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

What is torch Unsqueeze?

torch. unsqueeze (input, dim) → Tensor. 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.

How does torch squeeze work?

squeeze. Returns a tensor with all the dimensions of input of size 1 removed. For example, if input is of shape: ( A × 1 × B × C × 1 × D ) (A \times 1 \times B \times C \times 1 \times D) (A×1×B×C×1×D) then the out tensor will be of shape: ( A × B × C × D ) (A \times B \times C \times D) (A×B×C×D).

What is torch cat?

torch.cat() is used to concatenate two or more tensors, whereas torch. stack() is used to stack the tensors. We can join the tensors in different dimensions such as 0 dimension, -1 dimension. Both torch.cat() and torch. stack() are used to join the tensors.06-Nov-2021

How do you add a dimension to a torch tensor?

Using None indexing You can also add dimensions to the end of tensors, which can be useful for broadcasting operations like pairwise distance. All you have to do is rearrange the colons and the None (s). Notice that we have to specify the first two dimensions with colons and then add a None to the end.21-Oct-2021

What does Unsqueeze mean?

(transitive) To reformat (a video image) from 4:3 to 16:9 aspect ratio.

What does torch sigmoid do?

The PyTorch sigmoid function is an element-wise operation that squishes any real number into a range between 0 and 1.13-May-2021

What is Torch flatten?

PyTorch Flatten is used to reshape any tensor with different dimensions to a single dimension so that we can do further operations on the same input data. The shape of the tensor will be the same as that of the number of elements in the tensor.

How can I reduce my tensor size?

1 Answer

For the first case, use resize\_() to change second dimension from 512 to 256 and then allocate a tensor with your padding value and the target dimensions and assign the portion for which you have data.

For the second case, you can simply use resize\_() for resizing your tensor to half the size.

What does tensor squeeze do?

Squeezing a tensor removes the dimensions or axes that have a length of one. Unsqueezing a tensor adds a dimension with a length of one.

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

To squeeze a tensor we can apply the torch.squeeze() method and to unsqueeze a tensor we use the torch.unsqueeze() method. Let’s understand these methods in detail.

Squeeze a Tensor:

When we squeeze a tensor, the dimensions of size 1 are removed. The elements of the original tensor are arranged with the remaining dimensions. For example, if the input tensor is of shape: (m×1×n×1) then the output tensor after squeeze will be of shape: (m×n). The following is the syntax of the torch.squeeze() method.

Syntax: torch.squeeze(input, dim=None, \*, out=None)

Parameters:

input: the input tensor.

dim: an optional integer value, if given the input is squeezed in this dimension.

out: the output tensor, an optional key argument.

Return: It returns a tensor with all the dimensions of input tensor of size 1 removed.

Please note that we can squeeze the input tensor in a particular dimension dim. In this case, other dimensions of size 1 will remain unchanged. We have discussed Example 2 in more detail.

Example 1:

In the example below we squeeze a 5D tensor using torch.squeeze() method. The input tensor has two dimensions of size 1.

Python3

|  |
| --- |
| # Python program to squeeze the tensor  # importing torch  import torch    # creating the input tensor  input = torch.randn(3,1,2,1,4)  # print the input tensor  print("Input tensor Size:\n",input.size())    # squeeze the tensor  output = torch.squeeze(input)  # print the squeezed tensor  print("Size after squeeze:\n",output.size()) |

Output:

Input tensor Size:

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

Size after squeeze:

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

Notice that both dimensions of size 1 are removed in the squeezed tensor.

Example 2:

In this example, We squeeze the tensor into different dimensions.

Python3

|  |
| --- |
| # 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])

Notice that when we squeeze the tensor in dimension 0, there is no change in the shape of the output tensor. When we squeeze in dimension 1 or in dimension 3 (both are of size 1), only this dimension is removed in the output tensor. When we squeeze in dimension 2 or in dimension 4, there is no change in the shape of the output tensor.

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.

Example 3:

In the example below we unsqueeze a 1-D tensor to a 2D tensor.

Python3

|  |
| --- |
| # 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])

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.)

NumPy is the foundation of the Python machine learning stack. NumPy allows for efficient operations on the data structures often used in machine learning: vectors, matrices, and tensors. While NumPy is not the focus of this book, it will show up frequently throughout the following chapters. This chapter covers the most common NumPy operations we are likely to run into while working on machine learning workflows.

1.1 Creating a Vector

Problem

You need to create a vector.

Solution

Use NumPy to create a one-dimensional array:

# Load library

import numpy as np

# Create a vector as a row

vector\_row = np.array([1, 2, 3])

# Create a vector as a column

vector\_column = np.array([[1],

[2],

[3]])

Discussion

NumPy’s main data structure is the multidimensional array. To create a vector, we simply create a one-dimensional array. Just like vectors, these arrays can be represented horizontally (i.e., rows) or vertically (i.e., columns).

