BÀI TẬP 4

Mạng nơ ron đa tầng, đơn lớp, đa lớp

---

1. Xây dựng mạng nơ ron Perceptron đa tầng (2 tầng), 2 lớp cho bài toán XOR:

1.1. 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],
                 [1],
                 [1],
                 [0]])
```

Hình. Dữ liệu đầu vào và đầu ra mong muốn

- **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, 1], 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:

```
n = 2

W1 = tf.Variable(tf.random.normal((n, 8)))
b1 = tf.Variable(tf.random.normal((8, )))

W2 = tf.Variable(tf.random.normal((8, 1)))
b2 = tf.Variable(tf.random.normal((1, )))
```

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.
- n: Là một biến với ý nghĩa là số lượng đặc trưng (thuộc tính) đầu vào.
- W1: Là một biến tensor chứa các trọng số có kích thước [2, 8] (gồm 2 thuộc tính và 8 nơ ron) được tạo ngẫu nhiên bằng hàm **tf.random.normal**. Sử dụng 8 nơ ron thay vì 2 giúp cho dữ liệu được học kĩ hơn, giúp gia tăng tỉ lệ dự đoán cao hơn.
- b1: Là một biến tensor đại diện cho bias được tạo ngẫu nhiên bằng hàm **tf.random.normal** với kích thước [8,].
- W2: Là một biến tensor chứa các trọng số có kích thước [8, 1] (gồm 8 đặc trưng đầu vào và 1 nơ ron đầu ra) được tạo ngẫu nhiên bằng hàm **tf.random.normal**.
- b2: Là một biến tensor đại diện cho bias được tạo ngẫu nhiên bằng hàm **tf.random.normal** với kích thước [1,].

```
W1: <tf.Variable 'Variable:0' shape=(2, 8) dtype=float32, numpy=
array([[ 0.14372632, -0.27744102, -1.0567665 ,  1.28576   , -0.910939  ,
        -0.25930366, -0.21190499,  1.0104594 ],
       [ 0.10335889, -1.3034463 , -0.5610751 , -0.22024557,  1.6349254 ,
         0.07230361, -0.94838816, -0.18085854]], dtype=float32)>
b1: <tf.Variable 'Variable:0' shape=(8,) dtype=float32, numpy=
array([ 1.0591536 , -0.6865848 , -1.644775  ,  1.296202  ,  0.7627606 ,
        1.211262  , -0.31281024, -0.7927294 ], dtype=float32)>
W2: <tf.Variable 'Variable:0' shape=(8, 1) dtype=float32, numpy=
array([[-0.16974631],
       [-1.1307477 ],
       [ 0.4759914 ],
       [ 1.2657696 ],
       [-0.3952247 ],
       [-0.6036509 ],
       [-0.14411546],
       [-1.32489   ]], dtype=float32)>
b2: <tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([-1.519978], dtype=float32)>
```

Hình. Các trọng số và bias trước khi huấn luyện

1.3. Hàm dự đoán:

```
# Hàm dự đoán
1 usage
@tf.function
def layer1(X, W1, b1):
    return tf.nn.relu(tf.add(tf.matmul(X, W1), b1))


1 usage
@tf.function
def layer2(X, W2, b2):
    return tf.nn.sigmoid(tf.add(tf.matmul(X, W2), b2))


