

CH7. Working with Keras: A deep dive

Il-Youp Kwak
Chung-Ang University



Working with Keras: A deep dive

A spectrum of workflows

Different ways to build Keras models

Using built-in training and evaluation loops

Writing your own training and evaluation loops



A spectrum of workflows

The design of the **Keras API**: make it easy to get started, yet make it possible to handle high-complexity use cases, only requiring incremental learning at each step

Simple use cases should be easy and approachable, and arbitrarily advanced workflows should be possible



Different ways to build Keras models

Three APIs for building models:

- ① **Sequential model**: simple stacks of layers
- ② **Functional API**: focuses on graph-like model architectures
- ③ **Model subclassing**: a low-level option where you write everything yourself.

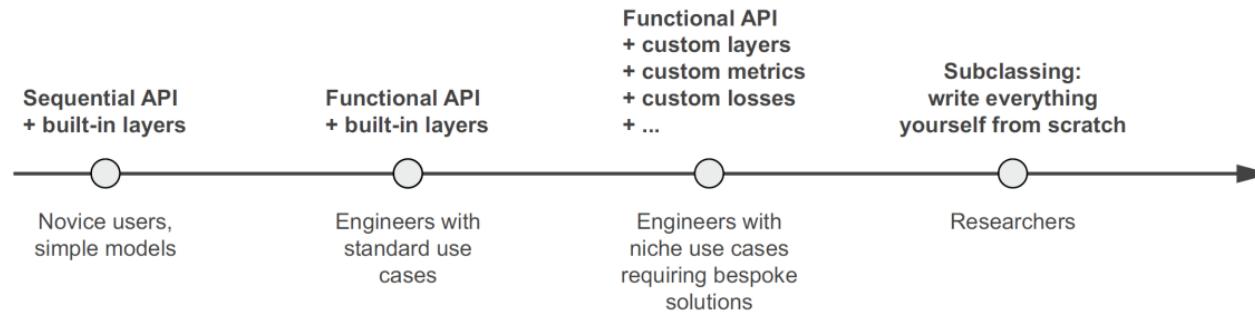


Figure 7.1 Progressive disclosure of complexity for model building



The Sequential model



Listing 7.1 The Sequential class

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```



Build the same model incrementally via the **add()** method

Listing 7.2 Incrementally building a Sequential model

```
model = keras.Sequential()
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
```



Sequential model does not have any weights until you actually call it on some data, or call its build() method with an input shape

Listing 7.3 Models that aren't yet built have no weights

```
>>> model.weights
ValueError: Weights for model sequential_1 have not yet been created.
```

At that point, the model isn't built yet.

$$y = f_2(f_1(x))$$

→ 2 layers

$$f_1(x) = \sigma(w_i x + b_i)$$

w_i : 64×3 b_i : 3×1 x : 64×1

Listing 7.4 Calling a model for the first time to build it

```
>>> model.build(input_shape=(None, 3))
>>> model.weights
[<tf.Variable "dense_2/kernel:0" shape=(3, 64) dtype=float32, ... >,
 <tf.Variable "dense_2/bias:0" shape=(64,) dtype=float32, ... >,
 <tf.Variable "dense_3/kernel:0" shape=(64, 10) dtype=float32, ... >,
 <tf.Variable "dense_3/bias:0" shape=(10,) dtype=float32, ... >]
```

Now you can retrieve the model's weights.

Builds the model—now the model will expect samples of shape (3,). The None in the input shape signals that the batch size could be anything.



After the model is built, you can display its contents via the summary()

Listing 7.5 The summary() method

```
>>> model.summary()
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 64)	256
dense_3 (Dense)	(None, 10)	650
<hr/>		
Total params: 906		
Trainable params: 906		
Non-trainable params: 0		

$$\begin{aligned} & \text{Input shape: } 3 \times 64 \\ & \text{Layer 1: } \underbrace{6(xw_1 + b_1)}_{3 \times 64 \rightarrow 64} = x' \\ & \text{Layer 2: } \underbrace{6(x'w_2 + b_2)}_{64 \times 10 \rightarrow 10} = y \end{aligned}$$



This model happens to be named “sequential_1.” You can give names to everything in Keras—every model, every layer

Listing 7.6 Naming models and layers with the name argument

```
>>> model = keras.Sequential(name="my_example_model") DEVELOPS
>>> model.add(layers.Dense(64, activation="relu", name="my_first_layer"))
>>> model.add(layers.Dense(10, activation="softmax", name="my_last_layer"))
>>> model.build((None, 3))
>>> model.summary()
Model: "my_example_model"
```

layer 0121320

Layer (type)	Output Shape	Param #
<hr/>		
my_first_layer (Dense)	(None, 64)	256
<hr/>		
my_last_layer (Dense)	(None, 10)	650
<hr/>		
Total params: 906		
Trainable params: 906		
Non-trainable params: 0		



If you'd like to print summary(), declare the shape of the model's inputs in advance

Listing 7.7 Specifying the input shape of your model in advance

```
model = keras.Sequential()  
model.add(keras.Input(shape=(3,)))  
model.add(layers.Dense(64, activation="relu"))
```

Use Input to declare the shape of the inputs. Note that the shape argument must be the shape of each sample, not the shape of one batch.

