深度學習 - 第七章

Status	Book
Created time	@2024年12月30日 下午9:28

第七章

7-1 Keras的工作流程

Keras API的設計原則為「逐步提升複雜度」，為了滿足不同需求Keras提供了一整個工作流程的「光譜」幫助幫助建構和訓練模型。工作流程都基於共同的API(如：Layer類別與Model類別)，因此某些工作流程的元件，也可以用在任何其他的工作流程中。

7-2 建構Keras模型的不同方法

Keras中建構模型有3種API可供選擇

1. 序列式模型：最容易使用的API，基本為一個Python串列，因而可以單純地堆疊不同神經層
2. 函數式API：專注於有分支結構的模型，為建構模型時最常用的API
3. 繼承Model類別(Model subclassing)：需要自己從零開始的低階選項，如果想要控制所有細節就是會最好的方法。但無法使用太多Keras內建功能且最容易錯

序列式API + 內建神經層	函數式API+內建神經層	函數式API+自訂神經層+自訂評量指標...	使用Subclass：DIY使用者
新手或處裡簡單的模型時	處理一般任務的工程師	處理特殊任務的工程師或需要客製化的solution	研究人員、專業學者

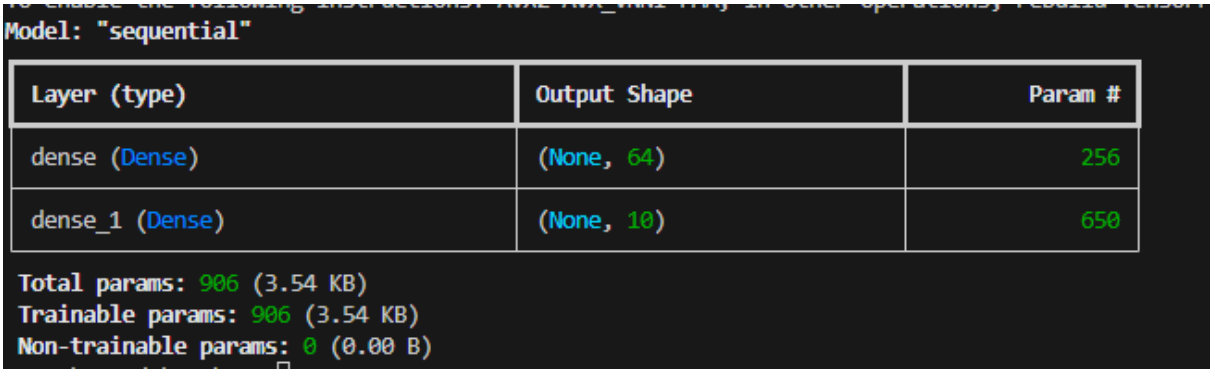
補充：序列式模型和序列式API是兩個不同的API，都可用來建立Keras模型，但前者只能建「序列式模型」，後者則可建立「任何結構的模型」。此外已建立好的模型也可

視為一個大型的神經層，再用「序列式API或函數式API」來將之與其他模型或神經層進行連接。Keras的模型及神經層可以任意地進行組合或拆解，稱為『積木式』建構模型。

7-2-1 序列式模型(Sequential Model)

建構Keras模型最簡單的方法就是使用Sequentia模型

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

model = keras.Sequential([ # 建立序列式模型(Sequential Model)
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax'),
])

model.build(input_shape=(None, 3))#指定模型輸入形狀

model.weights # 輸出出模型權重

model.summary() # print出模型內容
```



如上所見，Model被自動命名為 sequential ，而也可以用 下面的`name = 字串`的方式更改各項目的名稱

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

```
model = keras.Sequential(name = "my_model") # 建立一個序列模型

model.add(layers.Dense(64, activation='relu',name="輸入層")) # 新增一層Dense層，並稱之為"輸入層"
model.add(layers.Dense(10, activation='relu',name="輸出層")) # 新增一層Dense層，並稱之為"輸出層"

model.build((None, 3))#指定模型輸入形狀



model.weights # 輸出出模型權重

model.summary() # print出模型內容
```

```
Model: "my_model"
```

Layer (type)	Output Shape	Param #
輸入層 (Dense)	(None, 64)	256
輸出層 (Dense)	(None, 10)	650

```
Total params: 906 (3.54 KB)
Trainable params: 906 (3.54 KB)
Non-trainable params: 0 (0.00 B)
```

逐步序列式模型時，在每次新增神經層後使用summary()檢查模型是很有用的!! 且有一種加速建構序列式模型的方法，那就是提前宣告模型輸入的shape，這可以利用Input類別來辦到

```
model.add(keras.Input(shape=(3,))) # 宣告輸入的shape皆為3
```

7-2-2 函數式API(Functional API)

序列式模型的應用範圍有限：只能建構單一輸入、輸出的模型。現實中很常會遇到有多個輸入(如：影像及其中繼資料metadata)或多個輸出(如：同時預測商品的銷售數量及評價分類)的模型，甚至式中間有分支的分線性拓樸(topology)結構

簡單範例

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

inputs = keras.Input(shape=(3,), name='my_input')
#傳回一個input物件 ，內含該層的shape屬性和name屬性

features = layers.Dense(64, activation='relu')(inputs)
#創建一個Dense層，並用Inputs來呼叫

outputs = layers.Dense(10, activation="softmax")(features)
#創建一個Dense層取得最終的輸出張量，並用features來呼叫

model = keras.Model(inputs=inputs, outputs=outputs)
#創建一個模型，並用inputs和outputs為參數來呼叫Model來創建模型

model.summary()
```

inputs這樣的物件被稱為符號張量(symbolic tensor)，其中不包含任何真實資料，但記錄了模型運作時所處裡的真實資料張量的訊息。(在實際輸入資料時，其第0軸維度會隨批次量而自動調整大小)，且Keras的Input層和其他神經層不同，只是個用來建立代表模型輸入點的張量物件無實際神經層運算功能。

補充：函數式API會用到Python「**將物件當成函式來呼叫**」的技巧，在使用時要先建立神經層物件，接者把此物件當成函式來呼叫。如下所示：

```
h = Dense(8)(X)
```

以神經層的輸入張量(x)為參數，呼叫並執行函式後，會傳回該神經層的輸出張量並指定給一個參數(h)，接者h即可供下一層作為輸入張量

多輸入、多輸出的模型

多數深度學習模型長得並不像前述的線性序列模型，而是像一張有分支及合併的資料流程**圖**(graph)。

假如一個系統中，它具評估消費者意見單的優先性，並將之分配到合適的部門。模型會有3個輸入和2個輸出

輸入

1. 意見單的標題(文字輸入)
2. 意見單的內容(文字輸入)
3. 消費者勾選的標籤(分類輸入)

