Họ và tên: Quách Xuân Phúc

MSV: B20DCCN513

3.1

```
import tensorflow as tf

# creating nodes in computation graph
node1 = tf.constant(3, dtype=tf.int32)
node2 = tf.constant(5, dtype=tf.int32)
node3 = tf.add(node1, node2)

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

```
sum of node1 and node2 is : 8
```

* Giải thích:

Biến node1 là một hằng số với giá trị là 3 và kiểu dữ liệu là int32, nghĩa là nó sẽ lưu trữ một số nguyên 32-bit.

Biến node2 là một hằng số với giá trị là 5 và kiểu dữ liệu là int32

```
import tensorflow.compat.v1 as tf
x = tf.constant(5,tf.float32)
y = tf.constant([5], tf.float32)
z = tf.constant([5,3,4], tf.float32)
t = tf.constant([[5,3,4,6],[2,3,4,7]], tf.float32)
u = tf.constant([[[5,3,4,6],[2,3,4,0]]], tf.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)
print(y)
```

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

* Giải thích:

Biến x là một hằng số với giá trị là 5 và kiểu dữ liệu là float32.

Biến y là một hằng số với giá trị là [5] (một mảng) và kiểu dữ liệu là float32.

Biến z là một hằng số với giá trị là [5, 3, 4] (một mảng) và kiểu dữ liệu là float32.

Biến t là một hằng số với giá trị là một ma trận 2x4 và kiểu dữ liệu là float32.

Biến u là một hằng số với giá trị là một ma trận 1x2x4 và kiểu dữ liệu là float32.

Biến v là một hằng số với giá trị là một ma trận 3x2x4 và kiểu dữ liệu là float32.

Mỗi biến hằng số này có giá trị được xác định trước và không thay đổi trong quá trình tính toán. Chúng thường được sử dụng để đại diện cho dữ liệu đầu vào hoặc các tham số không thay đổi trong mạng neural network và các phép tính TensorFlow khác.

3.2

```
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

x1 = tf.Variable(5.3, tf.float32)
x2 = tf.Variable(4.3, tf.float32)
x = tf.multiply(x1,x2)

init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    t = sess.run(x)
    print(t)
```

```
22.79
```

* Giải thích:

Hàm Variable dùng để khởi tạo biến trong tensorflow.

x1 là biến có giá trị 5.3 và có kiểu dữ liệu là float32, nghĩa là biến này sẽ lưu trữ số thực 32-bit.

x2 là biến có giá trị 4.3 và có kiểu dữ liệu là float32.

```
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

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)
x = tf.multiply(x1,x2)

init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    t = sess.run(x)
    print(t)
```

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

* Giải thích:

x1 là biến có giá trị là một ma trận 2 chiều với các giá trị số thực và kiểu dữ liệu của biến là float32, nghĩa là biến này sẽ lưu trữ số thực 32-bit.

x2 là biến có giá trị là một ma trận 2 chiều với các giá trị số thực và kiểu dữ liệu của biến là float32.

```
import tensorflow.compat.v1 as tf

# creating nodes in computation graph
node = tf.Variable(tf.zeros([2,2]))

# running computation graph
with tf.Session() as sess:
    # initialize all global variables
    sess.run(tf.global_variables_initializer())
    # evaluating node
    print("Tensor value before addition:\n",sess.run(node))
    # elementwise addition to tensor
    node = node.assign(node + tf.ones([2,2]))
    # evaluate node again
    print("Tensor value after addition:\n", sess.run(node))
    sess.close()
```

```
Tensor value before addition:
 [[0. 0.]
 [0. 0.]]
Tensor value after addition:
 [[1. 1.]
 [1. 1.]]
```

* Giải thích:

node là biến được khởi tạo bởi hàm Variable và có giá trị là một ma trận 2x2 với tất cả các phần tử được đặt bằng 0.0

Khi dùng tf.Variable để khởi tạo, các đối tượng có thể thay đổi giá trị của chúng trong quá trình tính toán. Điều này rất hữu ích khi muốn lưu trữ và cập nhật các tham số của mô hình trong quá trình huấn luyện.

3.3

Trong TensorFlow, hàm tf.placeholder được sử dụng để tạo các "điểm gắn" (placeholders) trong đồ thị tính toán. Điểm gắn là các đối tượng mà bạn có thể sử dụng để truyền dữ liệu vào trong đồ thị tính toán trong quá trình chạy một phiên TensorFlow.