```
>>> model.summary()  
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 64)	256

```
=====  
Total params: 256  
Trainable params: 256  
Non-trainable params: 0
```

```
>>> model.add(layers.Dense(10, activation="softmax"))  
>>> model.summary()  
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 64)	256
dense_5 (Dense)	(None, 10)	650

```
=====  
Total params: 906  
Trainable params: 906  
Non-trainable params: 0
```



colab.research.google.com/github/fchollet/deep-learning-with-pyth
news study data science 학회사이트 업무관련 기타 Bookmark N

Pause 00:00:00 Select Area Audio Record Pointer keras.ipynb

변역 SR Translate scm Data 학술지관련 기타

Share

chapter07_working-with-keras.i

File Edit View Insert Runtime Tools Help

Table of contents

- Working with Keras: A deep dive
 - A spectrum of workflows
 - Different ways to build Keras models
 - The Sequential model
 - The Functional API
 - A simple example
 - Multi-input, multi-output models
 - Training a multi-input, multi-output model
 - The power of the Functional API: Access to layer connectivity
 - Subclassing the Model class
 - Rewriting our previous example as a subclassed model
 - Beware: What subclassed models don't support
 - Mixing and matching different components
 - Remember: Use the right tool for the job

+ Code + Text Copy to Drive Connect Editing

This is a companion notebook for the book [Deep Learning with Python, Second Edition](#). For readability, it only contains runnable code blocks and section titles, and omits everything else in the book: text paragraphs, figures, and pseudocode.

If you want to be able to follow what's going on, I recommend reading the notebook side by side with your copy of the book.

This notebook was generated for TensorFlow 2.6.

Working with Keras: A deep dive

A spectrum of workflows

Different ways to build Keras models

The Sequential model

The Sequential class

```
[ ] from tensorflow import keras
from tensorflow.keras import layers
```

model.layers

모델의 레이어 목록

model.weights[1]

모델의 가중치 목록

W1, W2, b1, b2

오늘 7:56
2022-04-02

The Functional API ②

Sequential model only express models with single input and single output

Functional API provide more power to define general model

Listing 7.8 A simple Functional model with two Dense layers

```
inputs = keras.Input(shape=(3,), name="my_input")
features = layers.Dense(64, activation="relu")(inputs)
outputs = layers.Dense(10, activation="softmax")(features)
model = keras.Model(inputs=inputs, outputs=outputs)
```



inputs object holds information about the shape and dtype of the data

```
>>> inputs.shape
(None, 3)
>>> inputs.dtype
float32
```

The model will process batches where each sample has shape (3,). The number of samples per batch is variable (indicated by the None batch size).

These batches will have dtype float32.

We call such an object a symbolic tensor. It doesn't contain any actual data, but it encodes the specifications of the actual tensors of data that the model will see when you use it.

나중에 쓰는 걸...
나중에 쓰는 걸...

```
features = layers.Dense(64, activation="relu")(inputs)
```

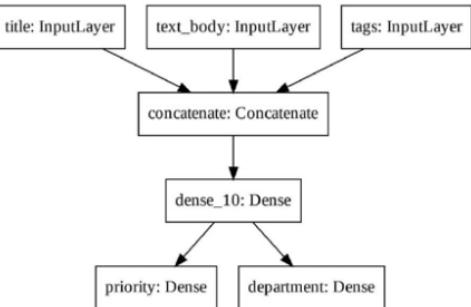
All Keras layers can be called both on real tensors of data and on these symbolic tensors. In the latter case, they return a new symbolic tensor, with updated shape and dtype information:

```
>>> features.shape  
(None, 64)
```

After obtaining the final outputs, we instantiated the model by specifying its inputs and outputs in the Model constructor:

```
outputs = layers.Dense(10, activation="softmax")(features)  
model = keras.Model(inputs=inputs, outputs=outputs)
```





Multi-input, Multi-output models

Most deep learning models look like graphs.

Consider multi-input, multi-output model:

Listing 7.9 A multi-input, multi-output Functional model

```
vocabulary_size = 10000
num_tags = 100
num_departments = 4
```

3 inputs

Define model inputs.

```
title = keras.Input(shape=(vocabulary_size,), name="title")
text_body = keras.Input(shape=(vocabulary_size,), name="text_body")
tags = keras.Input(shape=(num_tags,), name="tags")
```

Combine input features into a single tensor, features, by concatenating them.

→ 20100 dimension

```
features = layers.concatenate([title, text_body, tags])
features = layers.Dense(64, activation="relu")(features)
```

2 outputs *

Define model outputs.

