BÀI TẬP 2:

Mạng nơ-ron đơn tầng, 2 lớp (Tensorflow)

---

1. Xây dựng một mạng nơ-ron đơn tầng mô phỏng phép toán AND:

1.1. Dữ liệu đầu vào và đầu ra:

```
X = tf.constant([[0.0, 0], [0, 1], [1, 0], [1, 1]])
y = tf.constant([[0.0], [0], [0], [1]])
```

Hình. Input và output mong muốn

- **tf.constant**: Hàm này tạo ra một tensor bất biến, nghĩa là giá trị của tensor này không thể thay đổi sau khi được tạo.
- X: Là một tensor được tạo bằng hàm **tf.constant**, chứa các mẫu dữ liệu có kích thước [4, 2], với mỗi hàng là một mẫu, mỗi mẫu có hai đặc trưng, thể hiện các trường hợp trong phép toán AND.
- y: Là một tensor chứ các nhãn mong muốn tương ứng với các mẫu dữ liệu X, có kích thước [4, 1], mỗi hàng chứa một nhãn.

1.2. Khởi tạo trọng số và bias:

```
W = tf.Variable(tf.random.normal([2, 1]))
b = tf.Variable(tf.random.normal([1]))
```

Hình. Khởi tạo trọng số và bias

- **tf.Variable**: Hàm này tạo ra một tensor có thể thay đổi giá trị của nó sau khi được tạo. Do trọng số W và bias b sẽ thay đổi sau quá trình huấn luyện, việc sử dụng hàm này sẽ phù hợp hơn.
- **tf.random.normal**: Hàm này tạo ra các tensor có các phần tử được lấy mẫu ngẫu nhiên từ phân phối chuẩn, với trung bình và độ lệch chuẩn được xác định trước.
- W: Là một biến tensor chứa các trọng số có kích thước [2, 1] được tạo ngẫu nhiên bằng hàm **tf.random.normal.**
- b: Là một biến tensor đại diện cho bias, có kích thước [1] và cũng được tạo bằng hàm **tf.random.normal.**

```
W:
<tf.Variable 'Variable:0' shape=(2, 1) dtype=float32, numpy=
array([[1.2647331],
       [0.2662034]], dtype=float32)>
b:
<tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([-0.44265252], dtype=float32)>
```

Hình. Trọng số W và bias b

1.3. Hàm dự đoán:

```
@tf.function
def predict(X, W, b):
    return tf.nn.sigmoid(tf.matmul(X, W) + b)
```

Hình. Hàm dự đoán dựa trên hàm kích hoạt "sigmoid"

- **tf.function** là một decorator được sử dụng để chỉ định hàm **predict()** được tối ưu hóa bởi Tensorflow, giúp cải thiện hiệu suất khi thực hiện các phép tính. Khi gán **tf.function** cho hàm này, Tensorflow sẽ chuyển đổi hàm đó thành một đồ thị tính toán, giúp tăng độ thực thi.
- Hàm **predict()** nhận đầu vào là:
  + X: Ma trận dữ liệu đầu vào.
  + W: Ma trận trọng số.
  + b: Bias.
- Hàm **tf.matmul(X, W)** thực hiện phép nhân ma trận giữa X và W. Sau khi nhân, kết quả sẽ được cộng thêm b. Để thực hiện, b sẽ được *broadcast* để có cùng kích thước với ma trận.
- Kết quả trả về của hàm được tính bằng hàm **sigmoid** dựa trên kết quả vừa tính được để đưa kết quả luôn nằm trong khoảng (0, 1), phù hợp cho bài toán phân loại nhị phân.

$$sigmoid(u) = \frac{e^u}{e^u + 1}$$

Hình. Hàm kích hoạt sigmoid

```
Y_hat before training:
[[0.39110914]
 [0.4560018 ]
 [0.6946778 ]
 [0.74805844]]
```

Hình. Kết quả hàm predict với các trọng số được tạo ban đầu

1.4. Hàm mất mát:

- Hàm mất mát được định nghĩa là sự khác biệt giữa giá trị thực tế và giá trị dự đoán của mô hình. Mục tiêu của việc huấn luyện là tối thiểu hóa hàm mất mát, nghĩa là làm cho giá trị dự đoán càng gần với giá trị thực tế.

$$L = \frac{1}{m}\sum_{i=1}^{m}(y^i - \hat{y}^i)^2$$

Hình. Hàm mất mát MSE

- Mean Squared Error (MSE): Thường được sử dụng cho bài toán hồi quy. MSE đo lường bình phương của sự khác biệt của giá trị thực và giá trị dự đoán, sau đó tính trung bình cho tất cả các mẫu.

```
@tf.function
def L(y, y_hat):
    return tf.reduce_mean(tf.square(y - y_hat))
```

