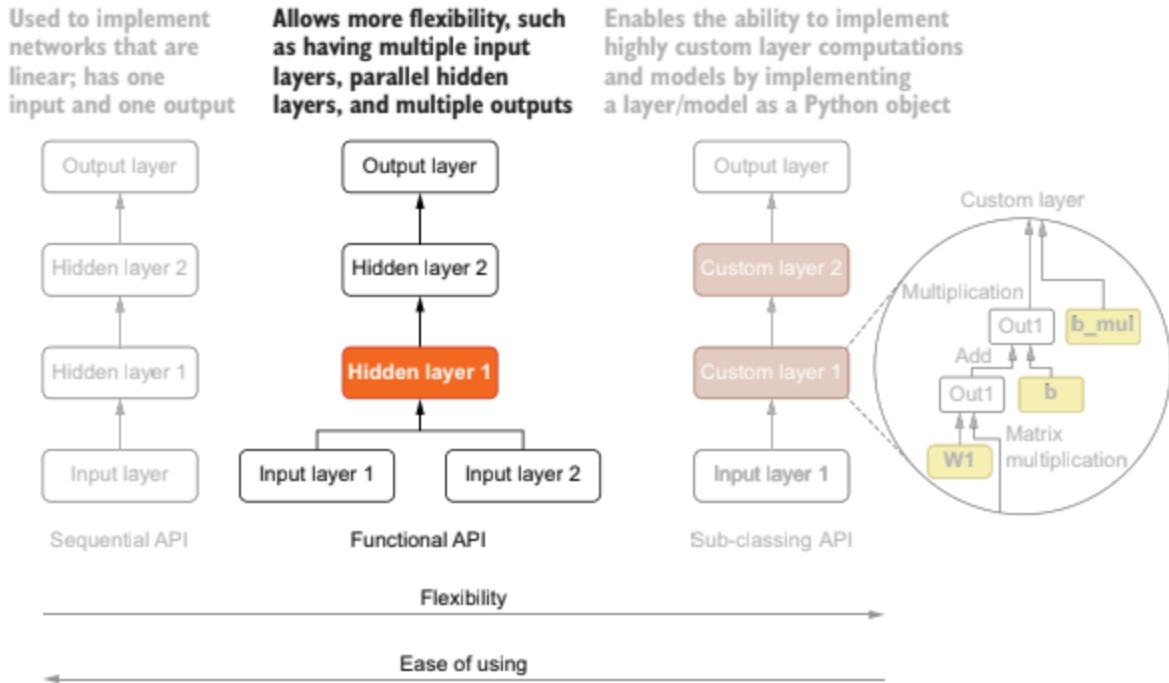


Figure 3.3 The functional API compared to other APIs (grayed out)

Before defining the model, the importance of **reproducibility** in machine learning is emphasized. Neural networks rely heavily on random initialization, which can lead to different results across runs. To ensure consistent behavior, a fixed random seed is set for NumPy, TensorFlow, and Python's random module. By fixing the seed, the same sequence of random numbers is generated in every run, resulting in identical weight and bias initialization and therefore reproducible results.

To build the functional model, the required components are first imported:

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

Unlike the Sequential API, the Functional API requires **explicit Input layers**. Two separate input layers are defined: one for the raw Iris features and one for the PCA features.

```
inp1 = Input(shape=(4,))
inp2 = Input(shape=(2,))
```

The first input layer accepts four features corresponding to sepal and petal measurements, while the second input layer accepts two features representing the first two principal components. Each input is then processed independently through its own Dense layer to produce separate hidden representations.

```
out1 = Dense(16, activation='relu')(inp1)
out2 = Dense(16, activation='relu')(inp2)
```

These two hidden representations are then merged using a **Concatenate** layer. The concatenation is performed along the feature axis, combining two tensors of shape [None, 16] into a single tensor of shape [None, 32].

```
out = Concatenate(axis=1)([out1, out2])
```

After concatenation, the model transitions back to a single sequential flow. The combined representation is passed through another hidden Dense layer with ReLU activation, followed by an output layer with three neurons and a softmax activation to generate class probabilities.

```
out = Dense(16, activation='relu')(out)
```

```
out = Dense(3, activation='softmax')(out)
```

At this stage, the layers exist but are not yet connected into a complete model. A **Model object** is created by explicitly specifying the inputs and outputs, after which the model is compiled using categorical cross-entropy as the loss function, the Adam optimizer, and accuracy as the evaluation metric.