```
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

x = tf.placeholder(tf.float32,None)
y = tf.add(x,x)

with tf.Session() as sess:
    x_data = 5
    result = sess.run(y,feed_dict={x:x_data})
    print(result)
```

```
10.0
```

* Giải thích:

x là điểm gắn được khởi tạo bởi hàm placeholder. Bạn đã chỉ định kiểu dữ liệu của điểm gắn là tf.float32, tức là nó sẽ truyền dữ liệu dạng số thực 32-bit. None được sử dụng cho kích thước của tensor đầu vào, có nghĩa là tensor có thể có bất kỳ kích thước nào cho mỗi chiều. Điều này cho phép bạn truyền vào tensor có kích thước linh hoạt.

```
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()\

x = tf.placeholder(tf.float32,[None,3])
y = tf.add(x,x)

with tf.Session() as sess:
    x_data= [[1.5, 2.0, 3.3]]
    result = sess.run(y,feed_dict={x:x_data})
    print(result)
```

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

* Giải thích:

Trong đoạn mã này, x là một điểm gắn với kiểu dữ liệu tf.float32. Trong phần kích thước của tensor đầu vào, bạn đã chỉ định [None, 3], có nghĩa rằng tensor có thể có số hàng linh hoạt (không xác định trước) nhưng mỗi hàng sẽ có đúng 3 phần tử. Điều này cho phép bạn truyền vào một danh sách các vectơ dữ liệu có 3 phần tử mỗi vectơ, nhưng số lượng vectơ không cố định.

```
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

x = tf.placeholder(tf.float32,[None,None,3])
y = tf.add(x,x)

with tf.Session() as sess:
    x_data= [[[1,2,3]]]
    result = sess.run(y,feed_dict={x:x_data})
    print(result)
```

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

* Giải thích:

Trong đoạn mã này, x là một điểm gắn với kiểu dữ liệu tf.int32. Trong phần kích thước của tensor đầu vào, bạn đã chỉ định [None, None, 3], có nghĩa rằng tensor có thể có số hàng linh hoạt và số cột linh hoạt (không xác định trước), nhưng mỗi hàng sẽ có đúng 3 phần tử. Điều này cho phép bạn truyền vào một danh sách các ma trận có 3 cột mỗi hàng, nhưng số hàng và số cột không cố định.

```
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

x = tf.placeholder(tf.float32,[None,4,3])
y = tf.add(x,x)

with tf.Session() as sess:
    x_data= [[[1,2,3], [2,3,4], [2,3,5], [0,1,2]]]
    result = sess.run(y,feed_dict={x:x_data})
    print(result)
```

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

* Giải thích:

Trong đoạn mã này, x là một điểm gắn có kiểu dữ liệu tf.float32. Phần kích thước của tensor đầu vào là [None, 4, 3], có nghĩa là bạn có thể truyền vào một tensor với số hàng linh hoạt (không xác định trước), mỗi hàng sẽ có đúng 4 dòng và 3 cột. Điều này cho phép bạn truyền vào một danh sách các ma trận 4x3 có số hàng linh hoạt.

```python
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

x = tf.placeholder(tf.float32,[2,4,3])
y = tf.add(x,x)

with tf.Session() as sess:
    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]]]
    result = sess.run(y,feed_dict={x:x_data})
    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.]]]
```

* Giải thích:

Trong đoạn mã này, x là một điểm gắn có kiểu dữ liệu tf.float32. Phần kích thước của tensor đầu vào được xác định là [2, 4, 3], có nghĩa là bạn có thể truyền vào một tensor với 2 hàng, mỗi hàng có 4 dòng và 3 cột. Điều này đã xác định cụ thể kích thước của x.

