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

A1. In PyTorch, unsqueeze() is a function that adds a new dimension to a tensor at the specified position. It can help us to solve certain broadcasting problems by adding new dimensions to a tensor that are not present in the original tensor, which enables PyTorch to broadcast the tensors together. For example, if we have two tensors with shapes (3,) and (2, 3), respectively, and we want to add them together, we can use unsqueeze() to add a new dimension to the first tensor so that it becomes (1, 3), and then use broadcasting to add the two tensors together.

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

A2. We can use indexing to add a new axis to a tensor, which is equivalent to unsqueezing the tensor. For example, if we have a 1D tensor **x** of shape **(3,)** and we want to add a new axis to make it a 2D tensor of shape **(3, 1)**, we can use indexing as follows:

x[:, None]

This creates a new axis at index 1, resulting in a tensor of shape **(3, 1)**. The **None** index is shorthand for **np.newaxis**, which creates a new axis with size 1. We can use this syntax to add new axes to tensors of any rank.

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

A3. We can use the **storage()** method in PyTorch to show the actual contents of the memory used for a tensor. This method returns the one-dimensional underlying storage of a tensor as a **torch.Storage** object, which can be used to inspect the values in memory.

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.)

A4. The elements of the vector are added to each row of the matrix. Here's a simple example in PyTorch to illustrate this:

import torch

m = torch.ones((3, 3))

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

print(m + v)

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

A5. Broadcasting and **expand\_as** do not result in increased memory use because they do not create new tensors. They only create views that represent the data in a different shape without copying the data itself. Therefore, the memory use is the same as before.

1. Implement matmul using Einstein summation.

A6. Here's an implementation of matmul using Einstein summation:

import torch

def matmul(a, b):

return torch.einsum('ik,kj->ij', a, b)

Here, **torch.einsum** is used to specify the matrix multiplication operation in a concise way. The first argument is a string specifying the indices to be summed over, and the second and third arguments are the two matrices to be multiplied. In this case, the string 'ik,kj->ij' indicates that we want to multiply the matrices **a** and **b**, where **a** has shape **(i,k)** and **b** has shape **(k,j)**, resulting in a matrix of shape **(i,j)**. The indices **i**, **j**, and **k** are used to specify the dimensions of the matrices.

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

A7. On the lefthand side of einsum, a repeated index letter represents a summation over that index.

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

A8. Einstein summation notation is a compact way of representing tensor operations using summation. The three rules of Einstein summation notation are:

1. Repeated indices are implicitly summed over.
2. An index that appears only once is not summed over, but is rather a free index.
3. An index can appear at most twice, once as a free index and once as a summed index.

These rules are used to simplify the expression of tensor operations. By using repeated indices, we can express summations more concisely and reduce the number of intermediate tensors that need to be created. The second rule helps us to distinguish between indices that are summed over and those that are not, while the third rule ensures that we do not have more than one sum over any given index.

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

A9. The forward pass and backward pass are two fundamental operations in training a neural network.

The forward pass refers to the process of propagating an input through the network to generate a prediction. It involves performing a series of matrix multiplications and activation functions to compute the output of each layer. At the end of the forward pass, we obtain a prediction that can be compared to the true label to compute the loss.

The backward pass, also known as backpropagation, refers to the process of computing the gradients of the loss with respect to the network parameters. This is done using the chain rule of calculus, by propagating the gradient of the loss backwards through the network. The gradients are then used to update the parameters using an optimizer, such as stochastic gradient descent (SGD), to improve the accuracy of the model.

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

A10. We need to store some of the activations calculated for intermediate layers in the forward pass because they are required during the backward pass for computing gradients using the chain rule of calculus. In particular, the gradient of the loss with respect to the weights of each layer depends on the activations of the previous layer. Therefore, we need to keep track of the intermediate activations during the forward pass in order to use them during the backward pass for computing gradients. This process is called backpropagation, and it is the main algorithm used to train deep neural networks.

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

A11. Activations with a standard deviation too far away from 1 can make training a neural network difficult. If the standard deviation is too large, then the gradients can be too small, leading to slow convergence and longer training times. Conversely, if the standard deviation is too small, then the gradients can be too large, causing numerical instability, exploding gradients, and making the optimization process fail. Therefore, having activations with a standard deviation close to 1 can help ensure that the gradients are neither too small nor too large, making it easier to train the neural network.

1. How can weight initialization help avoid this problem?

A12. Weight initialization can help avoid the problem of activations with a standard deviation too far away from 1 by setting the initial weights in a way that encourages the activations to have a desirable distribution. One popular method is Xavier initialization, which sets the initial weights to be drawn from a normal distribution with mean 0 and variance 1/n, where n is the number of inputs to the layer. This method ensures that the activations have a standard deviation of around 1, which can help avoid issues with vanishing or exploding gradients during training. Another popular method is He initialization, which is similar to Xavier initialization but uses a variance of 2/n instead. This is more suitable for activation functions such as ReLU, which can lead to a reduction in the signal magnitude otherwise.