

Họ và tên: Trần Văn Anh
MSV: B20DCCN075

BÀI TẬP 3: TENSORFLOW

3.1.

3.1.1.

```
#3.1.1

import tensorflow as tf

# creating nodes in computation graph
# Tạo các hằng số TensorFlow
node1 = tf.constant(3, dtype=tf.int32) # Tạo một hằng số kiểu int32 với giá trị 3
node2 = tf.constant(5, dtype=tf.int32) # Tạo một hằng số kiểu int32 với giá trị 5

# Tạo một nút mới là tổng của node1 và node2
node3 = tf.add(node1, node2)

# create tensorflow session object
sess = tf.compat.v1.Session() #TensorFlow 2.x, không cần tạo phiên bản của tf.compat.v1.Session()

# evaluating node3 and printing the result
print("sum of node1 and node2 is :", sess.run(node3))

sum of node1 and node2 is : 8
```

- **tf.constant** được sử dụng để tạo các hằng số
- **dtype=tf.int32** chỉ định kiểu dữ liệu của hằng số, ở đây là số nguyên 32-bit.
- **tf.add** để tạo một nút mới **node3** là tổng của **node1** và **node2**
- Trong TensorFlow 1.x, việc sử dụng phiên làm việc (**tf.Session()**) là cách để thực hiện tính toán và quản lý các biến và phép tính trong một "đồ thị tính toán" (computation graph). Các biến và phép tính trong TensorFlow 1.x được xây dựng trong đồ thị này, và để thực hiện các tính toán trên chúng, bạn phải tạo một phiên làm việc, sau đó chạy các phép tính trong phiên làm việc đó.
- Trong TensorFlow 2.x, không cần sử dụng **tf.compat.v1.Session()** để chạy các phép tính như trong TensorFlow 1.x. Lý do chính là vì TensorFlow 2.x đã kích hoạt eager execution (thực thi tức thì) mặc định, do đó ta có thể thực hiện các phép tính trực tiếp bằng cách sử dụng các biến và hằng số TensorFlow

3.1.2.

```
# 3.1.2

# Import phiên bản TensorFlow 1.x
import tensorflow.compat.v1 as tf

# Tạo các hằng số TensorFlow
x = tf.constant(5, tf.float32) # Một hằng số kiểu float32 với giá trị 5
y = tf.constant([5], tf.float32) # Một mảng có một phần tử kiểu float32 với giá trị 5
z = tf.constant([5, 3, 4], tf.float32) # Một mảng có ba phần tử kiểu float32
t = tf.constant([[5, 3, 4, 6], [2, 3, 4, 7]], tf.float32) # Một ma trận 2D kích thước 2x4 kiểu float32
u = tf.constant([[[5, 3, 4, 6], [2, 3, 4, 0]]], tf.float32) # Một ma trận 3D kích thước 1x2x4 kiểu float32
v = tf.constant([[[5, 3, 4, 6], [2, 3, 4, 0]],
                [[5, 3, 4, 6], [2, 3, 4, 0]],
                [[5, 3, 4, 6], [2, 3, 4, 0]]], tf.float32) # Một ma trận 3D kích thước 3x2x4 kiểu float32

# In giá trị của y
print(y)

tf.Tensor([5.], shape=(1,), dtype=float32)
```

- Sử dụng **t.shape[0]** để truy cập số hàng của **t**
- Sử dụng **v.shape[2]** để truy cập số cột của **v**

3.2.

3.2.1.

```
#3.2.1

# Import phiên bản TensorFlow 1.x và tắt eager execution (thực thi tức thì)
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

# Khai báo các biến và thực hiện phép nhân
x1 = tf.Variable(5.3, tf.float32) # Tạo biến x1 kiểu float32 với giá trị khởi tạo 5.3
x2 = tf.Variable(4.3, tf.float32) # Tạo biến x2 kiểu float32 với giá trị khởi tạo 4.3
x = tf.multiply(x1, x2) # Tạo biến x là tích của x1 và x2

# Khởi tạo biến
init = tf.global_variables_initializer()

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Chạy bước khởi tạo để khởi tạo tất cả các biến
    sess.run(init)

    # Đánh giá biến x và in kết quả
    t = sess.run(x)
    print("Kết quả của phép nhân x1 và x2 là:", t)
```

Kết quả của phép nhân x1 và x2 là: 22.79

- **tf.Variable** được sử dụng để tạo biến

- **tf.multiply** để tạo nút mới **x** là tích của **x1** và **x2**
- **tf.global_variables_initializer** để khởi tạo giá trị ban đầu cho tất cả các biến trong đồ thị tính toán
- **tf.Session()** để tạo 1 phiên làm việc với TensorFlow.
- **sess.run**: Thực hiện tính toán cho biến hoặc phép tính được chỉ định.

3.2.2.

```
#3.2.2

# Import phiên bản TensorFlow 1.x và tắt eager execution (thực thi tức thì)
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

# Khai báo và khởi tạo biến x1 và x2 với giá trị ban đầu
x1 = tf.Variable([[5.3, 4.5, 6.0], [4.3, 4.3, 7.0]], tf.float32)
x2 = tf.Variable([[4.3, 4.3, 7.0], [5.3, 4.5, 6.0]], tf.float32)

# Tạo biến x là tích của x1 và x2
x = tf.multiply(x1, x2)

# Khởi tạo biến
init = tf.global_variables_initializer()

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Chạy bước khởi tạo để khởi tạo tất cả các biến
    sess.run(init)

    # Đánh giá biến x và lấy giá trị kết quả
    t = sess.run(x)

    # In kết quả tích của x1 và x2
    print("Kết quả của phép nhân x1 và x2 là:")
    print(t)
```

```
[[22.79 19.35 42.  ]
 [22.79 19.35 42.  ]]
```

3.2.3.

```
#3.2.3

import tensorflow.compat.v1 as tf

# Tạo một biến TensorFlow có giá trị ban đầu là ma trận 2x2 chứa các giá trị 0.0
node = tf.Variable(tf.zeros([2, 2]))

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Khởi tạo tất cả các biến toàn cục trong đồ thị tính toán
    sess.run(tf.global_variables_initializer())

    # Đánh giá giá trị của biến "node" và in ra màn hình trước khi thực hiện phép cộng
    print("Tensor value before addition:\n", sess.run(node))

    # Thực hiện phép cộng từng phần tử của ma trận node với ma trận 2x2 chứa toàn giá trị 1.0
    node = node.assign(node + tf.ones([2, 2]))

    # Đánh giá giá trị của biến "node" sau khi thực hiện phép cộng và in ra màn hình
    print("Tensor value after addition:\n", sess.run(node))
    # Đóng phiên làm việc
    sess.close()
```

Tensor value before addition:

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

Tensor value after addition:

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

- **tf.zeros([2, 2]):** Hàm này tạo một ma trận có kích thước 2x2 chứa toàn giá trị 0.0
- **tf.ones([2, 2]):** Hàm này tạo một ma trận có kích thước 2x2 chứa toàn giá trị 1.0
- **node.assign(node + tf.ones([2, 2])):** Sử dụng phép gán **node.assign()** để thay đổi giá trị của node
- **node.assign(node + tf.ones([2, 2])) ~ node.assign_add(tf.ones([2, 2]))**

3.3.