```python
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

x = tf.placeholder(tf.float32,[2,4,3])
y = tf.placeholder(tf.float32,[2,4,3])
z = tf.add(x,y)
u = tf.multiply(x,y)

with tf.Session() as sess:
    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]]]
    result1 = sess.run(z,feed_dict={x:x_data, y:y_data})
    result2 = sess.run(u,feed_dict={x:x_data, y:y_data})
    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.]]]
```

* Giải thích:

Trong đoạn mã này, bạn đã tạo hai điểm gắn x và y có cùng kiểu dữ liệu tf.float32 và kích thước [2, 4, 3]. Điều này cho biết bạn đang truyền vào hai tensors có 2 hàng, mỗi hàng có 4 dòng và 3 cột.

3.4

```
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

x1 = tf.constant(5.3, tf.float32)
x2 = tf.constant(1.5, tf.float32)
w1 = tf.Variable(0.7, tf.float32)
w2 = tf.Variable(0.5, tf.float32)
u = tf.multiply(x1,w1)
v = tf.multiply(x2,w2)
z = tf.add(u,v)

result = tf.sigmoid(z)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    print(sess.run(result))
```

```
0.9885698
```

* Giải thích:

tf.constant: hàm tf.constant để tạo hai hằng số x1 và x2. x1 có giá trị 5.3 và kiểu dữ liệu tf.float32, còn x2 có giá trị 1.5 và kiểu dữ liệu tf.float32.

tf.Variable: hàm tf.Variable để tạo hai biến w1 và w2. w1 có giá trị khởi tạo là 0.7 và kiểu dữ liệu tf.float32, còn w2 có giá trị khởi tạo là 0.5 và kiểu dữ liệu tf.float32. Các biến này có thể được cập nhật trong quá trình tính toán.

tf.multiply: hàm tf.multiply để thực hiện phép nhân element-wise giữa x1 và w1, gán kết quả vào biến u, và phép nhân element-wise giữa x2 và w2, gán kết quả vào biến v.

tf.add: hàm tf.add để thực hiện phép cộng giữa u và v, gán kết quả vào biến z.

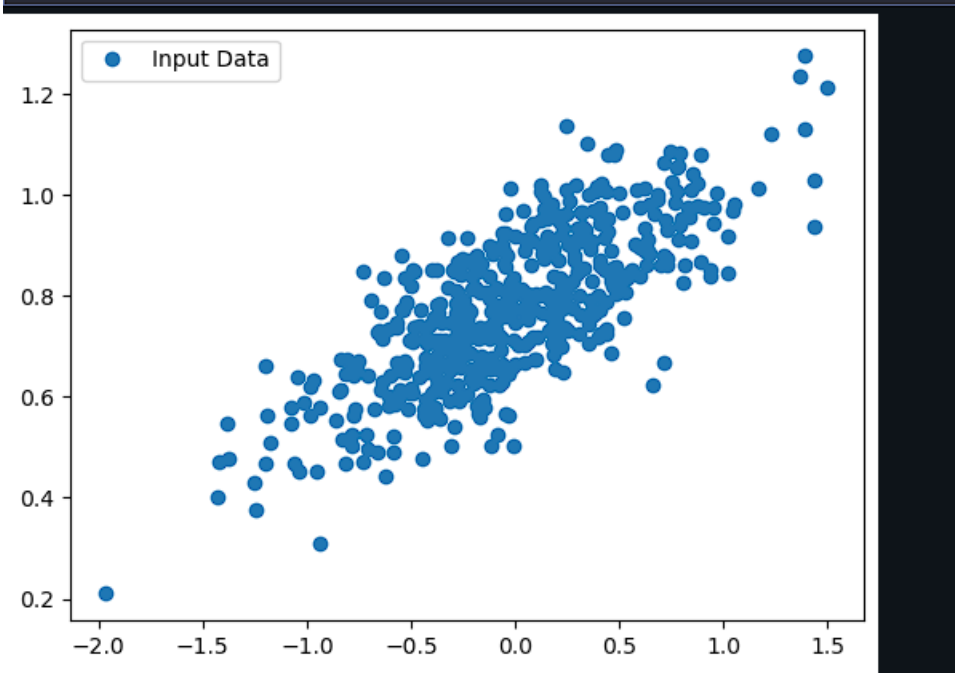
tf.sigmoid: hàm tf.sigmoid để tính giá trị sigmoid của z, gán kết quả vào biến result. Hàm sigmoid được sử dụng trong các mô hình neural network để chuyển đổi giá trị thành một giá trị nằm trong khoảng (0, 1), thường được sử dụng để biểu thị xác suất.

tf.global_variables_initializer: hàm này để khởi tạo tất cả các biến w1 và w2.

tf.Session: tạo một phiên TensorFlow sử dụng tf.Session() để thực hiện tính toán.

sess.run: sess.run để thực hiện tính toán giá trị của result sau khi đã khởi tạo tất cả các biến. Kết quả của phép tính result (giá trị sigmoid của z) được in ra màn hình.

