1. Write the Python code to implement a single neuron.

Ans:

import numpy as np

class Neuron:

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

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

self.bias = np.random.rand()

def forward(self, inputs):

z = np.dot(inputs, self.weights) + self.bias

a = self.activation\_function(z)

return a

def activation\_function(self, x):

return 1 / (1 + np.exp(-x))

*# create a neuron with 2 inputs*

neuron = Neuron(2)

*# get input values from the user*

input1 = float(input("Enter input 1: "))

input2 = float(input("Enter input 2: "))

*# make a prediction based on the inputs*

inputs = np.array([input1, input2])

output = neuron.forward(inputs)

print("The predicted output is:", output)

The predicted output is: 0.9978583846876371

2. Write the Python code to implement ReLU.

Ans:

import numpy as np

def relu(x):

return np.maximum(0, x)

*# get input value from the user*

x = float(input("Enter a value for x: "))

*# apply ReLU to the input value*

output = relu(x)

print("The output of ReLU applied to", x, "is:", output)

The output of ReLU applied to 2.7 is: 2.7

3. Write the Python code for a dense layer in terms of matrix multiplication.

Ans:

import numpy as np

class DenseLayer:

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

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

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

def forward(self, inputs):

z = np.dot(inputs, self.weights) + self.bias

return z

*# create a dense layer with 2 inputs and 3 outputs*

layer = DenseLayer(2, 3)

*# get input values from the user*

input1 = float(input("Enter input 1: "))

input2 = float(input("Enter input 2: "))

*# make a prediction based on the inputs*

inputs = np.array([input1, input2])

output = layer.forward(inputs)

print("The predicted output is:", output)

The predicted output is: [3.73812102 3.10664287 3.99025331]

4. Write the Python code for a dense layer in plain Python (that is, with list comprehensions and functionality built into Python).

Ans:

import random

class DenseLayer:

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

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):

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

return z

*# create a dense layer with 2 inputs and 3 outputs*

layer = DenseLayer(2, 3)

*# get input values from the user*

input1 = float(input("Enter input 1: "))

input2 = float(input("Enter input 2: "))

*# make a prediction based on the inputs*

inputs = [input1, input2]

output = layer.forward(inputs)

print("The predicted output is:", output)

The predicted output is: [3.3380793902882337, 6.1582139090639725, 4.882667126685612]

5. What is the “hidden size” of a layer?

Ans: The "hidden size" of a layer refers to the number of neurons in that layer. In other words, it is the number of nodes in the layer that are not directly connected to the input or output layers.

For example, in a neural network with an input layer, an output layer, and a single hidden layer, the hidden size of that layer would refer to the number of neurons in the hidden layer. The input and output layers have a fixed size determined by the input and output of the task being performed, but the number of neurons in the hidden layer can vary and is typically chosen by the user based on the complexity of the task.

The hidden size of a layer is an important hyperparameter that can affect the performance of a neural network. A larger hidden size may allow the network to learn more complex features, but may also increase the risk of overfitting. On the other hand, a smaller hidden size may simplify the network and reduce the risk of overfitting, but may not be able to learn as complex of features.

6. What does the t method do in PyTorch?

Ans:

In PyTorch, the t() method is used to transpose a tensor.

A tensor in PyTorch is a multi-dimensional array, similar to a NumPy array. Transposing a tensor means to swap its rows and columns, effectively rotating the tensor. For example, if we have a 2-dimensional tensor t with shape (3, 4), transposing it would result in a tensor with shape (4, 3).

Here's an example of using the t() method in PyTorch:

import torch

*# create a 2-dimensional tensor*

t = torch.tensor([[1, 2, 3], [4, 5, 6]])

*# print the original tensor*

print("Original tensor:")

print(t)

Original tensor:

tensor([[1, 2, 3],

[4, 5, 6]])

*# transpose the tensor*

t\_transposed = t.t()

*# print the transposed tensor*

print("Transposed tensor:")

print(t\_transposed)

Transposed tensor:

tensor([[1, 4],

[2, 5],

[3, 6]])

7. Why is matrix multiplication written in plain Python very slow?

Ans: Matrix multiplication involves a large number of arithmetic operations, and performing those operations using the basic Python operations can be slow because Python is an interpreted language, which means that it has to interpret and execute each line of code at runtime. This can lead to a significant overhead when working with large matrices.

