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

import numpy as np

class Neuron:

def \_\_init\_\_(self, num\_inputs):

# Initialize weights randomly

self.weights = np.random.randn(num\_inputs)

# Initialize bias randomly

self.bias = np.random.randn()

def forward(self, inputs):

# Perform weighted sum of inputs

weighted\_sum = np.dot(inputs, self.weights) + self.bias

# Apply activation function (e.g., sigmoid)

activation = self.sigmoid(weighted\_sum)

return activation

def sigmoid(self, x):

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

# Example usage

neuron = Neuron(2) # Create a neuron with 2 inputs

inputs = np.array([0.5, 0.7]) # Input values

output = neuron.forward(inputs) # Compute the output of the neuron

print("Neuron output:", output)

1. Write the Python code to implement ReLU.

def relu(x):

return max(0, x)

# Example usage

input\_value = 5

output = relu(input\_value)

print("ReLU output:", output)

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

import numpy as np

class DenseLayer:

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

# Initialize weights and bias randomly

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

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

def forward(self, inputs):

# Perform matrix multiplication and add bias

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

return outputs

# Example usage

dense\_layer = DenseLayer(input\_size=3, output\_size=2) # Create a dense layer with 3 inputs and 2 outputs

inputs = np.array([1, 2, 3]) # Input values

outputs = dense\_layer.forward(inputs) # Compute the outputs of the dense layer

print("Dense layer outputs:", outputs)

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

import random

class DenseLayer:

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

# Initialize weights and bias randomly

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

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

def forward(self, inputs):

# Perform matrix multiplication and add bias using list comprehensions

outputs = [sum(w \* x for w, x in zip(weights, inputs)) + b for weights, b in zip(self.weights, self.bias)]

return outputs

# Example usage

dense\_layer = DenseLayer(input\_size=3, output\_size=2) # Create a dense layer with 3 inputs and 2 outputs

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

The "hidden size" of a layer refers to the number of neurons or units in that particular layer. In a neural network, each layer consists of a set of neurons or units that perform computations on the input data. The hidden size determines the dimensionality or the number of features extracted by that layer.

1. What does the t method do in PyTorch?

In PyTorch, the t method is used to transpose a tensor. Transposing a tensor changes the order of its dimensions. It swaps the rows and columns of a 2D tensor or reorders the dimensions of a tensor with more than two dimensions.

import torch

# Create a 2D tensor

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

[4, 5, 6]])

# Transpose the tensor using the t method

x\_t = x.t()

print("Original tensor:")

print(x)

print("Transposed tensor:")

print(x\_t)

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

Matrix multiplication implemented in plain Python can be slow for several reasons:

1. **Interpreted Execution**: Python is an interpreted language, which means that each line of code is executed one at a time. This leads to slower execution compared to compiled languages like C or C++. In matrix multiplication, where there are many individual calculations to be performed, this interpretation overhead can significantly impact performance.
2. **Lack of Vectorization**: Plain Python does not inherently support vectorized operations, which are crucial for efficient matrix multiplication. In vectorized operations, computations are performed on entire arrays or matrices rather than individual elements. This allows for optimized execution using lower-level, highly efficient libraries and hardware optimizations. In plain Python, matrix multiplication is performed element-wise, resulting in slower execution.
3. **No Native Multithreading**: Plain Python does not provide native support for multithreading, which can limit parallelism and hinder performance in matrix multiplication. Multithreading allows for concurrent execution of multiple operations, which can significantly speed up calculations. Without multithreading, matrix multiplication in plain Python is executed serially, which can be inefficient for large matrices.
4. **No Optimization for Matrix Operations**: Plain Python does not have built-in optimizations specifically tailored for matrix operations. Libraries like NumPy and optimized frameworks like TensorFlow and PyTorch leverage low-level optimizations and utilize specialized linear algebra libraries (e.g., BLAS or MKL) to accelerate matrix multiplication. These libraries are typically implemented in lower-level languages like C or Fortran, providing highly efficient implementations of matrix operations.
5. In matmul, why is ac==br?

In matrix multiplication (matmul), the dimensions of the matrices involved must satisfy the condition ac == br to ensure compatibility and enable the operation to be performed.

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

In Jupyter Notebook, you can measure the time taken for a single cell to execute using the %%timeit magic command or the %%time magic command.

%%timeit:

* Place %%timeit at the beginning of the cell you want to time.
* When you run the cell, %%timeit will execute the code multiple times and measure the average execution time.
* The output will display the average time taken and the number of loops executed.

