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

**unsqueeze turns an n.d. tensor into an (n+1).d. one by adding an extra dimension of depth 1. However, since it is ambiguous which axis the new dimension should lie across (i.e. in which direction it should be "unsqueezed"), this needs to be specified by the**[**dim**](https://pytorch.org/docs/stable/generated/torch.unsqueeze.html)**argument.**

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

**Unsqueeze a Tensor:**

When we unsqueeze a tensor, a new dimension of size 1 is inserted at the specified position.  Always an unsqueeze operation increases the dimension of the output tensor. For example, if the input tensor is of shape:  (m×n) and we want to insert a new dimension at position 1 then the output tensor after unsqueeze will be of shape: (m×1×n). The following is the syntax of the *torch.unsqueeze()* method-

***Syntax:****torch.unsqueeze(input, dim)*

***Parameters:***

* ***input:****the input tensor.*
* ***dim:****an integer value, the index at which the singleton dimension is inserted.*

***Return:****It returns a new tensor with a dimension of size one inserted at the specified position****dim****.*

Please note that we can choose the *dim* value from the range [-input.dim() – 1, input.dim() + 1). The negative dim will correspond to dim = dim + input.dim() + 1.

**# Python program to unsqueeze the input tensor**

**# importing torch**

**import torch**

**# define the input tensor**

**input = torch.arange(8, dtype=torch.float)**

**print("Input tensor:\n", input)**

**print("Size of input Tensor before unsqueeze:\n",**

**input.size())**

**output = torch.unsqueeze(input, dim=0)**

**print("Tensor after unsqueeze with dim=0:\n", output)**

**print("Size after unsqueeze with dim=0:\n",**

**output.size())**

**output = torch.unsqueeze(input, dim=1)**

**print("Tensor after unsqueeze with dim=1:\n", output)**

**print("Size after unsqueeze with dim=1:\n",**

**output.size())**

**Output:**

**Input tensor:**

**tensor([0., 1., 2., 3., 4., 5., 6., 7.])**

**Size of input Tensor before unsqueeze:**

**torch.Size([8])**

**Tensor after unsqueeze with dim=0:**

**tensor([[0., 1., 2., 3., 4., 5., 6., 7.]])**

**Size after unsqueeze with dim=0:**

**torch.Size([1, 8])**

**Tensor after unsqueeze with dim=1:**

**tensor([[0.],**

**[1.],**

**[2.],**

**[3.],**

**[4.],**

**[5.],**

**[6.],**

**[7.]])**

**Size after unsqueeze with dim=1:**

**torch.Size([8, 1])**

**# Python program to squeeze the tensor in**

**# different dimensions**

**# importing torch**

**import torch**

**# creating the input tensor**

**input = torch.randn(3,1,2,1,4)**

**print("Dimension of input tensor:", input.dim())**

**print("Input tensor Size:\n",input.size())**

**# squeeze the tensor in dimension 0**

**output = torch.squeeze(input,dim=0)**

**print("Size after squeeze with dim=0:\n",**

**output.size())**

**# squeeze the tensor in dimension 0**

**output = torch.squeeze(input,dim=1)**

**print("Size after squeeze with dim=1:\n",**

**output.size())**

**# squeeze the tensor in dimension 0**

**output = torch.squeeze(input,dim=2)**

**print("Size after squeeze with dim=2:\n",**

**output.size())**

**# squeeze the tensor in dimension 0**

**output = torch.squeeze(input,dim=3)**

**print("Size after squeeze with dim=3:\n",**

**output.size())**

**# squeeze the tensor in dimension 0**

**output = torch.squeeze(input,dim=4)**

**print("Size after squeeze with dim=4:\n",**

**output.size())**

**# output = torch.squeeze(input,dim=5) # Error**

**Output:**

Dimension of input tensor: 5

Input tensor Size:

torch.Size([3, 1, 2, 1, 4])

Size after squeeze with dim=0:

torch.Size([3, 1, 2, 1, 4])

Size after squeeze with dim=1:

torch.Size([3, 2, 1, 4])

Size after squeeze with dim=2:

torch.Size([3, 1, 2, 1, 4])