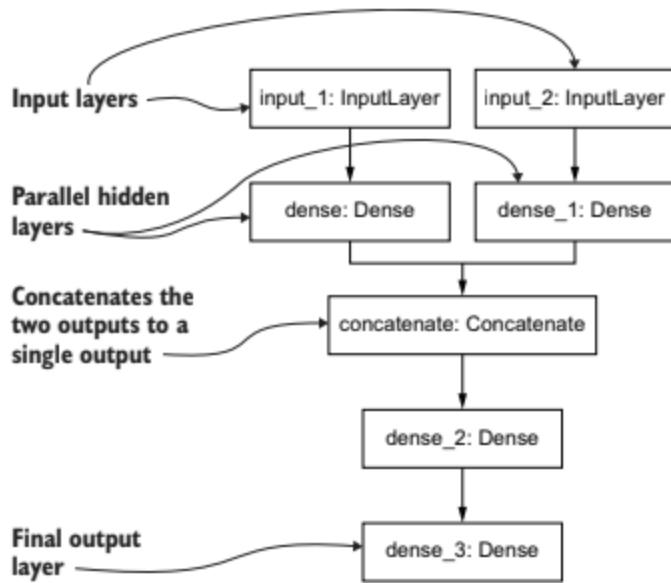
```
model = Model(inputs=[inp1, inp2], outputs=out)
model.compile(loss='categorical_crossentropy',
               optimizer='adam',
               metrics=['acc'])
```

The complete implementation of Model B using the Functional API is shown in Listing 3.2. Once defined, the model summary can be printed using `model.summary()`. Although the model contains parallel paths, the textual summary does not clearly convey this structure, making it difficult to visually understand the architecture from the summary alone.

To address this limitation, Keras provides a visualization utility that allows the model to be displayed as a network diagram:

```
tf.keras.utils.plot_model(model)
```

This visualization clearly shows the two parallel input layers, their respective hidden layers, the concatenation step, and the final output layer, as illustrated in Figure 3.4. The diagram can be saved to a file or enhanced to display tensor shapes by enabling the `show_shapes=True` parameter, which produces the visualization shown in Figure 3.5.



// Figure 3.4 An illustration of the model we created with the functional API. You can see the parallel input layers and hidden layers at the top. The final output layer is at the bottom. //

