Keras

Star

About Keras
Getting started
Introduction to Keras for engineers
Introduction to Keras for researchers
The Keras ecosystem
Learning resources
Frequently Asked Questions
Developer guides
Keras API reference
Code examples
Why choose Keras?
Community & governance
Contributing to Keras
KerasTuner
KerasCV
KerasNLP

Search Keras documentation...

» Getting started / Introduction to Keras for Engineers

Introduction to Keras for Engineers

Author: fchollet
Date created: 2020/04/01
Last modified: 2020/04/28
Description: Everything you need to know to use Keras to build real-world machine learning solutions.

View in Colab • **GitHub source**

Setup

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
```

Introduction

Are you a machine learning engineer looking to use Keras to ship deep-learning powered features in real products? This guide will serve as your first introduction to core Keras API concepts.

In this guide, you will learn how to:

- Prepare your data before training a model (by turning it into either NumPy arrays or `tf.data.Dataset` objects).
- Do data preprocessing, for instance feature normalization or vocabulary indexing.
- Build a model that turns your data into useful predictions, using the Keras Functional API.
- Train your model with the built-in Keras `fit()` method, while being mindful of checkpointing, metrics monitoring, and fault tolerance.
- Evaluate your model on a test data and how to use it for inference on new data.
- Customize what `fit()` does, for instance to build a GAN.
- Speed up training by leveraging multiple GPUs.
- Refine your model through hyperparameter tuning.

At the end of this guide, you will get pointers to end-to-end examples to solidify these concepts:

- Image classification
- Text classification
- Credit card fraud detection

Data loading & preprocessing

Neural networks don't process raw data, like text files, encoded JPEG image files, or CSV files. They process **vectorized** & **standardized** representations.

- Text files need to be read into string tensors, then split into words. Finally, the words need to be indexed & turned into integer tensors.
- Images need to be read and decoded into integer tensors, then converted to floating point and normalized to small values (usually between 0 and 1).
- CSV data needs to be parsed, with numerical features converted to floating point tensors and categorical features indexed and converted to integer tensors. Then each feature typically needs to be normalized to zero-mean and unit-variance.
- Etc.

Let's start with data loading.

Data loading

Keras models accept three types of inputs:

- **NumPy arrays**, just like Scikit-Learn and many other Python-based libraries. This is a good option if your data fits in memory.
- **TensorFlow `Dataset` objects**. This is a high-performance option that is more suitable for datasets that do not fit in memory and that are streamed from disk or from a distributed filesystem.
- **Python generators** that yield batches of data (such as custom subclasses of the `keras.utils.Sequence` class).

Before you start training a model, you will need to make your data available as one of these formats. If you have a large dataset and you are training on GPU(s), consider using `Dataset` objects, since they will take care of performance-critical details, such as:

- Asynchronously preprocessing your data on CPU while your GPU is busy, and buffering it into a queue.
- Prefetching data on GPU memory so it's immediately available when the GPU has finished processing the previous batch, so you can reach full GPU utilization.

Keras features a range of utilities to help you turn raw data on disk into a `Dataset`:

- `tf.keras.preprocessing.image_dataset_from_directory` turns image files sorted into class-specific folders into a labeled dataset of image tensors.
- `tf.keras.preprocessing.text_dataset_from_directory` does the same for text files.

In addition, the TensorFlow `tf.data` includes other similar utilities, such as `tf.data.experimental.make_csv_dataset` to load structured data from CSV files.

Example: obtaining a labeled dataset from image files on disk

Supposed you have image files sorted by class in different folders, like this:

```
main_directory/
...class_a/
......a_image_1.jpg
......a_image_2.jpg
...class_b/
......b_image_1.jpg
......b_image_2.jpg
```

Then you can do:

```
# Create a dataset.
dataset = keras.preprocessing.image_dataset_from_directory(
  'path/to/main_directory', batch_size=64, image_size=(200, 200))

