1. Write the Python code to implement a single neuron.
2. Write the Python code to implement ReLU.
3. Write the Python code for a dense layer in terms of matrix multiplication.
4. Write the Python code for a dense layer in plain Python (that is, with list comprehensions and functionality built into Python).
5. What is the “hidden size” of a layer?
6. What does the t method do in PyTorch?
7. Why is matrix multiplication written in plain Python very slow?
8. In matmul, why is ac==br?
9. In Jupyter Notebook, how do you measure the time taken for a single cell to execute?
10. What is elementwise arithmetic?
11. Write the PyTorch code to test whether every 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?
13. How does elementwise arithmetic help us speed up matmul?
14. What are the broadcasting rules?
15. What is expand\_as? Show an example of how it can be used to match the results of broadcasting.

Answer:

import numpy as np

class Neuron:

def \_\_init\_\_(self, weights, bias):

self.weights = weights

self.bias = bias

def forward(self, inputs):

# Compute weighted sum of inputs, add bias and apply activation function

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

def activation(self, x):

# Sigmoid activation function

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

2.

import numpy as np

def relu(x):

return np.maximum(0, x)

3.

import torch

class DenseLayer(torch.nn.Module):

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

super(DenseLayer, self).\_\_init\_\_()

self.weights = torch.nn.Parameter(torch.randn(input\_size, output\_size))

self.bias = torch.nn.Parameter(torch.randn(output\_size))

def forward(self, inputs):

return torch.matmul(inputs, self.weights) + self.bias

4.

def dense\_layer(inputs, weights, bias):

return [sum(x \* w for x, w in zip(inputs, weights[i])) + bias[i] for i in range(len(bias))]

1. The "hidden size" of a layer is the number of neurons (or units) in that layer. It refers to the number of outputs produced by the layer.
2. The t method in PyTorch is used to transpose a tensor. For example, if **x** is a tensor with shape **(3, 4)**, **x.t()** will return a tensor with shape **(4, 3)**.
3. Matrix multiplication written in plain Python is slow because it involves iterating over nested loops, which is inefficient for large matrices. On the other hand, libraries like NumPy or PyTorch use highly optimized C or CUDA code to perform matrix multiplication, which is much faster.
4. In **matmul**, **ac==br** is a requirement for matrix multiplication to be defined. Specifically, it means that the number of columns in the left matrix must be equal to the number of rows in the right matrix. This is required for the dot product operation to be defined.
5. In Jupyter Notebook, you can measure the time taken for a single cell to execute by using the **%timeit** magic command. For example:

%timeit my\_function()

This will run **my\_function** a number of times and output the average time taken.

1. Elementwise arithmetic is an operation that applies a scalar function to each element of a tensor independently. For example, adding a scalar value to a tensor or taking the square root of each element are elementwise arithmetic operations.

import torch

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

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

result = torch.all(a > b)

print(result)

1. A rank-0 tensor is a tensor with zero dimensions, also known as a scalar. You can convert a rank-0 tensor to a plain Python data type using the **.item()** method. For example, if **x** is a rank-0 tensor, you can convert it to a Python float using **x.item()**.
2. Elementwise arithmetic helps us speed up matmul by reducing the number of multiplications required. Matmul between two matrices involves multiplying each element of each row of the first matrix with each corresponding element of each column of the second matrix, and summing the products. By using elementwise multiplication, we can perform fewer operations by multiplying only the elements we need, rather than multiplying all the combinations of elements.
3. Broadcasting rules in PyTorch allow us to perform operations between tensors of different shapes, by automatically expanding or "broadcasting" the smaller tensor to match the shape of the larger tensor. The broadcasting rules are:

* If the two tensors have different ranks, prepend the smaller tensor with dimensions of size 1 until they have the same rank.
* For each dimension where one tensor has size 1 and the other tensor has size greater than 1, the tensor with size 1 is repeated along that dimension to match the size of the other tensor.
* If the two tensors have the same size for a given dimension, or if one of the tensors has size 1 for that dimension, the tensors are compatible for broadcasting.

1. The **expand\_as** method in PyTorch expands a tensor to match the shape of another tensor, specified as an argument. For example, if **x** is a rank-1 tensor of shape **(3,)** and **y** is a rank-2 tensor of shape **(2, 3)**, we can use **expand\_as** to expand **x** to the shape of **y** as follows:

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

y = torch.tensor([[4, 5, 6], [7, 8, 9]])

z = x.expand\_as(y)

This will result in **z** having shape **(2, 3)** and being equal to **torch.tensor([[1, 2, 3], [1, 2, 3]])**. We can use **expand\_as** to match the shapes of tensors for broadcasting operations.