See Also

[Vectors, Math Is Fun](http://bit.ly/2FB5q1v)

[Euclidean vector, Wikipedia](http://bit.ly/2FtnRoL)

1.2 Creating a Matrix

Problem

You need to create a matrix.

Solution

Use NumPy to create a two-dimensional array:

# Load library

import numpy as np

# Create a matrix

matrix = np.array([[1, 2],

[1, 2],

[1, 2]])

Discussion

To create a matrix we can use a NumPy two-dimensional array. In our solution, the matrix contains three rows and two columns (a column of 1s and a column of 2s).

NumPy actually has a dedicated matrix data structure:

matrix\_object = np.mat([[1, 2],

[1, 2],

[1, 2]])

matrix([[1, 2],

[1, 2],

[1, 2]])

However, the matrix data structure is not recommended for two reasons. First, arrays are the de facto standard data structure of NumPy. Second, the vast majority of NumPy operations return arrays, not matrix objects.

See Also

[Matrix, Wikipedia](http://bit.ly/2Ftnevp)

[Matrix, Wolfram MathWorld](http://bit.ly/2Fut7IJ)

1.3 Creating a Sparse Matrix

Problem

Given data with very few nonzero values, you want to efficiently represent it.

Solution

Create a sparse matrix:

# Load libraries

import numpy as np

from scipy import sparse

# Create a matrix

matrix = np.array([[0, 0],

[0, 1],

[3, 0]])

# Create compressed sparse row (CSR) matrix

matrix\_sparse = sparse.csr\_matrix(matrix)

Discussion

A frequent situation in machine learning is having a huge amount of data; however, most of the elements in the data are zeros. For example, imagine a matrix where the columns are every movie on Netflix, the rows are every Netflix user, and the values are how many times a user has watched that particular movie. This matrix would have tens of thousands of columns and millions of rows! However, since most users do not watch most movies, the vast majority of elements would be zero.

Sparse matrices only store nonzero elements and assume all other values will be zero, leading to significant computational savings. In our solution, we created a NumPy array with two nonzero values, then converted it into a sparse matrix. If we view the sparse matrix we can see that only the nonzero values are stored:

# View sparse matrix

print(matrix\_sparse)

(1, 1) 1

(2, 0) 3

There are a number of types of sparse matrices. However, in compressed sparse row (CSR) matrices, (1, 1) and (2, 0) represent the (zero-indexed) indices of the non-zero values 1 and 3, respectively. For example, the element 1 is in the second row and second column. We can see the advantage of sparse matrices if we create a much larger matrix with many more zero elements and then compare this larger matrix with our original sparse matrix:

# Create larger matrix

matrix\_large = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],

[3, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

# Create compressed sparse row (CSR) matrix

matrix\_large\_sparse = sparse.csr\_matrix(matrix\_large)

# View original sparse matrix

print(matrix\_sparse)

(1, 1) 1

(2, 0) 3

# View larger sparse matrix

print(matrix\_large\_sparse)

(1, 1) 1

(2, 0) 3

As we can see, despite the fact that we added many more zero elements in the larger matrix, its sparse representation is exactly the same as our original sparse matrix. That is, the addition of zero elements did not change the size of the sparse matrix.

As mentioned, there are many different types of sparse matrices, such as compressed sparse column, list of lists, and dictionary of keys. While an explanation of the different types and their implications is outside the scope of this book, it is worth noting that while there is no “best” sparse matrix type, there are meaningful differences between them and we should be conscious about why we are choosing one type over another.

See Also

[Sparse matrices, SciPy documentation](http://bit.ly/2HReBZR)

[101 Ways to Store a Sparse Matrix](http://bit.ly/2HS43cI)

1.4 Selecting Elements

Problem

You need to select one or more elements in a vector or matrix.

Solution

NumPy’s arrays make that easy:

# Load library

import numpy as np

# Create row vector

vector = np.array([1, 2, 3, 4, 5, 6])

# Create matrix

matrix = np.array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9]])

# Select third element of vector

vector[2]

3

# Select second row, second column

matrix[1,1]

5

Discussion

Like most things in Python, NumPy arrays are zero-indexed, meaning that the index of the first element is 0, not 1. With that caveat, NumPy offers a wide variety of methods for selecting (i.e., indexing and slicing) elements or groups of elements in arrays:

# Select all elements of a vector

vector[:]

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

# Select everything up to and including the third element

vector[:3]

array([1, 2, 3])

# Select everything after the third element

vector[3:]

array([4, 5, 6])

# Select the last element

vector[-1]

6

# Select the first two rows and all columns of a matrix

matrix[:2,:]

array([[1, 2, 3],

[4, 5, 6]])

# Select all rows and the second column

matrix[:,1:2]

array([[2],

[5],

[8]])

1.5 Describing a Matrix

Problem

You want to describe the shape, size, and dimensions of the matrix.