@tf.function
def predict(X, W1, b1, W2, b2):
    return layer2(layer1(X, W1, b1), W2, b2)
```

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 **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.
- Hàm **tf.add()** thực hiện phép tính cộng giữa hai tensor.
- Hàm **layer1()** là hàm nhận dữ liệu đầu vào, tương ứng với tầng đầu tiên của mạng. Tham số đầu vào là:
  + X: Dữ liệu đầu vào từ dữ liệu.
  + W1: Ma trận trọng số.
  + b1: Bias.

Kết quả đầu ra của hàm được tính bằng hàm **ReLU** dựa trên kết quả vừa tính được để giữ kết quả không có giá trị âm.

$$f(x) = \begin{cases} x & nếu\ x \geq 0 \\ 0 & nếu\ x < 0 \end{cases}$$

Hình. Hàm kích hoạt ReLU

- Hàm **layer2()** là nhận đặc trưng đầu vào từ hàm **layer1()** và cũng là hàm đưa ra kết quả cuối cùng của mạng. Tham số đầu vào là:
  + X: Đặc trưng đầu vào từ tầng trước.
  + W2: Ma trận trọng số.
  + b2: Bias.

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

- Hàm **predict()** có nhiệm vụ đưa ra kết quả dự đoán từ dữ liệu ban đầu khi đi qua các tầng trong mạng. Cụ thể, khi dữ liệu được đưa vào, nó sẽ được tính toán ở tầng đầu tiên tại hàm **layer1()**. Sau đó, các kết quả sẽ được truyền qua tầng tiếp theo để tiếp tục tính toán và đưa ra kết quả của dữ liệu cần dự đoán.

```
# Dự đoán ban đầu
y_hat = predict(X, W1, b1, W2, b2)
print(f"Predict before training:\n{y_hat}")
```

```
Predict before training:
[[0.25129578]
 [0.11126865]
 [0.6639368 ]
 [0.4982337 ]]
```

Hình. Kết quả hàm predict() trước khi huấn luyện

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.

```
# Hàm mất mát
@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 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.

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

```
Loss before training:
0.8961020112037659
```

Hình. Tính toán độ mất mát trước khi huấn luyện

1.5. Huấn luyện mô hình:

- Huấn luyện mạng nơ-ron 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:

```
# Huấn luyện
alpha = 0.01
for epoch in range(5000):
    with tf.GradientTape() as t:
        loss = L(y, predict(X, W1, b1, W2, b2))
    dW1, db1, dW2, db2 = t.gradient(loss, sources: [W1, b1, W2, b2])
    W1.assign_sub(alpha * dW1)
    b1.assign_sub(alpha * db1)
    W2.assign_sub(alpha * dW2)
    b2.assign_sub(alpha * db2)
```

Hình. Huấn luyện mạng bằng Gradient Descent

+ Mô hình sẽ được huấn luyện qua 5000 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 các trọng số và bias được cập nhật.

+ **t.gradient:** Tính toán gradient của hàm mất mát *loss* với các trọng số và bias.

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

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

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

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

```
W1: <tf.Variable 'Variable:0' shape=(2, 8) dtype=float32, numpy=
array([[ 0.02138988, -0.27744102, -1.0567665 ,  2.090304  , -1.9996271 ,
        -0.7274528 , -0.21190499,  2.0399282 ],
       [ 0.02646084, -1.3034463 , -0.5610751 ,  0.54091996,  1.9999012 ,
        -0.27844253, -0.94838816,  2.0397668 ]], dtype=float32)>
b1: <tf.Variable 'Variable:0' shape=(8,) dtype=float32, numpy=
array([ 1.1205806e+00, -6.8658477e-01, -1.6447750e+00,  8.8645673e-01,
        8.4830390e-05,  1.4495032e+00, -3.1281024e-01, -2.0398061e+00],
      dtype=float32)>
W2: <tf.Variable 'Variable:0' shape=(8, 1) dtype=float32, numpy=
array([[-0.36414132],
       [-1.1307477 ],
       [ 0.4759914 ],
       [ 1.9150884 ],
       [ 2.0175438 ],
       [-1.2380025 ],
       [-0.14411546],
       [-3.5421653 ]], dtype=float32)>
b2: <tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([-1.7818288], dtype=float32)>
```

Hình. Các trọng số và bias sau khi huấn luyện

- Thực hiện dự đoán lại giá trị mới sau khi huấn luyện:

```
Y_hat after training:
[[0.09224278]
 [0.9576827 ]
 [0.93142325]
 [0.0375171 ]]
```

Hình. Giá trị dự đoán mới

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

```
# Determine labels
predictions = tf.cast(y_hat_new > 0.5, dtype=tf.float32)
print(f"Predictions:\n{predictions}")
```

Hình. Xác định nhãn dự đoán

+ 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 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*.

