Deep Learning

Lecture Topic:
Working with Keras: A Deep Dive

Anxiao (Andrew) Jiang

Learning Objectives:

1. Understand how to build models using sequential API.

2. Understand how to build models using functional API.

3. Understand how to build models using subclassing.

Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

Different ways to build Keras models

There are three APIs for building models in Keras (see figure 7.1):

- The *Sequential model*, the most approachable API—it's basically a Python list. As such, it's limited to simple stacks of layers.
- The *Functional API*, which focuses on graph-like model architectures. It represents a nice mid-point between usability and flexibility, and as such, it's the most commonly used model-building API.
- *Model subclassing*, a low-level option where you write everything yourself from scratch. This is ideal if you want full control over every little thing. However, you won't get access to many built-in Keras features, and you will be more at risk of making mistakes.

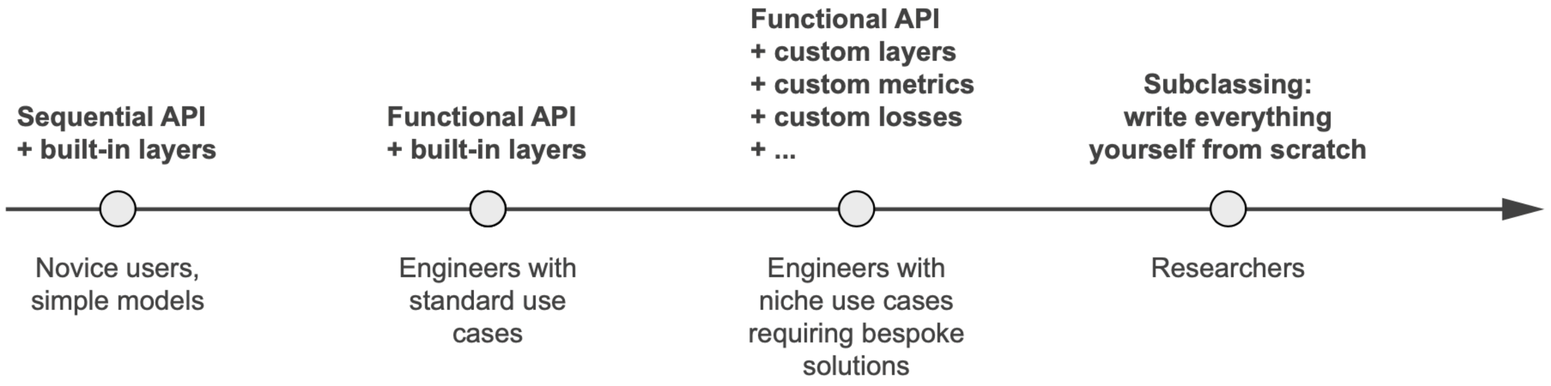


Figure 7.1 Progressive disclosure of complexity for model building

Sequential Model

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")
])
```

Note that it's possible to build the same model incrementally via the `add()` method, which is similar to the `append()` method of a Python list.

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"))

model.build((None, 3))
```

Input size

Sequential model

After the model is built, you can display its contents via the `summary()` method, which comes in handy for debugging.

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

Sequential model: Name model and even layers

Listing 7.6 Naming models and layers with the `name` argument

```
>>> model = keras.Sequential(name="my_example_model")
>>> 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"
```

Input size

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
my_first_layer (Dense)       (None, 64)                256
_________________________________________________________________
my_last_layer (Dense)        (None, 10)                650
=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
```

Sequential model: Specify input size 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.

Quiz questions:

1. How to build a sequential model?

2. Can all models be built using sequential API?

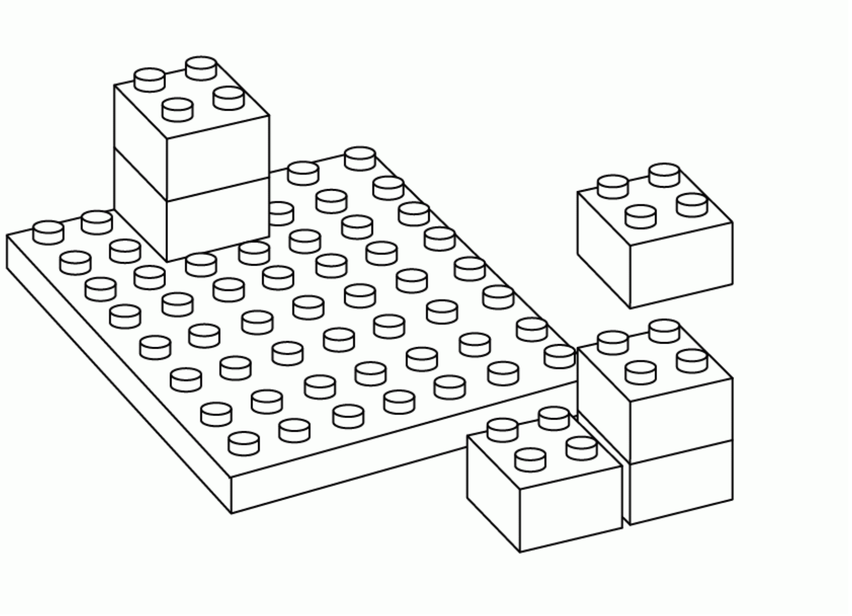
Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

Functional API

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)
```



Functional API

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

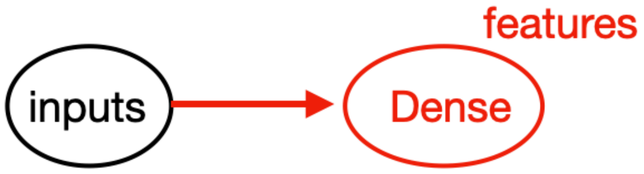
The model will process batches where each sample has shape (3,). The number of samples per batch is variable (indicated by the None batch size).

These batches will have dtype float32.

Functional API

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)
```

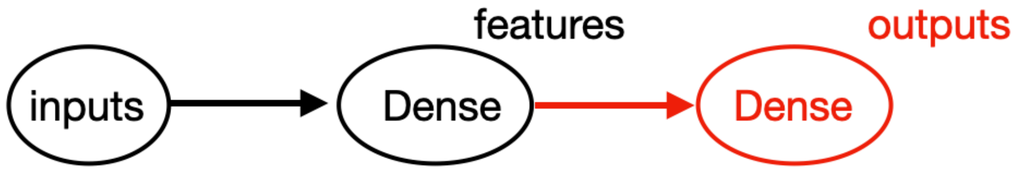



```
>>> features.shape
(None, 64)
```

Functional API

Listing 7.8 A simple Functional model with two `Dense` layers

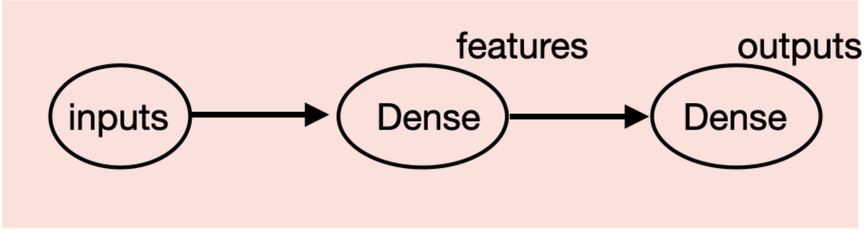
```
inputs = keras.Input(shape=(3,), name="my_input")
features = layers.Dense(64, activation="relu")(inputs)
outputs = layers.Dense(10, activation="softmax")(features)
model = keras.Model(inputs=inputs, outputs=outputs)
```



Functional API

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)
```



model

```
>>> model.summary()
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
my_input (InputLayer)        [(None, 3)]               0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                256
_________________________________________________________________
dense_7 (Dense)              (None, 10)                650
=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
```

Quiz questions:

1. How to build a model using functional API?

2. Can all models be built using functional API?

Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

Functional API: Multi-Input, Multi-Output Model

Let's say you're building a system to rank customer support tickets by priority and route them to the appropriate department. Your model has three inputs:

- The title of the ticket (text input)
- The text body of the ticket (text input)
- Any tags added by the user (categorical input, assumed here to be one-hot encoded)

Your model also has two outputs:

- The priority score of the ticket, a scalar between 0 and 1 (sigmoid output)
- The department that should handle the ticket (a softmax over the set of departments)

Functional API: Multi-Input, Multi-Output Model

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

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

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")

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

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])
```

Define model inputs.

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

Define model outputs.

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

Create the model by specifying its inputs and outputs.

Functional API: Multi-Input, Multi-Output Model

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

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

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")

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

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])
```

Define model inputs.

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

Define model outputs.

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

Create the model by specifying its inputs and outputs.

title

Functional API: Multi-Input, Multi-Output Model

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

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

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")

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

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])
```

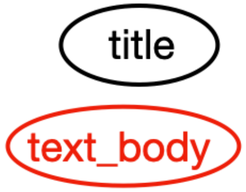
Define model inputs.

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

Define model outputs.

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

Create the model by specifying its inputs and outputs.



Functional API: Multi-Input, Multi-Output Model

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

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

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")

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

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])
```

Define model inputs.

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

Define model outputs.

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

Create the model by specifying its inputs and outputs.

Functional API: Multi-Input, Multi-Output Model

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

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

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")

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

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])
```

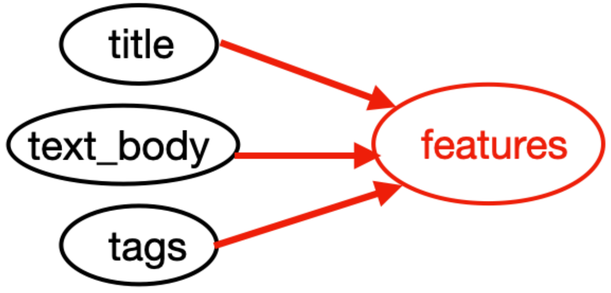
Define model inputs.

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

Define model outputs.

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

Create the model by specifying its inputs and outputs.



Functional API: Multi-Input, Multi-Output Model

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

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

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")

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

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])
```

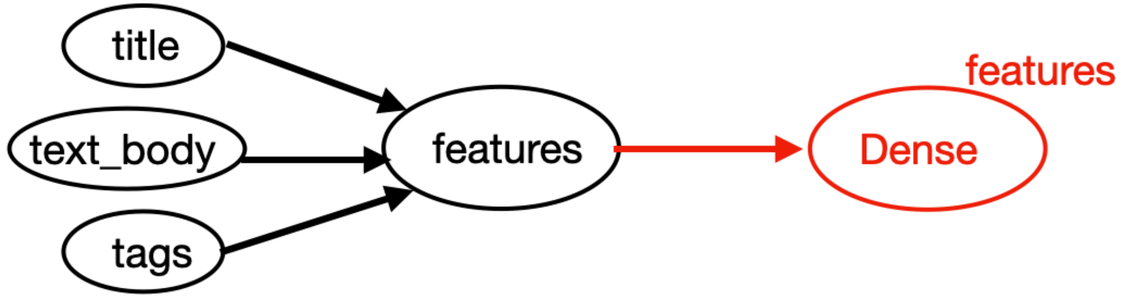
Define model inputs.

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

Define model outputs.

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

Create the model by specifying its inputs and outputs.



