**1. Python Code to Implement a Single Neuron**

import numpy as np

class SingleNeuron:

def \_\_init\_\_(self, input\_size):

# Initialize weights and bias with random values

self.weights = np.random.randn(input\_size)

self.bias = np.random.randn()

def forward(self, inputs):

# Compute the output of the neuron (weighted sum + bias)

return np.dot(inputs, self.weights) + self.bias

# Example usage

neuron = SingleNeuron(3) # A neuron with 3 inputs

inputs = np.array([0.5, -0.3, 0.8])

output = neuron.forward(inputs)

print("Neuron output:", output)

**2. Python Code to Implement ReLU**

import numpy as np

def relu(x):

return np.maximum(0, x)

# Example usage

inputs = np.array([-1, 2, -3, 4])

outputs = relu(inputs)

print("ReLU output:", outputs)

**3. Python Code for a Dense Layer Using Matrix Multiplication**

import numpy as np

class DenseLayer:

def \_\_init\_\_(self, input\_size, output\_size):

# Initialize weights and bias with random values

self.weights = np.random.randn(input\_size, output\_size)

self.bias = np.random.randn(output\_size)

def forward(self, inputs):

# Perform matrix multiplication for the dense layer

return np.dot(inputs, self.weights) + self.bias

# Example usage

dense\_layer = DenseLayer(3, 2) # Dense layer with 3 inputs and 2 outputs

inputs = np.array([0.5, -0.3, 0.8])

output = dense\_layer.forward(inputs)

print("Dense layer output:", output)

**4. Python Code for a Dense Layer in Plain Python (With List Comprehensions)**

import random

class DenseLayerPlain:

def \_\_init\_\_(self, input\_size, output\_size):

# Initialize weights and biases manually

self.weights = [[random.random() for \_ in range(output\_size)] for \_ in range(input\_size)]

self.bias = [random.random() for \_ in range(output\_size)]

def forward(self, inputs):

# Perform matrix multiplication using plain Python (list comprehensions)

return [sum(inputs[i] \* self.weights[i][j] for i in range(len(inputs))) + self.bias[j] for j in range(len(self.bias))]

# Example usage

dense\_layer\_plain = DenseLayerPlain(3, 2)

inputs = [0.5, -0.3, 0.8]

output = dense\_layer\_plain.forward(inputs)

print("Dense layer output (plain Python):", output)

**5. What is the “Hidden Size” of a Layer?**

The **hidden size** of a layer refers to the number of units or neurons in that layer, particularly in a **hidden layer** in a neural network. This is essentially the dimensionality of the output produced by that layer.

For example, if a hidden layer has 100 neurons, the hidden size is 100.

**6. What Does the t Method Do in PyTorch?**

In PyTorch, the .t() method is used to **transpose a 2D tensor**. It swaps the rows and columns of a matrix.

Example:

import torch

tensor = torch.tensor([[1, 2], [3, 4]])

transposed\_tensor = tensor.t()

print(transposed\_tensor)

**7. Why is Matrix Multiplication Written in Plain Python Very Slow?**

Matrix multiplication in plain Python is slow because:

* **Loop-based computation**: Python loops are slower than operations optimized in libraries like NumPy or PyTorch.
* **Inefficiency in memory access**: Plain Python does not utilize optimized low-level operations for efficient memory access or parallel computation.

Libraries like NumPy use optimized C or Fortran code, making operations faster than Python's native list comprehensions or loops.

**8. In matmul, Why is ac == br?**

In matrix multiplication, the dimensions of the matrices must align. Specifically:

* a is of shape (m, n)
* b is of shape (n, p)
* The result matrix c will have the shape (m, p).

For matrix multiplication to be valid, the **number of columns in a** must be equal to the **number of rows in b**, hence ac == br.

**9. In Jupyter Notebook, How Do You Measure the Time Taken for a Single Cell to Execute?**

You can use the %time or %timeit magic commands in Jupyter Notebook:

* **%time**: Measures the time taken to execute a single statement.

%time my\_function()

* **%timeit**: Automatically runs the code multiple times to give an average time.

%timeit my\_function()

**10. What is Elementwise Arithmetic?**

**Elementwise arithmetic** refers to performing arithmetic operations (addition, subtraction, multiplication, division, etc.) on corresponding elements of two arrays (or tensors) of the same shape. Each element in one array is operated on with the corresponding element in the other array.

Example:

import numpy as np

a = np.array([1, 2, 3])

b = np.array([4, 5, 6])

result = a + b # Elementwise addition

print(result)

**11. Write the PyTorch Code to Test Whether Every Element of a is Greater Than the Corresponding Element of b**

import torch

a = torch.tensor([1, 2, 3])

b = torch.tensor([0, 2, 1])

result = a > b

print(result)

**12. What is a Rank-0 Tensor? How Do You Convert It to a Plain Python Data Type?**

A **rank-0 tensor** is a tensor that has no dimensions (essentially a scalar). It represents a single value.

To convert it to a plain Python data type:

import torch

rank0\_tensor = torch.tensor(5)

python\_value = rank0\_tensor.item() # Converts to plain Python scalar

print(python\_value)

**13. How Does Elementwise Arithmetic Help Us Speed Up matmul?**

Elementwise arithmetic allows the **vectorized processing** of each element, enabling **SIMD (Single Instruction, Multiple Data)** optimizations in hardware. This improves computational efficiency and helps avoid slow explicit loops, especially when matrices are large.

For example, elementwise operations can be parallelized at a hardware level, whereas traditional matrix multiplication involves more complex computations that may not benefit from simple parallelization.

**14. What Are the Broadcasting Rules?**

**Broadcasting** allows NumPy (or PyTorch) to perform operations on tensors of different shapes. The smaller tensor is "broadcast" across the larger one to make them compatible. The key rules are:

1. Starting from the trailing dimensions, the dimensions of the tensors must match or be **1**.
2. If one tensor has a dimension of size 1 and the other doesn't, the smaller tensor is broadcast to match the larger tensor’s size.

Example:

import numpy as np

a = np.array([1, 2, 3])

b = np.array([[4], [5], [6]])

result = a + b # Broadcasting a (1,3) with b (3,1)

print(result)

**15. What is expand\_as? Show an Example of How it Can Be Used to Match the Results of Broadcasting.**

expand\_as is used in PyTorch to expand the size of a tensor to match the size of another tensor. It does not create new copies but broadcasts the tensor to match the desired shape.

Example:

import torch

a = torch.tensor([1, 2, 3]) # Shape: (3,)

b = torch.tensor([[4], [5], [6]]) # Shape: (3,1)

expanded\_a = a.unsqueeze(1).expand\_as(b) # Expands 'a' to match the shape of 'b'

result = expanded\_a + b

print(result)

This will produce a matrix where the values of a are broadcasted to match the dimensions of b.