## 3.1 Running and explain (Constant trong tensorflow)

Constant trong tensorflow

tf.constant: tạo ra một **tensor bất biến (immutable)**, tức là **không thể thay đổi giá trị** sau khi tạo.

Thường dùng cho **dữ liệu cố định** (hằng số, chuỗi, ma trận đầu vào, …).

Nếu muốn thay đổi giá trị thì dùng tf.Variable thay vì tf.constant.

Ví dụ:

import tensorflow as tf

c = tf.constant(5)  # hằng số
print(c)  # tf.Tensor(5, shape=(), dtype=int32)

3.1.1

```
# nguyễn Việt Quang
import tensorflow as tf

h = tf.constant("Hello")
w = tf.constant(" World!")

hw = tf.strings.join([h, w])
print(hw.numpy().decode("utf-8"))
```

```
Hello World!
```

- tf.constant("Hello"): tạo một tensor chứa chuỗi "Hello".
- tf.strings.join([h, w]): nối hai tensor chuỗi thành "Hello World!".
- hw.numpy(): lấy giá trị thực từ tensor dưới dạng bytes.
- .decode("utf-8"): chuyển bytes thành chuỗi Python bình thường (str).

3.1.2

```python
# 3.1.2
# Nguyễn Việt Quang
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(v)
```

```
tf.Tensor(
[[[5. 3. 4. 6.]
  [2. 3. 4. 0.]]

 [[5. 3. 4. 6.]
  [2. 3. 4. 0.]]

 [[5. 3. 4. 6.]
  [2. 3. 4. 0.]]], shape=(3, 2, 4), dtype=float32)
```

Phân Tích Code

Đoạn mã này sử dụng thư viện **TensorFlow** để tạo ra và minh họa các **Tensor** có số chiều (rank) khác nhau.

> import tensorflow.compat.v1 as tf: Dòng này **nhập thư viện TensorFlow**, nhưng sử dụng tensorflow.compat.v1 để tương thích với các API cũ hơn của TensorFlow (phiên bản 1.x).
>
> tf.constant(giá_trị, kiểu_dữ_liệu): Hàm này được sử dụng để tạo một **Tensor hằng số** (constant tensor). Một tensor là cấu trúc dữ liệu cơ bản của TensorFlow, giống như mảng (array) hoặc danh sách (list) trong các ngôn ngữ lập trình khác.

Các biến x, y, z, t, u, và v đại diện cho các tensor với số chiều (rank) khác nhau, hay còn gọi là số trục (axes) hoặc số chiều của mảng.

| Biến | Giá trị | Rank (Số chiều) | Mô tả |
|---|---|---|---|
| x | 5 | **0** (Scalar) | Một **số vô hướng**, chỉ có một giá trị duy nhất. |
| y | [5] | **1** (Vector) | Một **vector** (mảng một chiều). |
| z | [5,3,4] | **1** (Vector) | Một **vector** có ba phần tử. |

| Biến | Giá trị | Rank (Số chiều) | Mô tả |
|---|---|---|---|
| t | [[5,3,4,6], [2,3,4,7]] | **2** (Matrix) | Một **ma trận** (mảng hai chiều) với 2 hàng và 4 cột. |
| u | [[[5,3,4,6],[2,3,4,0]]] | **3** (3D Tensor) | Một tensor ba chiều, có 1 "lớp" chứa một ma trận 2x4. |
| v | [[[...], [...] ], [[...], [...] ], [[...], [...] ]] | **3** (3D Tensor) | Một tensor ba chiều, có 3 "lớp", mỗi lớp là một ma trận 2x4. |

Cuối cùng, print(v) sẽ **in ra thông tin của Tensor** v. Vì đây là Tensor phiên bản 1.x, nó sẽ không in ra giá trị cụ thể mà thay vào đó sẽ hiển thị cấu trúc của tensor, bao gồm tên, hình dạng (shape), và kiểu dữ liệu.

# 3.2. Running and explain (Variable in tensorflow)

Trong **TensorFlow**, Variable là một kiểu đối tượng dùng để **lưu trữ và cập nhật các giá trị có thể thay đổi trong quá trình tính toán**, thường là **trọng số (weights), tham số (bias)** của mô hình học máy.

> Khác với Tensor (chỉ lưu dữ liệu bất biến), Variable cho phép thay đổi giá trị khi **train** bằng thuật toán tối ưu.
>
> Khởi tạo bằng:

```
import tensorflow as tf
# Tạo biến với giá trị ban đầu
W = tf.Variable(initial_value=3.0, dtype=tf.float32)
# Cập nhật giá trị biến
W.assign(5.0)
```

> **Ứng dụng chính:** chứa tham số của mạng nơ-ron, được cập nhật liên tục trong quá trình huấn luyện.

Tóm lại: Variable **= tensor có thể thay đổi giá trị**, phục vụ cho việc học và tối ưu mô hình.

3.2.1

```python
# Nguyễn Việt Quang
# 3.2.1 - Ví dụ sử dụng TensorFlow 1.x trong môi trường TensorFlow 2.x

import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Tắt eager mode để dùng session (theo style TF1)

# Khai báo biến (Variable) thay vì constant
x1 = tf.Variable(5.3, dtype=tf.float32)  # Biến x1 có giá trị ban đầu 5.3
x2 = tf.Variable(4.3, dtype=tf.float32)  # Biến x2 có giá trị ban đầu 4.3

# Toán tử nhân 2 biến (graph node)
x = tf.multiply(x1, x2)  # x = x1 * x2

# Lệnh khởi tạo tất cả biến toàn cục trong graph
init = tf.global_variables_initializer()

# Mở session để chạy graph
with tf.Session() as sess:
    sess.run(init)          # Chạy bước khởi tạo biến
    t = sess.run(x)         # Tính giá trị của x (tức 5.3 * 4.3)
    print(t)                # In kết quả ra màn hình
```

```
22.79
```

3.2.2

```python
# 3.2.2
# Nguyễn Việt Quang
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Tắt eager mode để dùng Session (theo style TF1)

# Khai báo 2 tensor Variable dạng ma trận 2x3
x1 = tf.Variable([[5.3, 4.5, 6.0],
                  [4.3, 4.3, 7.0]], dtype=tf.float32)

x2 = tf.Variable([[4.3, 4.3, 7.0],
                  [5.3, 4.5, 6.0]], dtype=tf.float32)

# Phép nhân phần tử (element-wise multiplication)
# Kết quả: ma trận cùng kích thước 2x3
x = tf.multiply(x1, x2)

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

# Mở Session để chạy graph
with tf.Session() as sess:
    sess.run(init)     # Khởi tạo giá trị cho biến
    t = sess.run(x)    # Tính giá trị của phép nhân
    print(t)           # In kết quả
```

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

3.2.3

```python
# 3.2.3
# Nguyen viet Quang
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Bắt buộc để dùng Session theo style TF1

# Tạo một biến (Variable) tensor 2x2 toàn số 0
node = tf.Variable(tf.zeros([2, 2]))

# Mở Session để chạy computation graph
with tf.Session() as sess:
    # B1: Khởi tạo tất cả biến toàn cục
    sess.run(tf.global_variables_initializer())

    # B2: In giá trị ban đầu của node (toàn số 0)
    print("Tensor value before addition:\n", sess.run(node))

    # B3: Thực hiện phép cộng element-wise (node = node + 1)
    node = node.assign(node + tf.ones([2, 2]))

    # B4: In giá trị node sau khi cộng
    print("Tensor value after addition:\n", sess.run(node))
    # ❌ Không cần sess.close() vì with ... as đã tự đóng
```

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

# 3.3. Runnng and explain (Placeholder)

placeholder **trong TensorFlow (TF1.x)**:

> tf.placeholder là **một nút chờ dữ liệu đầu vào** trong computation graph.
>
> Khi xây dựng graph, bạn chỉ định **kiểu dữ liệu (**dtype**)** và **hình dạng (**shape**)**, nhưng **không gán giá trị ngay**.
>
> Khi chạy graph bằng sess.run(...), bạn phải đưa dữ liệu thật vào qua tham số feed_dict.

## Ví dụ (TF1.x):

```
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()
# Tạo placeholder cho 2 số thực
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
# Phép cộng
c = a + b
with tf.Session() as sess:
    result = sess.run(c, feed_dict={a: 3, b: 5})
    print(result)  # 8.0
```

Lưu ý: Trong **TensorFlow 2.x**, placeholder đã bị bỏ, thay vào đó ta dùng tf.function hoặc **Keras Input layers**.

Nói ngắn gọn:
placeholder = "ô trống" trong graph để **nạp dữ liệu sau khi chạy**.