Functional API: Multi-Input, Multi-Output Model

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

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

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")

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

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])
```

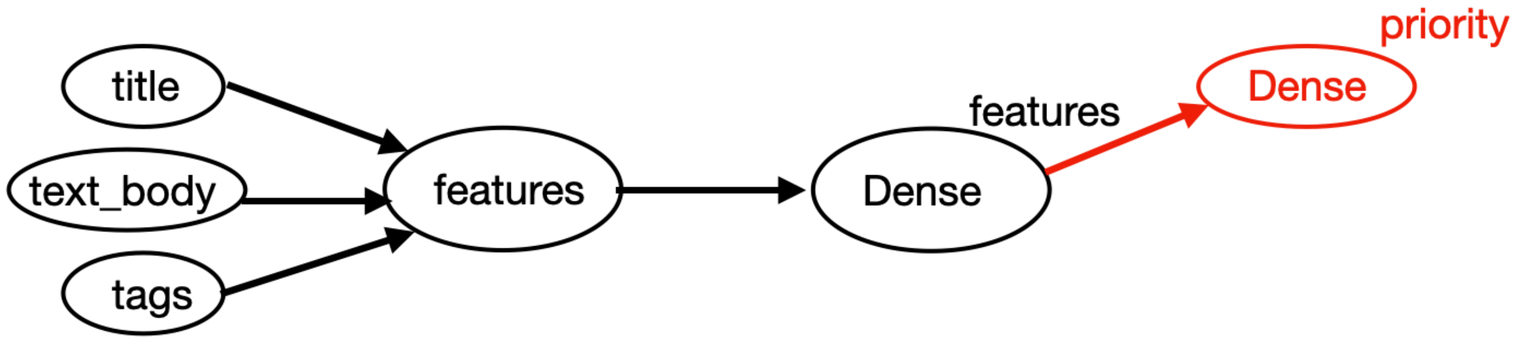
Define model inputs.

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

Define model outputs.

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

Create the model by specifying its inputs and outputs.



Functional API: Multi-Input, Multi-Output Model

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

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

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")

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

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])
```

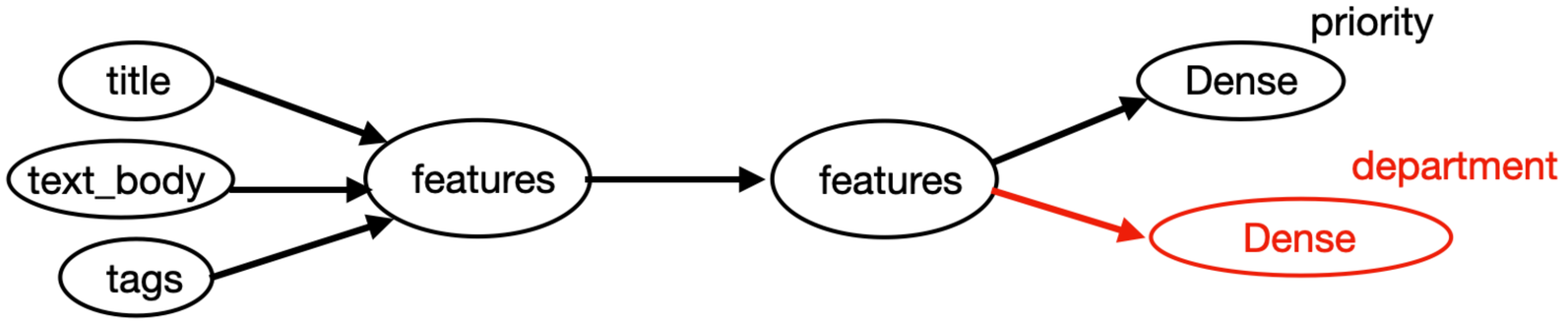
Define model inputs.

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

Define model outputs.

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

Create the model by specifying its inputs and outputs.



Functional API: Multi-Input, Multi-Output Model

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

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

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")

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

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])
```

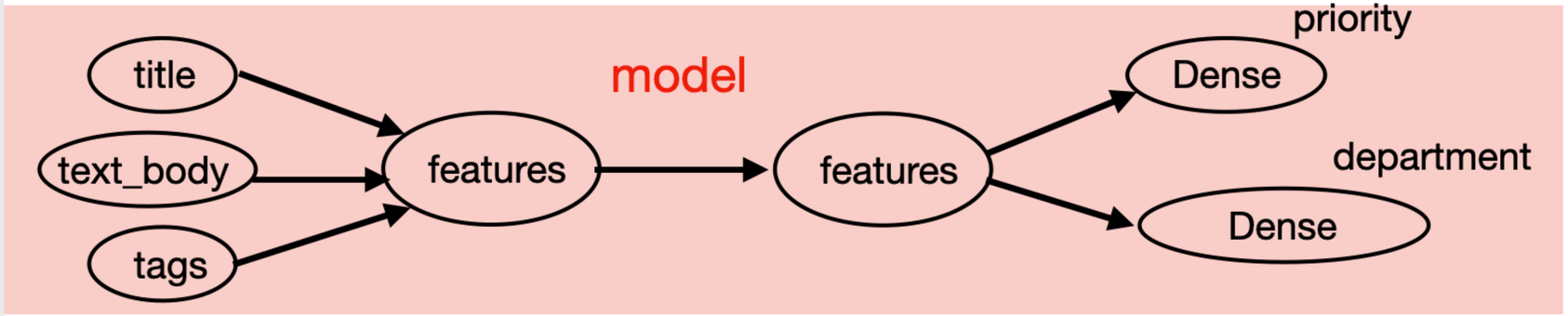
Define model inputs.

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

Define model outputs.

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

Create the model by specifying its inputs and outputs.



Functional API: Train 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])
```

Functional API: Train Multi-Input, Multi-Output Model

If you don't want to rely on input order (for instance, because you have many inputs or outputs), 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})
```

Functional API: Multi-Input, Multi-Output Model

Let's visualize the connectivity of the model we just defined (the *topology* of the model). You can plot a Functional model as a graph with the `plot_model()` utility (see figure 7.2).

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

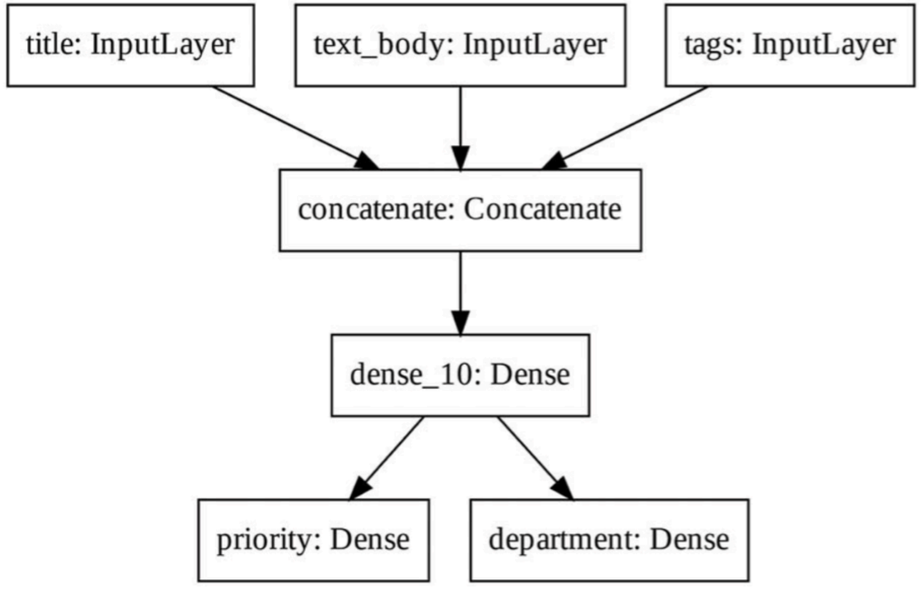


Figure 7.2 Plot generated by `plot_model()` on our ticket classifier model

Functional API: Multi-Input, Multi-Output Model

You can add to this plot the input and output shapes of each layer in the model, which can be helpful during debugging (see figure 7.3).

```
keras.utils.plot_model(
    model, "ticket_classifier_with_shape_info.png", show_shapes=True)
```

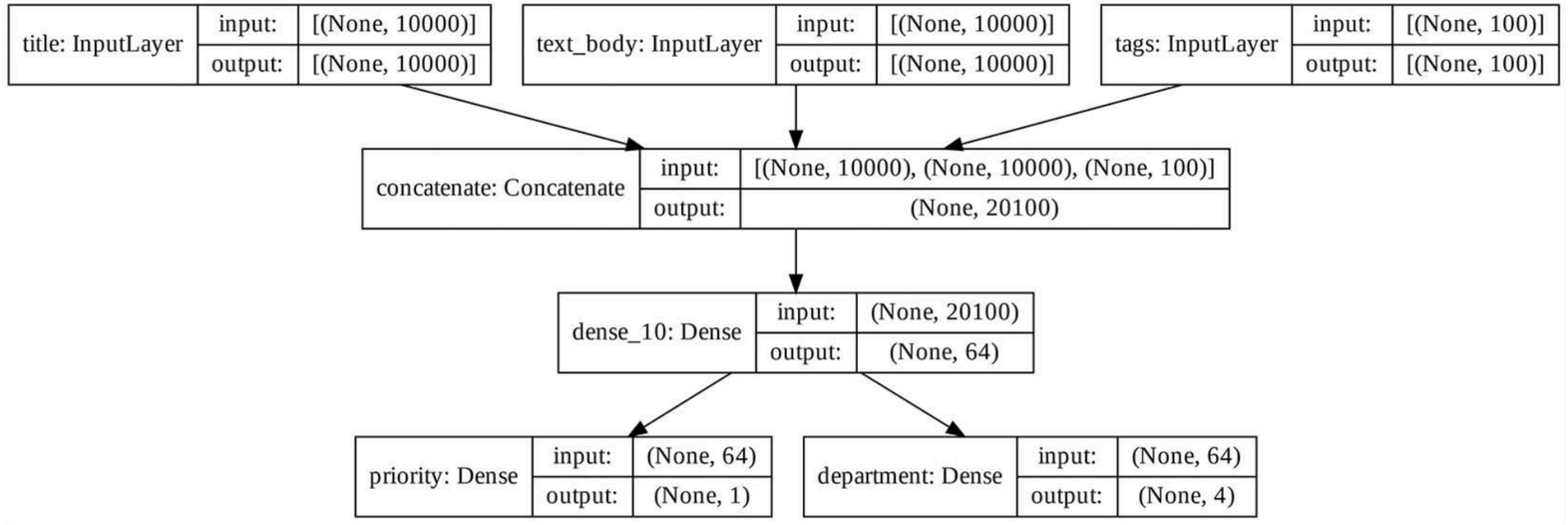


Figure 7.3 Model plot with shape information added

Quiz questions:

1. How to build a multi-input multi-output model?

2. Why are multi-input multi-output models useful?

Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

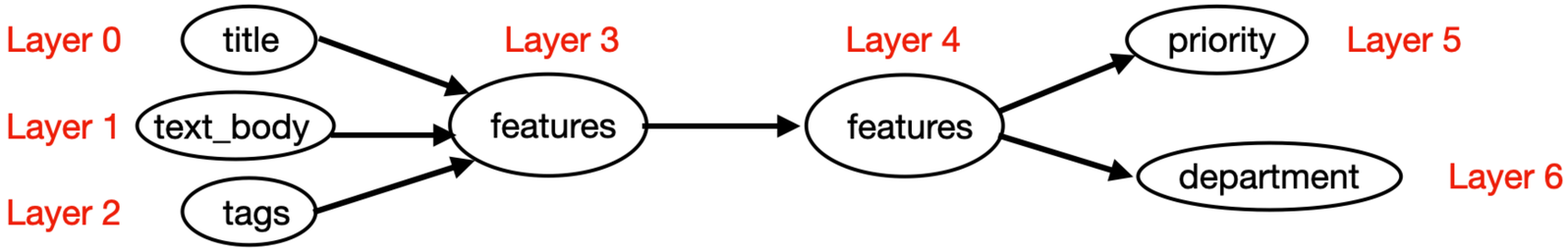
Functional API: Multi-Input, Multi-Output Model

Access to layer connectivity also means that you can inspect and reuse individual nodes (layer calls) in the graph. The `model.layers` model property provides the list of layers that make up the model, and for each layer you can query `layer.input` and `layer.output`.

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>
```

Functional API: Multi-Input, Multi-Output Model



Functional API: Multi-Input, Multi-Output Model

This enables you to do *feature extraction*, creating models that reuse intermediate features from another model.