```
* priority = layers.Dense(1, activation="sigmoid", name="priority")(features)
department = layers.Dense(
    num_departments, activation="softmax", name="department")(features)
```

```
model = keras.Model(inputs=[title, text_body, tags],
                     outputs=[priority, department])
```

Create the model by specifying its inputs and outputs.

Apply an intermediate layer to recombine input features into richer representations.



Training a Multi-input, Multi-output model

Listing 7.10 Training a model by providing lists of input and target arrays

```
import numpy as np

num_samples = 1280

Dummy input data | title_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
                  text_body_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
                  tags_data = np.random.randint(0, 2, size=(num_samples, num_tags))

Dummy target data | priority_data = np.random.random(size=(num_samples, 1)) ←
                  department_data = np.random.randint(0, 2, size=(num_samples, num_departments)) ←

model.compile(optimizer="rmsprop",
              loss=["mean_squared_error", "categorical_crossentropy"],
              metrics=[["mean_absolute_error"], ["accuracy"]])
model.fit([title_data, text_body_data, tags_data],
          [priority_data, department_data],
          epochs=1)
model.evaluate([title_data, text_body_data, tags_data],
               [priority_data, department_data])
priority_preds, department_preds = model.predict(
    [title_data, text_body_data, tags_data])
```



You can also leverage the names you gave to the Input objects and the output layers, and pass data via dictionaries.

Listing 7.11 Training a model by providing **dicts of input and target arrays**

```
model.compile(optimizer="rmsprop",
              loss={"priority": "mean_squared_error", "department":
                    "categorical_crossentropy"},
              metrics={"priority": ["mean_absolute_error"], "department":
                    ["accuracy"]})
model.fit({"title": title_data, "text_body": text_body_data,
           "tags": tags_data},
           {"priority": priority_data, "department": department_data},
           epochs=1)
model.evaluate({"title": title_data, "text_body": text_body_data,
                "tags": tags_data},
                {"priority": priority_data, "department": department_data})
priority_preds, department_preds = model.predict(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data})
```



The Power of the functional API: Access to layer connectivity

A Functional model is an explicit graph data structure. This makes it possible to inspect how layers are connected and reuse previous graph nodes as part of new models.

```
keras.utils.plot_model(model, "ticket_classifier.png")
```

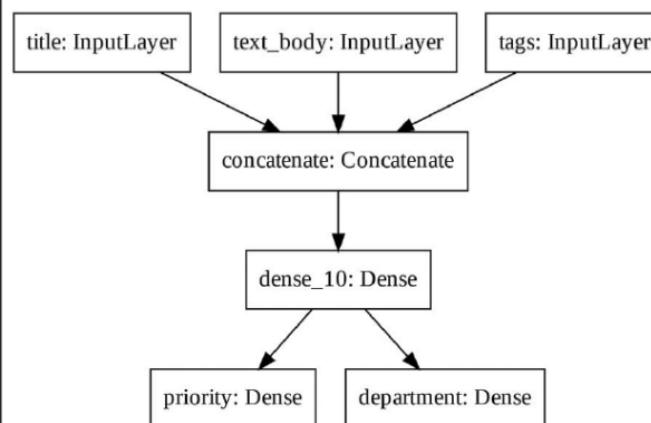


Figure 7.2 Plot generated by `plot_model()` on our ticket classifier model

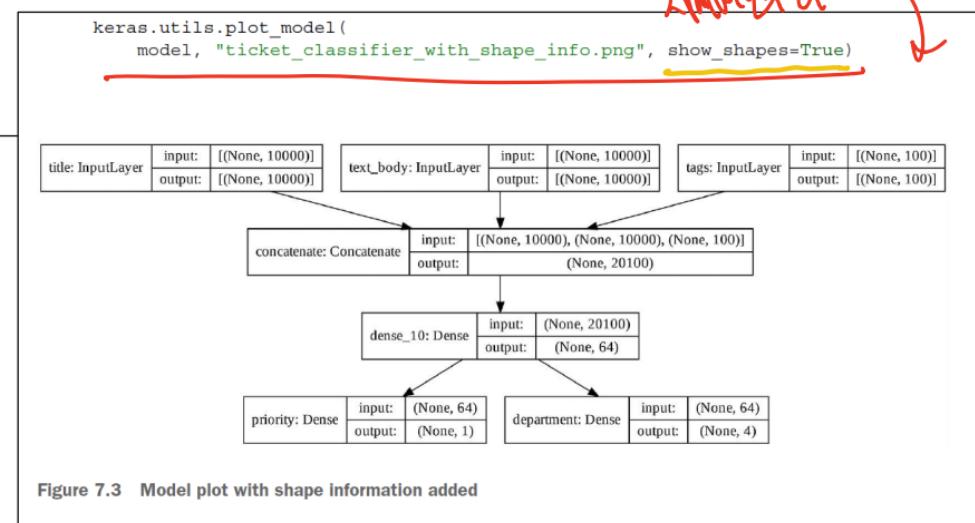


Figure 7.3 Model plot with shape information added



Access to layer connectivity also means that you can inspect and reuse individual nodes (layer calls) in the graph.

Listing 7.12 Retrieving the inputs or outputs of a layer in a Functional model

```
>>> model.layers
[<tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d358>,
 <tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d2e8>,
 <tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d470>,
 <tensorflow.python.keras.layers.merge.Concatenate at 0x7fa963f9d860>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa964074390>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa963f9d898>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa963f95470>]
>>> model.layers[3].input
[<tf.Tensor "title:0" shape=(None, 10000) dtype=float32>,
 <tf.Tensor "text_body:0" shape=(None, 10000) dtype=float32>,
 <tf.Tensor "tags:0" shape=(None, 100) dtype=float32>]
>>> model.layers[3].output
<tf.Tensor "concatenate(concat:0" shape=(None, 20100) dtype=float32>
```

A red bracket labeled "concatenate layer" points from the output of `model.layers[3].input` to the `concatenate` layer in the code. Another red bracket points from the output of the `concatenate` layer to `model.layers[3].output`.

Enables us to do feature extraction, creating models that reuse intermediate features from another model.

Listing 7.13 Creating a new model by reusing intermediate layer outputs