%%time:

* Place %%time at the beginning of the cell you want to time.
* When you run the cell, %%time will execute the code once and measure the execution time.
* The output will display the total time taken.

1. What is elementwise arithmetic?

Elementwise arithmetic refers to performing arithmetic operations on corresponding elements of two or more arrays or matrices. In this operation, the arithmetic operation is applied individually to each element without considering the relationship between elements in different arrays or matrices.

Elementwise arithmetic operations can be performed on arrays or matrices of the same shape, where the operation is applied element by element. The operation is applied to the corresponding elements based on their indices. For example, if you have two arrays A and B, the elementwise addition would involve adding the elements with the same indices to produce a new array C.

Here are some common elementwise arithmetic operations:

* Elementwise Addition: C = A + B
* Elementwise Subtraction: C = A - B
* Elementwise Multiplication: C = A \* B
* Elementwise Division: C = A / B
* Elementwise Exponentiation: C = A \*\*

1. 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, 2])

result = torch.all(a > b)

print(result)

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

In TensorFlow and PyTorch, a rank-0 tensor refers to a scalar tensor, which represents a single value with no dimensions. It does not have any shape or size.

To convert a rank-0 tensor to a plain Python data type, you can use the .item() method available in both TensorFlow and PyTorch. This method extracts the value from the rank-0 tensor and returns it as a native Python data type.

Here's an example of converting a rank-0 tensor to a plain Python data type:

import torch

x = torch.tensor(5) # Create a rank-0 tensor with value 5

x\_value = x.item() # Convert rank-0 tensor to a plain Python data type

print(x\_value)

print(type(x\_value))

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

Elementwise arithmetic operations, such as addition and multiplication, do not directly speed up the matrix multiplication (matmul) operation itself. Instead, they can be utilized as building blocks in optimized implementations of matrix multiplication algorithms to improve efficiency and performance.

Here's how elementwise arithmetic can contribute to speeding up matmul:

1. Vectorization: Elementwise arithmetic operations can be performed on entire arrays or matrices, taking advantage of hardware-level optimizations and optimized libraries like NumPy or deep learning frameworks like TensorFlow or PyTorch. These libraries are designed to efficiently execute vectorized operations using low-level, highly optimized code, enabling faster execution of elementwise operations and reducing the overall computational time of matmul.
2. Parallelism: Many modern hardware architectures support parallel execution of elementwise arithmetic operations. Utilizing parallel computing techniques, such as multi-threading or utilizing GPUs, can enable simultaneous execution of elementwise operations across multiple cores or processing units, effectively reducing the time taken to perform the elementwise computations and, subsequently, speeding up matmul.
3. Intermediate Computations: During the computation of matmul, there are intermediate results obtained from elementwise operations, such as elementwise multiplication and addition, that can be reused or optimized. These intermediate results can be stored in memory and reused in subsequent computations, reducing redundant calculations and minimizing memory access.
4. What are the broadcasting rules?

Broadcasting is a concept in NumPy and other array-based libraries that allows for implicit elementwise operations between arrays with different shapes. The broadcasting rules define how arrays with different shapes can be broadcasted to perform elementwise operations efficiently.

The broadcasting rules can be summarized as follows:

1. Rule 1: Dimensions of size 1: If the arrays have different numbers of dimensions, the array with fewer dimensions is expanded by inserting dimensions of size 1 at the appropriate locations. This is done to make the shapes of the arrays compatible for elementwise operations.
2. Rule 2: Dimensions of size > 1: If the arrays have at least one dimension of size greater than 1, the arrays' shapes must be compatible. Two dimensions are compatible if they are equal or one of them is 1. When the dimensions are not equal, the array with size 1 in that dimension is stretched or repeated to match the size of the other array.
3. Rule 3: Output shape: The resulting shape of the elementwise operation is determined by taking the elementwise maximum along each dimension. If the sizes of the corresponding dimensions are not equal, one of them must be 1, or an error is raised.
4. What is expand\_as? Show an example of how it can be used to match the results of broadcasting.

In PyTorch, the expand\_as method is used to expand the size of a tensor to match the size of another tensor. It is a convenient way to apply broadcasting-like behavior manually.

The expand\_as method takes another tensor as an argument and expands the size of the calling tensor to match the shape of the provided tensor. It replicates the dimensions of the calling tensor to match the dimensions of the provided tensor.