Hình. Hàm MSE biểu diễn bằng tensorflow

- **tf.reduce_mean**: Là một hàm trong Tensorflow để tính trung bình của các phần tử trong một tensor. Nó cho phép tính trung bình trên toàn bộ tensor hoặc trên một trục cụ thể, giúp giảm chiều của tensor bằng cách tổng hợp các giá trị.
- **tf.square**: Hàm này dùng để tính bình phương của từng phần tử trong một tensor.
- Hàm mất mát trên nhận giá trị đầu vào là: Giá trị dự đoán mong muốn (y) và giá trị thực tế (y_hat). Kết quả trả về của hàm là một giá trị số biểu thị mức độ khác biệt trung bình của các dự đoán so với giá trị thực tế.

```
# Tính hàm mất mát
loss = L(y, y_hat)
print(f"Loss before training: \n{loss}")
```

```
Loss before training: 0.22673895955085754
```

Hình. Tính toán độ mất mát giữa y và kết quả hàm predict ở trên

1.5. Huấn luyện mạng với Gradient Descent:

- Huấn luyện mạng nơ-ron đơn tầng thực chất là quá trình tối ưu hóa các trọng số và bias của mô hình để giảm thiểu độ mất mát.
- Thuật toán thô của hàm như sau:
  + Khởi tạo trọng số W, b.
  + Lặp:
    + Tính y_hat.
    + Tính giá trị hàm lỗi.
    + Tính đạo hàm riêng.
    + Cập nhật W và b theo công thức:

$$W = W - \alpha \frac{\partial L}{\partial W}$$

$$b = b - \alpha \frac{\partial L}{\partial b}$$

- Thuật toán trên được biểu diễn như sau:

```
# Gradient Descent
for epoch in range(500):
    with tf.GradientTape() as t:
        loss = L(y, predict(X, W, b))
    dW, db = t.gradient(loss, sources: [W, b])
    alpha = 0.1
    W.assign_sub(alpha * dW)
    b.assign_sub(alpha * db)
```

Hình. Huấn luyện mạng bằng Gradient Descent

+ Quá trình huấn luyện sẽ được thực hiện 500 epoch. Mỗi epoch là một lượt huấn luyện trên toàn bộ dữ liệu.

+ **tf.GradientTape:** Là một công cụ của Tensorflow để tự động tính toán gradient. Nó theo dõi các phép toán và tính gradient một cách hiệu quả.

+ Qua mỗi epoch, độ mất mát (loss) để được tính lại dựa trên giá trị dự đoán mới sau khi trọng số W và bias b được cập nhật.

+ **t.gradient:** Tính toán gradient của hàm mất mát *loss* với trọng số W và bias b tương ứng với dW và db.

+ alpha: Là tốc độ học, giúp điều chỉnh tốc độ cập nhật của trọng số W và b.

+ Trọng số W được cập nhật giá trị bằng cách trừ đi alpha*dW thông qua hàm **assign_sub**. Hàm này sẽ thực hiện phép trừ và gán giá trị mới. Tương tự với bias b.

1.6. Dự đoán lại sau huấn luyện và xác định nhãn:

- Sau quá trình huấn luyện, trọng số W và bias b được cập nhật như sau:

*Trước khi huấn luyện:

```
W:
<tf.Variable 'Variable:0' shape=(2, 1) dtype=float32, numpy=
array([[1.2647331],
       [0.2662034]], dtype=float32)>
b:
<tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([-0.44265252], dtype=float32)>
```

*Sau khi huấn luyện:

```
W:
<tf.Variable 'Variable:0' shape=(2, 1) dtype=float32, numpy=
array([[1.499189],
       [1.325916]], dtype=float32)>
b:
<tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([-2.3116968], dtype=float32)>
```

Hình. Các giá trị cũ và mới của trọng số W và bias

- Thực hiện dự đoán lại giá trị mới sau khi cập nhật trọng số W và bias b:

```
# Dự đoán lại
y_hat_new = predict(X, W, b)
print(f"Y_hat after training: \n{y_hat_new}")
```

```
Y_hat after training:
[[0.09015886]
 [0.27174628]
 [0.30735636]
 [0.6256052 ]]
```

Hình. Giá trị dự đoán mới

```
# Loss sau training
loss_new = L(y, y_hat_new)
print(f"Loss after training: {loss_new}")
```

```
Loss after training: 0.07915350794792175
```

Hình. Độ mất mát mới

- Cần xác định các nhãn dự đoán cuối cùng sau khi huấn luyện:

```
# Xác định nhãn
predictions = tf.cast(y_hat_new >= 0.5, tf.float32)
print(f"Predictions: \n{predictions}")
```

Hình. Xác định nhãn dự đoán của các mẫu dữ liệu

+ Biểu thức *y_hat_new >= 0.5* tạo ra một tensor chứa các giá trị boolean, dựa trên việc so sánh mỗi phần tử của *y_hat_new* với ngưỡng 0.5. Nếu kết quả lớn hơn bằng 0.5, kết quả sẽ là 'True', ngược lại là 'False'.