# For demonstration, iterate over the batches yielded by the dataset.
for data, labels in dataset:
   print(data.shape)  # (64, 200, 200, 3)
   print(data.dtype)  # float32
   print(labels.shape)  # (64,)
   print(labels.dtype)  # int32
```

The label of a sample is the rank of its folder in alphanumeric order. Naturally, this can also be configured explicitly by passing, e.g. `class_names=['class_a', 'class_b']`, in which cases label `0` will be `class_a` and `1` will be `class_b`.

Example: obtaining a labeled dataset from text files on disk

Likewise for text: if you have `.txt` documents sorted by class in different folders, you can do:

```
dataset = keras.preprocessing.text_dataset_from_directory(
  'path/to/main_directory', batch_size=64)

# For demonstration, iterate over the batches yielded by the dataset.
for data, labels in dataset:
   print(data.shape)  # (64,)
   print(data.dtype)  # string
   print(labels.shape)  # (64,)
   print(labels.dtype)  # int32
```

Data preprocessing with Keras

Once your data is in the form of string/int/float NumPy arrays, or a `Dataset` object (or Python generator) that yields batches of string/int/float tensors, it is time to **preprocess** the data. This can mean:

- Tokenization of string data, followed by token indexing.
- Feature normalization.
- Rescaling the data to small values (in general, input values to a neural network should be close to zero -- typically we expect either data with zero-mean and unit-variance, or data in the `[0, 1]` range.

The ideal machine learning model is end-to-end

In general, you should seek to do data preprocessing **as part of your model** as much as possible, not via an external data preprocessing pipeline. That's because external data preprocessing makes your models less portable when it's time to use them in production. Consider a model that processes text: it uses a specific tokenization algorithm and a specific vocabulary index. When you want to ship your model to a mobile app or a JavaScript app, you will need to recreate the exact same preprocessing setup in the target language. This can get very tricky: any small discrepancy between the original pipeline and the one you recreate has the potential to completely invalidate your model, or at least severely degrade its performance.

It would be much easier to be able to simply export an end-to-end model that already includes preprocessing. **The ideal model should expect as input something as close as possible to raw data: an image model should expect RGB pixel values in the `[0, 255]` range, and a text model should accept strings of `utf-8` characters.** That way, the consumer of the exported model doesn't have to know about the preprocessing pipeline.

Using Keras preprocessing layers

In Keras, you do in-model data preprocessing via **preprocessing layers**. This includes:

- Vectorizing raw strings of text via the `TextVectorization` layer
- Feature normalization via the `Normalization` layer
- Image rescaling, cropping, or image data augmentation

The key advantage of using Keras preprocessing layers is that **they can be included directly into your model**, either during training or after training, which makes your models portable.

Some preprocessing layers have a state:

- `TextVectorization` holds an index mapping words or tokens to integer indices
- `Normalization` holds the mean and variance of your features

The state of a preprocessing layer is obtained by calling `layer.adapt(data)` on a sample of the training data (or all of it).

Example: turning strings into sequences of integer word indices

```python
from tensorflow.keras.layers import TextVectorization

# Example training data, of dtype `string`.
training_data = np.array([["This is the 1st sample."], ["And here's the 2nd sample."]])

# Create a TextVectorization layer instance. It can be configured to either
# return integer token indices, or a dense token representation (e.g. multi-hot
# or TF-IDF). The text standardization and text splitting algorithms are fully
# configurable.
vectorizer = TextVectorization(output_mode="int")

# Calling `adapt` on an array or dataset makes the layer generate a vocabulary
# index for the data, which can then be reused when seeing new data.
vectorizer.adapt(training_data)

# After calling adapt, the layer is able to encode any n-gram it has seen before
# in the `adapt()` data. Unknown n-grams are encoded via an "out-of-vocabulary"
# token.
integer_data = vectorizer(training_data)
print(integer_data)
```

```
tf.Tensor(
[[4 5 2 9 3]
 [7 6 2 8 3]], shape=(2, 5), dtype=int64)
```

Example: turning strings into sequences of one-hot encoded bigrams

```python
from tensorflow.keras.layers import TextVectorization

# Example training data, of dtype `string`.
training_data = np.array([["This is the 1st sample."], ["And here's the 2nd sample."]])