3.3.1.

```
# 3.3.1

import tensorflow.compat.v1 as tf

# Tắt eager execution trong TensorFlow 2.x
tf.compat.v1.disable_eager_execution()

# Khai báo một placeholder x với kiểu dữ liệu float32 và kích thước None (được chỉ định sau)
x = tf.placeholder(tf.float32, None)

# Tạo một phép toán y là phép cộng của x với chính nó
y = tf.add(x, x)

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Định nghĩa giá trị cho x_data
    x_data = 5.0

    # Thực hiện phép toán y bằng cách cung cấp giá trị cho placeholder x
    # Sử dụng feed_dict để gán giá trị x_data cho x
    result = sess.run(y, feed_dict={x: x_data})

    # In kết quả của phép toán y ra màn hình
    print(result)
```

10.0

- **tf.placeholder**

- Placeholder là một biến đặc biệt được sử dụng để định nghĩa một vị trí trong biểu đồ tính toán mà sẽ được cung cấp dữ liệu sau này.
- Placeholder không chứa giá trị thực sự khi bạn định nghĩa chúng; thay vào đó, chúng chỉ định kiểu dữ liệu và hình dạng (shape) của dữ liệu sẽ được cung cấp sau này.
- Thường được sử dụng khi muốn tạo một biểu đồ tính toán linh hoạt và chạy nhiều lần với các dữ liệu khác nhau

- **feed_dict** được sử dụng để ánh xạ từng placeholder với giá trị tương ứng

3.3.2

```
# 3.3.2

import tensorflow.compat.v1 as tf

# Tắt eager execution trong TensorFlow 2.x
tf.compat.v1.disable_eager_execution()

# Định nghĩa placeholder x với kiểu dữ liệu float32 và hình dạng [None, 3]
# [None, 3] có nghĩa là có một số không xác định (None) của dòng và 3 cột
x = tf.placeholder(tf.float32, [None, 3])

# Tạo một phép toán y là phép cộng của x với chính nó
y = tf.add(x, x)

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Định nghĩa giá trị cho x_data, một ma trận có 1 dòng và 3 cột
    x_data = [[1.5, 2.0, 3.3]]

    # Thực hiện phép toán y bằng cách cung cấp giá trị cho placeholder x thông qua feed_dict
    result = sess.run(y, feed_dict={x: x_data})

    # In kết quả của phép toán y ra màn hình
    print(result)
```

```
[[3.  4.  6.6]]
```

3.3.3.

```
# 3.3.3

import tensorflow.compat.v1 as tf

# Tắt eager execution trong TensorFlow 2.x
tf.compat.v1.disable_eager_execution()

# Định nghĩa placeholder x với kiểu dữ liệu float32 và hình dạng [None, None, 3]
# [None, None, 3] có nghĩa là có một số không xác định (None) của các chiều dòng và cột, và mỗi dòng có 3 cột
x = tf.placeholder(tf.float32, [None, None, 3])

# Tạo một phép toán y là phép cộng của x với chính nó
y = tf.add(x, x)

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Định nghĩa giá trị cho x_data, một ma trận có 1 dòng và 3 cột trong chiều cuối cùng
    x_data = [[[1, 2, 3]]]

    # Thực hiện phép toán y bằng cách cung cấp giá trị cho placeholder x thông qua feed_dict
    result = sess.run(y, feed_dict={x: x_data})

    # In kết quả của phép toán y ra màn hình
    print(result)
```

```
[[[2.  4.  6.]]]
```

3.3.4.

```
# 3.3.4

import tensorflow.compat.v1 as tf

# Tắt eager execution trong TensorFlow 2.x
tf.compat.v1.disable_eager_execution()

# Định nghĩa placeholder x với kiểu dữ liệu float32 và hình dạng [None, 4, 3]
# [None, 4, 3] có nghĩa là có một số không xác định (None) của các chiều dòng, có 4 dòng và mỗi dòng có 3 cột
x = tf.placeholder(tf.float32, [None, 4, 3])

# Tạo một phép toán y là phép cộng của x với chính nó
y = tf.add(x, x)

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Định nghĩa giá trị cho x_data, một ma trận có 4 dòng và 3 cột
    x_data = [[1, 2, 3],
               [2, 3, 4],
               [2, 3, 5],
               [0, 1, 2]]

    # Thực hiện phép toán y bằng cách cung cấp giá trị cho placeholder x thông qua feed_dict
    result = sess.run(y, feed_dict={x: x_data})

    # In kết quả của phép toán y ra màn hình
    print(result)
```

```
[[[ 2.  4.  6.]
  [ 4.  6.  8.]
  [ 4.  6. 10.]
  [ 0.  2.  4.]]]
```

3.3.5.

```
# 3.3.5

import tensorflow.compat.v1 as tf

# Tắt eager execution trong TensorFlow 2.x
tf.compat.v1.disable_eager_execution()

# Định nghĩa placeholder x với kiểu dữ liệu float32 và hình dạng [2, 4, 3]
# [2, 4, 3] có nghĩa là có 2 ma trận, mỗi ma trận có 4 dòng và mỗi dòng có 3 cột
x = tf.placeholder(tf.float32, [2, 4, 3])

# Tạo một phép toán y là phép cộng của x với chính nó
y = tf.add(x, x)

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Định nghĩa giá trị cho x_data, gồm 2 ma trận mỗi ma trận có 4 dòng và 3 cột
    x_data = [[1, 2, 3],
               [2, 3, 4],
               [2, 3, 5],
               [0, 1, 2]],

               [[1, 2, 3],
               [2, 3, 4],
               [2, 3, 5],
               [0, 1, 2]]

    # Thực hiện phép toán y bằng cách cung cấp giá trị cho placeholder x thông qua feed_dict
    result = sess.run(y, feed_dict={x: x_data})

    # In kết quả của phép toán y ra màn hình
    print(result)
```

```
[[[ 2.  4.  6.]
  [ 4.  6.  8.]
  [ 4.  6. 10.]
  [ 0.  2.  4.]]
```

```
[[[ 2.  4.  6.]
  [ 4.  6.  8.]
  [ 4.  6. 10.]
  [ 0.  2.  4.]]]
```