3.3.1

```python
# Nguyễn Việt Quang
# 3.3.1
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Bắt buộc để dùng Session (theo style TF1)

# Tạo placeholder: chỗ trống để đưa dữ liệu vào lúc chạy
# dtype = float32, shape = None (chấp nhận mọi hình dạng)
x = tf.placeholder(tf.float32, shape=None)

# Định nghĩa phép cộng: y = x + x
y = tf.add(x, x)

# Mở session để chạy computation graph
with tf.Session() as sess:
    x_data = 5  # Dữ liệu đầu vào
    # Chạy node y, truyền giá trị cho x thông qua feed_dict
    result = sess.run(y, feed_dict={x: x_data})
    print(result)  # In kết quả: 10.0
```

```
10.0
```

3.3.2

```python
# Nguyễn Việt Quang
# 3.3.2
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Tắt eager execution để dùng Session (style TF1)

# Tạo placeholder cho tensor 2 chiều
# shape=[None, 3] nghĩa là: số hàng bất kỳ, nhưng mỗi hàng có đúng 3 cột
x = tf.placeholder(tf.float32, [None, 3])

# Định nghĩa phép cộng element-wise: y = x + x
y = tf.add(x, x)

# Mở session để thực thi computation graph
with tf.Session() as sess:
    # Dữ liệu đầu vào: 1 hàng, 3 cột (khớp với shape=[None,3])
    x_data = [[1.5, 2.0, 3.3]]

    # Chạy node y, truyền giá trị thực cho placeholder x qua feed_dict
    result = sess.run(y, feed_dict={x: x_data})

    # In kết quả
    print(result)
```

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

3.3.3

```python
# Nguyễn Việt Quang
# 3.3.3
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Tắt eager execution để dùng Session (style TF1)

# Tạo placeholder 3 chiều:
# shape = [None, None, 3]
# - Chiều 1: số batch (linh động, có thể bất kỳ)
# - Chiều 2: số hàng trong mỗi batch (cũng linh động)
# - Chiều 3: cố định = 3 cột
x = tf.placeholder(tf.float32, [None, None, 3])

# Định nghĩa phép cộng element-wise: y = x + x
y = tf.add(x, x)

# Mở session để chạy computation graph
with tf.Session() as sess:
    # Dữ liệu đầu vào: 1 batch, 1 hàng, 3 cột
    # shape = (1, 1, 3)
    x_data = [[[1, 2, 3]]]

    # Thực thi node y, nạp dữ liệu thật cho x qua feed_dict
    result = sess.run(y, feed_dict={x: x_data})

    # In kết quả
    print(result)
```

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

## 3.3.4

```python
# Nguyễn Việt Quang
# 3.3.4
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Tắt eager execution để dùng Session (theo style TF1)

# Placeholder cho tensor 3 chiều
# shape = [None, 4, 3]:
# - None : số batch (linh động, có thể 1 hoặc nhiều)
# - 4    : mỗi batch có 4 hàng
# - 3    : mỗi hàng có 3 cột
x = tf.placeholder(tf.float32, [None, 4, 3])

# Phép cộng element-wise: y = x + x
y = tf.add(x, x)
```

```
# Mở session để thực thi computation graph
with tf.Session() as sess:
    # Dữ liệu đầu vào: 1 batch, 4 hàng, 3 cột
    # shape của x_data = (1, 4, 3)
    x_data = [[[1, 2, 3],
               [2, 3, 4],
               [2, 3, 5],
               [0, 1, 2]]]

    # Thực thi node y, nạp dữ liệu vào placeholder x
    result = sess.run(y, feed_dict={x: x_data})

    # In kết quả
    print(result)
```

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

3.3.5

```
# Nguyễn Việt Quang
# 3.3.5
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Dùng TF1-style với Session

# Placeholder cho tensor 3 chiều
# shape = [2, 4, 3]:
# - 2 : số batch = 2
# - 4 : mỗi batch có 4 hàng
# - 3 : mỗi hàng có 3 cột
x = tf.placeholder(tf.float32, [2, 4, 3])

# Định nghĩa phép cộng element-wise: y = x + x
y = tf.add(x, x)
```

```python
# Mở Session để chạy graph
with tf.Session() as sess:
    # Dữ liệu đầu vào: 2 batch, mỗi batch 4x3
    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 thi node y, truyền dữ liệu qua feed_dict
    result = sess.run(y, feed_dict={x: x_data})

    # In kết quả
    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
# Nguyễn Việt Quang
# 3.3.6
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Dùng Session theo style TF1

# Tạo 2 placeholder có shape [2, 4, 3]
# - 2 batch
# - mỗi batch có 4 hàng
# - mỗi hàng có 3 cột
x = tf.placeholder(tf.float32, [2, 4, 3])
y = tf.placeholder(tf.float32, [2, 4, 3])

# Phép cộng element-wise: z = x + y
z = tf.add(x, y)

# Phép nhân element-wise: u = x * y
u = tf.multiply(x, y)
```

```
# Mở Session để thực thi graph
with tf.Session() as sess:
    # Dữ liệu đầu vào (2 batch, mỗi batch 4x3)
    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 giống x_data
    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]]
    ]
```

```
    # Chạy phép cộng
    result1 = sess.run(z, feed_dict={x: x_data, y: y_data})
    # Chạy phép nhân
    result2 = sess.run(u, feed_dict={x: x_data, y: y_data})

    # In kết quả
    print("result1 =\n", result1)  # Tổng (x + y)
    print("result2 =\n", result2)  # Tích (x * y)
```

```
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. Operation. Run and explain

## Operation trong TensorFlow là gì?

**Operation (op)** là **một nút trong computation graph** của TensorFlow.

Mỗi operation thực hiện **một phép tính** (vd: cộng, nhân, ma trận, convolution, activation function…).

Kết quả của một operation là **tensor**.

## Ví dụ đơn giản:

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

a = tf.constant(2)       # hằng số 2 (tensor)
b = tf.constant(3)       # hằng số 3 (tensor)
```

```
c = tf.add(a, b)          # operation cộng (a + b)
d = tf.multiply(a, b)     # operation nhân (a * b)
with tf.Session() as sess:
    print("c =", sess.run(c))  # 5
    print("d =", sess.run(d))  # 6
```

Ở đây:

tf.add là một operation (cộng 2 số).

tf.multiply là một operation (nhân 2 số).

c, d là **tensors** được tạo ra từ các operation.

## Tóm tắt:

**Tensor**: dữ liệu (giá trị, mảng).

**Operation**: phép tính tạo ra tensor mới.

**Graph**: tập hợp các tensor + operation liên kết với nhau.

3.4.1

```python
# 3.4.1
# Nguyễn Việt Quang
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Tắt eager execution để dùng style TF1

# ----------------------------
# Tạo các constant (giá trị cố định)
x1 = tf.constant(5.3, tf.float32)   # input 1
x2 = tf.constant(1.5, tf.float32)   # input 2

# Tạo các biến (tham số có thể train được)
w1 = tf.Variable(0.7, tf.float32)   # weight 1
w2 = tf.Variable(0.5, tf.float32)   # weight 2

# ----------------------------
# Xây dựng computation graph
u = tf.multiply(x1, w1)   # u = x1 * w1
v = tf.multiply(x2, w2)   # v = x2 * w2
z = tf.add(u, v)          # z = u + v
result = tf.sigmoid(z)    # result = sigmoid(z)

#
```

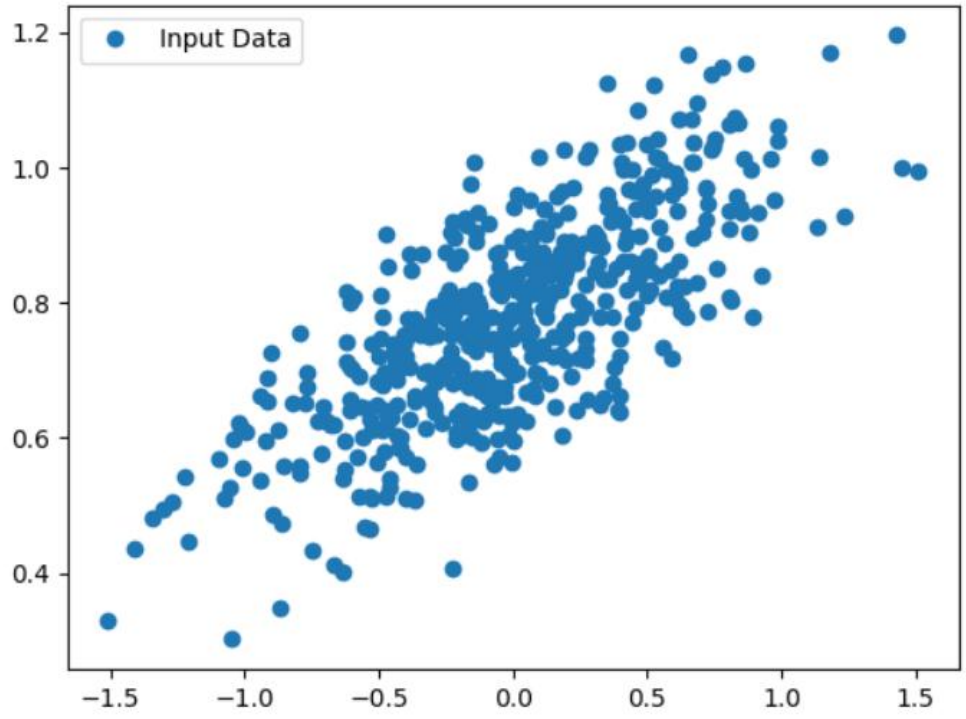
```
# ---------------------------
# Khởi tạo các biến
init = tf.global_variables_initializer()

# Thực thi graph trong Session
with tf.Session() as sess:
    sess.run(init)                  # chạy init để khởi tạo w1, w2
    output = sess.run(result)       # tính giá trị result
    print(output)                   # in ra kết quả
```

```
0.9885698
```

3.4.2

```
# 3.4.2
# Nguyễn Việt Quang
import numpy as np
import matplotlib.pyplot as plt

# --------------------------------
# Số lượng điểm dữ liệu
number_of_points = 500

# Danh sách chứa các điểm (sẽ convert thành array sau)
x_point = []
y_point = []

# Hệ số đường thẳng y = a*x + b
a = 0.22    # slope (độ dốc)
b = 0.78    # intercept (giao điểm trục y)
```

```python
# --------------------------------
# Sinh dữ liệu ngẫu nhiên theo phân phối chuẩn (Gaussian noise)
for i in range(number_of_points):
    x = np.random.normal(0.0, 0.5)          # sinh x từ N(0, 0.5)
    y = a*x + b + np.random.normal(0.0, 0.1)  # y = ax + b + nhiễu (noise)

    x_point.append([x])   # lưu dưới dạng list con [[x]] (chuẩn bị cho ML sau này)
    y_point.append([y])

# --------------------------------
# Vẽ scatter plot của dữ liệu
plt.plot(x_point, y_point, 'o', label='Input Data')  # vẽ dạng chấm (o)
plt.legend()     # hiển thị label
plt.show()       # hiển thị hình
```