輸出

1. 意見單的優先性分數，介於0和1的純量值(以sigmoid輸出)
2. 應該處裡這單的部門(以softmax輸出對應到各部門的分數)

依上述內容可以得到以下程式

```
vocabular_size = 10000 #設定字典大小
num_tags = 100 #設定標籤數量
num_departments = 4 #設定部門數量
title = keras.Input(shape=(vocabular_size,), name='title') #定義標題輸入
text_body = keras.Input(shape=(vocabular_size,), name='text_body') #定義內容輸入
tags = keras.Input(shape=(num_tags,), name='tags') #定義標籤輸入

features = layers.concatenate([title, text_body, tags]) #將標題、正文、標籤特徵合併到一起
features = layers.Dense(64, activation='relu')(features) #加入全連接層來增加模型的表示能力

priority = layers.Dense(1, activation='sigmoid', name='priority')(features) #定義優先級輸出
department = layers.Dense(num_departments, name='department')(features) #定義部門輸出

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

訓練多輸入、多輸出模型

訓練**多輸入、多輸出模型的過程和訓練序列式模型式差不多的：只需以輸入資料和輸出資料的list來呼叫fit()。list中的元素順序需與之前model中的list保持一致**

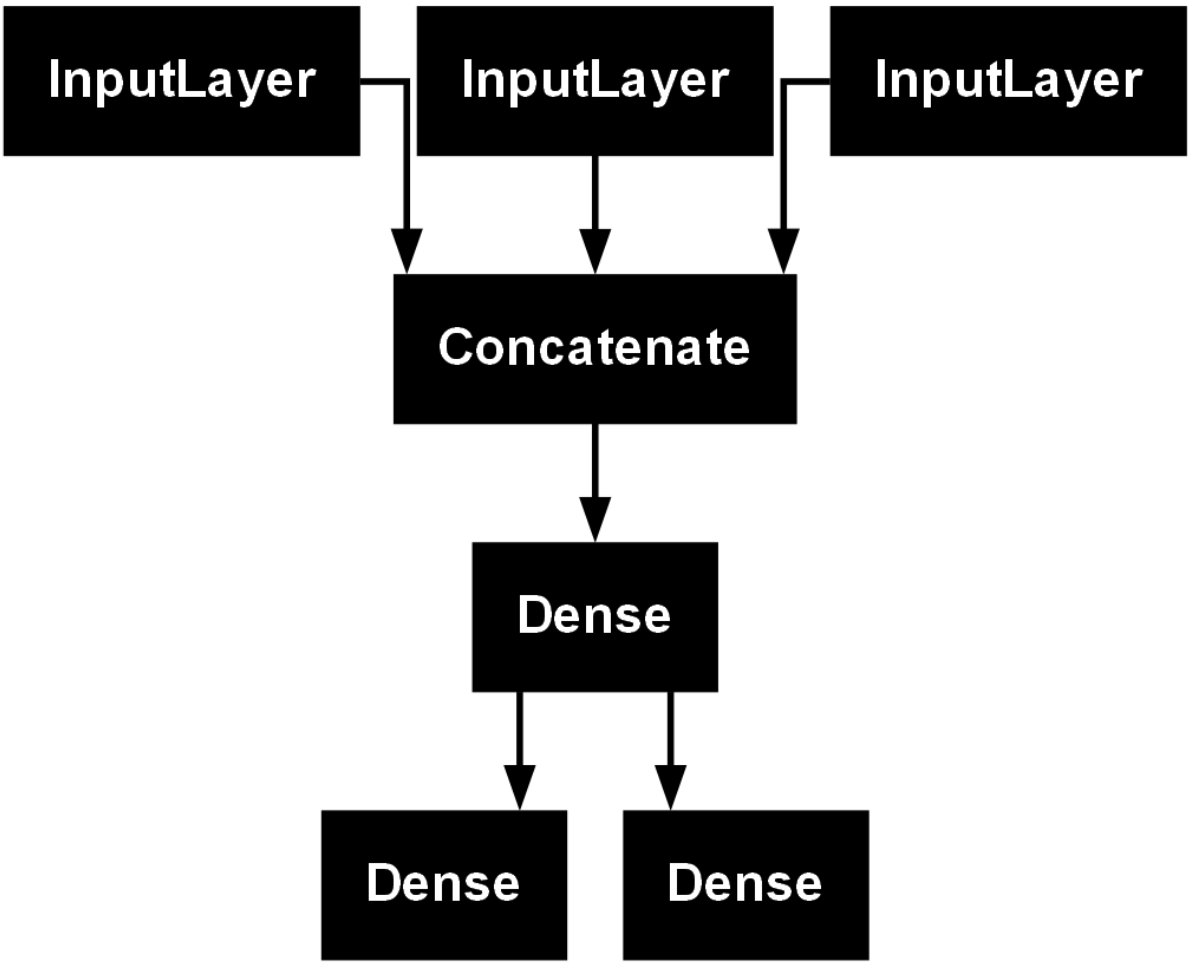
```
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np

vocabular_size = 10000 #設定字典大小
num_tags = 100 #設定標籤數量
num_departments = 4 #設定部門數量
title = keras.Input(shape=(vocabular_size,), name='title') #定義標題輸入
text_body = keras.Input(shape=(vocabular_size,), name='text_body') #定義正文輸入
tags = keras.Input(shape=(num_tags,), name='tags') #定義標籤輸入

features = layers.concatenate([title, text_body, tags]) #將標題、正文、標籤特徵合併到一起
features = layers.Dense(64, activation='relu')(features) #加入全連接層來增加模型的表示能力

priority = layers.Dense(1, activation='sigmoid', name='priority')(features) #定義優先級輸出
department = layers.Dense(num_departments, name='department')(features) #定義部門輸出

model = keras.Model(inputs=[title, text_body, tags], outputs=[priority, department]) #建立模型

num_samples = 1280

title_data = np.random.randint(0, 2, size=(num_samples, vocabular_size)) #產生標題數據
text_body_data = np.random.randint(0, 2, size=(num_samples, vocabular_size)) #產生內文數據
tags_data = np.random.randint(0, 2, size=(num_samples, num_tags)) #產生
```