Let's say you want to add another output to the previous model—you want to estimate how long a given issue ticket will take to resolve, a kind of difficulty rating. You could do this via a classification layer over three categories: "quick," "medium," and "difficult." You don't need to recreate and retrain a model from scratch. You can start from the intermediate features of your previous model, since you have access to them, like this.

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])
```

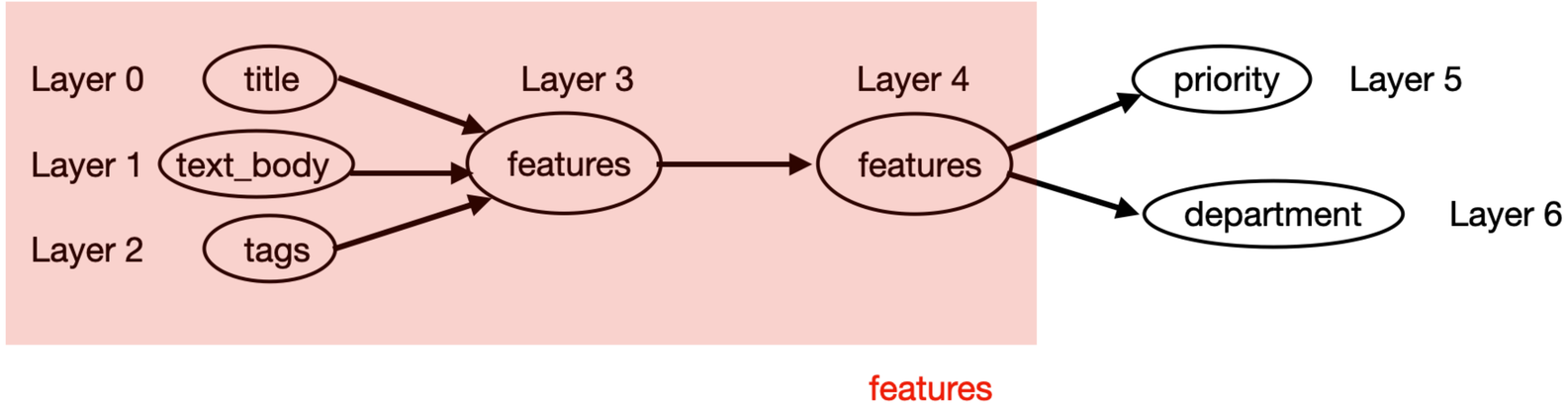
layers[4] is our intermediate Dense layer

Functional API: Multi-Input, Multi-Output Model

Listing 7.13 Creating a new model by reusing intermediate layer outputs

layers[4] is our intermediate Dense layer

```
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])
```

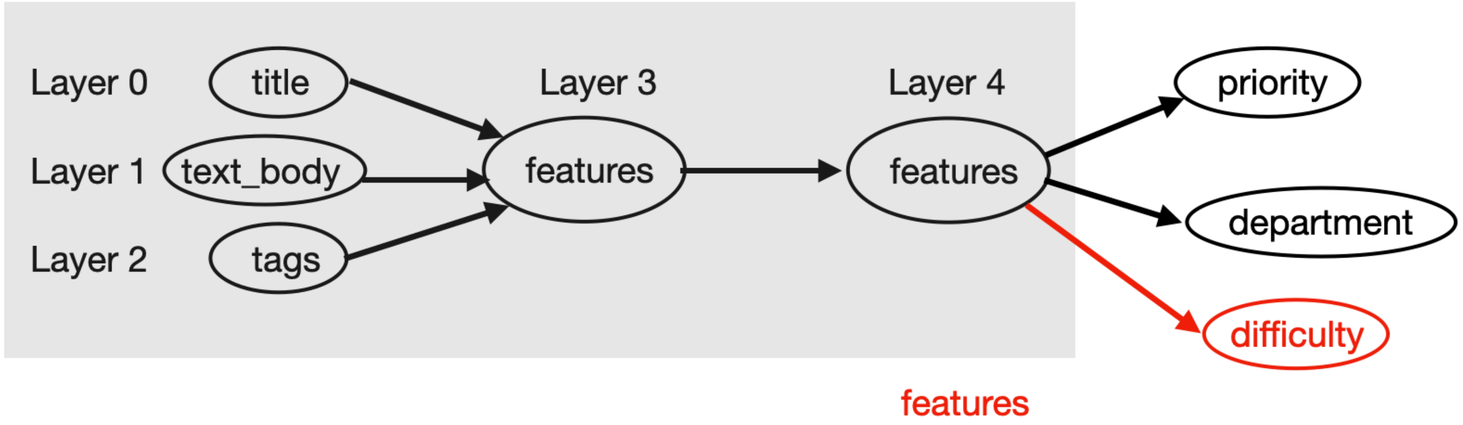


Functional API: Multi-Input, Multi-Output Model

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])
```

layers[4] is our intermediate Dense layer



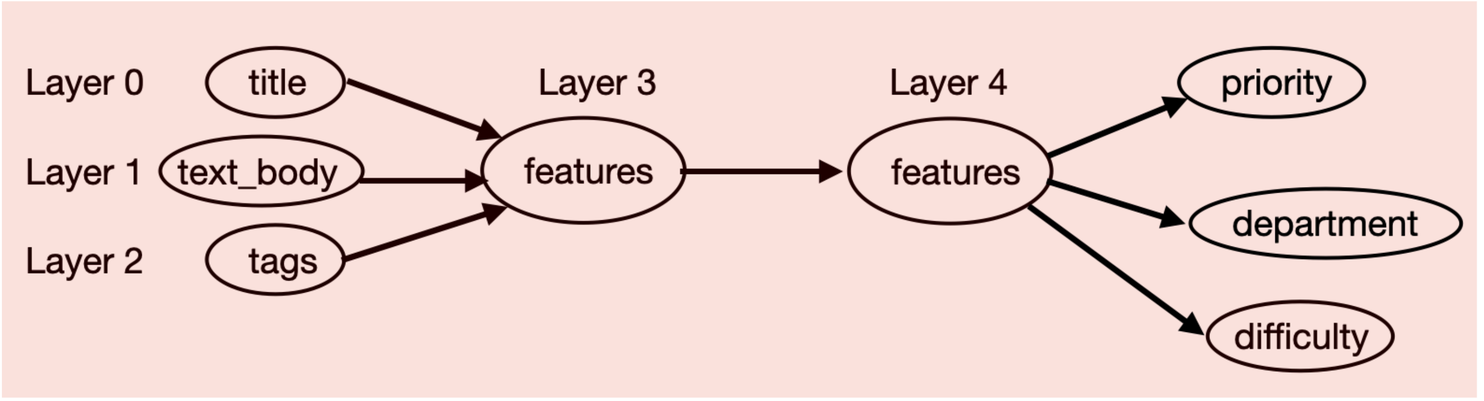
Functional API: Multi-Input, Multi-Output Model

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)
```

layers[4] is our intermediate Dense layer

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



new_model

Functional API: Multi-Input, Multi-Output Model

Let's plot our new model (see figure 7.4):

```
keras.utils.plot_model(
    new_model, "updated_ticket_classifier.png", show_shapes=True)
```

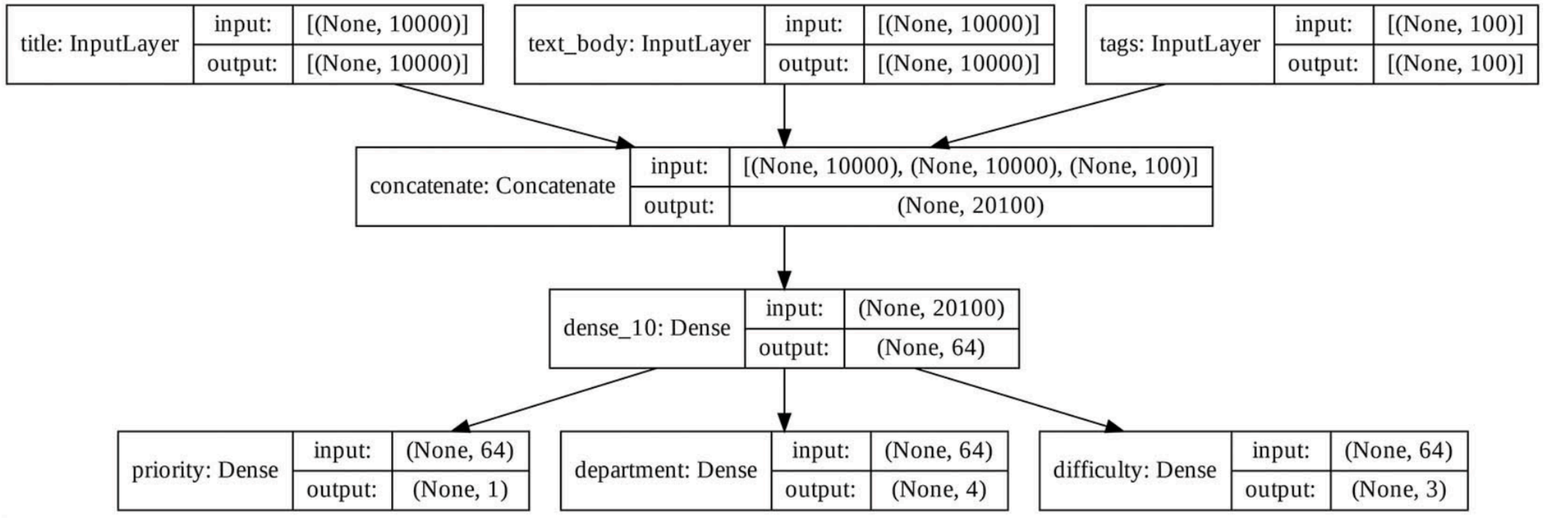


Figure 7.4 Plot of our new model

Quiz questions:

1. How to reuse a layer or a model?

2. When is it useful to reuse a layer or a model?

Roadmap of this lecture:

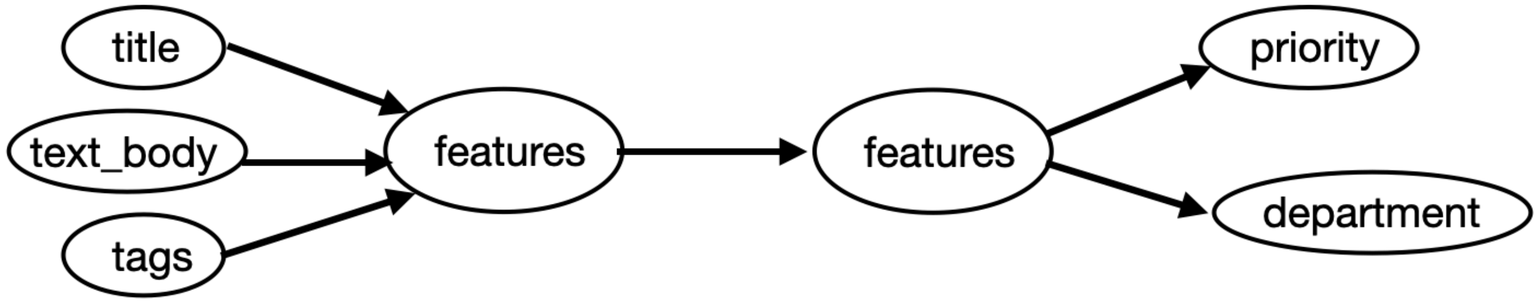
1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

Subclassing the Model Class

The last model-building pattern you should know about is the most advanced one: `Model` subclassing. You learned in chapter 3 how to subclass the `Layer` class to create custom layers. Subclassing `Model` is pretty similar:

- In the `__init__()` method, define the layers the model will use.
- In the `call()` method, define the forward pass of the model, reusing the layers previously created.
- Instantiate your subclass, and call it on data to create its weights.

Subclassing the Model Class Example:
Customer Support Ticket Management



Subclassing the Model Class Example: Customer Support Ticket Management

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.

Subclassing the Model Class Example: Customer Support Ticket Management

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.

Subclassing the Model Class Example: Customer Support Ticket Management

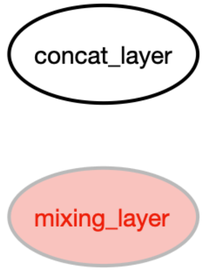
Listing 7.14 A simple subclassed model

```
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.