Solution

Use shape, size, and ndim:

# Load library

import numpy as np

# Create matrix

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

[5, 6, 7, 8],

[9, 10, 11, 12]])

# View number of rows and columns

matrix.shape

(3, 4)

# View number of elements (rows \* columns)

matrix.size

12

# View number of dimensions

matrix.ndim

2

Discussion

This might seem basic (and it is); however, time and again it will be valuable to check the shape and size of an array both for further calculations and simply as a gut check after some operation.

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

Broadcasting should not increase the memory usage, but you would of course need to store the result.

Broadcasting means that in an array expression if an array has a unitary dimension along an axis that dimension is automatically expanded to match the size of the other array. For example if an array has size [3,1] and another has size [1,4] both array are expanded as they have size [3,4]. In the actual Fortran one has to use spread to achieve the same result.  
Another nice feature of numpy is the possibility to add unitary dimensions on the fly with None . Python example:

a = np.ones(2)

b= np.ones(3)

a[:,None] \* b[None,:]

Moreover some functions that reduce the dimensionality like sum can have an extra arguments keepdims that doesn’t reduce the dimensionality but make that dimension unitary. This is useful as one can write (in Python) somenthing like:  
a/np.mean(a, axis=0,keepdims=True)  
So I suggest the following feature:  
Use the symbol “+” (or whatever other symbol) to add a dimension to a section of an array. The function sum and other that reduce the dimensionality should have an extra keyword argument like keepdims that doesn’t eliminate that dimension.  
As an example:

integer :: a(3), b(4), c(3,4), d(4), i

a = [(i=1,3)]

b = [(i=1,4)]

c = a(:,+) + b(+,:)

d = c/sum(c, dim=1, keepdim=.true.)

Contrary what happen to Matlab and Python adding the extra dimension is obligatory (no implicit unary dimensions).  
The point is that you may not know, in general, if an expression will involve broadcasting at compile time (except for those simple examples), so at run time the processor has to check the extent of all the arrays involved in the array expression, and that may affect the speed. On the other hand it may be safer to require the processor to report if in array expressions the dimensions are not compatible. It may make the program a little slower but safer (with an explicit loop the burden should be all on the shoulder of the programmer).

Or one can decide to make things more explicit adding an attribute to arrays, like broadcastable which indicates that any unitary dimension of that array can be expanded.  
A section with an added dimension with “+” like in the previous example will be broadcastable , and the return of a function like the sum with keepdim will be broadcastable (or one uses a different function like sum\_keepdim, which returns a broadcastable array, while the old one return a normal array).  
All other array expressions can remain as before, without the overhead to check the extent of all the involved arrays.

integer, broadcastable :: a(3,1), b(1,4)

integer :: c(3,4)

integer :: dd(3,1), ee(1,4)

c = a + b ! broadcasting

! c = dd + ee ! illegal in current and future Fortran

6.Implement matmul using Einstein summation.

Einstein’s summation in Deep Learning for making your life easier.

To deal with multi-dimensional computations back in 1916 Albert Einstein developed a compact form to show summation over some indexes. The so-called Einstein summation convention is what we use when we call einsum.

Einsum is a way of performing tensor and array operations. Einsum is extremely convenient and compact. It is an operation that can be replaced in so many tensor operations. If we are dealing with matrix multiplication, dot products, specific summation, and batch matrix multiplication, we discover that einsum can replace some of these, and even combine so we are using neat and clean notation, and in the code, we are getting rid of so many unnecessary loops, lines, proving also clean code is not a myth!

You don’t have to remember the syntax for matrix multiplication in NumPy, PyTorch, or TensorFlow. Or let’s say you need to permute the input in the function calls ordering, in batch matrix multiplication, that can be also done inside einsum. You don’t even have to permute the output.

SO HOW DOES IT WORK?

Best way to understand it would be on a concrete example. I assume you are already familiar with matrix multiplication.

Let’s compare this to the code using nested loops. The best way to check if you are using the einsum notation in the right way, looping is the excellent method you can use until you get the intuition of what’s going on, and gain confidence to write einsum more easily.

A = np.random.rand(3,5)  
B = np.random.rand(5,2)   
M = np.empty((3,2))   
   
for i in range(3):   
 for j in range(2):   
 total = 0   
 for k in range(5):  
 total +=A[i,k]\*B[k,j]   
 M[i,j] = total

Where we have the two outer loops, and the inner loop summing over the element-wise multiplication AB

This can be represented using einsum in one call:

M = np.einsum('ik,kj->ij', A, B)

Where the ‘ik’ specifies the dimension of the first input A, and the ‘kj’ specifies the dimensions of the second input B, the arrow suggests the shape and the dimension of the ‘ij’ that represents M.

‘k’ is repeated over input so this means these dimension indices are multiplied.

