Keras和Tensorflow簡介

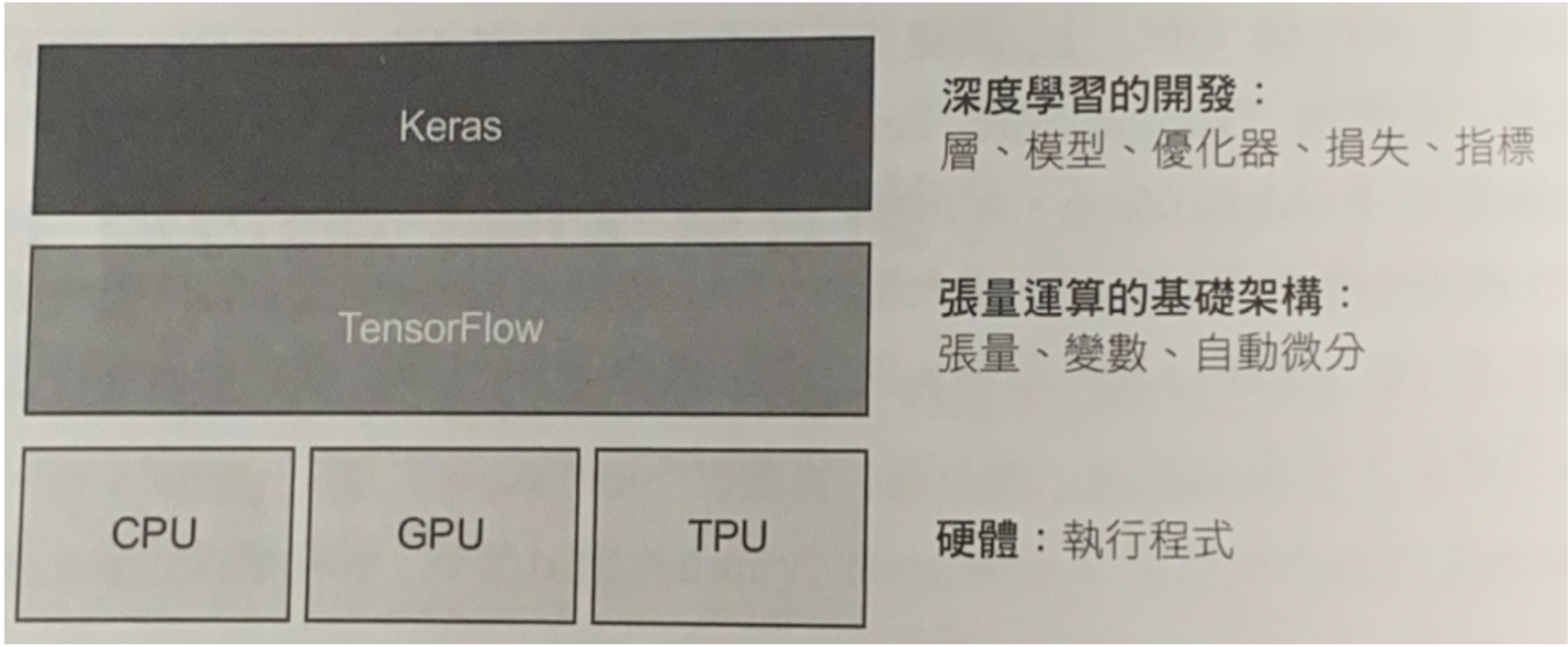
TensorFlow

- Python機器學習框架
- 能在任何可微分函數計算梯度
- 可在CPU、GPU、TPU等高度平行的硬體加速器運行
- 運算程序可輕易分散到多台機器共同執行
- 可匯出各種不同語言程式