Subclassing the Model Class Example: Customer Support Ticket Management

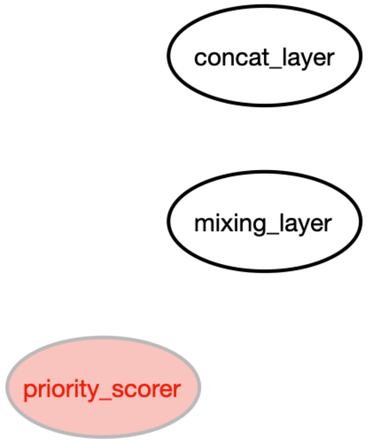
Listing 7.14 A simple subclassed model

```
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.



Subclassing the Model Class Example: Customer Support Ticket Management

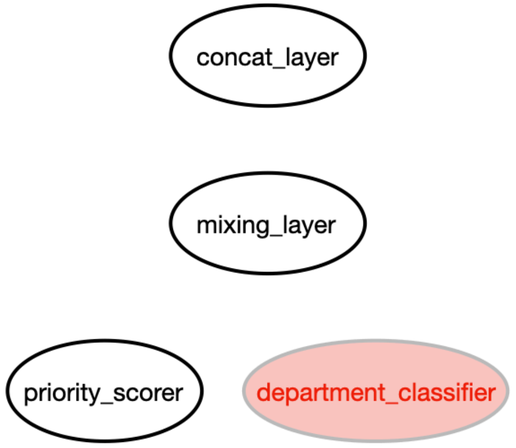
Listing 7.14 A simple subclassed model

```
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.



Subclassing the Model Class Example: Customer Support Ticket Management

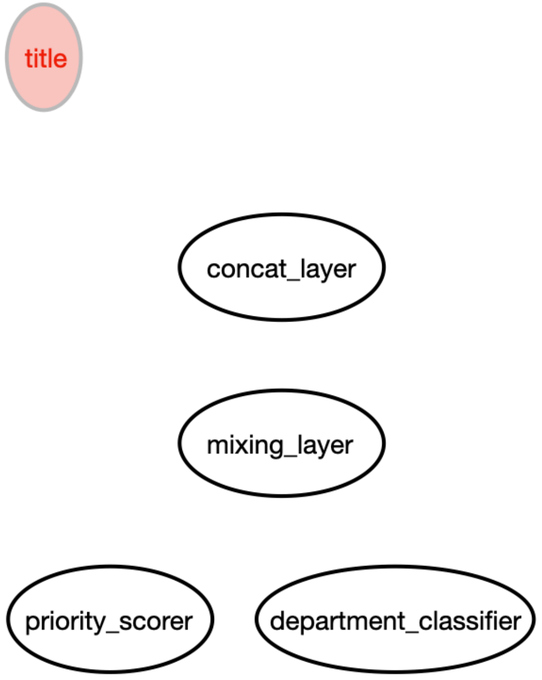
Listing 7.14 A simple subclassed model

```
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.



Subclassing the Model Class Example: Customer Support Ticket Management

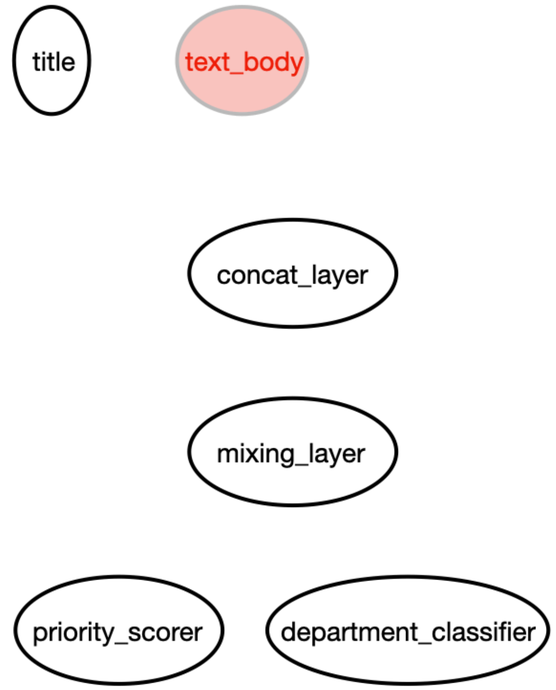
Listing 7.14 A simple subclassed model

```
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.



Subclassing the Model Class Example: Customer Support Ticket Management

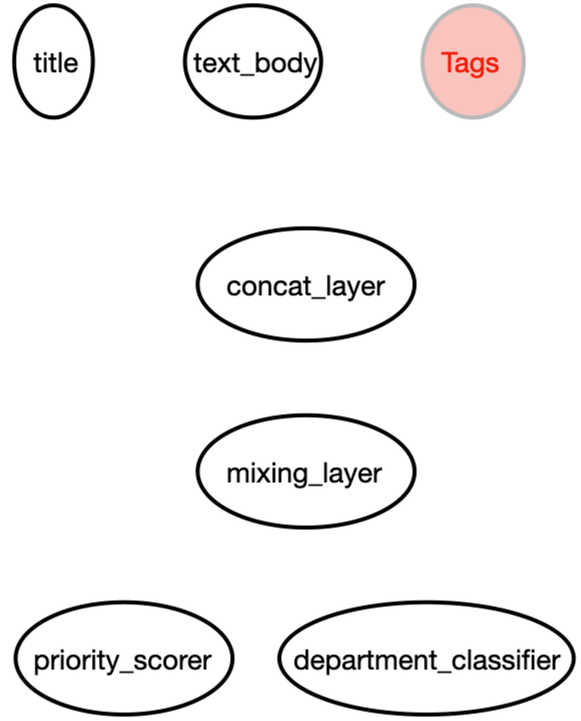
Listing 7.14 A simple subclassed model

```
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.



Subclassing the Model Class Example: Customer Support Ticket Management

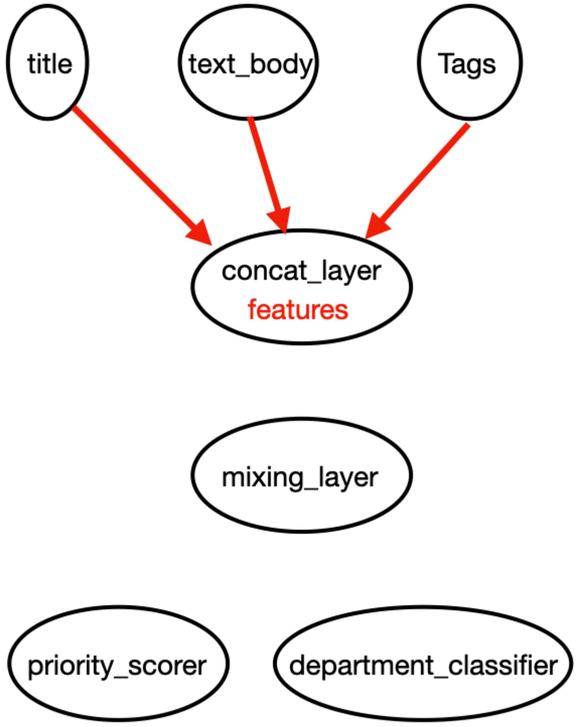
Listing 7.14 A simple subclassed model

```
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.



Subclassing the Model Class Example: Customer Support Ticket Management

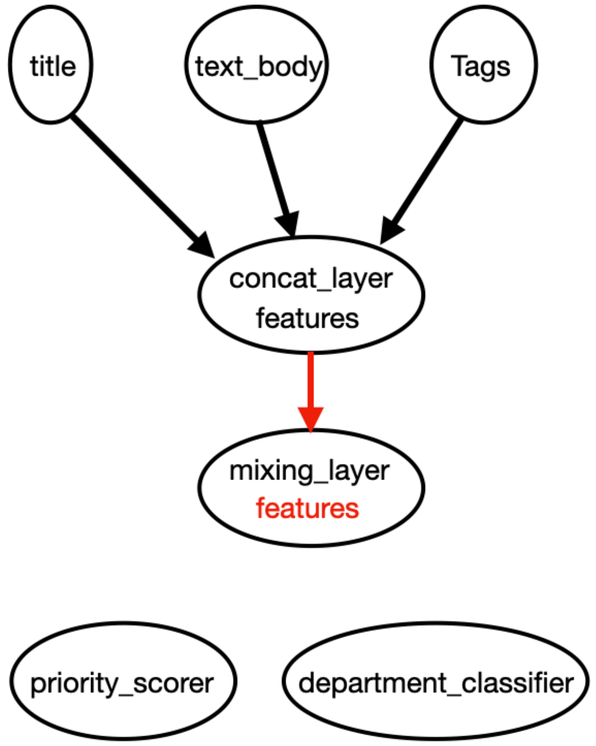
Listing 7.14 A simple subclassed model

```
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.



Subclassing the Model Class Example: Customer Support Ticket Management

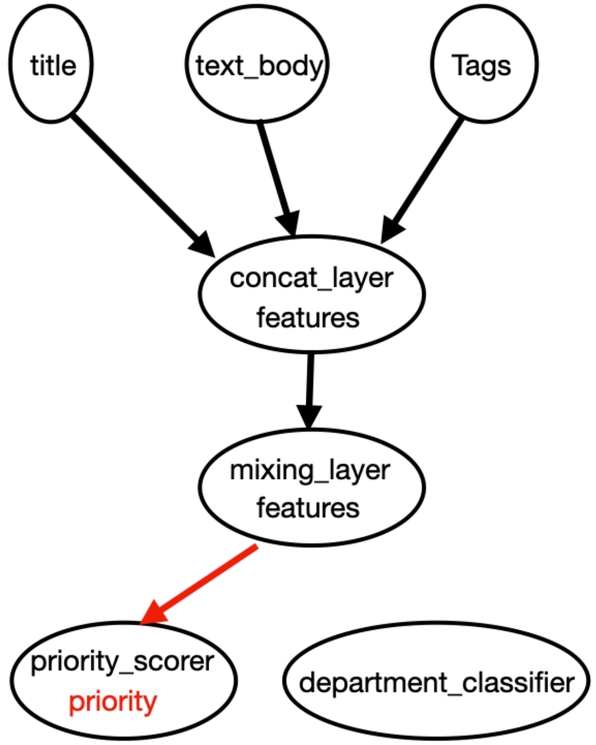
Listing 7.14 A simple subclassed model

```
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.



