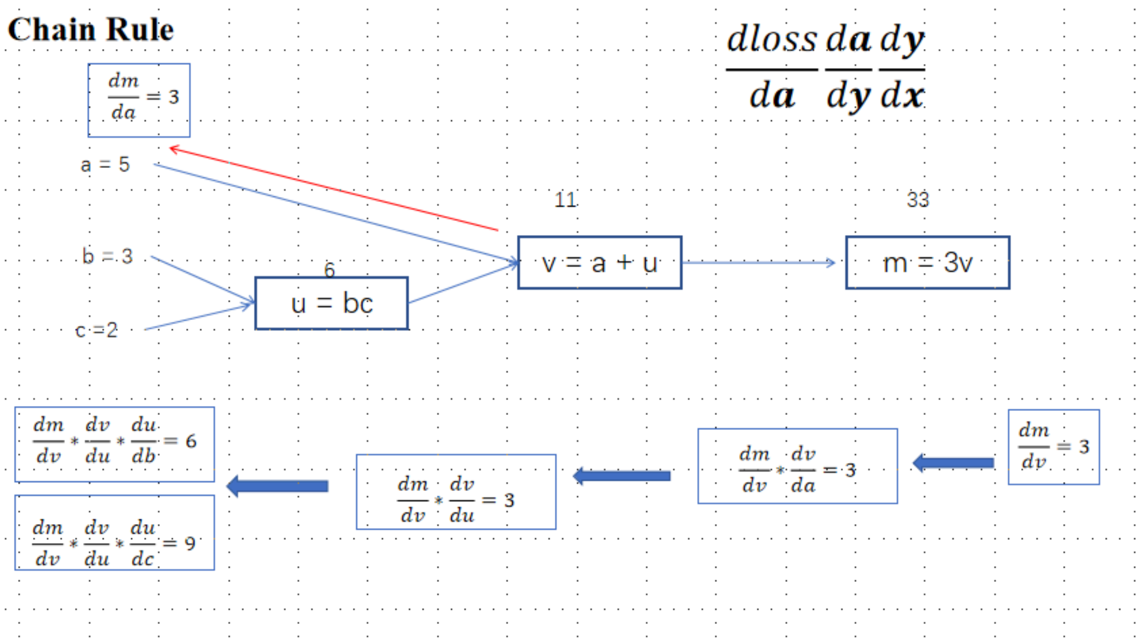
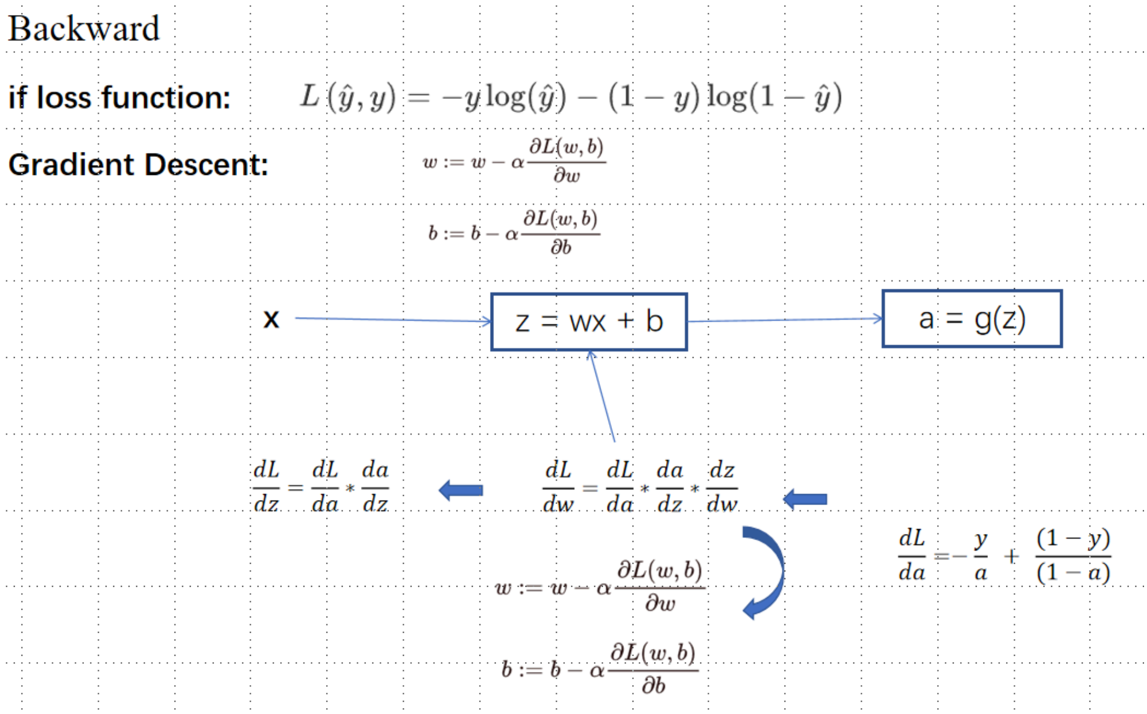
Chain rule



