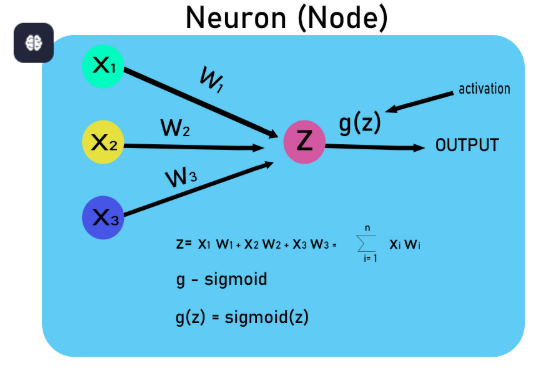
**Aim:** Write a program to build the deep neural network using NumPy

**Description:**

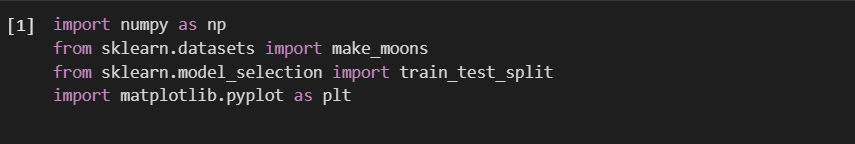
****

**Objective:** To implement a Deep Neural Network (DNN) using NumPy for binary classification on a toy dataset.

**Steps Overview:**

* Import libraries
* Generate or load dataset
* Initialize parameters
* Define activation functions
* Forward propagation
* Compute cost
* Backward propagation
* Update parameters
* Train the model
* Make predictions
* Evaluate accuracy

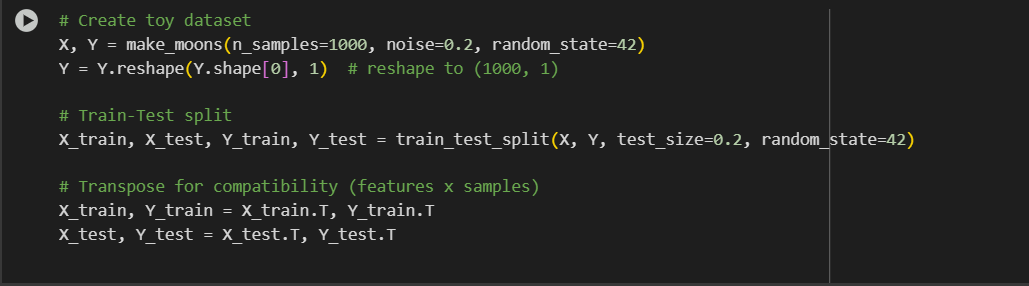
**Implementation:**

***1. Import Libraries*** *numpy:* A fundamental package for numerical computing in Python. We use it for matrix operations, random number generation, and array manipulation.

*make\_moons:* A function from sklearn.datasets that generates a toy dataset, useful for binary classification tasks.

*train\_test\_split:* A function that splits the dataset into training and test sets.

*matplotlib.pyplot:* A library for creating visualizations. We use it for plotting decision boundaries and training results.

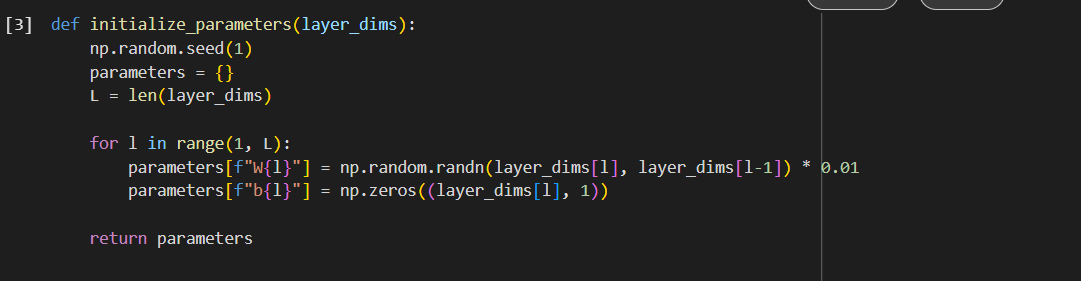
***2. Load and Preprocess Dataset*** 

make\_moons(): Generates a 2D dataset that resembles two moon-shaped clusters (often used for classification tasks). X is the feature set, and Y is the target (binary).

train\_test\_split(): Divides the dataset into training (80%) and testing (20%) sets.

