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

Answer:--->Returns a new tensor with a dimension of size one inserted at the specified position. The returned tensor shares the same underlying data with this tensor.To squeeze a tensor we can apply the torch. squeeze() method and to unsqueeze a tensor we use the torch. unsqueeze() method.unsqueeze is a PyTorch function that is used to add a dimension to a tensor. The unsqueeze function is often used to resolve broadcasting problems by adding an extra dimension to a tensor so that it can be broadcasted to match the shape of another tensor.

Broadcasting is a feature in PyTorch that allows you to perform elementwise operations between tensors of different shapes. However, in order for two tensors to be broadcasted together, their shapes must be compatible, which means that one of the tensors must be broadcastable to the shape of the other tensor.

Sometimes, the shapes of two tensors are not compatible, even though you want to perform an elementwise operation between them. In these cases, you can use unsqueeze to add an extra dimension to one of the tensors so that the two tensors can be broadcasted together.

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

Answer:---->Returns a new tensor with a dimension of size one inserted at the specified position. The returned tensor shares the same underlying data with this tensor. According to documentation, unsqueeze() inserts singleton dim at position given as parameter and view() creates a view with different dimensions of the storage associated with tensor. In the unsqueeze() function, we can insert the specific dimension at a specific location. The view() can only use a single argument. In unsqueeze, we can use more than one argument during the operation.

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

Answer:---->The commonly used way to store such data is in a single array that is laid out as a single, contiguous block within memory. More concretely, a 3x3x3 tensor would be stored simply as a single array of 27 values, one after the other.We can access the value of a tensor by using indexing and slicing. Indexing is used to access a single value in the tensor. slicing is used to access the sequence of values in a tensor. we can modify a tensor by using the assignment operator.The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string. We use the index operator ([] – not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0.You can access an array element using an expression which contains the name of the array followed by the index of the required element in square brackets. To print it simply pass this method to the println() method.We can access the value of a tensor by using indexing and slicing. Indexing is used to access a single value in the tensor. slicing is used to access the sequence of values in a tensor. we can modify a tensor by using the assignment.The easiest[A] way to evaluate the actual value of a Tensor object is to pass it to the Session. run() method, or call Tensor. eval() when you have a default session (i.e. in a with tf. Session(): block, or see below).

5. Do broadcasting and expand_as result in increased memory use? Why or why not?

Answer:----->In general, the use of broadcasting or expand_as does not increase memory usage. These operations do not create copies of the tensors, but instead create new views of the original tensors that can be used in operations. These new views still refer to the same underlying data as the original tensors and do not require additional memory.

However, in some cases, when broadcasting is used to expand tensors to a larger size, temporary arrays may be created to hold the intermediate results of the computation. This can result in increased memory usage during the computation, but these arrays are typically released after the computation is complete, so the overall memory usage does not increase.

Similarly, the use of expand_as can also result in temporary arrays being created, but these are also typically released after the computation is complete, so the overall memory usage does not increase.

It is worth noting that the exact behavior of broadcasting and expand_as can depend on the specific deep learning library and the implementation details, so it's always a good idea to check the documentation of the library you are using to get a better understanding of the memory usage implications of these operations.

6. Implement matmul using Einstein summation.

Answer:---->The matrix multiplication operation (matmul) can be implemented using Einstein summation. Einstein summation is a compact way of expressing operations that involve summing over multiple indices of arrays. The basic idea is to use Einstein's summation convention, which is to assume that repeated indices are summed over. Here's an implementation of matrix multiplication using Einstein summation:

import numpy as np

def matmul_einstein(A, B):

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

Here, the einsum function is used to perform the matrix multiplication. The first argument to einsum is a string that describes the input arrays and the operation to be performed. In this case, 'ij,jk->ik' describes two input arrays A and B, where i and j are indices of A, and j and k are indices of B. The arrow -> indicates the output, which is an array with indices i and k.

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

Answer:---->In the einsum function of Numpy, a repeated index letter on the lefthand side of the equation represents a sum over that index. For example, if you have an einsum expression like ij,jk->ik, the j in ij and the j in jk are both repeated, which means that the expression performs a sum over the index j. This can be thought of as a loop over the j index, where the value of j is used to index into both arrays A and B for each iteration of the loop.

In general, when you use einsum, you are expressing a multi-dimensional array operation in a compact form, where the repeated indices indicate a sum over that index. This is a powerful feature of einsum that allows you to express complex array operations in a concise and readable form. The repeated indices are particularly useful when you are working with arrays with a large number of dimensions, as they allow you to perform operations on arrays with many dimensions without having to write explicit loops over each dimension.