Backward Propagation



Activation Function

Using a linear activation function, the neural network just linearly combines the inputs and then outputs it!

Activation Function

$$\mathrm{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \qquad \mathrm{ReLU}(x) = \max(0, x) \qquad \mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

Why need a nonlinear activation function?

if activation function : $g(z) = z$

$$\left.\begin{aligned} &(1)\ a^{[1]} = z^{[1]} = W^{[1]}x + b^{[1]} \\ &(2)\ a^{[2]} = z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \end{aligned}\right\} \; a^{[2]} = z^{[2]} = W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]}$$

$$a^{[2]} = z^{[2]} = W'x + b'$$

Using a linear activation function, the neural network just linearly combines the inputs and then outputs it!

How to build a multi-layer neural network from scratch

1 Methods of Building a Neural Network

① The general method for building a neural network is:

1. Define the neural network structure (number of input units, number of hidden units, etc.).
2. Initialize the model's parameters.
3. Loop:
   - 3.1 Implement forward propagation.
   - 3.2 Compute the loss.
   - 3.3 Implement backward propagation.
   - 3.4 Update parameters (gradient descent).
4. Test

1.1 Import Required Packages - Generating Necessary Data

In [106…

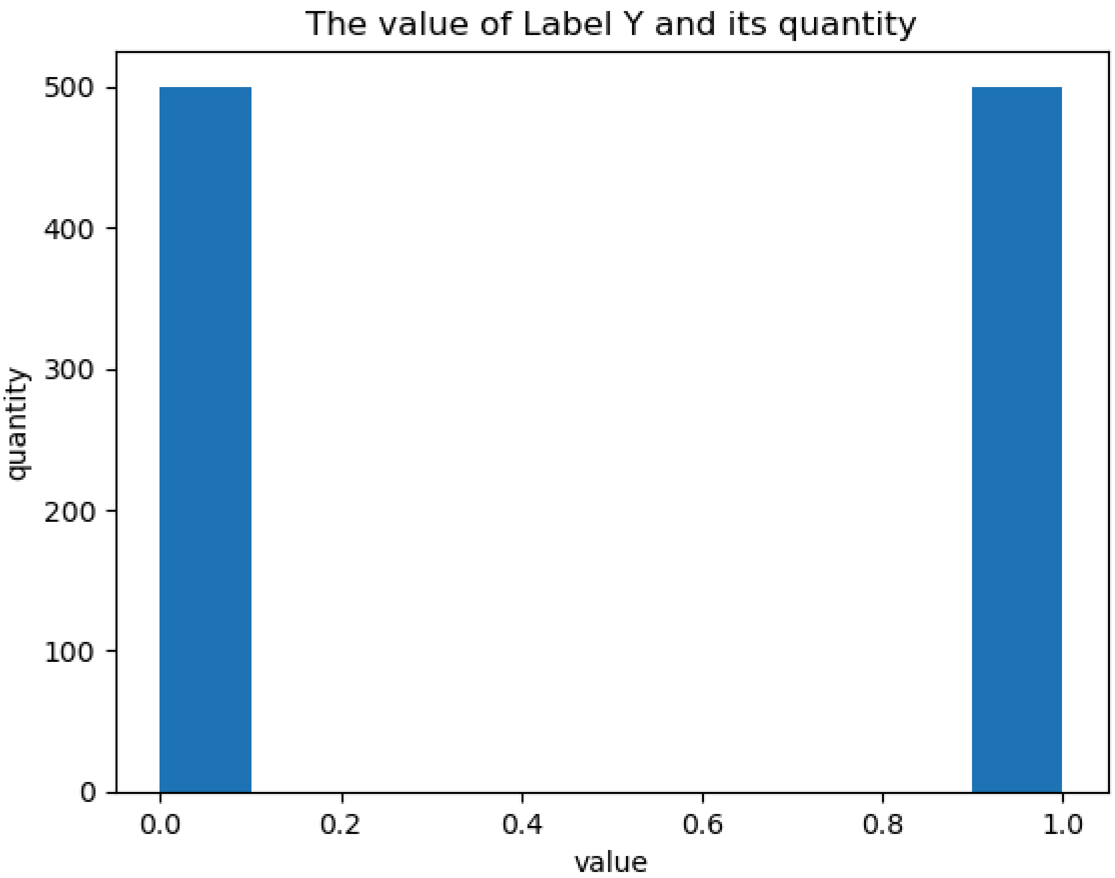
```
import numpy as np
# Import function to generate a moons dataset
from sklearn.datasets import make_moons
 # Import function to split dataset into training and test sets
from sklearn.model_selection import train_test_split
# Import function to calculate accuracy score
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt


# Generate synthetic data
# Generate a 2D dataset with 1000 samples
X, y = make_moons(n_samples=1000, noise=0.2, random_state=42)
```

```
 # Reshape labels y from (1000,) to (1000,1) to match the network's output s
y = y.reshape(-1, 1)
plt.hist(y)
plt.title("The value of Label Y and its quantity")

plt.xlabel("value")

plt.ylabel("quantity")
plt.show()
 # Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran
```



1.2 Initialize the Model's Parameters

① Here, we need to implement the function `initialize_parameters()`.

② We must ensure our parameters' sizes are appropriate.

③ We will initialize the weight matrix with random values.

- Use `np.random.randn(a, b) * 0.01` to randomly initialize a matrix of dimensions (a, b).
- Use `np.zeros((a, b))` to initialize a matrix (a, b) with zeros. Initialize the bias vector to zero.

In [96]:
```
# Parameter initialization
def initialize_parameters(input_dim, hidden_dim, output_dim):
    np.random.seed(42)  # Set random seed for reproducibility
    # Initialize weights for the first layer
    W1 = np.random.randn(input_dim, hidden_dim) / np.sqrt(input_dim)
    b1 = np.zeros((1, hidden_dim))  # Initialize biases for the first layer
```

```
    # Initialize weights for the second layer
    W2 = np.random.randn(hidden_dim, output_dim) / np.sqrt(hidden_dim)
    b2 = np.zeros((1, output_dim))  # Initialize biases for the second laye
    return {"W1": W1, "b1": b1, "W2": W2, "b2": b2}  # Return initialized pa
```

1.3 Activation Functions

In [97]:
```
# Activation function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))  # Sigmoid activation function
```

1.4 Forward Propagation

① We are now going to implement the forward propagation function `forward_propagation()`.

② We can use the `sigmoid()` function or the `np.tanh()` function.