# Create a TextVectorization layer instance. It can be configured to either
# return integer token indices, or a dense token representation (e.g. multi-hot
# or TF-IDF). The text standardization and text splitting algorithms are fully
# configurable.
vectorizer = TextVectorization(output_mode="binary", ngrams=2)

# Calling `adapt` on an array or dataset makes the layer generate a vocabulary
# index for the data, which can then be reused when seeing new data.
vectorizer.adapt(training_data)

# After calling adapt, the layer is able to encode any n-gram it has seen before
# in the `adapt()` data. Unknown n-grams are encoded via an "out-of-vocabulary"
# token.
integer_data = vectorizer(training_data)
print(integer_data)
```

```
tf.Tensor(
[[0. 1. 1. 1. 1. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 1. 1.]
 [0. 1. 1. 0. 0. 1. 0. 0. 0. 1. 1. 1. 1. 1. 1. 0. 0.]], shape=(2, 17), dtype=float32)
```

Example: normalizing features

```
from tensorflow.keras.layers import Normalization

# Example image data, with values in the [0, 255] range
training_data = np.random.randint(0, 256, size=(64, 200, 200, 3)).astype("float32")

normalizer = Normalization(axis=-1)
normalizer.adapt(training_data)

normalized_data = normalizer(training_data)
print("var: %.4f" % np.var(normalized_data))
print("mean: %.4f" % np.mean(normalized_data))
```

```
var: 1.0000
mean: -0.0000
```

Example: rescaling & center-cropping images

Both the `Rescaling` layer and the `CenterCrop` layer are stateless, so it isn't necessary to call `adapt()` in this case.

```
from tensorflow.keras.layers import CenterCrop
from tensorflow.keras.layers import Rescaling

# Example image data, with values in the [0, 255] range
training_data = np.random.randint(0, 256, size=(64, 200, 200, 3)).astype("float32")

cropper = CenterCrop(height=150, width=150)
scaler = Rescaling(scale=1.0 / 255)

output_data = scaler(cropper(training_data))
print("shape:", output_data.shape)
print("min:", np.min(output_data))
print("max:", np.max(output_data))
```

```
shape: (64, 150, 150, 3)
min: 0.0
max: 1.0
```

Building models with the Keras Functional API

A "layer" is a simple input-output transformation (such as the scaling & center-cropping transformations above). For instance, here's a linear projection layer that maps its inputs to a 16-dimensional feature space:

```
dense = keras.layers.Dense(units=16)
```

A "model" is a directed acyclic graph of layers. You can think of a model as a "bigger layer" that encompasses multiple sublayers and that can be trained via exposure to data.

The most common and most powerful way to build Keras models is the Functional API. To build models with the Functional API, you start by specifying the shape (and optionally the dtype) of your inputs. If any dimension of your input can vary, you can specify it as `None`. For instance, an input for 200x200 RGB image would have shape `(200, 200, 3)`, but an input for RGB images of any size would have shape `(None, None, 3)`.

```
# Let's say we expect our inputs to be RGB images of arbitrary size
inputs = keras.Input(shape=(None, None, 3))
```

After defining your input(s), you can chain layer transformations on top of your inputs, until your final output:

```
from tensorflow.keras import layers

# Center-crop images to 150x150
x = CenterCrop(height=150, width=150)(inputs)
# Rescale images to [0, 1]
x = Rescaling(scale=1.0 / 255)(x)

# Apply some convolution and pooling layers
x = layers.Conv2D(filters=32, kernel_size=(3, 3), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(3, 3))(x)
x = layers.Conv2D(filters=32, kernel_size=(3, 3), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(3, 3))(x)
x = layers.Conv2D(filters=32, kernel_size=(3, 3), activation="relu")(x)

# Apply global average pooling to get flat feature vectors
x = layers.GlobalAveragePooling2D()(x)

# Add a dense classifier on top
```

```
num_classes = 10
outputs = layers.Dense(num_classes, activation="softmax")(x)
```

Once you have defined the directed acyclic graph of layers that turns your input(s) into your outputs, instantiate a `Model` object:

```
model = keras.Model(inputs=inputs, outputs=outputs)
```

This model behaves basically like a bigger layer. You can call it on batches of data, like this:

```
data = np.random.randint(0, 256, size=(64, 200, 200, 3)).astype("float32")
processed_data = model(data)
print(processed_data.shape)
```

```
(64, 10)
```

You can print a summary of how your data gets transformed at each stage of the model. This is useful for debugging.

Note that the output shape displayed for each layer includes the **batch size**. Here the batch size is None, which indicates our model can process batches of any size.

