**Assignment 12 Solution**

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

**Ans:** The unsqueeze function is used in PyTorch to add an extra dimension to a tensor. This can help us solve certain broadcasting problems where the tensor shapes do not match.

Broadcasting is a technique in PyTorch that allows us to perform element-wise operations on tensors with different shapes. However, to do this, PyTorch requires that the shapes of the tensors are compatible. Two tensors are compatible for broadcasting if their shapes are equal, or if one of the tensors has a dimension of size 1.

If we have two tensors that are not compatible for broadcasting, we can use unsqueeze to add an extra dimension to one of the tensors, effectively making it compatible with the other tensor. For example, consider the following tensors:

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

B = torch.tensor([[4, 5, 6],

[7, 8, 9]])

If we want to add A to every row of B, we need to reshape A to be a 2D tensor of shape (3, 1), so that it has the same number of columns as B. We can do this using the unsqueeze function:

A = A.unsqueeze(1)

C = A + B

Now, A has shape (3, 1) and B has shape (2, 3), which are compatible for broadcasting. PyTorch automatically broadcasts A to have the same shape as B by copying its elements along the new dimension.

In general, unsqueeze is a useful function when we need to add an extra dimension to a tensor to make it compatible with another tensor for broadcasting.

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

**Ans:** We can use indexing to achieve the same operation as unsqueeze by creating a new axis or dimension in the tensor. In PyTorch, we can use the None keyword to create a new axis.

For example, consider the following tensor:

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

To add a new axis to A using indexing, we can write:

A[:, None]

This creates a new axis at position 1, effectively reshaping A from a 1D tensor of shape (3,) to a 2D tensor of shape (3, 1). We can then use this reshaped tensor for broadcasting with another tensor, just as we did with unsqueeze.

Similarly, to add a new axis to a 2D tensor like B from the previous example:

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

We can use indexing as follows:

B[:, :, None]

This creates a new axis at position 2, effectively reshaping B from a 2D tensor of shape (2, 3) to a 3D tensor of shape (2, 3, 1). Again, we can use this reshaped tensor for broadcasting with another tensor.

In general, using indexing to add new axes or dimensions can be a useful alternative to unsqueeze for reshaping tensors and making them compatible for broadcasting.

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

**Ans:** In PyTorch, we can use the storage and tolist methods to show the actual contents of the memory used for a tensor.

The storage method returns a one-dimensional tensor that contains the underlying storage of the tensor. This means that it returns a flattened version of the tensor's data, regardless of its shape. For example:

import torch

A = torch.tensor([[1, 2, 3], [4, 5, 6]])

print(A.storage())

This will output a one-dimensional tensor with the following values:

tensor([1, 2, 3, 4, 5, 6])

Note that the values are stored in row-major order, meaning that the first row of the tensor comes before the second row in the flattened storage.

To convert this flattened storage back to its original shape, we can use the tolist method to convert the storage to a Python list and then reshape it using the view method. For example:

A\_list = A.storage().tolist() A\_shape = A.shape

A\_reconstructed = torch.tensor(A\_list).view(A\_shape)

print(A\_reconstructed)

This will output the original tensor A:

tensor([[1, 2, 3], [4, 5, 6]])

In general, showing the actual contents of the memory used for a tensor can be useful for debugging and understanding how the data is stored and manipulated in memory. However, it is important to note that the flattened storage may not always correspond to the tensor's shape and layout, especially for tensors with complex strides or non-contiguous memory.

**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.)**

**Ans:** When adding a vector of size 3 to a matrix of size 3x3, the elements of the vector are added to each row of the matrix. This is because broadcasting in PyTorch operates along the last dimensions of the tensors.

For example, consider the following code:

import torch

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

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

C = A + B

print(C)

This will output the following tensor:

tensor([[2., 3., 4.],

`[2., 3., 4.],`

`[2., 3., 4.]])`

As we can see, the values of B have been added to each row of A. This is because the last dimension of B (size 3) is broadcasted to match the last dimension of A (also size 3), and the addition operation is applied element-wise along this dimension.

If we want to add the vector to each column of the matrix instead, we need to reshape the vector to be a column vector of size (3, 1) so that it matches the shape of the matrix along the last dimension. We can do this using the unsqueeze method or indexing.