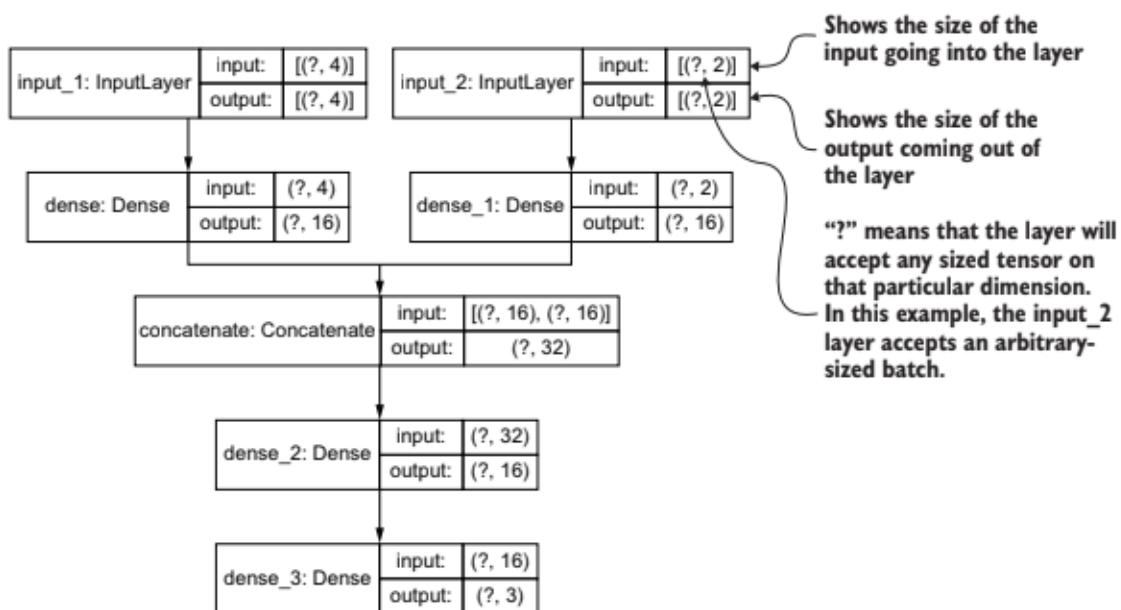


Figure 3.5 Keras model plot with show_shapes=True

Finally, the PCA features required for the second input are computed using the **scikit-learn** implementation of PCA. Only the first two principal components are retained, and a fixed random seed is used for consistency.

```
from sklearn.decomposition import PCA  
  
pca_model = PCA(n_components=2, random_state=4321)  
  
x_pca = pca_model.fit_transform(x)
```

With both the original features and PCA features prepared, the model can be trained by passing both inputs simultaneously to the fit() function:

```
model.fit([x, x_pca], y, batch_size=64, epochs=10)
```

This completes the implementation and training of Model B using the Keras Functional API, demonstrating how more complex architectures with multiple inputs can be built and managed effectively in TensorFlow.

3.1.4 The Sub-classing API

After observing that the inclusion of principal components did not significantly improve model performance, a new idea is proposed: modifying the internal computation of a dense layer itself. Normally, a Dense layer computes its output using an additive bias according to the equation $\mathbf{h} = \alpha(\mathbf{xW} + \mathbf{b})$, where α denotes a nonlinear activation. In this model, an additional **multiplicative bias** is introduced so that the computation becomes $\mathbf{h} = \alpha([\mathbf{xW} + \mathbf{b}] \times \mathbf{b_mul})$. Since no standard Keras layer provides this behavior, a more flexible modeling approach is required.

This need leads to the use of the **Keras sub-classing API**, which allows developers to define entirely custom layers or models by writing Python classes. Sub-classing is based on inheritance, where a new layer is created by extending the base Layer class and redefining its internal behavior. Unlike the Sequential and Functional APIs, which focus on connecting predefined layers, the sub-classing API focuses on **explicitly defining computations**, making it suitable for highly customized operations. The relationship between the three APIs is illustrated in Figure 3.6.