```
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, None, None, 3)]   0         
_________________________________________________________________
center_crop_1 (CenterCrop)   (None, 150, 150, 3)       0         
_________________________________________________________________
rescaling_1 (Rescaling)      (None, 150, 150, 3)       0         
_________________________________________________________________
conv2d (Conv2D)              (None, 148, 148, 32)      896       
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 49, 49, 32)        0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 47, 47, 32)        9248      
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 13, 13, 32)        9248      
_________________________________________________________________
global_average_pooling2d (Gl (None, 32)                0         
_________________________________________________________________
dense (Dense)                (None, 10)                330       
=================================================================
Total params: 19,722
Trainable params: 19,722
Non-trainable params: 0
_________________________________________________________________
```

The Functional API also makes it easy to build models that have multiple inputs (for instance, an image *and* its metadata) or multiple outputs (for instance, predicting the class of the image *and* the likelihood that a user will click on it). For a deeper dive into what you can do, see our guide to the Functional API.

Training models with `fit()`

At this point, you know:

- How to prepare your data (e.g. as a NumPy array or a `tf.data.Dataset` object)
- How to build a model that will process your data

The next step is to train your model on your data. The `Model` class features a built-in training loop, the `fit()` method. It accepts `Dataset` objects, Python generators that yield batches of data, or NumPy arrays.

Before you can call `fit()`, you need to specify an optimizer and a loss function (we assume you are already familiar with these concepts). This is the `compile()` step:

```
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=1e-3),
              loss=keras.losses.CategoricalCrossentropy())
```

Loss and optimizer can be specified via their string identifiers (in this case their default constructor argument values are used):

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

Once your model is compiled, you can start "fitting" the model to the data. Here's what fitting a model looks like with NumPy data:

```
model.fit(numpy_array_of_samples, numpy_array_of_labels,
          batch_size=32, epochs=10)
```

Besides the data, you have to specify two key parameters: the `batch_size` and the number of epochs (iterations on the data). Here our data will get sliced on batches of 32 samples, and the model will iterate 10 times over the data during training.

Here's what fitting a model looks like with a dataset:

```
model.fit(dataset_of_samples_and_labels, epochs=10)
```

Since the data yielded by a dataset is expected to be already batched, you don't need to specify the batch size here.

Let's look at it in practice with a toy example model that learns to classify MNIST digits:

```
# Get the data as Numpy arrays
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Build a simple model
inputs = keras.Input(shape=(28, 28))
x = layers.Rescaling(1.0 / 255)(inputs)
x = layers.Flatten()(x)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dense(128, activation="relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.summary()

# Compile the model
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")

# Train the model for 1 epoch from Numpy data
batch_size = 64
print("Fit on NumPy data")
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=1)

# Train the model for 1 epoch using a dataset
dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(batch_size)
print("Fit on Dataset")
history = model.fit(dataset, epochs=1)
```

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 28, 28)]          0
_________________________________________________________________
rescaling_2 (Rescaling)      (None, 28, 28)            0
_________________________________________________________________
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 118,282
Trainable params: 118,282
Non-trainable params: 0
_________________________________________________________________
Fit on NumPy data
938/938 [==============================] - 1s 940us/step - loss: 0.4771
Fit on Dataset
938/938 [==============================] - 1s 942us/step - loss: 0.1138
```

The `fit()` call returns a "history" object which records what happened over the course of training. The `history.history` dict contains per-epoch timeseries of metrics values (here we have only one metric, the loss, and one epoch, so we only get a single scalar):

```
print(history.history)
```

```
{'loss': [0.11384169012308121]}
```

For a detailed overview of how to use `fit()`, see the guide to training & evaluation with the built-in Keras methods.