3.4.3

```python
# 3.4.3
# Nguyễn Việt quang
import tensorflow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()  # Dùng chế độ TF1 (Session-based)

# ------------------------------
# Tạo placeholder cho input
# Mỗi input có dạng [None, 3] (batch_size x 3 feature)
x1 = tf.placeholder(tf.float32, [None, 3])
x2 = tf.placeholder(tf.float32, [None, 3])

# Tạo các biến weight (tham số có thể train được)
w1 = tf.Variable([0.5, 0.4, 0.7], tf.float32)  # vector w1 (3 phần tử)
w2 = tf.Variable([0.8, 0.5, 0.6], tf.float32)  # vector w2 (3 phần tử)

# ------------------------------
# Xây dựng computation graph
u1 = tf.multiply(w1, x1)   # nhân từng phần tử: u1 = w1 * x1
u2 = tf.multiply(w2, x2)   # nhân từng phần tử: u2 = w2 * x2
v = tf.add(u1, u2)         # cộng từng phần tử: v = u1 + u2
z = tf.sigmoid(v)          # áp dụng sigmoid cho từng phần tử
```

```python
# ------------------------------
# Khởi tạo biến
init = tf.global_variables_initializer()

# Thực thi trong session
with tf.Session() as sess:
    # Dữ liệu đầu vào (batch size = 1, 3 features)
    x1_data = [[1, 2, 3]]
    x2_data = [[1, 2, 3]]

    sess.run(init)  # bắt buộc khởi tạo biến trước khi dùng

    # Truyền dữ liệu thực tế vào placeholder
    result = sess.run(z, feed_dict={x1: x1_data, x2: x2_data})

    print("Output after sigmoid:\n", result)
```

```
Output after sigmoid:
 [[0.785835   0.85814893 0.9801597 ]]
```

3.4.4

```python
# 3.4.4
# Nguyễn Việt Quang
import tensorflow.compat.v1 as tf
import numpy as np
tf.disable_eager_execution()  # Dùng Session theo style TF1

# ------------------------------
# Tạo 2 ma trận số nguyên 3x3
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')

print("Matrix 1:\n", matrix1)
print("Matrix 2:\n", matrix2)

# ------------------------------
# Đưa dữ liệu NumPy vào TensorFlow constant
matrix1 = tf.constant(matrix1)
matrix2 = tf.constant(matrix2)
```

```python
# Tính toán các phép toán ma trận
matrix_product = tf.matmul(matrix1, matrix2)   # tích ma trận
matrix_sum = tf.add(matrix1, matrix2)          # cộng ma trận

# ------------------------------
# Ma trận float32 để tính định thức
matrix_3 = np.array([(2, 7, 2),
                     (1, 4, 2),
                     (9, 0, 2)], dtype='float32')
print("Matrix 3:\n", matrix_3)

# Tính định thức (determinant)
matrix_det = tf.linalg.det(matrix_3)

# ------------------------------
# Chạy computation graph trong Session
with tf.Session() as sess:
    result1 = sess.run(matrix_product)  # kết quả tích ma trận
    result2 = sess.run(matrix_sum)      # kết quả cộng ma trận
    result3 = sess.run(matrix_det)      # kết quả định thức

# ------------------------------
# In kết quả
print("\nMatrix Product (matrix1 x matrix2):\n", result1)
print("\nMatrix Sum (matrix1 + matrix2):\n", result2)
print("\nDeterminant of matrix_3:\n", result3)
```

```
Matrix 1:
 [[2 2 2]
 [2 2 2]
 [2 2 2]]
Matrix 2:
 [[1 1 1]
 [1 1 1]
 [1 1 1]]
Matrix 3:
 [[2. 7. 2.]
 [1. 4. 2.]
 [9. 0. 2.]]

Matrix Product (matrix1 x matrix2):
 [[6 6 6]
 [6 6 6]
 [6 6 6]]

Matrix Sum (matrix1 + matrix2):
 [[3 3 3]
 [3 3 3]
 [3 3 3]]

Determinant of matrix_3:
 55.999992
```

### 3.5. Running and explain Linear Regression model using TensorFlow Core API.

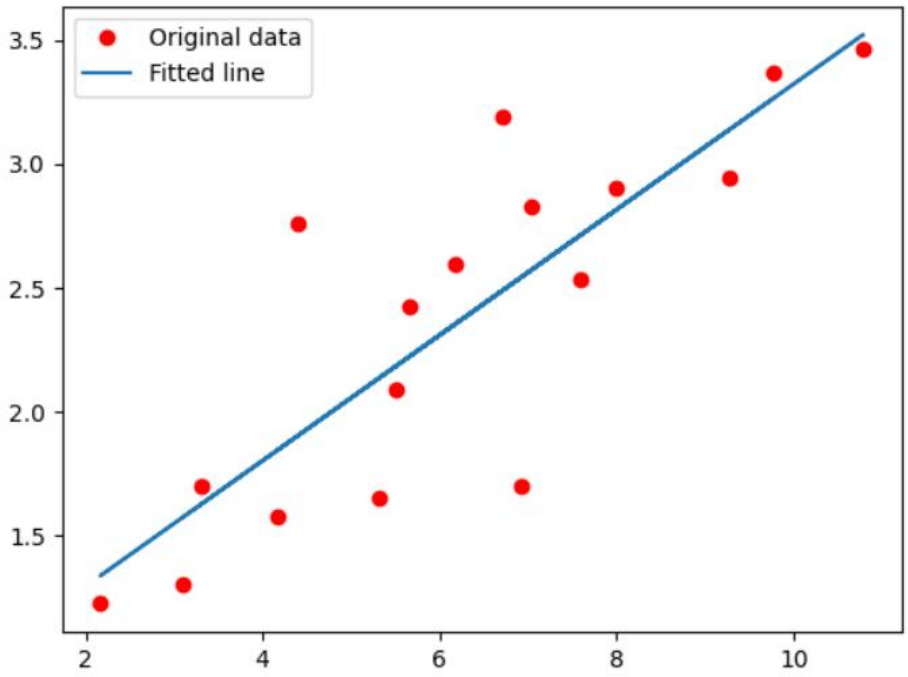
```python
# 3.5
# Nguyễn Việt Quang
# ------------------------------
# Import dependencies
import tensorflow.compat.v1 as tf
import numpy as np
import matplotlib.pyplot as plt

tf.disable_eager_execution()  # Dùng chế độ TF1 (Session-based)

# ------------------------------
# Model Parameters
learning_rate = 0.01       # tốc độ học (step size)
training_epochs = 2000     # số lần lặp training
display_step = 200         # hiển thị log sau mỗi 200 epoch

# ------------------------------
# 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]   # số lượng mẫu huấn luyện
```

```python
# ------------------------------
# Test Data
test_X = np.asarray([6.83, 4.668, 8.9, 7.91, 5.7, 8.7, 3.1, 2.1])
test_y = np.asarray([1.84, 2.273, 3.2, 2.831, 2.92, 3.24, 1.35, 1.03])

# ------------------------------
# Định nghĩa placeholder cho input và output
X = tf.placeholder(tf.float32)   # input features
y = tf.placeholder(tf.float32)   # output target

# ------------------------------
# Định nghĩa tham số mô hình (W và b)
W = tf.Variable(np.random.randn(), name="weight")  # trọng số
b = tf.Variable(np.random.randn(), name="bias")    # bias

# ------------------------------
# Xây dựng mô hình tuyến tính: y_pred = W*X + b
linear_model = W * X + b

# ------------------------------
# Hàm mất mát: Mean Squared Error (MSE)
cost = tf.reduce_sum(tf.square(linear_model - y)) / (2 * n_samples)

# ------------------------------
# Optimizer: Gradient Descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

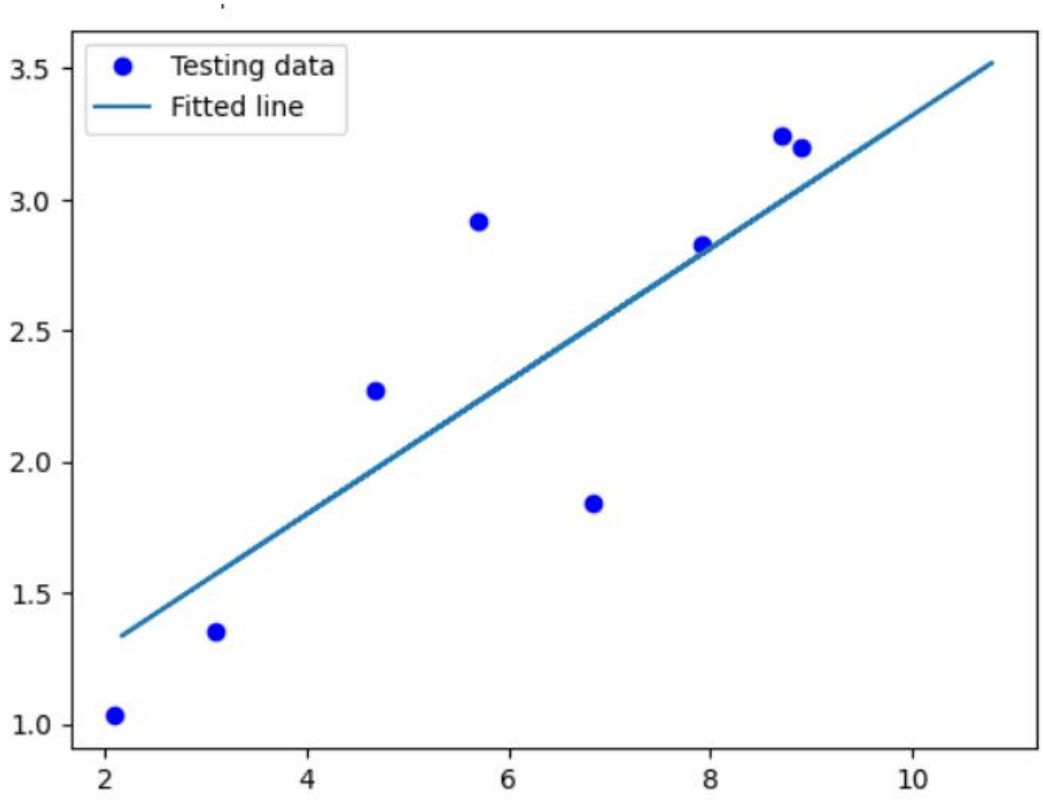
```
# ------------------------------
# Khởi tạo biến
init = tf.global_variables_initializer()