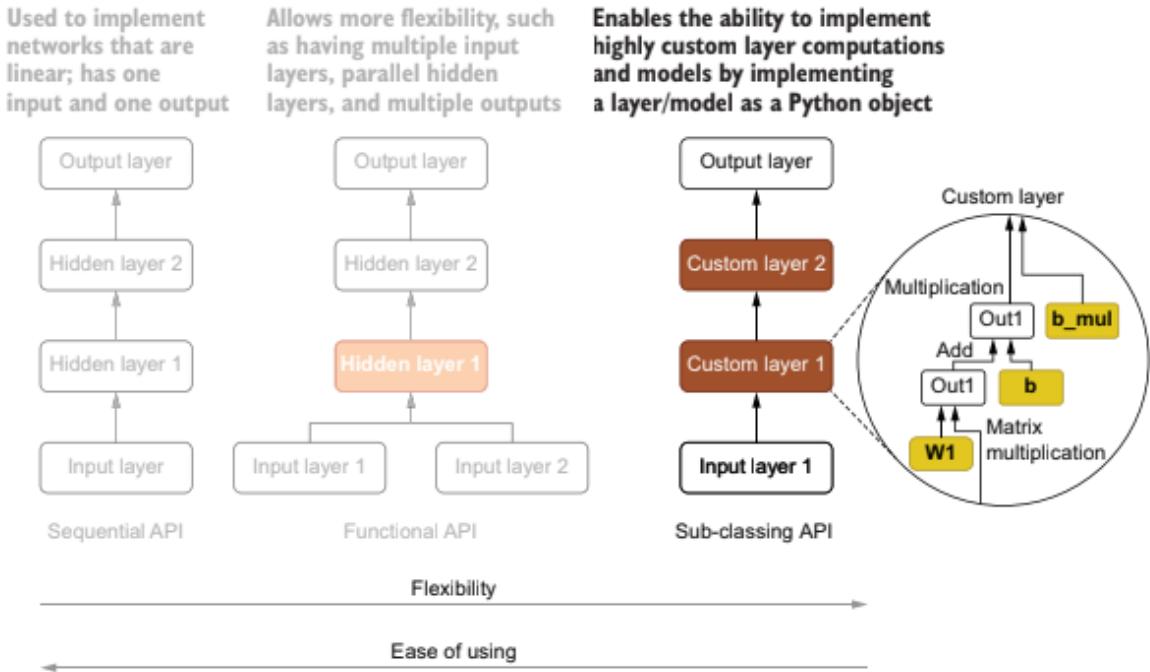


Figure 3.6 Sub-classing API compared to other APIs (grayed out)

When sub-classing a layer, three core methods must be overridden. The `__init__()` method initializes the layer and stores hyperparameters such as the number of units and the activation function. The `build()` method is responsible for creating trainable parameters such as weights and biases once the input shape is known. Finally, the `call()` method defines the forward computation that transforms inputs into outputs during training and inference.

To implement the proposed computation, a custom layer named **MulBiasDense** is created by inheriting from `tensorflow.keras.layers.Layer`. In the initialization step, the number of units and activation function are stored. The `build()` method defines three trainable parameters: the weight matrix W , the additive bias b , and the new multiplicative bias b_mul , all initialized using the popular `glorot_uniform` initializer. These parameters directly correspond to the mathematical formulation of the modified dense layer.

The forward computation is implemented in the `call()` method. First, matrix multiplication between the inputs and weights is performed, followed by the addition of the standard bias. The result is then multiplied element-wise by the multiplicative bias before applying the specified activation function. This explicitly encodes the custom computation

$\mathbf{h} = \mathbf{a}([\mathbf{x}\mathbf{W} + \mathbf{b}] \times \mathbf{b_mul})$, which cannot be achieved using standard Keras layers alone.

Beyond the essential methods, two additional functions are mentioned as important in advanced use cases. The `compute_output_shape()` method may be required when Keras cannot automatically infer output dimensions due to complex transformations. The `get_config()` method

becomes necessary if the model is to be saved and later restored, as it defines how layer parameters are serialized.

Once the custom layer is defined, it can be seamlessly integrated into a model using the **Functional API**, just like any built-in Keras layer. In this example, an input layer with four features is followed by two stacked MulBiasDense layers with ReLU activation, and finally a standard Dense output layer with softmax activation for classification. The model is then compiled using categorical cross-entropy as the loss function, the Adam optimizer, and accuracy as the evaluation metric.

Although this architectural innovation did not produce significantly better performance in experiments, it successfully demonstrates the **strength of the sub-classing API**: the ability to define and reuse entirely new layer behaviors. The section concludes by emphasizing that understanding when to use each Keras API is a valuable skill, especially under time constraints such as research deadlines.

Table 3.1 summarizes the strengths and weaknesses of the three APIs. The Sequential API is concise and easy to understand but lacks flexibility. The Functional API supports complex architectures but requires careful manual wiring of layers. The Sub-classing API offers maximum flexibility and enables custom computations but demands a deep understanding of TensorFlow internals and can be harder to debug due to its user-defined nature.

Comparison of Pros and Cons of Using Various Keras APIs		
	Pros	Cons
Sequential API	Pros	Models implemented with the Sequential API are easy to understand and are concise.
	Cons	Cannot implement models having complex architectural characteristics such as multiple inputs/outputs.
Functional API	Pros	Can be used to implement models with complex architectural elements such as multiple inputs/outputs.
	Cons	The developer needs to manually connect various layers correctly and create a model.
Sub-classing API	Pros	Can create custom layers and models that are not provided as standard layers.
	Cons	Requires thorough understanding of low-level functionality provided by TensorFlow. Due to the user-defined nature, it can lead to instabilities and difficulties in debugging.