Moreover, the built-in data structures in Python, such as lists or tuples, are not optimized for numerical operations and lack the efficient memory access patterns that are required for efficient matrix multiplication.

In contrast, specialized numerical libraries such as NumPy or TensorFlow are written in optimized C or Fortran code, which allows for much faster execution of numerical operations. They also offer various optimizations such as parallelism, vectorization, and caching, which further improve the performance of matrix multiplication.

Therefore, if you need to perform matrix multiplication, it's recommended to use a specialized library such as NumPy, which provides optimized functions for matrix operations and can execute them much faster than plain Python.

8. In matmul, why is ac==br?

Ans: In matrix multiplication, two matrices A and B can be multiplied only if the number of columns of A (denoted by 'c') is equal to the number of rows of B (denoted by 'r').

When we write matmul(A,B), A is assumed to be a matrix of shape (a, c) and B is assumed to be a matrix of shape (c, b). The resulting matrix C from the matrix multiplication of A and B would have the shape (a, b).

The element at row 'i' and column 'j' in C can be calculated by taking the dot product of the i-th row of A and the j-th column of B. This requires that the length of the row in A (i.e., 'c') must be equal to the length of the column in B (also 'c'). Therefore, we have the requirement that ac == br in order to perform matrix multiplication using matmul.

In summary, the requirement ac == br ensures that the matrices can be multiplied according to the rules of matrix multiplication, and that the resulting matrix has the expected shape.

9. In Jupyter Notebook, how do you measure the time taken for a single cell to execute?

Ans: To measure the time taken for a single cell to execute in a Jupyter Notebook, you can use the %%timeit magic command. This command runs the cell multiple times and provides the average time taken for the cell to execute.

To use the %%timeit command, you simply need to add it at the beginning of the cell you want to time. For example:

%%timeit

# Your code here

Once you run the cell with the %%timeit command, Jupyter Notebook will execute the cell multiple times with different inputs and record the time taken for each execution. It will then calculate the average time taken and display it along with other statistics such as standard deviation and confidence intervals.

Alternatively, you can also use the %time command to measure the time taken for a single execution of a cell. This command provides more detailed information than %%timeit, but it only runs the cell once. To use %time, simply add it at the beginning of the cell you want to time, like this:

%time

# Your code here

After running the cell with %time, Jupyter Notebook will display the time taken for the cell to execute, along with other detailed information about the execution, such as CPU time and memory usage.

10. What is elementwise arithmetic?

Ans: Elementwise arithmetic refers to performing arithmetic operations on the corresponding elements of two or more arrays or matrices. In elementwise arithmetic, each element of one array is paired with the corresponding element of another array, and the arithmetic operation is performed on those pairs of elements.

For example, given two arrays A and B, elementwise addition would involve adding each element of A to the corresponding element of B, resulting in a new array C where each element is the sum of the corresponding elements in A and B.

Elementwise arithmetic is commonly used in numerical computations, particularly in linear algebra and machine learning. In Python, libraries such as NumPy provide efficient support for elementwise arithmetic operations on arrays and matrices.

Elementwise arithmetic is different from matrix multiplication, where each element in the resulting matrix is the sum of the products of the corresponding elements of two input matrices. In matrix multiplication, the input matrices must have compatible shapes, whereas in elementwise arithmetic, the input arrays must have the same shape.

11. Write the PyTorch code to test whether every element of a is greater than the corresponding element of b.

Ans:

import torch

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

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

result = torch.gt(a, b)

if torch.all(result):

print("All elements of a are greater than b")

else:

print("Not all elements of a are greater than b")

print("Indices where the condition fails: ", torch.where(result == False))

Not all elements of a are greater than b

Indices where the condition fails: (tensor([1]),)

12. What is a rank-0 tensor? How do you convert it to a plain Python data type?

Ans: A rank-0 tensor is a tensor with zero dimensions, also known as a scalar or a 0-d tensor. It represents a single numerical value, such as an integer or a floating-point number.

In PyTorch, a rank-0 tensor is created using the torch.tensor() function with a single argument, which is the scalar value:

import torch

x = torch.tensor(42)

print(x)

tensor(42)

This creates a rank-0 tensor x with the value of 42.

To convert a rank-0 tensor to a plain Python data type, you can use the .item() method. This method returns the scalar value of the tensor as a Python primitive type, such as an integer or a floating-point number:

import torch

x = torch.tensor(42)

y = x.item()

print(type(y))

print(y)

<class 'int'>

42

In this example, y is assigned the value of 42 as a plain Python integer. The type(y) call confirms that y is a Python integer. Note that the .item() method only works on rank-0 tensors, and will raise an error if called on tensors with higher rank.

13. How does elementwise arithmetic help us speed up matmul?

Ans: Element-wise arithmetic can help speed up matrix multiplication by reducing the number of operations required to compute the result. Specifically, element-wise arithmetic can be used to implement a faster version of matrix multiplication known as “broadcasting matrix multiplication.”

In broadcasting matrix multiplication, the two matrices being multiplied are broadcasted to have the same shape. This allows us to perform element-wise multiplication between the two matrices, followed by a sum along one of the axes to compute the final result.

This approach can be much faster than traditional matrix multiplication because it takes advantage of the parallel processing capabilities of modern hardware. By performing many element-wise operations in parallel, we can significantly reduce the time it takes to compute the result of a matrix multiplication.

Here’s an example that shows how element-wise arithmetic can be used to implement a faster version of matrix multiplication:

14. What are the broadcasting rules?

Ans: Broadcasting is a powerful mechanism in NumPy and PyTorch that allows you to perform operations between tensors of different shapes. The broadcasting rules determine how tensors with different shapes are treated during arithmetic operations.

Here are the broadcasting rules:

If the two tensors have different numbers of dimensions, the shape of the tensor with fewer dimensions is padded with ones on its leading (left) side.

If the shape of the two tensors does not match in any dimension, the tensor with shape equal to 1 in that dimension is stretched to match the other shape.

If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

These rules allow you to perform element-wise operations between tensors of different shapes in a flexible and intuitive manner.

import torch

*# Create two matrices*

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

b = torch.tensor([[5, 6], [7, 8]])

*# Compute the result of matrix multiplication using broadcasting and element-wise arithmetic*

result = (a.unsqueeze(-1) \* b.unsqueeze(0)).sum(dim=1)

print(result)

tensor([[19, 22],

[43, 50]])

In this example, we use unsqueeze to add an extra dimension to a and b, allowing us to broadcast them to have the same shape. We then perform element-wise multiplication between a and b, followed by a sum along dimension 1 to compute the final result. This approach is much faster than traditional matrix multiplication for large matrices.

15. What is expand\_as? Show an example of how it can be used to match the results of broadcasting.

Ans: expand\_as is a PyTorch tensor method that allows you to expand a tensor to the size of another tensor. This can be useful when you want to perform operations between tensors of different sizes that are not broadcastable.

Here’s an example that shows how expand\_as can be used to match the results of broadcasting:

import torch

*# Create two tensors of different sizes*

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

y = torch.tensor([4, 5, 6])

*# Use expand\_as to expand x to the size of the result of broadcasting x and y*

x\_expanded = x.expand\_as(x + y)

*# Perform element-wise addition between x\_expanded and y*

result = x\_expanded + y

print(result)

tensor([[5, 6, 7],

[6, 7, 8],

[7, 8, 9]])