③ The steps are as follows:

- Retrieve each parameter from the dictionary type `parameters` (which is the output of `initialize_parameters()`).
- Implement forward propagation to compute $Z^{[1]}, A^{[1]}, Z^{[2]}$, and $A^{[2]}$ (the vector of all predictions for the training data).
- The values needed for backward propagation are stored in `cache`, which will be used as input for the backward propagation function.

In [98]:
```
# Forward propagation
def forward_propagation(X, parameters):
    Z1 = np.dot(X, parameters["W1"]) + parameters["b1"]  # Compute linear pa
    A1 = np.tanh(Z1)  # Apply tanh function as the activation function of th
    Z2 = np.dot(A1, parameters["W2"]) + parameters["b2"]  # Compute linear p
    A2 = sigmoid(Z2)  # Apply sigmoid function as the activation function of
    # Cache intermediate values for use in backpropagation
    cache = {"Z1": Z1, "A1": A1, "Z2": Z2, "A2": A2}
    # Return activation value of the output layer and cached intermediate va
    return A2, cache
```

1.5 Computing the Loss Function

① Now, we have computed $A^{[2]}$

② $A^{[2](i)}$ contains the prediction for every example in the training set, now we can construct the cost function.

③ We choose cross-entropy loss as our cost, the formula to compute cost is as follows:

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(A^{[2](i)}\right) + (1-y^{(i)})\log\left(1-A^{[2](i)}\right)\right)$$

In [99]:
```
# Compute cost
def compute_cost(A2, Y):
```

```
    m = Y.shape[0]  # Get number of samples
    # Compute cross-entropy loss
    logprobs = np.multiply(np.log(A2), Y) + np.multiply((1 - Y), np.log(1 -
    cost = - np.sum(logprobs) / m  # Compute average loss over the dataset
    return cost  # Return cost
```

1.6 Backward Propagation

How to get dZ2?

1. **A2**:

$$A^{[2]} = \sigma(Z^{[2]}) = \frac{1}{1 + e^{-Z^{[2]}}}$$

2. **Loss Function**:

$$L(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

3. **How to get dZ2**:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

so:

$$\frac{\partial L}{\partial Z^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}}$$

then:

$$\frac{\partial L}{\partial A^{[2]}} = -\left(\frac{y}{A^{[2]}} - \frac{1 - y}{1 - A^{[2]}}\right)$$

$$\frac{\partial A^{[2]}}{\partial Z^{[2]}} = A^{[2]}(1 - A^{[2]})$$

then:

$$\frac{\partial L}{\partial Z^{[2]}} = A^{[2]} - y$$

so we get:

$$dZ2 = A^{[2]} - Y$$

In [100…

```
# Backward propagation
def backward_propagation(parameters, cache, X, Y):
    m = X.shape[0]  # Get number of samples
    A1 = cache["A1"]  # Retrieve activation values of the first layer from c
    A2 = cache["A2"]  # Retrieve activation values of the second layer from

    dZ2 = A2 - Y  # Calculate gradient at output layer
    dW2 = np.dot(A1.T, dZ2) / m  # Calculate gradient of the loss with respe
    # Calculate gradient of the loss with respect to b2
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m
    dZ1 = np.dot(dZ2, parameters["W2"].T) * (1 - np.power(A1, 2))  # Calcula
    dW1 = np.dot(X.T, dZ1) / m  # Calculate gradient of the loss with respec
```

```
    # Calculate gradient of the loss with respect to b1
    db1 = np.sum(dZ1, axis=0, keepdims=True) / m

    grads = {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2}  # Package grad
    return grads  # Return gradients
```

1.7 Updating Parameters

① We need to update (W1, b1, W2, b2) using (dW1, db1, dW2, db2).

② The update formula is as follows: $\theta = \theta - \alpha\frac{\partial L}{\partial \theta}$

③ Where:

1. α represents the learning rate.
2. θ represents a parameter.