2. Xây dựng mạng nơ ron 2 tầng, 2 lớp cho dữ liệu sau:

	X1	X2	Classe
0	0.204	0.834	0
1	0.222	0.730	0
2	0.298	0.822	0
3	0.450	0.842	0
4	0.412	0.732	0
5	0.298	0.640	0
6	0.588	0.298	0
7	0.554	0.398	0
8	0.670	0.466	0
9	0.834	0.426	0
10	0.724	0.368	0
11	0.790	0.262	0
12	0.824	0.338	0
13	0.136	0.260	1
14	0.146	0.374	1
15	0.258	0.422	1
16	0.292	0.282	1
17	0.478	0.568	1
18	0.654	0.776	1
19	0.786	0.758	1
20	0.690	0.628	1
21	0.736	0.786	1
22	0.574	0.742	1

Hình. Dữ liệu huấn luyện

2.1. Thu thập dữ liệu:

```
# Lấy dữ liệu
data = pd.read_excel( io: "./data.xlsx", sheet_name='Sheet2')
print(data)

data = shuffle(data)
print(data)
```

Hình. Thu thập dữ liệu

- Dữ liệu là một file **Excel** nên sẽ được đọc bằng hàm **read_excel()** từ thư viện *Pandas*. Tham số *sheet_name* là sheet cần đọc dữ liệu.
- Sau khi đọc dữ liệu, thực hiện trộn dữ liệu để dữ liệu khi huấn luyện sẽ có số lượng các lớp không quá chênh lệch, làm cho mô hình dự đoán chính xác hơn.

	X1	X2	Classe
11	0.790	0.262	0
8	0.670	0.466	0
3	0.450	0.842	0
5	0.298	0.640	0
19	0.786	0.758	1
10	0.724	0.368	0
0	0.204	0.834	0
6	0.588	0.298	0
12	0.824	0.338	0
22	0.574	0.742	1
16	0.292	0.282	1
13	0.136	0.260	1
21	0.736	0.786	1
14	0.146	0.374	1
7	0.554	0.398	0
1	0.222	0.730	0
15	0.258	0.422	1
18	0.654	0.776	1
2	0.298	0.822	0
4	0.412	0.732	0
20	0.690	0.628	1
9	0.834	0.426	0
17	0.478	0.568	1

Hình. Dữ liệu sau khi trộn

2.2. Phân chia dữ liệu:

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

Hình. Tách dữ liệu

- X: Chứa các thuộc tính đầu vào từ tập dữ liệu, được chuyển sang dạng mảng và mang kiểu *float32*.
- y: Chứa cột nhãn của tập dữ liệu, được **reshape()** để có dạng ma trận cột và mang kiểu *float32*.

```
# Phân chia dữ liệu
X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2)
```

Hình. Phân chia tập dữ liệu huấn luyện và kiểm tra

- Phân chia tập dữ liệu thành tập huấn luyện (80%) và tập kiểm tra (20%). Kết quả sau khi chia như sau:

```
X_train shape: (18, 2)
X_test shape: (5, 2)
y_train shape: (18, 1)
y_test shape: (5, 1)
```

Hình. Kích thước tập huấn luyện và tập kiểm tra

2.3. Xây dựng mô hình:

2.3.1. Khởi tạo các trọng số và bias:

```
n = X_train.shape[1]

W1 = tf.Variable(tf.random.normal((n, 8)))
b1 = tf.Variable(tf.random.normal((8, )))

W2 = tf.Variable(tf.random.normal((8, 4)))
b2 = tf.Variable(tf.random.normal((4, )))

W3 = tf.Variable(tf.random.normal((4, 1)))
b3 = tf.Variable(tf.random.normal((1, )))
```

Hình. Khởi tạo các trọng số và bias

- n: Là một biến với ý nghĩa là số lượng đặc trưng (thuộc tính) đầu vào.
- W1: Là một biến tensor chứa các trọng số của tầng đầu tiên được tạo ngẫu nhiên có kích thước [n, 8] (gồm số thuộc tính và 8 nơ ron).