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

**Ans:** Broadcasting and expand\_as operations do not necessarily result in increased memory use, because they do not create new copies of the data. Instead, they rely on the underlying storage of the tensors and use different views or shapes to perform the desired operations.

Broadcasting works by implicitly expanding the shapes of the tensors to match each other along their non-singleton dimensions. This means that the broadcasted tensor does not actually contain new data, but rather a view of the same data with a different shape. Similarly, expand\_as creates a new tensor with the same underlying storage as the input tensor, but with additional dimensions of size 1 that are expanded to match the desired shape. This also means that no new data is created, but rather a different view or shape of the same data.

Therefore, broadcasting and expand\_as can be very memory-efficient and can save memory compared to explicitly copying the data. However, they may require additional memory for the metadata and bookkeeping necessary to keep track of the different views and shapes of the tensors.

It is worth noting that broadcasting and expand\_as may have performance implications, especially when dealing with large tensors and complex operations. In some cases, it may be more efficient to use explicit copying or to reshape the tensors manually to avoid the overhead of broadcasting or expanding. Additionally, broadcasting and expand\_as may result in non-contiguous memory layouts, which can affect performance and memory access patterns.

**6. Implement matmul using Einstein summation.**

**Ans:** here's an implementation of matrix multiplication (matmul) using Einstein summation notation in Python:

import numpy as np

def matmul(A, B):

return np.einsum('ij,jk->ik', A, B)

Here, we use the np.einsum function, which takes two arguments: the Einstein summation notation string and the matrices to multiply. In the notation string, ij represents the rows and columns of matrix A, and jk represents the rows and columns of matrix B. The ->ik part specifies the dimensions of the resulting matrix.

To use this function, you can simply pass in two matrices A and B:

A = np.array([[1, 2], [3, 4]])

B = np.array([[5, 6], [7, 8]])

C = matmul(A, B)

print(C)

This will output the result of the matrix multiplication:

array([[19, 22],

[43, 50]])|

Note that this implementation assumes that the number of columns of A is equal to the number of rows of B, which is necessary for matrix multiplication.

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

**Ans:** In Einstein summation notation, a repeated index letter on the lefthand side of the einsum function indicates summation over that index.

For example, consider the following notation:

np.einsum('ii', A)

Here, the index letter i is repeated, which means that we want to sum over all the entries in A where the row and column indices are the same. This is equivalent to taking the trace of the matrix A.

In general, if an index letter appears more than once on the lefthand side of einsum, then it represents summation over that index. The resulting output will have all instances of that index letter eliminated, leaving only the indices that were not repeated.

For example, consider the following notation:

np.einsum('ijk, jkl -> il', A, B)

Here, the index letter j appears twice, once in the ijk subscript and once in the jkl subscript. This means that we want to sum over all possible values of j. The resulting output will have indices i and l left over, since they were not repeated.

Overall, repeated indices are a key feature of Einstein summation notation and allow for concise representation of matrix and tensor operations.

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

**Ans:** The three rules of Einstein summation notation are as follows:

* Repeated indices are summed over: Whenever an index appears twice in a single term, it is assumed to be summed over. For example, in the expression A\_ij B\_jk, the index j is repeated and therefore summed over.
* Free indices are not summed over: An index that appears only once in a term is called a free index, and is not summed over. For example, in the expression A\_ij x\_j, the index i is a free index and is not summed over.
* The indices on the left-hand side of the einsum notation determine the indices in the output: The indices that appear in the left-hand side of the einsum notation represent the indices that will appear in the output. For example, in the expression np.einsum('ij,jk->ik', A, B), the resulting output will have indices i and k, since those are the indices that appear on the left-hand side of the einsum notation.

These rules are necessary because Einstein summation notation allows for compact representation of matrix and tensor operations. The notation allows us to express complex operations in a concise way, without having to write out every single term explicitly. The rules ensure that the notation is unambiguous and consistent, and that the resulting output has the expected dimensions and indices.

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

**Ans:** The forward pass and backward pass are two essential steps in the training of a neural network. They are also known as the feedforward step and backpropagation step, respectively.

