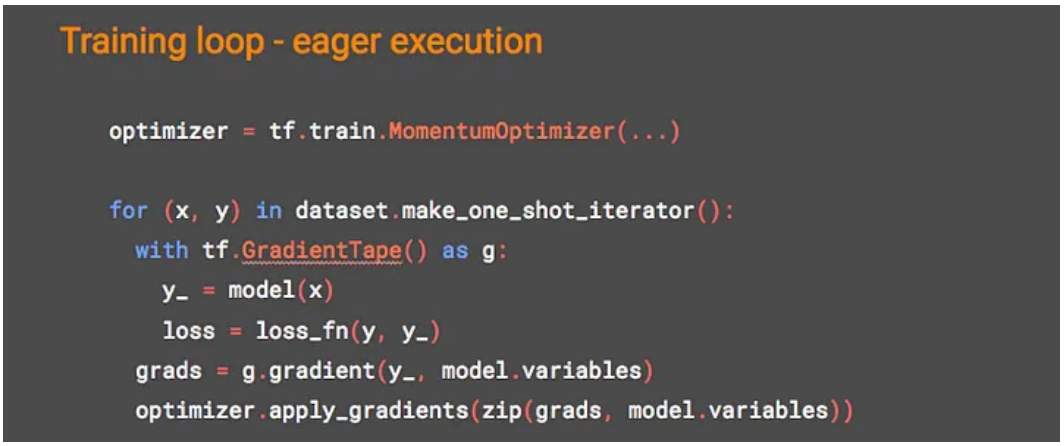
深度學習 - 第三章

Status	Book
Created time	@2024年12月3日 下午9:03

3-1TensorFlow是什麼?

TensorFlow是一個免費、開源的機械學習框架。由Google開發和NumPy一樣，TensorFlow的主要目的是讓工程師和研究人員可以在數值張量進行數學運算。以下是它的特色:

- **TensoeFlow能在任何可微分函數(第二章所示)上自動計算梯度。**
- **TensorFlow可在CPU、GPU、TPU等高度平行的硬體加速器上運行。**
- **TensorFlow的eager execution能像Python與TensorFlow互動。能流暢地寫程式、除錯**

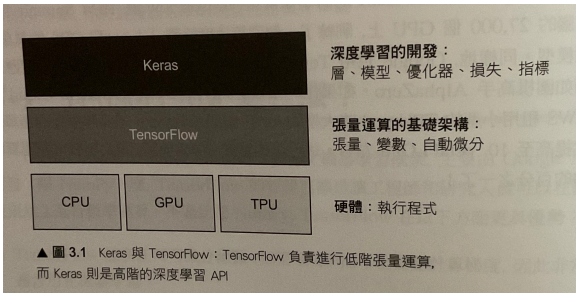


- **支援的語言多元**

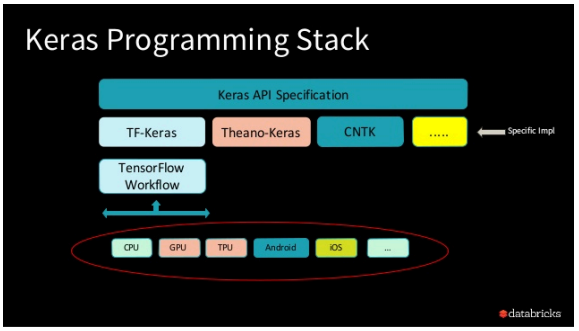
- **使用者可在瀏覽器內訓練並執行模型**
- **TensorFlow Hub可以幫助前期開發**
- **Keras + TensorFlow = 更容易建構神經網路**

3-2Keras是什麼?

Keras是Python的深度學習API，建立在TensorFlow之上，提供了簡易的方法來定義及訓練深度學習模型。架構如下:



▲ 圖 3.1 Keras 與 TensorFlow：TensorFlow 負責進行低階張量運算，而 Keras 則是高階的深度學習 API



Keras的優點是早已將訓練模型的輸入層、隱藏層、輸出層建好架構，只需插入所需的參數或函式即可，因此可使用最少的程式碼和時間完成深度學習模型的建構，後開始進行訓練、修正誤差並拿去做應用和預測。同時可以依據需求提供各式地工作流程，使用者可視情況依據易用性和彈性選擇使用。運作上則僅處理深度學習模型的建立、訓練、預測等，然而底層的運算，如張量（矩陣）運算，則是交給TensorFlow做配合。