Keeping track of performance metrics

As you're training a model, you want to keep track of metrics such as classification accuracy, precision, recall, AUC, etc. Besides, you want to monitor these metrics not only on the training data, but also on a validation set.

Monitoring metrics

You can pass a list of metric objects to `compile()`, like this:

```
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=[keras.metrics.SparseCategoricalAccuracy(name="acc")],
)
history = model.fit(dataset, epochs=1)
```

```
938/938 [==============================] - 1s 929us/step - loss: 0.0835 - acc: 0.9748
```

Passing validation data to `fit()`

You can pass validation data to `fit()` to monitor your validation loss & validation metrics. Validation metrics get reported at the end of each epoch.

```
val_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(batch_size)
history = model.fit(dataset, epochs=1, validation_data=val_dataset)
```

```
938/938 [==============================] - 1s 1ms/step - loss: 0.0563 - acc: 0.9829 - val
```

Using callbacks for checkpointing (and more)

If training goes on for more than a few minutes, it's important to save your model at regular intervals during training. You can then use your saved models to restart training in case your training process crashes (this is important for multi-worker distributed training, since with many workers at least one of them is bound to fail at some point).

An important feature of Keras is **callbacks**, configured in `fit()`. Callbacks are objects that get called by the model at different point during training, in particular:

- At the beginning and end of each batch
- At the beginning and end of each epoch

Callbacks are a way to make model trainable entirely scriptable.

You can use callbacks to periodically save your model. Here's a simple example: a `ModelCheckpoint` callback configured to save the model at the end of every epoch. The filename will include the current epoch.

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath='path/to/my/model_{epoch}',
        save_freq='epoch')
]
model.fit(dataset, epochs=2, callbacks=callbacks)
```

You can also use callbacks to do things like periodically changing the learning of your optimizer, streaming metrics to a Slack bot, sending yourself an email notification when training is complete, etc.

For detailed overview of what callbacks are available and how to write your own, see the callbacks API documentation and the guide to writing custom callbacks.

Monitoring training progress with TensorBoard

Staring at the Keras progress bar isn't the most ergonomic way to monitor how your loss and metrics are evolving over time. There's a better solution: TensorBoard, a web application that can display real-time graphs of your metrics (and more).

To use TensorBoard with `fit()`, simply pass a `keras.callbacks.TensorBoard` callback specifying the directory where to store TensorBoard logs:

```
callbacks = [
    keras.callbacks.TensorBoard(log_dir='./logs')
]
model.fit(dataset, epochs=2, callbacks=callbacks)
```

You can then launch a TensorBoard instance that you can open in your browser to monitor the logs getting written to this location:

```
tensorboard --logdir=./logs
```

What's more, you can launch an in-line TensorBoard tab when training models in Jupyter / Colab notebooks. Here's more information.

After `fit()`: evaluating test performance & generating predictions on new data

Once you have a trained model, you can evaluate its loss and metrics on new data via `evaluate()`:

```
loss, acc = model.evaluate(val_dataset)  # returns loss and metrics
print("loss: %.2f" % loss)
print("acc: %.2f" % acc)
```

```
157/157 [==============================] - 0s 688us/step - loss: 0.1041 - acc: 0.9692
loss: 0.10
acc: 0.97
```

You can also generate NumPy arrays of predictions (the activations of the output layer(s) in the model) via `predict()`:

```
predictions = model.predict(val_dataset)
print(predictions.shape)
```

```
(10000, 10)
```

Using `fit()` with a custom training step

By default, `fit()` is configured for **supervised learning**. If you need a different kind of training loop (for instance, a GAN training loop), you can provide your own implementation of the `Model.train_step()` method. This is the method that is repeatedly called during `fit()`.

Metrics, callbacks, etc. will work as usual.

Here's a simple example that reimplements what `fit()` normally does:

```
class CustomModel(keras.Model):
  def train_step(self, data):
    # Unpack the data. Its structure depends on your model and
    # on what you pass to `fit()`.
    x, y = data
    with tf.GradientTape() as tape:
      y_pred = self(x, training=True)  # Forward pass
      # Compute the loss value
      # (the loss function is configured in `compile()`)
      loss = self.compiled_loss(y, y_pred,
                                regularization_losses=self.losses)
    # Compute gradients
    trainable_vars = self.trainable_variables
    gradients = tape.gradient(loss, trainable_vars)
    # Update weights
    self.optimizer.apply_gradients(zip(gradients, trainable_vars))
    # Update metrics (includes the metric that tracks the loss)
    self.compiled_metrics.update_state(y, y_pred)
    # Return a dict mapping metric names to current value
    return {m.name: m.result() for m in self.metrics}

# Construct and compile an instance of CustomModel
inputs = keras.Input(shape=(32,))
outputs = keras.layers.Dense(1)(inputs)
model = CustomModel(inputs, outputs)
model.compile(optimizer='adam', loss='mse', metrics=[...])

# Just use `fit` as usual
model.fit(dataset, epochs=3, callbacks=...)
```

For a detailed overview of how you customize the built-in training & evaluation loops, see the guide: "Customizing what happens in `fit()`".

Debugging your model with eager execution

If you write custom training steps or custom layers, you will need to debug them. The debugging experience is an integral part of a framework: with Keras, the debugging workflow is designed with the user in mind.

By default, your Keras models are compiled to highly-optimized computation graphs that deliver fast execution times. That means that the Python code you write (e.g. in a custom `train_step`) is not the code you are actually executing. This introduces a layer of indirection that can make debugging hard.

Debugging is best done step by step. You want to be able to sprinkle your code with `print()` statement to see what your data looks like after every operation, you want to be able to use `pdb`. You can achieve this by **running your model eagerly**. With eager execution, the Python code you write is the code that gets executed.

Simply pass `run_eagerly=True` to `compile()`:

```
model.compile(optimizer='adam', loss='mse', run_eagerly=True)
```

Of course, the downside is that it makes your model significantly slower. Make sure to switch it back off to get the benefits of compiled computation graphs once you are done debugging!

In general, you will use `run_eagerly=True` every time you need to debug what's happening inside your `fit()` call.

Speeding up training with multiple GPUs

Keras has built-in industry-strength support for multi-GPU training and distributed multi-worker training, via the `tf.distribute` API.

If you have multiple GPUs on your machine, you can train your model on all of them by:

- Creating a `tf.distribute.MirroredStrategy` object
- Building & compiling your model inside the strategy's scope
- Calling `fit()` and `evaluate()` on a dataset as usual

```
# Create a MirroredStrategy.
strategy = tf.distribute.MirroredStrategy()

# Open a strategy scope.
with strategy.scope():
  # Everything that creates variables should be under the strategy scope.
  # In general this is only model construction & `compile()`.
  model = Model(...)
  model.compile(...)

# Train the model on all available devices.
train_dataset, val_dataset, test_dataset = get_dataset()
model.fit(train_dataset, epochs=2, validation_data=val_dataset)

# Test the model on all available devices.
model.evaluate(test_dataset)
```

For a detailed introduction to multi-GPU & distributed training, see this guide.

Doing preprocessing synchronously on-device vs. asynchronously on host CPU

You've learned about preprocessing, and you've seen example where we put image preprocessing layers (`CenterCrop` and `Rescaling`) directly inside our model.

Having preprocessing happen as part of the model during training is great if you want to do on-device preprocessing, for instance, GPU-accelerated feature normalization or image augmentation. But there are kinds of preprocessing that are not suited to this setup: in particular, text preprocessing with the `TextVectorization` layer. Due to its sequential nature and due to the fact that it can only run on CPU, it's often a good idea to do **asynchronous preprocessing**.

With asynchronous preprocessing, your preprocessing operations will run on CPU, and the preprocessed samples will be buffered into a queue while your GPU is busy with previous batch of data. The next batch of preprocessed samples will then be fetched from the queue to the GPU memory right before the GPU becomes available again (prefetching). This ensures that preprocessing will not be blocking and that your GPU can run at full utilization.

To do asynchronous preprocessing, simply use `dataset.map` to inject a preprocessing operation into your data pipeline:

```
# Example training data, of dtype `string`.
samples = np.array([["This is the 1st sample."], ["And here's the 2nd sample."]])
labels = [[0], [1]]

