BÀI TẬP 3

Mạng nơ ron đơn tầng, đa lớp (thực hành 1 + 2 slide 4)

---

1. Xây dựng mạng nơ ron đơn tầng, 3 lớp với dữ liệu huấn luyện như sau:

1.1. Dữ liệu đầu vào và đầu ra:

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

y = tf.constant([[1.0, 0, 0],
                 [0, 1, 0],
                 [0, 1, 0],
                 [0, 0, 1]])
```

Hình. Dữ liệu huấn luyện

- **tf.constant**: Hàm này tạo ra một tensor không thể thay đổi giá trị sau khi được khởi tạo. Trong đồ thị tính toán (tf.Tensor), **tf.constant** được xem là một hằng, nghĩa là giá trị sau khi khởi tạo là bất biến. X và y là dữ liệu đầu vào và đầu ra mong muốn, không có sự thay đổi giá trị nên sử dụng hàm **tf.constant** là hợp lý.
- 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.
- y: Là một tensor được tạo bằng hàm **tf.constant**, chứa các mẫu dữ liệu có kích thước [4, 3], với mỗi hàng là một mẫu, mỗi mẫu được biểu diễn dưới dạng one-hot encoding.

1.2. Khởi tạo trọng số W và bias b:

```
# Khởi tạo W và b
W = tf.Variable(tf.random.normal((2, 3)))
b = tf.Variable([0.0, 0.0, 0.0])
```

Hình. Khởi tạo W và b

- **tf.Variable**: Hàm này tạo ra một tensor có thể thay đổi giá trị sau khi được khởi tạo. Trong đồ thị tính toán (tf.Tensor), **tf.Variable** được xem là một biến, nghĩa là giá trị sau khi khởi tạo có thể thay đổi, nên việc sử dụng hàm này với trọng số W và bias b là hợp lý.
- **tf.random.normal**: Hàm này tạo ra các tensor có các phần tử được lấy ngẫu nhiên theo kích thước được truyền vào theo phân phối chuẩn.
- W: Là một biến tensor chứ các trọng số có kích thước [2, 3] (gồm 2 thuộc tính và 3 lớp) được tạo ngẫu nhiên bằng hàm **tf.random.normal**.
- b: Là một biến tensor đại diện cho bias được tạo ngẫu nhiên các giá trị 0.

```
W:
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[ 0.2663834 ,  0.40988106, -0.8331785 ],
       [-0.0642236 , -0.27561954, -0.74230015]], dtype=float32)>
b:
<tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([0., 0., 0.], dtype=float32)>
```

Hình. Trọng số W và bias b trước khi huấn luyện

1.3. Hàm dự đoán:

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

Hình. Hàm dự đoán

- **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 **softmax** dựa trên kết quả vừa tính được để trả về xác suất dự đoán của mỗi lớp, phù hợp cho bài toán phân loại đa lớp.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

Hình. Hàm kích hoạt softmax

```
Y_hat before traning:
[[0.33333334 0.33333334 0.33333334]
 [0.43158403 0.3493472  0.21906877]
 [0.40203896 0.46407527 0.13388577]
 [0.47542247 0.4442136  0.08036393]]
```

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

1.4. Hàm mất mát:

$$L = -\frac{1}{m}\sum_{i=1}^{m}\left(\sum_{j=1}^{k} y_j^{(i)} * log\left(\hat{y}_j^{(i)}\right)\right)$$

Hình. Hàm mất mát Categorical_crossentropy

- Categorical_crossentropy: Là một hàm mất mát phổ biến được sử dụng trong các mô hình phân loại đa lớp. Hàm đo lường sự khác biệt giữa phân phối xác suất dự đoán bởi mô hình và phân phối xác suất mục tiêu.

