Họ và tên: Trần Vân 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.

```python
#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.

```python
# 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.placehoder**
    - 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

```python
# 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.

```python
# 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.

```python
# 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.

```python
# 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.

```python
# 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.

```python
# 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
```

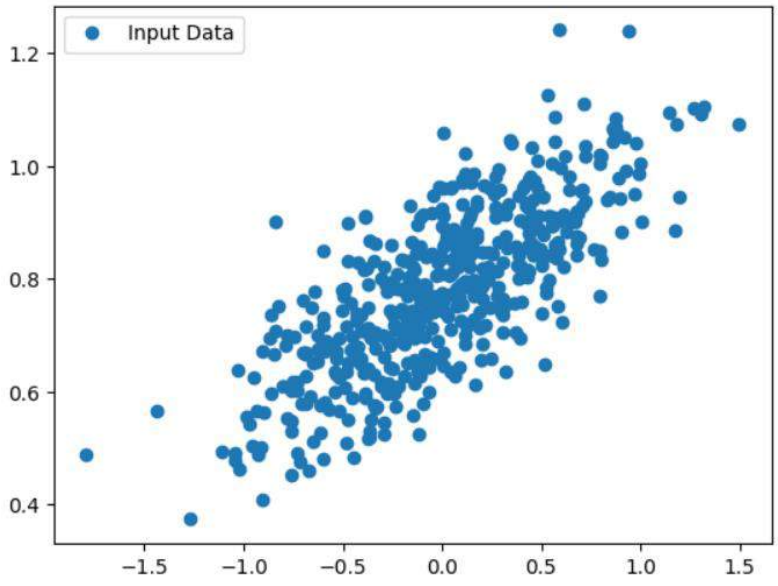
- **tf.sigmoid(z)**
  - Là một hàm số được sử dụng để tính giá trị sigmoid của một số thực. Hàm sigmoid được định nghĩa bởi công thức:

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

  - 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.

```python
# 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()
```


Input Data
1.2
1.0
0.8
0.6
0.4
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

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.

```python
# 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.

```python
# importing the dependencies
import tensorflow.compat.v1 as tf
import numpy as np
import matplotlib.pyplot as plt
tf.compat.v1.disable_eager_execution()

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

- Các dòng này import các thư viện vần 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.

```python
# 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()
```

+ `**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
```

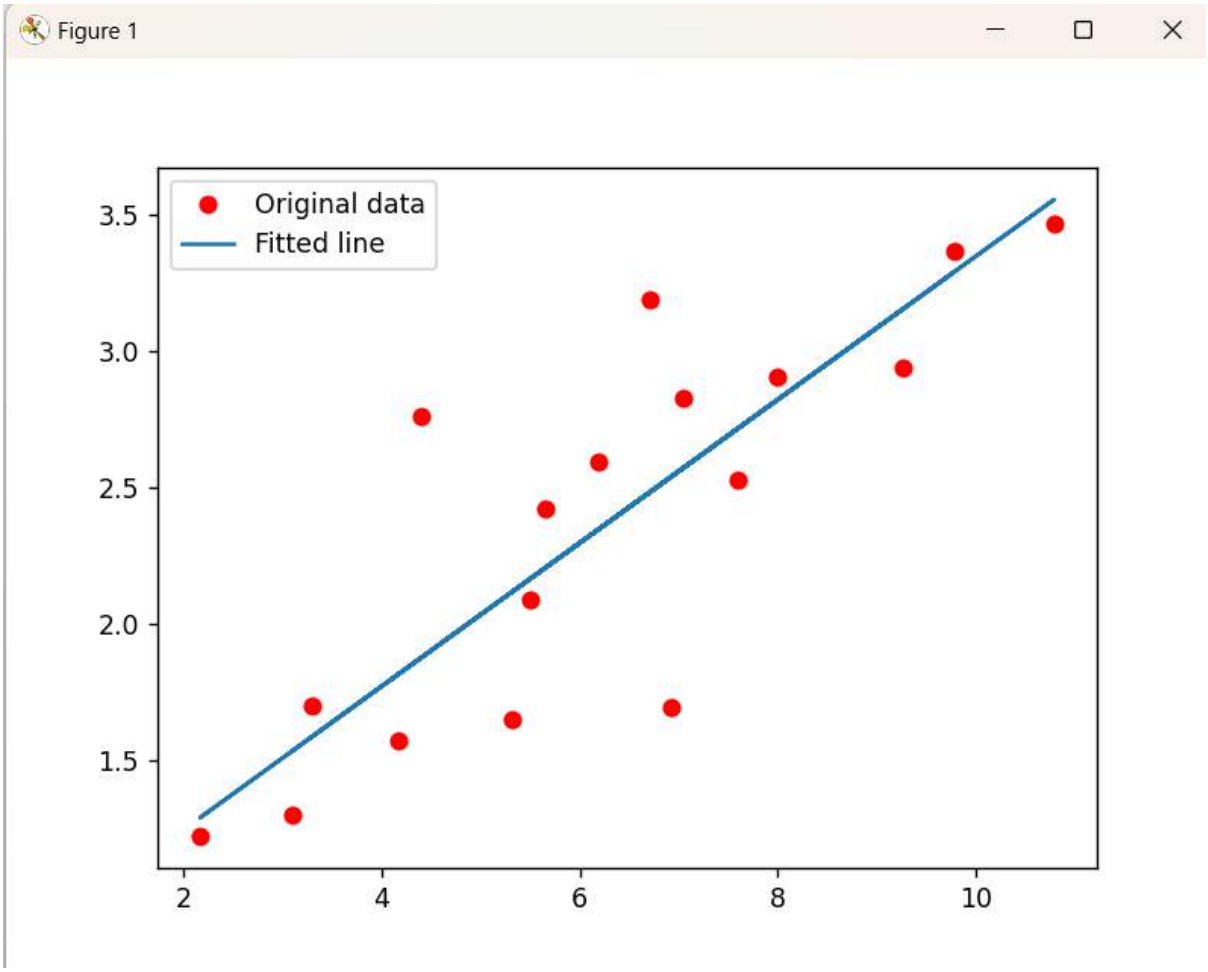


Figure 1
Original data
Fitted line
3.5
3.0
2.5
2.0
1.5
2
4
6
8
10

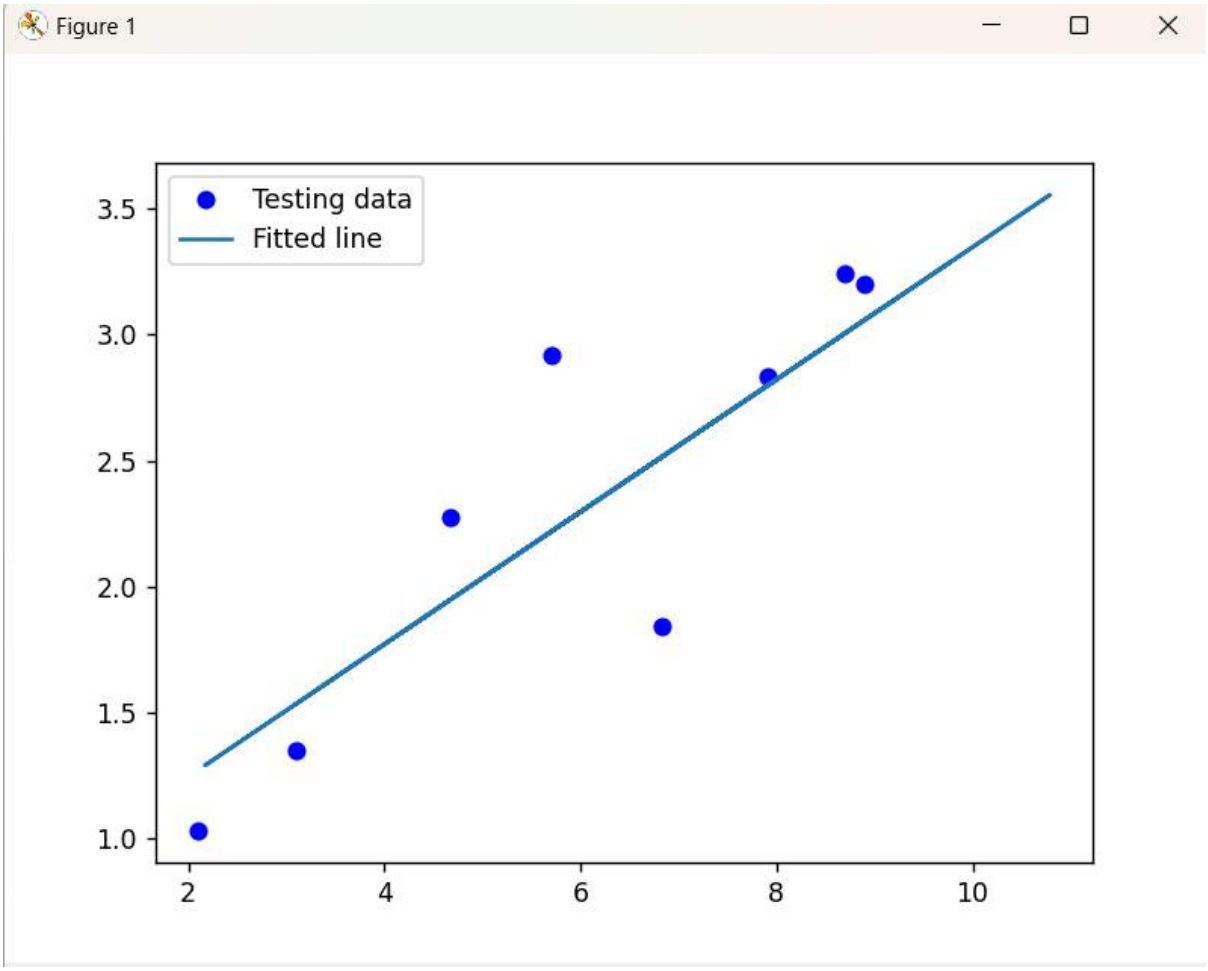


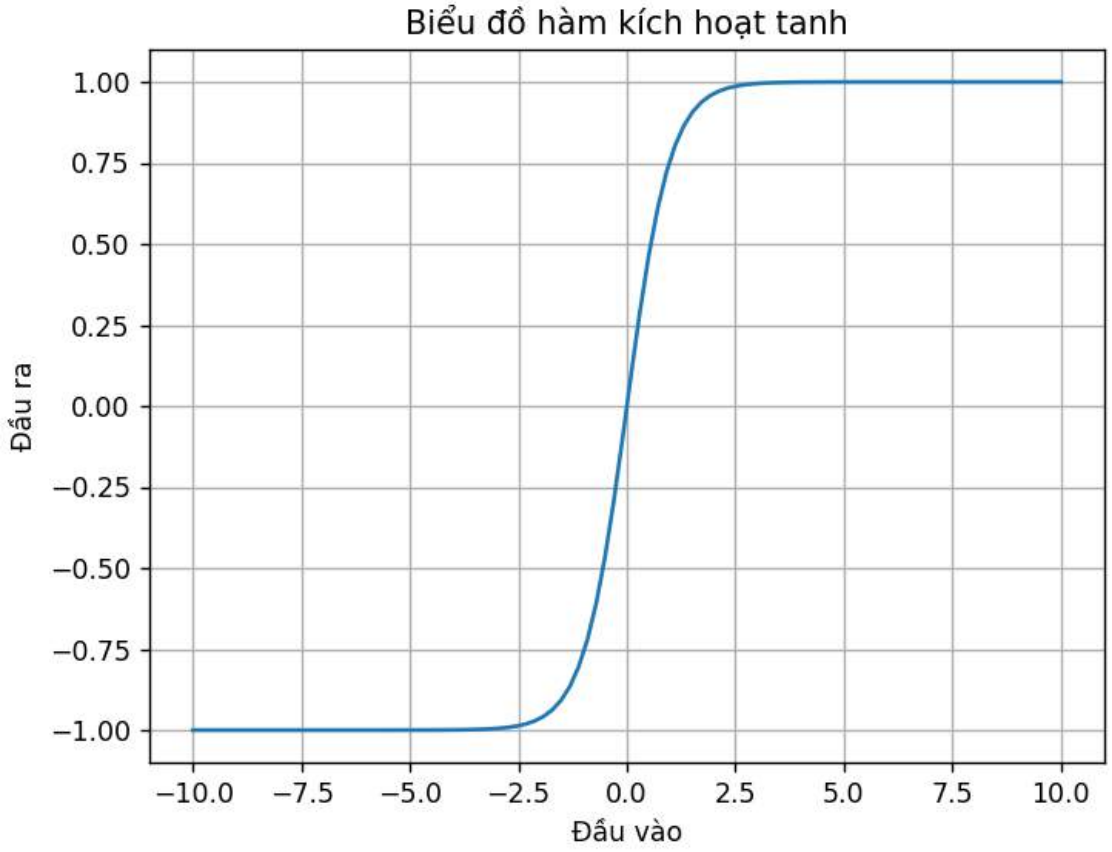
Figure 1
Testing data
Fitted line
3.5
3.0
2.5
2.0
1.5
1.0
2
4
6
8
10

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.

bai37.py > ...

```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).

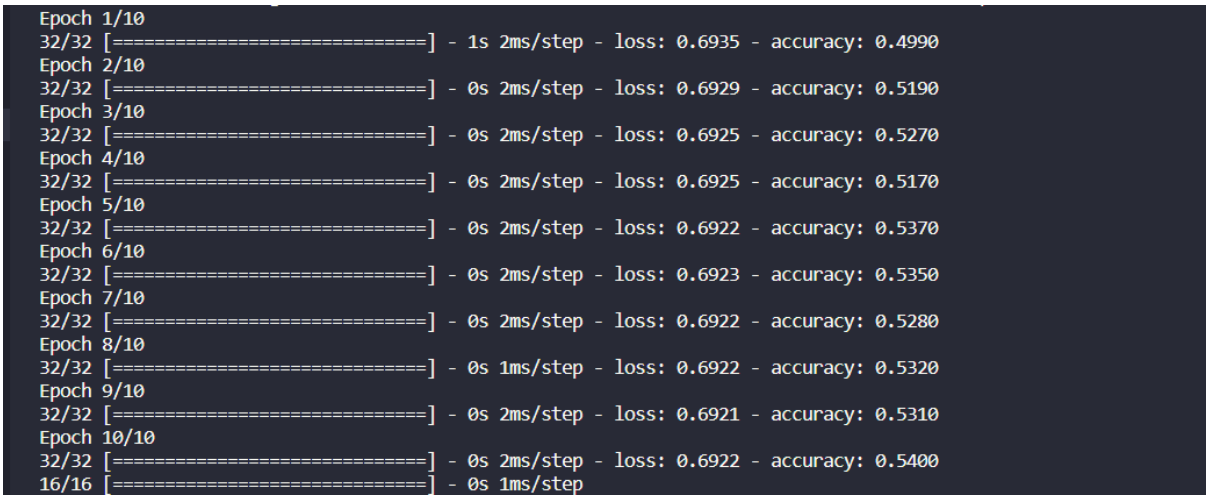
  - **`epochs=10`:** Đây là số lần mô hình sẽ đi qua toàn bộ dữ liệu huấn luyện trong quá trình huấn luyện. Mỗi lượt qua toàn bộ dữ liệu được gọi là một epoch.

  - **`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:



Họ và tên: Trần Vân Anh
MSV: B20DCCN075

BT4: DEEP LEARNING

4.1. Trình bày hiểu biết của mình về tensorflow và 5 ví dụ minh họa

1. Khái niệm

TensorFlow là một thư viện phần mềm mã nguồn mở được phát triển bởi Google, đặc biệt được sử dụng rộng rãi trong lĩnh vực học sâu (deep learning) và máy học (machine learning). Nó cung cấp một nền tảng mạnh mẽ để xây dựng và huấn luyện các mô hình máy học, đặc biệt là mạng thần kinh sâu (neural networks). TensorFlow cho phép xây dựng và huấn luyện các mô hình phức tạp thông qua việc tạo và quản lý các đồ thị tính toán.

Đặc trưng của TensorFlow là khả năng xây dựng các mô hình sử dụng các đồ thị dữ liệu (data flow graphs), trong đó các nút trong đồ thị đại diện cho các phép toán toán học và các cạnh đại diện cho dữ liệu (tensors) được truyền đi giữa các nút. TensorFlow tối ưu hóa việc tính toán trên dữ liệu này, đặc biệt là khi dữ liệu có kích thước lớn và yêu cầu tính toán song song trên nhiều thiết bị như CPU và GPU.

Được thiết kế để linh hoạt và dễ sử dụng, TensorFlow cung cấp nhiều API khác nhau cho các mức độ và mục tiêu sử dụng khác nhau. Người dùng có thể sử dụng TensorFlow để xây dựng các mô hình từ đơn giản đến phức tạp, từ ứng dụng học máy đơn lẻ đến các hệ thống thông minh phức tạp.\

2. API

TensorFlow cung cấp nhiều API (Giao diện lập trình ứng dụng). Chúng có thể được phân loại thành 2 nhóm chính:

- API cấp thấp:
  - Điều khiển lập trình toàn bộ
  - Được khuyến nghị cho các nhà nghiên cứu học máy
  - Cung cấp mức kiểm soát tốt đối với các mô hình
  - TensorFlow Core là API cấp thấp của TensorFlow.

- API cấp cao:

- Xây dựng trên nền TensorFlow Core
- Dễ học và sử dụng hơn so với TensorFlow Core
- Làm cho các công việc lặp lại dễ dàng hơn và nhất quán hơn giữa các người dùng khác nhau
- "tf.contrib.learn" là một ví dụ về API cấp cao.

3. Tensor

Một tensor có thể được hiểu đơn giản là một mảng đa chiều. Cụ thể, nó có thể là một scalar (mảng 0 chiều), vector (mảng 1 chiều), ma trận (mảng 2 chiều), hoặc một tensor với số chiều lớn hơn. Chẳng hạn:

- Scalar (0 chiều): Một số duy nhất.
- Vector (1 chiều): Một mảng các số.
- Ma trận (2 chiều): Một bảng số.
- Tensor (nhiều chiều): Một mảng nhiều chiều.

Ví dụ:

```python
import tensorflow as tf

# Tạo một scalar (0 chiều)
scalar_tensor = tf.constant(5)  # Đây là một tensor 0 chiều

# Tạo một vector (1 chiều)
vector_tensor = tf.constant([1, 2, 3])  # Đây là một tensor 1 chiều

# Tạo một ma trận (2 chiều)
matrix_tensor = tf.constant([[1, 2], [3, 4]])  # Đây là một tensor 2 chiều

# Tạo một tensor 3 chiều
tensor_3d = tf.constant([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])  # Đây là một tensor 3 chiều
```

4. Constant

Trong TensorFlow, tf.constant là một hàm được sử dụng để tạo một Tensor có giá trị không thay đổi (constant), nghĩa là giá trị của Tensor này không thể được thay đổi sau khi đã được tạo. Đây là một cách phổ biến để đưa dữ liệu vào mô hình hoặc định nghĩa các hằng số trong đồ thị tính toán.

Dưới đây là một ví dụ về cách sử dụng tf.constant:

```python
import tensorflow as tf

# Tạo một Tensor constant với giá trị là 10 và kiểu dữ liệu là int32
constant_tensor = tf.constant(10, dtype=tf.int32)

# Khởi tạo một phiên TensorFlow
with tf.Session() as sess:
    # Chạy đồ thị tính toán để lấy giá trị của Tensor constant
    constant_value = sess.run(constant_tensor)
    print("Giá trị của Tensor constant:", constant_value)  # Kết quả: 10
```

Ở đây, chúng ta tạo một Tensor constant với giá trị là 10 và kiểu dữ liệu là int32 bằng cách sử dụng tf.constant. Sau đó, chúng ta chạy đồ thị tính toán để lấy giá trị của Tensor constant bằng cách sử dụng sess.run.

Lưu ý rằng giá trị của Tensor constant không thể thay đổi sau khi đã được tạo, nó chỉ mang giá trị mà bạn đã gán ban đầu.

5. Operation

Trong TensorFlow, một Operation (hoặc Op) đại diện cho một phép toán hoặc một hành động được thực hiện trên dữ liệu để tạo ra một kết quả. Các Operation là các nút trong đồ thị tính toán của TensorFlow, và chúng thực hiện các phép toán số học, logic, ma trận, hoặc bất kỳ phép toán nào khác cần thiết trong quá trình tính toán.

Các Loại Operation:
TensorFlow cung cấp nhiều loại Operation khác nhau, bao gồm phép toán số học (cộng, trừ, nhân, chia), phép toán logic (AND, OR, NOT), phép toán ma trận, activation functions, loss functions, optimizer operations, và nhiều loại khác.
Phép Toán (Operation) trong Đồ Thị Tính Toán:

Mỗi Operation là một nút trong đồ thị tính toán của TensorFlow.
Đầu vào của một Operation là các Tensor đại diện cho dữ liệu vào, và đầu ra là một Tensor mới chứa kết quả của phép toán.

Khởi tạo và Thực Thi Operation:
Để thực thi một Operation và lấy kết quả, bạn cần tạo một phiên TensorFlow và sử dụng phương thức run.
Khi chạy một Operation, nó có thể cần dữ liệu đầu vào từ các Tensor khác trong đồ thị, và TensorFlow tự động xác định thứ tự thực thi dựa trên các phụ thuộc.
Dưới đây là một ví dụ về cách tạo và sử dụng một Operation trong TensorFlow:

```python
import tensorflow as tf

# Tạo các Tensor đầu vào
a = tf.constant(5)
b = tf.constant(3)

# Tạo một Operation thực hiện phép cộng
addition = tf.add(a, b)

# Khởi tạo một phiên TensorFlow
with tf.Session() as sess:
    # Chạy đồ thị tính toán để lấy kết quả của Operation
    result = sess.run(addition)
    print("Kết quả của phép cộng:", result)  # Kết quả: 8
```

Trong ví dụ này, chúng ta tạo hai Tensor constant a và b, sau đó tạo một Operation (addition) để thực hiện phép cộng giữa a và b. Khi chúng ta chạy đồ thị tính toán và thực thi Operation, kết quả của phép cộng được in ra.

6. Variable

Khi huấn luyện một mô hình trong TensorFlow, ta sử dụng các biến (variables) để lưu trữ và cập nhật các tham số của mô hình. Các biến là các bộ đệm trong bộ nhớ (in-memory buffers) chứa các tensors, và chúng được thiết kế để chứa các giá trị có thể thay đổi trong quá trình huấn luyện.

Trước đó, chúng ta đã làm việc với các tensors constant (hằng số) trong TensorFlow, chúng là các tensors có giá trị không thay đổi sau khi được khởi tạo. Các tensors constant được sử dụng để đại diện cho dữ liệu không thay đổi hoặc các giá trị mà chúng ta không muốn thay đổi trong quá trình tính toán.

Trong khi đó, biến (variables) trong TensorFlow được sử dụng để đại diện cho các tham số mô hình như trọng số (weights) và bias. Các giá trị của các biến có thể thay đổi theo thời gian, đặc biệt là trong quá trình huấn luyện mô hình, khi chúng được cập nhật để tối ưu hóa hiệu suất của mô hình.

Tóm lại, biến (variables) là một khái niệm quan trọng trong TensorFlow, cho phép lưu trữ và cập nhật các tham số mô hình trong quá trình huấn luyện, khác với các tensors constant mà chúng ta đã làm việc trước đó.

```python
import tensorflow as tf

# Khởi tạo một biến với giá trị ban đầu là 0 và kiểu dữ liệu là float32
my_variable = tf.Variable(0.0, dtype=tf.float32)

# Tạo một phép cộng để tăng giá trị của biến lên 1
add_operation = tf.add(my_variable, 1)

# Tạo một phép gán để cập nhật giá trị của biến
update_operation = tf.assign(my_variable, add_operation)

# Khởi tạo biến trước khi sử dụng
init = tf.global_variables_initializer()

# Bắt đầu một phiên tính toán TensorFlow
with tf.Session() as sess:
    # Khởi tạo biến
    sess.run(init)

    # Hiển thị giá trị ban đầu của biến
    print("Giá trị ban đầu của biến: {}".format(sess.run(my_variable)))

    # Thực hiện 5 lần cập nhật biến
    for _ in range(5):
        sess.run(update_operation)
        print("Giá trị biến sau {} lần cập nhật: {}".format(_, sess.run(my_variable)))
```

7. Placeholder

Một đồ thị có thể được tham số hóa để chấp nhận đầu vào từ bên ngoài, được gọi là placeholders. Một placeholder là một lời hứa để cung cấp một giá trị sau này. Trong quá trình đánh giá đồ thị mà liên quan đến các nút placeholder, một tham số feed_dict được truyền vào phương thức run của phiên để xác định các Tensor cung cấp giá trị cụ thể cho những nút placeholder này. Xem xét ví dụ dưới đây:

```python
import tensorflow as tf

# Tạo một placeholder với kiểu dữ liệu float32 và shape None (kích thước có thể thay đổi)
input_placeholder = tf.placeholder(tf.float32, shape=(None,))

# Tạo một phép toán sử dụng placeholder
output = input_placeholder * 2

# Tạo một phiên TensorFlow
with tf.Session() as sess:
    # Tạo một feed dictionary để cung cấp dữ liệu đầu vào cho placeholder
    input_data = np.array([1.0, 2.0, 3.0])
    feed_dict = {input_placeholder: input_data}

    # Chạy đồ thị tính toán với dữ liệu được cung cấp
    result = sess.run(output, feed_dict=feed_dict)
    print("Kết quả:", result)  # Kết quả: [2.0, 4.0, 6.0]
```

Trong ví dụ này:

Chúng ta định nghĩa một placeholder với kiểu dữ liệu float32 và shape None, cho phép đầu vào có thể thay đổi kích thước theo mong muốn.
Sau đó, chúng ta định nghĩa một phép toán sử dụng placeholder, ở đây là nhân đầu vào với
Trong phiên TensorFlow, chúng ta tạo một feed dictionary để cung cấp dữ liệu đầu vào cho placeholder.
Khi chúng ta chạy đồ thị tính toán, chúng ta truyền feed dictionary để cung cấp giá trị cho placeholder và tính toán output.
Placeholder cho phép ta linh hoạt đưa dữ liệu vào trong quá trình tính toán và rất hữu ích khi xây dựng mô hình học máy.

4.2.Giải thích các dòng dánh dấu # trong Bài tập 3.5

 # importing the dependencies:
import các thư viện cần thiết (dependencies)

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

- TensorFlow (tf): Thư viện học sâu mạnh mẽ được sử dụng để xây dựng, huấn luyện và triển khai các mô hình học máy. Import thư viện TensorFlow. tensorflow.compat.v1 cho biết đây là phiên bản tương thích (compatibility version 1.x) của TensorFlow. Tên ngắn gọn tf là tên gọi được sử dụng trong mã để gọi các hàm và lớp từ thư viện TensorFlow.
- NumPy (np): Thư viện cực kỳ quan trọng trong Python cho tính toán khoa học và toán học trên mảng nhiều chiều.
- Matplotlib.pyplot (plt): Thư viện được sử dụng để vẽ đồ thị và biểu đồ.

 # Model Parameters:
Khai báo thông số mô hình:

```
learning_rate = 0.01
training_epochs = 2000
display_step = 200
```

- Tốc độ học (learning_rate): Đây là một siêu tham số quan trọng trong quá trình huấn luyện. Nó quy định tốc độ mà mô hình sẽ cập nhật các trọng số của nó.
- Số lượng epochs (training_epochs): Số lượng lần mô hình sẽ duyệt qua toàn bộ dữ liệu huấn luyện.
- Bước hiển thị thông tin (display_step): Số lượng epochs mà sau đó thông tin về quá trình huấn luyện sẽ được hiển thị.

 # Training Data:
Dữ liệu huấn luyện và kiểm tra:

```
train_X = np.asarray([3.3,4.4,5.5,...])
train_y = np.asarray([1.7,2.76,2.09,...])
test_X = np.asarray([6.83, 4.668, 8.9,...])
test_y = np.asarray([1.84, 2.273, 3.2,...])
```

Dữ liệu huấn luyện (train_X và train_y) và dữ liệu kiểm tra (test_X và test_y): Đây là các tập dữ liệu mà mô hình sẽ được huấn luyện và kiểm tra sau khi huấn luyện xong.

 # Test Data:
Dòng này khai báo dữ liệu kiểm tra (test_X). Đây là các đặc trưng đầu vào được sử dụng để kiểm tra mô hình sau khi đã được huấn luyện.

 # Set placeholders for feature and target vectors:
Khai báo placeholders và biến mô hình:

```
X = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
W = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")
```

Placeholders (X và y): Đối tượng trong TensorFlow để cung cấp dữ liệu đầu vào và đầu ra khi huấn luyện và kiểm tra mô hình.
Biến (W và b): Các tham số mô hình sẽ được tối ưu hoá trong quá trình huấn luyện.

 # Set model weights and bias:
Tạo biến (W và b) để lưu trữ trọng số và sai số (bias) của mô hình.

 # Construct a linear model:
xây dựng mô hình tuyến tính (linear_model) dựa trên đặc trưng đầu vào và các trọng số.

```
linear_model = W*X + b
```

Mô hình tuyến tính (linear_model): Đây là mô hình đơn giản với một đầu vào (X), một trọng số (W), và một sai số (b).

 # Mean squared error:
Hàm mất mát

```
cost = tf.reduce_sum(tf.square(linear_model - y)) / (2*n_samples)
```

Mean Squared Error (cost): Hàm mất mát sẽ đánh giá sự khác biệt giữa đầu ra dự đoán và đầu ra thực tế, và mục tiêu là giảm hàm mất mát này.

 # Gradient descent:
Tối ưu hóa.

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Optimizer (optimizer): Sử dụng thuật toán Gradient Descent để tối ưu hóa hàm mất mát và cập nhật các tham số mô hình.

 # Initializing the variables:
Khởi tạo biến mô hình trước khi bắt đầu quá trình huấn luyện.

```
init = tf.global_variables_initializer()
```

Khởi tạo biến (init): Đây là bước quan trọng để khởi tạo tất cả các biến mô hình.

 # Launch the graph with tf.Session() as sess:
Bắt đầu phiên TensorFlow và huấn luyện mô hình:
Phiên TensorFlow (sess): Đây là phiên làm việc với TensorFlow, nơi mà ta thực hiện các phép tính và huấn luyện mô hình.

 # Fit all training data:
Dòng này mô tả việc huấn luyện mô hình trên dữ liệu huấn luyện (train_X và train_y).

 # perform gradient descent step
Huấn luyện mô hình

```
sess.run(optimizer, feed_dict={X: train_X, y: train_y})
```

Huấn luyện mô hình (sess.run(optimizer, ...)) : Chạy quá trình huấn luyện mô hình bằng cách thực hiện một bước tối ưu hóa trên dữ liệu huấn luyện.

 # Display logs per epoch step:

Hiển thị thông tin sau mỗi epoch để theo dõi tiến trình của quá trình huấn luyện.

```
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)
```

In ra thông tin sau mỗi epoch: Sau mỗi số epoch được chỉ định bởi display_step, in ra mất mát và các thông số mô hình.

 # Print final parameter values:

In ra giá trị cuối cùng của trọng số và sai số sau khi huấn luyện kết thúc.

```
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:

Dòng này mô tả việc hiển thị biểu đồ để trực quan hóa dữ liệu huấn luyện và mô hình đã học.

```
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()
```

Hiển thị biểu đồ: Vẽ biểu đồ để so sánh dữ liệu huấn luyện và đường tuyến tính mà mô hình đã học.

 # Testing the model:

Kiểm tra mô hình trên dữ liệu kiểm tra (test_X).

```
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))
```

Kiểm tra mô hình trên dữ liệu kiểm tra (testing_cost) : Đánh giá hiệu suất của mô hình bằng cách tính toán mất mát trên dữ liệu kiểm tra.

In ra kết quả kiểm tra: In ra mất mát trên dữ liệu kiểm tra và sự khác biệt giữa mất mát trên dữ liệu huấn luyện và kiểm tra.

 # Display fitted line on test data:
Hiển thị đường tuyến tính đã học được trên dữ liệu kiểm tra để đánh giá mô hình.

```
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()
```

Hiển thị đường tuyến tính (plt.plot(...)) trên dữ liệu kiểm tra: Đây là bước để đánh giá mô hình bằng cách so sánh đầu ra dự đoán trên dữ liệu kiểm tra với đường tuyến tính đã học được.

4.3. Từ tensorflow đến keras. Trình bày hiểu biết của mình về keras và 5 ví dụ minh họa

1. **Khái niệm**

Keras là một giao diện lập trình ứng dụng (API) dành cho Python được sử dụng để xây dựng và đào tạo các mô hình học máy và học sâu một cách dễ dàng và hiệu quả. Keras được phát triển bởi François Chollet và ban đầu là một dự án độc lập, sau đó đã được tích hợp chặt chẽ vào TensorFlow, một thư viện học máy và học sâu phổ biến.

Keras giúp các nhà phát triển xây dựng mô hình neural network một cách nhanh chóng và dễ dàng thông qua các lớp và hàm API trực quan. Nó được thiết kế để làm cho việc định nghĩa, đào tạo, và đánh giá các mô hình học máy trở nên đơn giản hơn, giúp tiết kiệm thời gian và công sức cho các dự án học máy và học sâu.

Một số đặc điểm quan trọng của Keras bao gồm:

- Dễ sử dụng: Keras cung cấp cú pháp đơn giản và dễ hiểu cho việc xây dựng mô hình neural network, điều này làm giảm ngưỡng cho người mới học về học máy.

- Tích hợp: Keras được tích hợp chặt chẽ với TensorFlow, cho phép sử dụng toàn bộ khả năng của TensorFlow trong các mô hình Keras.

- Hỗ trợ nhiều backend: Keras hỗ trợ nhiều backend như TensorFlow, Theano, và CNTK, cho phép bạn lựa chọn backend tùy theo nhu cầu của dự án.

- Học chuyển tiếp: Keras cung cấp sẵn các mô hình được đào tạo trước (pre-trained models) cho nhiều nhiệm vụ, như phân loại ảnh hoặc xử lý ngôn ngữ tự nhiên, giúp bạn áp dụng học chuyển tiếp (transfer learning) cho dự án của mình.

Tóm lại, Keras là một công cụ mạnh mẽ và linh hoạt giúp bạn xây dựng các mô hình học máy và học sâu một cách dễ dàng và nhanh chóng, đặc biệt là khi bạn làm việc với dự án liên quan đến học máy và sử dụng TensorFlow như backend.

2. Keras so với TensorFlow:

TensorFlow là một thư viện mã nguồn mở toàn diện để tạo và làm việc với mạng neural, chẳng hạn như những mô hình được sử dụng trong dự án Học máy (ML) và Học sâu.
Keras là một API cấp cao chạy trên nền tảng của TensorFlow. Keras giúp đơn giản hóa việc triển khai các mạng neural phức tạp với khung làm việc dễ sử dụng.

Mức độ trừu tượng hóa:

- Keras: Keras là một giao diện lập trình ứng dụng (API) cao cấp, nơi bạn có thể định nghĩa mô hình neural network bằng cách xếp các lớp lại với nhau một cách dễ dàng. Nó cung cấp một cách trừu tượng hóa để định nghĩa và đào tạo các mô hình học máy và học sâu.
- TensorFlow: TensorFlow là một thư viện mạnh mẽ cho việc tính toán số học và học máy, chứ không phải là một API trừu tượng. Bạn cần định nghĩa mô hình neural network bằng cách xây dựng và tùy chỉnh các phép tính trên biểu đồ tính toán.

Tích hợp:

- Keras: Keras ban đầu là một thư viện độc lập, nhưng sau đó đã được tích hợp chặt chẽ vào TensorFlow. Hiện nay, Keras là một phần của TensorFlow và được sử dụng như một lớp trừu tượng cao cấp giúp đơn giản hóa việc xây dựng mô hình trên nền tảng TensorFlow.
- TensorFlow: TensorFlow là một thư viện riêng biệt và không phụ thuộc vào bất kỳ API cụ thể nào. Nó cung cấp cho bạn sự linh hoạt tuyệt vời trong việc xây dựng và tùy chỉnh các mô hình học máy và học sâu.

Độ linh hoạt:

- Keras: Keras thích hợp cho người mới bắt đầu với học máy và học sâu, cũng như cho các dự án đòi hỏi sự nhanh chóng trong việc xây dựng mô hình. Nó

giới hạn tính linh hoạt một chút, nhưng làm cho quá trình đào tạo mô hình đơn giản hơn.
- TensorFlow: TensorFlow cung cấp cho bạn sự linh hoạt tối đa để tạo và tùy chỉnh mô hình của mình. Điều này thích hợp cho các dự án phức tạp và đòi hỏi tùy chỉnh cao.

Học chuyển tiếp (Transfer Learning):
- Keras: Keras cung cấp nhiều mô hình được đào tạo trước (pre-trained models) cho các nhiệm vụ phổ biến như phân loại ảnh hoặc xử lý ngôn ngữ tự nhiên. Điều này giúp bạn áp dụng học chuyển tiếp (transfer learning) một cách dễ dàng.
- TensorFlow: TensorFlow cũng hỗ trợ học chuyển tiếp, nhưng việc triển khai có thể phức tạp hơn và đòi hỏi kiến thức sâu về cách hoạt động của mô hình.

Tóm lại, Keras là một lớp trừu tượng cao cấp giúp đơn giản hóa việc xây dựng mô hình trên TensorFlow và thích hợp cho những người mới bắt đầu hoặc các dự án đòi hỏi tính nhanh chóng. TensorFlow là một thư viện mạnh mẽ và linh hoạt hơn, thích hợp cho các dự án phức tạp và yêu cầu tùy chỉnh cao.

3. Sử dụng Keras khi:

- Bạn mới bắt đầu với học máy hoặc học sâu và muốn học một cách dễ dàng và nhanh chóng.
- Bạn muốn xây dựng mô hình nhanh chóng và tập trung vào cấu trúc của mô hình hơn là chi tiết triển khai.
- Dự án của bạn đủ phức tạp, nhưng bạn muốn giữ mã của mình ngắn gọn và dễ hiểu.
- Bạn muốn sử dụng các mô hình được đào tạo trước (pre-trained models) và thực hiện học chuyển tiếp một cách dễ dàng.

Thực tế, một cách phổ biến là sử dụng Keras như một giao diện trừu tượng cho TensorFlow. Bạn có thể xây dựng mô hình của mình bằng Keras, nhưng dưới lớp trừu tượng này, Keras sẽ sử dụng TensorFlow để thực hiện các phép tính. Điều này kết hợp sự dễ dàng sử dụng của Keras với tính linh hoạt và hiệu suất của TensorFlow.

4. 5 ví dụ

VD1: Phân loại ảnh với mạng neural đơn giản:

```python
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.datasets import mnist
from keras.utils import to_categorical

# Tải dữ liệu từ MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Chuẩn hóa dữ liệu và mã hóa one-hot
X_train = X_train / 255.0
X_test = X_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Xây dựng mô hình
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Biên dịch và đào tạo mô hình
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

Ví dụ này mô tả cách xây dựng và đào tạo một mạng neural để phân loại chữ số viết tay từ bộ dữ liệu MNIST.

VD2: Dự đoán giá nhà với mạng neural hồi quy:

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split
import pandas as pd

# Đọc dữ liệu từ tệp CSV
data = pd.read_csv('house_prices.csv')
X = data.drop('price', axis=1)
y = data['price']

# Chia dữ liệu thành tập huấn luyện và tập kiểm tra
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Xây dựng mô hình hồi quy
model = Sequential()
model.add(Dense(64, input_dim=3, activation='relu'))
model.add(Dense(1))

# Biên dịch và đào tạo mô hình
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)

# Đánh giá mô hình
mse = model.evaluate(X_test, y_test)
```

Ví dụ này minh họa việc xây dựng một mạng neural hồi quy để dự đoán giá nhà dựa trên các đặc trưng như diện tích, số phòng, và vị trí.

VD3: Phân loại văn bản với LSTM:

```
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

# Dữ liệu văn bản và nhãn
texts = ["This is a positive sentence.", "That is a negative statement.", "I feel great today."]
labels = [1, 0, 1]

# Chuyển đổi văn bản thành dãy số và đệm chuỗi
tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
padded_sequences = pad_sequences(sequences)

# Xây dựng mô hình LSTM
model = Sequential()
model.add(Embedding(input_dim=1000, output_dim=64, input_length=padded_sequences.shape[1]))
model.add(LSTM(128))
model.add(Dense(1, activation='sigmoid'))

# Biên dịch và đào tạo mô hình
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(padded_sequences, np.array(labels), epochs=5)
```

Ví dụ này minh họa việc sử dụng mạng LSTM để phân loại văn bản thành hai lớp: tích cực (1) và tiêu cực (0).

VD4: Mô hình Học sâu đơn giản với một lớp ẩn

```
from keras.models import Sequential
from keras.layers import Dense

# Khởi tạo mô hình
model = Sequential()

# Thêm lớp ẩn
model.add(Dense(units=10, activation='relu', input_dim=20))

# Lớp output
model.add(Dense(units=1, activation='sigmoid'))

# Biên dịch mô hình
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

VD5: Transfer Learning với mô hình đã được đào tạo trước

```python
from keras.applications import VGG16
from keras.models import Model
from keras.layers import Flatten, Dense

# Load mô hình VGG16 đã được đào tạo trước
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Thêm các lớp mới
x = base_model.output
x = Flatten()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# Xây dựng mô hình mới
model = Model(inputs=base_model.input, outputs=predictions)
```

4.4 Chạy 3 ví dụ dưới đây và giải thích các dòng lệnh theo hiểu biết của mình

VD1:

In [2]:
```python
import tensorflow as tf
import numpy as np
#print("TensorFlow version:", tf.__version__)
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
model = tf.keras.models.Sequential([
 tf.keras.layers.Flatten(input_shape=(28, 28)),
 tf.keras.layers.Dense(128, activation='relu'),
 tf.keras.layers.Dropout(0.2),
 tf.keras.layers.Dense(10)
])
predictions = model(x_train[:1])
predictions
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
loss_fn(y_train[:1], predictions)
model.compile(optimizer='adam',
 loss=loss_fn,
 metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test, verbose=2)
```

```
A local file was found, but it seems to be incomplete or outdated because the auto file hash does not match the original value
of 731c5ac602752760c8e48fbffcf8c3b850d9dc2a2aedcf2cc48468fc17b673d1 so we will re-download the data.
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 17s 1us/step
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss: 0.2990 - accuracy: 0.9140
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1447 - accuracy: 0.9564
Epoch 3/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1078 - accuracy: 0.9674
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0889 - accuracy: 0.9725
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0749 - accuracy: 0.9768
313/313 - 1s - loss: 0.0738 - accuracy: 0.9781 - 1s/epoch - 3ms/step
```

Out[2]: [0.07379115372896194, 0.9781000018119812]

```
# Nhập các thư viện cần thiết
import tensorflow as tf
import numpy as np

# print("TensorFlow version:", tf.__version__) --> in ra phiên bản của thư viện TensorFlow hiện đang được sử dụng.

# Tải bộ dữ liệu
mnist = tf.keras.datasets.mnist
"""
+) x_train và y_train là các biến lưu trữ dữ liệu huấn luyện.
   x_train là tập hợp các hình ảnh chữ số viết tay
   y_train là tập hợp các nhãn tương ứng cho mỗi hình ảnh.
+)x_test và y_test là các biến lưu trữ dữ liệu kiểm tra.
   x_test là tập hợp các hình ảnh chữ số viết tay được sử dụng để kiểm tra mô hình
   y_test là tập hợp các nhãn tương ứng cho mỗi hình ảnh trong tập kiểm tra.
"""
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Tiền xử lý dữ liệu
# Giá trị pixel của các hình ảnh được chuẩn hóa về phạm vi [0, 1] bằng cách chia chúng cho 255.
x_train, x_test = x_train / 255.0, x_test / 255.0

# Xác định kiến trúc mô hình
"""
Mô hình được xác định dưới dạng một mô hình tuần tự với ba lớp:
lớp định hình (flatten) để chuyển đổi hình ảnh đầu vào 2D thành 1D,
lớp kết nối đầy đủ (dense) với 128 đơn vị và hàm kích hoạt ReLU,
```

lớp dropout với tỷ lệ dropout là 0.2 để ngăn chặn overfitting, và
lớp kết nối đầy đủ với 10 đơn vị (bằng với số lớp).
"""

```
model = tf.keras.models.Sequential([
 tf.keras.layers.Flatten(input_shape=(28, 28)),
 tf.keras.layers.Dense(128, activation='relu'),
 tf.keras.layers.Dropout(0.2),
 tf.keras.layers.Dense(10)
])

# Tạo ra dự đoán cho hình ảnh đầu tiên trong tập huấn luyện.
predictions = model(x_train[:1])
```

"""
Hàm mất mát được sử dụng là hàm mất mát chéo danh mục thưa (sparse categorical cross-entropy).
Nó so sánh các nhãn thực tế (y_train[:1]) với các logits dự đoán (predictions).
"""

```
predictions
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
loss_fn(y_train[:1], predictions)