Table 3.1 Pros and cons of using various Keras APIs

3.2.1 tf.data API

An input pipeline for an image classification task typically consists of several sequential steps, starting from reading raw data on disk to producing batched tensors ready for training. The pipeline developed in this section follows a structured flow: reading filenames and labels from a CSV file, loading and preprocessing images, converting labels into one-hot encoded vectors, and finally batching the data. This overall process is conceptually illustrated below.

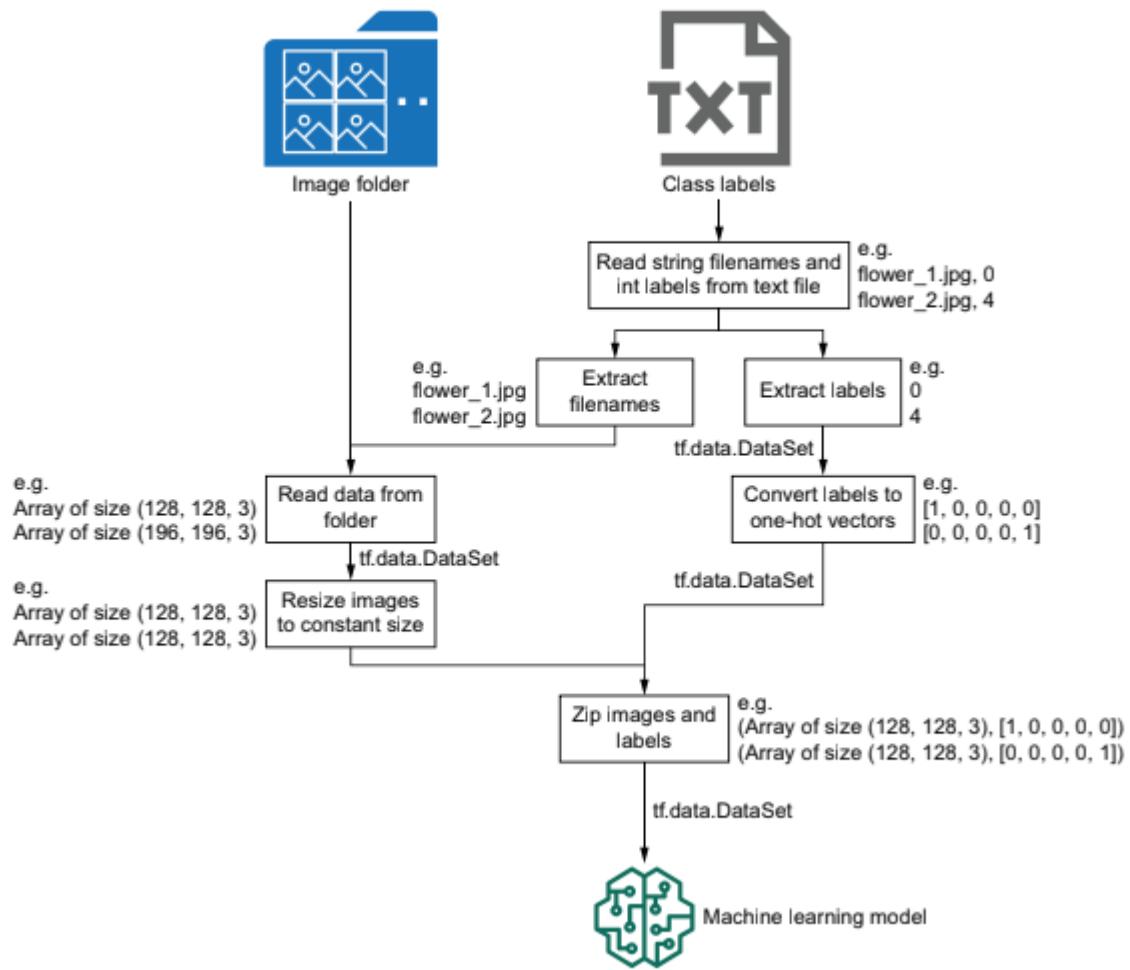


Figure 3.7 The input pipeline that you'll be developing using the tf.data API

The dataset used contains 210 flower images in PNG format and a CSV file mapping filenames to integer labels. The pipeline begins by reading the CSV file using `tf.data.experimental.CsvDataset`, which converts the file into a `tf.data.Dataset` object where each element consists of a filename and its corresponding label. Default values are provided to handle unreadable or corrupted records safely.