```
標籤數據

priority_data = np.random.random(size=(num_samples, 1)) #產生優先級數據
department_data = np.random.randint(0, 2, size=(num_samples, num_depa
rtments)) #產生部門數據

model.compile(optimizer='rmsprop',
          loss={'priority': 'mean_squared_error', 'department': 'categorical_cr
ossentropy'},
          metrics={"priority":['mean_squared_error'], "department":["categori
cal_crossentropy"]}) #編譯模型

model.fit({'title': title_data, 'text_body': text_body_data, 'tags': tags_data},
        {"priority":priority_data,"department": department_data},
        epochs=1) #訓練模型

model.evaluate({'title': title_data, 'text_body': text_body_data, 'tags': tags_d
ata},
          {"priority":priority_data,"department":department_data}) #評估模型

priority_preds, department_preds = model.predict({'title': title_data, 'text_b
ody': text_body_data, 'tags': tags_data}) #預測模型
```

函數式API的威力：檢視神經層的連接方式

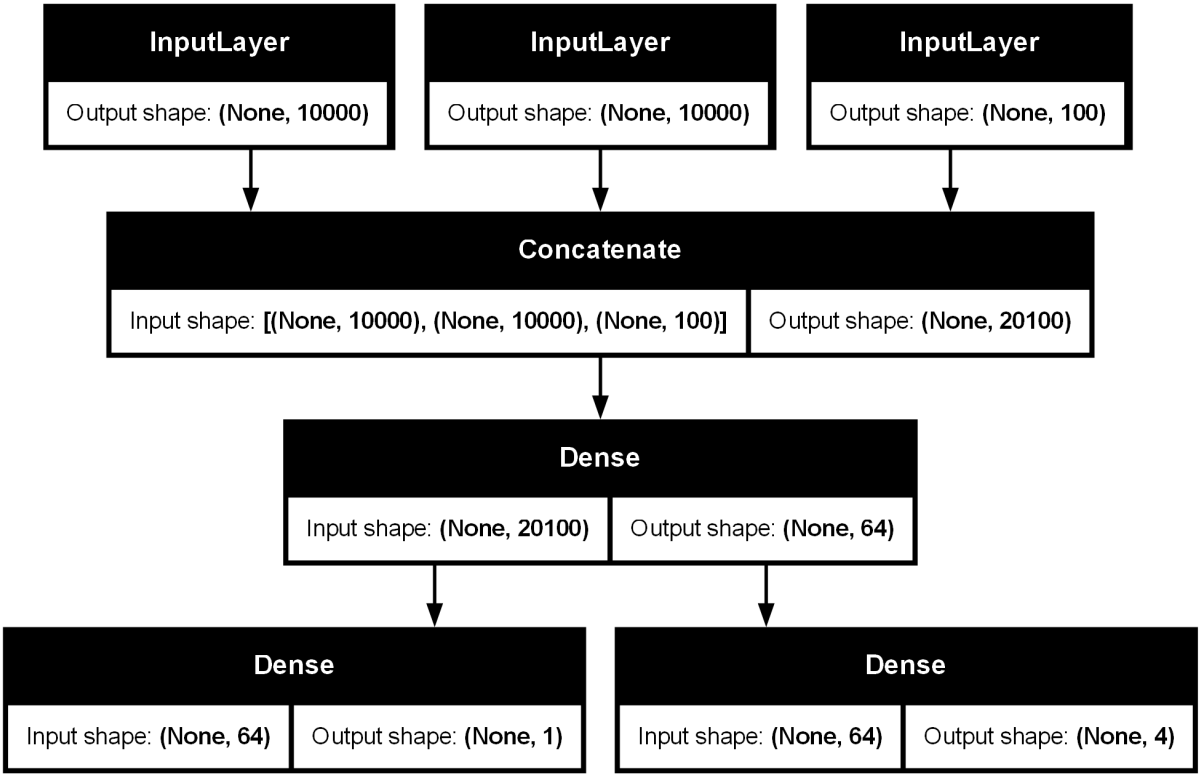
函數式模型就是一個明顯的資料流動圖(graph)，設計者可以檢視各個神經層之間如何連結並重複利用之前的**圖節點(graph nodes)**,神經層的輸出張量)來組成新模型。這促成了兩個非常重要的功能：模型視覺化(model cisualization)和特徵萃取(feature extraction)。只要使用**plot_model()**功能，就可將函數是模型畫成圖。

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



若要再加入各神經層的輸入和輸出shape，只要將show_shape參數設定為Ture即可。

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



張量shape中的「None」代表批次量：也就是模型可以接受任意大小的批次量。

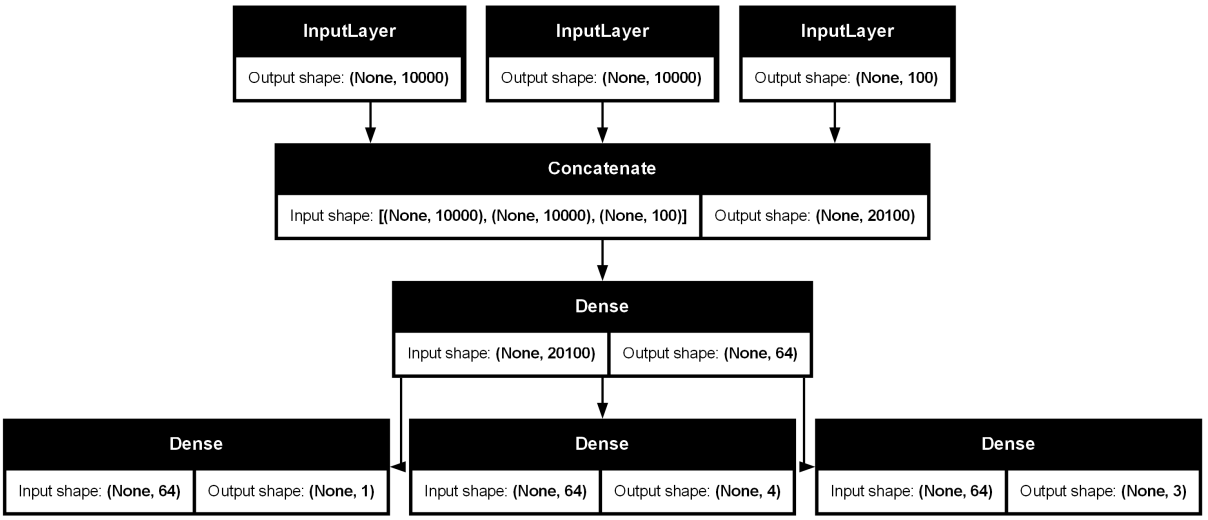
利用model.layer屬性可以取得包含所有神經層的串列，而其中的每一個神經層還可以利用layer.input 和layer.output屬性來查詢其輸入或輸入的張量物件。

```
model.layers #檢視神經層

model.layers[3].input #檢視第三層輸入

model.layer[2].output #檢視第二層輸出
```

藉由這特性可讓我們進行特徵萃取，也就是利用模型的中間特徵來建構出一個新模型。在上述模型中加入下列程式後

```
features = model.layers[4].output #取出模型的第三層

difficulty = (layers.Dense(3, activation='softmax', name='difficulty'))(features) #加入困難度輸出
```