Size after squeeze with dim=3:

torch.Size([3, 1, 2, 4])

Size after squeeze with dim=4:

torch.Size([3, 1, 2, 1, 4])

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

**sys.getsizeof() will return the size of the python object. It will the same for all tensors as all tensors are a python object containing a tensor. For each tensor, you have a method element\_size() that will give you the size of one element in byte. And a function nelement() that returns the number of elements. So the size of a tensor a in memory (cpu memory for a cpu tensor and gpu memory for a gpu tensor) is a.element\_size() \* a.nelement().**

**Therefore, the (simple) answer appears to be as follows, and holds true when calculating expected memory usage and release in various scenarios:**

**a.element\_size() \* a.nelement()**

**This is supported by experimenting with the following:**

**torch.tensor(1, dtype=torch.int8).element\_size() # 1 (byte)**

**torch.tensor(1, dtype=torch.int16).element\_size() # 2 (bytes)**

**torch.tensor(1, dtype=torch.int32).element\_size() # 4 (bytes)**

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.)**

I start with A a N by 3 matrix generated from data, which I use to create another N by 3 matrix B , with which using a cross product I create the last N by 3 Matrix C.

Each of these matrixes rows ( or columns with permutation) are the column of my last matrix Result.

I want to take in a vectorised operation, the first row ( or column in permutated) of each of these matrixes to create my first 3 by 3 matrix, do the same for the secon, third and so on until the nth element.

which I will then transform in a 3 by 3 by N matrix with reshape.

However I'm having a lot of trouble create the Result matrix because anything that I tried ended up giving me the concatenation of all A by all B by all, and I do not know of the existence of a column operator that restarts the count if it arrives to the end( otherwise I could re- arrange the matrix using the number [A(1:3N:N)),B(N+1:3N,N)C(2N+1:3N:N)].

So in matrix representation I'm looking for this:

A=[1 ,10, 19;2, 11, 20 ;3, 12, 21]

A =

1 10 19

2 11 20

3 12 21

>> B=[4 13, 22;5,14, 23; 6, 15 ,24]

B =

4 13 22

5 14 23

6 15 24

>> C=[7 16, 25; 8 17,26; 9 18, 27]

C =

7 16 25

8 17 26

9 18 27

>> D=[1 ,4,7,10,13,16,19,22,25;2,5,8,11,14,17,20,23,26;3,6,9,12,15,18,21,24,27]

D =

1 4 7 10 13 16 19 22 25

2 5 8 11 14 17 20 23 26

3 6 9 12 15 18 21 24 27

%and then Result=Reshape(D,3,3,[])

Result=reshape(D,3,3,[])

Result(:,:,1) =

1 4 7

2 5 8

3 6 9

Result(:,:,2) =

10 13 16

11 14 17

12 15 18

Result(:,:,3) =

19 22 25

20 23 26

21 24 27

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

**Broadcasting and expand\_as are techniques used in tensor operations, particularly in libraries like PyTorch and NumPy, to make tensors compatible for element-wise operations without actually copying data, thus minimizing memory usage.**

**Broadcasting:**

* **Definition: Broadcasting automatically expands the dimensions of smaller tensors to match the dimensions of larger tensors during arithmetic operations.**
* **Memory Use: Does not increase memory use significantly because it does not create new copies of the data. Instead, it uses strides to simulate the expanded dimensions.**
* **Criteria:**
  + **Tensors must be compatible in shape.**
  + **Dimensions are compared from the trailing dimension (rightmost) to the leading dimension (leftmost).**
  + **A dimension can be expanded if it is 1 or if it matches the corresponding dimension of the other tensor.**

**expand\_as:**

* **Definition: expand\_as explicitly expands a tensor to the shape of another tensor.**
* **Memory Use: Similar to broadcasting, it does not increase memory use significantly because it does not create new data but rather changes the view of the tensor.**
* **Criteria:**
  + **The tensor to be expanded must have dimensions that are either 1 or match the corresponding dimensions of the target tensor.**
  + **The expanded tensor shares the same data with the original tensor.**

**Example:**

**Python**

**import torch**

**# Broadcasting example**

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

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

**result = a + b # Broadcasting happens here**

**# expand\_as example**

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

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

**expanded\_c = c.expand\_as(d)**

**In summary, both broadcasting and expand\_as do not significantly increase memory use because they avoid creating new copies of the data. Instead, they manipulate the tensor's shape and strides to perform operations efficiently.**

1. **Implement matmul using Einstein summation.**

**Explanation:**

To calculate the net flux of the vectors a and b at the point (1,2,3), evaluate the dot product of the vectors with the outward unit normal vector at that point and sum them up.