```
features = model.layers[4].output
difficulty = layers.Dense(3, activation="softmax", name="difficulty")(features)

new_model = keras.Model(
    inputs=[title, text_body, tags],
    outputs=[priority, department, difficulty])
```

A callout box points from the line `layers[4] is our intermediate Dense layer` to the `difficulty` layer in the code. A red bracket connects the `difficulty` layer to the `new_model` definition.

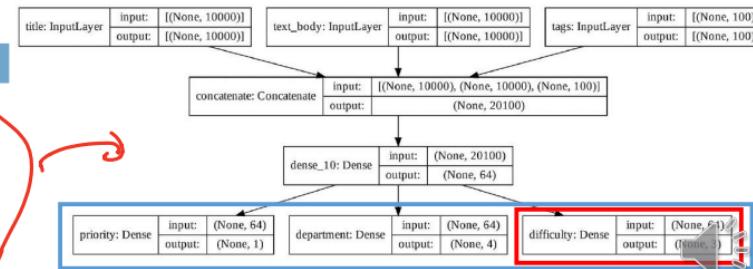


Figure 7.4 Plot of our new model

Subclassing the Model class

3

In the `__init__()` method, define the layers the model will use

In the `call()` method, define the forward pass of the model

Instantiate your subclass, and call it on data to create its weights

Listing 7.14 A simple subclassed model

```
class CustomerTicketModel(keras.Model):
    def __init__(self, num_departments):
        super().__init__()
        self.concat_layer = layers.concatenate()
        self.mixing_layer = layers.Dense(64, activation="relu")
        self.priority_scorer = layers.Dense(1, activation="sigmoid")
        self.department_classifier = layers.Dense(
            num_departments, activation="softmax")

    def call(self, inputs):
        title = inputs["title"]
        text_body = inputs["text_body"]
        tags = inputs["tags"]

        features = self.concat_layer([title, text_body, tags])
        features = self.mixing_layer(features)
        priority = self.priority_scorer(features)
        department = self.department_classifier(features)
        return priority, department
```

Don't forget to
call the `super()`
constructor!

Define
sublayers
in the
constructor.

Define the forward
pass in the `call()`
method.



Once you've defined the model, you can instantiate it.

```
model = CustomerTicketModel(num_departments=4)

priority, department = model(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data})
```

You can compile and train a Model subclass just like a Sequential or Functional model:

```
model.compile(optimizer="rmsprop",
              loss=["mean_squared_error", "categorical_crossentropy"],
              metrics=[["mean_absolute_error"], ["accuracy"]])

model.fit({"title": title_data,
           "text_body": text_body_data,
           "tags": tags_data},
           [priority_data, department_data],
           epochs=1)

model.evaluate({"title": title_data,
                "text_body": text_body_data,
                "tags": tags_data},
                [priority_data, department_data])

priority_preds, department_preds = model.predict({"title": title_data,
                                                 "text_body": text_body_data,
                                                 "tags": tags_data})
```

The structure of what you pass as the loss and metrics arguments must match exactly what gets returned by call()—here, a list of two elements.

The structure of the target data must match exactly what is returned by the call() method—here, a list of two elements.

The structure of the input data must match exactly what is expected by the call() method—here, a dict with keys title, text_body, and tags.



Beware: what subclassed models don't support

This freedom comes at a cost: with subclassed models, you are responsible for more of the model logic, which means your potential error surface is much larger



Mixing and matching different components

All models in the Keras API can smoothly interoperate with each other, whether they're Sequential models, Functional models, or subclassed models

Listing 7.15 Creating a Functional model that includes a subclassed model

```
class Classifier(keras.Model):

    def __init__(self, num_classes=2):
        super().__init__()
        if num_classes == 2:
            num_units = 1
            activation = "sigmoid"
        else:
            num_units = num_classes
            activation = "softmax"
        self.dense = layers.Dense(num_units, activation=activation)

    def call(self, inputs):
        return self.dense(inputs)

inputs = keras.Input(shape=(3,))
features = layers.Dense(64, activation="relu")(inputs)
outputs = Classifier(num_classes=10)(features)
model = keras.Model(inputs=inputs, outputs=outputs)
```



Inversely, you can use a Functional model as part of a subclassed layer or model

Listing 7.16 Creating a subclassed model that includes a Functional model

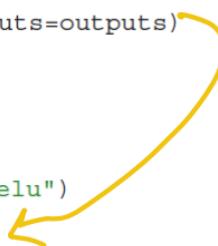
```
inputs = keras.Input(shape=(64,))
outputs = layers.Dense(1, activation="sigmoid")(inputs)
binary_classifier = keras.Model(inputs=inputs, outputs=outputs)

class MyModel(keras.Model):

    def __init__(self, num_classes=2):
        super().__init__()
        self.dense = layers.Dense(64, activation="relu")
        self.classifier = binary_classifier

    def call(self, inputs):
        features = self.dense(inputs)
        return self.classifier(features)

model = MyModel()
```



Using built-in training and evaluation loops

Listing 7.17 The standard workflow: `compile()`, `fit()`, `evaluate()`, `predict()`

