deeplizard.com

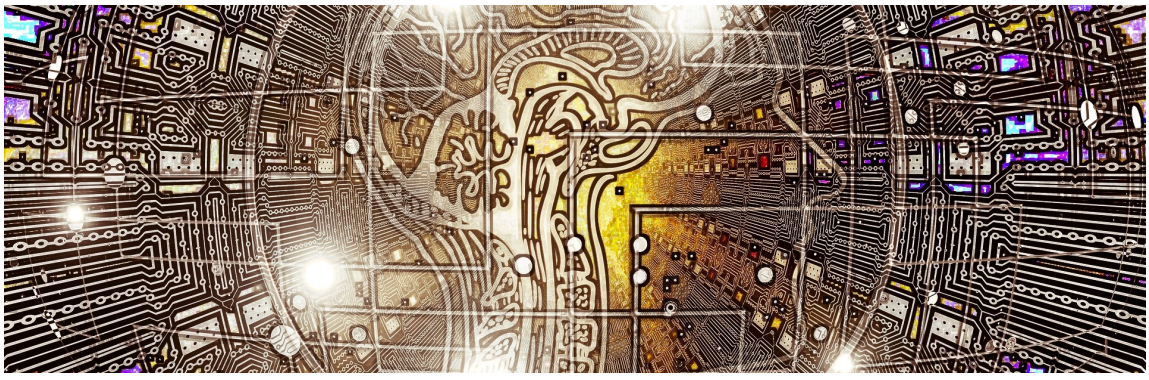
Flatten, Reshape, and Squeeze Explained - Tensors for Deep Learning with PyTorch

deeplizard

26-33 minutes

Reshaping operations - Tensors for deep learning

Welcome back to this series on neural network programming. Starting with this post in this series, we'll begin using the knowledge we've learned about tensors up to this point and start covering essential tensor operations for neural networks and deep learning.



We'll kick things off with reshaping operations. Without further ado, let's get started.

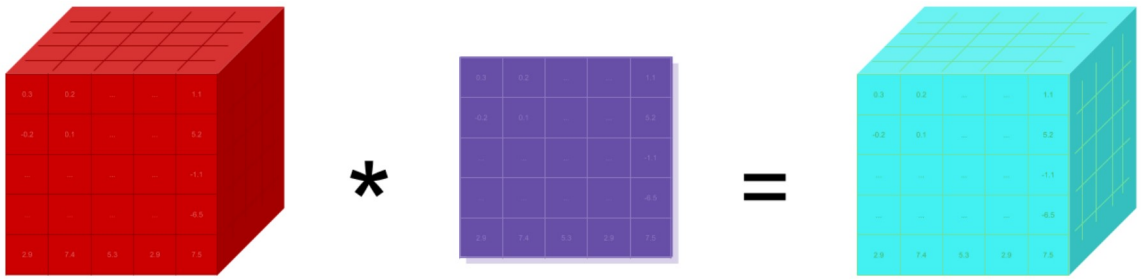
Tensor operation types

Before we dive in with specific tensor operations, let's get a quick overview of the landscape by looking at the main operation categories that encompass the operations we'll cover. We have the following high-level categories of operations:

1. Reshaping operations
2. Element-wise operations
3. Reduction operations
4. Access operations

There are a lot of individual operations out there, so much so that it can sometimes be intimidating when you're just beginning, but grouping similar operations into categories based on their likeness can help make learning about tensor operations more manageable.

The reason for showing these categories is to give you the goal of understanding all four of these by the end of this section in the series.



The goal of these posts on tensor operations is to not only showcase specific tensor operations commonly used, but to also describe the operation landscape. Having knowledge of the types of operations that exist can stay with us longer than just knowing or memorizing individual

operations.

Keep this in mind and work towards understanding these categories as we explore each of them. Let's jump in now with reshaping operations.

Reshaping operations for tensors

Reshaping operations are perhaps the most important type of tensor operations. This is because, like we mentioned in the post where we introduced tensors, the shape of a tensor gives us something concrete we can use to *shape* an intuition for our tensors.

An analogy for tensors

Suppose we are neural network programmers, and as such, we typically spend our days building neural networks. To do our job, we use various tools.

We use math tools like calculus and linear algebra, computer science tools like Python and PyTorch, physics and engineering tools like CPUs and GPUs, and machine learning tools like neural networks, layers, activation functions, etc.