```python
import numpy as np
import matplotlib.pyplot as plt

number_of_points = 500
x_point = []
y_point = []
a = 0.22
b = 0.78

for i in range(number_of_points):
    x = np.random.normal(0.0,0.5)
    y = a*x + b +np.random.normal(0.0,0.1)
    x_point.append([x])
    y_point.append([y])

plt.plot(x_point,y_point, 'o', label = 'Input Data')
plt.legend()
plt.show()
```



* Giải thích:

Tạo dữ liệu ngẫu nhiên: Mã này sử dụng thư viện NumPy để tạo một loạt điểm dữ liệu ngẫu nhiên x và tạo dữ liệu tương ứng y dựa trên phương trình y = ax + b với sự nhiễu ngẫu nhiên.

Vẽ biểu đồ: Mã này sử dụng Matplotlib để vẽ biểu đồ các điểm dữ liệu x và y trên một đồ thị.

Operation là các phép tính hoặc thao tác trong TensorFlow, chúng thực hiện các phép toán và tính toán giữa các tensors. Một số mục đích chính của việc sử dụng Operation trong các ví dụ này là:

- Xây dựng đồ thị tính toán: Các Operation được sử dụng để xây dựng một đồ thị tính toán trong TensorFlow. Đồ thị này biểu diễn mối quan hệ giữa các tensors và phép tính.
- Tính toán kết quả: Các Operation thực hiện các phép toán trên các tensors để tính toán kết quả mong muốn. Ví dụ: trong ví dụ đầu tiên, Operation tf.sigmoid(z) tính toán kết quả của hàm sigmoid dựa trên giá trị của z.
- Lưu trữ trạng thái: Biến được tạo bằng Operation là các biến TensorFlow, chúng lưu trữ các trạng thái mà mô hình có thể cập nhật trong quá trình huấn luyện.
- Cung cấp dữ liệu đầu vào: Placeholder là các Operation đặc biệt được sử dụng để cung cấp dữ liệu đầu vào cho mô hình. Chúng không chứa giá trị cụ thể mà thay vào đó được cung cấp thông qua feed_dict trong quá trình chạy session.
- Thực hiện phép nhân, cộng, sigmoid, và các phép toán khác: Các Operation có thể là bất kỳ phép toán nào bạn cần để thực hiện tính toán phức tạp. Ví dụ: phép nhân tf.multiply, phép cộng tf.add, hàm sigmoid tf.sigmoid, và nhiều phép toán khác có thể được sử dụng để xây dựng mô hình và tính toán.