```

from tensorflow.keras.datasets import mnist
def get_mnist_model():
    inputs = keras.Input(shape=(28 * 28,))
    features = layers.Dense(512, activation="relu")(inputs)
    features = layers.Dropout(0.5)(features)
    outputs = layers.Dense(10, activation="softmax")(features)
    model = keras.Model(inputs, outputs)
    return model
(images, labels), (test_images, test_labels) = mnist.load_data()
images = images.reshape((60000, 28 * 28)).astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28)).astype("float32") / 255
train_images, val_images = images[10000:], images[:10000]
train_labels, val_labels = labels[10000:], labels[:10000]

model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=3,
          validation_data=(val_images, val_labels))
test_metrics = model.evaluate(test_images, test_labels)
predictions = model.predict(test_images)

```

Use `evaluate()` to compute the loss and metrics on new data.

Use `predict()` to compute classification probabilities on new data.

Create a model (we factor this into a separate function so as to reuse it later).

Load your data, reserving some for validation.

Compile the model by specifying its optimizer, the loss function to minimize, and the metrics to monitor.

Use `fit()` to train the model, optionally providing validation data to monitor performance on unseen data.

Provide a custom metric

Pass callbacks to the `fit()` to schedule actions at specific points during training

easy





Writing your own metrics

A Keras metric is a subclass of the `keras.metrics.Metric` class

Write the state-update logic, which happens in the `update_state()` method.

Listing 7.18 Implementing a custom metric by subclassing the Metric class

```

import tensorflow as tf

class RootMeanSquaredError(keras.metrics.Metric):
    def __init__(self, name="rmse", **kwargs):
        super().__init__(name=name, **kwargs)
        self.mse_sum = self.add_weight(name="mse_sum", initializer="zeros")
        self.total_samples = self.add_weight(
            name="total_samples", initializer="zeros", dtype="int32")

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_true = tf.one_hot(y_true, depth=tf.shape(y_pred)[1])
        mse = tf.reduce_sum(tf.square(y_true - y_pred))
        self.mse_sum.assign_add(mse)
        num_samples = tf.shape(y_pred)[0]
        self.total_samples.assign_add(num_samples)

    def result(self):
        return tf.sqrt(self.mse_sum / tf.cast(self.total_samples, tf.float32))

    def reset_state(self):
        self.mse_sum.assign(0.)
        self.total_samples.assign(0.)

```

Define the state variables in the constructor. Like for layers, you have access to the `add_weight()` method.

To match our MNIST model, we expect categorical predictions and integer labels.

Subclass the Metric class.

Implement the state update logic in `update_state()`. The `y_true` argument is the targets (or labels) for one batch, while `y_pred` represents the corresponding predictions from the model. You can ignore the `sample_weight` argument—we won't use it here.

return current metric:

reset for each epoch:



Custom metrics can be used just like built-in ones

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy", RootMeanSquaredError()])
model.fit(train_images, train_labels,
          epochs=3,
          validation_data=(val_images, val_labels))
test_metrics = model.evaluate(test_images, test_labels)
```

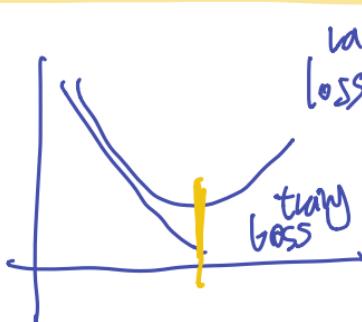


Using callbacks

Launching a training run on a large dataset for tens of epochs using `model.fit()` can be a bit like launching a paper airplane: you don't have any control over its trajectory or its landing spot.

A **callback** is an object (a class instance implementing specific methods) that is passed to the model in the call to `fit()`

It has access to all the available data about the state of the model and its performance, and it can take action: interrupt training, save a model, load a different weight set, or otherwise alter the state of the model.



Using callbacks

Model checkpointing—Saving the current state of the model at different points during training. keras.callbacks.ModelCheckpoint



Early stopping—Interrupting training when the validation loss is no longer improving (saving the best model obtained during training). keras.callbacks.EarlyStopping

Dynamically adjusting the value of certain parameters during training—Such as the learning rate of the optimizer. keras.callbacks.LearningRateScheduler, keras.callbacks.ReduceLROnPlateau

Logging training and validation metrics during training, or visualizing the representations learned by the model as they're updated. keras.callbacks.CSVLogger



The **EarlyStopping** callback interrupts training once a target metric being monitored has stopped improving for a fixed number of epochs

ModelCheckpoint lets you continually save the model during training

Listing 7.19 Using the callbacks argument in the fit() method

Callbacks are passed to the model via the callbacks argument in fit(), which takes a list of callbacks. You can pass any number of callbacks.