Given the vectors:

a=x2yex−3yey+2z2ez and

b=xy2ex−13y3ey+xyez

Let's denote the unit outward normal vector as n^.

At the point (1,2,3), the unit vector is the normalized gradient vector of the function:

f(x,y,z)=x2y−3y+2z2

The gradient of f(x,y,z) is :

△f=(𝜕f𝜕x,𝜕f𝜕y,𝜕f𝜕z)

So,

𝜕f𝜕x=(x2y−3y+2z2)𝜕x=2xy  [ Considering y, z as constant ]

𝜕f𝜕y=(x2y−3y+2z2)𝜕y=x2−3  [ Considering x, z as constant ]

Hence,

△f=(2xy,x2−3,4z)

Now, at point (1,2,3) the gradient becomes :

△f(1,2,3)=(4,−2,12)

**Explanation:**

Normalize the above vector to calculate the unit vector.

n^=△f|△f|

=(4,−2,12)42+(−2)2+122

=(4,−2,12)42+(−2)2+122

=(4,−2,12)164=(4164,−2164,12164)

=(4241,−2241,12241)=(241,−141,641)

Hence, the required unit vector is

Now,

The two vectors are:

a=x2y−3y+2z2

b=xy2−13y3+xy

Calculate the dot product of a and b with n^ at point (1,2,3) and sum them up to find the net flux.

For vector a:

a.n^=(x2y,−3y,2z2).(241,−141,641)

=(12.2,−3.2,2.32).(241,−141,641)

=(2,−6,18)(241,−141,641)

=4/41+6/41+108/41=11841

For vector b

b.n^=(xy2,−13y3,xy).(241,−141,641)=(1.22,−1323,1.2).(241,−141,641)=(4,−83,2).(241,−141,641)=841+8341+1241=24+8+36341=68341

Hence, the net flux is the sum of dot products :

The required net flux of the vectors a and b at points (x,y,z)=(1,2,3) is

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

**We use -> to indicate the order of the output array. So think of 'ij, i->j' as having left hand side (LHS) and right hand side (RHS). Any repetition of labels on the LHS computes the product element wise and then sums over. By changing the label on the RHS (output) side, we can define the axis in which we want to proceed with respect to the input array, i.e. summation along axis 0, 1 and so on.**

**import numpy as np**

**>>> a**

**array([[1, 1, 1],**

**[2, 2, 2],**

**[3, 3, 3]])**

**>>> b**

**array([[0, 1, 2],**

**[3, 4, 5],**

**[6, 7, 8]])**

**>>> d = np.einsum('ij, jk->ki', a, b)**

**Notice there are three axes, i, j, k, and that j is repeated (on the left-hand-side). i,j represent rows and columns for a. j,k for b.**

**In order to calculate the product and align the j axis we need to add an axis to a. (b will be broadcast along(?) the first axis)**