+ Hàm **tf.cast** chuyển đổi các giá trị boolean này thành giá trị số thực ('1.0' cho 'True' và '0.0' cho 'False'), biểu diễn nhãn dự đoán của mô hình.

+ Kết quả sau cùng được gán vào biến *predictions*.

```
Predictions:
[[0.]
 [0.]
 [0.]
 [1.]]
```

Hình. Kết quả nhãn dự đoán

2. Làm lại bài 1 với hàm lỗi binary_crossentropy:

2.1. Hàm mất mát:

$$L = -\frac{1}{m}\sum_{i=1}^{m}(y^i \log(\hat{y}^i) + (1-y^i)\log(1-\hat{y}^i))$$

Hình. Hàm mất mát binary_crossentropy

- Binary_crossentropy: Là hàm mất mát phổ biến được sử dụng trong các bài toán nhị phân, nơi đầu ra của mô hình chỉ có thể là một trong hai giá trị.

```python
# Hàm mất mát (Binary_crossentropy)
@tf.function
def L(y, y_hat):
    y_hat = tf.clip_by_value(y_hat, 1e-7, 1 - 1e-7)
    return -tf.reduce_mean(y * tf.math.log(y_hat) + (1 - y) * tf.math.log(1 - y_hat))
```

Hình. Hàm binary_crossentropy biểu diễn bằng tensorflow

- **tf.reduce_mean**: Là một hàm trong Tensorflow để tính trung bình của các phần tử trong một tensor. Nó cho phép tính trung bình trên toàn bộ tensor hoặc trên một trục cụ thể, giúp giảm chiều của tensor bằng cách tổng hợp các giá trị.
- **tf.clip_by_value:** Hàm này sẽ giới hạn giá trị trong một khoảng giá trị cụ thể, giúp cho các giá trị đầu vào cho hàm mất mát luôn nằm trong khoảng an toàn. Đối với bài này, *y_hat* được giới hạn trong khoảng (1e-7, 1 - 1e-7) là do: Nếu *y_hat* bằng 0, log(0) sẽ không xác định và gây ra lỗi. Tương tự, nếu *y_hat* bằng 1, log(1 - y_hat) = log(0) cũng gây ra lỗi.
- **tf.math.log**: Hàm này được sử dụng để tính toán logarithm cơ số e của các phần tử trong một tensor.
- Hàm *L* nhận giá trị đầu vào là nhãn thực tế *y* và nhãn dự đoán *y_hat.* Kết quả đầu ra sẽ là giá trị duy nhất biểu thị mức độ sai số trung bình của dự đoán so với nhãn thực tế. Do kết quả đầu ra luôn âm, nên đảo dấu giá trị trung bình để phù hợp cho việc tối ưu hóa.

```
W:
<tf.Variable 'Variable:0' shape=(2, 1) dtype=float32, numpy=
array([[-1.0684402],
       [ 2.0052438]], dtype=float32)>
b:
<tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([-0.11452741], dtype=float32)>
Y_hat before training:
[[0.47139937]
 [0.8688372 ]
 [0.23451902]
 [0.6947193 ]]
Loss before training: 0.8250840902328491
```

Hình. Trọng số W, bias b, y_hat ban đầu và độ mất mát loss trước khi huấn luyện

2.2. Dự đoán lại sau khi huấn luyện và xác định nhãn:

- Sau khi huấn luyện, trọng số W và bias b được cập nhật như sau:

```
W:
<tf.Variable 'Variable:0' shape=(2, 1) dtype=float32, numpy=
array([[1.9978064],
       [2.2829409]], dtype=float32)>
b:
<tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([-3.4738512], dtype=float32)>
```

Hình. Trọng số W và bias b mới

- Thực hiện dự đoán lại giá trị mới và tính độ mất mát sau khi cập nhật trọng số W và bias b:

```
Y_hat after training:
[[0.03006547]
 [0.23309615]
 [0.18602557]
 [0.69144773]]
Loss after training: 0.21767865121364594
```

Hình. Giá trị dự đoán và độ mất mát mới

- Cần xác định các nhãn cuối cùng sau khi huấn luyện:

```
Predictions:
[[0.]
 [0.]
 [0.]
 [1.]]
```

Hình. Kết quả nhãn dự đoán

3. Phân loại iris (2 lớp) với tensorflow:

3.1. Lấy dữ liệu và xử lí:

```
# Lấy dữ liệu và xử lí
data = pd.read_csv( filepath_or_buffer: "iris.data", header=None).iloc[:100]
data.iloc[:, -1] = np.where(data.iloc[:, -1] == 'Iris-setosa', 0, 1)
print(data)
```

Hình. Lấy dữ liệu và xử lí cột nhãn

- Đọc dữ liệu bằng hàm **read_csv()** từ thư viện **pandas** để lấy dữ liệu từ file "iris.data", tham số "header" = None là để chỉ định rằng dữ liệu không có dòng tiêu đề. Sau đó, sử dụng **iloc[:100]** để giữ lại 100 hàng đầu tiên của dữ liệu sau khi đọc được.