Saves the current weights after every epoch

Path to the destination model file

```
callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=2,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath="checkpoint_path.keras",
        monitor="val_loss",
        save_best_only=True,
    )
]
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          callbacks=callbacks_list,
          validation_data=(val_images, val_labels))
```

Interrupts training when improvement stops

Monitors the model's validation accuracy

Interrupts training when accuracy has stopped improving for two epochs

These two arguments mean you won't overwrite the model file unless val_loss has improved, which allows you to keep the best model seen during training.

You monitor accuracy, so it should be part of the model's metrics.

Note that because the callback will monitor validation loss and validation accuracy, you need to pass validation_data to the call to fit().



Writing your own callbacks

simple example that saves a list of per-batch loss values during training and saves a graph of these values at the end of each epoch.

Listing 7.20 Creating a custom callback by subclassing the `Callback` class

```
from matplotlib import pyplot as plt

class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs):
        self.per_batch_losses = [] list = [] loss = [] loss_end

    def on_batch_end(self, batch, logs):
        self.per_batch_losses.append(logs.get("loss")) loss = 2.50 loss = 0.50 loss = 0.30

    def on_epoch_end(self, epoch, logs):
        plt.clf()
        plt.plot(range(len(self.per_batch_losses)), self.per_batch_losses,
                  label="Training loss for each batch")
        plt.xlabel(f"Batch (epoch {epoch})")
        plt.ylabel("Loss")
        plt.legend()
        plt.savefig(f"plot_at_epoch_{epoch}.png") Import OS os.listdir() os.listdir()
        self.per_batch_losses = []
```

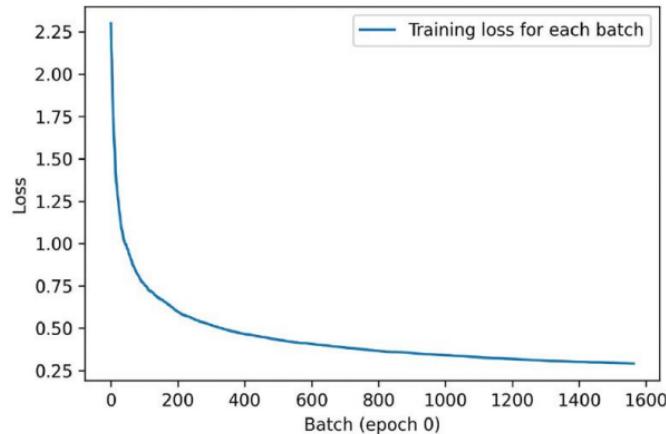
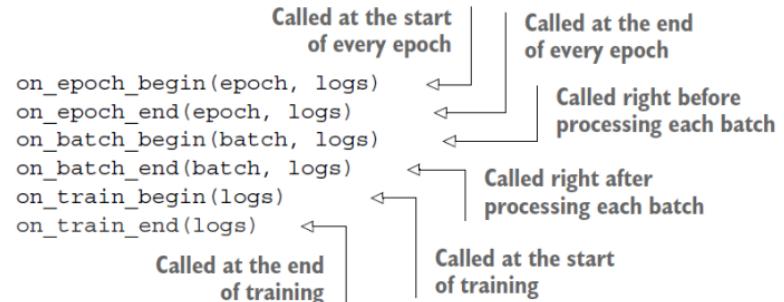


Figure 7.5 The output of our custom history plotting callback



Monitoring and visualization with TensorBoard

TensorBoard (www.tensorflow.org/tensorboard) is a browser-based application that you can run locally

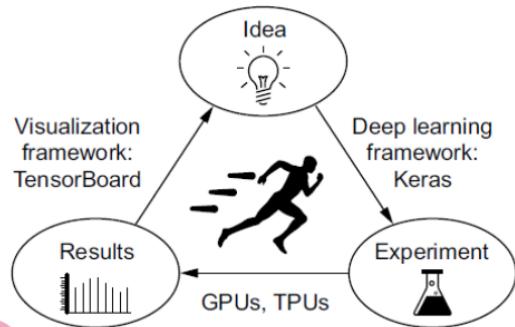


Figure 7.6 The loop of progress

- **Visually monitor metrics during training**
- **Visualize your model architecture**
- **Visualize histograms of activations and gradients**
- **Explore embeddings in 3D**



Easiest way to use TensorBoard with a Keras model and the fit() method is to use the keras.callbacks.TensorBoard callback.

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

tensorboard = keras.callbacks.TensorBoard(
    log_dir="/full_path_to_your_log_dir",
)
model.fit(train_images, train_labels,
          epochs=10,
          validation_data=(val_images, val_labels),
          callbacks=[tensorboard])
```

If you are running your script in a Colab notebook