3.3.6.

```
# 3.3.6

import tensorflow.compat.v1 as tf

# Tắt eager execution trong TensorFlow 2.x
tf.compat.v1.disable_eager_execution()

# Định nghĩa placeholder x và y với kiểu dữ liệu float32 và hình dạng [2, 4, 3]
# [2, 4, 3] có nghĩa là có 2 ma trận, mỗi ma trận có 4 dòng và mỗi dòng có 3 cột
x = tf.placeholder(tf.float32, [2, 4, 3])
y = tf.placeholder(tf.float32, [2, 4, 3])

# Tạo phép toán z là phép cộng của x và y
z = tf.add(x, y)

# Tạo phép toán u là phép nhân element-wise (nhân từng phần tử tương ứng của x và y)
u = tf.multiply(x, y)

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Định nghĩa giá trị cho x_data và y_data, mỗi ma trận có 4 dòng và 3 cột
    x_data = [[1, 2, 3],
               [2, 3, 4],
               [2, 3, 5],
               [0, 1, 2]],

               [[1, 2, 3],
                [2, 3, 4],
                [2, 3, 5],
                [0, 1, 2]]

    y_data = [[1, 2, 3],
               [2, 3, 4],
               [2, 3, 5],
               [0, 1, 2]],

               [[1, 2, 3],
                [2, 3, 4],
                [2, 3, 5],
                [0, 1, 2]]

    # Thực hiện phép toán z bằng cách cung cấp giá trị cho placeholder x và y thông qua feed_dict
    result1 = sess.run(z, feed_dict={x: x_data, y: y_data})

    # Thực hiện phép toán u bằng cách cung cấp giá trị cho placeholder x và y thông qua feed_dict
    result2 = sess.run(u, feed_dict={x: x_data, y: y_data})

    # In kết quả của phép toán z và u ra màn hình
    print("result1 =", result1)
    print("result2 =", result2)
```

```

result1 = [[[ 2.  4.  6.]
 [ 4.  6.  8.]
 [ 4.  6. 10.]
 [ 0.  2.  4.]]]

[[ 2.  4.  6.]
 [ 4.  6.  8.]
 [ 4.  6. 10.]
 [ 0.  2.  4.]]]
result2 = [[[ 1.  4.  9.]
 [ 4.  9. 16.]
 [ 4.  9. 25.]
 [ 0.  1.  4.]]]

[[ 1.  4.  9.]
 [ 4.  9. 16.]
 [ 4.  9. 25.]
 [ 0.  1.  4.]]]

```

3.4.

3.4.1.

```

# 3.4.1

import tensorflow.compat.v1 as tf

# Tắt eager execution trong TensorFlow 2.x
tf.compat.v1.disable_eager_execution()

# Tạo hai hằng số x1 và x2 với kiểu dữ liệu float32 và giá trị cụ thể
x1 = tf.constant(5.3, tf.float32)
x2 = tf.constant(1.5, tf.float32)

# Tạo hai biến w1 và w2 với kiểu dữ liệu float32 và giá trị khởi tạo
w1 = tf.Variable(0.7, tf.float32)
w2 = tf.Variable(0.5, tf.float32)

# Tính tích của x1 và w1
u = tf.multiply(x1, w1)

# Tính tích của x2 và w2
v = tf.multiply(x2, w2)

# Tính tổng của u và v
z = tf.add(u, v)

# Áp dụng hàm sigmoid lên z để tính result
result = tf.sigmoid(z)

# Khởi tạo tất cả các biến
init = tf.global_variables_initializer()

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Chạy bước khởi tạo để khởi tạo tất cả các biến
    sess.run(init)

    # Thực hiện tính toán result và in kết quả ra màn hình
    print(sess.run(result))

```

0.9885698

- **tf.sigmoid(z)**

- Là một hàm số được sử dụng để tính giá trị sigmoid của một số thực. Hàm sigmoid được định nghĩa bởi công thức:

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

- Kết quả của **tf.sigmoid(z)** là một số thực nằm trong khoảng từ 0 đến 1, thể hiện xác suất trong các bài toán mạng neuron

3.4.2.

```
# 3.4.2

import numpy as np
import matplotlib.pyplot as plt

# Số lượng điểm dữ liệu
number_of_points = 500

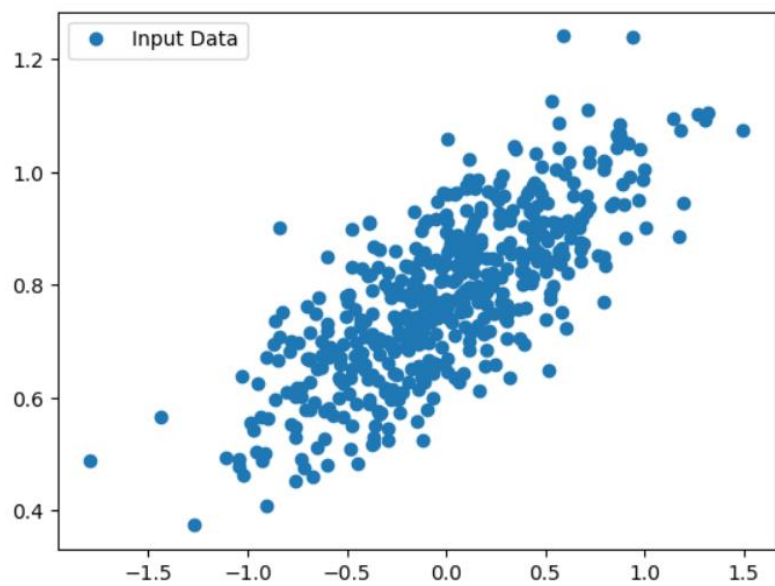
# Danh sách để lưu trữ các điểm dữ liệu
x_point = []
y_point = []

# Định nghĩa các hệ số a và b cho phương trình tuyến tính y = ax + b
a = 0.22
b = 0.78

# Tạo dữ liệu giả lập
for i in range(number_of_points):
    # Tạo một giá trị ngẫu nhiên x theo phân phối chuẩn (normal distribution)
    x = np.random.normal(0.0, 0.5)

    # Tính giá trị y dựa trên phương trình tuyến tính y = ax + b và thêm nhiễu ngẫu nhiên
    y = a * x + b + np.random.normal(0.0, 0.1)
    # Thêm x và y vào danh sách
    x_point.append([x])
    y_point.append([y])

# Vẽ biểu đồ dữ liệu
plt.plot(x_point, y_point, 'o', label='Input Data')
plt.legend()
plt.show()
```



3.4.3.

: # 3.4.3

```
import tensorflow.compat.v1 as tf

# Tắt eager execution trong TensorFlow 2.x
tf.compat.v1.disable_eager_execution()

# Định nghĩa hai placeholder x1 và x2 với kiểu dữ liệu float32 và hình dạng [None, 3]
# [None, 3] có nghĩa là số hàng không xác định (có thể là bất kỳ số hàng nào), và mỗi hàng có 3 cột
x1 = tf.placeholder(tf.float32, [None, 3])
x2 = tf.placeholder(tf.float32, [None, 3])

# Định nghĩa hai biến w1 và w2 với kiểu dữ liệu float32 và giá trị khởi tạo
w1 = tf.Variable([0.5, 0.4, 0.7], tf.float32)
w2 = tf.Variable([0.8, 0.5, 0.6], tf.float32)

# Tính tích element-wise của w1 và x1
u1 = tf.multiply(w1, x1)

# Tính tích element-wise của w2 và x2
u2 = tf.multiply(w2, x2)

# Tính tổng của u1 và u2
v = tf.add(u1, u2)

# Áp dụng hàm sigmoid lên v để tính z
z = tf.sigmoid(v)

# Khởi tạo tất cả các biến
init = tf.global_variables_initializer()

# Bắt đầu một phiên làm việc với TensorFlow
with tf.Session() as sess:
    # Định nghĩa giá trị cho x1_data và x2_data, mỗi ma trận có 3 cột
    x1_data = [[1, 2, 3]]
    x2_data = [[1, 2, 3]]

    # Chạy bước khởi tạo để khởi tạo tất cả các biến
    sess.run(init)

    # Thực hiện tính toán z bằng cách cung cấp giá trị cho các placeholder x1 và x2 thông qua feed_dict
    result = sess.run(z, feed_dict={x1: x1_data, x2: x2_data})

    # In kết quả ra màn hình
    print(result)
```

```
[[0.785835  0.85814893 0.9801597 ]]
```