Forward pass: In the forward pass, the input data is passed through the neural network to produce an output. Each neuron in the network receives input from the previous layer, applies an activation function to it, and outputs the result to the next layer. This process is repeated for each layer until the output layer produces the final result. During the forward pass, the weights and biases of the network are fixed and not updated.

Backward pass: In the backward pass, the error in the output is calculated and propagated back through the network to adjust the weights and biases. This process is called backpropagation. The goal of backpropagation is to update the weights and biases in a way that reduces the error in the output. This is done by computing the gradient of the error with respect to each weight and bias in the network. The gradient is then used to update the weights and biases using an optimization algorithm such as gradient descent.

The forward pass and backward pass are repeated multiple times during the training process, with each iteration updating the weights and biases of the network to improve its performance. The goal is to minimize the error between the network's predicted output and the true output.

Overall, the forward pass and backward pass are essential components of training a neural network. The forward pass computes the output of the network given the input, and the backward pass updates the network's weights and biases to improve its performance.

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

**Ans:** Storing the activations calculated for intermediate layers during the forward pass is essential for several reasons:

* Backpropagation: During the backward pass, the gradients of the loss with respect to the weights in each layer are computed by propagating the error backwards through the network. The gradients are computed using the activations and the derivatives of the activation functions in each layer. Therefore, the activations calculated during the forward pass are needed to perform the backward pass and update the weights of the network.
* Reuse of activations: In some cases, the activations calculated during the forward pass can be reused for multiple purposes. For example, in certain architectures such as residual networks, the activations from earlier layers are added to the activations from later layers. Storing the activations from earlier layers during the forward pass allows them to be reused later in the network.
* Debugging: Storing the activations from intermediate layers during the forward pass can be useful for debugging the network. By examining the values of the activations, we can gain insights into how the network is processing the input and identify potential problems.
* Visualization: Storing the activations from intermediate layers during the forward pass can also be useful for visualizing the features learned by the network. By examining the activations from different layers, we can gain insights into the types of patterns and features that the network is learning.

Overall, storing the activations calculated for intermediate layers during the forward pass is necessary for performing the backward pass, reusing activations, debugging, and visualization.

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

**Ans:** Having activations with a standard deviation that is too far away from 1 can lead to several downsides in deep learning models.

One of the main downsides is that it can cause the gradients to vanish or explode during the backpropagation process. When the standard deviation of the activations is too small, the gradients can become extremely small, making it difficult for the model to learn effectively. Conversely, when the standard deviation is too large, the gradients can become extremely large, causing the weights to update too drastically and potentially leading to unstable training.

Another downside is that it can lead to slow or unstable convergence during training. If the standard deviation of the activations is too small, the model may not be able to learn quickly enough to reach the desired performance level. On the other hand, if the standard deviation is too large, the model may oscillate or diverge during training, making it difficult to obtain good results.

Overall, having activations with a standard deviation that is too far away from 1 can make it more difficult for deep learning models to learn effectively and converge to a good solution. Therefore, it is important to carefully initialize the weights and monitor the standard deviation of the activations during training to ensure that it stays within a reasonable range.

**12. How can weight initialization help avoid this problem?**

**Ans:** Weight initialization can help avoid the problem of having activations with a standard deviation that is too far away from 1 by ensuring that the initial weights of the neural network are set appropriately.

One common initialization technique is the Xavier initialization, which initializes the weights randomly according to a Gaussian distribution with mean 0 and variance 2/(n\_inputs + n\_outputs), where n\_inputs is the number of input neurons and n\_outputs is the number of output neurons. This initialization method ensures that the standard deviation of the activations is roughly equal to 1, which can help avoid the problem of vanishing or exploding gradients during training.

Another initialization technique is the He initialization, which is similar to the Xavier initialization but uses a variance of 2/n\_inputs instead. This initialization method is more appropriate for activation functions that have a non-zero mean, such as the ReLU activation function.

By using appropriate weight initialization techniques, the standard deviation of the activations can be kept within a reasonable range, which can help improve the stability and convergence of the neural network during training. However, it is important to note that weight initialization is just one aspect of building a well-performing neural network, and other factors such as the choice of optimization algorithm and learning rate can also have a significant impact on the performance of the model.