```
%load_ext tensorboard
%tensorboard --logdir /full_path_to_your_log_dir
```

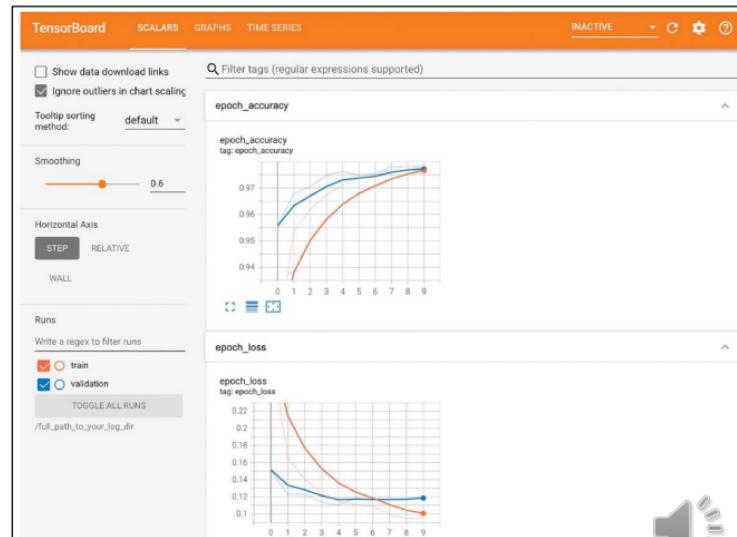


Figure 7.7 TensorBoard can be used for easy monitoring of training and evaluation metrics.



colab.research.google.com/github/fchollet/deep-learning-with-python-notebooks/blob/master/chapter07_working-with-keras.ipynb#scrollTo=5hNizEX2O8Sh

news study data science 학회사이트 업무관련 기타 Bookmark Manager Gmail YouTube 지도 뉴스 번역 SR Translate scm Data 학술지관련 기타

Share

Table of contents

- Subclassing the Model class
 - Rewriting our previous example as a subclassed model
- Beware: What subclassed models don't support
- Mixing and matching different components
- Remember: Use the right tool for the job
- Using built-in training and evaluation loops
- Writing your own metrics
- Using callbacks
 - The EarlyStopping and ModelCheckpoint callbacks
- Writing your own callbacks**
 - Monitoring and visualization with TensorBoard
- Writing your own training and evaluation loops
- Training versus inference
- Low-level usage of metrics
- A complete training and

+ Code + Text Copy to Drive

```
['plot_at_epoch_0.png',  
 'plot_at_epoch_2.png',  
 'sample_data']
```

RAM Disk

Editing

Monitoring and visualization with TensorBoard

```
[ ] model = get_mnist_model()  
model.compile(optimizer="rmsprop",  
              loss="sparse_categorical_crossentropy",  
              metrics=["accuracy"])  
  
tensorboard = keras.callbacks.TensorBoard(  
    log_dir="/full_path_to_your_log_dir",  
)  
model.fit(train_images, train_labels,  
          epochs=10,  
          validation_data=(val_images, val_labels),  
          callbacks=[tensorboard])  
  
[ ] %load_ext tensorboard  
%tensorboard --logdir /full_path_to_your_log_dir
```

TensorBoard

Writing your own training and evaluation loops

Training versus inference

0s completed at 4:03 PM

15°C 맑음

오후 4:08
2022-04-03

Writing your own training and evaluation loops

Built-in `fit()` workflow is solely focused on supervised learning, However, not every form of machine learning falls into this category.

Typical training loop look like this:

- 1. Run the forward pass inside a gradient tape to obtain a loss for the current batch of data.
- 2. Retrieve the gradients of the loss with regard to the model's weights.
- 3. Update the model's weights so as to lower the loss value on the current batch of data.

We will learn to reimplement `fit()`



Supervised-learning training step ends up looking like this:

```
def train_step(inputs, targets):  
    with tf.GradientTape() as tape:  
        predictions = model(inputs, training=True)  
        loss = loss_fn(targets, predictions)  
    gradients = tape.gradients(loss, model.trainable_weights)  
    optimizer.apply_gradients(zip(model.trainable_weights, gradients))
```

← training
3/22/2021

Set training = True: some Keras layers, such as the Dropout layer, have different behaviors during training and during inference

Layers and models own two kinds of weights: trainable weights, non-trainable weights



A complete training and evaluation loop

Training step function takes a batch of data and targets

Listing 7.21 Writing a step-by-step training loop: the training step function

```

model = get_mnist_model()
loss_fn = keras.losses.SparseCategoricalCrossentropy()
optimizer = keras.optimizers.RMSprop()
metrics = [keras.metrics.SparseCategoricalAccuracy()]
loss_tracking_metric = keras.metrics.Mean()

def train_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs, training=True)
        loss = loss_fn(targets, predictions)
        gradients = tape.gradient(loss, model.trainable_weights)
        optimizer.apply_gradients(zip(gradients, model.trainable_weights))

    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs[metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["loss"] = loss_tracking_metric.result()

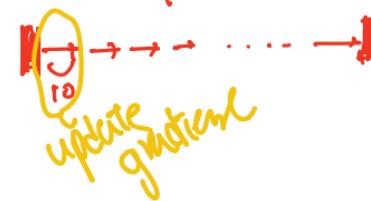
    return logs
  
```

Annotations and callouts:

- Prepare the loss function.
- Prepare the optimizer.
- Prepare the list of metrics to monitor.
- Prepare a Mean metric tracker to keep track of the loss average.
- Run the forward pass. Note that we pass `training=True`.
- Run the backward pass. Note that we use `model.trainable_weights`.
- Keep track of metrics.
- Keep track of the loss average.
- Return the current values of the metrics and the loss.

1000 data
10 : batch size

100 : batches
epoch 0



Reset the state of our metrics at the start of each epoch and before running evaluation.

Listing 7.22 Writing a step-by-step training loop: resetting the metrics