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

Answer:___>The “rules” of summation convention are: Each index can appear at most twice in any term. Repeated indices are implicitly summed over. Each term must contain identical non-repeated indices.Summation runs over 1 to 3 since we are 3 dimension •No indices appear more than two times in the equation •Indices which is summed over is called dummy indices appear only in one side of equation •Indices which appear on both sides of the equation is free indices.There are essentially three rules of Einstein summation notation, namely: Repeated indices are implicitly summed over. Each index can appear at most twice in any term. Each term must contain identical non-repeated indices. The Einstein Field Equations are ten equations, contained in the tensor equation shown above, which describe gravity as a result of spacetime being curved by mass and energy.

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

Answer:---->Forward Propagation is the way to move from the Input layer (left) to the Output layer (right) in the neural network. The process of moving from the right to left i.e backward from the Output to the Input layer is called the Backward Propagation.The "forward pass" refers to calculation process, values of the output layers from the inputs data. It's traversing through all neurons from first to last layer. A loss function is calculated from the output values.In the backward pass, the flow is reversed so that we start by propagating the error to the output layer until reaching the input layer passing through the hidden layer(s). The process of propagating the network error from the output layer to the input layer is called backward propagation, or simple backpropagation.Forward Propagation is the way to move from the Input layer (left) to the Output layer (right) in the neural network. The process of moving from the right to left i.e backward from the Output to the Input layer is called the Backward Propagation.Forward pass is a technique to move forward through network diagram to determining project duration and finding the critical path or Free Float of the project. Whereas backward pass represents moving backward to the end result to calculate late start or to find if there is any slack in the activity.

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

Answer:---->In broad terms, activation functions are necessary to prevent linearity. Without them, the data would pass through the nodes and layers of the network only going through linear functions (a*x+b).An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations.Properties of activation functions Non Linearity.

Continuously differentiable.

Range.

Monotonic.

Approximates identity near the origin. The activation function compares the input value to a threshold value. If the input value is greater than the threshold value, the neuron is activated. It's disabled if the input value is less than the threshold value, which means its output isn't sent on to the next or hidden layer.An activation function is the last component of the convolutional layer to increase the non-linearity in the output. Generally, ReLu function or Tanh function is used as an activation function in a convolution layer.Clearly you can use different activations in a neural network. An MLP with any activation and a softmax readout layer is one example (for example, multi-class classification). An RNN with LSTM units has at least two activation functions (logistic, tanh and any activations used elsewhere).

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

Answer:--->If the standard deviation of the activations in a neural network is too far from 1, it can lead to a few problems:

1.Vanishing Gradients: If the standard deviation of the activations is too small (i.e., far from 1), then the gradients of the model's parameters with respect to the loss may become very small during backpropagation. This leads to slow convergence or even non-convergence of the optimization algorithm. This problem is known as vanishing gradients.

2.Exploding Gradients: If the standard deviation of the activations is too large (i.e., far from 1), then the gradients of the model's parameters with respect to the loss may become very large during backpropagation. This can cause the optimizer to overshoot the optimal parameters and may result in instability in training or even non-convergence. This problem is known as exploding gradients.

3.Reduced Generalization: If the activations in a network have a standard deviation that is far from 1, this can result in the network's learned representations becoming overly specialized to the training data, reducing the network's ability to generalize to new, unseen data.

In general, it is best to keep the standard deviation of the activations close to 1 as this helps ensure that the network's optimization problem is well-conditioned and that the network is able to learn useful representations that generalize well to new data.

12. How can weight initialization help avoid this problem?

Answer:---->Weight initialization is a crucial factor in training deep neural networks. The choice of weight initialization can have a significant impact on the network's ability to converge to a good solution during training. By choosing appropriate initial weights, we can help avoid the problems associated with activations with a standard deviation that is too far from 1.

Here are a few common weight initialization methods that can help avoid this problem:

1. Glorot (Xavier) Initialization: This initialization method scales the weights based on the number of inputs and outputs to each neuron. The goal is to keep the variance of the activations close to 1, which helps ensure that the gradients are not too small or too large during backpropagation.
2. He Initialization: This initialization method is similar to Glorot initialization, but is specifically designed for rectified linear unit (ReLU) activation functions. It uses a scaling factor that is larger than in Glorot initialization to account for the fact that ReLU activations are 0 for half of the input space.
3. LeCun Initialization: This initialization method is also similar to Glorot initialization, but is specifically designed for sigmoid and hyperbolic tangent activation functions. It uses a scaling factor that is smaller than in Glorot initialization to account for the fact that sigmoid and hyperbolic tangent activations are squashed into a narrow range.

Each of these weight initialization methods has its own strengths and weaknesses, and the best choice of initialization method depends on the specifics of the problem at hand. However, by using one of these initialization methods, you can help avoid the problems associated with activations with a standard deviation that is too far from 1 and improve the training stability and performance of your network.1