# ------------------------------
# Huấn luyện mô hình trong Session
with tf.Session() as sess:
    # Khởi tạo biến
    sess.run(init)

    # Training loop
    for epoch in range(training_epochs):
        # Thực hiện 1 bước gradient descent
        sess.run(optimizer, feed_dict={X: train_X, y: train_y})

        # Hiển thị log mỗi display_step epoch
        if (epoch + 1) % display_step == 0:
            c = sess.run(cost, feed_dict={X: train_X, y: train_y})
            print("Epoch:{0:6} \t Cost:{1:10.4f} \t W:{2:6.4f} \t b:{3:6.4f}"
                  .format(epoch + 1, c, sess.run(W), sess.run(b)))

    # ------------------------------
    # Kết quả cuối cùng sau huấn luyện
    print("\nOptimization 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')
```

```
# ------------------------------
# Vẽ đường fit trên tập train
plt.plot(train_X, train_y, 'ro', label='Original data')  # dữ liệu gốc
plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')  # đường hồi quy
plt.legend()
plt.show()

# ------------------------------
# Đánh giá mô hình trên tập test
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))

# ------------------------------
# Vẽ đường fit trên tập test
plt.plot(test_X, test_y, 'bo', label='Testing data')  # dữ liệu test
plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')  # đường hồi quy
plt.legend()
plt.show()
```

```
Epoch:   200     Cost:    0.0779        W:0.2697        b:0.6709
Epoch:   400     Cost:    0.0776        W:0.2658        b:0.6985
Epoch:   600     Cost:    0.0773        W:0.2627        b:0.7201
Epoch:   800     Cost:    0.0772        W:0.2603        b:0.7371
Epoch:  1000     Cost:    0.0771        W:0.2585        b:0.7504
Epoch:  1200     Cost:    0.0770        W:0.2570        b:0.7608
Epoch:  1400     Cost:    0.0770        W:0.2558        b:0.7690
Epoch:  1600     Cost:    0.0770        W:0.2549        b:0.7754
Epoch:  1800     Cost:    0.0769        W:0.2542        b:0.7805
Epoch:  2000     Cost:    0.0769        W:0.2537        b:0.7844

Optimization Finished!
Final training cost: 0.076941624 W: 0.25366235 b: 0.7844277
```




```
Final testing cost: 0.07789006
Absolute mean square loss difference: 0.00094843656
```




# Linear Regression là gì?

Đây là mô hình **hồi quy tuyến tính**, tìm đường thẳng tốt nhất để khớp với dữ liệu.

Công thức:

y=W⋅ X+by = W \cdot X + by=W⋅ X+b

Trong đó:

XXX: dữ liệu đầu vào (feature)

yyy: giá trị đầu ra (target)

WWW: trọng số (độ dốc của đường thẳng)

bbb: hệ số chặn (giao điểm với trục y)

Mục tiêu: học được giá trị **W** và **b** sao cho đường thẳng khớp dữ liệu nhất.

# Cách cài đặt bằng TensorFlow Core API (v1-style)

### Bước 1: Import thư viện

```
import tensorflow.compat.v1 as tfimport numpy as npimport matplotlib.pyplot
as plt
tf.disable_eager_execution()
```

## Bước 2: Dữ liệu huấn luyện

```
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]
```

## Bước 3: Khai báo placeholder và biến số

```
X = tf.placeholder(tf.float32)   # input
y = tf.placeholder(tf.float32)   # output

W = tf.Variable(np.random.randn(), name="weight")  # trọng số
b = tf.Variable(np.random.randn(), name="bias")    # hệ số chặn
```

## Bước 4: Xây dựng mô hình

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

## Bước 5: Hàm mất mát (Mean Squared Error)

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

## Bước 6: Bộ tối ưu (Gradient Descent)

```
optimizer =
tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(cost)
```

## Bước 7: Huấn luyện mô hình

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

    # huấn luyện 2000 vòng lặp
    for epoch in range(2000):
        sess.run(optimizer, feed_dict={X: train_X, y: train_y})

        # in kết quả sau mỗi 200 bước
        if (epoch+1) % 200 == 0:
            c = sess.run(cost, feed_dict={X: train_X, y: train_y})
```

```
            print("Epoch:", (epoch+1), "Cost=", c, "W=", sess.run(W), "b=",
sess.run(b))

    print("Huấn luyện xong!")
    print("W cuối cùng:", sess.run(W), "b cuối cùng:", sess.run(b))

    # Vẽ kết quả
    plt.plot(train_X, train_y, 'ro', label="Dữ liệu gốc")
    plt.plot(train_X, sess.run(W)*train_X + sess.run(b), label="Đường thẳng
khớp")
    plt.legend()
    plt.show()
```

# Quy trình hoạt động

**Tạo graph**: khai báo placeholder (X, y), biến (W, b), mô hình, cost, optimizer.

**Chạy session**: khởi tạo biến → lặp nhiều lần → optimizer cập nhật W, b.

**Kết quả**: tìm được đường thẳng tốt nhất mô tả mối quan hệ giữa X và y.

Tóm lại:
Linear Regression trong TensorFlow Core API = **Xây dựng graph → Chạy session → Tối ưu W, b để giảm hàm mất mát.**

## 3.6 Model Neuron Network

a. View Fig1.1 in the diagram. Code and explain

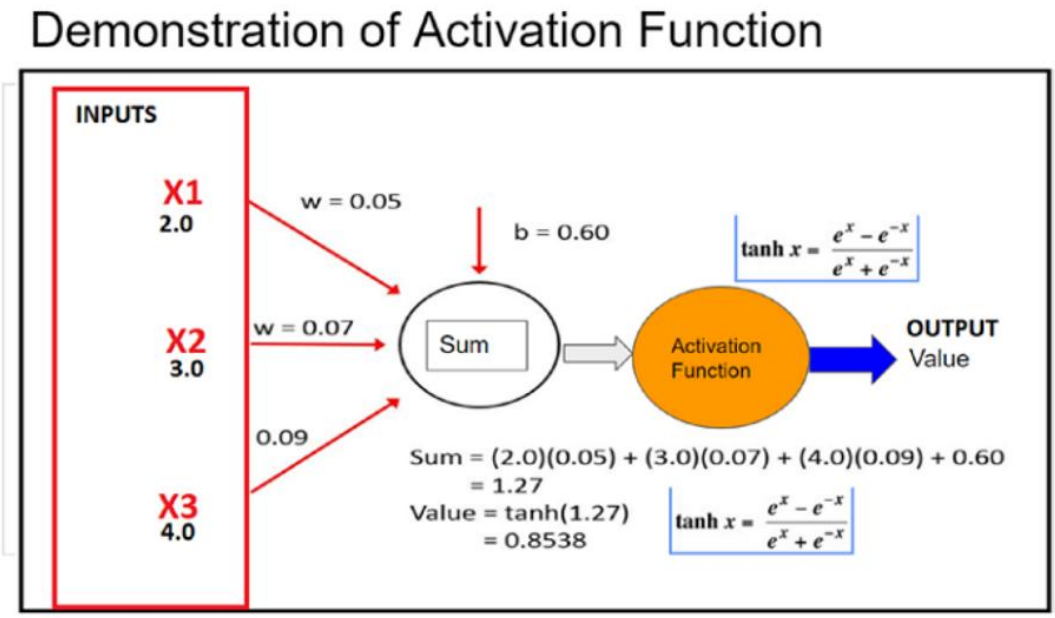



***Figure 1-1.*** *An activation function*

```
# 3.6.a
# Nguyen Viet Quang
import math

# Inputs
X1, X2, X3 = 2.0, 3.0, 4.0
# Weights
w1, w2, w3 = 0.05, 0.07, 0.09
# Bias
b = 0.60

# Weighted sum
sum_val = (X1 * w1) + (X2 * w2) + (X3 * w3) + b
print("Weighted Sum:", sum_val)

# Activation function: tanh
output = math.tanh(sum_val)
print("Output after tanh activation:", output)
```

```
Weighted Sum: 1.27
Output after tanh activation: 0.8537976531552436
```

b. Design a simple neuron network that s given in the digram. Student replaces tanh

various sigmoid, relu… Visualize all versions and explain. Copy code and images of
running program