```python
# Hàm mất mát (Categorical Crossentropy)
@tf.function
def L(y, y_hat):
    return -tf.reduce_mean(tf.reduce_sum(y*tf.math.log(y_hat), axis=1))
```

Hình. Hàm categorical_crossentropy biểu diễn bằng tensorflow

- **tf.reduce_mean**: Là một hàm trong Tensorflow để tính trung bình 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.reduce_sum**: Là một hàm trong Tensorflow để tính tổng các phần tử trong một tensor dọc theo các trục được chỉ định hoặc trên tất cả phần tử của 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 traning:
1.3597999811172485
```

Hình. Tính toán độ mất mát trước khi huấn luyện

1.5. Huấn luyện mô hình:

- 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 được biểu diễn như sau:

```
# Gradient Descent
alpha = 0.1
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])
    W.assign_sub(alpha * dW)
    b.assign_sub(alpha * db)
```

Hình. Huấn luyện mạng bằng Gradient Descent

+ Mô hình sẽ được huấn luyện qua 500 epoches. Việc sử dụng nhiều epoch giúp mô hình học được đầy đủ từ dữ liệu, tối ưu hóa các trọng số, giảm thiểu độ mất mát, và đạt được độ chính xác cao.

+ **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 khi huấn luyện và xác định nhãn:

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

```
W:
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[-2.367984  ,  0.2062533 ,  2.004816  ],
       [-2.6867688 , -0.10203464,  1.7066582 ]], dtype=float32)>
b:
<tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([ 1.4998614,  0.7016405, -2.2015007], dtype=float32)>
```

Hình. Giá trị mới của trọng số W và bias b

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

```
Y_hat after training:
[[0.6780493  0.3052097  0.01674095]
 [0.11152696 0.6656604  0.22281267]
 [0.11282441 0.66637266 0.22080298]
 [0.00420743 0.32950836 0.66628426]]
```

Hình. Giá trị dự đoán mới

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

```
Loss after training:
0.4018639922142029
```

Hình. Độ mất mát mới

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

```
# Determine labels
predictions = tf.one_hot(tf.argmax(y_hat_new, axis=1), depth=3)
print(f"Predictions:\n{predictions}")
```

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

+ **tf.argmax**: Hàm này được sử dụng để tìm chỉ số của phần tử có giá trị lớn nhất trong tensor dọc theo ‘axis=1’, nghĩa là theo tìm theo hàng, tương ứng với từng kết quả dự đoán của mỗi mẫu dữ liệu.

+ **tf.one_hot**: Hàm này dùng để chuyển đổi các nhãn phân loại từ dạng số nguyên sang dạng mã hóa one-hot, nghĩa là mỗi nhãn được biểu diễn dưới dạng một vector chỉ có một phần tử là '1' và các phần tử còn lại là '0'. Tham số *depth* thể hiện số lượng lớp trong dữ liệu. Đây là độ dài của mỗi vector one-hot.

+ Kết quả cuối cùng chứa các vector one-hot đại diện cho nhãn dự đoán của mô hình cho mỗi mẫu trong dữ liệu, được lưu vào biến *predictions*.

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

Hình. Kết quả nhãn dự đoán cuối cùng

Kết quả trên có thể được giải thích như sau: Từ kết quả dự đoán sau khi huấn luyện *y_hat_new*:

- Đưa *y_hat_new* vào hàm **tf.argmax** để tìm chỉ số của xác suất lớn nhất của từng mẫu, nên kết quả là một con số, cụ thể:

+ Mẫu 1: [0.6780493. 0.3052097, 0.01674095] cho thấy lớp 0 có xác suất lớn nhất (0.6780493), nên ở dòng này, kết quả là chỉ số: 0.

+ Mẫu 2: [0.11152696, 0.6656604, 0.22281267] cho thấy lớp 1 có xác suất lớn nhất (0.6656604), nên ở dòng này, kết quả là chỉ số 1.

+ Mẫu 3: [0.11282441, 0.66637266, 0.22080298] cho thấy lớp 1 có xác suất lớn nhất (0.66637266), nên ở dòng này, kết quả là chỉ số 1.

+ Mẫu 4: [0.00420743, 0.32950836, 0.66628426] cho thấy lớp 2 có xác suất lớn nhất (0.66628426), nên ở dòng này, kết quả là chỉ số 2.

+ Kết quả của **tf.argmax** sẽ là: [0, 1, 1, 2].

- Đưa kết quả của hàm **tf.argmax** vào hàm **tf.one_hot** với *depth=3,* nghĩa là mỗi vector one-hot sẽ có 3 phần tử, tương ứng với 3 lớp của dữ liệu, trong đó:

+ Phần tử '0' mang ý nghĩa đại diện cho lớp '0', nên sau khi one_hot, phần tử '0' sẽ được thay bằng [1, 0, 0].

