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

unsqueeze helps us to solve certain broadcasting problems by adding a new dimension to a tensor. It increases the dimensionality of the tensor without changing the underlying data. This is useful when broadcasting requires matching dimensions, and unsqueeze is a way to explicitly add those dimensions to make the shapes compatible for elementwise operations.

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

import torch

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

expanded\_a = a[:, None]

print(expanded\_a)

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

import torch

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

array = x.numpy()

print(array)

1. 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.)

When adding a vector of size 3 to a matrix of size 3x3, the elements of the vector are added to each column of the matrix. This operation is performed through broadcasting, where the vector is effectively "stretched" along the rows to match the shape of the matrix, and elementwise addition is carried out.

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

Broadcasting and expand\_as do not result in increased memory use because they operate on existing tensors without creating new copies of data. They are memory-efficient operations that manipulate the dimensions and strides of tensors without consuming additional memory.

1. Implement matmul using Einstein summation.

import torch

# Define tensors

A = torch.tensor([[1, 2], [3, 4]])

B = torch.tensor([[5, 6], [7, 8]])

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

print(C)

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

A repeated index letter on the lefthand side of einsum represents a summation or reduction operation along that axis. It indicates that the specified dimensions should be contracted when performing the Einstein summation.

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

The three rules of Einstein summation notation are:

a. Repeated indices are summed over.

b. An index that appears only once on the left side is a free index on the right side.

c. The order of indices on the left side should match the order on the right side to specify which dimensions are being contracted.

These rules are used to define and perform tensor contractions and elementwise operations.

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

The forward pass of a neural network involves feeding input data through the network's layers to produce predictions or activations. These activations are calculated based on the network's weights and biases using activation functions.

The backward pass, also known as backpropagation, is the process of computing gradients of the loss with respect to the network's weights. These gradients are used in optimization algorithms like gradient descent to update the network's parameters during training.

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

Intermediate layer activations are stored during the forward pass to compute gradients during the backward pass. Gradients are calculated with respect to these intermediate activations, which are essential for backpropagation and weight updates.

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

The downside of having activations with a standard deviation too far away from 1 is that it can lead to issues during training. If activations have a high standard deviation (e.g., much greater than 1), gradients can become too small or too large, causing convergence problems and slow learning. It's important to have stable and well-scaled activations for efficient training.

1. How can weight initialization help avoid this problem?

Weight initialization helps avoid the problem of activations with a standard deviation that is too far from 1. Proper weight initialization methods set initial weights in a way that ensures activations are well-scaled and within a reasonable range. Techniques like Xavier/Glorot initialization or He initialization help mitigate the issues related to vanishing or exploding gradients during training by considering the number of input and output units in a layer.