Subclassing the Model Class Example: Customer Support Ticket Management

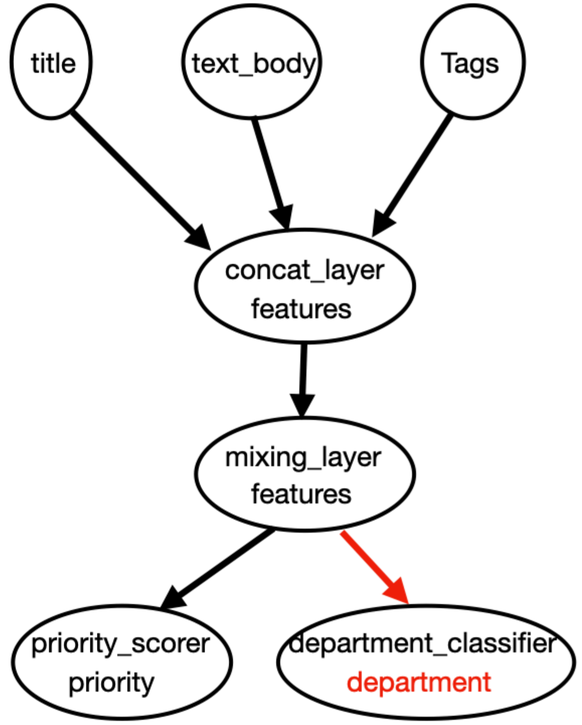
Listing 7.14 A simple subclassed model

```
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
```

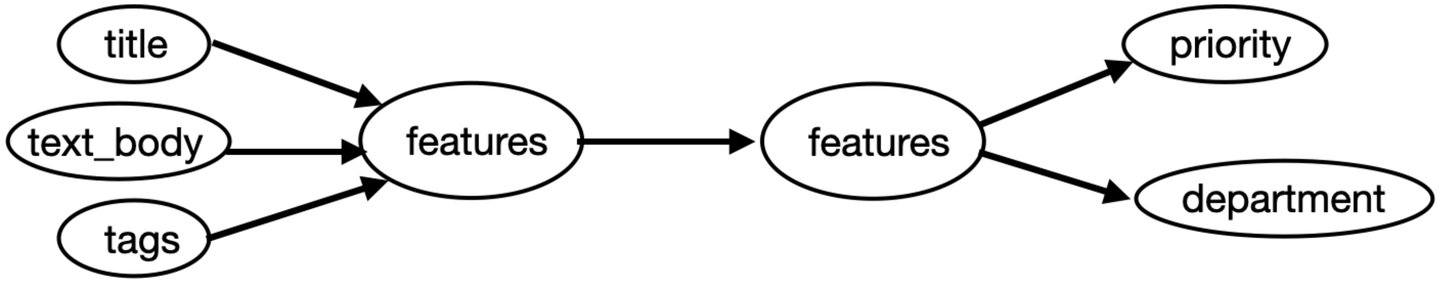
Don't forget to call the super() constructor!

Define sublayers in the constructor.

Define the forward pass in the call() method.



Subclassing the Model Class Example:
Customer Support Ticket Management



Once you've defined the model, you can instantiate it. Note that it will only create its weights the first time you call it on some data, much like `Layer` subclasses:

```
model = CustomerTicketModel(num_departments=4)

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

Subclassing the Model Class Example: Customer Support Ticket Management

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.

Quiz questions:

1. How to build a model using sub-classing?

2. Why is sub-classing useful?

Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

The `Model` subclassing workflow is the most flexible way to build a model. It enables you to build models that cannot be expressed as directed acyclic graphs of layers—imagine, for instance, a model where the `call()` method uses layers inside a `for` loop, or even calls them recursively. Anything is possible—you're in charge.

Mixing Sequential Model, Functional API, and Model Subclassing

For instance, you can use a subclassed layer or model in a Functional model.

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)
```

Model Subclassing

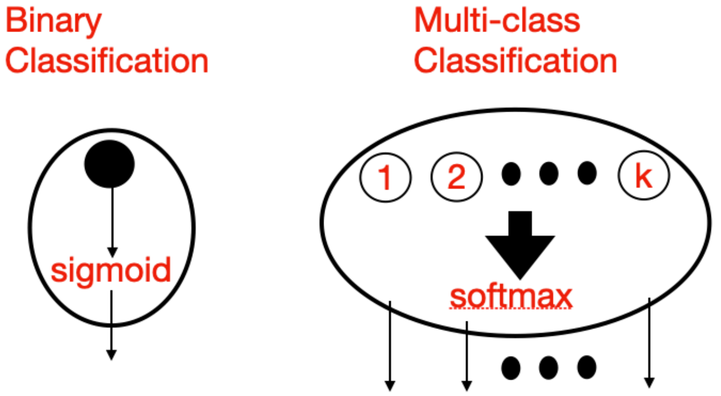
Functional API

Mixing Sequential Model, Functional API, and Model Subclassing

For instance, you can use a subclassed layer or model in a Functional model.

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)
```

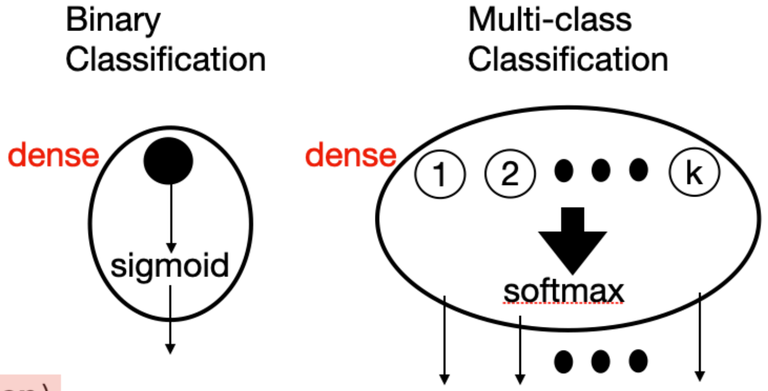


Mixing Sequential Model, Functional API, and Model Subclassing

For instance, you can use a subclassed layer or model in a Functional model.

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)
```

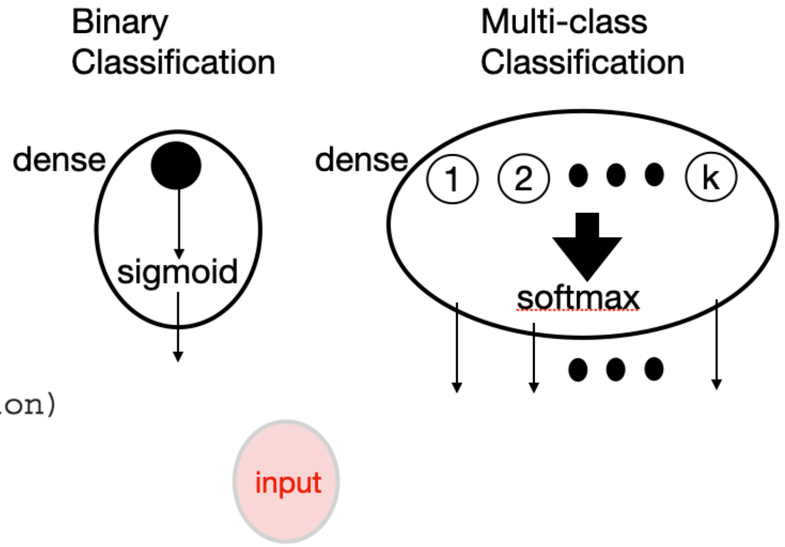


Mixing Sequential Model, Functional API, and Model Subclassing

For instance, you can use a subclassed layer or model in a Functional model.

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)
```

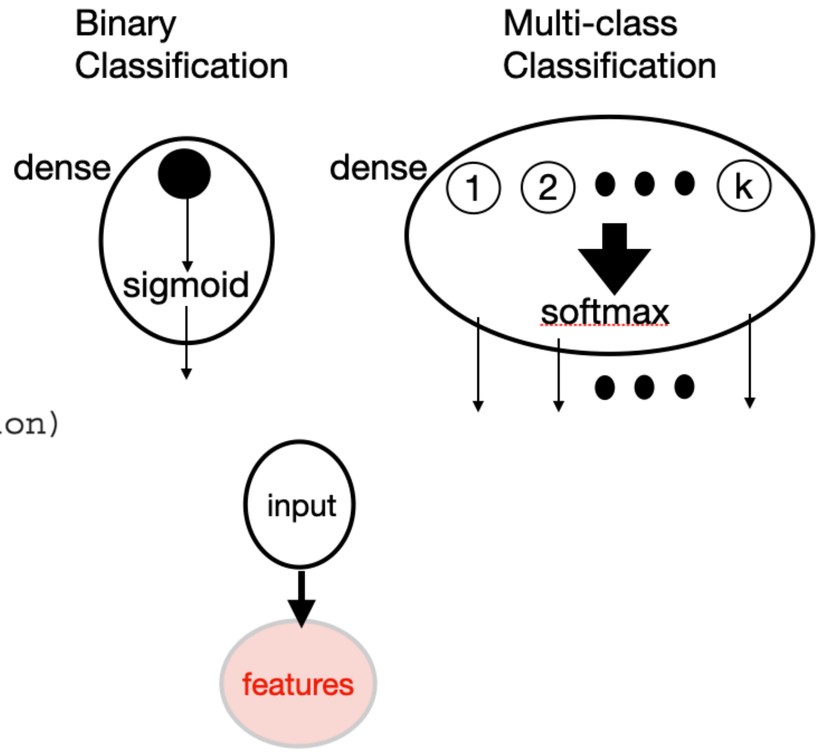


Mixing Sequential Model, Functional API, and Model Subclassing

For instance, you can use a subclassed layer or model in a Functional model.

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)
```

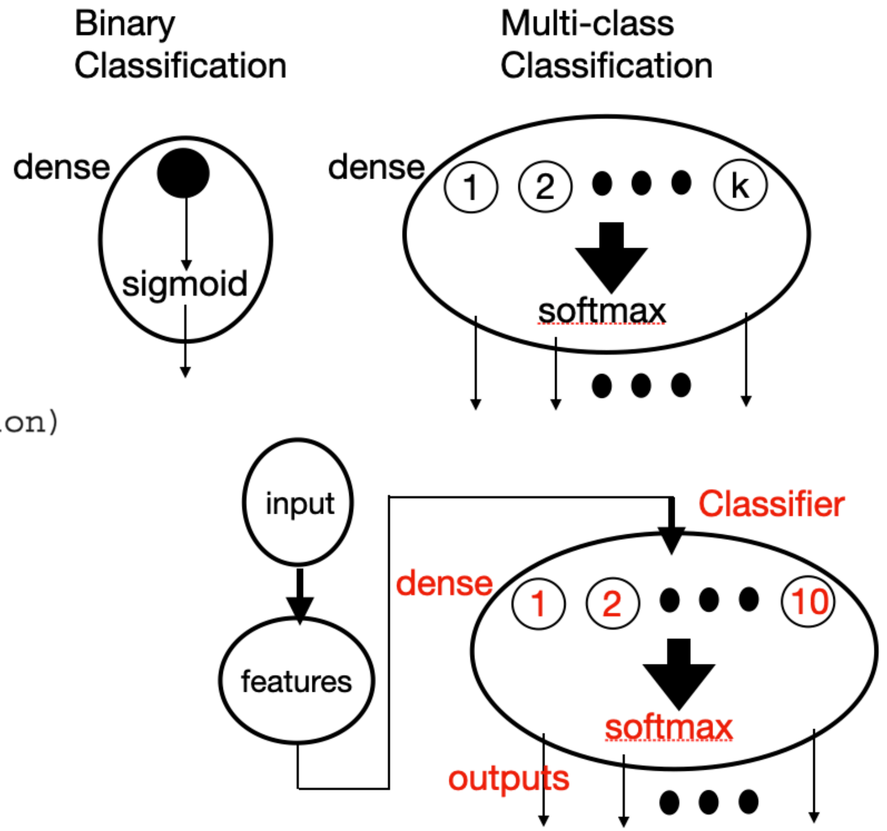


Mixing Sequential Model, Functional API, and Model Subclassing

For instance, you can use a subclassed layer or model in a Functional model.