```
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()

x1 = tf.placeholder(tf.float32,[None,3])
x2 = tf.placeholder(tf.float32,[None,3])
w1 = tf.Variable([0.5,0.4,0.7],tf.float32)
w2 = tf.Variable([0.8,0.5,0.6], tf.float32)
u1 = tf.multiply(w1,x1)
u2 = tf.multiply(w2,x2)
v = tf.add(u1,u2)
z = tf.sigmoid(v)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    x1_data= [[1,2,3]]
    x2_data= [[1,2,3]]
    sess.run(init)
    result = sess.run(z,feed_dict={x1:x1_data, x2:x2_data})
    print(result)
```

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

* Giải thích:

- import tensorflow.compat.v1 as tf và tf.compat.v1.disable_eager_execution(): Dòng đầu tiên là việc import TensorFlow và tắt tính năng eager execution trong TensorFlow (đối với phiên bản TensorFlow cũ).

- x1 = tf.placeholder(tf.float32, [None, 3]) và x2 = tf.placeholder(tf.float32, [None, 3]): Đây là hai placeholders, x1 và x2, được sử dụng để định nghĩa hai đầu vào dạng tensor. Mỗi tensor có kích thước [None, 3], trong đó None cho phép bạn truyền vào bất kỳ số lượng hàng nào, nhưng mỗi hàng phải có 3 phần tử. Điều này đặc tả rằng chúng ta sẽ truyền vào các dữ liệu với 3 đặc trưng (hoặc chiều) cho x1 và x2.

- w1 = tf.Variable([0.5, 0.4, 0.7], tf.float32) và w2 = tf.Variable([0.8, 0.5, 0.6], tf.float32): Đây là hai biến w1 và w2 được sử dụng để định nghĩa trọng số cho việc nhân element-wise với các đầu vào x1 và x2. Mỗi biến có một tensor 1D với 3 giá trị và kiểu dữ liệu tf.float32.

- u1 = tf.multiply(w1, x1) và u2 = tf.multiply(w2, x2): Đây là hai phép nhân element-wise giữa w1 và x1, và giữa w2 và x2. Các phép nhân này thực hiện nhân từng phần tử của các tensor tương ứng với nhau.

- v = tf.add(u1, u2): Đây là phép cộng giữa u1 và u2, tạo ra một tensor v là kết quả của phép cộng element-wise giữa u1 và u2.

- z = tf.sigmoid(v): Đây là phép tính giá trị sigmoid của v. Hàm sigmoid được sử dụng để biến đổi giá trị thành một giá trị nằm trong khoảng (0, 1). Kết quả của phép tính này được gán vào biến z.

- init = tf.global_variables_initializer(): Đây là hàm để khởi tạo tất cả các biến đã được định nghĩa (w1 và w2). Điều này cần thiết trước khi chúng ta có thể thực hiện tính toán.

- with tf.Session() as sess: Chúng ta bắt đầu một phiên TensorFlow để thực hiện các phép toán.

- x1_data = [[1, 2, 3]] và x2_data = [[1, 2, 3]]: Đây là các dữ liệu đầu vào cho x1 và x2. Mỗi tensor là một danh sách chứa một hàng với 3 phần tử.

- sess.run(init): Chúng ta chạy hàm khởi tạo biến init để khởi tạo các biến.

- result = sess.run(z, feed_dict={x1: x1_data, x2: x2_data}): Cuối cùng, chúng ta chạy phép tính giá trị của z bằng cách truyền dữ liệu vào các placeholders x1 và x2 thông qua feed_dict. Kết quả của phép tính này được gán vào biến result.

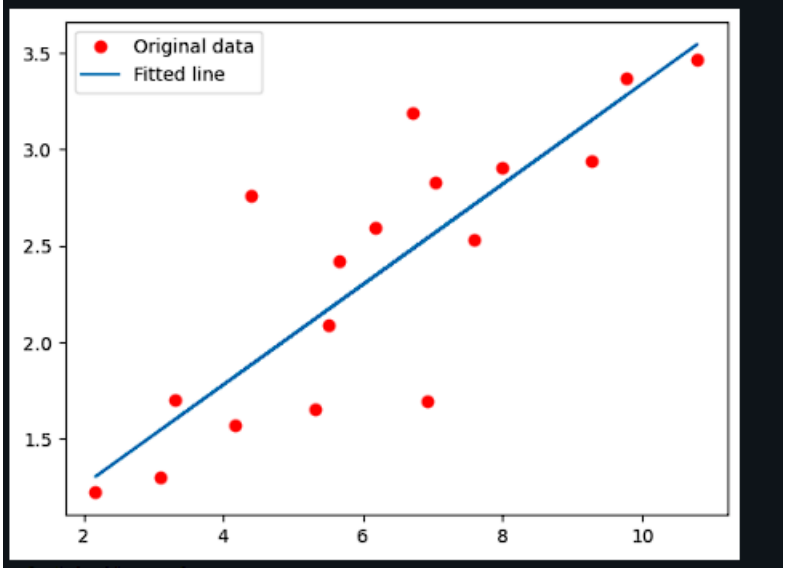
- print(result): Kết quả cuối cùng của phép tính z được in ra màn hình.

3.5

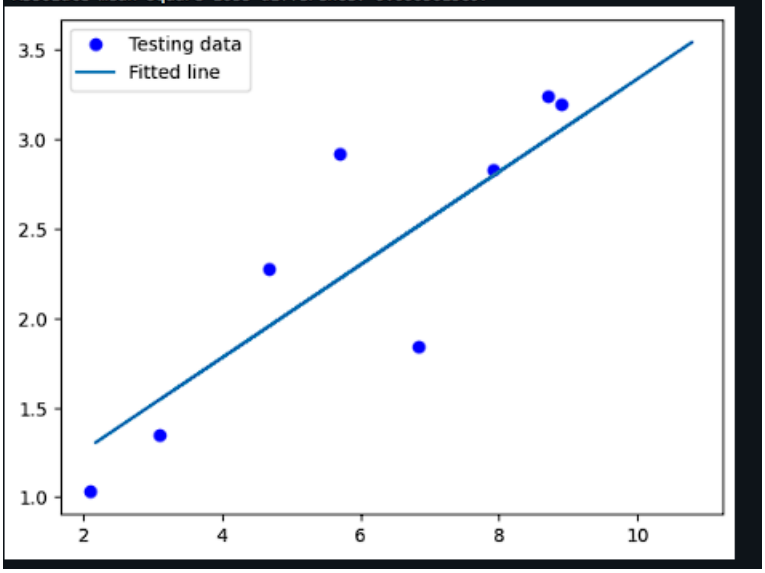
```
import tensorflow.compat.v1 as tf
import numpy as np
import matplotlib.pyplot as plt