```
new_model = keras.Model(inputs=[title,text_body,tags], outputs=[priority, department, difficulty]) #建立新模型

keras.utils.plot_model(new_model, to_file = 'multi_input_and_output_model.png', show_shapes=True) #繪製模型結構圖
```



可以看到多了一個新的中間層和新的輸出

7-2-3繼承Model類別(Subclassing the Model class)

繼承Model類別(Subclassing the Model class)是最近進階的方法，以下則為它的實作方法。

- 在__init__()中，定義模型會用到的神經層
- 在call()中，定義模型中各神經層(在__init__()中定義的神經層)的正向傳播流程。
- 實例化定義好的子類別，以輸入資料為參數呼叫模型(以模型物件當成函式呼叫)，以便創建模型的權重。

範例：由前一案例相同但不同做法

```
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
```

```
from tensorflow.keras.utils import plot_model

class CustomerTicketModel(keras.Model):
    def __init__(self, num_departments): #指定num_departments是部門的數量
        super().__init__() #初始化父類別
        self.concat_layer = layers.Concatenate()
        self.mixing_layer = layers.Dense(64, activation='relu')
        self.priority_scorer = layers.Dense(1, name='priority', activation='sigm
oid')
        self.department_classifier = layers.Dense(num_departments, name='d
epartment', activation='softmax')

    def call(self, inputs): #定義模型的前向傳播
        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

model = CustomerTicketModel(num_departments=4)
num_samples = 1280

title_data = np.random.randint(0, 2, size=(num_samples, vocabular_size)) #
產生標題數據
text_body_data = np.random.randint(0, 2, size=(num_samples, vocabular_si
ze)) #產生內文數據
tags_data = np.random.randint(0, 2, size=(num_samples, num_tags)) #產生
標籤數據

priority_data = np.random.random(size=(num_samples, 1)) #產生優先級數據
department_data = np.random.randint(0, 2, size=(num_samples, num_depa
rtments)) #產生部門數據
priority, department = model(
    {"title":title_data,"text_body":text_body_data,"tags":tags_data})
```

```
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},[[p
riority_data,department_data]],epochs=1)

model.evaluate({"title":title_data,"text_body":text_body_data,"tags":tags_da
ta},[[priority_data,department_data]])
priority_preds, department_preds = model({"title":title_data,"text_body":text
_body_data,"tags":tags_data})
```

在建構模型時，繼承Model 類別是最具彈性的做法。序列式及函數式API只能建立有向無環圖(有方向且不可有循環迴圈的資料流動圖)架構的模型，但繼承Model 類別沒有限制，。它可以在call()方法中用for迴圈來運作神經層，甚至可以遞迴的方式來呼叫。但缺點是需要負責更多模型中邏輯的部分，也就是說更容易出錯了。

函數式模型與Model 子類別在本質上也差異很大。

函數式模型：

- 明確且可查看
- 由不同層組成

Model 子類別：

- 一堆位元組碼
- 一個call()包含原始的Python類別

同時由於神經層之間的連接方式藏在call()方法中，因此無法用summary()顯示出，也不能用plot_model()繪製出拓樸結構。不能取得神經層的圖節點(不能進行特徵萃取)因為根本就沒有圖

7-2-4 混合搭配不同設計模式

無論哪一種設計模式都不會，都不會限制只能在同一模式運作

如：

含有Model子類別的函數式模型

```
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np


class Classifier(keras.Model):
    def __init__(self, num_classes):
        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) # 將Model子類別當作一層la
yer使用
model = keras.Model(inputs=inputs, outputs=outputs)
```

包含函數式模型的Model子類別

```
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np


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

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

class MyModel(keras.Model):
    def __init__(self, num_classes=2):
        super().__init__()
        self.dense = layers.Dense(1, activation="relu")
        self.classifier = binary_classifier

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

model = MyModel()
```

7-3使用內建的訓練與評估迴圈

「逐步提升複雜度」的設計原則一樣可以套用在訓練過程上。其中最簡單的方式就是呼叫fit()。而以下則是還可以調整的面向

- 設定自訂義的評量指標
- 將回呼(callback)物件傳遞個fit()方法，以排定在訓練階段的特定時間所要採取的動作(如：當驗證準確度連續2個週期都沒有進步時就停止訓練，以避免過度配適)

7-3-1設計自己的評量指標

評量指是用來評量模型表現的關鍵，特別是用來衡量模型在訓練資料及評估資料上的表現差異。分類及迴歸中常用的評量指標，在內建的Keras.metrics模組中都有。但還是有一些新的想法需要自行設計評量指標。

Keras的所有評量指標都是Keras.metrics.Metric類別的子類別。和神經層一樣有些評量指標也被用來儲存內部狀態的TensorFlow變數。但和神經層不同的是這些反向傳播的過程並不會自動更新。因此要將更新狀態的程式寫在update_state()中。

以下是一個測量方均根誤差(root mean squared error , RMSE)的自訂義評量指標

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

class RootMeanSquaredError(keras.metrics.Metric): # Metric子類別

    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=tf.int32)

    def update_state(self, y_true, y_pred, sample_weight=None): # 更新狀態
        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_true)[0]
        self.total_samples.assign_add(num_samples)

    def result(self): # 計算最終結果
        return tf.sqrt(self.mse_sum / tf.cast(self.total_samples, tf.float32))

    def reset_states(self): # 重設狀態
        self.mse_sum.assign(0.0)
        self.total_samples.assign(0)

def get_mnist_model():
    model = keras.Sequential([
        layers.Flatten(input_shape=(28, 28)),
        layers.Dense(128, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    return model

#MNIST 資料集
(train_images, train_labels), (test_images, test_labels) = keras.datasets.mni
st.load_data()
```

```
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0

# 將測試集的一部分作為驗證集
val_images = test_images[:5000]
val_labels = test_labels[:5000]
test_images = test_images[5000:]
test_labels = test_labels[5000:]


model = get_mnist_model()
model.compile(optimizer="rmsprop", loss="sparse_categorical_crossentropy", metrics=["accuracy", RootMeanSquaredError()]) #同時使用內建和自訂指標

model.fit(train_images, train_labels, epochs=3, validation_data=(val_images, val_labels))

test_metrics = model.evaluate(test_images, test_labels)

print("測試指標:", test_metrics)
```

```
Epoch 1/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8852 - loss: 0.4130 - rmse: 0.4156 - val_accuracy: 0.9452 - val_loss: 0.1879 - val_rmse: 0.2918
Epoch 2/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9613 - loss: 0.1309 - rmse: 0.2419 - val_accuracy: 0.9578 - val_loss: 0.1412 - val_rmse: 0.2546
Epoch 3/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9735 - loss: 0.0891 - rmse: 0.2014 - val_accuracy: 0.9644 - val_loss: 0.1281 - val_rmse: 0.2384
157/157 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9837 - loss: 0.0559 - rmse: 0.1548
測試指標: [0.0662517100572586, 0.9801999926567078, 0.1736181527376175]
```