In [101…

```
# Update parameters
def update_parameters(parameters, grads, learning_rate=0.01):
    parameters["W1"] -= learning_rate * grads["dW1"]  # Update first layer w
    parameters["b1"] -= learning_rate * grads["db1"]  # Update first layer b
    parameters["W2"] -= learning_rate * grads["dW2"]  # Update second layer
    parameters["b2"] -= learning_rate * grads["db2"]  # Update second layer
    return parameters  # Return updated parameters
```

1.8 Batch Processing Data

In [102…

```
# Batch generator for mini-batch gradient descent
def batch_generator(X, Y, batch_size=32):
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)
    for start_idx in range(0, X.shape[0] - batch_size + 1, batch_size):
        excerpt = indices[start_idx:start_idx + batch_size]
        yield X[excerpt], Y[excerpt]
```

1.9 Integrating the Model

In [103…

```
# Model training
def model(X_train, Y_train, X_test, Y_test, n_hidden=4, n_epochs=20, batch_s
    np.random.seed(42)  # Seed random number generator
    n_input = X_train.shape[1]  # Determine input dimension
    n_output = Y_train.shape[1]  # Determine output dimension
    print(f'n_input = {n_input}, n_output = {n_output}')
    parameters = initialize_parameters(n_input, n_hidden, n_output)  # Init
    for epoch in range(n_epochs):
        for X_batch, Y_batch in batch_generator(X_train, Y_train, batch_size
            A2, cache = forward_propagation(X_batch, parameters)  # Forward
            cost = compute_cost(A2, Y_batch)  # Compute cost
            grads = backward_propagation(parameters, cache, X_batch, Y_batch
            parameters = update_parameters(parameters, grads)  # Update para

        if print_cost:
            print(f"Cost after epoch {epoch}: {cost}")


    return parameters  # Return learned parameters
```

1.10 Training the Model

In [104…

```
# Train the model
parameters = model(X_train, y_train, X_test, y_test, n_hidden=4, n_epochs=1(
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
n_input = 2, n_output = 1
Cost after epoch 0: 0.7344604426989605
Cost after epoch 1: 0.6900291469216853
Cost after epoch 2: 0.721638557758888
Cost after epoch 3: 0.6615573895260174
Cost after epoch 4: 0.6703755091619054
Cost after epoch 5: 0.6705761721554763
Cost after epoch 6: 0.647638752226586
Cost after epoch 7: 0.665273512753088
Cost after epoch 8: 0.6456694787650644
Cost after epoch 9: 0.6295844689767961
W1 = [[ 0.29617788  0.11059671  0.29357749  1.05608563]
 [-0.12744831 -0.28892865  1.19445454  0.56565585]]
b1 = [[-0.00291607  0.02034294 -0.04771555  0.00040832]]
W2 = [[ 0.02508533]
 [ 0.36635204]
 [-0.39475393]
 [ 0.04798859]]
b2 = [[0.06651659]]
```

1.11 Prediction

① Build the `predict()` function to make predictions using the model.

② Use forward propagation to predict the outcomes.

note: $y_{prediction} = \begin{cases} 1 & \text{if } A2 > 0.5 \\ 0 & \text{otherwise} \end{cases}$

Note: If you want to set a matrix (X) to 0s and 1s based on a threshold, you can perform the operation as follows: X_new = (X > threshold)

In [105…

```
def predict(X_test, parameters):
    A2, _ = forward_propagation(X_test, parameters)  # Forward propagation v
    predictions = (A2 > 0.5)  # Convert probabilities to binary predictions
    accuracy = accuracy_score(y_test, predictions)  # Calculate accuracy on
    print(f"Accuracy: {accuracy}")
# Testing
# Prediction
predict(X_test, parameters)
```

```
Accuracy: 0.82
```

Part of the content is referenced from Andrew Ng - Deep Learning