**a[i, j, k]**

**b[j, k]**

**>>> c = a[:,:,np.newaxis] \* b**

**>>> c**

**array([[[ 0, 1, 2],**

**[ 3, 4, 5],**

**[ 6, 7, 8]],**

**[[ 0, 2, 4],**

**[ 6, 8, 10],**

**[12, 14, 16]],**

**[[ 0, 3, 6],**

**[ 9, 12, 15],**

**[18, 21, 24]]])**

**j is absent from the right-hand-side so we sum over j which is the second axis of the 3x3x3 array**

**>>> c = c.sum(1)**

**>>> c**

**array([[ 9, 12, 15],**

**[18, 24, 30],**

**[27, 36, 45]])**

**Finally, the indices are (alphabetically) reversed on the right-hand-side so we transpose.**

**>>> c.T**

**array([[ 9, 18, 27],**

**[12, 24, 36],**

**[15, 30, 45]])**

**>>> np.einsum('ij, jk->ki', a, b)**

**array([[ 9, 18, 27],**

**[12, 24, 36],**

**[15, 30, 45]])**

**>>>**

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

**The three rules of Einstein summation notation are:**

1. **Implicit Summation Over Repeated Indices: When an index variable appears twice in a single term, it implies summation over all possible values of that index. This eliminates the need for explicit summation symbols.**
   * **Example: aibi implies ∑iaibi.**
2. **Free and Dummy Indices: Indices that appear only once in a term are called free indices, and they must appear in each term of the equation. Indices that appear twice are called dummy indices and are summed over.**
   * **Example: In the expression ci=aijbj, i is a free index and j is a dummy index.**
3. **Index Consistency: The same index should not appear more than twice in a single term. This ensures clarity and avoids ambiguity in the summation process.**
   * **Example: aii is not allowed, but aijbij is allowed.**

**Correct answer: The three rules of Einstein summation notation are implicit summation over repeated indices, distinction between free and dummy indices, and index consistency.**

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

**The forward pass and backward pass are two fundamental processes in training a neural network.**

**Forward Pass:**

* **Definition: The forward pass involves passing input data through the network to obtain the output predictions.**
* **Required Attributes:**
  + **Input data: The initial data fed into the network.**
  + **Weights and biases: Parameters of the network layers.**
  + **Activation functions: Functions applied to the outputs of each layer.**
* **Variable Attributes:**
  + **Intermediate layer outputs: Outputs from each hidden layer.**
  + **Final output: The prediction made by the network.**

**Backward Pass:**

* **Definition: The backward pass involves propagating the error back through the network to update the weights and biases.**
* **Required Attributes:**
  + **Loss function: A function that measures the difference between the predicted output and the actual target.**
  + **Learning rate: A hyperparameter that controls the step size of the weight updates.**
* **Variable Attributes:**
  + **Gradients: Partial derivatives of the loss function with respect to the weights and biases.**
  + **Updated weights and biases: Adjusted parameters after applying the gradients.**

**Forward Pass Steps:**

1. **Input data is fed into the network.**
2. **Each layer processes the input using its weights, biases, and activation function.**
3. **The final layer produces the output prediction.**

**Backward Pass Steps:**

1. **Compute the loss using the loss function.**
2. **Calculate the gradient of the loss with respect to each weight and bias using backpropagation.**
3. **Update the weights and biases using the gradients and the learning rate.**

**Example of Forward Pass Calculation: Given an input x, weight w, bias b, and activation function f:**

**z=w×x+ba=f(z)**

**Example of Backward Pass Calculation: Given a loss L, weight w, and learning rate 𝛼:**

**dw=dLdww=w−𝛼×dw**

**Python Code Example for Forward and Backward Pass:**

**Python**

**import numpy as np**

**# Forward pass**

**def forward\_pass(x, w, b):**

**z = np.dot(w, x) + b**

**a = np.maximum(0, z) # ReLU activation**

**return a**

**# Backward pass**

**def backward\_pass(x, w, b, a, y, learning\_rate):**

**dz = a - y**

**dw = np.dot(dz, x.T)**

**db = np.sum(dz, axis=1, keepdims=True)**

**w -= learning\_rate \* dw**

**b -= learning\_rate \* db**

**return w, b**

**# Example usage**

**x = np.array([[1], [2]])**

**w = np.array([[0.5, -0.5]])**

**b = np.array([[0]])**

**y = np.array([[1]])**

**learning\_rate = 0.01**

**a = forward\_pass(x, w, b)**

**w, b = backward\_pass(x, w, b, a, y, learning\_rate)**

**This code demonstrates a simple forward and backward pass for a single-layer neural network using ReLU activation and gradient descent for weight updates.**

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

**We need to store some of the activations calculated for intermediate layers in the forward pass primarily for the backpropagation process during training. Backpropagation requires these intermediate activations to compute gradients efficiently. Here are the key points:**