.T: The .T operation transposes the matrices to ensure the data is in the shape (features, samples).

***3. Initialize Parameters***

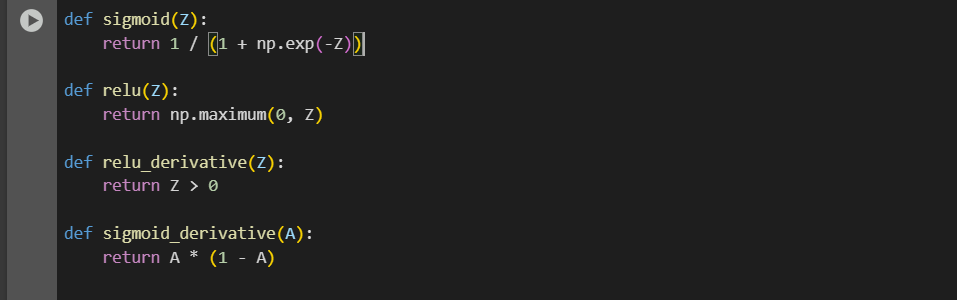
layer\_dims: A list of integers representing the number of neurons in each layer (e.g., [2, 10, 5, 1] for 2 input neurons, 10 neurons in the first hidden layer, 5 in the second, and 1 output neuron).

**Initialization**:

* **Weights (W)**: Initialized randomly using np.random.randn() and scaled by 0.01.
* **Biases (b)**: Initialized to zero for all layers.

**Purpose**: We initialize the parameters (W and b) for each layer in the network

***4. Activation Functions***

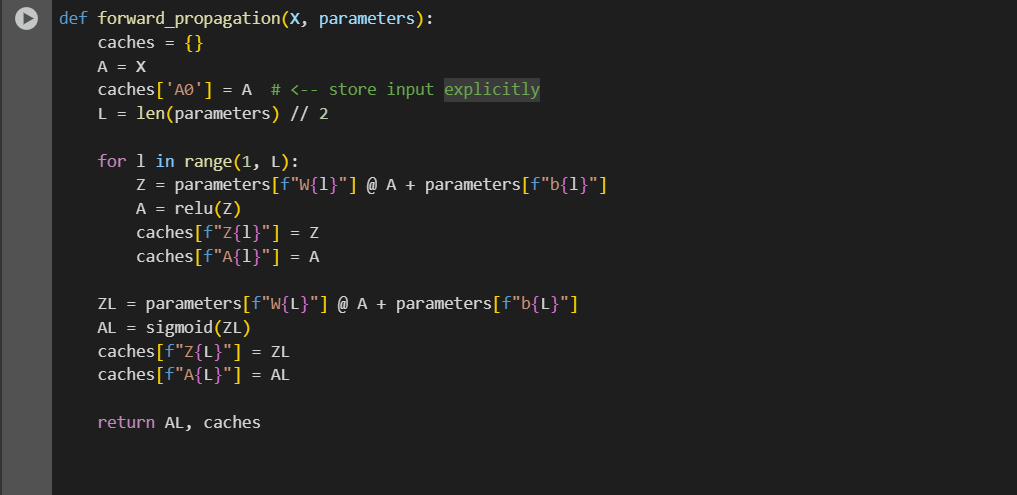
**Sigmoid**: A logistic function that squashes inputs to a range between 0 and 1. Often used for binary classification (like our output layer).

**ReLU (Rectified Linear Unit)**: A function that outputs the input directly if it’s positive; otherwise, it outputs 0. This is commonly used in hidden layers to avoid vanishing gradients.

**Derivatives**:

* **sigmoid\_derivative**: For the output layer, it computes the derivative of the sigmoid function, which is used in the backward pass for weight updates.
* **relu\_derivative**: Computes the derivative of the ReLU function for the backward pass.

***5. Forward Propagation***

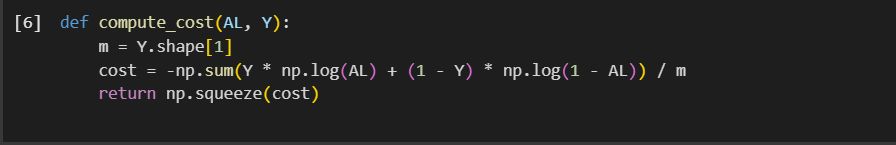
**Input Layer (A0)**: The input X is stored as A0 in the caches dictionary.