3.4.4.

```
# 3.4.4

import tensorflow as tf
import numpy as np

# Tạo hai ma trận numpy matrix1 và matrix2 với kiểu dữ liệu int32
matrix1 = np.array([(2, 2, 2), (2, 2, 2), (2, 2, 2)], dtype='int32')
matrix2 = np.array([(1, 1, 1), (1, 1, 1), (1, 1, 1)], dtype='int32')

# In ma trận matrix1 và matrix2
print(matrix1)
print(matrix2)

# Chuyển matrix1 và matrix2 thành các hằng số TensorFlow
matrix1 = tf.constant(matrix1)
matrix2 = tf.constant(matrix2)

# Tính tích ma trận giữa matrix1 và matrix2
matrix_product = tf.matmul(matrix1, matrix2)

# Tính tổng của matrix1 và matrix2
matrix_sum = tf.add(matrix1, matrix2)

# Tạo một ma trận numpy matrix_3 với kiểu dữ liệu float32
matrix_3 = np.array([(2, 7, 2), (1, 4, 2), (9, 0, 2)], dtype='float32')

# In ma trận matrix_3
print(matrix_3)

# Tính định thức của matrix_3
matrix_det = tf.compat.v1.matrix_determinant(matrix_3)

# Bắt đầu một phiên làm việc với TensorFlow
with tf.compat.v1.Session() as sess:
    # Thực hiện tính toán matrix_product, matrix_sum, và matrix_det
    result1 = sess.run(matrix_product)
    result2 = sess.run(matrix_sum)
    result3 = sess.run(matrix_det)

# In kết quả của các tính toán
print(result1)
print(result2)
print(result3)
```

```

[[2 2 2]
 [2 2 2]
 [2 2 2]]
[[1 1 1]
 [1 1 1]
 [1 1 1]]
[[2. 7. 2.]
 [1. 4. 2.]
 [9. 0. 2.]]
[[6 6 6]
 [6 6 6]
 [6 6 6]]
[[3 3 3]
 [3 3 3]
 [3 3 3]]
55.999992

```

3.5.

```

1  # importing the dependencies
2  import tensorflow.compat.v1 as tf
3  import numpy as np
4  import matplotlib.pyplot as plt
5  tf.compat.v1.disable_eager_execution()
6
7  # Model Parameters
8  learning_rate = 0.01
9  training_epochs = 2000
10 display_step = 200

```

- Các dòng này import các thư viện cần thiết và định nghĩa các tham số của mô hình:

+ **learning_rate**: Tốc độ học (learning rate) cho thuật toán gradient descent. Nó quyết định khoảng cách chúng ta di chuyển trong mỗi bước cập nhật.

+ **training_epochs**: Số lần lặp qua toàn bộ tập huấn luyện.

+ **display_step**: Khoảng cách giữa các epoch để hiển thị thông tin tiến trình.

```

# Training Data
train_X = np.asarray([3.3,4.4,5.5,6.71,6.93,4.168,9.779,6.182,7.59,2.167,7.042,10.791,5.313,7.997,5.654,9.27,3.1])
train_y = np.asarray([1.7,2.76,2.09,3.19,1.694,1.573,3.366,2.596,2.53,1.221,2.827,3.465,1.65,2.904,2.42,2.94,1.3])
n_samples = train_X.shape[0]

```

+ **train_X** là biến chứa dữ liệu đầu vào của tập huấn luyện. Nó là một mảng NumPy chứa các giá trị đầu vào, được đưa vào mô hình để huấn luyện.

+ **train_y** là biến chứa giá trị mục tiêu tương ứng với dữ liệu đầu vào trong tập huấn luyện. Đây là các giá trị thực tế mà mô hình sẽ cố gắng dự đoán sau quá trình huấn luyện.

+ `np.asarray()` là một hàm trong thư viện NumPy để chuyển đổi danh sách hoặc mảng Python thành mảng NumPy. Trong trường hợp này, các danh sách giá trị đầu vào và giá trị mục tiêu được chuyển đổi thành mảng NumPy để có thể sử dụng trong các phép toán số học và trong TensorFlow.

+ `n_samples` là biến lưu trữ số lượng mẫu (dòng) trong tập dữ liệu huấn luyện, để biết kích thước của tập dữ liệu huấn luyện.

```
# Test Data
test_x = np.asarray([6.83, 4.668, 8.9, 7.91, 5.7, 8.7, 3.1, 2.1])

# Set placeholders for feature and target vectors
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
```

+ `test_X` là biến chứa dữ liệu đầu vào của tập kiểm tra. Tương tự như tập huấn luyện, `test_X` là một mảng NumPy chứa các giá trị đầu vào của các mẫu kiểm tra.

+ Tiếp theo, được định nghĩa hai **placeholders** trong TensorFlow:

- `x` là placeholder cho dữ liệu đầu vào. Placeholder này sẽ được sử dụng để cung cấp giá trị đầu vào cho mô hình khi thực hiện dự đoán trên dữ liệu kiểm tra.

- `y` là placeholder cho giá trị mục tiêu tương ứng với dữ liệu đầu vào. Placeholder này sẽ được sử dụng để cung cấp giá trị thực tế mà mô hình sẽ so sánh với các dự đoán để tính toán sai số trên dữ liệu kiểm tra.

```
24 # Set model weights and bias)
25 test_y = np.asarray([1.84, 2.273, 3.2, 2.831, 2.92, 3.24, 1.35, 1.03])
26 W = tf.Variable(np.random.randn(), name="weight")
27 b = tf.Variable(np.random.randn(), name="bias")
28
29 # Construct a linear model
30 linear_model = W*x + b
31
32 # Mean squared error
33 cost = tf.reduce_sum(tf.square(linear_model - y)) / (2*n_samples)
34
35 # Gradient descent
36 optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
37
38 # Initializing the variables
39 init = tf.global_variables_initializer()
40
```

+ `test_y` là biến chứa giá trị mục tiêu tương ứng với dữ liệu kiểm tra.

+ `W` là biến TensorFlow đại diện cho trọng số (weight) của mô hình.

+ ``b`` là biến TensorFlow đại diện cho sai số (bias) của mô hình.

Trọng số và Sai số này là biến ngẫu nhiên ban đầu và sẽ được cập nhật trong quá trình huấn luyện.