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)
```

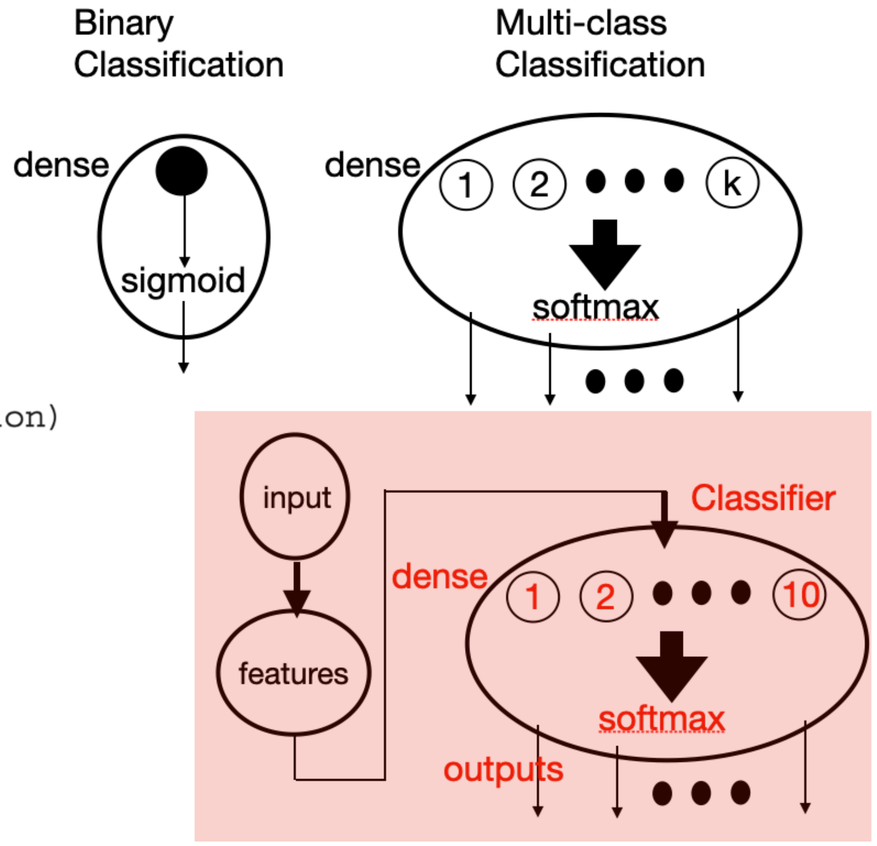


Mixing Sequential Model, Functional API, and Model Subclassing

For instance, you can use a subclassed layer or model in a Functional model.

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)
```



Quiz question:

1. How to build a flexible model using if, loop, etc.?

Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

Mixing Sequential Model, Functional API, and Model Subclassing: Another Example

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

Functional API

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

Model Subclassing

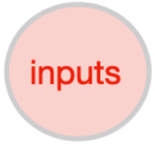
```
class MyModel(keras.Model):

    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()
```

Mixing Sequential Model, Functional API, and Model Subclassing: Another Example

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()
```

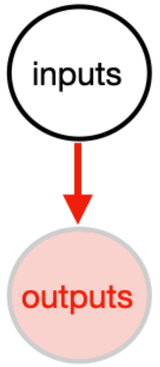


Mixing Sequential Model, Functional API, and Model Subclassing: Another Example

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()
```

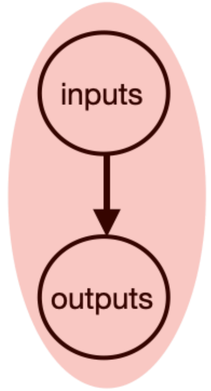


Mixing Sequential Model, Functional API, and Model Subclassing: Another Example

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)
```



binary_classifier

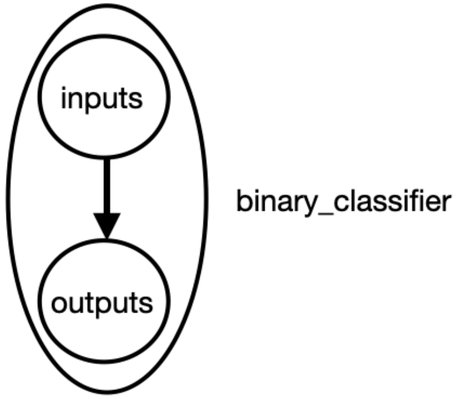
```
class MyModel(keras.Model):

    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()
```

Mixing Sequential Model, Functional API, and Model Subclassing: Another Example

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)
```



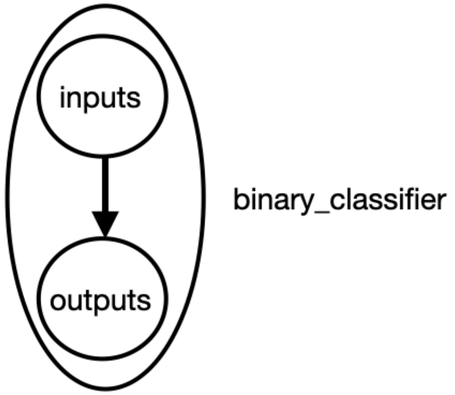

```
class MyModel(keras.Model):

    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()
```

Mixing Sequential Model, Functional API, and Model Subclassing: Another Example

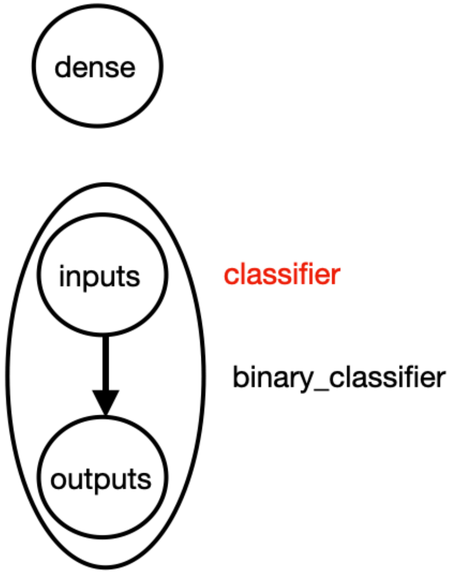
Inversely, you can use a Functional model as part of a subclassed layer or model.

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()
```

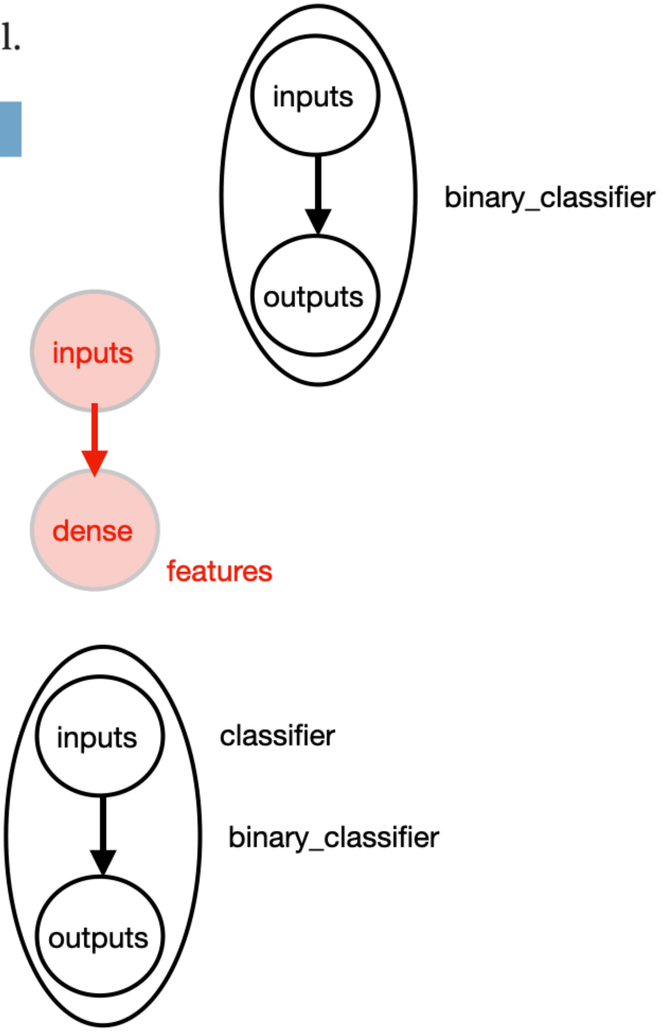


Mixing Sequential Model, Functional API, and Model Subclassing: Another Example

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()
```

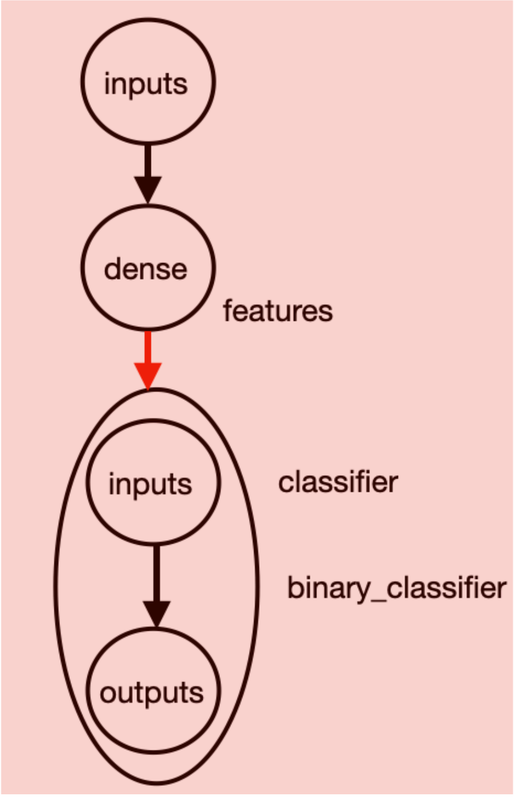


Mixing Sequential Model, Functional API, and Model Subclassing: Another Example

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()
```

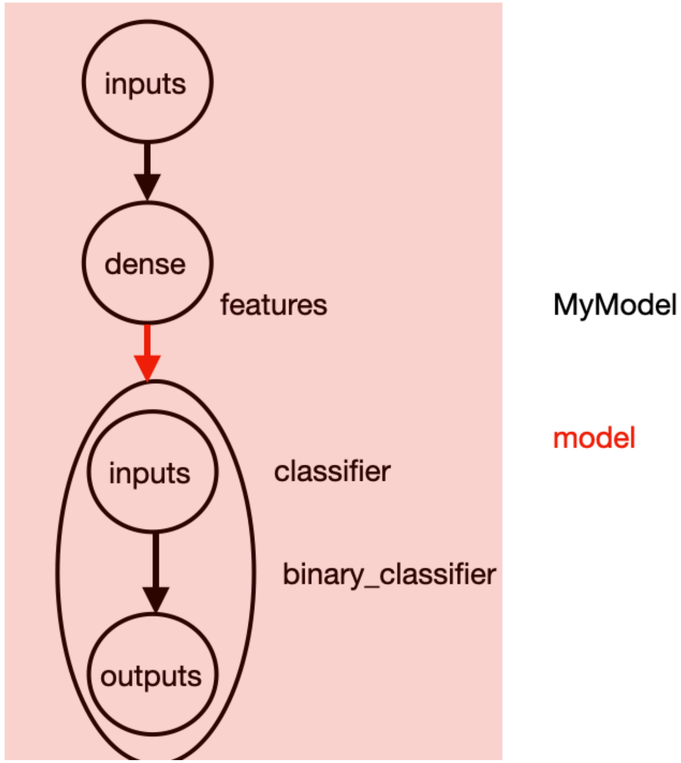


Mixing Sequential Model, Functional API, and Model Subclassing: Another Example

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()
```



Quiz questions:

1. How to use functional API and sub-classing together when building a model?

2. When is it useful to combine functional API with sub-classing?

Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

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)
```

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.

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

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

Using Built-in Training and Evaluation Loops

There are a couple of ways you can customize this simple workflow:

- Provide your own custom metrics.
- Pass *callbacks* to the `fit()` method to schedule actions to be taken at specific points during training.

Writing your own metrics

Commonly used metrics for classification and regression are already part of the keras.metrics module.

But your can also write your own metrics, which will become a subclass of the keras.metrics.Metric class.

A Keras metric is a subclass of the `keras.metrics.Metric` class. Like layers, a metric has an internal state stored in TensorFlow variables. Unlike layers, these variables aren't updated via backpropagation, so you have to write the state-update logic yourself, which happens in the `update_state()` method.

For example, here's a simple custom metric that measures the root mean squared error (RMSE).

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(L_i - y_i)^2}$$

Writing your own metrics: RMSE

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)
```

Subclass the Metric class.