- Dữ liệu sau khi đọc 100 hàng đầu tiên:

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
..	...	...	...	...	...
95	5.7	3.0	4.2	1.2	Iris-versicolor
96	5.7	2.9	4.2	1.3	Iris-versicolor
97	6.2	2.9	4.3	1.3	Iris-versicolor
98	5.1	2.5	3.0	1.1	Iris-versicolor
99	5.7	2.8	4.1	1.3	Iris-versicolor

[100 rows x 5 columns]

Hình. Dữ liệu sau khi đọc và giữ lại 100 hàng đầu tiên

- Sau đó, cột nhãn được chuyển thành các giá trị số để đưa vào mô hình, cụ thể: '0' cho "Iris-setosa" và '1' cho "Iris-versicolor". Tại cột nhãn, hàng **data.iloc[:, -1] = np.where(data.iloc[:, -1] == 'Iris-setosa', 0, 1)** nghĩa là: Xét từng mẫu dữ liệu, nếu tại đó đang có giá trị "Iris-setosa" thì sẽ được thay bằng 0, ngược lại, giá trị đó sẽ được thay bằng 1. Kết quả:

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
..	...	...	...	...	..
95	5.7	3.0	4.2	1.2	1
96	5.7	2.9	4.2	1.3	1
97	6.2	2.9	4.3	1.3	1
98	5.1	2.5	3.0	1.1	1
99	5.7	2.8	4.1	1.3	1

Hình. Dữ liệu sau khi xử lý cột nhãn

3.2. Trộn dữ liệu:

```
# Trộn dữ liệu
data = data.iloc[np.random.permutation(len(data))].reset_index(drop=True)
print(data)
```

Hình. Trộn dữ liệu

- Để tránh hiện tượng quá khớp (overfitting), cần trộn ngẫu nhiên các hàng trong dữ liệu. Theo đó, sử dụng hàm **np.random.permutation()** để trộn các hàng dữ liệu trong tập dữ liệu vừa được xử lí. Cụ thể:

+ len(data): Tính số lượng hàng trong 'data'.

+ np.random.permutation(len(data)): Tạo ra một hoán vị ngẫu nhiên của các chỉ số từ '0' đến 'len(data) - 1'. Đây cũng là các chỉ số hàng trong 'data'.

+ data.iloc[np.random.permutation(len(data))]: Sử dụng kết quả từ phía trên để sắp xếp lại các hàng của 'data' theo thứ tự ngẫu nhiên.

+ Hàm "reset_index(drop=True)": Đặt lại chỉ số hàng sau khi hoán vị và loại bỏ chỉ số cũ.

+ Kết quả sau cùng được gán vào biến "data".

Kết quả:

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	0
1	6.4	2.9	4.3	1.3	1
2	5.1	2.5	3.0	1.1	1
3	5.7	3.0	4.2	1.2	1
4	5.7	4.4	1.5	0.4	0
..	...	...	...	...	..
95	4.4	2.9	1.4	0.2	0
96	5.7	2.8	4.5	1.3	1
97	5.9	3.2	4.8	1.8	1
98	4.7	3.2	1.6	0.2	0
99	5.4	3.7	1.5	0.2	0

[100 rows x 5 columns]

Hình. Dữ liệu sau khi trộn

3.3. Tách dữ liệu:

```
# Tách dữ liệu
X = data.iloc[:, :-1].values.astype('float32')
y = data.iloc[:, -1].values.astype('float32').reshape(-1, 1)

X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2, random_state=42)
print(f"X_train.shape: {X_train.shape}")
print(f"X_test.shape: {X_test.shape}")
print(f"y_train.shape: {y_train.shape}")
print(f"y_test.shape: {y_test.shape}")
```

Hình. Tách dữ liệu huấn luyện và kiểm tra

- Trước khi tách thành các tập huấn luyện và tập kiểm tra, cần tách các đặc trưng và các nhãn, cụ thể:

+ X = data[iloc:, :-1].values.astype('float32'): Chọn từ cột đầu tiên đến trước cột cuối cùng, tương ứng với các cột đặc trưng của mỗi mẫu trong tập dữ liệu. Sau đó, chuyển các mẫu dữ liệu trong dataframe thành một mảng. Tiếp theo, chuyển đổi các giá trị trong mảng sang kiểu dữ liệu 'float32', là kiểu dữ liệu số thực.