# Disable eager execution
tf.compat.v1.disable_eager_execution()

# Model Parameters
learning_rate = 0.01
training_epochs = 2000
display_step = 200

# 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]

# 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.compat.v1.placeholder(tf.float32)
y = tf.compat.v1.placeholder(tf.float32)

# Set model weights and bias
test_y = np.asarray([1.84, 2.273, 3.2, 2.831, 2.92, 3.24, 1.35, 1.03])
W = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")

# Construct a linear model
linear_model = W*X + b

# Mean squared error
cost = tf.reduce_sum(tf.square(linear_model - y)) / (2*n_samples)

# Gradient descent
optimizer = tf.compat.v1.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Initializing the variables
init = tf.compat.v1.global_variables_initializer()
```

```
# Launch the graph
with tf.compat.v1.Session() as sess:
    # Load initialized variables in the 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)))

    # Print final parameter values
    print("Optimization Finished!")
    training_cost = sess.run(cost, feed_dict={X: train_X, y: train_y})
    print("Final training cost:", training_cost, "W:", sess.run(W), "b:", sess.run(b), '\n')

    # Graphic display
    plt.plot(train_X, train_y, 'ro', label='Original data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
    plt.legend()
    plt.show()

    # Testing the model
    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})
    print("Final testing cost:", testing_cost)
    print("Absolute mean square loss difference:", abs(training_cost - testing_cost))

    # Display fitted line on test data
    plt.plot(test_X, test_y, 'bo', label='Testing data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
    plt.legend()
    plt.show()
```

```
Epoch:   200     Cost:   0.09387         W:0.3254        b:0.2756
Epoch:   400     Cost:   0.08735         W:0.3095        b:0.3885
Epoch:   600     Cost:   0.08334         W: 0.297        b: 0.477
Epoch:   800     Cost:   0.08087         W:0.2872        b:0.5464
Epoch:  1000     Cost:   0.07936         W:0.2796        b:0.6008
Epoch:  1200     Cost:   0.07842         W:0.2735        b:0.6435
Epoch:  1400     Cost:   0.07785         W:0.2688        b: 0.677
Epoch:  1600     Cost:   0.07749         W:0.2651        b:0.7033
Epoch:  1800     Cost:   0.07728         W:0.2622        b:0.7239
Epoch:  2000     Cost:   0.07714         W:0.2599        b:  0.74
Optimization Finished!
Final training cost: 0.077142656 W: 0.2599249 b: 0.7400293