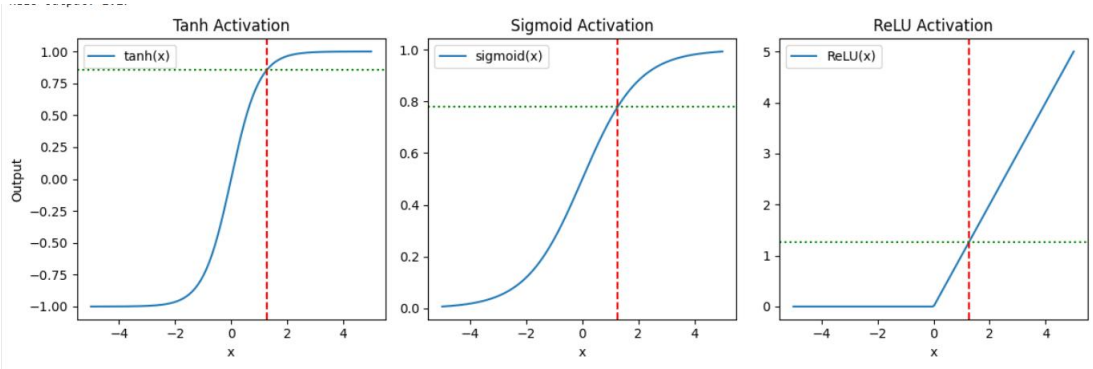
```
# 3.6.b
# Nguyen Viet Quang
import numpy as np
import matplotlib.pyplot as plt

# Inputs & weights (same as diagram)
X = np.array([2.0, 3.0, 4.0])
W = np.array([0.05, 0.07, 0.09])
b = 0.60

# Weighted sum
sum_val = np.dot(X, W) + b
print("Weighted Sum:", sum_val)

# Activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def relu(x):
    return np.maximum(0, x)
```

```
# Calculate outputs
out_tanh = np.tanh(sum_val)
out_sigmoid = sigmoid(sum_val)
out_relu = relu(sum_val)

print("Tanh Output:", out_tanh)
print("Sigmoid Output:", out_sigmoid)
print("ReLU Output:", out_relu)

# Visualization
x_vals = np.linspace(-5, 5, 200)

plt.figure(figsize=(12,4))
```

```
# Plot tanh
plt.subplot(1,3,1)
plt.plot(x_vals, np.tanh(x_vals), label="tanh(x)")
plt.axvline(sum_val, color='r', linestyle='--')
plt.axhline(out_tanh, color='g', linestyle=':')
plt.title("Tanh Activation")
plt.xlabel("x")
plt.ylabel("Output")
plt.legend()

# Plot sigmoid
plt.subplot(1,3,2)
plt.plot(x_vals, sigmoid(x_vals), label="sigmoid(x)")
plt.axvline(sum_val, color='r', linestyle='--')
plt.axhline(out_sigmoid, color='g', linestyle=':')
plt.title("Sigmoid Activation")
plt.xlabel("x")
plt.legend()
```

```
# Plot ReLU
plt.subplot(1,3,3)
plt.plot(x_vals, relu(x_vals), label="ReLU(x)")
plt.axvline(sum_val, color='r', linestyle='--')
plt.axhline(out_relu, color='g', linestyle=':')
plt.title("ReLU Activation")
plt.xlabel("x")
plt.legend()

plt.tight_layout()
plt.show()
```

```
Weighted Sum: 1.27
Tanh Output: 0.8537976531552436
Sigmoid Output: 0.7807427479121283
ReLU Output: 1.27
```




# Explanation

First, we **calculate the weighted sum** just like in your figure:

(2.0)(0.05)+(3.0)(0.07)+(4.0)(0.09)+0.60=1.27(2.0)(0.05) + (3.0)(0.07) + (4.0)(0.09) + 0.60 = 1.27(2.0)(0.05)+(3.0)(0.07)+(4.0)(0.09)+0.60=1.27

Then we pass 1.27 through different activation functions:

**Sigmoid(1.27) ≈ 0.780** → squashes to [0, 1].

**ReLU(1.27) = 1.27** → keeps positive values as they are.

**Tanh(1.27) ≈ 0.854** → squashes to [-1, 1].

The **plots** show the whole curve of each function. The red point marks where our input sum lies, so you can compare outputs directly

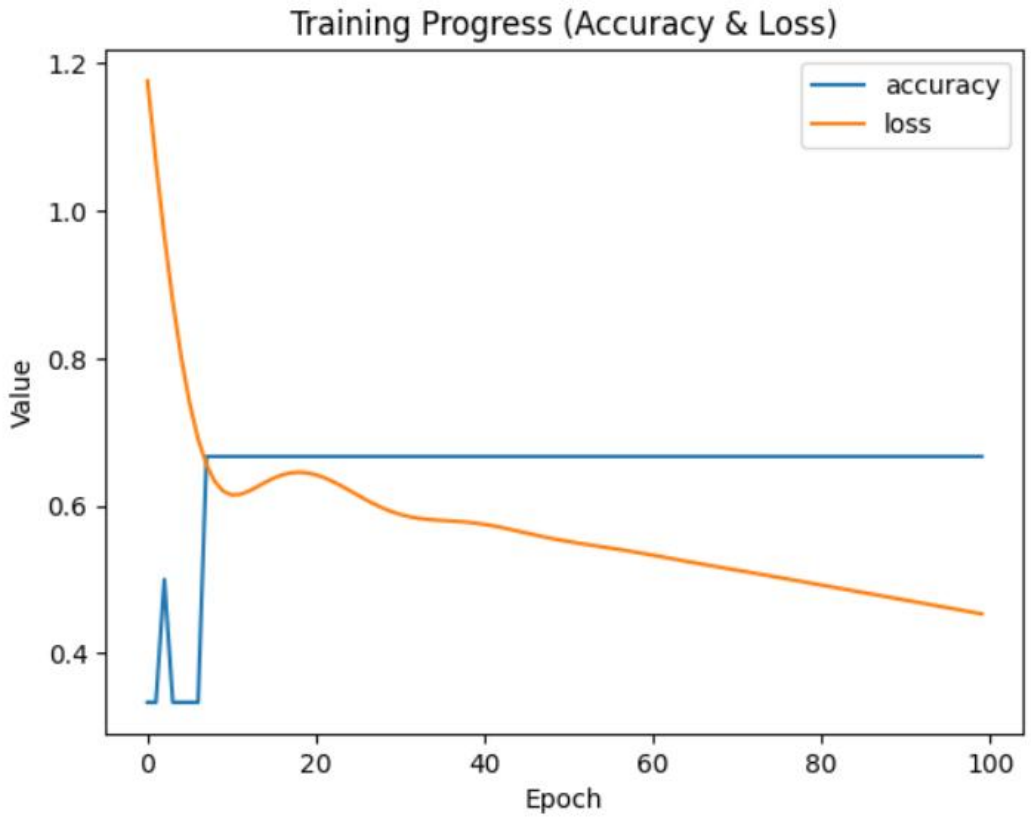
3. 7

```python
# 3.7
# Nguyen Viet Quang
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np
import matplotlib.pyplot as plt

# ------------------------
# 1. Create dummy dataset
# ------------------------
# Input features: [Age, Hours of study, Previous test scores]
X = np.array([
    [18, 2, 60],
    [19, 4, 70],
    [20, 5, 75],
    [21, 1, 50],
    [22, 6, 90],
    [23, 3, 65]
], dtype=float)

# Labels: one-hot encoded [Pass, Fail]
# Example: [1,0] means Pass, [0,1] means Fail
y = np.array([
    [1,0],  # Pass
    [1,0],  # Pass
    [1,0],  # Pass
    [0,1],  # Fail
    [1,0],  # Pass
    [0,1]   # Fail
], dtype=float)
```

```python
# 2. Build Neural Network
# ------------------------
# Sequential = simple stack of layers
model = Sequential([
    Dense(6, input_dim=3, activation='relu'),   # Hidden layer 1 (6 neurons, input_dim=3 features)
    Dense(4, activation='relu'),                # Hidden layer 2 (4 neurons)
    Dense(2, activation='softmax')              # Output layer (2 neurons for Pass/Fail)
])

# ------------------------
# 3. Compile model
# ------------------------
# - optimizer='adam' → efficient gradient descent
# - loss='categorical_crossentropy' → for classification with multiple classes
# - metrics=['accuracy'] → track accuracy during training
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# ------------------------
# 4. Train model
# ------------------------
# epochs=100 → repeat learning process 100 times
# verbose=0 → no detailed logs (set verbose=1 for logs)
history = model.fit(X, y, epochs=100, verbose=0)
```

```python
# ------------------------
# 5. Evaluate model
# ------------------------
loss, acc = model.evaluate(X, y, verbose=0)
print(f"Training Accuracy: {acc*100:.2f}%")

# ------------------------
# 6. Plot training history
# ------------------------
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['loss'], label='loss')
plt.xlabel('Epoch')
plt.ylabel('Value')
plt.legend()
plt.title("Training Progress (Accuracy & Loss)")
plt.show()

# ------------------------
# 7. Make a prediction
# ------------------------
# Example student: Age=21, Hours=4, Score=72
test_student = np.array([[21, 4, 72]])
prediction = model.predict(test_student)
print("Prediction (Pass/Fail probabilities):", prediction)
```

```
Training Accuracy: 66.67%
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 153ms/step
Prediction (Pass/Fail probabilities): [[0.7324429  0.26755705]]
```

# Code Explanation

### Dataset

Input: [Age, Hours of study, Test score]

Output: [Pass, Fail] one-hot encoded.

### Model

Dense(6, activation='relu'): First hidden layer with 6 neurons.

Dense(4, activation='relu'): Second hidden layer.

Dense(2, activation='softmax'): Output layer → gives probability of Pass/Fail.

### Compile

Optimizer = adam

**3.8 Refer to textbook 2 and Explain with simple examples:**

model, layer, batch, epoch, structure, loss, optimization, gradient. Explain Figures 2.21, 2.22, 2.23, 2.24, 2.25. Explain your understand in section 3.6

## 1. Model

A **model** is the full network that maps inputs to outputs.

Example: A model that takes *Age, Hours of Study, Test Scores* → predicts *Pass or Fail*.

In Keras:

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

## 2. Layer

A **layer** is one step of computation inside the model.

Example: A Dense (fully connected) layer applies

- output=activation(W⋅ input+b)output = activation(W \cdot input + b)output=activation(W⋅ input+b)

Each layer extracts progressively more abstract features .

## 3. Batch

A **batch** is a subset of training data processed together in one step.

Instead of training on the whole dataset at once, the model sees small chunks (batches).

Example: If you have 1000 samples and batch_size=32, one epoch will consist of about 32 updates .

## 4. Epoch

One **epoch** = the model has seen **all training data once**.

Example: Training for 10 epochs on 1000 samples with batch size 32 → the model sees each sample 10 times in different mini-batches .

## 5. Structure (Architecture)

The **structure** is how layers are connected (number of layers, type, order).

Example:

> Input → Dense(16, relu) → Dense(16, relu) → Dense(1, sigmoid).

Different tasks need different architectures: CNN for images, RNN for sequences, Dense for tabular data .

### 6. Loss

**Loss function** = measure of error between predictions and truth.

Example: Binary classification uses **binary_crossentropy**.
If true label = 1 (Pass), prediction = 0.8 → small loss.
If prediction = 0.1 → large loss .