+ ``linear_model`` là biểu diễn của mô hình hồi quy tuyến tính. Nó được tính bằng cách nhân trọng số ``W`` với dữ liệu đầu vào ``X`` và sau đó cộng thêm sai số ``b``. Tạo ra một dự đoán tuyến tính.

+ ``cost`` là hàm mục tiêu (cost function) của mô hình. Trong trường hợp này, hàm cost là Mean Squared Error (MSE), là sai khác bình phương trung bình giữa giá trị dự đoán ``linear_model`` và giá trị thực tế ``y`` chia cho đôi ($/(2 * n_samples)$).

+ ``optimizer`` là bước tối ưu hóa của mô hình. Nó sử dụng thuật toán Gradient Descent để tối thiểu hóa hàm cost bằng cách điều chỉnh trọng số ``W`` và sai số ``b``.

+ ``init`` là bước khởi tạo tất cả các biến trong TensorFlow. Sau khi định nghĩa mô hình và thuật toán tối ưu hóa, cần khởi tạo tất cả các biến trước khi có thể sử dụng chúng.

```
# Launch the graph
with tf.Session() as sess:
    # Load initialized variables in current session
    sess.run(init)
    # Fit all training data
    for epoch in range(training_epochs):
        # perform gradient descent step
        sess.run(optimizer, feed_dict={X: train_X, y: train_y})
        # Display logs per epoch step
        if (epoch+1) % display_step == 0:
            c = sess.run(cost, feed_dict={X: train_X, y: train_y})
            print("Epoch:{0:6} \t Cost:{1:10.4} \t W:{2:6.4} \t b:{3:6.4}".format(epoch+1, c, sess.run(W), sess.run(b)))
```

+ ``with tf.Session() as sess:``: Đoạn mã bắt đầu một phiên TensorFlow mới sử dụng ``tf.Session()``. Việc sử dụng ``with`` đảm bảo rằng phiên TensorFlow sẽ tự động được đóng sau khi hoàn thành.

+ ``sess.run(init)``: Dòng này chạy bước khởi tạo tất cả các biến trong mô hình bằng cách sử dụng biến ``init`` đã được định nghĩa trước đó. Đảm bảo rằng tất cả các biến trong mô hình đã được khởi tạo với các giá trị ban đầu.

+ Vòng lặp ``for epoch in range(training_epochs):`` bắt đầu quá trình huấn luyện và lặp qua các epoch. ``training_epochs`` đã được định nghĩa trước đó là số lượng epoch cần thực hiện.

+ ``sess.run(optimizer, feed_dict={X: train_X, y: train_y})``: Dòng này thực hiện một bước tối ưu hóa bằng cách chạy optimizer đã được định nghĩa trước đó. Các giá trị đầu vào ``X`` và ``y`` được cung cấp thông qua ``feed_dict``. Điều này đẩy dữ liệu

huấn luyện vào mô hình và cập nhật trọng số và sai số bằng thuật toán Gradient Descent.

+ `if (epoch+1) % display_step == 0`: Kiểm tra xem có đến lượt hiển thị thông tin tiến trình sau mỗi số epoch là `display_step` hay không.

+ `c = sess.run(cost, feed_dict={X: train_X, y: train_y})`: Dòng này tính toán giá trị hàm `cost` trên dữ liệu huấn luyện hiện tại bằng cách chạy biến `cost` và sử dụng `feed_dict` để cung cấp dữ liệu đầu vào và giá trị thực tế. Giá trị này sau đó được lưu vào biến `c`.

+ `print('Epoch:{0:6}\tCost:{1:10.4}\tW:{2:6.4}\tb:{3:6.4}'.format(epoch+1,c, sess.run(W), sess.run(b)))`: Dòng này in ra thông tin về epoch hiện tại, giá trị `cost`, trọng số `W`, và sai số `b`. Điều này giúp theo dõi tiến trình huấn luyện và xem làm thế nào mô hình đang học.

```
53
54     # Print final parameter values
55     print("Optimization Finished!")
56     training_cost = sess.run(cost, feed_dict={X: train_X, y: train_y})
57     print("Final training cost:", training_cost, "W:",sess.run(W), "b:",sess.run(b), '\n')
58     # Graphic display
59     plt.plot(train_X, train_y, 'ro', label='Original data')
60     plt.plot(train_X, sess.run(W) * train_X + sess.run(b),
61              label='Fitted line')
62     plt.legend()
63     plt.show()
64
```

+ `print("Optimization Finished!")`: In ra thông báo cho biết quá trình tối ưu hóa đã hoàn thành, tức là mô hình đã được huấn luyện xong.

+ `training_cost = sess.run(cost, feed_dict={X: train_X, y: train_y})`: Tính toán giá trị hàm `cost` cuối cùng trên dữ liệu huấn luyện đã được huấn luyện bằng cách chạy biến `cost` và sử dụng `feed_dict` để cung cấp dữ liệu đầu vào và giá trị thực tế.

+ `print('Final training cost:', training_cost, "W:",sess.run(W), "b:",sess.run(b), '\n')`: In ra thông tin về giá trị hàm `cost` cuối cùng trên dữ liệu huấn luyện, giá trị trọng số `W`, và giá trị sai số `b` của mô hình sau khi huấn luyện xong. Giúp bạn đánh giá hiệu suất cuối cùng của mô hình.

+ Tiếp theo sử dụng thư viện Matplotlib để tạo biểu đồ:

- `plt.plot(train_X, train_y, 'ro', label='Original data')`: Đây là biểu đồ dữ liệu huấn luyện ban đầu, trong đó các điểm dữ liệu được biểu diễn bằng dấu "ro" (red circle).

- `plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')`: Đây là đường thẳng mô hình hồi quy tuyến tính sau khi huấn luyện. Nó được tính bằng cách nhân trọng số 'W' với dữ liệu huấn luyện và cộng thêm sai số 'b'.

- `plt.legend()`: Thêm chú thích vào biểu đồ để hiển thị rõ ràng các dấu hiệu "Original data" và "Fitted line".

- `plt.show()`: Hiển thị biểu đồ lên màn hình.

```
65 # Testing the model
66 testing_cost = sess.run(tf.reduce_sum(tf.square(linear_model - y)) / (2 * test_X.shape[0]), feed_dict={X: test_X, y: test_y})
67 print("Final testing cost:", testing_cost)
68 print("Absolute mean square loss difference:",
69       abs(training_cost - testing_cost))
70 # Display fitted line on test data
71 plt.plot(test_X, test_y, 'bo', label='Testing data')
72 plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
73 plt.legend()
74 plt.show()
```

+ `testing_cost = sess.run(tf.reduce_sum(tf.square(linear_model - y)) / (2 * test_X.shape[0]), feed_dict={X: test_X, y: test_y})`: Tính toán giá trị hàm cost trên dữ liệu kiểm tra bằng cách chạy biểu thức `tf.reduce_sum(tf.square(linear_model - y)) / (2 * test_X.shape[0])`. Tương tự như việc tính cost trên dữ liệu huấn luyện, nhưng áp dụng cho dữ liệu kiểm tra. Giá trị cost này được lưu vào biến `testing_cost`.