7-3-2 使用回呼(callbacks)模組

在大型資料集上進行數十個週期的訓練一旦開始就無法停止。若想避免不良後果就要使用回呼來將資訊回傳給操作者，然後在根據當前狀態自動地做決策使其修正。

回呼(callbacks)是在呼叫fit()方法時可以傳遞給模型的一個物件(一個執行特定方法的物件)，並且可以在訓練階段的各個時間點由模型所呼叫。它可以檢視有關模型狀態及表現的所有資料，進而執行各類操作，包括中斷訓練、儲存模型、載入不同權重集合或更改模型狀態。

以下是使用回呼的範例：

- 早期停止(Early Stopping)：當驗證損失不再改善時就中斷訓練(會自動保存訓練中獲得的最佳模型)
- 模型檢查點(Model checkpointing)：在訓練階段的不同時間點保存模型的當前權重
- 在訓練期間動態調整某些參數的值，如：優化器的學習率
- 在訓練期間紀錄訓練和驗證指標值,或在模型更新時視覺化模型學習到的表示法(權重)，如：fit()的進度條

Early Stopping和Model checkpointing回呼

為了不浪費資源，當證損失不再改善時中斷訓練就可以使用Early Stopping回呼，而為了保存當下的最佳模型會與Model checkpointing回呼搭配使用

```
from tensorflow import keras
from tensorflow.keras import layers
from keras.datasets import mnist
import tensorflow as tf

# 定義 MNIST 模型
def get_mnist_model():
    model = keras.Sequential([
        layers.Flatten(input_shape=(28, 28)),
        layers.Dense(128, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    return model

# 載入 MNIST 資料集
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0

# 將部分測試集分割為驗證集
```

```
val_images = test_images[:5000]
val_labels = test_labels[:5000]
test_images = test_images[5000:]
test_labels = test_labels[5000:]

# 定義回呼函數
callback_list = [
    keras.callbacks.EarlyStopping(  # EarlyStopping
        monitor="val_accuracy",  # 修正為正確的指標名稱
        patience=2,  # 如果在兩輪中沒有改善，就中斷訓練
    ),
    keras.callbacks.ModelCheckpoint(  # ModelCheckpoint
        filepath="my_model.keras",  # 使用 .keras 格式
        monitor="val_loss",  # 監控驗證集損失
        save_best_only=True,  # 只儲存最佳模型
    ),
]

# 建立與編譯模型
model = get_mnist_model()
model.compile(optimizer="rmsprop", loss="sparse_categorical_crossentro
py", metrics=["accuracy"])

# 訓練模型
model.fit(
    train_images, train_labels,
    epochs=10,
    callbacks=callback_list,
    validation_data=(val_images, val_labels)
)

# 評估模型
test_metrics = model.evaluate(test_images, test_labels)
print("測試指標:", test_metrics)
```

7-3-3 設計自己的回呼

如果內建的回呼不夠用，也可以設計自己的回呼。透過繼承Keras.callbacks.Callback類別，就可以開始設計所需的method

▼ 如下所視：

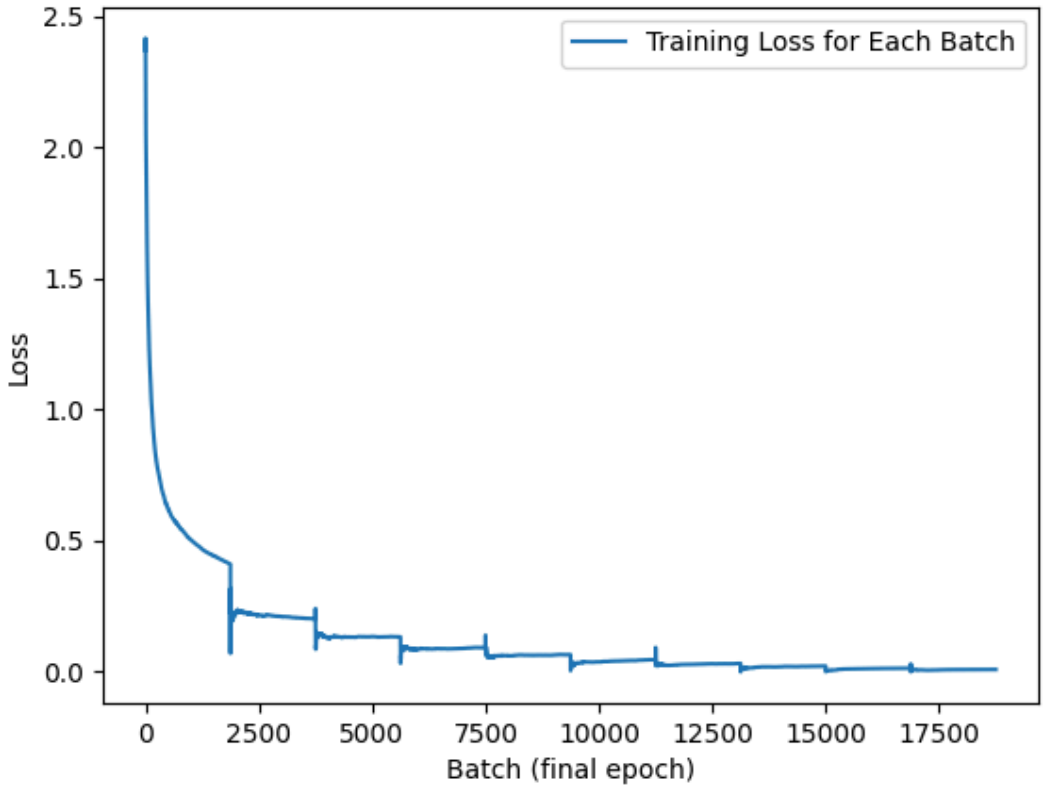
- on_epoch_begin(epoch, logs) 每個訓練週期的開始呼叫
- on_epoch_end(epoch, logs) 每個訓練週期的結束呼叫
- on_batch_begin(batch , logs) 處理每個批次資料前呼叫
- on_train_begin(logs) 在訓練開始時呼叫

