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

Answer:--->Here's a simple implementation of a single neuron in Python:

class Neuron:

```
def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias

def activate(self, inputs):
    weighted_inputs = [inputs[i] * self.weights[i] for i in range(len(inputs))]
    activation = sum(weighted_inputs) + self.bias
    return activation
```

weights = [0.1, 0.2, 0.3] bias = 0.4 neuron = Neuron(weights, bias)

inputs = [1.0, 2.0, 3.0] output = neuron.activate(inputs) print(output)

This implementation of a single neuron takes in a list of weights and a single bias value and uses them to calculate the activation of the neuron given some input inputs. The activate method multiplies each input by its corresponding weight, sums the weighted inputs, adds the bias, and returns the activation. The code then creates an instance of the Neuron class with some example weights and a bias, and uses the activate method to calculate the activation of the neuron given some example inputs.

2. Write the Python code to implement ReLU.

Answer:---->def relu(x): return max(0, x)

input = -0.5

output = relu(input)

print(output)

The relu function takes in a single argument x and returns the value of x if x is positive, and returns 0 if x is negative. The code then uses the relu function to calculate the output for an example input of -0.5.

Note that this is a simplified implementation of ReLU for demonstration purposes. In practice, you would typically use a library such as TensorFlow to implement ReLU as part of a larger neural network.

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

Answer:---->import numpy as np

class DenseLayer:

```
def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias

def call(self, inputs):
    outputs = np.dot(inputs, self.weights) + self.bias
    return outputs
```

weights = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])

bias = np.array([0.1, 0.2])

layer = DenseLayer(weights, bias)

inputs = np.array([1.0, 2.0, 3.0])

outputs = layer.call(inputs)

print(outputs)

This implementation of a dense layer takes in a matrix of weights and a vector of bias values, and uses them to calculate the outputs of the layer given some input inputs. The call method calculates the matrix product of the inputs and the weights, adds the bias, and returns the outputs. The code then creates an instance of the DenseLayer class with some example weights and a bias, and uses the call method to calculate the outputs of the layer given some example inputs.

5. What is the "hidden size" of a layer?

Answer:---->The size of the hidden layer is normally between the size of the input and output-. It should be should be 2/3 the size of the input layerplus the size of the o/p layer The number of hidden neurons should be less than twice the size of the input layer.A hidden layer in an artificial neural network is a layer in between input layers and output layers, where artificial neurons take in a set of weighted inputs and produce an output through an activation function.The "hidden size" of a layer refers to the number of neurons or units in the layer. Each neuron or unit in a layer is responsible for processing a portion of the input data, and the output of each neuron is used as input for the next layer. The hidden size determines the number of neurons in the layer, and therefore the capacity of the layer to learn complex representations of the input data.

6. What does the t method do in PyTorch?

Answer:----->Expects input to be <= 2-D tensor and transposes dimensions 0 and 1. 0-D and 1-D tensors are returned as is. When input is a 2-D tensor this is equivalent to transpose(input, 0, 1) .A PyTorch Tensor is basically the same as a numpy array: it does not know anything about deep learning or computational graphs or gradients, and is just a generic n-dimensional array to be used for arbitrary numeric computation.

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

Answer:---->

8. In matmul, why is ac==br?

Answer:---->Rows come first, so first matrix provides row numbers. Columns come second, so second matrix provide column numbers. Matrix multiplication is really just a way of organizing vectors we want to find the dot product .The product of two matrices A and B is defined if the number of columns of A is equal to the number of rows of B. If both A and B are square matrices of the same order, then both AB and BA are defined. If AB and BA are both defined, it is not necessary that AB = BA.In general, AB = BA, even if A and B are both square.A+B = B+A →Commutative Law of Addition. A+B+C = A +(B+C) = (A+B)+C →Associative law of addition.

ABC = A(BC) = (AB)C →Associative law of multiplication.

A(B+C) = AB + AC →Distributive law of matrix algebra.

R(A+B) = RA + RB.

A modulus function is a function which gives the absolute value of a number or variable. It produces the magnitude of the number of variables. It is also termed as an absolute value function. The outcome of this function is always positive, no matter what input has been given to the function.

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

Answer:---->In Jupyter Notebook (IPython), you can use the magic commands %timeit and %%timeit to measure the execution time of your code.

Copy code import time start = time.time()

[code cell to be executed]

end = time.time() print("Time taken to execute the cell:", end - start, "seconds")

Example 1: Using time module Save the timestamp at the beginning of the code start using time() .

Save the timestamp at the end of the code end .