Free indices: are specified in the output. ‘i,j’

Summation index: all other indices. (Those that appear in the input, but NOT in the output specification) ‘k’

GENERAL RULES FOR THE EINSUM

Repeating letters in different inputs means those values will be multiplied, and those products will be the output

M = np.einsum('ik,kj->ij', A, B)

2. The omitting letter means the axis would be summed.

x = np.ones(3) sum\_x = np.einsum('i->', x)

3. .We can return the not-summed axis in any order:

x = np.ones((5, 4, 3)) np.einsum('ijk->kji', x)

Operations are well presented on NumPy documentation. I warmly suggest you practice every one of them so you get more familiar with when to use them.

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

A basic introduction to NumPy's einsum

The einsum function is one of NumPy’s jewels. It can often outperform familiar array functions in terms of speed and memory efficiency, thanks to its expressive power and smart loops. On the downside, it can take a little while understand the notation and sometimes a few attempts to apply it correctly to a tricky problem.

There are quite a few questions on sites like Stack Overflow which about what einsum does and how it works, so this post hopes to serve as a basic introduction to the function and what you need to know to begin using it.

What einsum does

Using the einsum function, we can specify operations on NumPy arrays using the [Einstein summation convention](https://en.wikipedia.org/wiki/Einstein_notation).

Suppose we have two arrays, A and B. Now suppose that we want to:

multiply A with B in a particular way to create new array of products, and then maybe

sum this new array along particular axes, and/or

transpose the axes of the array in a particular order.

Then there’s a good chance einsum will help us do this much faster and more memory-efficiently that combinations of the NumPy functions multiply, sum and transpose would allow.

As a small example of the function’s power, here are two arrays that we want to multiply element-wise and then sum along axis 1 (the rows of the array):

A = np.array([0, 1, 2])

B = np.array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

How do we normally do this in NumPy? The first thing to notice is that we need to reshape A so that we can broadcast it with B (specifically A needs to be column vector). Then we can multiply 0 with the first row of B, multiply 1 with the second row, and 2 with the third row. This will give us a new array and the three rows can then be summed.

Putting this together, we have:

>>> (A[:, np.newaxis] \* B).sum(axis=1)

array([ 0, 22, 76])

This works fine, but using einsum we can do better:

>>> np.einsum('i,ij->i', A, B)

array([ 0, 22, 76])

Why better? In short because we didn’t need to reshape A at all and, most importantly, the multiplication didn’t create a temporary array like A[:, np.newaxis] \* B did. Instead, einsum simply summed the products along the rows as it went. Even for this tiny example, I timed einsum to be about three times faster.

How to use einsum

The key is to choose the correct labelling for the axes of the inputs arrays and the array that we want to get out.

The function lets us do that in one of two ways: using a string of letters, or using lists of integers. For simplicity, we’ll stick to the strings (this appears to be the more commonly used of the two options).

A good example to look at is matrix multiplication, which involves multiplying rows with columns and then summing the products. For two 2D arrays A and B, matrix multiplication can be done with np.einsum('ij,jk->ik', A, B).

What does this string mean? Think of 'ij,jk->ik' as split in two at the arrow ->. The left-hand part labels the axes of the input arrays: 'ij' labels A and 'jk' labels B. The right-hand part of the string labels the axes of the single output array with the letters 'ik'. In other words, we’re putting two 2D arrays in and we want a new 2D array out.

The two arrays we’ll multiply are:

A = np.array([[1, 1, 1],

[2, 2, 2],

[5, 5, 5]])

B = np.array([[0, 1, 0],

[1, 1, 0],

[1, 1, 1]])

Drawing on the labels, our matrix multiplication with np.einsum('ij,jk->ik', A, B) looks like this:

To understand how the output array is calculated, remember these three rules:

Repeating letters between input arrays means that values along those axes will be multiplied together. The products make up the values for the output array.

In this case, we used the letter j twice: once for A and once for B. This means that we’re multiplying each row of A with each column of B. This will only work if the axis labelled by j is the same length in both arrays (or the length is 1 in either array).

Omitting a letter from the output means that values along that axis will be summed.

Here, j is not included among the labels for the output array. Leaving it out sums along the axis and explicitly reduces the number of dimensions in the final array by 1. Had the output signature been 'ijk' we would have ended up with a 3x3x3 array of products. (And if we gave no output labels but just write the arrow, we’d simply sum the whole array.)

We can return the unsummed axes in any order we like.

If we leave out the arrow '->', NumPy will take the labels that appeared once and arrange them in alphabetical order (so in fact 'ij,jk->ik' is equivalent to just 'ij,jk'). If we want to control what our output looked like we can choose the order of the output labels ourself. For example, 'ij,jk->ki' delivers the transpose of the matrix multiplication (notice that k and i were switched in the output labelling).

Note that with np.einsum('ij,jk->ik', A, B), the function doesn’t construct a 3D array and then sum, it just accumulates the sums into a 2D array.

A handful of simple operations

That’s all we need to know to start using einsum. Knowing how to multiply different axes together and then how to sum the products, we can express a lot of different operations succinctly. This allows us to generalise problems to higher-dimensions relatively easily. For example, we don’t have to insert new axes or transpose arrays to get them to line up correctly.