Keras

- Python的深度學習API
- 處理模型的建立、訓練等
- 專為人類設計的API
- 根據使用者需求，提供各種工作流程

Keres 與TensorFlow



深度學習開發環境

在GPU上進行深度學習選擇

- GPU顯示卡
- Google Cloud或AWS EC2上使用
- Google Colaboratory

低階張量運算

- 存放網路狀態的特殊張量
- 張量操作
- 反向傳播

高階深度學習概念

- 層
- 模型
- 損失函數
- 優化器
- 用來評估模型表現的各種評量指標
- 執行小批次隨機梯度下降的訓練迴圈

1或0組成的張量

```
[4]  import tensorflow as tf
     x=tf.ones(shape=(2,1))
     print(x)
     y=tf.zeros(shape=(2,1))
     print(y)
```

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

亂數組成的張量

```
[4]  x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
     print(x)
     y = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
     print(y)
```

```
tf.Tensor(
[[ 0.5287678]
 [-1.1640913]
 [ 0.5501221]], shape=(3, 1), dtype=float32)
tf.Tensor(
[[0.31097543]
 [0.5328946 ]
 [0.03935754]], shape=(3, 1), dtype=float32)
```

指派新值

```
[5]  import numpy as np
     x = np.ones(shape=(2, 2))
     x[0, 0] = 0.
     print(x)
```

```
     [[0. 1.]
      [1. 1.]]
```

```
[23] import tensorflow as tf
     v=tf.Variable(initial_value=tf.random.normal(shape=(3,1)))
     print(v)
```

```
     <tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
     array([[-1.0191846 ],
            [ 0.30869547],
            [-1.0606197 ]], dtype=float32)>
```

```
[24] v.assign(tf.ones((3,1)))
```

```
     <tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
     array([[1.],
            [1.],
            [1.]], dtype=float32)>
```

```
[25] v[0,0].assign(3)
```

```
     <tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
     array([[3.],
            [1.],
            [1.]], dtype=float32)>
```

```
[27] v.assign_add(tf.ones((3,1)))
```

```
     <tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
     array([[4.],
            [2.],
            [2.]], dtype=float32)>
```

數學運算

a=tf.ones((2,2))	
b=tf.**square**(a)	平方
c=tf.**sqrt**(a)	平方根
d=b**+**c	張量相加
e=tf.**matmul**(a,b)	張量點積
e***=**d	張量相乘

GradientTape

取得任意可微分函數對任意輸入項的梯度

```
[34] input_var = tf.Variable(initial_value=3.)
     with tf.GradientTape() as tape:
        result = tf.square(input_var)
     gradient = tape.gradient(result, input_var)
     print(gradient)
```

```
     tf.Tensor(6.0, shape=(), dtype=float32)
```

```
[35] input_const = tf.constant(3.)
     with tf.GradientTape() as tape:
        tape.watch(input_const)
        result = tf.square(input_const)
     gradient = tape.gradient(result, input_const)
     print(gradient)
```

```
     tf.Tensor(6.0, shape=(), dtype=float32)
```

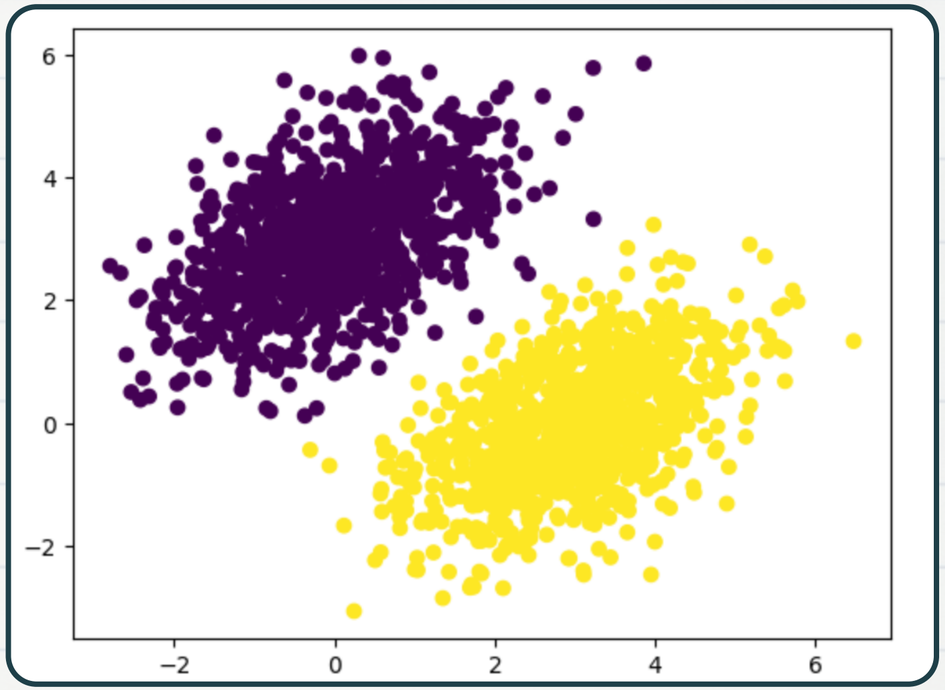
二階梯度

```
[39] time = tf.Variable(0.)
     with tf.GradientTape() as outer_tape:
         with tf.GradientTape() as inner_tape:
             position =  4.9 * time ** 2
         speed = inner_tape.gradient(position, time)
     acceleration = outer_tape.gradient(speed, time)
     print(acceleration)
```

```
tf.Tensor(9.8, shape=(), dtype=float32)
```

線性分類器

```
[46] import matplotlib.pyplot as plt
     num_samples_per_class = 1000
     negative_samples = np.random.multivariate_normal(
         mean=[0, 3],
         cov=[[1, 0.5],[0.5, 1]],
         size=num_samples_per_class)
     positive_samples = np.random.multivariate_normal(
         mean=[3, 0],
         cov=[[1, 0.5],[0.5, 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")))

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



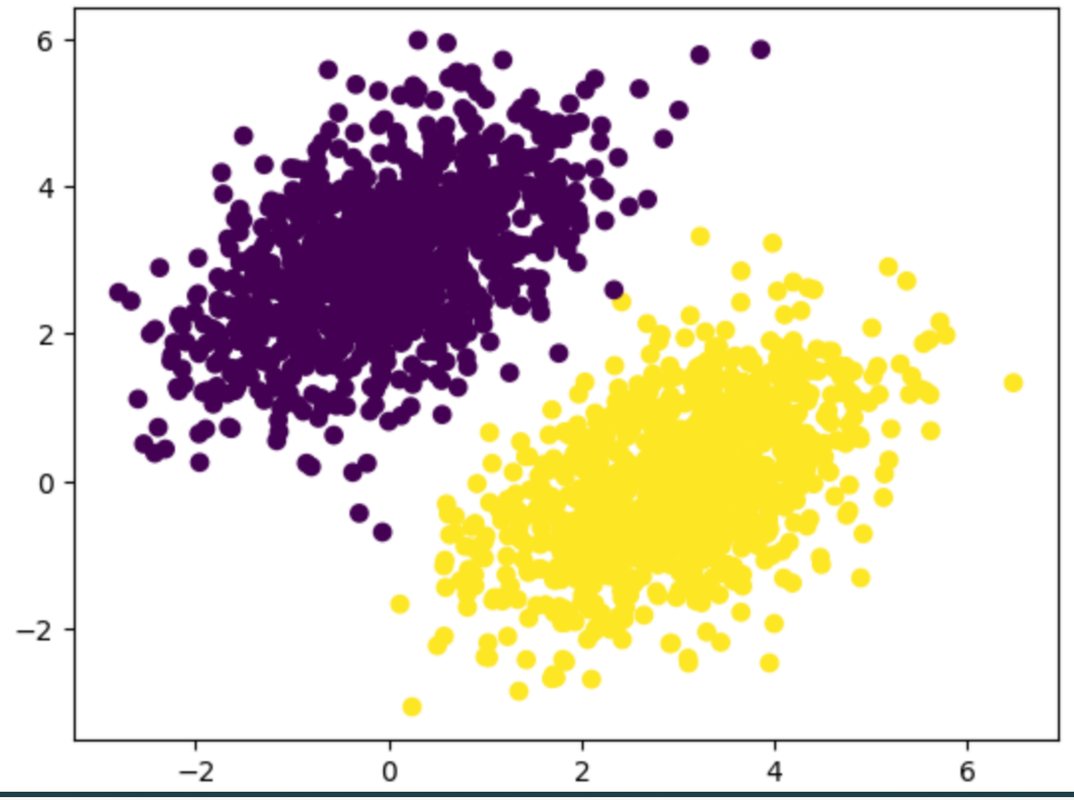
線性分類器

```
[49] input_dim = 2
     output_dim = 1
     W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
     b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))

     def model(inputs):
         return tf.matmul(inputs, W) + b

     def square_loss(targets, predictions):
         per_sample_losses = tf.square(targets - predictions)
         return tf.reduce_mean(per_sample_losses)

     learning_rate = 0.1

     def training_step(inputs, targets):
         with tf.GradientTape() as tape:
             predictions = model(inputs)
             loss = square_loss(targets, predictions)
         grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
         W.assign_sub(grad_loss_wrt_W * learning_rate)
         b.assign_sub(grad_loss_wrt_b * learning_rate)
         return loss

     for step in range(40):
         loss = training_step(inputs, targets)
         print(f"Loss at step {step}: {loss:.4f}")
```

```
Loss at step 0: 0.7333
Loss at step 1: 0.1755
Loss at step 2: 0.1102
Loss at step 3: 0.0964
Loss at step 4: 0.0891
Loss at step 5: 0.0832
Loss at step 6: 0.0778
Loss at step 7: 0.0729
Loss at step 8: 0.0685
Loss at step 9: 0.0645
Loss at step 10: 0.0608
Loss at step 11: 0.0575
Loss at step 12: 0.0545
Loss at step 13: 0.0518
Loss at step 14: 0.0493
Loss at step 15: 0.0470
Loss at step 16: 0.0449
Loss at step 17: 0.0430
Loss at step 18: 0.0413
Loss at step 19: 0.0398
Loss at step 20: 0.0384
Loss at step 21: 0.0371
Loss at step 22: 0.0359
Loss at step 23: 0.0349
Loss at step 24: 0.0339
Loss at step 25: 0.0330
Loss at step 26: 0.0322
Loss at step 27: 0.0315
Loss at step 28: 0.0309
Loss at step 29: 0.0303
Loss at step 30: 0.0297
Loss at step 31: 0.0292
Loss at step 32: 0.0288
Loss at step 33: 0.0284
Loss at step 34: 0.0280
Loss at step 35: 0.0276
Loss at step 36: 0.0273
Loss at step 37: 0.0271
Loss at step 38: 0.0268
Loss at step 39: 0.0266
```

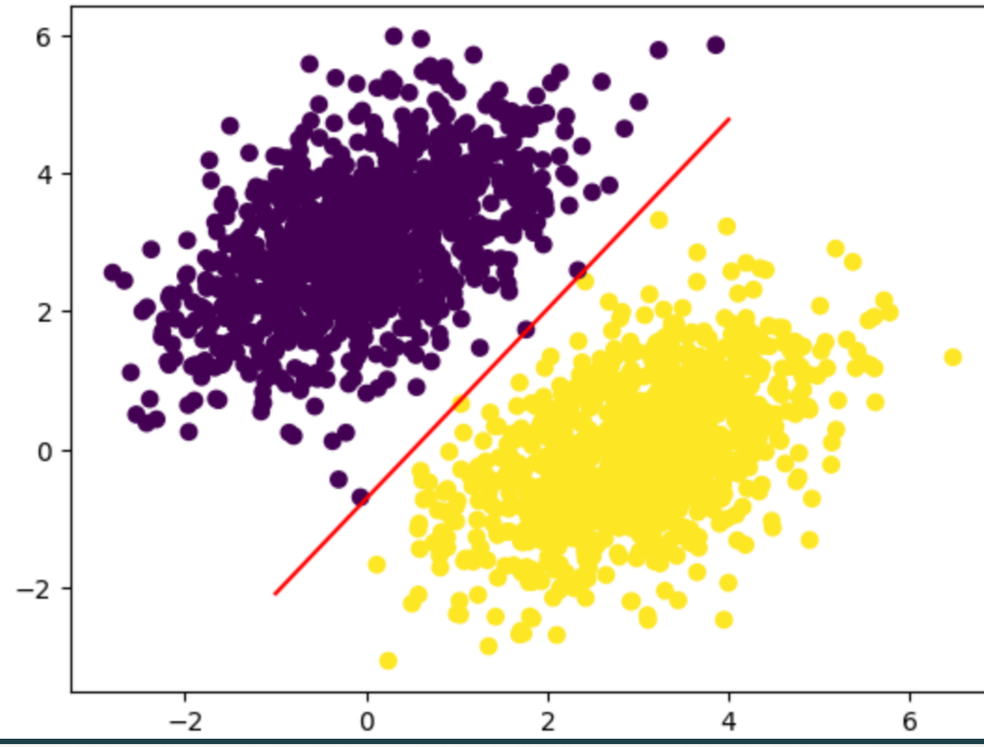
The writers

```
[50] predictions = model(inputs)
     plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
     plt.show()
```




```
[51] x = np.linspace(-1, 4, 100)
     y = - W[0] /  W[1] * x + (0.5 - b) / W[1]
     plt.plot(x, y, "-r")
     plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
```

```
<matplotlib.collections.PathCollection at 0x79a918474e80>
```



Layer(層)

Layer類別是keras的核心。layer將一些狀態（權重）和運算（正向傳播）包在一起的物件。會使用build()來建立權重，正向傳播的運算過程使用call()來定義

```
52] from tensorflow import keras

    class SimpleDense(keras.layers.Layer):

        def __init__(self, units, activation=None):
            super().__init__()
            self.units = units
            self.activation = activation

        def build(self, input_shape):
            input_dim = input_shape[-1]
            self.W = self.add_weight(shape=(input_dim, self.units),
                                     initializer="random_normal")
            self.b = self.add_weight(shape=(self.units,),
                                     initializer="zeros")

        def call(self, inputs):
            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=tf.nn.relu)
    input_tensor = tf.ones(shape=(2, 784))
    output_tensor = my_dense(input_tensor)
    print(output_tensor.shape)

    (2, 32)
```

自動推論出權重的shape

keras模型中每一層權重張量，都是由keras自動依照第一次輸入的張量即時建立的，會自動配合上一層的shape，不需指定有關輸入的shape

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

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

自動推論出權重的shape

將第二章的NaiveDense改寫為能自動推論shape的keras層，就需要有build()、call()方法，以build()來創建權重，接受輸入資料為其參數。

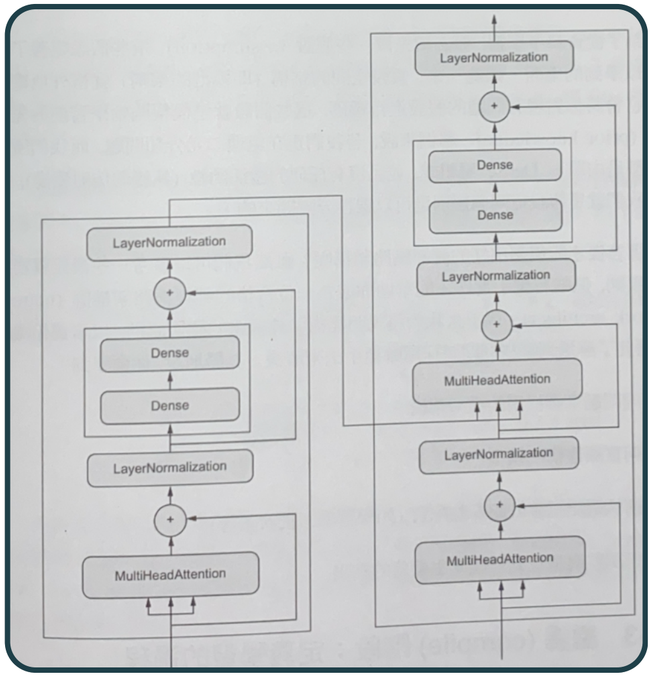
```
[ ]  model = NaiveSequential([
         NaiveDense(input_size=784,output_size=32, activation="relu"),
         NaiveDense(input_size=32,output_size=64, activation="relu"),
         NaiveDense(input_size=64,output_size=32, activation="relu"),
         NaiveDense(input_size=32,output_size=10, activation="relu"),
     ])
```

```
[ ]  model = keras.Sequential([
         SimpleDense(32, activation="relu"),
         SimpleDense(64, activation="relu"),
         SimpleDense(32, activation="relu"),
         SimpleDense(10, activation="softmax")
     ])
```

模型

由多個層組成
以Model類別來建立模型物件

- 序列式模型
- 雙分支神經網路
- 多端口網路
- 殘差連接



模型

模型的拓樸定義一個假設空間，選擇特定的神經網路拓樸，就能將可能空間綁到特定的一 系列張量運算上，借此將輸入資料轉換為對應的輸出資料。

選好模型拓樸之後，接下來要搜尋一組可以讓張量運算(預測)發揮最佳效果的權重張量 。

為了從資料中學習，必須先做一些假設，這些假設定義了模型可以學到的東西。假設空間的結構(模型的架構)會將現有問題的假設進行編碼，這些假設就是模型開始學習前的先驗知識

編譯階段

損失函數
(目標函數)

訓練期間要逐漸將此函數的傳回職最小化

優化器

決定如何根據損失函數更新神經網路，使損失變小

評量指標

評量訓練及測試階段的模型表現

compile()方法

```
[ ]  model = keras.Sequential([keras.layers.Dense(1)])
     model.compile(optimizer="rmsprop",
                   loss="mean_squared_error",
                   metrics=["accuracy"])
```

選擇損失函數

神經網路會根據損失函數調整權重參數，以減少損失值。

- 二元交叉熵：處理二元分類問題
- 分類交叉熵：處理多類別分類問題

fit()方法關鍵參數

- inputs(輸入)、targets(目標值)：用來訓練的資料，包括輸入樣本和目標答案
- epochs(週期數)：訓練迴圈的重複次數
- batch_size(批次量)：每次訓練並更新權重時用來計算梯度的訓練樣本數量

fit()

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

```
[69] history.history

    {'loss': [3.5874383449554443,
      3.3375840187072754,
      3.1346139907836914,
      2.944870710372925,
      2.7645723819732666],
     'binary_accuracy': [0.4724999964237213,
      0.4704999923706054 7,
      0.46700000762939453,
      0.4650000035762787,
      0.4629999952316284]}
```

驗證資料監控損失值和指標

為得知模型在新資料上表現，會從訓練資料保留一部分作為驗證資料。

驗證資料用來計算損失值和指標值，不會用來訓練模型。

在訓練階段時，模型不可先見過驗證資料，否則會有瑕疵。

validation_data

```
[70] model = keras.Sequential([keras.layers.Dense(1)])
     model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
                   loss=keras.losses.MeanSquaredError(),
                   metrics=[keras.metrics.BinaryAccuracy()])

     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))
     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=5,
         batch_size=16,
         validation_data=(val_inputs, val_targets)
     )
```

推論階段

```
[71] predictions = model.predict(val_inputs, batch_size=128)
     print(predictions[:10])
```

```
5/5 [==============================] - 0s 3ms/step
[[1.2259885 ]
 [0.97408795]
 [0.20160255]
 [0.35629275]
 [0.16749787]
 [0.16357791]
 [1.1469212 ]
 [1.0141813 ]
 [0.9651272 ]
 [1.1175606 ]]
```

Thank's For Watching