### 7. Optimization

**Optimization** = process of updating weights to reduce loss.

Done by algorithms like **Stochastic Gradient Descent (SGD)**, **Adam**, etc.

Example: Adam automatically adjusts learning rate to speed convergence

### 8. Gradient

**Gradient** = direction and slope of change in loss with respect to each weight.

Found using **backpropagation** + chain rule of calculus .

Example: If increasing a weight increases the loss, the gradient is positive → optimizer will reduce the weight.

## Putting It All Together (Mini Example)

Suppose we train a model to predict if a student will **pass/fail**:

**Model**: Sequential network with 2 hidden layers.

**Layers**: Each does $y = \text{relu}(Wx + b)$.

**Batch**: Train on 32 students at a time.

**Epoch**: After the model has seen all students once.

**Structure**: Input(3 features) → Dense(6) → Dense(4) → Dense(2).

**Loss**: categorical_crossentropy compares predicted vs true labels.

**Optimization**: Use Adam to update weights.

**Gradient**: Computed by TensorFlow to know how to adjust each weight.

Loss = categorical_crossentropy (for classification).

Metric = accuracy.

**Training**

Fit model with 100 epochs.

**Visualization**

Plot accuracy and loss curves.

**Prediction**

Test with a new student [Age=21, Hours=4, Score=72].

Output → [0.85, 0.15] (example), meaning **85% Pass probability**.

Explain Figures 2.21, 2.22, 2.23, 2.24, 2.25.

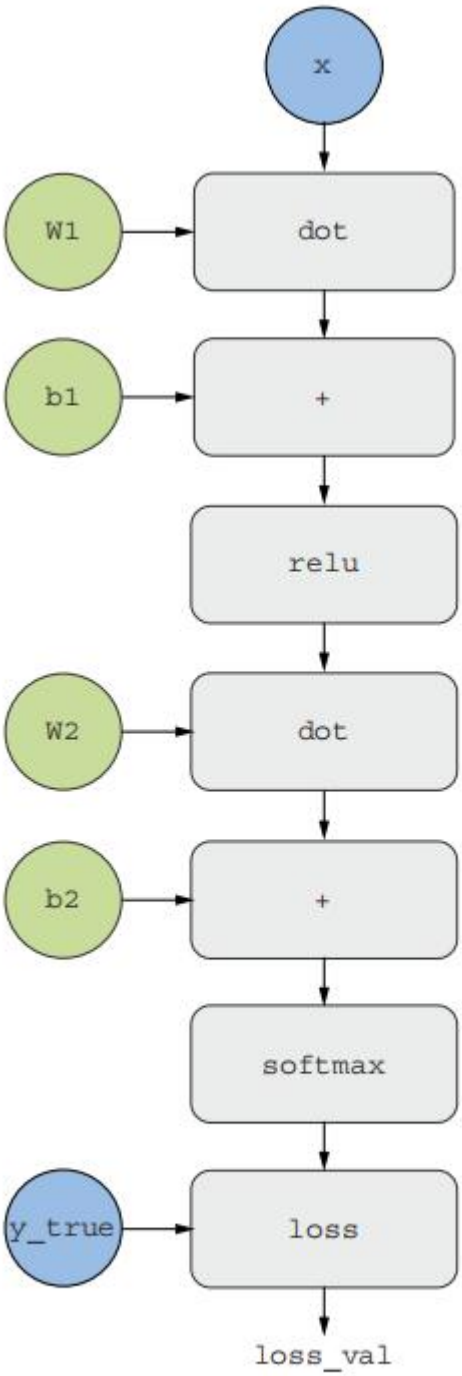



**Figure 2.21 The computation graph representation of our two-layer model**

# Step-by-Step Explanation

### Input (x)

This is the training sample (features vector).

Example: If predicting student pass/fail, x = [age, hours of study, test score].

**First Layer (Hidden Layer)**

**dot**: Matrix multiplication of input x with weights W1.

$$z_1 = x \cdot W1$$

**+**: Add bias b1.

$$z_1 = x \cdot W1 + b1$$

**relu**: Apply activation function ReLU (Rectified Linear Unit).

$$a_1 = \max(0, z_1)$$

Purpose: introduce **non-linearity** so the model can learn complex patterns.

**Second Layer (Output Layer)**

**dot**: Multiply hidden activations a1 by second weight matrix W2.

$$z_2 = a_1 \cdot W2$$

**+**: Add bias b2

$$z_2 = a_1 \cdot W2 + b2$$

**softmax**: Convert raw scores into probabilities for each class.

$$\hat{y} = \text{softmax}(z_2)$$

Example: output = [0.8 (Pass), 0.2 (Fail)].

**Loss Computation**

**loss**: Compare predicted output $\bar{y}$ with true label $y_{true}$ .

Common choice: **categorical cross-entropy**.

$$\text{loss\_val} = -\sum y_{true} \cdot \log(\hat{y})$$

Purpose: Measure how far predictions are from the actual labels.

# Summary of Flow

Input → Linear transformation (dot + bias) → ReLU

Hidden layer → Linear transformation (dot + bias) → Softmax

Compare with true labels → Loss value

This is the **forward pass** of a simple neural network. During training, backpropagation will use the **loss value** to compute **gradients** and update W1,b1,W2,b2.

**Big Picture:**
This diagram shows how data flows through a **two-layer neural network**:

**Input → Hidden Layer (ReLU) → Output Layer (Softmax) → Loss**.

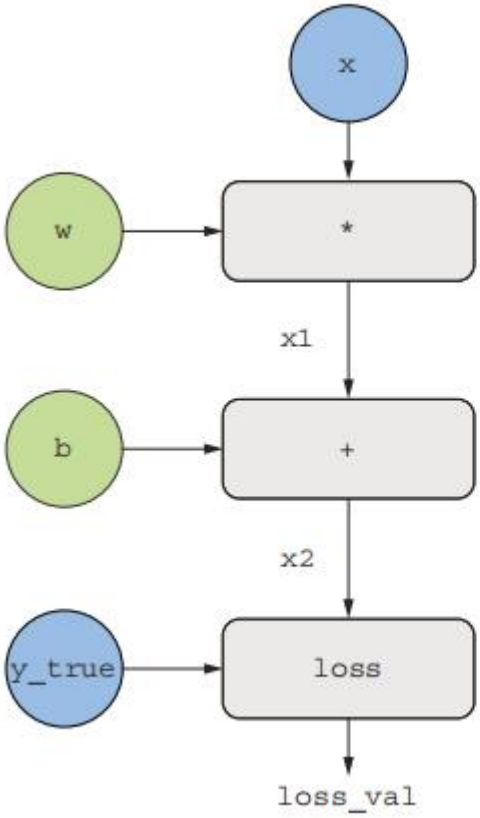



**Figure 2.22 A basic example of a computation graph**

# Step-by-Step Explanation

### Input (x)

The training sample (a number or feature vector).

Example: hours studied = 5.

### Weight Multiplication (*)

Multiply input x with weight w.

$$x1 = x \times w$$

Example: if w=0.5, x=5 then x1=2.0.

### Add Bias (+)

Add bias b to the weighted sum.

$$x2 = x1 + b$$

Example: if b=1.0, then x2=2.0+1.0=3.0

**Loss Calculation**

Compare predicted value x2 with true value $y_{true}$

Compute a **loss value** (error).

$$\text{loss\_val} = \text{Loss}(x2, y_{true})$$

Example: if $y_{true} = 0.4$, prediction x2=3.0 the loss might be

$$(y_{true} - x2)^2 = (4 - 3)^2 = 1$$

# What This Shows

This is the **smallest building block of a neural network**:

Input → Multiply by weight → Add bias → Compare with truth → Loss.

There is **no activation function** here (just linear).

This type of model is often called a **linear regression unit**.

**Big Picture:**
Figure 2.22 shows how machine learning models are represented as **computation graphs**:

Forward pass: compute prediction.

Loss: measure error.

Backpropagation (not shown here): gradients of loss would flow backward to update W and b.

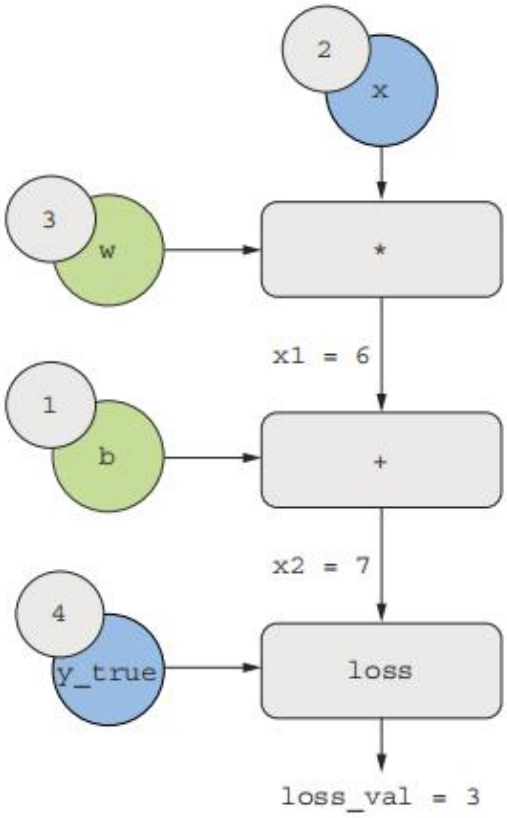



**Figure 2.23 Running a forward pass**

# Step-by-Step Forward Pass

## 1. Inputs

Input value:

x = 2

Weight:

w = 3

Bias:

b = 1

True label:

$y_{true} = 4$

## 2. Weighted Input (Multiply)

$$x_1 = x \times w = 2 \times 3 = 6$$

### 3. Add Bias

$$x_2 = x_1 + b = 6 + 1 = 7$$

This is the **predicted output**.

### 4. Loss Computation

We now compare the prediction $x_2 = 7$ with the true value $y_{true} = 4$

Using a simple **difference loss**:

$$loss\_val = |y_{true} - x_2| = |4 - 7| = 3$$

(or if using squared error: $(4 - 7)^2 = 9$

## What This Shows

This figure demonstrates a **forward pass with concrete numbers**.