+ y = data[iloc:, -1].values.astype('float32').reshape(-1, 1): Chọn cột cuối cùng của tập dữ liệu, tương ứng với cột nhãn của mỗi mẫu trong tập dữ liệu. Các bước sau được trình bày như trên.

- Sau khi đã tách các đặc trưng và nhãn, tiến hành chia dữ liệu thành các tập huấn luyện (80%) và tập kiểm tra (20%) bằng hàm **train_test_split**. Các tham số đầu vào như sau:

+ X, y lần lượt là các dữ liệu cần chia.

+ test_size = 0.2: Xác định kích thước của tập kiểm tra là 20% của toàn bộ dữ liệu, 80% còn lại đưa vào tập huấn luyện.

+ random_state = 42: Đây là một giá trị cố định được sử dụng để đảm bảo việc chia dữ liệu luôn cho ra cùng một kết quả khi chạy ở lần kế tiếp.

Kết quả:

```
X_train.shape: (80, 4)
X_test.shape: (20, 4)
y_train.shape: (80, 1)
y_test.shape: (20, 1)
```

Hình. Kích cỡ của mỗi tập huấn luyện và kiểm tra

3.4. Xây dựng mô hình:

- Đầu tiên, cần khởi tạo ngẫu nhiên trọng số W và bias b.

```
W:
<tf.Variable 'Variable:0' shape=(4, 1) dtype=float32, numpy=
array([[-0.20376006],
       [ 1.162847  ],
       [ 0.07540151],
       [ 1.514738  ]], dtype=float32)>
b:
<tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([-1.1396781], dtype=float32)>
```

Hình. Trọng số W và bias b

- Tiếp theo, cần xác định hàm dự đoán kết quả đầu ra và hàm mất mát.

```
@tf.function
def predict(X, W, b):
    return tf.nn.sigmoid(tf.matmul(X, W) + b)
```

Hình. Hàm dự đoán

```
@tf.function
def L(y, y_hat):
    y_hat = tf.clip_by_value(y_hat, 1e-7, 1 - 1e-7)
    return - tf.reduce_mean(y * tf.math.log(y_hat) + (1 - y) * tf.math.log(1 - y_hat))
```

Hình. Hàm mất mát

3.5. Huấn luyện mô hình:

```
# Huấn luyện mô hình
alpha = 0.01
epochs = 150
batch_size = 32

train_data = tf.data.Dataset.from_tensor_slices((X_train, y_train)).batch(batch_size)

arr_acc = [] # Lưu accuracy của từng epoch
arr_loss = [] # Lưu loss của từng epoch

for epoch in range(epochs):
    for batch_X, batch_y in train_data:
        with tf.GradientTape() as t:
            loss = L(batch_y, predict(batch_X, W, b))
        dW, db = t.gradient(loss, sources: [W, b])
        W.assign_sub(alpha * dW)
        b.assign_sub(alpha * db)

    train_loss = L(y_train, predict(X_train, W, b))

    train_y_hat = predict(X_train, W, b)
    train_predictions = tf.cast(train_y_hat >= 0.5, tf.float32)
    train_accuracy = tf.reduce_mean(tf.cast(train_predictions == y_train, tf.float32))

    arr_acc.append(train_accuracy.numpy())
    arr_loss.append(train_loss.numpy())
```

Hình. Quá trình huấn luyện mô hình

Quá trình huấn luyện mô hình được mô tả như sau:

- alpha: Là tốc độ học, giúp điều chỉnh tốc độ cập nhật của trọng số W và b.
- epochs: Số lần lặp trên toàn bộ tập dữ liệu.
- batch_size: Kích thước của mỗi batch.
- Dòng 'train_data = tf.data.Dataset.from_tensor_slices((X_train, y_train)).batch(batch_size)' sẽ tạo ra một Dataset từ các tensor X_train và y_train, trong đó mỗi phần tử là một cặp (X_train[i], y_train[i]). Sau đó chia Dataset này thành các batch nhỏ, mỗi batch chứa *batch_size* mẫu.
- arr_acc và arr_loss sẽ lưu trữ kết quả độ chính xác (accuracy) và độ mất mát (loss) trong mỗi lần lặp.
- Với mỗi lần lặp (epoch):

+ Lặp qua từng batch của tập dữ liệu huấn luyện *train_data*, thực hiện cập nhật các trọng số W và bias b.