# Prepare a TextVectorization layer.
vectorizer = TextVectorization(output_mode="int")
vectorizer.adapt(samples)

# Asynchronous preprocessing: the text vectorization is part of the tf.data pipeline.
# First, create a dataset
dataset = tf.data.Dataset.from_tensor_slices((samples, labels)).batch(2)
# Apply text vectorization to the samples
dataset = dataset.map(lambda x, y: (vectorizer(x), y))
```

```
# Prefetch with a buffer size of 2 batches
dataset = dataset.prefetch(2)

# Our model should expect sequences of integers as inputs
inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(input_dim=10, output_dim=32)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer="adam", loss="mse", run_eagerly=True)
model.fit(dataset)
```

```
1/1 [==============================] - 0s 13ms/step - loss: 0.5028

<tensorflow.python.keras.callbacks.History at 0x147777490>
```

Compare this to doing text vectorization as part of the model:

```
# Our dataset will yield samples that are strings
dataset = tf.data.Dataset.from_tensor_slices((samples, labels)).batch(2)

# Our model should expect strings as inputs
inputs = keras.Input(shape=(1,), dtype="string")
x = vectorizer(inputs)
x = layers.Embedding(input_dim=10, output_dim=32)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer="adam", loss="mse", run_eagerly=True)
model.fit(dataset)
```

```
1/1 [==============================] - 0s 16ms/step - loss: 0.5258

<tensorflow.python.keras.callbacks.History at 0x1477b1910>
```

When training text models on CPU, you will generally not see any performance difference between the two setups. When training on GPU, however, doing asynchronous buffered preprocessing on the host CPU while the GPU is running the model itself can result in a significant speedup.

After training, if you want to export an end-to-end model that includes the preprocessing layer(s), this is easy to do, since `TextVectorization` is a layer:

```
inputs = keras.Input(shape=(1,), dtype='string')
x = vectorizer(inputs)
outputs = trained_model(x)
end_to_end_model = keras.Model(inputs, outputs)
```

Finding the best model configuration with hyperparameter tuning

Once you have a working model, you're going to want to optimize its configuration -- architecture choices, layer sizes, etc. Human intuition can only go so far, so you'll want to leverage a systematic approach: hyperparameter search.

You can use KerasTuner to find the best hyperparameter for your Keras models. It's as easy as calling `fit()`.

Here how it works.

First, place your model definition in a function, that takes a single `hp` argument. Inside this function, replace any value you want to tune with a call to hyperparameter sampling methods, e.g. `hp.Int()` or `hp.Choice()`:

```
def build_model(hp):
    inputs = keras.Input(shape=(784,))
    x = layers.Dense(
        units=hp.Int('units', min_value=32, max_value=512, step=32),
        activation='relu'))(inputs)
    outputs = layers.Dense(10, activation='softmax')(x)
    model = keras.Model(inputs, outputs)
    model.compile(
        optimizer=keras.optimizers.Adam(
            hp.Choice('learning_rate',
                      values=[1e-2, 1e-3, 1e-4])),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
    return model
```

The function should return a compiled model.

Next, instantiate a tuner object specifying your optimization objective and other search parameters:

```
import keras_tuner

tuner = keras_tuner.tuners.Hyperband(
  build_model,
  objective='val_loss',
  max_epochs=100,
  max_trials=200,
  executions_per_trial=2,
  directory='my_dir')
```

Finally, start the search with the `search()` method, which takes the same arguments as `Model.fit()`:

```
tuner.search(dataset, validation_data=val_dataset)
```

When search is over, you can retrieve the best model(s):

```
models = tuner.get_best_models(num_models=2)
```

Or print a summary of the results:

```
tuner.results_summary()
```

End-to-end examples

To familiarize yourself with the concepts in this introduction, see the following end-to-end examples:

- Text classification
- Image classification
- Credit card fraud detection

What to learn next

- Learn more about the Functional API.
- Learn more about the features of `fit()` and `evaluate()`.
- Learn more about callbacks.
- Learn more about creating your own custom training steps.
- Learn more about multi-GPU and distributed training.
- Learn how to do transfer learning.