**Hidden Layers**:

* The forward pass computes the activation for each layer using the formula
* The activation function (ReLU) is applied to the result of the linear transformation Z.
* Store the intermediate values A (activations) and Z (linear combinations) in caches for later use in backward propagation.

**Output Layer**: The final layer uses the sigmoid function to get a probability value between 0 and 1.

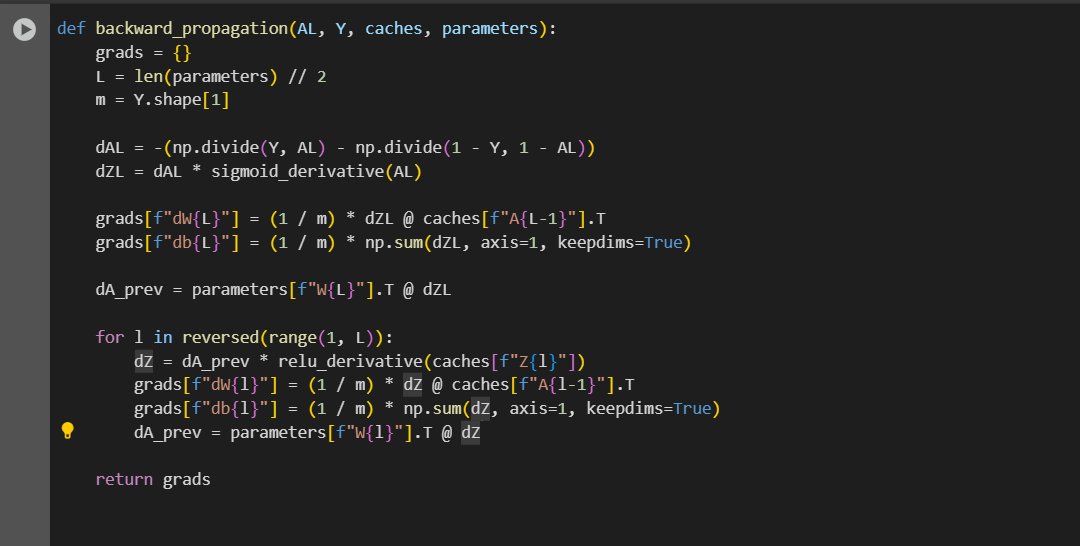
***6. Compute Cost***

**Cost Function**: The cost function is the binary cross-entropy loss, which measures how well the predictions match the actual labels:

where AL is the predicted output, and Y is the actual label.

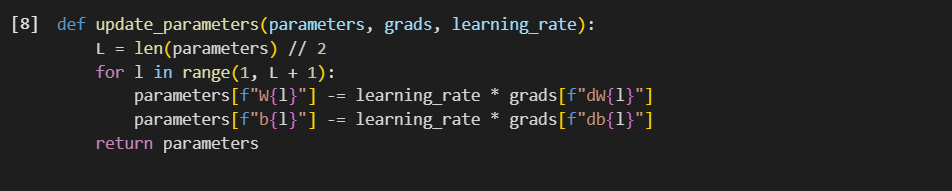
**Why we use this?** Since we are performing binary classification, the binary cross-entropy loss is appropriate for measuring prediction accuracy.

***7. Backward Propagation***

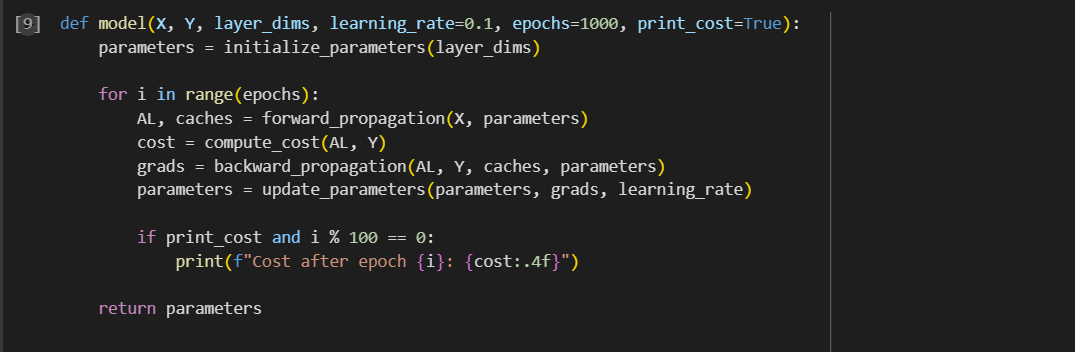
**Backpropagation**:

* **Output Layer**: We first compute the gradient of the cost with respect to the output activation (dAL) and use the derivative of the sigmoid to compute dZL (the error term for the output layer).
* **Hidden Layers**: For each hidden layer, the error (dZ) is computed by propagating the error from the next layer (dA\_prev) and applying the derivative of the ReLU function.
* **Gradients for Weights and Biases**: We compute gradients of the cost with respect to the weights (dW) and biases (db) for all layers.

***8. Update Parameters***

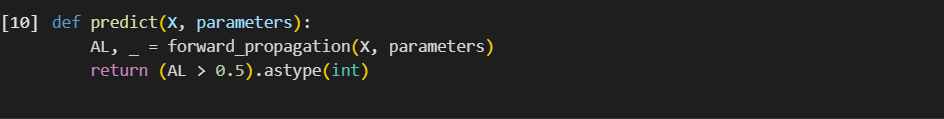
**Gradient Descent**: We update the weights and biases using the gradients calculated during backpropagation. The weights are adjusted by subtracting the gradient scaled by the learning rate.

***9. Train the Model***

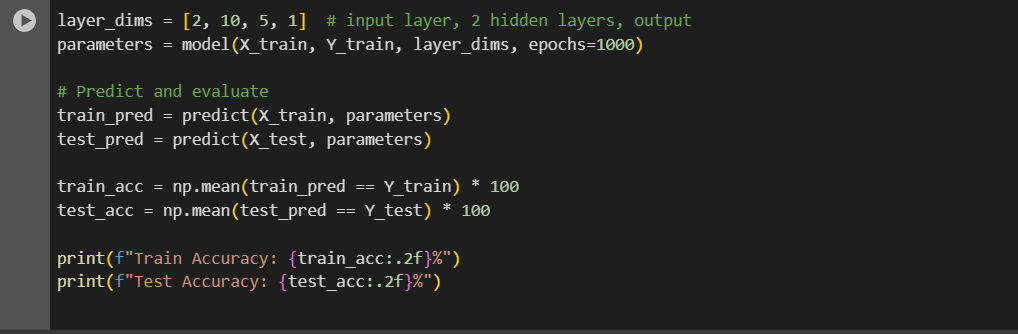
**Model Training**:

* **Loop over epochs**: For each epoch, we perform forward propagation, compute the cost, perform backward propagation to get gradients, and update the parameters using gradient descent.
* **Print cost**: Every 100 epochs, we print the current cost to monitor the training process.

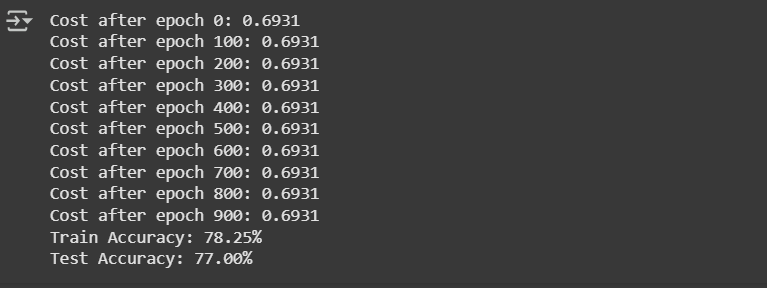
***10. Make Predictions***

**Prediction**: Given the input X and trained parameters, the model predicts the output. If the output is greater than 0.5, it predicts 1, else it predicts 0.

***11. Evaluate***

**Accuracy**: After training, we predict on both the training and test sets and compute the accuracy by comparing the predictions with the actual labels.

**Output:**



**Conclusion:**

This code implements a deep neural network from scratch using NumPy, with key components like forward propagation, backward propagation, cost computation, and gradient descent for training. We can use this as a base for experimenting with different architectures and learning rates