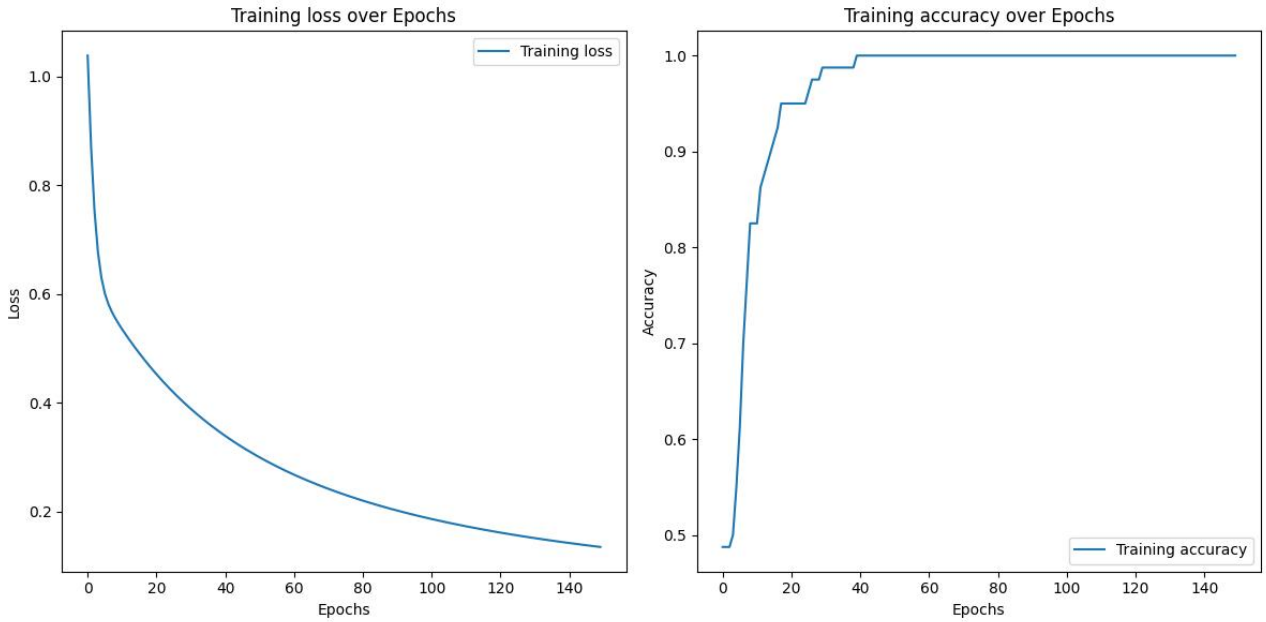
+ Tính toán lại giá trị mất mát cho toàn bộ X_train sau khi các trọng số W và bias b được cập nhật sau từng batch.

+ Dự đoán giá trị đầu ra cho X_train với các trọng số W và bias b đã được cập nhật.

+ Chuyển đổi các dự đoán thành các giá trị nhị phân bằng cách kiểm tra xem chúng có lớn hơn 0.5 không.

+ Tính toán độ chính xác bằng cách so sánh các giá trị nhị phân với giá trị y_train.

+ Chuyển độ mất mát và độ chính xác của từng epoch về kiểu numpy và đưa vào arr_loss, arr_acc.



Hình. Biểu đồ quá trình huấn luyện

3.6. Đánh giá mô hình:

```
# Đánh giá mô hình
y_hat_test = predict(X_test, W, b)
print(f"Y_hat test: {y_hat_test.numpy()}")

predictions = tf.cast(y_hat_test >= 0.5, tf.float32)

TP = tf.reduce_sum(tf.cast((predictions == 0) & (y_test == 0), tf.float32))
TN = tf.reduce_sum(tf.cast((predictions == 1) & (y_test == 1), tf.float32))
FP = tf.reduce_sum(tf.cast((predictions == 0) & (y_test == 1), tf.float32))
FN = tf.reduce_sum(tf.cast((predictions == 1) & (y_test == 0), tf.float32))

precision = (TP / (TP + FP + 1e-7)).numpy()
recall = (TP / (TP + FN + 1e-7)).numpy()
f1 = 2 * (precision * recall) / (precision + recall + 1e-7)
accuracy = ((TP + TN) / (TP + TN + FP + FN + 1e-7)).numpy()
confusion_matrix = tf.convert_to_tensor([[TP, FN], [FP, TN]], dtype=tf.float32)

print(f"Test precision: {precision * 100:.2f}%")
print(f"Test recall: {recall * 100:.2f}%")
print(f"Test accuracy: {accuracy * 100:.2f}%")
print(f"Test f1: {f1 * 100:.2f}%")
print(f"Confusion matrix:\n{confusion_matrix.numpy()}")
```

Hình. Đánh giá mô hình

- Dự đoán kết quả trên tập kiểm tra X_test với các trọng số W và bias b sau khi huấn luyện.
- Chuyển đổi các giá trị trên kết quả vừa dự đoán thành nhãn nhị phân dựa trên ngưỡng 0.5.
- Tính các giá trị True Positives (TP), True Negatives (TN), False Positives (FP) và False Negatives (FN):
  + TP: Số mẫu được dự đoán đúng là “Iris-setosa”.
  + TN: Số mẫu được dự đoán đúng là “Iris-versicolor”.
  + FP: Số mẫu được dự đoán sai là “Iris-setosa”, nhưng thực tế là “Iris-versicolor”.
  + FN: Số mẫu được dự đoán sai là “Iris-versicolor”, nhưng thực tế là “Iris-setosa”.
- Tính các chỉ số đánh giá:

+ Precision: Tỉ lệ giữa số lượng mẫu dự đoán đúng là “Iris-setosa” trong tất cả các mẫu được dự đoán là “Iris-setosa”.

