1.How does unsqueeze help us to solve certain broadcasting problems?

Answer: Unsqueeze is a PyTorch function that allows us to add a new dimension to a tensor. This can be useful in broadcasting operations where we want to perform element-wise operations between tensors of different shapes. By adding a new dimension to a tensor using unsqueeze, we can align the shapes of two tensors for element-wise operations.

2.How can we use indexing to do the same operation as unsqueeze?

Answer: We can use indexing in PyTorch to add a new dimension to a tensor by using the None keyword. For example, we can add a new dimension to a 1D tensor by indexing it with tensor[:, None]. This is equivalent to using unsqueeze with dim=1.

3.How do we show the actual contents of the memory used for a tensor?

Answer: In PyTorch, we can use the numpy function on a tensor to convert it to a NumPy array, which can then be printed to show the actual contents of the memory used for the tensor. For example, we can use print(tensor.numpy()) to show the contents of a PyTorch tensor.

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? (Be sure to check your answer by running this code in a notebook.)

Answer: When adding a vector of size 3 to a matrix of size 3×3, the elements of the vector are added to each column of the matrix. This can be verified by running the following code in a notebook:

import torch

matrix = torch.ones(3, 3)

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

result = matrix + vector

print(result)

The output will show that each column of the matrix has been increased by the corresponding element of the vector.

5.Do broadcasting and expand\_as result in increased memory use? Why or why not?

Answer: Broadcasting and expand\_as do not result in increased memory use. This is because they do not create new tensors or copy data to new memory locations. Instead, they create views of existing tensors with modified shapes or sizes. This means that they can be used to perform operations on large tensors without incurring additional memory overhead.

6.Implement matmul using Einstein summation.

Answer: The following code implements matrix multiplication using Einstein summation in PyTorch:

import torch

A = torch.randn(2, 3)

B = torch.randn(3, 4)

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

print(C)

Here, we use the einsum function with the string argument 'ij,jk->ik' to specify the matrix multiplication operation between A and B. The resulting tensor C has shape (2, 4).

7.What does a repeated index letter represent on the lefthand side of einsum?

Answer: A repeated index letter on the lefthand side of einsum indicates a summation over that index. For example, if we have the string 'ij,jj->i' in einsum, it means that we want to perform a summation over the second dimension of the second tensor.

8.What are the three rules of Einstein summation notation? Why?

Answer: The three rules of Einstein summation notation are:

Repeated indices are implicitly summed over.

Each index can appear at most twice in an expression.

The order of the indices in the output must be the same as the order of the indices in the input.

These rules help to simplify and generalize mathematical operations involving tensors by eliminating the need for explicit summation signs and allowing for flexible indexing.

9.What are the forward pass and backward pass of a neural network?

Answer: The forward pass of a neural network refers to the process of taking input data, passing it through the layers of the network, and producing an output prediction. During the forward pass, each layer of the network applies a set of linear or nonlinear transformations to the input data to generate an intermediate output, which is then passed to the next layer. The final output of the network is produced by the last layer.

The backward pass, also known as backpropagation, refers to the process of computing the gradients of the loss function with respect to the parameters of the network. During the backward pass, the gradients are computed starting from the output layer and working backwards through the network using the chain rule of calculus. The gradients are then used to update the parameters of the network through an optimization algorithm such as stochastic gradient descent.

10.Why do we need to store some of the activations calculated for intermediate layers in the forward pass?

Answer: Storing activations for intermediate layers in the forward pass is necessary for computing the gradients during the backward pass. During backpropagation, the gradients are computed by multiplying the gradients from the next layer with the local gradient at the current layer. The local gradient is computed based on the activations at the current layer. Therefore, we need to store the activations during the forward pass to use them during backpropagation for computing the gradients.

11.What is the downside of having activations with a standard deviation too far away from 1?

Answer: When activations have a standard deviation that is too far away from 1, it can lead to two problems: vanishing gradients and exploding gradients. Vanishing gradients occur when the gradients become very small, making it difficult for the network to learn. Exploding gradients occur when the gradients become very large, causing the network to diverge during training. Both of these problems can make it difficult for the network to learn and converge to a good solution.

12.How can weight initialization help avoid this problem?

Answer: Proper weight initialization can help to avoid the problem of vanishing or exploding gradients. By initializing the weights of the network to appropriate values, we can ensure that the standard deviation of the activations remains close to 1. One common initialization technique is the Xavier initialization, which scales the weights based on the number of inputs and outputs to a layer. Another technique is the He initialization, which is similar to Xavier initialization but takes into account the nonlinearity of the activation function. These techniques can help to ensure that the gradients during training remain stable and the network can learn effectively.