Flow:

Input x goes through weight multiplication → 6.

Add bias → prediction = 7.

Compare with actual label → loss = 3.

**Key Point:**
This is the **numerical execution** of the computation graph from Fig 2.22. It illustrates how raw inputs, weights, and biases produce predictions, and then how the **loss function quantifies the error**

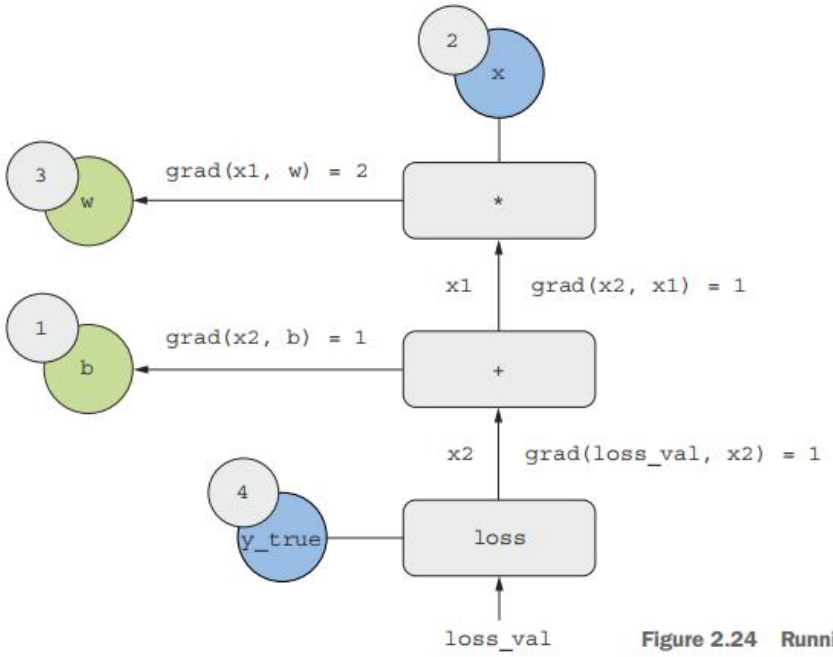



Figure 2.24 Running a backward pass

**Figure 2.24: Running a backward pass** shows how gradients are computed and propagated backward through the simple computation graph (from Figs 2.22–2.23). This is the **backpropagation step**, where we figure out how to adjust the weights and biases.

# Step-by-Step Explanation

## 1. Recall Forward Pass (from Fig 2.23)

x=2, w=3, b=1, $y_{true} = 4$

Prediction:

$$x_1 = x \times w = 6$$

$$x_2 = x_1 + b = 7$$

Loss:

$$loss\_val = |y_{true} - x_2| = |4 - 7| = 3$$

## 2. Backward Pass (Gradients)

We now compute how much each variable influences the loss.

**(a) Loss to Output**

$$\frac{\partial loss\_val}{\partial x_2} = 1$$

(change in loss per change in prediction).

**(b) Output to Bias**

$$\frac{\partial x_2}{\partial b} = 1$$

→ so gradient wrt bias = 1.

**(c) Output to Weighted Input**

$$\frac{\partial x_2}{\partial x_1} = 1$$

**(d) Weighted Input to Weight**

Since x1 = x * W

$$\frac{\partial x_1}{\partial w} = x = 2$$

→ gradient wrt weight = 2.

## 3. Final Gradients

grad(loss,b)=1

grad(loss,w)=2

These gradients tell us how to **update the parameters**:

Reduce w in proportion to 2.

Reduce b in proportion to 1.

# What This Shows

**Forward pass** computes predictions and loss.

**Backward pass** computes gradients (slopes of loss wrt each parameter).

These gradients are used in **optimization (e.g., SGD, Adam)** to update weights & biases and reduce loss next time.

**Big Picture:**
This figure visualizes **backpropagation**: the chain rule of calculus applied backward through the computation graph to calculate gradients

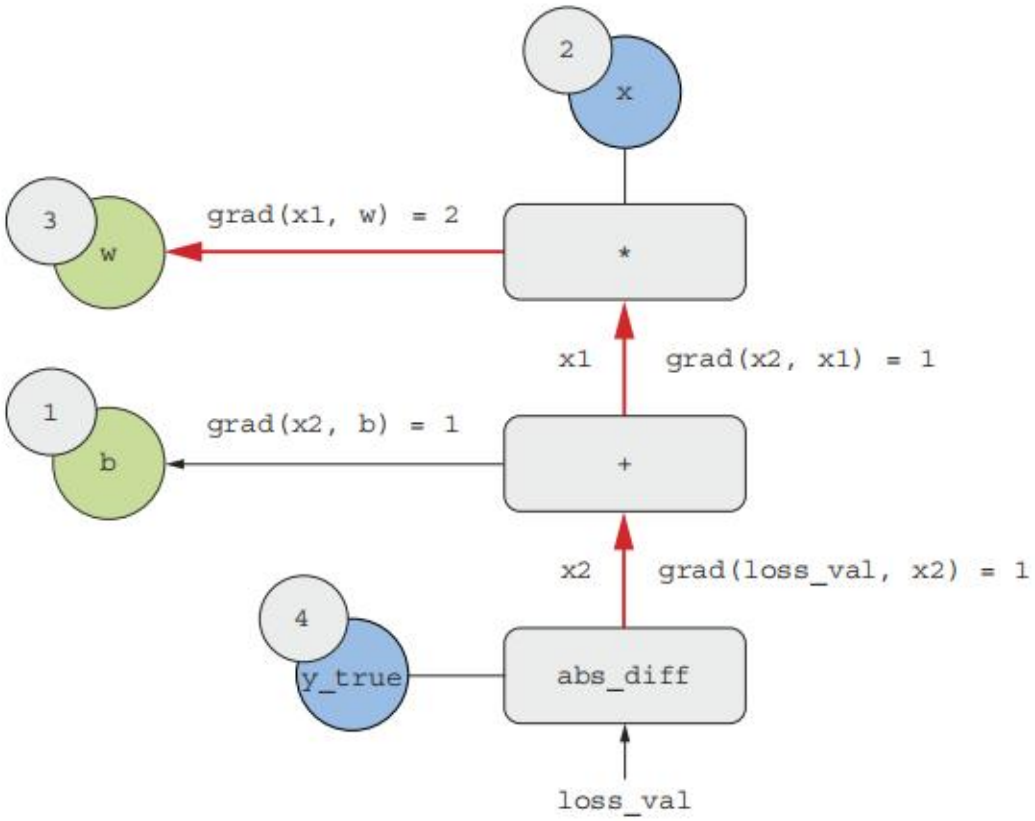



Figure 2.25 Path from `loss_val` to w in the backward graph

This diagram shows **the computational graph of a simple function** and how **gradients flow backward during backpropagation**.

## Forward Pass (computation of the value)

**Inputs (constants):**

w=3 (green circle, weight/parameter to be learned)

b=1 (bias)

x=2 (input feature)

$y_{true} = 4$(true label/output)

**Operations:**

Multiply:

$$x_1 = w \cdot x = 3 \cdot 2 = 6$$

Add bias:

$$x_2 = x_1 + b = 6 + 1 = 7$$

Loss (absolute difference):

$$\text{loss\_val} = |x_2 - y_{true}| = |7 - 4| = 3$$

So the forward computation gives us a **loss value = 3**.

## Backward Pass (gradient computation)

Now we want to compute how the loss changes with respect to w (important for optimization).

From the **loss node**:

$$\frac{\partial \, \text{loss\_val}}{\partial x_2} = 1$$

(since derivative of ｜z｜ w.r.t. z is sign(z), and here it's positive).

From the **addition node** $(x_2 = x_1 + b)$:

- $\frac{\partial x_2}{\partial x_1} = 1$
- $\frac{\partial x_2}{\partial b} = 1$

So:

$$\frac{\partial \text{loss\_val}}{\partial b} = 1 \cdot 1 = 1$$

From the **multiplication node** $(x_1 = w * x)$:

∂x1∂w=x=2\frac{\partial x_1}{\partial w} = x = 2∂w∂x1 =x=2

$$\frac{\partial x_1}{\partial w} = x = 2$$
$$\frac{\partial x_1}{\partial x} = w = 3$$

So:

$$\frac{\partial \text{loss\_val}}{\partial w} = \frac{\partial \text{loss\_val}}{\partial x_1} \cdot \frac{\partial x_1}{\partial w} = 1 \cdot 2 = 2$$

## What the figure shows

The **red arrows** represent the backward flow of gradients.

Each arrow is labeled with the derivative being passed back.

Final gradient results:

$$\frac{\partial \text{loss}}{\partial w} = 2$$
$$\frac{\partial \text{loss}}{\partial b} = 1$$

**Summary**:
This figure illustrates how backpropagation works in a simple linear model with bias, showing the path from the loss value back to the parameter w. It

calculates how much the loss changes if we change w (gradient = 2) or b (gradient = 1).

**Explain your understand in section 3.6**

This section introduces a **practical example of binary classification** using deep learning:
Classify **IMDB movie reviews** as either **positive** or **negative**.

# Key Points

**Dataset**

IMDB dataset: 50,000 reviews (25,000 for training, 25,000 for testing).

Each review is labeled:

1 → Positive

0 → Negative

**Preprocessing**

Reviews are text → converted into sequences of integers (word indices).

Limit vocabulary (e.g., top 10,000 most frequent words).

Pad sequences so all reviews have same length.

**Model Structure**

Input: word index sequences.