這些method都會傳遞**logs參數**，它是個Python字典，包括上一批次、訓練週期、訓練與驗證指標等資訊

```
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow as tf
from matplotlib import pyplot as plt
import numpy as np

# 自訂回呼函數：紀錄每批次的損失並繪圖（只在最後一次 epoch 結束時繪圖）
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs=None):
        self.per_batch_losses = []  # 初始化列表，用於存儲每批次的損失

    def on_batch_end(self, batch, logs=None):
        # 在每批次結束時，將損失添加到列表中
        self.per_batch_losses.append(logs.get("loss"))

    def on_epoch_end(self, epoch, logs=None):
        pass  # 不再在每個 epoch 結束時繪圖

    def on_train_end(self, logs=None):
        # 在訓練結束後，繪製最後一次的訓練損失圖
        plt.clf()  # 清除當前圖形
        plt.plot(range(len(self.per_batch_losses)), self.per_batch_losses, label
```

```
="Training Loss for Each Batch")
        plt.xlabel("Batch (final epoch)")  # 設置 x 軸標籤
        plt.ylabel("Loss")  # 設置 y 軸標籤
        plt.legend()  # 顯示圖例
        plt.savefig("batch_loss_final_epoch.png")  # 保存圖形為 .png 文件

        self.per_batch_losses = []  # 清空列表

# 定義 MNIST 模型
def get_mnist_model():
    model = keras.Sequential([
        layers.Flatten(input_shape=(28, 28)),
        layers.Dense(128, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    return model



# 載入 MNIST 資料集
(train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.load_data()

# 資料標準化（0~1）
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0

# 添加隨機噪聲到訓練資料
noise_factor = 0.3
train_images_noisy = train_images + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=train_images.shape)
train_images_noisy = np.clip(train_images_noisy, 0.0, 1.0)  # 確保像素值仍在 0~1 範圍內

# 調整測試資料亮度
test_images_bright = np.clip(test_images + 0.5, 0.0, 1.0)  # 增加亮度，但仍在 0~1 範圍內
```

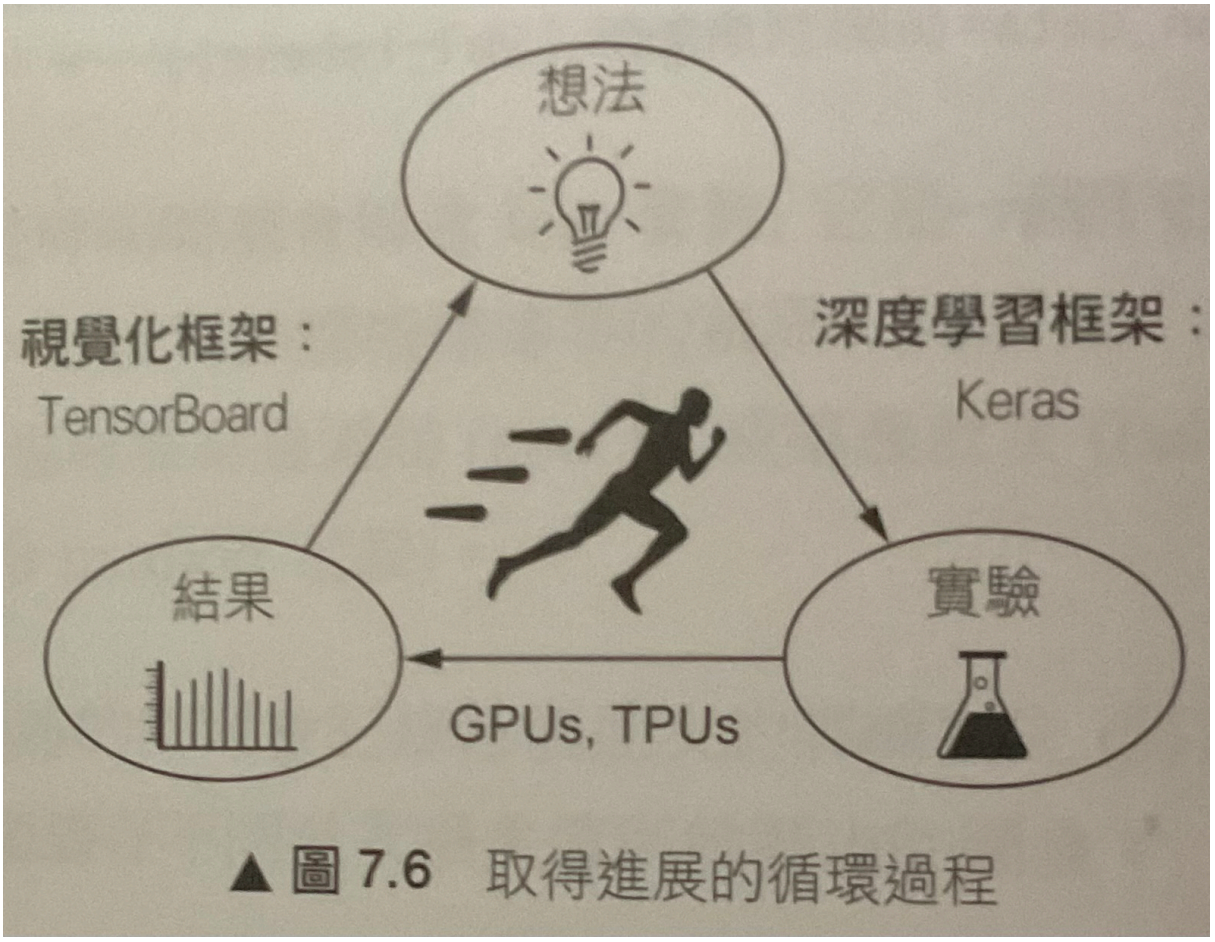
```
# 劃分驗證集
val_images = test_images_bright[:5000]
val_labels = test_labels[:5000]
test_images_bright = test_images_bright[5000:]
test_labels = test_labels[5000:]

# 初始化模型
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

# 訓練模型並應用回呼函數
history_callback = LossHistory()
model.fit(
    train_images_noisy, train_labels,  # 使用加了噪聲的訓練資料
    epochs=10,
    callbacks=[history_callback],  # 設置回呼
    validation_data=(val_images, val_labels)  # 使用增加亮度的驗證資料
)

# 評估模型
test_metrics = model.evaluate(test_images_bright, test_labels)  # 使用增加亮度的測試資料
print("測試指標:", test_metrics)
```




```
測試指標: [3.415092945098877, 0.6294000148773193]
```

7-3-4 利用TensorBoard進行監控與視覺化

若要對研究有好的掌握或開發出成效佳的模型，就需要對實驗中對模型內部發生的事情頻繁地取得回饋資訊。取得進程是一個迭代過程或是一個循環過程如下：



▲圖 7.6　取得進展的循環過程

從想法開始，轉換成實驗散後評估原先的想法是否成立，執行這樣的實驗並掌握過程中的資訊，再激發下一個想法。迭代越多想法就越精準。而結果則可以交給**TensorBoard**。