+ `print("Final testing cost:", testing_cost)`: In ra giá trị cost cuối cùng trên dữ liệu kiểm tra. Giúp đánh giá hiệu suất của mô hình trên dữ liệu mới, không được sử dụng trong quá trình huấn luyện.

+ `print("Absolute mean square loss difference:", abs(training_cost - testing_cost))`: Tính và in ra sự khác biệt tuyệt đối giữa cost trên dữ liệu huấn luyện và cost trên dữ liệu kiểm tra. Giúp bạn xác định xem mô hình có bị quá khớp (overfitting) hay không. Nếu sự khác biệt lớn, có thể xem xét điều chỉnh mô hình để tránh overfitting.

+ Tiếp theo sử dụng Matplotlib để hiển thị biểu đồ:

- `plt.plot(test_X, test_y, 'bo', label='Testing data')`: Đây là biểu đồ dữ liệu kiểm tra, trong đó các điểm dữ liệu được biểu diễn bằng dấu "bo" (blue circle).

- `plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')`: Đây là đường thẳng mô hình hồi quy tuyến tính đã được huấn luyện trên dữ liệu

huấn luyện. Điều này được thêm vào biểu đồ để so sánh dự đoán của mô hình trên dữ liệu kiểm tra.

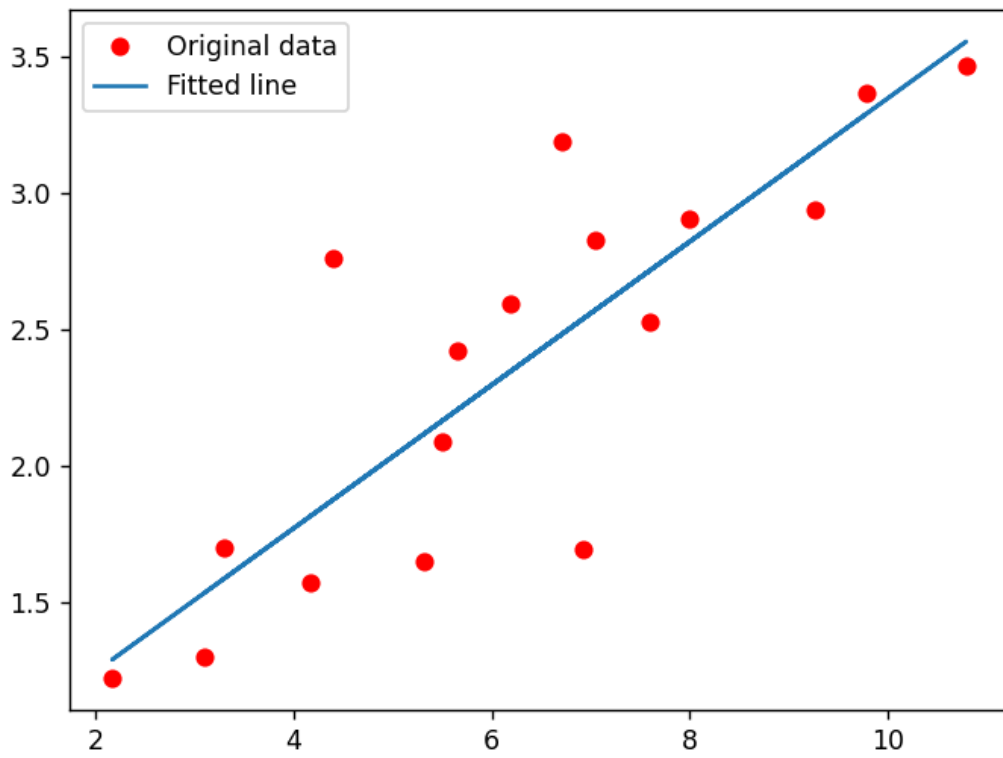
- `plt.legend()`: Thêm chú thích vào biểu đồ để hiển thị rõ ràng các dấu hiệu "Testing data" và "Fitted line".

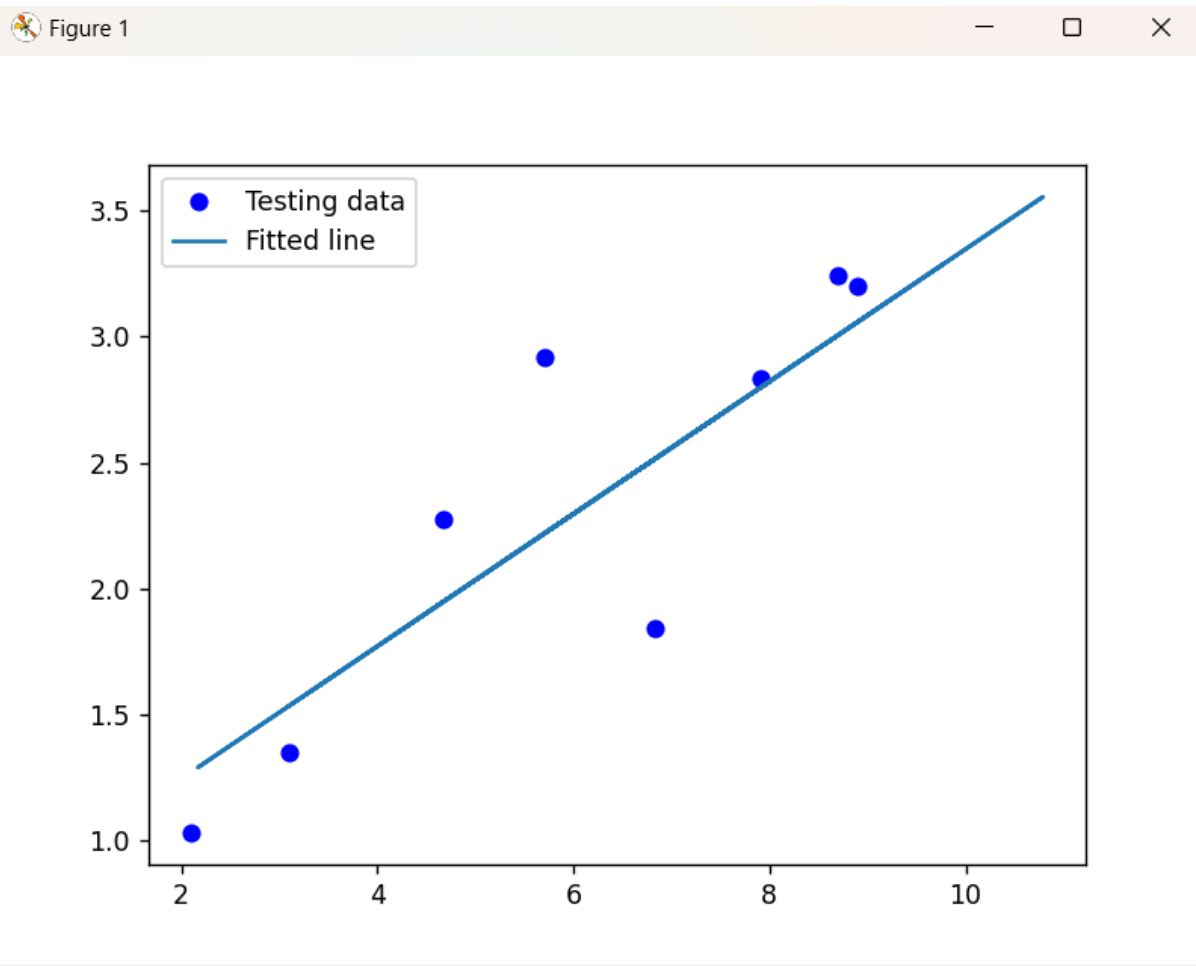
- `plt.show()`: Hiển thị biểu đồ lên màn hình.

-Chạy mô hình:

```
Epoch: 200    Cost: 0.1078    W:0.3512    b:0.09295
Epoch: 400    Cost: 0.0959    W:0.3297    b:0.2452
Epoch: 600    Cost: 0.0886    W:0.3129    b:0.3646
Epoch: 800    Cost: 0.08411   W:0.2997    b:0.4582
Epoch: 1000   Cost: 0.08135   W:0.2893    b:0.5317
Epoch: 1200   Cost: 0.07965   W:0.2812    b:0.5893
Epoch: 1400   Cost: 0.0786    W:0.2748    b:0.6345
Epoch: 1600   Cost: 0.07796   W:0.2698    b:0.6699
Epoch: 1800   Cost: 0.07756   W:0.2659    b:0.6977
Epoch: 2000   Cost: 0.07732   W:0.2628    b:0.7195
Optimization Finished!
Final training cost: 0.07731805 W: 0.26281983 b: 0.7195054
```

Figure 1





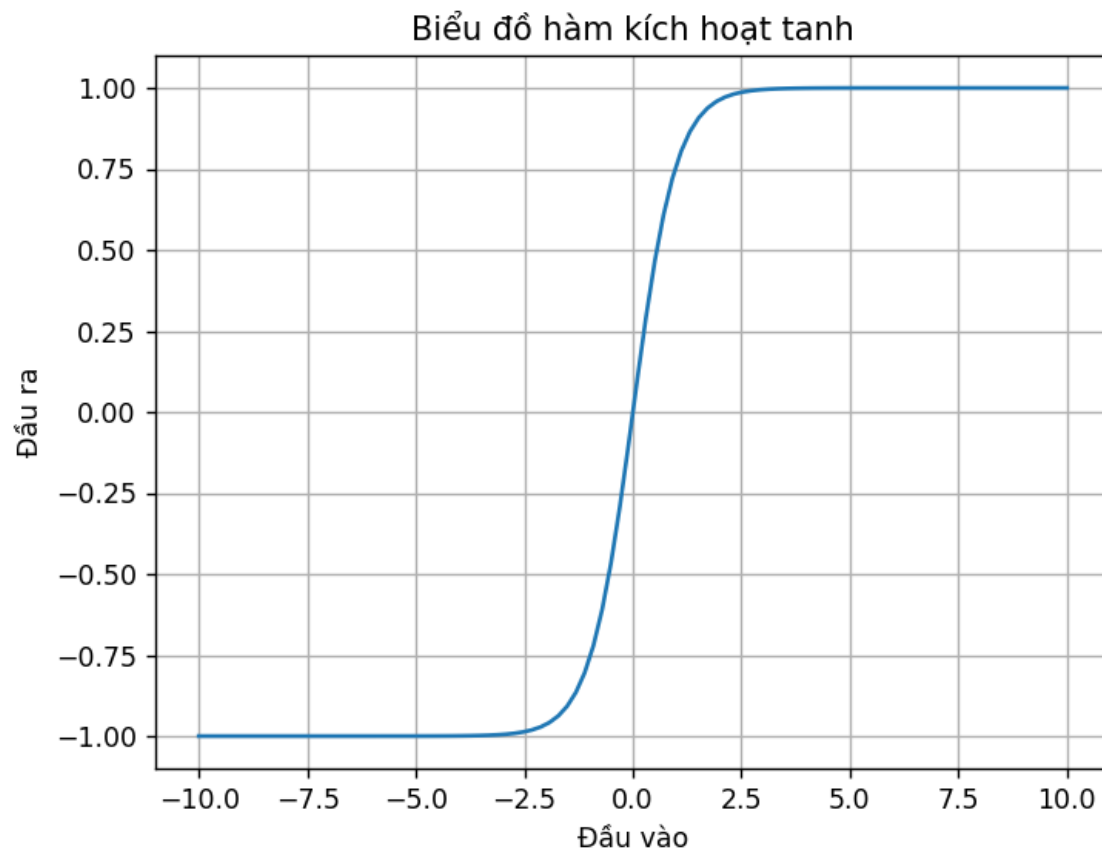
3.6.

```
1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3
4 # Khởi tạo đầu vào và trọng số
5 X = tf.constant([2.0, 3.0, 4.0], dtype=tf.float32)
6 W = tf.constant([0.05, 0.07, 0.09], dtype=tf.float32)
7
8 # Khởi tạo sai số
9 b = tf.constant(0.6, dtype=tf.float32)
10
11 # Tính toán tổng đầu vào đã được nhân với trọng số, sau đó cộng với sai số
12 z = tf.reduce_sum(tf.multiply(X, W)) + b
13
14 # Sử dụng hàm kích hoạt tanh
15 output = tf.nn.tanh(z)
16
17 # In kết quả
18 print("Output:", output.numpy())
19 # Vẽ biểu đồ cho hàm kích hoạt tanh
20 x = tf.linspace(-10.0, 10.0, 100) # Tạo một dãy giá trị đầu vào từ -10 đến 10
21 y = tf.nn.tanh(x) # Tính giá trị đầu ra tương ứng với hàm kích hoạt tanh
22
23 plt.plot(x, y)
24 plt.xlabel("Đầu vào")
25 plt.ylabel("Đầu ra")
26 plt.title("Biểu đồ hàm kích hoạt tanh")
27 plt.grid(True)
28 plt.show()
29
```

Kết

quả:

To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Output: 0.8537976



3.7.

```
bai37.py > ...
1 # Import các thư viện cần thiết
2 from keras.models import Sequential
3 from keras.layers import Dense
4 import numpy as np
5
6 # Chuẩn bị dữ liệu
7 # Tạo dữ liệu giả lập huấn luyện cho 1000 học sinh và dữ liệu giả lập kiểm tra cho 500 học sinh
8 # Các cột: Tuổi, Số giờ học & Điểm trung bình các kỳ thi trước đó
9 np.random.seed(2018) # Đặt hạt giống để tái tạo (reproducibility)
10 train_data, test_data = np.random.random((1000, 3)), np.random.random((500, 3))
11 # Tạo kết quả giả cho 1000 học sinh: Passed (1) or Failed (0)
12 labels = np.random.randint(2, size=(1000, 1))
13 # Định nghĩa cấu trúc mô hình với các lớp cần thiết, số nơ-ron, hàm kích hoạt và thuật toán tối ưu
14 model = Sequential()
15 model.add(Dense(5, input_dim=3, activation="relu"))
16 model.add(Dense(4, activation="relu"))
17 model.add(Dense(1, activation="sigmoid"))
18 model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
19 # Huấn luyện mô hình và thực hiện dự đoán
20 model.fit(train_data, labels, epochs=10, batch_size=32)
21 # Thực hiện dự đoán từ mô hình đã được huấn luyện
22 predictions = model.predict(test_data)
23
```


Đầu tiên `np.random.seed(2018)` là để đặt hạt giống (seed) cho bộ sinh số ngẫu nhiên trong thư viện NumPy. Điều này có tác dụng làm cho việc sinh các số ngẫu nhiên trở nên dự đoán được và tái tạo kết quả. Việc đặt hạt giống này có ý nghĩa khi bạn muốn có khả năng tái tạo các kết quả hoặc khi bạn muốn đảm bảo rằng mọi người sử dụng mã của bạn sẽ có cùng một kết quả khi họ chạy mã của bạn.