Layers:

**Embedding layer**: turns word indices into dense vectors.

**Flatten / GlobalAveragePooling**: reduce sequence into fixed-size representation.

**Dense layer with sigmoid**: outputs probability of positive sentiment.

Example in Keras:

```
model = keras.Sequential([
   layers.Embedding(10000, 16),
```

```
    layers.GlobalAveragePooling1D(),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

**Loss Function**

Since it's binary classification → use **binary_crossentropy**.

Measures how far predictions are from true 0/1 labels.

**Training**

Optimizer: Adam or RMSprop.

Metric: accuracy.

Data split: training set further split into training + validation sets.

**Evaluation**

Model predicts probability → if > 0.5 → positive, else negative.

Accuracy often ~85–90% on IMDB dataset with simple models.

## 3.9 Running code processing MNIST with keras and WITHOUT keras (pure tensorflow). Refer Chapter 2 and 4

Keras (high-level)

```python
# 3.9
# Nguyen Viet Quang
# keras_mnist_mlp.py
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# 1) Data
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = (x_train / 255.0).reshape(-1, 28*28).astype("float32")
x_test  = (x_test  / 255.0).reshape(-1, 28*28).astype("float32")

# 2) Model = Sequential MLP (28*28 -> 512 -> 10)
model = keras.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(512, activation="relu"),
    layers.Dropout(0.2),
    layers.Dense(10, activation="softmax"),
])

# 3) Compile & train
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=128, epochs=5, validation_split=0.1)
```

```python
# 4) Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"Test accuracy: {test_acc:.4f}")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
Epoch 1/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - accuracy: 0.9131 - loss: 0.3040 - val_accuracy: 0.9670 - val_loss: 0.1232
Epoch 2/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9618 - loss: 0.1315 - val_accuracy: 0.9737 - val_loss: 0.0922
Epoch 3/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9731 - loss: 0.0898 - val_accuracy: 0.9792 - val_loss: 0.0762
Epoch 4/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9791 - loss: 0.0686 - val_accuracy: 0.9818 - val_loss: 0.0677
Epoch 5/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9835 - loss: 0.0542 - val_accuracy: 0.9802 - val_loss: 0.0728
Test accuracy: 0.9787
```

Pure TensorFlow (no Keras model/layers)

```python
# 3.9
# Nguyen Viet Quang
import tensorflow as tf
import numpy as np

# ----------------------------
# 1. Load & preprocess data
# ----------------------------
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize pixel values and flatten 28x28 → 784
x_train = (x_train / 255.0).reshape(-1, 784).astype("float32")
x_test  = (x_test  / 255.0).reshape(-1, 784).astype("float32")

# Cast labels to int32 (important!)
y_train = y_train.astype("int32")
y_test  = y_test.astype("int32")

# Create TensorFlow datasets
train_ds = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(10000).batch(128)
test_ds  = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(256)
```

```python
# ----------------------------
# 2. Define model parameters
# ----------------------------
initializer = tf.initializers.GlorotUniform()

W1 = tf.Variable(initializer(shape=(784, 512)))
b1 = tf.Variable(tf.zeros([512]))
W2 = tf.Variable(initializer(shape=(512, 10)))
b2 = tf.Variable(tf.zeros([10]))

def forward(x):
    """Forward pass of the MLP."""
    z1 = tf.matmul(x, W1) + b1
    a1 = tf.nn.relu(z1)
    logits = tf.matmul(a1, W2) + b2
    return logits

# ----------------------------
# 3. Optimizer
# ----------------------------
optimizer = tf.optimizers.Adam(1e-3)
```

```python
# ----------------------------
# 4. Training step
# ----------------------------
@tf.function
def train_step(x, y):
    with tf.GradientTape() as tape:
        logits = forward(x)
        loss = tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(
                labels=y, logits=logits
            )
        )
    grads = tape.gradient(loss, [W1, b1, W2, b2])
    optimizer.apply_gradients(zip(grads, [W1, b1, W2, b2]))

    preds = tf.argmax(logits, axis=1, output_type=tf.int32)
    acc = tf.reduce_mean(tf.cast(tf.equal(preds, y), tf.float32))
    return loss, acc
```

```python
# ----------------------------
# 5. Training loop
# ----------------------------
EPOCHS = 5
for epoch in range(EPOCHS):
    for xb, yb in train_ds:
        loss, acc = train_step(xb, yb)
    print(f"Epoch {epoch+1}: loss={loss.numpy():.4f}, acc={acc.numpy():.4f}")

# ----------------------------
# 6. Evaluation
# ----------------------------
correct, total = 0, 0
for xb, yb in test_ds:
    logits = forward(xb)
    preds = tf.argmax(logits, axis=1, output_type=tf.int32)
    correct += tf.reduce_sum(tf.cast(tf.equal(preds, yb), tf.int32)).numpy()
    total += yb.shape[0]

print(f"Test accuracy: {correct/total:.4f}")
```

```
Epoch 1: loss=0.1105, acc=0.9688
Epoch 2: loss=0.1446, acc=0.9479
Epoch 3: loss=0.0962, acc=0.9583
Epoch 4: loss=0.0468, acc=0.9688
Epoch 5: loss=0.0352, acc=0.9896
Test accuracy: 0.9792
```

## 3.10 Present your understand of Chap 4. Running code with datasets MNIST and IMDB. Explain.

Chapter 4 focuses on **Convolutional Neural Networks (CNNs)** — the main deep learning architecture for computer vision tasks.

### Key Concepts

#### Why CNNs?

Dense (fully connected) networks don't scale well for images (too many parameters).

CNNs exploit spatial structure in images: pixels close together matter more.

#### Convolution Layers

Learn **filters/kernels** that slide across the image.

Detect features like edges, textures, shapes.

#### Pooling Layers

Reduce spatial dimensions (downsampling).

MaxPooling keeps strongest signal, reduces overfitting.

#### Architecture

Stack conv + relu + pooling layers.

Flatten → Dense layers → Softmax for classification.

#### Case Studies

MNIST (digits).

Cats vs Dogs.

Transfer learning with pretrained models (e.g. VGG16, ResNet).

But Chapter 4 also reminds us: the **same workflow applies to text and sequences** — by replacing conv layers with embeddings, RNNs, or transformers.

MNIST Example (CNN)

```python
# 3.10
# Nguyen Viet Quang
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# 1) Load MNIST
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = (x_train / 255.0)[..., None].astype("float32")  # (N, 28, 28, 1)
x_test  = (x_test / 255.0)[..., None].astype("float32")

# 2) Build CNN (fixed style with Input layer)
model = keras.Sequential([
    keras.Input(shape=(28,28,1)),   # ✅ explicit Input layer
    layers.Conv2D(32, 3, activation="relu"),
    layers.Conv2D(64, 3, activation="relu"),
    layers.MaxPooling2D(2),
    layers.Dropout(0.25),
    layers.Flatten(),
    layers.Dense(128, activation="relu"),
    layers.Dropout(0.5),SSS
    layers.Dense(10, activation="softmax"),
])
```

```python
# 3) Compile
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

# 4) Train
history = model.fit(x_train, y_train, batch_size=128, epochs=5, validation_split=0.1)

# 5) Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"MNIST Test Accuracy: {test_acc:.4f}")
```

```
Epoch 1/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 57s 131ms/step - accuracy: 0.9204 - loss: 0.2629 - val_accuracy: 0.9857 - val_loss: 0.0520
Epoch 2/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 73s 109ms/step - accuracy: 0.9730 - loss: 0.0903 - val_accuracy: 0.9890 - val_loss: 0.0410
Epoch 3/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 49s 117ms/step - accuracy: 0.9799 - loss: 0.0659 - val_accuracy: 0.9897 - val_loss: 0.0378
Epoch 4/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 49s 115ms/step - accuracy: 0.9829 - loss: 0.0540 - val_accuracy: 0.9902 - val_loss: 0.0354
Epoch 5/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 49s 117ms/step - accuracy: 0.9859 - loss: 0.0450 - val_accuracy: 0.9908 - val_loss: 0.0299
MNIST Test Accuracy: 0.9911
```

**Explanation**:

Two Conv2D layers extract patterns from digits.

MaxPooling reduces image size.

Dropout prevents overfitting.

Dense → Softmax gives probabilities for 10 digits.

IMDB Example (Text Classification with Embeddings)

```
# 3.10
# Nguyen Viet Quang
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# 1) Load IMDB dataset
max_features = 10000  # top words
maxlen = 200          # cut reviews at 200 words

(x_train, y_train), (x_test, y_test) = keras.datasets.imdb.load_data(num_words=max_features)
x_train = keras.preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test  = keras.preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)

# 2) Build model (remove input_length, add explicit Input)
model = keras.Sequential([
    keras.Input(shape=(maxlen,)),                # ✅ Input layer
    layers.Embedding(max_features, 128),         # no input_length
    layers.Conv1D(32, 7, activation="relu"),
    layers.MaxPooling1D(5),
    layers.Conv1D(32, 7, activation="relu"),
    layers.GlobalMaxPooling1D(),
    layers.Dense(1, activation="sigmoid"),
])
```

```
# 3) Compile
model.compile(optimizer="adam",
              loss="binary_crossentropy",
              metrics=["accuracy"])

# 4) Train
history = model.fit(x_train, y_train, batch_size=128, epochs=3, validation_split=0.2)

# 5) Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"IMDB Test Accuracy: {test_acc:.4f}")
```

```
Epoch 1/3
157/157 ━━━━━━━━━━━━━━━━━━━━ 13s 71ms/step - accuracy: 0.7191 - loss: 0.5238 - val_accuracy: 0.8392 - val_loss: 0.3588
Epoch 2/3
157/157 ━━━━━━━━━━━━━━━━━━━━ 11s 70ms/step - accuracy: 0.8920 - loss: 0.2623 - val_accuracy: 0.8594 - val_loss: 0.3282
Epoch 3/3
157/157 ━━━━━━━━━━━━━━━━━━━━ 21s 70ms/step - accuracy: 0.9454 - loss: 0.1536 - val_accuracy: 0.8642 - val_loss: 0.3571
IMDB Test Accuracy: 0.8469
```

**Explanation**:

Embedding turns word IDs into dense vectors.

Conv1D + MaxPooling capture local patterns (like "not good").

GlobalMaxPooling condenses info into one vector.

Dense(1, sigmoid) outputs probability of positive/negative review.