```




```
Final testing cost: 0.076841086
Absolute mean square loss difference: 0.0003015697
```



* Giải thích:

- Import dependencies: Bắt đầu bằng việc import các thư viện cần thiết, bao gồm TensorFlow, NumPy và Matplotlib.

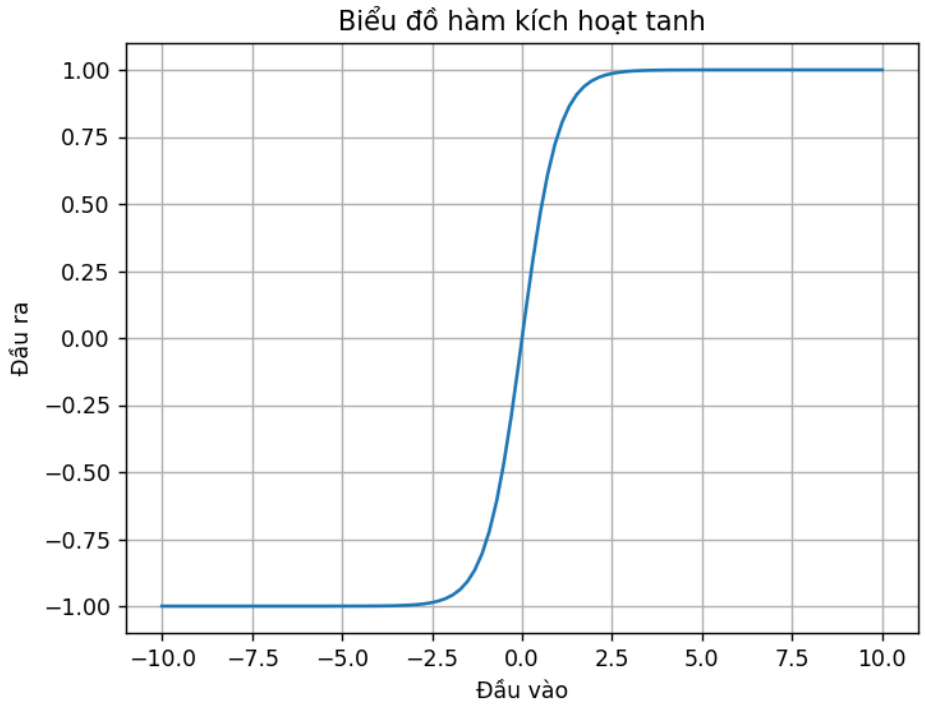
- Model Parameters: Định nghĩa các tham số cho mô hình như learning_rate (tốc độ học), số epoch (vòng lặp), và display_step (số epoch cần in ra thông tin).
- Training Data: Xác định dữ liệu huấn luyện train_X và các giá trị mục tiêu tương ứng train_y. Đây là các điểm dữ liệu mà mô hình sẽ học để thực hiện Linear Regression.
- Test Data: Xác định dữ liệu kiểm tra test_X và các giá trị mục tiêu tương ứng test_y. Dữ liệu này được sử dụng để kiểm tra hiệu suất của mô hình sau khi huấn luyện.
- Placeholders for feature and target vectors: Tạo hai placeholders X và y để làm nơi chứa dữ liệu đầu vào và đầu ra. Các giá trị sẽ được cung cấp thông qua các feed_dict trong quá trình chạy.
- Set model weights and bias: Sử dụng biến TensorFlow để đại diện cho trọng số W và độ lệch b của mô hình Linear Regression. Chúng được khởi tạo một cách ngẫu nhiên.
- Construct a linear model: Xây dựng mô hình tuyến tính linear_model bằng cách kết hợp trọng số và độ lệch với đầu vào X.
- Mean squared error (MSE): Định nghĩa hàm mất mát bằng cách tính toán sai số bình phương trung bình giữa mục tiêu thực tế và dự đoán của mô hình.
- Gradient Descent: Sử dụng thuật toán gradient descent để tối ưu hóa hàm mất mát. Phương thức minimize của tf.train.GradientDescentOptimizer được sử dụng để thực hiện quá trình này.
- Initializing the variables: Khởi tạo tất cả biến TensorFlow bằng phương thức tf.global_variables_initializer().
- Training Loop: Bắt đầu vòng lặp huấn luyện, trong đó gradient descent được thực hiện cho một số epoch xác định. Các thông tin về mất mát và tham số mô hình được in ra sau mỗi display_step epoch.
- Optimization Finished: In thông báo khi quá trình huấn luyện kết thúc.
- Graphic display: Vẽ biểu đồ để trực quan hóa dữ liệu huấn luyện và đường phù hợp của mô hình Linear Regression.
- Testing the model: Sử dụng dữ liệu kiểm tra để đánh giá hiệu suất của mô hình. In ra mất mát kiểm tra và hiệu suất tương đối với mất mát huấn luyện.
- Display fitted line on test data: Vẽ biểu đồ để hiển thị dữ liệu kiểm tra và đường phù hợp của mô hình trên dữ liệu kiểm tra.
- Kết quả của ví dụ này là mô hình Linear Regression đã được huấn luyện và được kiểm tra trên dữ liệu kiểm tra, và đồ thị trực quan hóa dữ liệu và đường phù hợp của mô hình.

3.6