3-4設定深度學習工作站

要在CPU上進行深度學習有三個選擇

- **買張NVDIA GPU獨顯，裝在電腦上 (AMD我不知道行不行、Intel的就...Intel)**
- **Google Cloud或AWS EC2上使用(須根據訂閱方案付費)**
- **用Google Colaboratory(Colab)上的免費GPU**

3-4-1,3-4-2 皆為colab軟體教學 故不做介紹

3-3跳過

3-5使用TensorFlow的第一步

訓練神經網路涉及以下觀念：

- **低階的張量運算會貫穿整個機械學習過程，這部分會轉換成TensorFlow API**
  - **存放網路狀態(變數)的特殊張量(tensors)**
  - **張量操作(tensors operaion)：如:加減法、線性整流函數(relu)、矩陣乘法(matmul)**
  - **反向傳播(backpropagation)：一種計算數學運算式之梯度的方式(TensorFlow中以GradientTape物件來處理)**

- **高階的概念則轉換成Keras API**
  - **層(layer)：組合成一起就是模型(model)**
  - **損失函數(loss function)：定義學習階段所用的回饋信號**
  - **優化器(opimizer)：決定學習過程如何推進**
  - **用來評估模型表現的各種指標(metric)如：準確度(accuracy)**
  - 執行小批次隨機梯度下降(mini-batch stochastic gradient descent)的訓練迴圈

3-5-1 常數張量與變數

不管要利用TensorFlow來做什麼事都離不開張量。在建立張量時需要為其指定一個初始值

```
import tensorflow as  tf
import random

#創建張量

x = tf.ones(shape=(2,1)) #1的張量 = Numpy中的np.ones(shape=(2,1))

y = tf.zeros(shape=(2,1)) #0的張量 = Numpy中的np.zeros(shape=(2,1))

z = tf.random.normal(shape=(3,1), mean=0., stddev=1.)  #亂數張量從平均值0和標準差1中生成 相當於Numpy中的np.random.normal(loc=0., scale=1., size=(3,1))
w = tf.random.uniform(shape=(3,1), minval=0., maxval=1) #均勻分布的亂數張量，從0到1之間生成 相當於Numpy中的np.random.uniform(low=0., high=1., size=(3,1))

print(x)
print(y)
print(z)
print(w)
```

NumPy陣列與TensorFlow張量的關鍵差異在於，TensorFlow張量的值為常數(constant)無法指派新的值

```
import numpy as np
x = np.ones(shape(2,2))
x[0,0] = 0
print(x)
#TensorFlow無法執行類似的操作 會回報EagerTensorobject does not support item assignment
```

訓練模型時需要不斷更新其狀態(一組張量)。這時需要利用 tf.Variable類別了，它是TensorFlow中負責操作可變狀態的類別。

```
import tensorflow as  tf
import random
```

```
v = tf.Variable(initial_value=tf.random.normal(shape=(3,1)))
#用亂數張量創建一個Variable物件
print(v)


——→ 輸出結果
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[ 0.62077934],
       [-1.1574291 ],
       [-0.72336376]], dtype=float32)>
```

可以利用assign()這個方法(method)來修改Variable物件的狀態

```
import tensorflow as  tf
import random

v = tf.Variable(initial_value=tf.random.normal(shape=(3,1)))
#用亂數張量創建一個Variable物件

v.assign(tf.ones(shape=(3,1)))
#將Variable物件的值設為1
print(v)

——→ 輸出結果 全部值被設為1
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

```
import tensorflow as  tf
import random

v = tf.Variable(initial_value=tf.random.normal(shape=(3,1)))
#用亂數張量創建一個Variable物件
```

```
v[0,0].assign(3)
#也可以局部指定Variable物件的元素
print(v)

-——> 輸出結果
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[3.        ], ← [0,0] 的位置被設定為3
       [1.6305209 ],
       [0.04337573]], dtype=float32)>
```

Variable物件中assign_add()和assign_sub() 則分別代表 「+=」 和「-=」運算

```
import tensorflow as  tf
import random

v = tf.Variable(initial_value=tf.random.normal(shape=(3,1)))
#用亂數張量創建一個Variable物件

v.assign(tf.ones((3,1)))
#將Variable物件的元素設為1

v.assign_add(tf.ones((3,1)))
#Variable物件的元素加1
print(v)