```
def reset_metrics():
    for metric in metrics:
        metric.reset_state()
    loss_tracking_metric.reset_state()
```



Our complete training loop

Listing 7.23 Writing a step-by-step training loop: the loop itself

```
training_dataset = tf.data.Dataset.from_tensor_slices(
    (train_images, train_labels))
training_dataset = training_dataset.batch(32)
epochs = 3
for epoch in range(epochs):
    reset_metrics() ← 7.22
    for inputs_batch, targets_batch in training_dataset:
        logs = train_step(inputs_batch, targets_batch)
    print(f"Results at the end of epoch {epoch}")
    for key, value in logs.items():
        print(f"...{key}: {value:.4f}")
```



The evaluation loop

Listing 7.24 Writing a step-by-step evaluation loop

72 32

TOP evaluation loop

```

def test_step(inputs, targets):
    predictions = model(inputs, training=False) ← Note that we pass
    loss = loss_fn(targets, predictions)
    training=False.

    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["val_loss"] = loss_tracking_metric.result()
    return logs

val_dataset = tf.data.Dataset.from_tensor_slices((val_images, val_labels))
val_dataset = val_dataset.batch(32)
reset_metrics() ← WA
for inputs_batch, targets_batch in val_dataset:
    logs = test_step(inputs_batch, targets_batch)
print("Evaluation results:")
for key, value in logs.items():
    print(f"...{key}: {value:.4f}")

```



Make it fast with `tf.function`

Listing 7.25 Adding a `@tf.function` decorator to our evaluation-step function

```
@tf.function ← GRV
def test_step(inputs, targets):
    predictions = model(inputs, training=False)
    loss = loss_fn(targets, predictions)

    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["val_loss"] = loss_tracking_metric.result()
    return logs

val_dataset = tf.data.Dataset.from_tensor_slices((val_images, val_labels))
val_dataset = val_dataset.batch(32)
reset_metrics()

for inputs_batch, targets_batch in val_dataset:
    logs = test_step(inputs_batch, targets_batch)
print("Evaluation results:")
for key, value in logs.items():
    print(f"...{key}: {value:.4f}")
```

This is the
only line that
changed.

On the Colab CPU, we go from taking 1.80 s to run the evaluation loop to only 0.8 s.



Leveraging fit() with a custom training loop

There's actually a middle ground between fit() and a training loop written from scratch: you can provide a custom training step function and let the framework do the rest.

Listing 7.26 Implementing a custom training step to use with fit()

```
loss_fn = keras.losses.SparseCategoricalCrossentropy()
loss_tracker = keras.metrics.Mean(name="loss")
```

This metric object will be used to track the average of per-batch losses during training and evaluation.

```
class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True)
            loss = loss_fn(targets, predictions)
        gradients = tape.gradient(loss, model.trainable_weights)
        optimizer.apply_gradients(zip(gradients, model.trainable_weights))

        loss_tracker.update_state(loss)
        return {"loss": loss_tracker.result()}
```

We override the train_step method.

We use self(inputs, training=True) instead of model(inputs, training=True), since our model is the class itself.

We update the loss tracker metric that tracks the average of the loss.

We return the average loss so far by querying the loss tracker metric.

```
@property
def metrics(self):
    return [loss_tracker]
```

Any metric you would like to reset across epochs should be listed here.



news study data science 학회사이트 업무관련 기타 Bookmark Manager Gmail YouTube 지도 뉴스 번역 SR Translate scm Data 학술지관련 기타

Share

chapter07_working-with-keras.i

File Edit View Insert Runtime Tools Help Cannot save changes

Table of contents

- Remember: Use the right tool for the job
- Using built-in training and evaluation loops
 - Writing your own metrics
 - Using callbacks
 - The EarlyStopping and ModelCheckpoint callbacks
 - Writing your own callbacks
 - Monitoring and visualization with TensorBoard
 - Writing your own training and evaluation loops
 - Training versus inference
 - Low-level usage of metrics**
 - A complete training and evaluation loop
 - Make it fast with tf.function
 - Leveraging fit() with a custom training loop
- Summary
- Section

+ Code + Text Copy to Drive RAM Disk Editing

Training versus inference

Low-level usage of metrics

```
metric = keras.metrics.SparseCategoricalAccuracy()  
targets = [0, 1, 2]  
predictions = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]  
metric.update_state(targets, predictions)  
current_result = metric.result()  
print(f"result: {current_result:.2f}")
```

result: 1.00

```
[16] values = [0, 1, 2, 3, 4]  
mean_tracker = keras.metrics.Mean()  
for value in values:  
    mean_tracker.update_state(value)  
print(f"Mean of values: {mean_tracker.result():.2f}")
```

Mean of values: 2.00

A complete training and evaluation loop

Writing a step-by-step training loop: the training step function

0s completed at 4:56 PM

검색하려면 여기에 입력하십시오.

15°C 맑음

오후 4:56
2022-04-03

Thank you!