# Biên dịch mô hình
model.compile(optimizer='adam',
 loss=loss_fn,
 metrics=['accuracy'])

# Huấn luyện mô hình
model.fit(x_train, y_train, epochs=5)
```

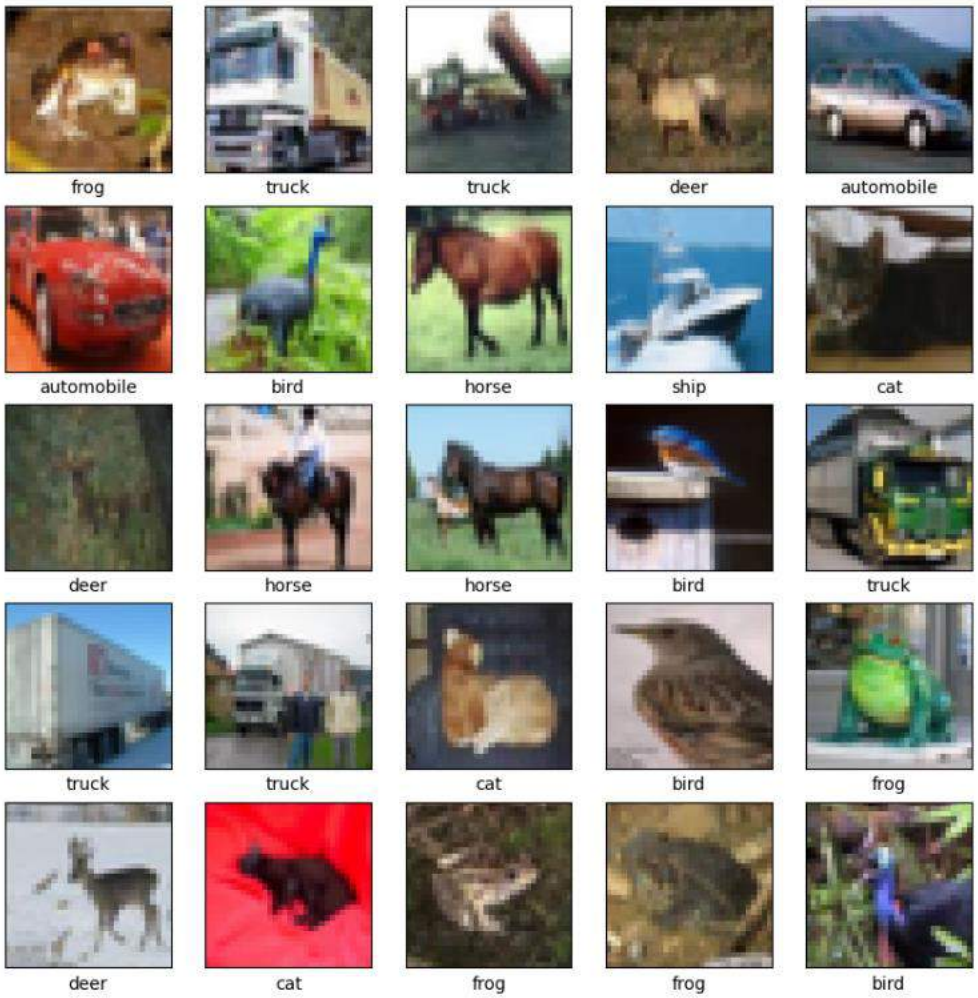
#Đánh giá mô hình dựa trên trên dữ liệu kiểm tra, và kết quả mất mát và độ chính xác trên kiểm tra được in ra.

```
model.evaluate(x_test, y_test, verbose=2)
```

VD2:

In [3]:
```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
 'dog', 'frog', 'horse', 'ship', 'truck']
plt.figure(figsize=(10,10))
for i in range(25):
 plt.subplot(5,5,i+1)
 plt.xticks([])
 plt.yticks([])
 plt.grid(False)
 plt.imshow(train_images[i])
 # The CIFAR labels happen to be arrays,
 # which is why you need the extra index
 plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 114s 1us/step
```



import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

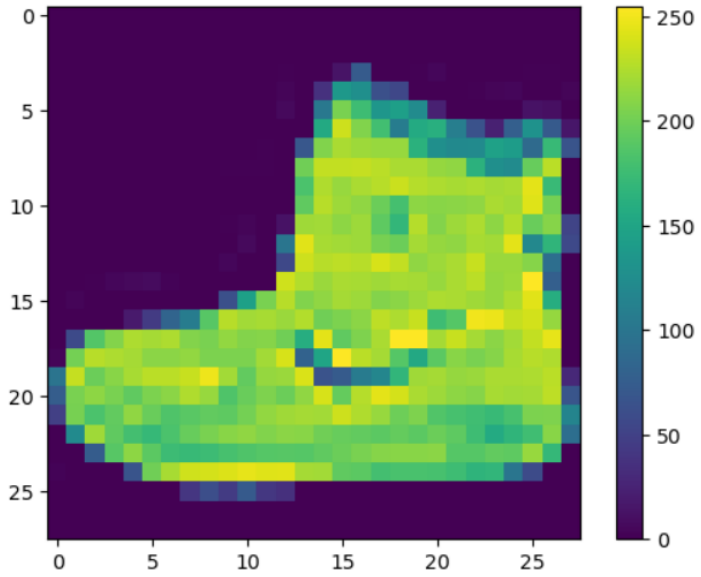
Tải dữ liệu CIFAR-10 và chia thành tập huấn luyện và tập kiểm tra

```
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()

# Chuẩn hóa giá trị pixel để nằm trong khoảng từ 0 đến 1
train_images, test_images = train_images / 255.0, test_images / 255.0

# Định nghĩa tên các lớp trong CIFAR-10
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
 'dog', 'frog', 'horse', 'ship', 'truck']

# Trực quan hóa 25 ảnh từ tập huấn luyện
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i]) # Hàm hiển thị ảnh

# Nhãn trong bộ dữ liệu CIFAR-10 được biểu diễn dưới dạng mảng, đó là lý do tại
sao bạn cần chỉ số bổ sung.
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

VD3:

```
import tensorflow as tf
# Helper libraries
import numpy as np
import matplotlib.pyplot as plt
print(tf.__version__)
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
train_images.shape
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
train_images = train_images / 255.0
test_images = test_images / 255.0
plt.figure(figsize=(10,10))
for i in range(25):
 plt.subplot(5,5,i+1)
 plt.xticks([])
 plt.yticks([])
 plt.grid(False)
 plt.imshow(train_images[i], cmap=plt.cm.binary)
 plt.xlabel(class_names[train_labels[i]])
plt.show()
```

```
2.13.0
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 2us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 43s 2us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0s/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 4s 1us/step
```




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

# In ra phiên bản của TensorFlow
print(tf.__version__)

# Tải dữ liệu Fashion MNIST và chia thành tập huấn luyện và tập kiểm tra
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

# Định nghĩa tên các lớp trong Fashion MNIST
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

```
# Hiển thị kích thước của tập huấn luyện
train_images.shape

# Trực quan hóa một ảnh từ tập huấn luyện
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()

# Chuẩn hóa giá trị pixel để nằm trong khoảng từ 0 đến 1
train_images = train_images / 255.0
test_images = test_images / 255.0

# Trực quan hóa 25 ảnh từ tập huấn luyện
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```