-——> 輸出結果
array([[2.],
       [2.],
       [2.]], dtype=float32)>
```

```
import tensorflow as  tf
import random

v = tf.Variable(initial_value=tf.random.normal(shape=(3,1)))
#用亂數張量創建一個Variable物件
```

```
v.assign(tf.ones((3,1)))
#將Variable物件的元素設為1

v.assign_sub(tf.ones((3,1)))
#Variable物件的元素減1
print(v)

-——> 輸出結果
array([[0.],
       [0.],
       [0.]], dtype=float32)>
```

3-5-2 張量操作：在TensorFlow中進行數學運算

```
a = tf.ones((2,2)) #設成2*2的矩陣 值皆為1
b = tf.square(a)  #平方
c = tf.sqrt(a) #平方根
d = b+c #張量相加(逐元素相加)
e = tf.matmul(a,b) #張量點積 如2-3-3中討論
e *= d  #張量相乘(逐元素相乘)
```

每一項操作都是立即執行，可以立即將最新結果印出來，這種方法稱為 即時執行(eagerexecution)

3-5-3 GradientTape API的進一步說明

相較NumPy TensorFlow能夠取得任意輸入項的梯度，只要在TensorFlow設置一個GradientTape區塊並在區塊內輸入張量(一個或多個)進行計算，就可以取得計算結果對各個輸入張量的梯度。

```
import tensorflow as  tf

input_var = tf.constant(3.0)  # <————----這裡要用浮點數不然結果是None 課本程式3-10也要用浮點數才能跑
#建立一個tf.Variable物件，並將其初始值為3

with tf.GradientTape() as tape:
```

```
    tape.watch(input_var)  # ←——--這裡要用用tape.watch不然輸出會變None
    #監控input_var的變化

    result = tf.square(input_var)
    #輸出為輸入的平方

gradient = tape.gradient(result, input_var)
#計算output_var對input_var的梯度

print(gradient)


--——→ 輸出結果
tf.Tensor(6.0, shape=(), dtype=float32)
```

此方法最常用來計算模型損失值(loss)對權重(weights)的梯度: gradients = tape.gradient(loss, weights) 。

tape.gradient()可以輸入任何類型的張量，但系統預設只會追蹤GradientTTape區塊中的可訓練變數(trainable variable) 如：Variable物件。至於常數張量，就必須用tape.watch()指定後才會進行追蹤

為什麼要用tape.watch()? 這是因為計算所有張量的資料和計算量太大了，為避免浪費磁帶必須知道哪些才是需要計算跟追蹤的張量。系統預設會自動追蹤可訓練變數的原因是，計算「損失值對可訓練變數的梯度」就是梯度磁帶最主要的用途。

梯度磁帶可以計算二階梯度(second-order gradiens) 也就是梯度的平方。

```
import tensorflow as  tf

time = tf.Variable(0.0)
# 將時間設定為0的Variable
# 這樣就可以使用GradientTape追蹤時間的變化
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
```

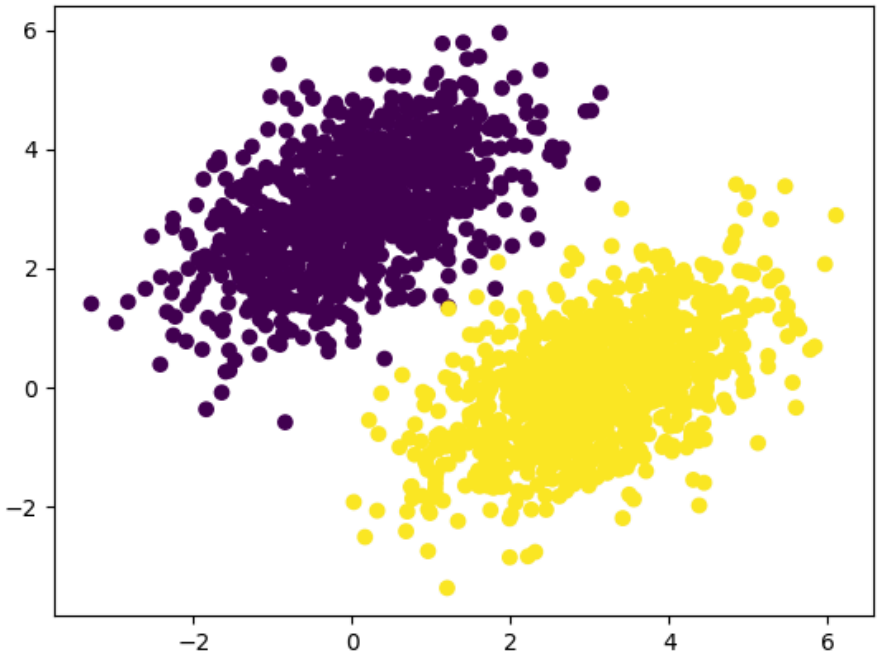
```
        position = 4.9 * time**2
        # 計算位置
    speed = inner_tape.gradient(position, time)
    # 計算速度