Dòng tiếp theo tạo dữ liệu huấn luyện và kiểm tra giả lập sử dụng `np.random.random`.

- `np.random.random((1000, 3))` tạo ra một ma trận có kích thước (1000, 3), nghĩa là có 1000 dòng và 3 cột, và các giá trị trong ma trận này là ngẫu nhiên từ phân phối đều (uniform distribution) trong khoảng từ 0 đến 1. Đây là dữ liệu giả lập cho huấn luyện.

- `np.random.random((500, 3))` tạo ra một ma trận tương tự nhưng với kích thước (500, 3), đại diện cho dữ liệu kiểm tra.

Tổng cộng, chúng ta có 1000 mẫu huấn luyện và 500 mẫu kiểm tra, mỗi mẫu có 3 đặc trưng (cột).

Tạo ra ma trận nhãn (`labels`) cho dữ liệu huấn luyện. Nhãn là một chỉ số ngẫu nhiên, 0 hoặc 1, để biểu thị liệu mỗi mẫu trong dữ liệu huấn luyện đã đỗ hoặc trượt.

- `np.random.randint(2, size=(1000, 1))` tạo một ma trận có kích thước (1000, 1), có nghĩa là có 1000 dòng và 1 cột. Hàm `np.random.randint(2, size=(1000, 1))` tạo ngẫu nhiên các số nguyên từ 0 đến 1 (bao gồm cả 0 và 1) để đại diện cho nhãn.

Trong đó, mỗi số nguyên 0 hoặc 1 biểu thị liệu mỗi mẫu trong dữ liệu huấn luyện đã trượt (0) hoặc đỗ (1). Tạo nhãn ngẫu nhiên như vậy làm cho dữ liệu huấn luyện có tính ngẫu nhiên và không chịu ảnh hưởng từ bất kỳ quy tắc hay mối quan hệ nào giữa các đặc trưng và nhãn thực tế.

- `model = Sequential()`: Đây là cách khởi tạo một mô hình mạng nơ-ron tuần tự (Sequential model) trong Keras. Mô hình này sẽ bao gồm một chuỗi các lớp liên tiếp, từ lớp đầu tiên đến lớp cuối cùng.

- `model.add(Dense(5, input_dim=3, activation="relu"))`: Đây là lớp đầu tiên của mô hình. Mô hình bắt đầu bằng một lớp kết nối đầy đủ (fully connected layer) có 5 nơ-ron.

- `input_dim=3` xác định số lượng đặc trưng đầu vào, trong trường hợp này là 3 (Age, Hours of Study, và Avg Previous Test Scores).

- ``activation="relu"``` xác định hàm kích hoạt là Rectified Linear Activation (ReLU), một hàm kích hoạt phổ biến trong mạng nơ-ron.

- ``model.add(Dense(4, activation="relu"))``: Đây là lớp thứ hai của mô hình. Nó cũng là một lớp kết nối đầy đủ với 4 nơ-ron và sử dụng hàm kích hoạt ReLU.

- ``model.add(Dense(1, activation="sigmoid"))``: Lớp cuối cùng của mô hình có 1 nơ-ron và sử dụng hàm kích hoạt sigmoid. Lớp này thường được sử dụng trong bài toán phân loại nhị phân, vì nó biến đổi đầu ra thành một giá trị trong khoảng từ 0 đến 1, tương ứng với xác suất thuộc vào lớp tích cực (positive class).

- ``model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"]``: Ở đây, bạn đang biên dịch (compile) mô hình:

- ``loss="binary_crossentropy"```: Hàm mất mát được sử dụng cho bài toán phân loại nhị phân. Trong trường hợp này, nó là hàm mất mát nhị phân (binary cross-entropy).

- ``optimizer="adam"```: Thuật toán tối ưu hóa được sử dụng để điều chỉnh trọng số của mạng. Adam là một trong những thuật toán tối ưu hóa phổ biến trong deep learning.

- ``metrics=["accuracy"]``: Đây là các chỉ số đánh giá mô hình, trong trường hợp này, chỉ số độ chính xác (accuracy) được sử dụng để đo hiệu suất của mô hình.

- ``model.fit(train_data, labels, epochs=10, batch_size=32)``: Đây là bước huấn luyện mô hình sử dụng dữ liệu huấn luyện (``train_data``) và nhãn tương ứng (``labels``).

- ``train_data``: Là dữ liệu huấn luyện, bao gồm thông tin về tuổi, số giờ học và điểm trung bình của các kỳ thi trước đó.

- ``labels``: Là nhãn cho dữ liệu huấn luyện, biểu thị xem mỗi mẫu đã đỗ hoặc trượt (1 hoặc 0).

- ``epochs=10``: Đây là số lần mô hình sẽ đi qua toàn bộ dữ liệu huấn luyện trong quá trình huấn luyện. Mỗi lượt qua toàn bộ dữ liệu được gọi là một epoch.

- ``batch_size=32``: Đây là số lượng mẫu dữ liệu sẽ được sử dụng trong mỗi lần cập nhật trọng số mô hình. Điều này giúp tăng tốc quá trình huấn luyện.

- ``predictions = model.predict(test_data)``: Sau khi mô hình đã được huấn luyện, bạn sử dụng mô hình này để đưa ra dự đoán trên dữ liệu kiểm tra (``test_data``). Dự

đoán được lưu vào biến `predictions`, và nó biểu thị xác suất mỗi mẫu trong dữ liệu kiểm tra thuộc vào lớp tích cực (positive class), trong trường hợp này, lớp đã đổ.

Kết quả:

```
Epoch 1/10
32/32 [=====] - 1s 2ms/step - loss: 0.6935 - accuracy: 0.4990
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 0.6929 - accuracy: 0.5190
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 0.6925 - accuracy: 0.5270
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 0.6925 - accuracy: 0.5170
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.6922 - accuracy: 0.5370
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.6923 - accuracy: 0.5350
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.6922 - accuracy: 0.5280
Epoch 8/10
32/32 [=====] - 0s 1ms/step - loss: 0.6922 - accuracy: 0.5320
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.6921 - accuracy: 0.5310
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.6922 - accuracy: 0.5400
16/16 [=====] - 0s 1ms/step
```