Once loaded, the filename and label columns are separated into two independent datasets using the `map()` function. This functional transformation applies a lambda expression to extract either filenames or labels, demonstrating how `tf.data` enables clean, declarative data transformations.

To retrieve the actual image data, filenames are mapped to a custom image-loading function. This function reads each image file from disk, decodes the PNG content into a tensor, converts pixel values to floating-point format in the $[0, 1]$ range, and resizes images to a uniform resolution of 64×64 pixels. This ensures consistent input dimensions for downstream models.

After preprocessing images, labels are transformed into **one-hot encoded vectors**, which is a standard representation for multi-class classification tasks. At this point, images and labels exist as separate datasets, but they must be combined to preserve their correspondence. This is achieved using `tf.data.Dataset.zip()`, which merges them into a single dataset of (image, label) pairs.

The resulting dataset behaves like a Python iterator, allowing iteration with loops or calls to `next()`. Each element consists of a $64 \times 64 \times 3$ image tensor and a one-hot encoded label vector. To improve training performance and prevent learning artifacts caused by ordering, the dataset is shuffled using a buffer. The buffer size controls the degree of randomness and the memory footprint of the pipeline.

Before feeding the data to a model, the dataset is batched. Batching groups multiple samples into a single tensor, which aligns with how deep neural networks process data internally. For example, batching five samples produces a tensor of shape $(5, 64, 64, 3)$ for images and $(5, 10)$ for labels. This batching step finalizes the pipeline and prepares it for efficient training.

The final pipeline combines all these steps—reading, preprocessing, encoding, zipping, shuffling, and batching—into a concise and readable sequence. Although the fully connected models used earlier for the Iris dataset are not suitable for image data, a simple convolutional neural network is provided to demonstrate compatibility with the pipeline. Using this setup, the model can be trained directly with `model.fit(data_ds, epochs=10)`, yielding rapid improvements in accuracy.

The section concludes by highlighting the effectiveness and clarity of the `tf.data` pipeline and raises the question of whether **Keras data generators** could provide an even better solution, setting the stage for further exploration.

3.2.2 Keras DataGenerators

In addition to the `tf.data` API, Keras provides built-in **data generators** as an alternative approach for feeding data into models. These generators offer a simpler and more concise way to load data, although they are generally less flexible and customizable than `tf.data` pipelines. At present, Keras includes two main generators: `ImageDataGenerator` for image data and

`TimeSeriesDataGenerator` for sequential data. This section focuses on using `ImageDataGenerator` to handle image-based datasets.

The `ImageDataGenerator` supports a wide range of parameters, but only a minimal configuration is required to load image data effectively. In this example, image files and their corresponding labels are stored in a directory, with filenames and labels listed in a CSV file. This structure makes the `flow_from_dataframe()` function particularly suitable, as it can directly read image paths and labels from a pandas DataFrame.

To begin, the image directory is defined, and an `ImageDataGenerator` object is instantiated using default parameters. The CSV file containing filenames and integer labels is then loaded into a pandas DataFrame. The `flow_from_dataframe()` function is used to create an iterator that generates batches of image–label pairs. Key parameters include specifying the DataFrame containing the metadata, the directory where images are located, the column names for filenames and labels, the label format (`class_mode='raw'`), the batch size, and the target image size.

When iterating over the generator, each batch is returned as a tuple consisting of a batch of images and a batch of corresponding labels. For example, with a batch size of five, the generator produces image tensors of shape $5 \times 64 \times 64 \times 3$ along with a vector of labels. This iterator can be passed directly into a Keras model for training, making the overall data-loading process extremely compact.