acceleration = outer_tape.gradient(speed, time)
# 計算加速度

print(acceleration)

-——→ 輸出結果
tf.Tensor(9.8, shape=(), dtype=float32)
```

3-5-4 端到端的範例:使用TensorFlow建立線性分類器

首先要在平面創建資料點，這些資料點分屬不同的類別。透過特定共變異數矩陣及平均數的隨機分佈來抽出資料點的座標位置。共變異數矩陣描述了點雲的形狀，平均值會在點雲的中心位置。通常會用同一個共變異數、不同平均值的兩個點雲做訓練

```
import numpy as np
import tensorflow as  tf
import matplotlib.pyplot as plt #pip install matplotlib

num_samples_per_class = 1000    #設定每個類別有1000筆資料

negative_samples = np.random.multivariate_normal( #第一個類別的資料
    mean=[0, 3], #設定平均值
    cov=[[1, .5],[.5, 1]],  #設定共變異數 圖形會呈現橢圓形 從左上到右下
    size=num_samples_per_class
    )


positive_samples = np.random.multivariate_normal(  #anthor one
mean=[3, 0],  #不同的平均值
cov=[[1, .5],[.5, 1]],
```

```
size=num_samples_per_class)

#negative_samples 和 positive_samples 各自是一個1000x2的矩陣，每一行代表一個樣本，每一列代表一個特徵。

inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
#將negative_samples和positive_samples進行堆疊，得到一個2000x2的矩陣，即為輸入
#np.vstack()是將兩個矩陣垂直堆疊在一起，np.hstack()是將兩個矩陣水平堆疊在一起。
#前1000筆資料是negative_samples ，後1000筆資料是positive_samples

targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype='float32'), np.ones((num_samples_per_class, 1), dtype='float32')))
#前1000筆資料的標籤設為0，後1000筆資料的標籤設為1。

plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])

#繪製散點圖，x軸為inputs的第一列，y軸為inputs的第二列，c參數為顏色，targets的第一列為0時，顏色為深色，為1時，顏色為淺色。

plt.show()
```



接下來要讓分類器學會如何分類資料點，線性分類器是以「最小化預測值與目標值誤差的平方」為目標來進行訓練的仿射變數(affine transformation) ，數學上來說即 prediciton = W * input + b 其中W、b均為變數

```
input_dim = 2 #設定輸入維度 代表樣本的維度
output_dim = 1 #設定輸出維度 代表個別樣本的預測分數(若某樣本類別為0，則分數應該接近於0)

W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
# 初始值隨機分佈在0~1之間的變數W

b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
# 初始值0的變數B
```

由於分類器處裡的是2D輸入(第0軸是資料點數量、第1軸是個別資料點的座標值)，因此W是由兩個純量(w1,w2)所組成：

W =[[w1],[w2]]；而b則是一個純量。對於任一輸入點[x,y]預測值就會是、 [[w1],[w2]] * [x,y] + b = w1*x + w2 * y + b 。

再來是以透過之前的點雲並訓練資料來更新權重W與b，讓損失值最小化，為簡單起見使用批次訓練，一次性對所有資料進行訓練盡可能讓過程中納入所有訓練樣本的資訊，而非部分樣本的資訊。

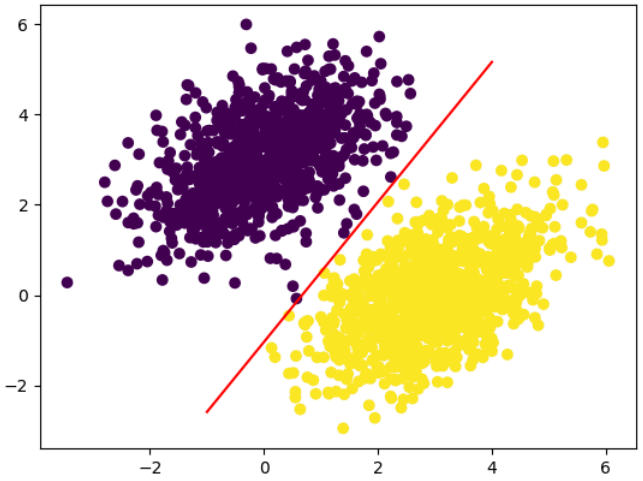
```
import numpy as np
import tensorflow as  tf
import matplotlib.pyplot as plt #pip install matplotlib

def model (input): #定義模型
    return tf.matmul(input, w) + b

def square_loss(targets, predictions): #定義損失函數
    per_sample_loss = tf.square(targets - predictions)
    # per_sample_loss張量的shape與target和predictions張量相同，內存有每個樣本的損失
    return tf.reduce_mean(per_sample_loss)
    # 返回的損失值是所有樣本的損失的平均值 進而輸出單一損失值

def training_step(inputs, targets): #定義訓練迴圈

    #在磁區磁帶內進行計算，以便在後續計算梯度
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(targets, predictions)
    grad_loss_w, grad_loss_b = tape.gradient(loss, [w, b]) #計算損失函數對於w和b的梯度

    #更新權重
    w.assign_sub(grad_loss_w * learning_rate)
    b.assign_sub(grad_loss_b * learning_rate)

    return loss


learning_rate = 0.1

input_dim = 2 #設定輸入維度 代表樣本的維度
```

```
output_dim = 1 #設定輸出維度 代表個別樣本的預測分數(若某樣本類別為0，則分數應該接近於0)

w = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
# 初始值隨機分佈在0~1之間的變數W

b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
# 初始值0的變數B

num_samples_per_class = 1000

negative_samples = np.random.multivariate_normal(
    mean=[0, 3],
    cov=[[1, .5],[.5, 1]],
    size=num_samples_per_class
    )


positive_samples = np.random.multivariate_normal(
    mean=[3, 0],
    cov=[[1, .5],[.5, 1]],
    size=num_samples_per_class
    )



inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)

targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype='float32'), np.ones((num_samples_per_class, 1), dtype='float32')))

# 訓練迴圈
for step in range(40):
    loss = training_step(inputs, targets)
    print('Loss at step %d: %.4f' % (step, loss))
```

```
predictions = model(inputs)

x = np.linspace(-1, 4, 100) #直線的x軸
y = -w[0] / w[1] * x + (0.5 - b) / w[1] #直線方程式
plt.plot(x, y, '-r') #"-r表示直線為紅色"
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)

plt.show()
```

損失值逐漸在0.028附近穩定，之後透過圖表檢查分類器表現 圖中區隔兩類別的便是 w1*x + w2*y + b = 0.5 的紅線。上方的屬於類別0，下方則是類別1

```
Loss at step 25: 0.0391
Loss at step 26: 0.0378
Loss at step 27: 0.0367
Loss at step 28: 0.0356
Loss at step 29: 0.0347
Loss at step 30: 0.0338
Loss at step 31: 0.0330
Loss at step 32: 0.0323
Loss at step 33: 0.0316
Loss at step 34: 0.0310
Loss at step 35: 0.0305
Loss at step 36: 0.0300
Loss at step 37: 0.0295
Loss at step 38: 0.0291
Loss at step 39: 0.0287
```



以上就是訓練線性流程器的流程：可以透過找到一條直線的權重參數，使其可以整齊區隔不同類別的資料

3-6剖析神經網路：了解Keras API的核心

3-6-1 Layer(層)：深度學習的基石

不同格式的資料需要用不同的層(Layer)處理

- **1D向量資料 ，儲存在2D張量 shape為(樣本 samples, 特徵 features) 密集連接層(densely connected layer)/全連接層(fully connected layer)/密集層**

(dense layer)處理

- **2D序列資料，儲存在3D張量 shape為(樣本 samples , 時戳timesteps , 特徵 features) 循環層(recurrent layer LSTM層)處理**
- **3D影像資料，儲存在4D張量 2D卷積層(Conv2D layer)處理**

Layer類別是Keras的核心，每個Keras元件都是一個Layer物件或與Layer密切互動。Layer是將依些狀態(權重)和運算(正向傳播)包在一起的物件。權重可以在建構子__int__()中建立， 一般會使用build()來建立，而正向傳播的運算過程則是用call()定義。

```
import tensorflow as  tf
from tensorflow import keras

class SimpleDense(keras.layers.Layer): # 所有的層都會繼承 keras.layers.Lay
er

    def __init__(self, units, activation=None):
        super(SimpleDense, self).__init__()
        self.units = units
        self.activation = tf.keras.activations.get(activation)

    def build(self, input_shape): #在build方法中，我們可以訪問輸入的形狀，並
根據輸入的形狀來創建權重
        input_dim = input_shape[-1]
        self.w = self.add_weight(shape=(input_dim, self.units), initializer='rand
om_normal',) # add_weight方法用於創建權重 類似於tf.Variable
        self.b = self.add_weight(shape=(self.units,),initializer='zeros')

    def call(self, inputs): #在call()方法中，定義層的正向傳播邏輯
        y = tf.matmul(inputs, self.w) + self.b
        if self.activation is not None:
            y = self.activation(y)
        return y

my_dense = SimpleDense(units=32, activation='relu') # 實例化layer

input = tf.ones(shape=(2, 784)) #創建張量
```

```
output = my_dense(input) #呼叫layer python會自動調用__call__()方法 根據情況呼叫 build() 和 call() 方法
print(output.shape)

——→ 輸出結果
(2, 32)
```

Layer和樂高積木一樣，只有具備相容性的Layer才能扣在一起。所謂相容性及每個layer只能接受特定shape的輸入張量

```
from tensorflow.keras import layers

layer = layers.Dense(32 , activation="relu") #有32個輸出單元的dense層
```

以上的layer會傳回一個張量且第1軸已經被固定成32了(shape為 (批次量,32))，因此layer必須連結到預設輸入 shape(批次量,32)的下游layer。

不過Keras不用擔心相容性的問題，因為模型中的每一層權重張量keras都會自動配合上一層的張量的shape

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

model = models.Sequential([
    layers.Dense(32 , activation="relu"),
    layers.Dense(32)
])
```

以上程式中layer並沒有指定任何輸入的shape，但卻會根據第一次輸入的shape來推論各層的shape

keras簡化了不少流程上的步驟，詳情可以參考這裡或是第二章的部分，目前只須注意：實作自己的Layer物件時，請把正向傳播的部分另外寫在call()方法中。

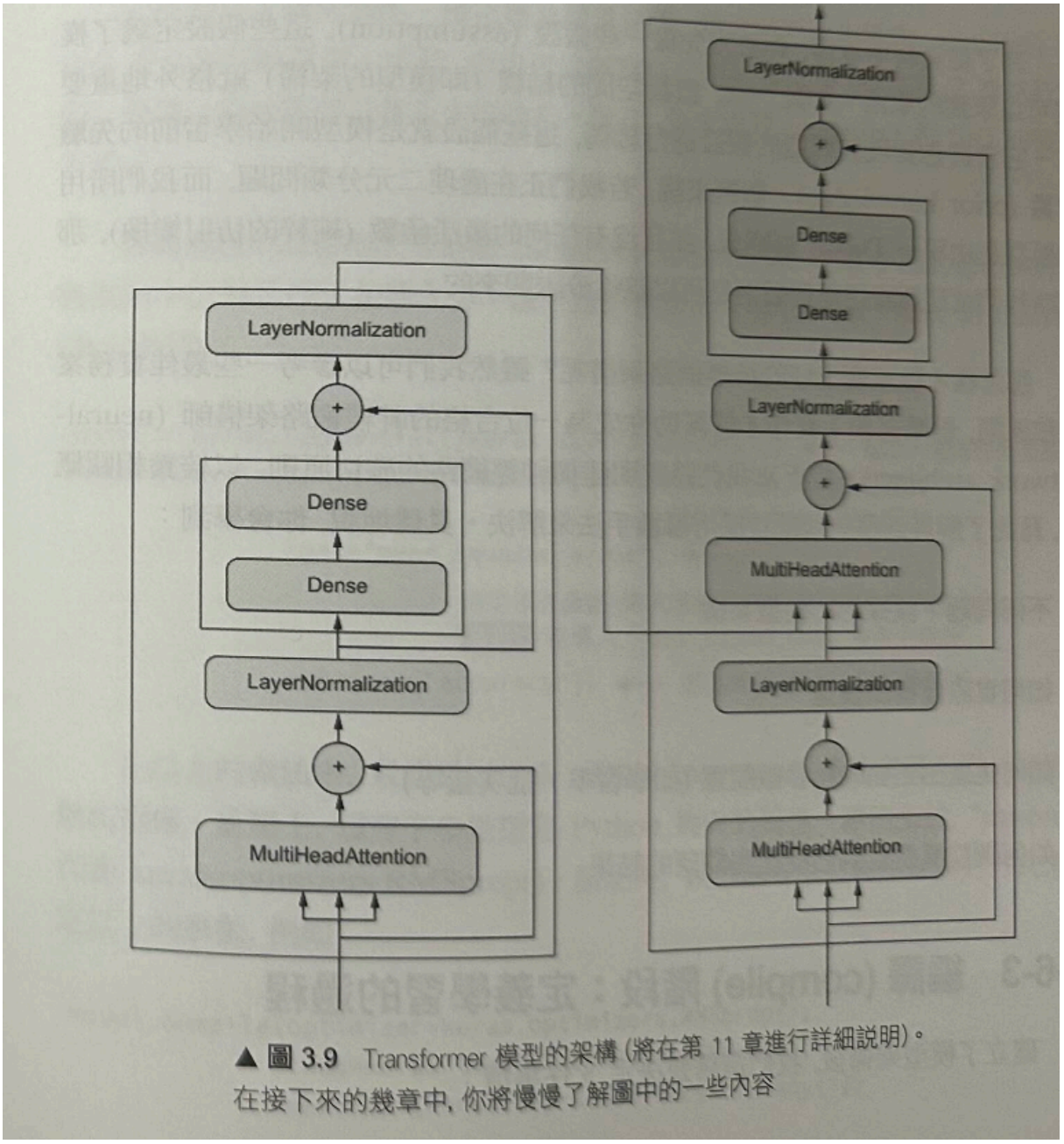
3-6-2 從層的模型

深度學習模型就是由多個層組成的結構，在Keras中是以Model類別來建立。 model就是layer堆疊組合而成，而layer還具備不同的組合模式其中一種稱為神經網路拓樸。

常見的網路拓樸如下：

- **雙分支神經網路(Two-branch network)：中間有分支，而非只有線性連接**
- **多端口網路(Multtihead networks)：有多個輸入或輸出**
- **殘差連接(Residual connections)：某些層的輸出會多一條分支跳接到較遠的層**

網路拓樸可以很複雜，以下展示TensorFlow模型的拓樸圖，該架構常用來處理文字資料



▲ 圖 3.9 Transformer 模型的架構 (將在第 11 章進行詳細說明)。在接下來的幾章中, 你將慢慢了解圖中的一些內容

模型的拓樸定義了一個假設空間，選擇特定的神經網路拓樸後就能將假設空間邦定的一系列張量運算上，藉此將輸入轉換成對應的輸出。選好網路拓樸後下一步就是找一組讓張量運算(預測)發揮到最佳效果的權重張量。

為了從資料中學習必須做一些假設(assumption)，假設會定義模型可以學到的東西。如此一來假設空間的結構(即模型的架構)就很重要了，這些假設就是模型開始學習前的先備知識(prior knowledge)。舉例來說，若一個二元分類問題，而用的模型式油

單一的Dense層組成，且沒有其他的激活函式(純粹的仿設變換)，那此時假設就是這兩類別是可以被線性分隔開來的。

3-6-3 編譯(compile)階段：定義學習的過程

確立了模型架構後，還需決定3件事。

1. **損失函數(目標函數)：訓練期間要將此函數的傳回值最小化，這是衡量任務成功與否的關鍵。(給優化器看的)**
2. **優化器：決定如何根據損失參數來更新神經網路，使損失變小。一般來說會以隨機梯度下降法(SGD)所言生的方法來執行**
3. **評量指標：用來評量訓練階段和測試階段的模型表現，如分類準確度。與損失值不同，訓練並不會直接使用這些指標進行優化，因此指標不一定要是可以微分的。(給人看的)**