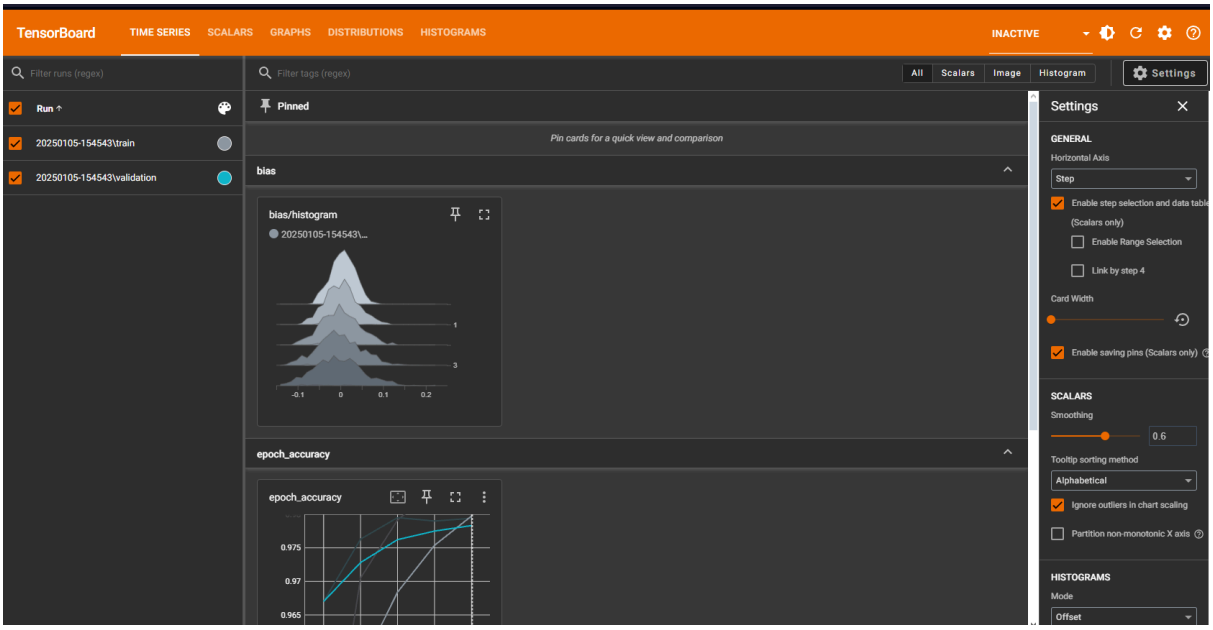
TensorBoard是一個以瀏覽器為基礎的視覺化工具，可以在本地端執行。它是訓練進行時用來監控模型內部變化的最佳方法。以下是TensorBoard可以做到的事。

- 在訓練期間以視覺化方式監控損失及評量指標
- 將模型架構視覺化
- 將激活結果和梯度變化以視覺化直方圖呈現
- 以3D方式探索嵌入向量(embeddings)

若監控的不僅僅是模型的最終損失，就可以更清清晰地了解模型內部做了與沒做哪些事，並可更快地取得進展。

> 使用TensorBoard最簡單的方法就是在fit()中使用Keras.callbacks.TensorBoard回呼。

```
import tensorflow as tf
import datetime

mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

def create_model():
  return tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
  ])

model = create_model()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# 設定log的資料夾
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%
H%M%S")
# 設定tensorboard的callback
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
histogram_freq=1)

model.fit(x=x_train,
          y=y_train,
          epochs=5,
          validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])
```



7-4 設計自己的訓練及評估迴圈

Keras的fit()可以在易用性與彈性間取得良好的平衡。但內建的fit()只適用於監督式學習(supervised learning)，但是並不是所有機器學習任務都屬於這個領域。有些機器學習任務並不具備明確的目標值如：生成學習、自監督學習和強化式學習。而這時就需要些低階的設計彈性了。

7-4-1 訓練 vs. 推論

一些Keras層(如：Dropout層)在訓練階段盒推論階段(僅會用來產生預測值)有著不同的行為模式，而這種神經層會在call()中用一個名為training的布林參數。舉例來說，呼叫dropout(inputs, **training = True**)會隨機拋棄一定比例參數(訓練階段的行為模式)；而呼叫dropout(inputs ,**training=False**)時則什麼都不會做(推論階段的行為模式)。此外函數型與序列式模型也會在call()中使用training參數。

正向傳播時呼叫Keras模型時要將 training參數設定為True。這樣正向傳播過程指令會變成 predicitons = model(inputs, training = True)。此外取得模型權重時的梯度時，應該使用tape.gradients(loss, model.trainable_weights),而不是tape.gradients(loss, model.weights)這是因為模型和神經層中存在兩種權重：

- 可訓練權重(Trainable weights)：這類權重在反向傳播時要進行更新以降低損失值，例如Dense層的kernel和偏值(bias)

- 非訓練權重(Non-Trainable weights)：這類權重會在正向傳播時更新。如計數器這類資訊會被儲存在非訓練權重中

Keras內建的神經層中，只有BatchNormalization層有非訓練權重。BatchNormalization層需要非訓練權重來追蹤資料的平均層(mean)與標準差(standard deviation)，以便執行即時的特徵正規化

7-4-2 評價指標的低階用法

評量指標的API：只要對每批次的目標值與預測值呼叫update_state(y_true, y_pred)，然後呼叫result()查詢當前的指標值：

```
from tensorflow import keras
import numpy as np

metric = keras.metrics.SparseCategoricalAccuracy()
targets = [0,1,2]

predictions = [[1,0,0],[0,1,0],[0,0,1]]
metric.update_state(targets, predictions)
current_result = metric.result()
print(f'current_result:{current_result:.2f}' )

輸出結果-----
current_result:1.00
```

若想追蹤某個純量(如：訓練損失)的平均值，這時可以使用Keras.metrics.Mean()物件：

```
from tensorflow import keras
import numpy as np

values = [0, 1, 2, 3, 4]
mean_tracker = keras.metrics.Mean()
for value in values:
  mean_tracker.update_state(value)
```

```
print(f"Mean of values: {mean_tracker.result():.2f}")

輸出結果-----
Mean of values: 2.00
```