Below are two tables showing how einsum can stand in for various NumPy operations. It’s useful to play about with these to get the hang of the notation.

Let A and B be two 1D arrays of compatible shapes (meaning the lengths of the axes we pair together either equal, or one of them has length 1):

|  |  |  |
| --- | --- | --- |
| Call signature | NumPy equivalent | Description |
| ('i', A) | A | returns a view of A |
| ('i->', A) | sum(A) | sums the values of A |
| ('i,i->i', A, B) | A \* B | element-wise multiplication of A and B |
| ('i,i', A, B) | inner(A, B) | inner product of A and B |
| ('i,j->ij', A, B) | outer(A, B) | outer product of A and B |

Now let A and B be two 2D arrays with compatible shapes:

|  |  |  |
| --- | --- | --- |
| Call signature | NumPy equivalent | Description |
| ('ij', A) | A | returns a view of A |
| ('ji', A) | A.T | view transpose of A |
| ('ii->i', A) | diag(A) | view main diagonal of A |
| ('ii', A) | trace(A) | sums main diagonal of A |
| ('ij->', A) | sum(A) | sums the values of A |
| ('ij->j', A) | sum(A, axis=0) | sum down the columns of A (across rows) |
| ('ij->i', A) | sum(A, axis=1) | sum horizontally along the rows of A |
| ('ij,ij->ij', A, B) | A \* B | element-wise multiplication of A and B |
| ('ij,ji->ij', A, B) | A \* B.T | element-wise multiplication of A and B.T |
| ('ij,jk', A, B) | dot(A, B) | matrix multiplication of A and B |
| ('ij,kj->ik', A, B) | inner(A, B) | inner product of A and B |
| ('ij,kj->ikj', A, B) | A[:, None] \* B | each row of A multiplied by B |
| ('ij,kl->ijkl', A, B) | A[:, :, None, None] \* B | each value of A multiplied by B |

When working with larger numbers of dimensions, keep in mind that einsum allows the ellipses syntax '...'. This provides a convenient way to label the axes we’re not particularly interested in, e.g. np.einsum('...ij,ji->...', a, b) would multiply just the last two axes of a with the 2D array b. There are more examples in the documentation.

A few quirks to watch out for

Here a few things to be mindful/wary of when using the function.

einsum [does not promote data types when summing](http://stackoverflow.com/a/18366008/3923281). If you’re using a more limited datatype, you might get unexpected results:

>>> a = np.ones(300, dtype=np.int8)

>>> np.sum(a) # correct result

300

>>> np.einsum('i->', a) # produces incorrect result

44

Also einsum [might not permute axes in the order inteded](http://stackoverflow.com/a/28233465/3923281). The documentation highlights np.einsum('ji', M) as a way to transpose a 2D array. You’d be forgiven for thinking that for a 3D array, np.einsum('kij', M) moves the last axis to the first position and shifts the first two axes along. Actually, einsum creates its own output labelling by rearranging labels in alphabetical order. Therefore 'kij' becomes 'kij->ijk' and we have a sort of inverse permutation instead.

Finally, einsum is not always the fastest option in NumPy. Functions such as dot and inner often link to lightening-quick BLAS routines which can outperform einsum and certainly shouldn’t be forgotten about. The tensordot function is also worth comparing for speed. If you search around, you’ll find examples of posts highlighting cases where einsum appears to be slow, especially when operating on several input arrays (such as [this GitHub issue](https://github.com/numpy/numpy/issues/5366)).

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

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.

Einstein summation is a [notational convention](https://mathworld.wolfram.com/Notation.html) for simplifying expressions including [summations](https://mathworld.wolfram.com/Sum.html) of [vectors](https://mathworld.wolfram.com/Vector.html), [matrices](https://mathworld.wolfram.com/Matrix.html), and general [tensors](https://mathworld.wolfram.com/Tensor.html). There are essentially three rules of Einstein summation notation, namely:

1. Repeated indices are implicitly summed over.

2. Each index can appear at most twice in any term.

3. Each term must contain identical non-repeated indices.

The first item on the above list can be employed to greatly simplify and shorten equations involving [tensors](https://mathworld.wolfram.com/Tensor.html). are invalid because the index  appears three times in the first term of (), while the non-repeated index  in the first term of () doesn't match the non-repeated  of the second term.

The convention was introduced by Einstein (1916, sec. 5), who later jested to a friend, "I have made a great discovery in mathematics; I have suppressed the summation sign every time that the summation must be made over an index which occurs twice..." (Kollros 1956; Pais 1982, p. 216).

In practice, the convention tends to occur alongside both the [Kronecker delta](https://mathworld.wolfram.com/KroneckerDelta.html) and [permutation symbol](https://mathworld.wolfram.com/PermutationSymbol.html). Moreover, the Einstein summation convention easily accommodates both superscripts and subscripts for [contravariant](https://mathworld.wolfram.com/ContravariantTensor.html) and [covariant tensors](https://mathworld.wolfram.com/CovariantTensor.html), respectively.