Find the difference between the end and start, which gives the execution time

10. What is elementwise arithmetic?

Answer:--->An element-wise operation is an operation between two tensors that operates on corresponding elements within the respective tensors. An element-wise operation operates on corresponding elements between tensors. Two elements are said to be corresponding if the two elements occupy the same position within the tensor.In mathematics, the Hadamard product (also known as the element-wise product, entrywise product or Schur product) is a binary operation that takes two matrices of the same dimensions and produces another matrix of the same dimension as the operands, where each element i, j is the product of elements i, j. Elementwise multiplication is also known as the Schur or Hadamard product. Elementwise multiplication (which uses the # operator) should not be confused with matrix multiplication (which uses the * operator).Elementwise functions apply a function to each element of a vector or matrix, returning a result of the same shape as the argument. There are many functions that are vectorized in addition to the ad hoc cases listed in this section; see section function vectorizationfor the general cases. Element-wise means handling data element by element.

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

Answer;-----> You can use the torch.gt function from PyTorch to test whether every element of tensor a is greater than the corresponding element of tensor b. The torch.gt function returns a tensor of the same shape as the input tensors, where each element is 1 if the corresponding element of a is greater than the corresponding element of b, and 0 otherwise.

Here's the code to implement this:

import torch

a = torch.tensor([1, 2, 3, 4, 5]) b = torch.tensor([0, 1, 2, 3, 4])

result = torch.gt(a, b) print(result) This will output:

tensor([1, 1, 1, 1, 1], dtype=torch.uint8) The result is a tensor of type torch.uint8, with each element being 1 or 0, indicating whether the corresponding element of a is greater than the corresponding element of b.

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

Answer:----->The rank of a null matrix is zero. A null matrix has no non-zero rows or columns. So, there are no independent rows or columns. Hence the rank of a null matrix is zero.The rank of a tensor is the number of indices required to uniquely select each element of the tensor. Rank is also known as "order", "degree", or "ndims."A tensor with rank 0 is a zero-dimensional array. The element of a zero-dimensional array is a point. This is represented as a Scalar in Math and has magnitude. A tensor with rank 1 is a one-dimensional array.The term rank of a tensor extends the notion of the rank of a matrix in linear algebra, although the term is also often used to mean the order (or degree) of a tensor. The rank of a matrix is the minimum number of column vectors needed to span the range of the matrix.The rank of a tensor tells us how many indexes are required to access (refer to) a specific data element contained within the tensor data structure. A tensor's rank tells us how many indexes are needed to refer to a specific element within the tensor.

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

Answer:----->Elementwise arithmetic is used to speed up matrix multiplication (matmul) in PyTorch by taking advantage of parallel processing.

In traditional matrix multiplication, the result is calculated by taking the dot product of each row of the first matrix with each column of the second matrix. However, this requires a lot of computation and can be slow for large matrices.

With elementwise arithmetic, we can perform multiple operations at the same time, which allows us to take advantage of parallel processing and speed up the calculation. This is because many operations in deep learning are elementwise operations, such as addition, subtraction, and multiplication.

For example, we can use elementwise arithmetic to perform multiple dot products at the same time, which reduces the overall computation time compared to performing each dot product sequentially.

In PyTorch, this can be done by using the torch.mul and torch.sum functions, which perform elementwise multiplication and elementwise sum, respectively. By using these functions, we can perform matrix multiplication in a more efficient manner, as the operations are performed in parallel.

Here's an example of using elementwise arithmetic to perform matrix multiplication in PyTorch:

import torch

a = torch.tensor([[1, 2], [3, 4]]) b = torch.tensor([[5, 6], [7, 8]])

result = torch.zeros((2, 2))

for i in range(2): for j in range(2): result[i, j] = torch.sum(torch.mul(a[i, :], b[:, j]))

print(result)

This will output:

tensor([[19., 22.], [43., 50.]])

As you can see, the elementwise arithmetic approach to matrix multiplication is more efficient than performing the dot product operation one by one, and is therefore a useful technique for speeding up matrix multiplications in PyTorch.

14. What are the broadcasting rules?

Answer:----->The Cable Television Networks (Regulation) Act, 1995, Amendment Act 2011, Telecom Regulatory Authority of India Act, 1997, and Prasar Bharati Act, 1990 are the major laws that govern broadcasting in India.

Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.

Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised

The field of broadcasting includes both government-managed services such as public radio, community radio and public television, and private commercial radio and commercial television.Broadcasting rights (often also called media rights) are rights which a broadcasting organization negotiates with a commercial concern - such as a sports governing body or film distributor - in order to show that company's products on television or radio, either live, delayed or highlights.

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

Answer:---->expand_as is a PyTorch function that is used to expand the shape of a tensor to match the shape of another tensor. The expand_as function allows you to expand the shape of a tensor in a way that is consistent with the shape of another tensor, without having to manually specify the shape of the output tensor.

For example, if you have two tensors a and b and you want to perform elementwise multiplication, you can use broadcasting to perform the calculation. However, if the shapes of a and b are not the same, you will need to manually expand the shape of one of the tensors to match the shape of the other tensor before performing the calculation.

With the expand_as function, you can easily match the shapes of the two tensors without having to manually specify the shape of the output tensor. The function will automatically expand the shape of the tensor to match the shape of the other tensor, as long as the number of dimensions of the two tensors are the same.