一旦決定了上述三項，就可以利用內建的compile()和fit()方法來訓練模型。當然也可以自行撰寫訓練迴圈。

訓練過程中的各項配置是由compile()方法定義。參數有：optimizer、loss和metrics(一個串列)

```
model =  keras.Sequential([keras.layers.Dense(1)]) #建立一個線性分類器

model.compile(optimizer = "rmsprop", #指定優化器
                    loss = "mean_squared_error",  #指定損失函數
                    metrics = ["accuracy"]#定義指標
)
```

用字串(rmsprop)定義優化器、損失函數和指標。這些字串是指定Python物件的捷徑，如：rmsprop代表 keras.optimizers.RMSprop()。

```
model.compile(optimizer = keras.optimizers.RMSprop(), #指定優化器
                    loss = keras.losses.meanSquardedError,  #指定損失函數
```

```
                    metrics = [keras.metrics.BinaryAccuracy()]#定義指標
)
```

以上則是物件的方式來指定參數

3-6-4 選擇損失函數

為特定問題選正確的損失參數十分重要，因為神經網路會據此調整參數以減少損失值。神經網路會徹底的執行降低損失函數的任務，所以要明確的選擇損失函數，否則會導致意想不到的副作用。

當遇到分類、遞迴和序列化預測等常見問題時可以遵循簡單的準則來選擇正確的損失函數。如：二元交叉熵(binary crossentropy)可以處裡二元分類問題；分類交叉熵(categorical crossentropy)處理多類別分類問題。只有在處理全新的研究問題時才需開發自己的損失函數。

3-6-5 搞懂fit()方法

使用 fit()後會資型實作訓練迴圈，其關鍵參數如下

- **輸入(input)和目標值(targets)：用來訓練的資料，包括輸入樣本和目標答案。會以Numpy陣列或TensorFlow的Dataset物件形式傳入**
- **週期數(epochs)：訓練迴圈的重複次數，也就是要用所有資料重複訓練多少次。**
- **批次量(batch_size)：在每一週期中，進行小批次梯度下降訓練時的批次量，也就是每次訓練並更新權重時用來計算梯度的訓練樣本數。**

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

inputs = np.array([
    [1, 2],
    [3, 4],
    [5, 6]
])
```

```
targets = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

model = Sequential([
    Dense(3, input_dim=2)
])

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

history = model.fit(inputs, targets, epochs=5, batch_size=128)

print(history.history)

——→輸出結果
{'loss': [29.25340461730957, 29.1826229095459, 29.11199378967285, 29.
04152488708496, 28.971206665039062]}
```

呼叫fit()會傳一個history物件。該物件的history屬性是一個字典，key為"loss"或其他評量指標名稱，value則是每一訓練週期的損失值或指標值。

3-6-6 用驗證資料來監控損失和指標

機械學習的目標並非取得只在訓練資料上表現良好的模型，而是取得在大部分狀況下都表現良好的模型。為了分析模型在資料上的表現，一般會保留訓練資料的一部份作為驗證資料(validation data)。驗證資料不會用來訓練模型，但會用它來計算損失值和指標值。可以透過fit()中的validation_data參數來傳入驗證資料。驗證資料可以是Numpy陣列或Dataset物件。

```
import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
inputs = np.array([
    [1, 2],
    [3, 4],
    [5, 6]
])
targets = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mean_squared_error'])

indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs)) #保留30%的訓練資料作為驗證資料
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]

model.fit(
    training_inputs,
    training_targets,
    epochs=100,
    batch_size=16,
    validation_data=(val_inputs, val_targets) #驗證資料,只用來計算驗證損失和指標
    )
```

在驗證資料上的損失值稱為「驗證損失」和訓練損失不同。

在訓練結束後，若想用特定資料來計算驗證損失和指標值可以使用evaluate()方法

```
loss_and_metrics = model.evaluate(val_inputs, val_targets, batch_size=16)
```

該方法會對傳入的資料進行小批次(量由batch_size決定)，進而回傳由多個純量組成的串列，其中第一個純量為驗證損失，緊接便是驗證指標值。若沒設定指標則指會回傳驗證損失

3-6-7 推論(Inference)階段：使用訓練好的模型來預測

訓練好模型後，就可以用它對新資料進行預測，這稱之為推論(inference)。用predict()方法來進行推論。這樣就會在小批次的資料上迭代並回傳存有預測值的Numpy陣列或Dataset物件

```
predictions = model.predict(inputs,batch_size=16)

print(predictions[:10]) #回傳10個樣本預測值
```