- b1: Là một biến tensor đại diện cho bias của tầng đầu tiên được tạo ngẫu nhiên với kích thước [8,].
- W2: Là một biến tensor chứa các trọng số của tầng ẩn ở giữa được tạo ngẫu nhiên có kích thước [8, 4] (gồm 8 đặc trưng đầu vào và 4 nơ ron đầu ra).
- b2: Là một biến tensor đại diện cho bias của tầng ẩn ở giữa được tạo ngẫu nhiên với kích thước [4,].
- W3: Là một tensor chứa các trọng số của tầng đầu ra được tạo ngẫu nhiên có kích thước [4, 1] (gồm 4 đặc trưng đầu vào và 1 nơ ron đầu ra vì đây là mô hình phân loại nhị phân).
- b3: Là một biến tensor đại diện cho bias của tầng đầu ra được tạo ngẫu nhiên với kích thước [1,].

```
W1: <tf.Variable 'Variable:0' shape=(2, 8) dtype=float32, numpy=
array([[ 0.3657123 , -0.6159254 ,  1.6834162 , -0.34822595,  1.3734236 ,
         0.9711219 ,  2.0167296 , -0.10664044],
       [-0.5478127 , -0.7154204 , -0.7981361 ,  0.30281383, -2.5689921 ,
        -1.483876  , -0.03096466, -1.3106552 ]], dtype=float32)>
b1: <tf.Variable 'Variable:0' shape=(8,) dtype=float32, numpy=
array([-1.1930922 ,  1.6261388 ,  1.4876761 ,  2.2021084 , -1.3358599 ,
        0.23797835, -0.11243115, -0.50830305], dtype=float32)>
W2: <tf.Variable 'Variable:0' shape=(8, 4) dtype=float32, numpy=
array([[ 0.7469834 ,  1.1609818 ,  0.06045932,  1.0428544 ],
       [ 1.512527  , -0.44112515,  0.9323863 ,  1.1241934 ],
       [ 0.5000829 ,  2.3010533 , -0.96111184, -0.00908731],
       [ 0.61038935, -0.92909586,  0.6542082 , -0.3780185 ],
       [ 1.0231053 ,  0.23702115,  2.6296358 , -0.06215349],
       [ 0.12485609, -1.3551203 , -0.2023148 , -0.02668927],
       [ 0.07019497,  0.02787207, -0.13585646,  1.1818671 ],
       [ 0.72007215, -1.6682435 ,  0.5057326 ,  0.10579818]],
      dtype=float32)>
b2: <tf.Variable 'Variable:0' shape=(4,) dtype=float32, numpy=array([-0.05171251,  1.2542738 , -1.1124135 ,  0.73956513], dtype=float32)>
W3: <tf.Variable 'Variable:0' shape=(4, 1) dtype=float32, numpy=
array([[-1.7471818 ],
       [-0.10679635],
       [-2.44429   ],
       [ 0.22070812]], dtype=float32)>
b3: <tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([-0.7828403], dtype=float32)>
```

Hình. Các trọng số và bias sau khi khởi tạo

2.3.2. Các hàm cần thiết:

```python
@tf.function
def layer1(X, W1, b1):
    return tf.nn.relu(tf.add(tf.matmul(X, W1), b1))


1 usage
@tf.function
def layer2(X, W2, b2):
    return tf.nn.relu(tf.add(tf.matmul(X, W2), b2))

1 usage
@tf.function
def layer3(X, W3, b3):
    return tf.nn.sigmoid(tf.add(tf.matmul(X, W3), b3))


@tf.function
def predict(X, W1, b1, W2, b2, W3, b3):
    return layer3(layer2(layer1(X, W1, b1), W2, b2), W3, b3)
```