* **Clear Definition: Activations are the outputs of neurons in a neural network layer after applying the activation function.**
* **Required Attributes:**
  + **Intermediate Activations: Outputs from each layer that are needed for gradient calculations.**
  + **Gradients: Derivatives of the loss function with respect to weights, which are computed using the chain rule.**
* **Variable Attributes:**
  + **Memory Usage: Storing activations consumes memory, which can vary based on network size and batch size.**
  + **Computation Time: Accessing stored activations can affect the speed of backpropagation.**

**Distinction:**

* **Forward Pass: Computes and stores activations.**
* **Backward Pass: Uses stored activations to compute gradients.**

**Math Expression: The gradient of the loss L with respect to a weight w in layer l can be expressed as:**

**dLdwl=(dLdal)×(daldzl)×(dzldwl)**

**Example in Python:**

**Python**

**import torch**

**import torch.nn as nn**

**class SimpleNN(nn.Module):**

**def \_\_init\_\_(self):**

**super(SimpleNN, self).\_\_init\_\_()**

**self.fc1 = nn.Linear(10, 5)**

**self.fc2 = nn.Linear(5, 1)**

**def forward(self, x):**

**self.a1 = torch.relu(self.fc1(x)) # Store intermediate activation**

**output = self.fc2(self.a1)**

**return output**

**model = SimpleNN()**

**input\_data = torch.randn(1, 10)**

**output = model(input\_data)**

**In this example, self.a1 stores the intermediate activation from the first layer, which will be used during backpropagation to compute gradients.**

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

**Having activations with a standard deviation too far away from 1 can lead to several issues in neural networks:**

1. **Vanishing Gradients: If the standard deviation is too small, the gradients during backpropagation can become very small, making it difficult for the network to learn.**
2. **Exploding Gradients: If the standard deviation is too large, the gradients can become excessively large, causing unstable updates and potentially leading to numerical overflow.**
3. **Poor Convergence: Both vanishing and exploding gradients can result in slow or poor convergence, making the training process inefficient or ineffective.**
4. **Ineffective Weight Initialization: Proper weight initialization often assumes that activations have a standard deviation close to 1. Deviations from this can disrupt the balance and scaling of the network.**

**To summarize, maintaining activations with a standard deviation close to 1 is crucial for stable and efficient training of neural networks.**

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

**Weight initialization helps avoid the problem of vanishing or exploding gradients in neural networks. Proper initialization ensures that the weights are set to values that allow the network to learn effectively during training.**

**Clear definition:**

* **Weight Initialization: The process of setting the initial values of the weights in a neural network before training begins.**

**Criteria for effective weight initialization:**

1. **Avoiding Vanishing Gradients: Ensuring that the gradients do not become too small, which would slow down learning.**
2. **Avoiding Exploding Gradients: Ensuring that the gradients do not become too large, which would cause instability in learning.**
3. **Symmetry Breaking: Ensuring that weights are initialized differently to prevent neurons from learning the same features.**

**Common methods of weight initialization:**

1. **Zero Initialization: Setting all weights to zero.**
   * **Not recommended as it fails to break symmetry.**
2. **Random Initialization: Setting weights to small random values.**
   * **Helps in breaking symmetry but may still cause vanishing/exploding gradients.**
3. **Xavier/Glorot Initialization: Setting weights based on the number of input and output neurons.**
   * **Formula:**

**W U(−6n∈+nout,6n∈+nout)**

* + **Balances the variance of activations and gradients.**

1. **He Initialization: Setting weights based on the number of input neurons.**
   * **Formula:**

**W N(0,2n∈)**

* + **Suitable for ReLU activation functions.**

**Example in Python using TensorFlow:**

**Python**

**import tensorflow as tf**

**# Xavier/Glorot Initialization**

**initializer = tf.keras.initializers.GlorotUniform()**

**# He Initialization**

**initializer = tf.keras.initializers.HeNormal()**

**By using appropriate weight initialization techniques, neural networks can train more efficiently and effectively, avoiding common pitfalls associated with poor initialization.**