Our task is to build neural networks that can transform or map input data to the correct output we are seeking.

The primary ingredient we use to produce our product, a function that maps inputs to correct outputs, is data.

Data is somewhat of an abstract concept, so when we want to actually use the concept of data to implement something, we use a specific data structure called a tensor that can be efficiently implemented in code. Tensors have properties, mathematical and otherwise, that allow us to do our work.

Tensors are the primary ingredient that neural network programmers use to produce their product, intelligence.

This is very similar to how a baker uses dough to produce, say, a pizza. The dough is the input used to create an output, but before the pizza is produced there is usually some form of reshaping of the input that is required.

As neural network programmers, we have to do the same with our tensors, and usually shaping and reshaping our tensors is a frequent task.

Our networks operate on tensors, after all, and this is why understanding a tensor's shape and the available reshaping operations are super important.

Instead of producing pizzas, we are producing intelligence! This may be lame, but whatever. Let's jump in with reshaping operations.

Tensor shape review

Suppose that we have the following tensor:

```
> t = torch.tensor([
    [1,1,1,1],
    [2,2,2,2],
    [3,3,3,3]
], dtype=torch.float32)
```

To determine the shape of this tensor, we look first at the rows `3` and then the columns `4`, and so this tensor is a `3 x`

`4` rank `2` tensor. Remember, *rank* is a word that is commonly used and just means the number of dimensions present within the tensor.

In PyTorch, we have two ways to get the shape:

```
> t.size()
torch.Size([3, 4])

> t.shape
torch.Size([3, 4])
```

In PyTorch the *size* and *shape* of a tensor mean the same thing.

Typically, after we know a tensor's shape, we can deduce a couple of things. First, we can deduce the tensor's rank. The rank of a tensor is equal to the length of the tensor's shape.

```
> len(t.shape)
2
```

We can also deduce the number of elements contained within the tensor. The number of elements inside a tensor (12 in our case) is equal to the product of the shape's component values.

```
> torch.tensor(t.shape).prod()
tensor(12)
```

In PyTorch, there is a dedicated function for this:

```
> t.numel()
12
```

The number of elements contained within a tensor is important for reshaping because the reshaping must account for the total number of elements present. Reshaping changes the tensor's shape but not the underlying data. Our tensor has `12` elements, so any reshaping must account for exactly `12` elements.

Reshaping a tensor in PyTorch

Let's look now at all the ways in which this tensor `t` can be reshaped without changing the rank:

```
> t.reshape([1,12])
tensor([[1., 1., 1., 1., 2., 2., 2., 2.,
3., 3., 3., 3.]])

> t.reshape([2,6])
tensor([[1., 1., 1., 1., 2., 2.],
        [2., 2., 3., 3., 3., 3.]])

> t.reshape([3,4])
tensor([[1., 1., 1., 1.],
        [2., 2., 2., 2.],
        [3., 3., 3., 3.]])

> t.reshape([4,3])
tensor([[1., 1., 1.],
        [1., 2., 2.],
        [2., 2., 3.],
        [3., 3., 3.]])
```

```
> t.reshape(6,2)
tensor([[1., 1.],
        [1., 1.],
        [2., 2.],
        [2., 2.],
        [3., 3.],
        [3., 3.]])

> t.reshape(12,1)
tensor([[1.],
        [1.],
        [1.],
        [1.],
        [2.],
        [2.],
        [2.],
        [2.],
        [3.],
        [3.],
        [3.],
        [3.]])
```

Using the `reshape()` function, we can specify the `row x column` shape that we are seeking. Notice how all of the shapes have to account for the number of elements in the tensor. In our example this is:

rows * columns = 12 elements

We can use the intuitive words *rows* and *columns* when

we are dealing with a rank `2` tensor. The underlying logic is the same for higher dimensional tenors even though we may not be able to use the intuition of rows and columns in higher dimensional spaces. For example:

```
> t.reshape(2,2,3)
tensor(
[
    [
        [1., 1., 1.],
        [1., 2., 2.]
    ],

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

In this example, we increase the rank to `3`, and so we lose the *rows and columns* concept. However, the product of the shape's components (`2`,`2`,`3`) still has to be equal to the number of elements in the original tensor (`12`).

Note that PyTorch has another function that you may see called `view()` that does the same thing as the `reshape()` function, but don't let these names through you off. No matter which deep learning framework we are using, these concepts will be the same.

Changing shape by squeezing and unsqueezing

The next way we can change the shape of our tensors is by *squeezing* and *unsqueezing* them.

- *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.

These functions allow us to expand or shrink the rank (number of dimensions) of our tensor. Let's see this in action.

```
> print(t.reshape([1,12]))
> print(t.reshape([1,12]).shape)
tensor([[1., 1., 1., 1., 2., 2., 2., 2.,
3., 3., 3., 3.]])
torch.Size([1, 12])

> print(t.reshape([1,12]).squeeze())
> print(t.reshape([1,12]).squeeze().shape)
tensor([1., 1., 1., 1., 2., 2., 2., 2., 3.,
3., 3., 3.])
torch.Size([12])

>
print(t.reshape([1,12]).squeeze().unsqueeze(dim=
>
print(t.reshape([1,12]).squeeze().unsqueeze(dim=
tensor([[1., 1., 1., 1., 2., 2., 2., 2.,
3., 3., 3., 3.]])
```

```
torch.Size([1, 12])
```

Notice how the shape changes as we squeeze and unsqueeze the tensor.

Let's look at a common use case for squeezing a tensor by building a *flatten* function.

Flatten a tensor

A *flatten* operation on a tensor reshapes the tensor to have a shape that is equal to the number of elements contained in the tensor. This is the same thing as a 1d-array of elements.

Flattening a tensor means to remove all of the dimensions except for one.

Let's create a Python function called `flatten()`:

```
def flatten(t):
    t = t.reshape(1, -1)
    t = t.squeeze()
    return t
```

The `flatten()` function takes in a tensor `t` as an argument.

Since the argument `t` can be any tensor, we pass `-1` as the second argument to the `reshape()` function. In PyTorch, the `-1` tells the `reshape()` function to figure out what the value should be based on the number of elements contained within the tensor. Remember, the shape must equal the product of the shape's component values. This is

how PyTorch can figure out what the value should be, given a `1` as the first argument.

Since our tensor `t` has `12` elements, the `reshape()` function is able to figure out that a `12` is required for the length of the second axis.

After squeezing, the first axis (axis-0) is removed, and we obtain our desired result, a 1d-array of length `12`.

Here's an example of this in action:

```
> t = torch.ones(4, 3)
> t
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])


> flatten(t)
tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1.])
```

In a future post when we begin building a convolutional neural network, we will see the use of this `flatten()` function. We'll see that *flatten* operations are required when passing an output tensor from a convolutional layer to a linear layer.

In these examples, we have flattened the entire tensor, however, it is possible to flatten only specific parts of a tensor. For example, suppose we have a tensor of shape `[2,1,28,28]` for a CNN. This means that we have a

batch of `2` grayscale images with height and width dimensions of `28 x 28`, respectively.

Here, we can specifically flatten the two images. To get the following shape: `[2,1,784]`. We could also squeeze off the channel axes to get the following shape: `[2,784]`.

Concatenating tensors

We combine tensors using the `cat()` function, and the resulting tensor will have a shape that depends on the shape of the two input tensors.

Suppose we have two tensors:

```
> t1 = torch.tensor([
    [1,2],
    [3,4]
])
> t2 = torch.tensor([
    [5,6],
    [7,8]
])
```

We can combine `t1` and `t2` row-wise (axis-0) in the following way:

```
> torch.cat((t1, t2), dim=0)
tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
```

We can combine them column-wise (axis-1) like this:

```
> torch.cat((t1, t2), dim=1)
tensor([[1, 2, 5, 6],
        [3, 4, 7, 8]])
```

When we concatenate tensors, we increase the number of elements contained within the resulting tensor. This causes the component values within the shape (lengths of the axes) to adjust to account for the additional elements.

```
> torch.cat((t1, t2), dim=0).shape
torch.Size([4, 2])

> torch.cat((t1, t2), dim=1).shape
torch.Size([2, 4])
```

Conclusion about reshaping tensors

We should now have a good understanding of what it means to reshape a tensor. Any time we change a tensor's shape, we are said to be *reshaping* the tensor.

Remember the analogy. Bakers work with dough and neural network programmers work with tensors. Even though the concept of shaping is the same, instead of creating baked goods, we are creating intelligence. I'll see you in the next one.