+ Recall: Tỉ lệ giữa số lượng mẫu dự đoán đúng là “Iris-setosa” trong tất cả các mẫu thực tế là “Iris-setosa”.

+ F1-score: Trung bình điều hòa của Precision và Recall.

+ Accuracy: Tỷ lệ giữa tổng số dự đoán đúng trong tổng số mẫu.

- Tạo ma trận nhầm lẫn dựa trên TP, TN, FP, FN.

```
Test precision: 100.00%
Test recall: 100.00%
Test accuracy: 100.00%
Test f1: 100.00%
Confusion matrix:
[[ 9.  0.]
 [ 0. 11.]]
```

Hình. Kết quả trên tập dữ liệu kiểm tra

4. Code đầy đủ của các bài tập:

4.1. Các thư viện cần thiết:

Tên thư viện	Phiên bản
tensorflow	2.16.1
pandas	2.1.4
numpy	1.26.2
matplotlib	3.8.0
sklearn	0.0.post10

4.2. Mạng nơ-ron đơn tầng mô phỏng phép AND:

```python
import tensorflow as tf


# Hàm sigmoid
@tf.function
def predict(X, W, b):
    return tf.nn.sigmoid(tf.matmul(X, W) + b)


# Hàm mất mát (MSE)
@tf.function
def L(y, y_hat):
    return tf.reduce_mean(tf.square(y - y_hat))


# Dữ liệu đầu vào và đầu ra mong muốn
X = tf.constant([[0.0, 0], [0, 1], [1, 0], [1, 1]])
y = tf.constant([[0.0], [0], [0], [1]])

# Khởi tạo trọng số và bias
W = tf.Variable(tf.random.normal([2, 1]))
b = tf.Variable(tf.random.normal([1]))

print(f"W:\n{W}")
print(f"b:\n{b}")

# Dự đoán ban đầu
y_hat = predict(X, W, b)
print(f"Y_hat before training: \n{y_hat}")

# Tính hàm mất mát
loss = L(y, y_hat)
print(f"Loss before training: {loss}")
```

```python
# Gradient Descent
for epoch in range(500):
    with tf.GradientTape() as t:
        loss = L(y, predict(X, W, b))
    dW, db = t.gradient(loss, sources: [W, b])
    alpha = 0.1
    W.assign_sub(alpha * dW)
    b.assign_sub(alpha * db)

# Trọng số W và bias b sau training
print(f"W:\n{W}")
print(f"b:\n{b}")

# Dự đoán lại
y_hat_new = predict(X, W, b)
print(f"Y_hat after training: \n{y_hat_new}")

# Loss sau training
loss_new = L(y, y_hat_new)
print(f"Loss after training: {loss_new}")

# Xác định nhãn
predictions = tf.cast(y_hat_new >= 0.5, tf.float32)
print(f"Predictions: \n{predictions}")
```

4.3. Hàm lỗi binary_crossentropy:

```python
import tensorflow as tf


# Hàm sigmoid
@tf.function
def predict(X, W, b):
    return tf.nn.sigmoid(tf.matmul(X, W) + b)


# Hàm mất mát (Binary_crossentropy)
@tf.function
def L(y, y_hat):
    y_hat = tf.clip_by_value(y_hat, 1e-7, 1 - 1e-7)
    return -tf.reduce_mean(y * tf.math.log(y_hat) + (1 - y) * tf.math.log(1 - y_hat))


# Dữ liệu đầu vào và đầu ra mong muốn
X = tf.constant([[0.0, 0], [0, 1], [1, 0], [1, 1]])
y = tf.constant([[0.0], [0], [0], [1]])

# Khởi tạo trọng số và bias
W = tf.Variable(tf.random.normal([2, 1]))
b = tf.Variable(tf.random.normal([1]))

print(f"W:\n{W}")
print(f"b:\n{b}")

# Dự đoán ban đầu
y_hat = predict(X, W, b)
print(f"Y_hat before training: \n{y_hat}")

# Tính hàm mất mát
loss = L(y, y_hat)
print(f"Loss before training: {loss}")
```

```python
# Gradient Descent
for epoch in range(500):
    with tf.GradientTape() as t:
        loss = L(y, predict(X, W, b))
    dW, db = t.gradient(loss, sources: [W, b])
    alpha = 0.1
    W.assign_sub(alpha * dW)
    b.assign_sub(alpha * db)

# Trọng số W và bias b sau training
print(f"W:\n{W}")
print(f"b:\n{b}")

# Dự đoán lại
y_hat_new = predict(X, W, b)
print(f"Y_hat after training: \n{y_hat_new}")

# Loss sau training
loss_new = L(y, y_hat_new)
print(f"Loss after training: {loss_new}")

# Xác định nhãn
predictions = tf.cast(y_hat_new >= 0.5, tf.float32)
print(f"Predictions: \n{predictions}")
```