The main advantage of the Keras data generator approach is its **simplicity and brevity**. With only a few lines of code, a complete data pipeline can be constructed, which is often sufficient for straightforward use cases. However, this conciseness comes with limitations. Compared to the `tf.data` API, Keras data generators offer less control over customization and optimization. As a result, while `ImageDataGenerator` is convenient for rapid development, the `tf.data` API remains the preferred choice for more complex or performance-critical data pipelines.

3.2.3 The tensorflow-datasets Package

The simplest way to retrieve data in TensorFlow is by using the **`tensorflow-datasets (tfds)`** package. This package provides direct access to a large collection of well-known and standardized datasets, significantly reducing the effort required to download, preprocess, and organize data. However, a fundamental limitation of `tensorflow-datasets` is that it only supports a predefined set of datasets. Unlike the `tf.data` API or Keras data generators, it cannot be used to ingest arbitrary custom datasets.

The `tensorflow-datasets` package is distributed separately from the core TensorFlow library, although it is typically preinstalled when following standard TensorFlow environment setup instructions. If it is not available, it can be installed easily using `pip install tensorflow-datasets`. Once installed, the package can be verified by importing it without errors. After successful installation, `tensorflow-datasets` exposes a wide range of datasets across multiple domains, including audio, images, and text. A comprehensive catalog of available datasets is provided on

the official TensorFlow website, and several commonly used datasets are summarized in Table 3.2.

Data type	Dataset name	Task
Audio	librispeech	Speech recognition
	ljspeech	Speech recognition
Images	caltech101	Image classification
	cifar10 and cifar100	Image classification
	imagenet2012	Image classification
Text	imdb_reviews	Sentiment analysis
	tiny_shakespeare	Language modelling
	wmt14_translate	Machine translation

To demonstrate its usage, the section focuses on loading the **CIFAR-10** dataset, a widely used benchmark for image classification tasks. CIFAR-10 consists of 60,000 RGB images of size 32×32 , distributed across 10 object categories. Before loading the dataset, available datasets can be listed using `tfds.list_builders()`, which confirms that CIFAR-10 is supported. The dataset is then loaded using the `tfds.load()` function with the `with_info=True` flag, which downloads the dataset if necessary and provides detailed metadata.

The metadata returned by tensorflow-datasets is highly informative. It describes the dataset's version, total number of examples, data splits, feature structure, and citation information. In the case of CIFAR-10, the dataset is split into 50,000 training images and 10,000 testing images. Each sample consists of an image tensor with shape $(32, 32, 3)$ and an integer label representing one of the ten classes. The loaded dataset itself is returned as a dictionary containing separate `tf.data.Dataset` objects for the training and test splits.

Because tensorflow-datasets outputs data as `tf.data.Dataset` objects, the same data processing concepts introduced earlier with the `tf.data` API apply. Initially, the dataset yields one sample at a time, which is not suitable for model training. To resolve this, the training dataset must be batched explicitly using the `batch()` method. Once batched, each iteration produces a dictionary containing three elements: a unique identifier for each record, a batch of image tensors, and a batch of corresponding integer labels.

When using a `tf.data.Dataset` with Keras models, the dataset must produce tuples of the form (x, y) , where x represents the input features and y represents the target labels. Since the dataset currently outputs dictionaries, an additional transformation step is required. This is achieved by defining a mapping function that extracts the image tensor and converts the integer labels into

one-hot encoded vectors. Applying this function with the `map()` method converts the dataset into a format compatible with Keras training workflows.

After this transformation, the dataset can be passed directly to a Keras model using the `model.fit()` method. At this point, data loading, batching, and formatting are handled entirely by the dataset pipeline. With this approach, tensorflow-datasets provides a highly convenient and reliable method for working with standard datasets, especially for experimentation and benchmarking.

This section concludes the discussion on data ingestion methods by highlighting that TensorFlow offers three primary mechanisms for retrieving data: the `tf.data` API, Keras data generators, and the `tensorflow-datasets` package. While `tf.data` provides the highest level of flexibility and control, `tensorflow-datasets` offers the fastest and easiest way to begin model development, albeit with the restriction that only supported datasets can be used.