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

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.

We have dived deep into [what is a Neural Network, its structure and components, Gradient Descent, its limitations](https://www.analyticsvidhya.com/blog/2021/04/is-gradient-descent-sufficient-for-neural-network/) and [how are neurons estimated, and the working of the forward propagation](https://www.analyticsvidhya.com/blog/2021/04/estimation-of-neurons-and-forward-propagation-in-neural-net/).

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.

Backward Propagation is the preferable method of adjusting or correcting the weights to reach the minimized loss function. In this article, we shall explore this second technique of Backward Propagation in detail by understanding how it works mathematically, why it is the preferred method. A caution, the article is going to be a mathematically heavy one but wait for the end to see how this method looks in action

Table of Contents

Setting up the Base

Contribution of each Weight and Bias on the Error

Matrix Form of the Backward Propagation

Setting up the Base

Let’s say we want to use the neural network to predict house prices. For our understanding purpose here, we will take a subset dummy dataset having four input variables and six observations here with the input having a dimension of 4\*5:

The architecture of the neural network is [4, 5, 1] with:

4 independent variables, Xs in the input layer

5 nodes in the hidden layer, and

Since we have a regression problem at hand, we will have one node in the output layer.

A Neural Network operates by:

Initializing the weights with some random values, which are mostly between 0 and 1.

Compute the output to calculate the loss or the error term.

Then, adjust the weights so that to minimize the loss.

We repeat these three steps until have reached the optimum solution of the minimum loss function or exhausted the pre-defined epochs (i.e. the number of iterations).

In case you are wondering how and from where this equation arrived and why there will be matrix dimensions then request you to read the [previous article](https://www.analyticsvidhya.com/blog/2021/04/estimation-of-neurons-and-forward-propagation-in-neural-net/) to understand the mechanism of how neural networks work and are estimated.

Building on this, the first step in Backward Propagation to calculate the error. In our regression problem, we shall take the loss function = (Y-Y^)2/2 where Y is actual values and Y^ is predicted values. For simplicity, replacing Y^ with O, so the error E becomes = (Y-O)2/2.

Our goal is to minimize the error that is clearly dependent on Y, which is the actual observed values, and on the output, which is further is dependent on the:

input values

coefficients or betas of the input variables

biases, the [activation function](https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6?gi=2074f5569281), and

[Optimizers](https://www.analyticsvidhya.com/blog/2021/04/is-gradient-descent-sufficient-for-neural-network/)

Now, we can neither change the input variables nor the actual Y values however, we can change the other factors. The activation function and the optimizers are the tuning parameters – and we can change these based on our requirement.

The other two factors: the coefficients or betas of the input variables (Wis) and the biases (bho, bih) are updated using the Gradient descent algorithm with the following equation:

Wnew = Wold – (α \* dE/dW)

where,

Wnew = the new weight of Xi

Wold = the old weight of the Xi

α = learning rate

dE/dW is the partial derivative of the error for each of the Xs. It is the rate of change of the error to the change in weight.

In the backward propagation, we adjust these weights or the betas in the output. The weights and biases between the respective input, hidden and output layers we have here are Wih, bih, Who, and bho:

In the first iteration, we randomly initialize the weights. In the second iteration, we change the weights of the hidden layer that is closest to the output layer. In this case, we go from the output layer, hidden layer, and then to the input layer.

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

However, we need to store all intermediate activations during training because they are needed to compute gradients during back-propagation: (5)-(8) involve not just the values of the incoming gradient, but also the values of the activations themselves.

The neural network is one of the most widely used machine learning algorithms. The successful applications of neural networks in fields such as image classification, time series forecasting, and many others have paved the way for its adoption in business and research. It is fair to say that the neural network is one of the most important machine learning algorithms. A clear understanding of the algorithm will come in handy in diagnosing issues and also in understanding other advanced deep learning algorithms. The goal of this article is to explain the workings of a neural network. We will do a step-by-step examination of the algorithm and also explain how to set up a simple neural network in PyTorch. We will also compare the results of our calculations with the output from PyTorch.

1.0 Combination of functions:

Let’s start by considering the following two arbitrary linear functions:

The coefficients -1.75, -0.1, 0.172, and 0.15 have been arbitrarily chosen for illustrative purposes.

used above is called the sigmoid function. It is an S-shaped curve. The function f(x) has a special role in a neural network. We will discuss it in more detail in a subsequent section. For now, we simply apply it to construct functions a₁ and a₂.

Once again, the coefficients 0.25, 0.5, and 0.2 are arbitrarily chosen. Figure 1 shows a plot of the three functions a₁, a₂, and z₃.