4.4. Phân loại iris (2 lớp) với tensorflow:

```python
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

# Lấy dữ liệu và xử lí
data = pd.read_csv( filepath_or_buffer: "iris.data", header=None).iloc[:100]
data.iloc[:, -1] = np.where(data.iloc[:, -1] == 'Iris-setosa', 0, 1)
# print(data)

# Trộn dữ liệu
data = data.iloc[np.random.permutation(len(data))].reset_index(drop=True)
# print(data)

# Tách dữ liệu
X = data.iloc[:, :-1].values.astype('float32')
y = data.iloc[:, -1].values.astype('float32').reshape(-1, 1)

X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2, random_state=42)
print(f"X_train.shape: {X_train.shape}")
print(f"X_test.shape: {X_test.shape}")
print(f"y_train.shape: {y_train.shape}")
print(f"y_test.shape: {y_test.shape}")

# Xây dựng mô hình
W = tf.Variable(tf.random.normal([4, 1]))
b = tf.Variable(tf.random.normal([1]))

print(f"W:\n{W}")
print(f"b:\n{b}")

@tf.function
def predict(X, W, b):
    return tf.nn.sigmoid(tf.matmul(X, W) + b)
```

```python
@tf.function
def L(y, y_hat):
    y_hat = tf.clip_by_value(y_hat, 1e-7, 1 - 1e-7)
    return - tf.reduce_mean(y * tf.math.log(y_hat) + (1 - y) * tf.math.log(1 - y_hat))


# Huấn luyện mô hình
alpha = 0.01
epochs = 150
batch_size = 32

train_data = tf.data.Dataset.from_tensor_slices((X_train, y_train)).batch(batch_size)

arr_acc = [] # Lưu accuracy của từng epoch
arr_loss = [] # Lưu loss của từng epoch

for epoch in range(epochs):
    for batch_X, batch_y in train_data:
        with tf.GradientTape() as t:
            loss = L(batch_y, predict(batch_X, W, b))
        dW, db = t.gradient(loss, sources: [W, b])
        W.assign_sub(alpha * dW)
        b.assign_sub(alpha * db)

    train_loss = L(y_train, predict(X_train, W, b))

    train_y_hat = predict(X_train, W, b)
    train_predictions = tf.cast(train_y_hat >= 0.5, tf.float32)
    train_accuracy = tf.reduce_mean(tf.cast(train_predictions == y_train, tf.float32))

    arr_acc.append(train_accuracy.numpy())
    arr_loss.append(train_loss.numpy())
```

```
plt.figure(figsize=(12, 6))

plt.subplot( *args: 1, 2, 1)
plt.plot( *args: range(epochs), arr_loss, label='Training loss')
plt.title("Training loss over Epochs")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()

plt.subplot( *args: 1, 2, 2)
plt.plot( *args: range(epochs), arr_acc, label='Training accuracy')
plt.title("Training accuracy over Epochs")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()

plt.tight_layout()
plt.show()
# Đánh giá mô hình
y_hat_test = predict(X_test, W, b)
print(f"Y_hat test: {y_hat_test.numpy()}")

predictions = tf.cast(y_hat_test >= 0.5, tf.float32)

TP = tf.reduce_sum(tf.cast((predictions == 0) & (y_test == 0), tf.float32))
TN = tf.reduce_sum(tf.cast((predictions == 1) & (y_test == 1), tf.float32))
FP = tf.reduce_sum(tf.cast((predictions == 0) & (y_test == 1), tf.float32))
FN = tf.reduce_sum(tf.cast((predictions == 1) & (y_test == 0), tf.float32))

precision = (TP / (TP + FP + 1e-7)).numpy()
recall = (TP / (TP + FN + 1e-7)).numpy()
f1 = 2 * (precision * recall) / (precision + recall + 1e-7)
accuracy = ((TP + TN) / (TP + TN + FP + FN + 1e-7)).numpy()
confusion_matrix = tf.convert_to_tensor([[TP, FN], [FP, TN]], dtype=tf.float32)

print(f"Test precision: {precision * 100:.2f}%")
print(f"Test recall: {recall * 100:.2f}%")
print(f"Test accuracy: {accuracy * 100:.2f}%")
print(f"Test f1: {f1 * 100:.2f}%")
print(f"Confusion matrix:\n{confusion_matrix.numpy()}")
```