若想重製指標值(在每個訓練周期或評估階段的一開始)時，可以使用metrics.reset_state()。這樣就可以不用重新實例化metrics物件了

7-4-3 完整的訓練及評估迴圈

```
from tensorflow import keras
from tensorflow.keras import layers
from keras.datasets import mnist
import tensorflow as tf

# 定義 MNIST 模型
def get_mnist_model():
    model = keras.Sequential([
        layers.Flatten(input_shape=(28, 28)),
        layers.Dense(128, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    return model

def train_step(inputs, targets):  # 定義訓練步驟
    with tf.GradientTape() as tape:
        predictions = model(inputs, training=True)  # 正向傳播 training 必須設定為 True
        loss = loss_fn(targets, predictions)
    gradients = tape.gradient(loss, model.trainable_weights)
    optimizer.apply_gradients(zip(gradients, model.trainable_weights))

    # 計算並更新指標
    logs = {}
    for metric in metrics:
```

```
        metric.update_state(targets, predictions)
        logs[metric.name] = metric.result()

    # 更新損失指標
    loss_tracking_metric.update_state(loss)
    logs["loss"] = loss_tracking_metric.result()
    return logs

def reset_metrics():  # 重置所有指標
    for metric in [loss_tracking_metric] + metrics:
        metric.reset_state()  # 使用 reset_state

def test_step(inputs, targets):  # 定義測試步驟
    predictions = model(inputs, training=False)  # 正向傳播 training 必須設定為 False
    loss = loss_fn(targets, predictions)

    logs = {}
    for metric in metrics:  # 計算並更新指標
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()

    logs["val_loss"] = loss
    return logs

# 載入 MNIST 資料集
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0

# 將部分測試集分割為驗證集
val_images = test_images[:5000]
val_labels = test_labels[:5000]
test_images = test_images[5000:]
test_labels = test_labels[5000:]

# 建立模型
model = get_mnist_model()
```

```
# 定義損失函數與優化器
loss_fn = keras.losses.SparseCategoricalCrossentropy()
optimizer = keras.optimizers.RMSprop()

# 定義指標
metrics = [keras.metrics.SparseCategoricalAccuracy()]
loss_tracking_metric = keras.metrics.Mean(name="loss")

# 訓練資料集
training_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_la
bels))
training_dataset = training_dataset.batch(32)

# 訓練週期
epochs = 3

for epoch in range(epochs):
    reset_metrics()
    print(f"\n開始訓練 Epoch {epoch+1}/{epochs}")
    for step, (inputs_batch, targets_batch) in enumerate(training_dataset):
        logs = train_step(inputs_batch, targets_batch)

        # 每 900 步顯示一次訓練結果
        if step % 900 == 0:
            print(f"步驟 {step}, 損失: {logs['loss']:.4f}, 準確率: {logs['sparse_cate
gorical_accuracy']:.4f}")

    print(f"Epoch {epoch+1} 訓練結束，訓練損失: {logs['loss']:.4f}, 訓練準確
率: {logs['sparse_categorical_accuracy']:.4f}")

    # 驗證
    reset_metrics()
    val_dataset = tf.data.Dataset.from_tensor_slices((val_images, val_labels))
    #使用tf.data.Dataset物件，以此在Numpy放入迭代器
    val_dataset = val_dataset.batch(32)

    for inputs_batch, targets_batch in val_dataset:
```

```
        val_logs = test_step(inputs_batch, targets_batch)

    print(f"Epoch {epoch+1} 驗證結果 - 損失: {val_logs['val_loss']:.4f}, 準確率: {val_logs['val_sparse_categorical_accuracy']:.4f}")

# 最終測試
reset_metrics()
test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
test_dataset = test_dataset.batch(32)

for inputs_batch, targets_batch in test_dataset:
    test_logs = test_step(inputs_batch, targets_batch)

print("\n測試結果:")
for key, value in test_logs.items():
    print(f"...{key}: {value:.4f}")


----輸出 只放最後一次--

開始訓練 Epoch 3/3
步驟 0, 損失: 0.0785, 準確率: 0.9688
步驟 900, 損失: 0.0898, 準確率: 0.9742
步驟 1800, 損失: 0.0869, 準確率: 0.9752
Epoch 3 訓練結束，訓練損失: 0.0857, 訓練準確率: 0.9756
Epoch 3 驗證結果 - 損失: 0.0242, 準確率: 0.9554

測試結果:
...val_sparse_categorical_accuracy: 0.9796
...val_loss: 0.0059
```

fit()和evaluate()還支援很多功能，包括大規模分散式運算的功能，但要實現該通能需要花費更多心力。

7-4-4 利用tf.function來加速

自行設定的迴圈比內建的fit()或evaluate()還要慢上很多，雖然邏輯相同，這是因為tensorflow的程式碼是**逐行執行**的，雖然方便除錯但是卻拖累了效率。

如果講TensorFlow程式碼編譯成**運算圖**(computation graph)，就能進行全域的效率優化。實現此功能的番是很簡單：只要在編譯的函式前加上**「@tf.function」裝飾器**就可以了

如：

```
@tf.function
def test_step(inputs, targets):  # 定義測試步驟
    predictions = model(inputs, training=False)  # 正向傳播 training 必須設定為 False
    loss = loss_fn(targets, predictions)

    logs = {}
    for metric in metrics:  # 計算並更新指標
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()

    logs["val_loss"] = loss
    return logs
```

這樣速度就快了很多，只是當除錯時最好還是不要加上@tf.function裝飾器，這樣比較好找到問題。若程式可以正常執行，而想提升速度則可以為訓練函式、評估函式或對其他對運行效率有嚴格要求的函式加上@tf.function裝飾器。

7-4-5 搭配fit()和自訂義的訓練迴圈

若想自己設計訓練演算法，也想用內建函式的力量。有一個方法：僅提供訓練步驟的函式(training step function)，然後讓框架處裡剩餘的事情。

要做的只有改寫Model類別的**train_step()方法**，它會由fit()在每一批次的資料上呼叫。改些完成後就可以如往常般使用fit()

簡單的案例

- 先創建一個繼承Keras.Model的子類別
- 修改train_step(self, data)：會傳回一個Python字典，key為評量指標名稱(包括損失)，value為對應的指標值
- 加入一個**metrics屬性**來追蹤模型的Metric物件。這樣模型就可以在每個訓練週期或呼叫evaluate()的一開始，自動呼叫reset_state()

```
from tensorflow import keras
from tensorflow.keras import layers
from keras.datasets import mnist
import tensorflow as tf

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

# 載入並預處理 MNIST 資料
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((-1, 28 * 28)).astype("float32") / 255
# 將影像展平並正規化
test_images = test_images.reshape((-1, 28 * 28)).astype("float32") / 255

class CustomModel(keras.Model):
  def train_step(self, data): #改寫Model子類別的train_step方法
    inputs, targets = data
    with tf.GradientTape() as tape:
      predictions = self(inputs, training=True) #這裡用self()來取代model()，因為目前model不存在，而self即代表CustomModel
      loss = self.compiled_loss(targets, predictions)
      #透過self.compiled_loss()來取得loss，這是因為在compile時，我們已經設定了loss函數

    gradients = tape.gradient(loss, model.trainable_weights)
    self.optimizer.apply_gradients(zip(gradients, model.trainable_weights))
    self.compiled_metrics.update_state(targets, predictions)
    #這是更新metrics的方法，用包裝器的方式來更新metrics
```

```
    return {m.name: m.result() for m in self.metrics} #傳回python字典，包含所有metrics的名稱與結果



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)

print(model.metrics_names)

---輸出結果---
['loss', 'compile_metrics']
```