We can see from Figure 1 that the linear combination of the functions a₁ and a₂ is a more complex-looking curve. In other words, by linearly combining curves, we can create functions that are capable of capturing more complex variations. We can extend the idea by applying the sigmoid function to z₃ and linearly combining it with another similar function to represent an even more complex function. In theory, by combining enough such functions we can represent extremely complex variations in values. The coefficients in the above equations were selected arbitrarily. What if we could change the shapes of the final resulting function by adjusting the coefficients? That would allow us to fit our final function to a very complex dataset. This is the basic idea behind a neural network. The neural network provides us a framework to combine simpler functions to construct a complex function that is capable of representing complicated variations in data. Let us now examine the framework of a neural network.

2.0 A simple neural network:

Figure 2 is a schematic representation of a simple neural network. We will use this simple network for all the subsequent discussions in this article. The network takes a single value (x) as input and produces a single value y as output. There are four additional nodes labeled 1 through 4 in the network.

The input node feeds node 1 and node 2. Node 1 and node 2 each feed node 3 and node 4. Finally, node 3 and node 4 feed the output node. w₁ through w₈ are the weights of the network, and b₁ through b₈ are the biases. The weights and biases are used to create linear combinations of values at the nodes which are then fed to the nodes in the next layer. For example, the input x combined with weight w₁ and bias b₁ is the input for node 1. Similarly, the input x combined with weight w₂ and bias b₂ is the input for node 2. AF at the nodes stands for the activation function. The sigmoid function presented in the previous section is one such activation function. We will discuss more activation functions soon. For now, let us follow the flow of the information through the network. The outputs produced by the activation functions at node 1 and node 2 are then linearly combined with weights w₃ and w₅ respectively and bias b₃. The linear combination is the input for node 3. Similarly, outputs at node 1 and node 2 are combined with weights w₆ and w₄ respectively and bias b₄ to feed to node 4. Finally, the output from the activation function at node 3 and node 4 are linearly combined with weights w₇ and w₈ respectively, and bias b₅ to produce the network output yhat.

This Flow of information from the input to the output is also called the forward pass. Before we work out the details of the forward pass for our simple network, let’s look at some of the choices for activation functions.

The plots of each activation function and its derivatives are also shown. While the sigmoid and the tanh are smooth functions, the RelU has a kink at x=0. The choice of the activation function depends on the problem we are trying to solve. There are applications of neural networks where it is desirable to have a continuous derivative of the activation function. For such applications, functions with continuous derivatives are a good choice. The tanh and the sigmoid activation functions have larger derivatives in the vicinity of the origin. Therefore, if we are operating in this region these functions will produce larger gradients leading to faster convergence. In contrast, away from the origin, the tanh and sigmoid functions have very small derivative values which will lead to very small changes in the solution. We will discuss the computation of gradients in a subsequent section. There are many other activation functions that we will not discuss in this article. Since the RelU function is a simple function, we will use it as the activation function for our simple neural network. We are now ready to perform a forward pass.

3.0 Forward pass:

Figure 3 shows the calculation for the forward pass for our simple neural network.

z₁ and z₂ are obtained by linearly combining the input x with w₁ and b₁ and w₂ and b₂ respectively. a₁ and a₂ are the outputs from applying the RelU activation function to z₁ and z₂ respectively. z₃ and z₄ are obtained by linearly combining a₁ and a₂ from the previous layer with w₃, w₅, b₃, and w₄, w₆, b₄ respectively. Finally, the output yhat is obtained by combining a₃ and a₄ from the previous layer with w₇, w₈, and b₅. In practice, the functions z₁, z₂, z₃, and z₄ are obtained through a matrix-vector multiplication as shown in figure 4.

Here we have combined the bias term in the matrix. In general, for a layer of r nodes feeding a layer of s nodes as shown in figure 5, the matrix-vector product will be (s X r+1) \* (r+1 X 1).

The final step in the forward pass is to compute the loss. Since we have a single data point in our example, the loss L is the square of the difference between the output value yhat and the known value y. In general, for a regression problem, the loss is the average sum of the square of the difference between the network output value and the known value for each data point. It is called the mean squared error. This completes the first of the two important steps for a neural network. Before discussing the next step, we describe how to set up our simple network in PyTorch.

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

Why Initialize Weights

The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network. If either occurs, loss gradients will either be too large or too small to flow backwards beneficially, and the network will take longer to converge, if it is even able to do so at all.

Matrix multiplication is the essential math operation of a neural network. In deep neural nets with several layers, one forward pass simply entails performing consecutive matrix multiplications at each layer, between that layer’s inputs and weight matrix. The product of this multiplication at one layer becomes the inputs of the subsequent layer, and so on and so forth.

For a quick-and-dirty example that illustrates this, let’s pretend that we have a vector x that contains some network inputs. It’s standard practice when training neural networks to ensure that our inputs’ values are scaled such that they fall inside such a normal distribution with a mean of 0 and a standard deviation of 1.

Let’s also pretend that we have a simple 100-layer network with no activations , and that each layer has a matrix a that contains the layer’s weights. In order to complete a single forward pass we’ll have to perform a matrix multiplication between layer inputs and weights at each of the hundred layers, which will make for a grand total of 100 consecutive matrix multiplications.

It turns out that initializing the values of layer weights from the same standard normal distribution to which we scaled our inputs is never a good idea. To see why, we can simulate a forward pass through our hypothetical network.

Whoa! Somewhere during those 100 multiplications, the layer outputs got so big that even the computer wasn’t able to recognize their standard deviation and mean as numbers. We can actually see exactly how long that took to happen.

The activation outputs exploded within 29 of our network’s layers. We clearly initialized our weights to be too large.

Unfortunately, we also have to worry about preventing layer outputs from vanishing. To see what happens when we initialize network weights to be too small — we’ll scale our weight values such that, while they still fall inside a normal distribution with a mean of 0, they have a standard deviation of 0.01.

During the course of the above hypothetical forward pass, the activation outputs completely vanished.

To sum it up, if weights are initialized too large, the network won’t learn well. The same happens when weights are initialized too small.

12.How can weight initialization help avoid this problem?

Building even a simple neural network can be a confusing task and upon that tuning it to get a better result is extremely tedious. But, the first step that comes in consideration while building a neural network is the initialization of parameters, if done correctly then optimization will be achieved in the least time otherwise converging to a minima using gradient descent will be impossible.

This article has been written under the assumption that the reader is already familiar with the concept of neural network, weight, bias, activation functions, forward and backward propagation etc.

Basic notations

Consider an L layer neural network, which has L-1 hidden layers and 1 input and output layer each. The parameters (weights and biases) for layer l are represented as.

In this article, we’ll have a look at some of the basic initialization practices in the use and some improved techniques that must be used in order to achieve a better result. Following are some techniques generally practised to initialize parameters :

Zero initialization

Random initialization

Zero initialization :

In general practice biases are initialized with 0 and weights are initialized with random numbers, what if weights are initialized with 0?

In order to understand let us consider we applied sigmoid activation function for the output layer.

If all the weights are initialized with 0, the derivative with respect to loss function is the same for every w in W[l], thus all weights have the same value in subsequent iterations. This makes hidden units symmetric and continues for all the n iterations i.e. setting weights to 0 does not make it better than a linear model. An important thing to keep in mind is that biases have no effect what so ever when initialized with 0.

W[l] = np.random.zeros((l-1,l))

let us consider a neural network with only three hidden layers with ReLu activation function in hidden layers and sigmoid for the output layer.

Using the above neural network on the dataset “make circles” from sklearn.datasets, the result obtained as the following :

for 15000 iterations, loss = 0.6931471805599453, accuracy = 50 %

Random initialization :

Assigning random values to weights is better than just 0 assignment. But there is one thing to keep in my mind is that what happens if weights are initialized high values or very low values and what is a reasonable initialization of weight values.

a) If weights are initialized with very high values the term np.dot(W,X)+b becomes significantly higher and if an activation function like sigmoid() is applied, the function maps its value near to 1 where the slope of gradient changes slowly and learning takes a lot of time.

b) If weights are initialized with low values it gets mapped to 0, where the case is the same as above.

This problem is often referred to as the vanishing gradient.

To see this let us see the example we took above but now the weights are initialized with very large values instead of 0 :

W[l] = np.random.randn(l-1,l)\*10

Neural network is the same as earlier, using this initialization on the dataset “make circles” from sklearn.datasets, the result obtained as the following :

for 15000 iterations, loss = 0.38278397192120406, accuracy = 86 %

This solution is better but doesn’t properly fulfil the needs so, let us see a new technique.

New Initialization techniques

As we saw above that with large or 0 initialization of weights(W), not significant result is obtained even if we use appropriate initialization of weights it is probable that training process is going to take longer time. There are certain problems associated with it :

a) If the model is too large and takes many days to train then what

b) What about vanishing/exploding gradient problem

These were some problems that stood in the path for many years but in 2015, He et al. (2015) proposed activation aware initialization of weights (for ReLu) that was able to resolve this problem. ReLu and leaky ReLu also solves the problem of vanishing gradient.

To see how effective this solution is, let us use the previous dataset and neural network we took for above initialization and results are :

for 15000 iterations, loss =0.07357895962677366, accuracy = 96 %

There are also some other techniques other than He initialization in use that is comparatively better than old techniques and are used frequently.

Xavier initialization: It is same as He initialization but it is used for tanh() activation function, in this method 2 is replaced with 1.

These methods serve as good starting points for initialization and mitigate the chances of exploding or vanishing gradients. They set the weights neither too much bigger than 1, nor too much less than 1. So, the gradients do not vanish or explode too quickly. They help avoid slow convergence, also ensuring that we do not keep oscillating off the minima. There exist other variants of the above, where the main objective again is to minimize the variance of the parameters.