Name of this metric is rmse

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.

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.

Writing your own metrics: RMSE

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)
```

Subclass the Metric class.

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.

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.

$$\text{mse\_sum} = \sum_{i=1}^{n} (L_i - y_i)^2$$

Writing your own metrics: RMSE

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)
```

Subclass the Metric class.

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.

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.

$$\text{mse\_sum} = \sum_{i=1}^{n} (L_i - y_i)^2$$

$$\text{total\_samples} = n$$

Writing your own metrics: RMSE

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)
```

Subclass the Metric class.

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.

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.

$$\text{mse\_sum} = \sum_{i=1}^{n} (L_i - y_i)^2$$

$$\text{total\_samples} = n$$

Writing your own metrics: RMSE

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)
```

Subclass the Metric class.

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.

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.

$$\text{mse\_sum} = \sum_{i=1}^{n} (L_i - y_i)^2$$

$$\text{total\_samples} = n$$

$$\text{mse} = \sum_{\text{batch}} (L_i - y_i)^2$$

L_i, y_i : one-hot encoded vectors

Writing your own metrics: RMSE

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)
```

Subclass the Metric class.

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.

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.

$$\text{mse\_sum} = \sum_{i=1}^{n} (L_i - y_i)^2$$

$$\text{total\_samples} = n$$

$$\text{mse} = \sum_{\text{batch}} (L_i - y_i)^2$$

L_i, y_i : one-hot encoded vectors

Writing your own metrics: RMSE

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)
```

Subclass the Metric class.

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.

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.

$$\text{mse\_sum} = \sum_{i=1}^{n} (L_i - y_i)^2$$

$\text{total\_samples} = n$

num_samples = batch size

$$\text{mse} = \sum_{\text{batch}} (L_i - y_i)^2$$

L_i, y_i : one-hot encoded vectors

Writing your own metrics: RMSE

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)
```

Subclass the Metric class.

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.

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.

$$\text{mse\_sum} = \sum_{i=1}^{n} (L_i - y_i)^2$$

$$\text{total\_samples} = n$$

$$\text{mse} = \sum_{\text{batch}} (L_i - y_i)^2$$

num_samples = batch size

L_i, y_i : one-hot encoded vectors

Writing your own metrics: RMSE

$$\text{mse\_sum} = \sum_{i=1}^{n} (L_i - y_i)^2 \qquad \text{total\_samples} = n$$

You use the `result()` method to return the current value of the metric:

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

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (L_i - y_i)^2}$$

Writing your own metrics: RMSE

$$\text{mse\_sum} = \sum_{i=1}^{n} (L_i - y_i)^2 \qquad \text{total\_samples} = n$$

Meanwhile, you also need to expose a way to reset the metric state without having to reinstantiate it—this enables the same metric objects to be used across different epochs of training or across both training and evaluation. You do this with the `reset_state()` method:

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

Writing your own metrics: RMSE

Custom metrics can be used just like built-in ones. Let's test-drive our own metric:

```
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)
```

Quiz questions:

1. How to define a new metric?

2. When is it useful to define a new metric for a model?

Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

Using Callbacks

Here are some examples of ways you can use callbacks:

- *Model checkpointing*—Saving the current state of the model at different points during training.
- *Early stopping*—Interrupting training when the validation loss is no longer improving (and of course, saving the best model obtained during training).
- *Dynamically adjusting the value of certain parameters during training*—Such as the learning rate of the optimizer.
- *Logging training and validation metrics during training, or visualizing the representations learned by the model as they're updated*—The `fit()` progress bar that you're familiar with is in fact a callback!

The `keras.callbacks` module includes a number of built-in callbacks (this is not an exhaustive list):

```
keras.callbacks.ModelCheckpoint
keras.callbacks.EarlyStopping
keras.callbacks.LearningRateScheduler
keras.callbacks.ReduceLROnPlateau
keras.callbacks.CSVLogger
```

Callback that streams epoch results to a CSV file.

Using Callbacks: EarlyStopping and ModelCheckpointing

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.

Interrupts training when improvement stops

Monitors the model's validation accuracy

Saves the current weights after every epoch

Interrupts training when accuracy has stopped improving for two epochs

Path to the destination model file

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().

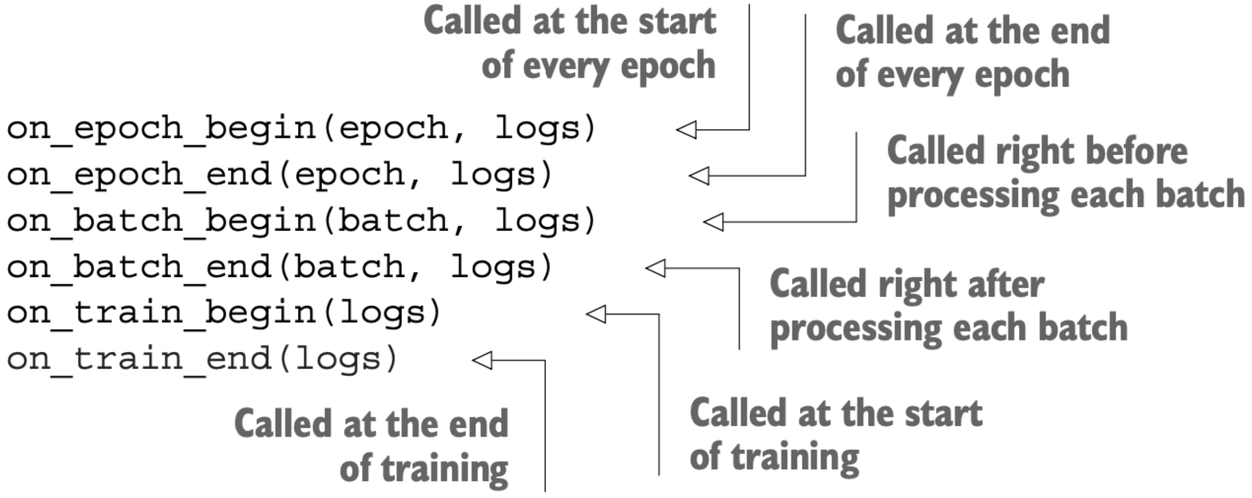
```
callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=2,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath="checkpoint_path.keras",
        monitor="val_loss",
        save_best_only=True,
    )
]
model = get_mnist_model()
model.compile(optimizer="rmsprop",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

Note that you can always save models manually after training as well—just call `model.save('my_checkpoint_path')`. To reload the model you've saved, just use

```
model = keras.models.load_model("checkpoint_path.keras")
```

Writing your own callbacks

If you need to take a specific action during training that isn't covered by one of the built-in callbacks, you can write your own callback. Callbacks are implemented by sub-classing the `keras.callbacks.Callback` class. You can then implement any number of the following transparently named methods, which are called at various points during training:



Writing your own callbacks

These methods are all called with a `logs` argument, which is a dictionary containing information about the previous batch, epoch, or training run—training and validation metrics, and so on. The `on_epoch_*` and `on_batch_*` methods also take the epoch or batch index as their first argument (an integer).

Here's a 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 = []

    def on_batch_end(self, batch, logs):
        self.per_batch_losses.append(logs.get("loss"))

    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}")
        self.per_batch_losses = []
```

Writing your own callbacks

Let's test-drive it:

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          callbacks=[LossHistory()],
          validation_data=(val_images, val_labels))
```

We get plots that look like figure 7.5.

Writing your own callbacks

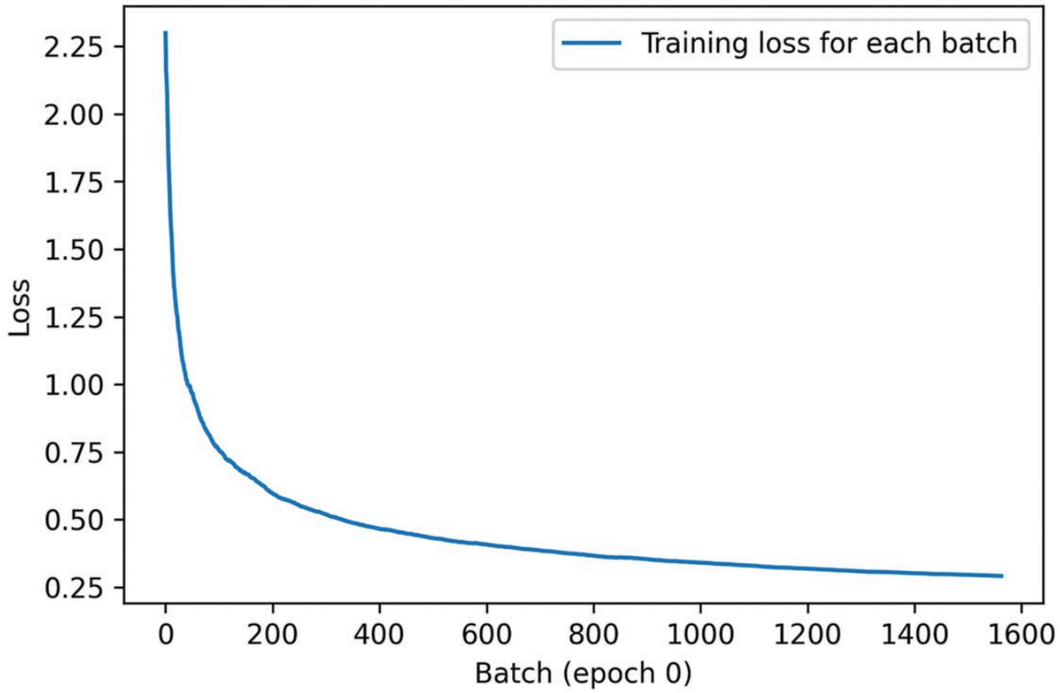
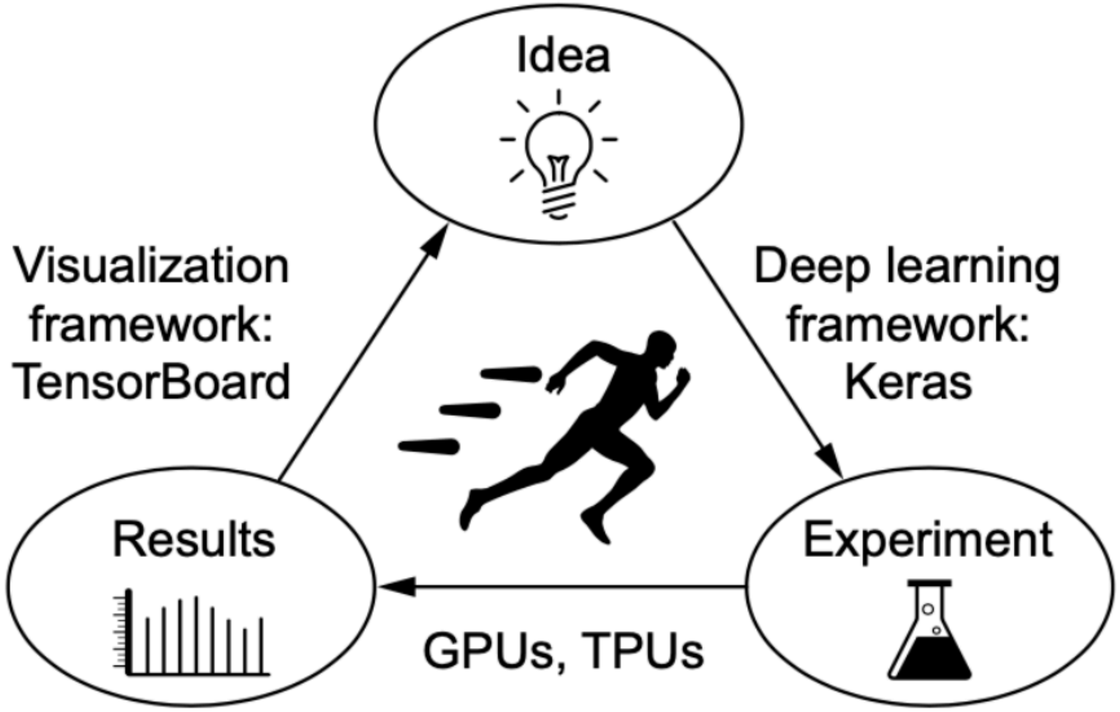