Hình. Hàm dự đoán

- Hàm **layer1()** nhận đầu vào là dữ liệu ban đầu cùng với trọng số W1 và bias b1 được khởi tạo để tính toán và đưa kết quả vừa tính toán qua tầng sau.
- Hàm **layer2()** nhận đầu vào là dữ liệu từ tầng trước đó (layer1) cùng với trọng số W2 và bias b2 được khởi tạo để trích xuất thêm các đặc trưng của tầng 1 và đưa kết quả qua tầng cuối cùng.
- Hàm **layer3()** là tầng cuối cùng của mạng nhận nhiệm vụ là đưa ra kết quả cuối cùng (nhãn '0' hoặc '1') dựa trên các xác suất được chuyển đổi bởi hàm **sigmoid**.
- Hàm **predict()** sử dụng chuỗi các tầng mạng để thực hiện dự đoán.

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

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.

```
@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.

2.4. Huấn luyện mô hình:

```
# Huấn luyện
alpha = 0.01
for epoch in range(5000):
    with tf.GradientTape() as t:
        loss = L(y_train, predict(X_train, W1, b1, W2, b2, W3, b3))
    dW1, db1, dW2, db2, dW3, db3 = t.gradient(loss, sources: [W1, b1, W2, b2, W3, b3])
    W1.assign_sub(alpha * dW1)
    b1.assign_sub(alpha * db1)
    W2.assign_sub(alpha * dW2)
    b2.assign_sub(alpha * db2)
    W3.assign_sub(alpha * dW3)
    b3.assign_sub(alpha * db3)
```

Hình. Huấn luyện mô hình

Quá trình huấn luyện được mô tả như sau:

- alpha: Là tốc độ học, giúp điều chỉnh tốc độ cập nhật của các trọng số và bias.
- Mô hình sẽ được huấn luyện qua 5000 epoches. Sử dụng nhiều lần lặp giúp cho mô hình học được kĩ càng hơn về dữ liệu, giúp việc dự đoán các dữ liệu mới chính xác hơn.
- **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.
- Độ mất mát sẽ được tính toán lại dựa trên giá trị dự đoán mới sau khi các trọng số và bias được cập nhật.
- **t.gradient**: Tính đạo hàm của hàm mất mát dựa trên các trọng số và bias

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

```
# Dự đoán
y_hat = predict(X_test, W1, b1, W2, b2, W3, b3)
print(f"Y_hat :\n{y_hat}")

# Độ mất mát
loss = L(y_test, y_hat)
print(f"Loss after training:\n{loss}")

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

Hình. Xác định nhãn trên giá trị dự đoán sau khi huấn luyện

- Thực hiện dự đoán trên tập dữ liệu kiểm tra *X_test* với các trọng số và bias sau khi huấn luyện mô hình.

```
Y_hat :
[[9.4742537e-01]
 [9.8531908e-01]
 [3.5629575e-03]
 [5.8680732e-04]
 [1.3890720e-03]]
```

Hình. Kết quả dự đoán

- Độ mất mát sau khi huấn luyện:

```
Loss before training: 3.1384897232055664
Loss after training: 0.014868651516735554
```

Hình. Độ mất mát trước và sau khi huấn luyện

- Xác định nhãn sau khi dự đoán:

+ Biểu thức *y_hat >= 0.5* tạo ra một tensor chứa các giá trị boolean, dựa trên việc so sánh mỗi phần tử trong *y_hat* 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:
[[1.]
 [1.]
 [0.]
 [0.]
 [0.]]
Y_true:
[[1.]
 [1.]
 [0.]
 [0.]
 [0.]]
```

Hình. Kết quả dự đoán so với giá trị thực tế

3. Xây dựng mạng MLP với tập dữ liệu Speaker Accent Recognition:

3.1. Thu thập dữ liệu và xử lí:

```
df = pd.read_csv("accent-mfcc-data-1.csv")

X = df.drop( labels: "language", axis=1).values
y = df["language"].values.reshape(-1, 1)

X = MinMaxScaler().fit_transform(X)
y = OneHotEncoder(sparse_output=False).fit_transform(y)
```

Hình. Thu thập và xử lí dữ liệu

- Tập dữ liệu Speaker Accent Recognition tập trung vào phân loại ngôn ngữ dựa trên các đặc trưng MFCC (Mel-Frequency Cepstral Coefficients), thường được sử dụng trong ứng dụng nhận dạng giọng nói hoặc âm thanh. MFCC mô tả các đặc trưng phổ tần số của âm thanh.
- Dữ liệu được đọc bằng hàm **read_csv()** của thư viện Pandas và lưu trữ vào biến *df*.
- Dữ liệu được tách riêng các đặc trưng và nhãn để phục vụ cho quá trình huấn luyện, cụ thể:
  + X: Lưu trữ tất cả các cột của dữ liệu, trừ cột nhãn (language), bao gồm các đặc trưng mô tả đặc điểm của âm thanh và tiếng nói.
  + y: Lưu trữ cột nhãn ngôn ngữ của mỗi mẫu trong dữ liệu.
- Do dữ liệu X khá chênh lệch các giá trị giữa các cột, nên nó sẽ được chuẩn hóa về khoảng (0, 1) để đồng nhất khoảng giá trị thông qua *MinMaxScaler()*.
- Dữ liệu cột nhãn được mã hóa bằng *OneHotEncoder()* để đưa nó về dạng one-hot.

3.2. Phân chia dữ liệu:

```
X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2, random_state=42, shuffle=True)

X_train = tf.convert_to_tensor(X_train, dtype=tf.float32)
X_test = tf.convert_to_tensor(X_test, dtype=tf.float32)
y_train = tf.convert_to_tensor(y_train, dtype=tf.float32)
y_test = tf.convert_to_tensor(y_test, dtype=tf.float32)
```

Hình. Phân chia dữ liệu

- Dữ liệu được chia thành các tập huấn luyện và tập kiểm tra theo tỉ lệ: 80% cho huấn luyện và 20% cho kiểm tra.
- Các tập huấn luyện và tập kiểm tra sau đó được đưa về dạng tensor bằng hàm **tf.convert_to_tensor()** và có kiểu dữ liệu là *float32*.

Kết quả sau khi phân chia như sau:

```
X_train shape: (263, 12)
X_test shape: (66, 12)
y_train shape: (263, 6)
y_test shape: (66, 6)
```

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

3.3. Xây dựng mô hình:

3.3.1. Khởi tạo trọng số và bias:

```
n_input = X_train.shape[1]
n_hidden = 64
n_output = y_train.shape[1]

W1 = tf.Variable(tf.random.normal((n_input, n_hidden)))
b1 = tf.Variable(tf.zeros((n_hidden,)))

W2 = tf.Variable(tf.random.normal((n_hidden, n_output)))
b2 = tf.Variable(tf.zeros((n_output,)))
```

Hình. Trọng số và bias được khởi tạo

- n_input: Là biến thể hiện số lượng đặc trưng đầu vào của mô hình.
- n_hidden: Là biến thể hiện số lượng lớp ẩn của mô hình.
- n_output: Là biến thể hiện số lượng lớp đầu ra của mô hình.
- Mô hình mạng MLP gồm có 2 tầng, cụ thể:
  + W1 và W2 là các trọng số của mô hình, được khởi tạo bằng cách sử dụng hàm **tf.random.normal()**.
  + b1 và b2 là các bias của mô hình, được khởi tạo bằng cách sử dụng vector zero.

3.3.2. Các hàm cần thiết:

```
1 usage
@tf.function
def layer1(X, W, b):
    return tf.nn.relu(tf.matmul(X, W) + b)

1 usage
@tf.function
def layer2(X, W, b):
    return tf.nn.softmax(tf.matmul(X, W) + b)

@tf.function
def predict(X, W1, b1, W2, b2):
    return layer2(layer1(X, W1, b1), W2, b2)

@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))

4 usages
@tf.function
def to_label(y_hat):
    return tf.argmax(y_hat, axis=1, output_type=tf.int32)
```

Hình. Các hàm cần thiết

- Hàm **layer1()** là lớp ẩn của mô hình, sử dụng hàm ReLU để tính toán đầu ra.
- Hàm **layer2()** là lớp đầu ra của mô hình, sử dụng hàm kích hoạt softmax để tính toán đầu ra.
- Hàm **predict()** là hàm dự đoán đầu ra của mô hình, sử dụng các tầng **layer1** và **layer2** để tính toán đầu ra.
- Hàm **L()** là hàm mất mát của mô hình, sử dụng Categorical_Crossentropy vì đây là bài toán phân loại đa lớp.
- Hàm **to_label()** là hàm đưa về nhãn chính xác của một mẫu dữ liệu vì cột nhãn đang được thể hiện ở dạng one-hot.

3.4. Huấn luyện mô hình:

```
epochs = 500
batch_size = 32
alpha = 0.1
arr_acc = []
arr_loss = []

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

for epoch in range(epochs):
    epoch_loss = 0
    epoch_acc = 0
    batches = train_data.batch(batch_size)
    for X_batch, y_batch in batches:
        with tf.GradientTape() as tape:
            y_hat = predict(X_batch, W1, b1, W2, b2)
            loss = L(y_batch, y_hat)

        dW1, db1, dW2, db2 = tape.gradient(loss, sources: [W1, b1, W2, b2])
        W1.assign_sub(alpha * dW1)
        b1.assign_sub(alpha * db1)
        W2.assign_sub(alpha * dW2)
        b2.assign_sub(alpha * db2)
        epoch_loss += loss.numpy()

        y_hat = predict(X_batch, W1, b1, W2, b2)
        y_hat_labels = to_label(y_hat)
        y_batch_labels = to_label(y_batch)
        acc = tf.reduce_mean(tf.cast(tf.equal(y_hat_labels, y_batch_labels), tf.float32))
        epoch_acc += acc.numpy()

    print(f"Epoch {epoch + 1}, Loss: {epoch_loss / len(batches)}, Accuracy: {epoch_acc / len(batches)}")
    arr_acc.append(epoch_acc / len(batches))
    arr_loss.append(epoch_loss / len(batches))
```

Hình. Huấn luyện mô hình

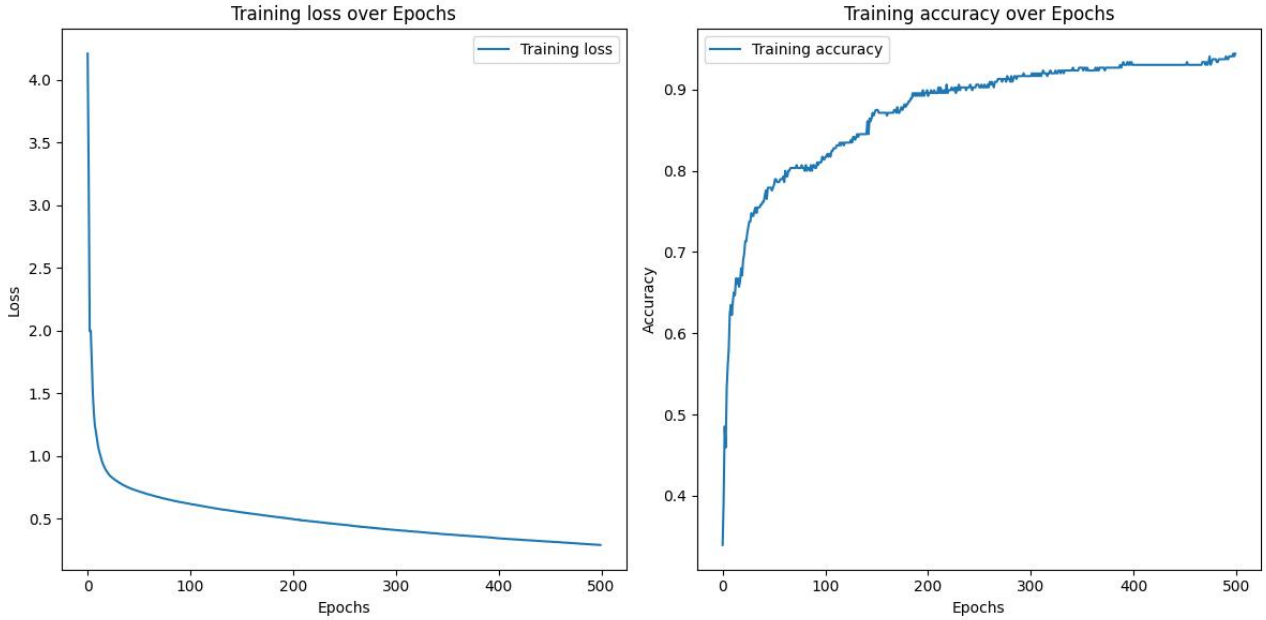
- Quá trình huấn luyện mạng nơ-ron sẽ được lặp 500 lần, mỗi lần lặp là 32 mẫu dữ liệu (batch_size = 32) với hệ số học là 0.1 (alpha = 0.1). Arr_acc và arr_loss sẽ lưu trữ độ chính xác và giá trị mất mát của mỗi vòng lặp.
- train_data được tạo ra từ hàm **tf.data.Dataset.from_tensor_slices()** giúp chia dữ liệu thành các batch nhỏ.
- Trong mỗi vòng lặp:

+ Epoch_loss, epoch_acc sẽ lưu trữ độ chính xác và giá trị mất mát của mỗi vòng lặp.

+ Chia dữ liệu thành các batch nhỏ với kích thước là 32 mẫu dữ liệu mỗi batch.

+ Lặp qua từng batch trong mỗi epoch, mô hình sẽ thực hiện các tính toán để cập nhật trọng số và bias. Độ chính xác và giá trị mất mát mỗi batch sẽ được cộng dồn vào epoch_acc và epoch_loss.

+ Tính trung bình độ chính xác và giá trị mất mát của mỗi epoch. Sau đó, lưu những giá trị đó vào arr_acc, arr_loss.



Hình. Quá trình huấn luyện

3.5. Đánh giá mô hình:

```
y_hat = predict(X_test, W1, b1, W2, b2)
loss = L(y_test, y_hat)

predictions = tf.equal(to_label(y_hat), to_label(y_test))
accuracy = tf.reduce_mean(tf.cast(predictions, tf.float32))

print(f"Accuracy: {accuracy.numpy()}")
print(f"Loss: {loss.numpy()}")
```

Hình. Đánh giá mô hình

- Thực hiện dự đoán nhãn cho tập kiểm tra với các trọng số và bias được cập nhật. Sau đó, tính độ mất mát giữa giá trị vừa dự đoán với giá trị nhãn thực tế.
- Chuyển các giá trị *y_hat* và *y_test* về chính xác nhãn cụ thể bằng hàm **to_label()** và so sánh với nhau bằng hàm **tf.equal()**.
- Tính độ chính xác bằng cách chuyển đổi mảng boolean thành số, sau đó tính trung bình.

```
Accuracy: 0.8484848737716675
Loss: 0.4111064672470093
```

Hình. Kết quả trên tập kiểm tra

---HẾT---