**1. How Does unsqueeze Help Us Solve Certain Broadcasting Problems?**

The unsqueeze function is used to add a new dimension to a tensor. This is particularly useful in broadcasting because it allows tensors with incompatible shapes to be aligned in a way that makes them compatible for element-wise operations.

For example, if you want to add a 1D tensor to a 2D tensor, unsqueeze can add a new dimension to the 1D tensor, making it broadcast correctly with the 2D tensor.

Example:

import torch

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

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

# Use unsqueeze to align the dimensions

a\_unsqueezed = a.unsqueeze(1) # Shape becomes (3,1)

result = a\_unsqueezed + b # Broadcasting works now

print(result)

**2. How Can We Use Indexing to Do the Same Operation as unsqueeze?**

You can use indexing to achieve the same result as unsqueeze. Specifically, you can use None or : to add a new axis.

Example:

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

a\_unsqueezed = a[:, None] # Adds a new axis, shape becomes (3,1)

This is equivalent to using unsqueeze(1).

**3. How Do We Show the Actual Contents of the Memory Used for a Tensor?**

To show the actual contents of the memory used by a tensor, you can use the .data\_ptr() method, which returns the memory address of the first element in the tensor. However, to inspect the actual values, simply print the tensor itself:

import torch

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

print(tensor) # Shows the contents

print(tensor.data\_ptr()) # Shows the memory address

**4. When Adding a Vector of Size 3 to a Matrix of Size 3×3, Are the Elements of the Vector Added to Each Row or Each Column of the Matrix?**

When adding a vector of size (3,) to a matrix of size (3, 3), the vector is added **to each row** of the matrix.

Example:

import torch

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

matrix = torch.tensor([[4, 5, 6], [7, 8, 9], [10, 11, 12]]) # Shape (3,3)

result = matrix + vector

print(result)

Output:

tensor([[ 5, 7, 9],

[ 8, 10, 12],

[11, 13, 15]])

Here, the vector [1, 2, 3] is added to each row of the matrix.

**5. Do Broadcasting and expand\_as Result in Increased Memory Use? Why or Why Not?**

**No**, broadcasting and expand\_as do not result in increased memory use. Both operations are **view-based** rather than copying data. Broadcasting works by "virtually" expanding the dimensions of the smaller tensor, and expand\_as provides a view that mimics the expanded shape without actually duplicating the data.

**6. Implement matmul Using Einstein Summation**

Einstein summation allows us to express matrix multiplication concisely. Here's how you can implement matmul using einsum:

import torch

# Two matrices for multiplication

A = torch.randn(3, 2)

B = torch.randn(2, 3)

# Implement matmul using einsum

C = torch.einsum('ik,kj->ij', A, B)

print(C)

**7. What Does a Repeated Index Letter Represent on the Lefthand Side of einsum?**

A repeated index letter in Einstein summation notation represents **summation over that index**. The repeated index corresponds to the dimensions that are summed over when performing the operation.

For example, in einsum('ik,kj->ij', A, B), the k is repeated, which means the summation happens over the k dimension during matrix multiplication.

**8. What Are the Three Rules of Einstein Summation Notation? Why?**

The three main rules of Einstein summation notation are:

1. **Implicit summation**: When an index appears twice (once on the left and once on the right), it means summing over that index.
2. **No summation over unindexed dimensions**: If an index only appears once, it is not summed over, and it represents the resulting dimension in the output.
3. **Index contraction**: For matrix multiplication or other operations, index contraction means that repeated indices represent a summation over them.

These rules simplify the notation for complex tensor operations, making them more concise and easier to understand.

**9. What Are the Forward Pass and Backward Pass of a Neural Network?**

* **Forward Pass**: The forward pass is the process where input data is passed through the network, layer by layer, to compute the output.
* **Backward Pass**: The backward pass involves computing the gradients of the loss with respect to the model's parameters using backpropagation. This is done by applying the chain rule to propagate the error back through the network and update the weights.

**10. Why Do We Need to Store Some of the Activations Calculated for Intermediate Layers in the Forward Pass?**

We store intermediate activations because they are required to compute the gradients during the backward pass. Without these activations, we wouldn't have the necessary data to calculate how each parameter should be updated.

**11. What is the Downside of Having Activations with a Standard Deviation Too Far Away from 1?**

If activations have a standard deviation too far from 1 (either too large or too small), it can lead to the **vanishing or exploding gradient problem**. This makes training deep networks harder because:

* Large activations may cause exploding gradients, leading to numerical instability.
* Small activations may cause vanishing gradients, leading to slow or no learning.

**12. How Can Weight Initialization Help Avoid This Problem?**

Proper **weight initialization** can mitigate the vanishing and exploding gradient problems. Techniques such as **Xavier/Glorot initialization** (for sigmoid or tanh activations) and **He initialization** (for ReLU activations) are designed to keep the variance of activations and gradients within reasonable bounds, making it easier to train deep networks.