Figure 7.5 The output of our custom history plotting callback

Monitoring and visualization with TensorBoard



Monitoring and visualization with TensorBoard

With TensorBoard, you can

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

Monitoring and visualization with TensorBoard

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

In the simplest case, just specify where you want the callback to write logs, and you're good to go:

```
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])
```

Monitoring and visualization with TensorBoard

Once the model starts running, it will write logs at the target location. If you are running your Python script on a local machine, you can then launch the local TensorBoard server using the following command (note that the `tensorboard` executable should be already available if you have installed TensorFlow via `pip`; if not, you can install TensorBoard manually via `pip install tensorboard`):

```
tensorboard --logdir /full_path_to_your_log_dir
```

You can then navigate to the URL that the command returns in order to access the TensorBoard interface.

If you are running your script in a Colab notebook, you can run an embedded TensorBoard instance as part of your notebook, using the following commands:

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

In the TensorBoard interface, you will be able to monitor live graphs of your training and evaluation metrics (see figure 7.7).

Monitoring and visualization with TensorBoard

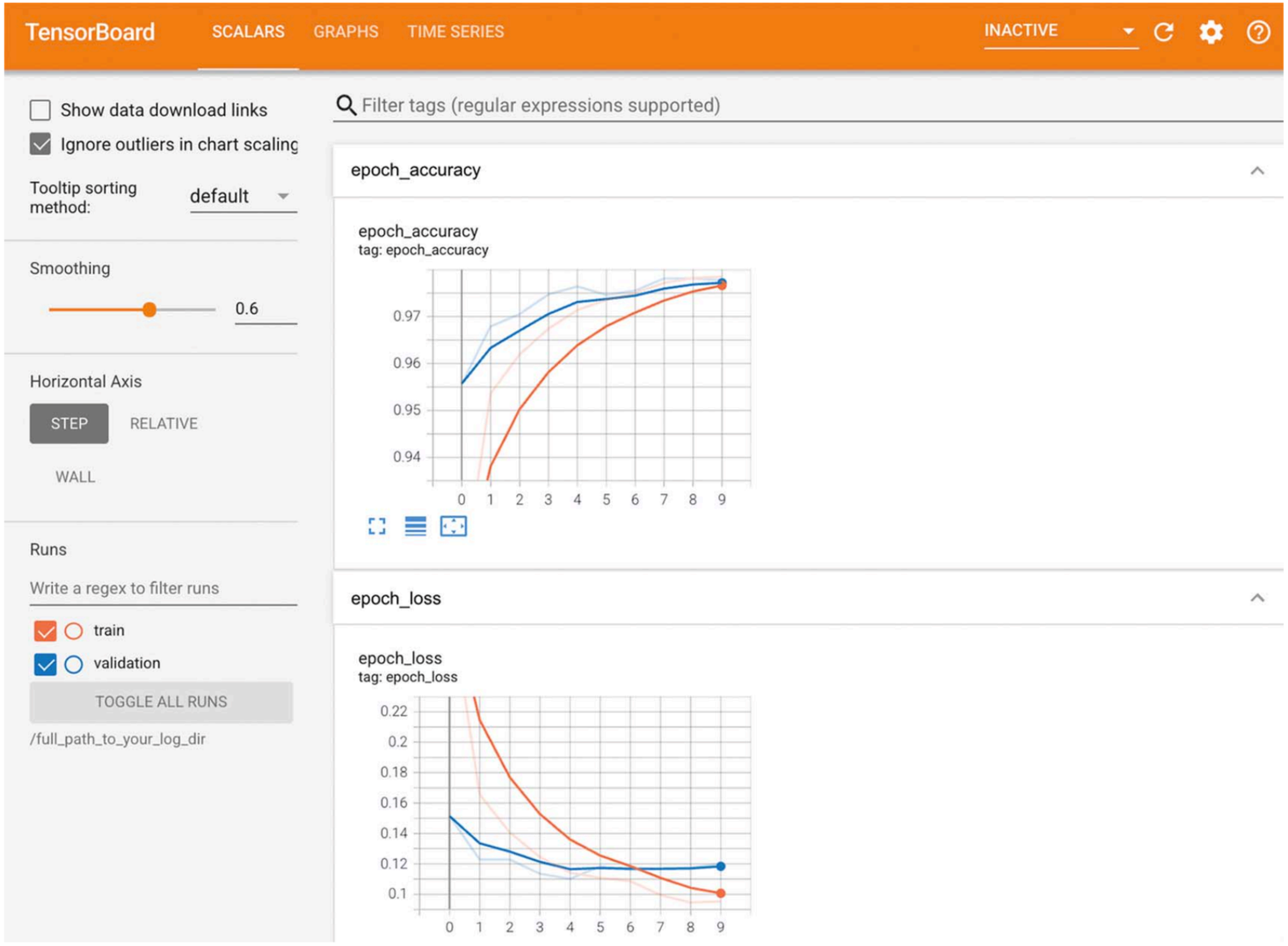


Figure 7.7 TensorBoard can be used for easy monitoring of training and evaluation metrics.

Quiz questions:

1. How to define a new Callback function?

2. How to use an existing Callback function?

Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

Writing your own training and evaluation loops

Why we want to learn it:

1) the built-in fit( ) workflow only focuses on supervised learning.

If we want to do generative learning, self-supervised learning, reinforcement learning, etc., we need to write our own training/ evaluation loops.

2) Even for supervised learning, we may want to customize details of the training process.

A complete training and evaluation loop

Let's combine the forward pass, backward pass, and metrics tracking into a `fit()`-like training step function that takes a batch of data and targets and returns the logs that would get displayed by the `fit()` progress bar.

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
```

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.

A complete training and evaluation loop

We will need to reset the state of our metrics at the start of each epoch and before running evaluation. Here's a utility function to do it.

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()
```

A complete training and evaluation loop

We can now lay out our complete training loop. Note that we use a `tf.data.Dataset` object to turn our NumPy data into an iterator that iterates over the data in batches of size 32.

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()
    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}")
```

A complete training and evaluation loop

And here's the evaluation loop: a simple `for` loop that repeatedly calls a `test_step()` function, which processes a single batch of data. The `test_step()` function is just a subset of the logic of `train_step()`. It omits the code that deals with updating the weights of the model—that is to say, everything involving the `GradientTape` and the optimizer.

Listing 7.24 Writing a step-by-step evaluation loop

```
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
```

Note that we pass training=False.

To continue on the next page …

A complete training and evaluation loop

Continued:

```
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}")
```

Make it fast with tf.function

You may have noticed that your custom loops are running significantly slower than the built-in `fit()` and `evaluate()`, despite implementing essentially the same logic. That's because, by default, TensorFlow code is executed line by line, *eagerly*, much like NumPy code or regular Python code. Eager execution makes it easier to debug your code, but it is far from optimal from a performance standpoint.

It's more performant to *compile* your TensorFlow code into a *computation graph* that can be globally optimized in a way that code interpreted line by line cannot. The syntax to do this is very simple: just add a `@tf.function` to any function you want to compile before executing, as shown in the following listing.

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

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

    logs = {}
    for metric in metrics:
        ...
```

This is the only line that changed.

Quiz questions:

1. How to write your own training or evaluation loop?

2. When is it useful to write your own training or evaluation loop?

Roadmap of this lecture:

1. Sequential Model
2. Functional API
    - 2.1 Single-input Single-output model
    - 2.2 Multiple-input multiple-output model
    - 2.3 Reuse layer or model
3. Subclassing
    - 3.1 Flexible model: for loop, if, recursive, etc.
    - 3.2 Use subclassing and functional API together
    - 3.3 Define a new metric
    - 3.4 Using (or define) Callbacks
    - 3.5 Write your own training and evaluation loop
    - 3.6 Leveraging fit( ) with a custom training loop

Leveraging fit( ) with a custom training loop

What if you need a custom training algorithm, but you still want to leverage the power of the built-in Keras training logic? 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.

You can do this by overriding the `train_step()` method of the `Model` class. This is the function that is called by `fit()` for every batch of data. You will then be able to call `fit()` as usual, and it will be running your own learning algorithm under the hood.

Leveraging fit( ) with a custom training loop

Here's a simple example:

- We create a new class that subclasses `keras.Model`.
- We override the method `train_step(self, data)`. Its contents are nearly identical to what we used in the previous section. It returns a dictionary mapping metric names (including the loss) to their current values.
- We implement a `metrics` property that tracks the model's `Metric` instances. This enables the model to automatically call `reset_state()` on the model's metrics at the start of each epoch and at the start of a call to `evaluate()`, so you don't have to do it by hand.

Leveraging fit( ) with a custom training loop

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

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

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()}

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

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

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.

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

Leveraging fit( ) with a custom training loop

We can now instantiate our custom model, compile it (we only pass the optimizer, since the loss is already defined outside of the model), and train it using `fit()` as usual:

```
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 = CustomModel(inputs, outputs)

model.compile(optimizer=keras.optimizers.RMSprop())
model.fit(train_images, train_labels, epochs=3)
```

Leveraging fit( ) with a custom training loop

An alternative solution: configure loss and metrics via compile( )

```
class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True)
            loss = self.compiled_loss(targets, predictions)
        gradients = tape.gradient(loss, model.trainable_weights)
        optimizer.apply_gradients(zip(gradients, model.trainable_weights))
        self.compiled_metrics.update_state(targets, predictions)
        return {m.name: m.result() for m in self.metrics}
```

Compute the loss via self.compiled_loss.

Update the model's metrics via self.compiled_metrics.

Return a dict mapping metric names to their current value.

No need to update loss manually.
It is updated automatically.

Leveraging fit( ) with a custom training loop

```
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 = CustomModel(inputs, outputs)

model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.SparseCategoricalCrossentropy(),
              metrics=[keras.metrics.SparseCategoricalAccuracy()])
model.fit(train_images, train_labels, epochs=3)
```

Quiz question:

1. How to leverage fit() with a custom training loop?