+ Phần tử ‘1’ mang ý nghĩa đại diện cho lớp ‘1’, nên sau khi one_hot, phần tử ‘1’ sẽ được thay bằng [0, 1, 0].

+ Phần tử ‘2’ mang ý nghĩa đại diện cho lớp ‘2’, nên sau khi one_hot, phần tử ‘2’ sẽ được thay bằng [0, 0, 1].

+ Giá trị cuối cùng khi các vector one_hot đã thay thế các chỉ số tương ứng được gán vào biến *predictions* với kết quả cụ thể như hình phía trên.

2. Phân loại iris 3 lớp:

2.1. Lấy dữ liệu:

```
# Lấy dữ liệu và xử lý
iris = load_iris()
print(iris)

data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['label'] = iris.target
print(data)
```

Hình. Lấy dữ liệu

- Đầu tiên, gọi hàm **load_iris** để tải dữ liệu iris, dữ liệu này sẽ được lưu vào biến iris.
- Chuyển dữ liệu *iris.data* thành DataFrame của Pandas với các cột được đặt tên theo các đặc trưng (sepal length, sepal width, petal lenth, petal width) để dễ quan sát.
- Thêm một cột mới có tên *label* vào DataFrame *data*. Cột này sẽ chứ các nhãn tương ứng từ *iris.target*.

	sepal length (cm)	sepal width (cm)	...	petal width (cm)	label
0	5.1	3.5	...	0.2	0
1	4.9	3.0	...	0.2	0
2	4.7	3.2	...	0.2	0
3	4.6	3.1	...	0.2	0
4	5.0	3.6	...	0.2	0
..	...	...	...	...	...
145	6.7	3.0	...	2.3	2
146	6.3	2.5	...	1.9	2
147	6.5	3.0	...	2.0	2
148	6.2	3.4	...	2.3	2
149	5.9	3.0	...	1.8	2

Hình. Dữ liệu sau khi tải về

2.2. Trộn dữ liệu:

```
# Trộn dữ liệu
data = shuffle( *arrays: data, random_state=42)
print(data)
```

Hình. Trộn dữ liệu

- Thực hiện trộn dữ liệu ngẫu nhiên các hàng trong DataFrame *data* bằng hàm **shuffle** được lấy từ thư viện **sklearn** để giúp dữ liệu huấn luyện không bị lệch số lượng đáng kể các nhãn.
- Dữ liệu sau khi trộn được gán lại vào *data*.

	sepal length (cm)	sepal width (cm)	...	petal width (cm)	label
73	6.1	2.8	...	1.2	1
18	5.7	3.8	...	0.3	0
118	7.7	2.6	...	2.3	2
78	6.0	2.9	...	1.5	1
76	6.8	2.8	...	1.4	1
..	...	...	...	...	...
71	6.1	2.8	...	1.3	1
106	4.9	2.5	...	1.7	2
14	5.8	4.0	...	0.2	0
92	5.8	2.6	...	1.2	1
102	7.1	3.0	...	2.1	2

Hình. Dữ liệu sau khi trộn

2.3. Tách dữ liệu:

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

Hình. Tách dữ liệu

- X là biến lưu các thuộc tính từ *data* và chuyển chúng thành mảng numpy với kiểu dữ liệu *float32*.

- y là biến lưu nhãn từ cột *label* của tập *data*, cũng được chuyển thành mảng numpy và kiểu dữ liệu là *float32*.

2.4. Chuyển cột nhãn thành dạng one_hot:

```
# One-hot encode các nhãn y
encoder = OneHotEncoder(sparse_output=False)
y = encoder.fit_transform(y).astype('float32')
```

Hình. Chuyển cột nhãn thành dạng one_hot

- Khởi tạo biến *encoder* là một đối tượng *OneHotEncoder* để mã hóa các nhãn thành dạng one-hot. Tham số *sparse_output=False* chỉ ra rằng kết quả sẽ là một mảng dày đặc.
- Áp dụng quá trình one-hot encoding lên cột nhãn *y*, chuyển kết quả thành mảng numpy với kiểu *float32*.

Trước khi chuyển	Sau khi chuyển
[[1.] [0.] [2.] [1.] [1.] [0.] [1.] [2.] [1.] [1.]]	[[0. 1. 0.] [1. 0. 0.] [0. 0. 1.] [0. 1. 0.] [0. 1. 0.] [1. 0. 0.] [0. 1. 0.] [0. 0. 1.] [0. 1. 0.] [0. 1. 0.]]

Hình. Trước và sau khi chuyển hóa cột nhãn

(Minh họa trên 10 hàng đầu tiên)

2.5. Phân chia dữ liệu:

```
# Chia tập dữ liệu thành tập huấn luyện và kiểm tra
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. Phân chia dữ liệu huấn luyện và kiểm tra

- Sau khi đã tách các đặc trưng và xử lý cột 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: (120, 4)
X_test.shape: (30, 4)
y_train.shape: (120, 3)
y_test.shape: (30, 3)
```

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

2.6. Xây dựng mô hình:

2.6.1. Khởi tạo trọng số W và bias b:

```
# Xây dựng mô hình
W = tf.Variable(tf.random.normal((4, 3)))
b = tf.Variable(tf.random.normal((3, )))
```

Hình. Khởi tạo W và b

- Khởi tạo trọng số W có kích thước (4, 3), là do dữ liệu có 4 thuộc tính và 3 lớp, được khởi tạo ngẫu nhiên bằng hàm **tf.random.normal**. Sử dụng **tf.Variable** để W có thể thay đổi sau quá trình huấn luyện.

- Bias b được khởi tạo có kích thước (3,), do dữ liệu có 3 lớp, cũng được khởi tạo ngẫu nhiên bằng hàm **tf.random.normal**. Sử dụng **tf.Variable** để b có thể thay đổi sau quá trình huấn luyện.

2.6.2. Hàm dự đoán:

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

Hình. Hàm dự đoán

- **tf.function**: Đánh dấu hàm **predict** như một hàm của Tensorflow, giúp tối ưu hóa và chạy nhanh hơn trong biểu đồ tính toán.
- Tính toán đầu ra bằng cách nhân ma trận đầu vào với ma trận trọng số W, cộng thêm độ lệch b, sau đó áp dụng hàm **softmax** để chuyển đổi thành xác suất phân lớp.

2.6.3. Hàm mất mát:

```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(tf.reduce_sum(y * tf.math.log(y_hat), axis=1))
```

Hình. Hàm mất mát

- Hàm **tf.clip_by_value** giúp giới hạn giá trị của *y_hat* để tránh các giá trị gây lỗi trong tính toán.
- Hàm mất mát được tính bằng hàm **categorical_crossentropy**, vì đây là bài toán phân loại đa lớp (cụ thể là 3 lớp).

2.7. Huấn luyện mô hình:

```python
# Training the model
alpha = 0.01
epochs = 500
batch_size = 32

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

arr_acc = []
arr_loss = []

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.argmax(train_y_hat, axis=1)
    train_actuals = tf.argmax(y_train, axis=1)
    train_accuracy = tf.reduce_mean(tf.cast(train_predictions == train_actuals, tf.float32))

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

Hình. 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.

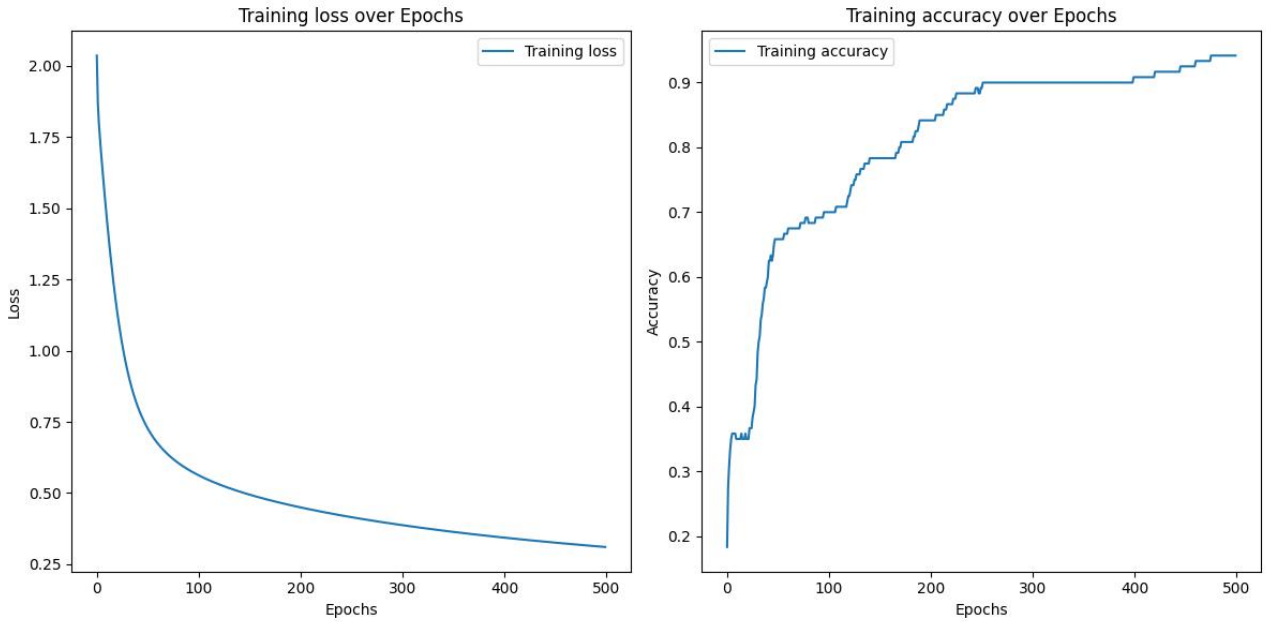
+ Tính toán lại giá trị mất mát cho toàn bộ X_train sau khi các trọng số W và bias b được cập nhật.

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

+ Tìm lại các nhãn chính xác của tập *y_train* bằng hàm **tf.argmax** để so sánh.

+ Tính toán độ chính xác bằng cách so sánh các giá trị dự đoán của X_train 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

Biểu đồ trên cho thấy:

- Hàm mất mát bắt đầu ở giá trị khá cao và giảm dần theo số lượng epoch. Điều này cho thấy mô hình đang học dần dần các đặc trưng của dữ liệu và dự đoán càng chính xác hơn. Sự giảm mạnh của hàm mất mát ở những epoch đầu tiên chứng tỏ mô hình đang học khá nhanh và hiệu quả, càng về sau, tốc độ giảm chậm dần.
- Độ chính xác bắt đầu ở giá trị thấp và tăng mạnh ở 100 epoch đầu tiên phản ánh sự tiến bộ của mô hình. Từ khoảng epoch 100 trở đi, độ chính xác vẫn tăng dần nhưng với tốc độ chậm hơn và cuối cùng đạt được khoảng 95%.

2.8. Đánh giá mô hình:

```
# Đánh giá mô hình
y_hat_test = predict(X_test, W, b)

predictions = tf.argmax(y_hat_test, axis=1)
actuals = tf.argmax(y_test, axis=1)

accuracy = tf.reduce_mean(tf.cast(predictions == actuals, tf.float32)).numpy()
confusion_matrix = tf.math.confusion_matrix(actuals, predictions)

print(f"Accuracy: {accuracy * 100:.2f}%")
print(f"Confusion Matrix:\n{confusion_matrix}")
```

Hình. Đánh giá mô hình

- Tính toán đầu ra dự đoán của mô hình trên tập kiểm tra.
- Thực hiện tìm nhãn chính xác của đầu ra vừa dự đoán và đầu ra của tập kiểm tra để so sánh.
- Tính toán độ chính xác của mô hình trên tập kiểm tra.
- Tính toán ma trận nhầm lẫn của mô hình trên tập kiểm tra.

```
Accuracy: 100.00%
Confusion Matrix:
[[11  0  0]
 [ 0 11  0]
 [ 0  0  8]]
```

Hình. Kết quả đánh giá

Kết quả: Độ chính xác là 100% trên tập kiểm tra và dựa trên ma trận nhầm lẫn, không có phần tử bị dự đoán sai. Tuy nhiên, mô hình cần được kiểm chứng thêm để đảm bảo mô hình không bị overfitting.

--- HẾT ---