```python
import tensorflow as tf
import matplotlib.pyplot as plt

# Khởi tạo đầu vào và trọng số
X = tf.constant([2.0, 3.0, 4.0], dtype=tf.float32)
W = tf.constant([0.05, 0.07, 0.09], dtype=tf.float32)

# Khởi tạo sai số
b = tf.constant(0.6, dtype=tf.float32)

# Tính toán tổng đầu vào đã được nhân với trọng số, sau đó cộng với sai số
z = tf.reduce_sum(tf.multiply(X, W)) + b

# Sử dụng hàm kích hoạt tanh
output = tf.nn.tanh(z)

# In kết quả
print("Output:", output.numpy())
# Vẽ biểu đồ cho hàm kích hoạt tanh
x = tf.linspace(-10.0, 10.0, 100)  # Tạo một dãy giá trị đầu vào từ -10 đến 10
y = tf.nn.tanh(x)  # Tính giá trị đầu ra tương ứng với hàm kích hoạt tanh

plt.plot(x, y)
plt.xlabel("Đầu vào")
plt.ylabel("Đầu ra")
plt.title("Biểu đồ hàm kích hoạt tanh")
plt.grid(True)
plt.show()

```

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

```python
# Import các thư viện cần thiết
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

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

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 tái tạo kết quả. Việc đặt hạt giống này có ý nghĩa khi bạn muốn có khả năng tái 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 đỗ.

Kết quả: