PyTorch from Scrach - 01

- PyTorch is an open source machine learing and deep learning framework
- PyTorch allows you to manipulate and precess data and write machine learning algorithms using Python

Author: Ramdas Dharavath

Linkedin link: https://www.linkedin.com/in/dharavath-ramdas/
GitHub link: https://github.com/dharavathramdas101

Show code

Importing PyTorch and Checking the version

```
import torch
print(torch.__version__)
```

```
2.0.1+cu118
```

Tensors

> creating PyTorch tensors using torch.Tensor()

Scalar

> Scalar is a single number and in tensor - speak it's a zero dimention tensor

```
scalar = torch.tensor(11)
print(scalar)
```

```
tensor(11)
```

ndim

> check the dimentions of a tensor using the ndim attribute

```
scalar.ndim
```

```
0
```

item()

> what if we wanted to retrieve the number from the tensor? as in, turn it from torch.tensor to a python integer? to do we can use the item() method

```
# get the python number within a tensor (only works with one - element tensor)
scalar.item()
```

```
11
```

Vector

> A vector is a single dimention tensor but can contain many numbers.
> example [55,66]

```
# vector
vector = torch.tensor([11,11])
print(vector)
```

```
tensor([11, 11])
```

```
# Check the number of dimentions of vector
vector.ndim
```

```
1
```

Hmm, that's strange, vector contains two numbers but only has a single dimension.

I'll let you in on a trick.

You can tell the number of dimensions a tensor in PyTorch has by the number of square brackets on the outside ([) and you only need to count one side.

How many square brackets does vector have?

Shape

Another important concept for tensors is their shape attribute. The shape tells you how the elements inside them are arranged.

Let's check out the shape of vector.

```
# Check the shape of vector
vector.shape
```

```
torch.Size([2])
```

The above returns torch.Size([2]) which means our vector has a shape of [2]. This is because of the two elements we placed inside the square brackets ([1,1]).

Matrix

> MATRIX has two dimentions

```
# matrix
MATRIX = torch.tensor([[11,22],
                       [99,33]])
MATRIX
```

```
tensor([[11, 22],
        [99, 33]])
```

```
# Check the number of dimentions
MATRIX.ndim
```

```
2
```

```
# Check the shape
MATRIX.shape
```

```
torch.Size([2, 2])
```

We get the output torch.Size([2, 2]) because MATRIX is two elements deep and two elements wide.

Tensor

```
# tensor
TENSOR =  torch.tensor([[[1,4,6],
                         [4,3,2],
                         [0,44,11]]])
print(TENSOR)
```

```
tensor([[[ 1,  4,  6],
         [ 4,  3,  2],
         [ 0, 44, 11]]])
```

```
# Check number of dimentions for TENSOR
TENSOR.ndim
```

```
3
```

```
# Check shape of TENSOR
TENSOR.shape
```

```
torch.Size([1, 3, 3])
```

Alright, it outputs torch.Size([1, 3, 3]).

The dimensions go outer to inner.

That means there's 1 dimension of 3 by 3.

Random tensors

> Randon tensors are important because the way many neural networks learn is that they start with tensors full of random numbers and then adjust those random numbers to better represent the data.

Start with random numbers -> look at data -> update random numbers -> look at data -> update random numbers

using torch.rand() and passing in the size parameter

```
# Creating a random tensor of size (4,3)
random_tensor = torch.rand(size=(4,3))
print(random_tensor, random_tensor.dtype)
```

```
tensor([[0.7765, 0.4320, 0.4077],
        [0.5842, 0.8199, 0.1312],
        [0.2427, 0.6836, 0.3346],
        [0.5088, 0.9050, 0.3494]]) torch.float32
```

1. The flexibility of torch.rand() is that we can adjust the size to be whatever we want.
2. For example, say you wanted a random tensor in the common image shape of [224, 224, 3] ([height, width, color_channels]).

```
# Create a random tensor of size (224,224, 3)
random_image_size_tensor = torch.rand(size=(224,224,3))
random_image_size_tensor.shape, random_image_size_tensor.ndim
```

```
(torch.Size([224, 224, 3]), 3)
```

Zeros & Ones

- sometimes you will just want to fill tensors with zeros or ones
- this happens a lot with masking (like masking some of the values in one tensor with zeros to let a model know not to learn them).
- let's create a tensor full of zeros with torch.zeros()
- again the size parameter into play

```
# Create a tensor of all zeros
zeros =  torch.zeros(size=(4,4))
zeros, zeros.shape
```

```
(tensor([[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]),
 torch.Size([4, 4]))
```

```
# Checking the shape
zeros.shape
```

```
torch.Size([4, 4])
```

```
# Checking the dimention
zeros.ndim
```

```
2
```

```
# Checking the data type
zeros.dtype
```

```
torch.float32
```

torch.ones()

- we can do the same to create a tensor of all ones except using torch.ones() insted.

```
# Creating a tensor of all ones
ones = torch.ones(size=(4,4))
ones
```

```
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
```

```
# Checking the shape
ones.shape
```

```
torch.Size([4, 4])
```

```
# Checking the datatype
ones.dtype
```

```
torch.float32
```

```
# Checking the dimention
ones.ndim
```

```
2
```

Range & Tensors like

- sometimes you might want a range of numbers, such as 1 to 10 or 0 to 100.
- you can use torch.arange(start, end, step) to do so

where:

- start = start of range(0)
- end = end of range(100)
- step = how many steps in between each value (e.g 10)

```
# Use torch.arange()
zero_to_ten = torch.arange(start=0, end=10,step=1)
zero_to_ten
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- Sometimes you might want one tensor of a certain type with the same shape as another tensor.
- For example, a tensor of all zeros with the same shape as a previous tensor.
- To do so you can use torch.zeros_like(input) or torch.ones_like(input) which return a tensor filled with zeros or ones in the same shape as the input respectively.

```
# Can also create a tensor of zeros similar to another tensor
ten_zeros = torch.zeros_like(input=zero_to_ten)
# Will have the same shape
ten_zeros
```

```
tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
# Can also create a tensor of ones similar to another tensor
ten_ones = torch.ones_like(input=zero_to_ten)
ten_ones
```

```
tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

Tensor datatypes

- 32-bit floating point -> torch.float32 or torch.float
- 64-bit floating point -> torch.float64 or torch.double
- 16-bit floating point [1] -> torch.float16 or torch.half
- 16-bit floating point [2] -> torch.bfloat16
- 32-bit complex -> torch.complex32 or torch.chalf
- 64-bit complex -> torch.complex64
- 128-bit complex -> torch.complex128
- 8-bit integer (unsigned) -> torch.uint8
- 8-bit integer (signed) -> torch.int8
- 16-bit integer (signed) -> torch.int16 or torch.short
- 32-bit integer (signed) -> torch.int32
- 64-bit integer (signed) -> torch.int64 or torch.long
- Boolean -> torch.bool
- Let's see how to create some tensors with specific datatypes. We can do so using the dtype parameter.

Note : Tensor datatypes is one of the 3 big errors you'll run into with PyTorch & deep learning:

Tensors not right datatype
Tensors not right shape
Tensors not on the right device

```
# default datatype for tensors is float32
float32_tensor = torch.tensor([3.0,6.0,4.0],
                              dtype = None, #default to None, which is torch.float32 or what ever datatype is passed
                              device= None, #What device is your tensor on, default to None, which uses the default tenosr type
                              requires_grad=False) # whether or not to track gradients with this tensor operation)
```

```
float32_tensor.shape, float32_tensor.dtype, float32_tensor.device
```

```
(torch.Size([3]), torch.float32, device(type='cpu'))
```

- Aside from shape issues (tensor shapes don't match up), two of the other most common issues you'll come across in PyTorch are datatype and device issues.
- For example, one of tensors is torch.float32 and the other is torch.float16 (PyTorch often likes tensors to be the same format).
- Or one of your tensors is on the CPU and the other is on the GPU (PyTorch likes calculations between tensors to be on the same device).
- let's create a tensor with dtype=torch.float16

- Changing the datatype from float32 to float16

```
float16_tensor2 = float32_tensor.type(torch.float16)
float16_tensor2
```

```
tensor([3., 6., 4.], dtype=torch.float16)
```

```
float16_tensor = torch.tensor([5.0,4.9,9.9],
                              dtype = torch.float16)
```

```
float16_tensor
```

```
tensor([5.0000, 4.8984, 9.8984], dtype=torch.float16)
```

```
float16_tensor * float16_tensor2
```

```
tensor([15.0000, 29.3906, 39.5938], dtype=torch.float16)
```

```
int16_tensor = torch.tensor([4,5,6,7], dtype=torch.int16)
int16_tensor
```

```
tensor([4, 5, 6, 7], dtype=torch.int16)
```

```
int32_tensor = torch.tensor([55,6,2,4], dtype=torch.int32)
int32_tensor
```

```
tensor([55,  6,  2,  4], dtype=torch.int32)
```

```
int16_tensor * int32_tensor
```

```
tensor([220,  30,  12,  28], dtype=torch.int32)
```

```
int16_tensor2 = int32_tensor.type(torch.int16)
int16_tensor2
```

```
tensor([55,  6,  2,  4], dtype=torch.int16)
```

Getting information from tensors (tensor attributes)

1. dtype - Tensor not right datatype - to do get datatype from a tensor , can use tensor.dtype
2. shape - Tensor not right shape - to get shape from a tensor, can use tensor.shape
3. device - Tensors not on the right device - to get device from a tensor, can use tensor.device

```
# Create a tensor
some_tensor = torch.rand(3,4)

# find out details about it
print(some_tensor)
print(f"Shape of tensor: {some_tensor.shape}")
print(f"Datatype of tensor: {some_tensor.dtype}")
print(f"Device tensor is stored on: {some_tensor.device}")
```

```
tensor([[0.5332, 0.8086, 0.3213, 0.5151],
        [0.1653, 0.3205, 0.0821, 0.4948],
        [0.4713, 0.6927, 0.9475, 0.6035]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Note: When you run into issues in PyTorch, it's very often one to do with one of the three attributes above. So when the error messages show up, sing yourself a little song called "what, what, where":

"what shape are my tensors? what datatype are they and where are they stored? what shape, what datatype, where where where"

Manipulating tensors (tensor operations)

In deep learning, data (images, text, video, audio, protein structures, etc) gets represented as tensors.

A model learns by investigating those tensors and performing a series of operations (could be 1,000,000s+) on tensors to create a representation of the patterns in the input data.

These operations are often a wonderful dance between:

- Addition
- Substraction
- Multiplication (element-wise)
- Division
- Matrix multiplication

  And that's it. Sure there are a few more here and there but these are the basic building blocks of neural networks.

Stacking these building blocks in the right way, you can create the most sophisticated of neural networks (just like lego!).

Basic operations

lets start with few of the fundamental operations, addition(+), subtraction(-), multiplication(*)

```
# create a tensor of values and add a number to it
tensor =  torch.tensor([1,2,3])
tensor + 100
```

```
tensor([101, 102, 103])
```

```
# Multiply it by 10
tensor * 20
```

```
tensor([20, 40, 60])
```

Notice how the tensor values above didn't end up being tensor([110, 102, 103]), this is because the values inside the tensor don't change unless they're reassigned.

```
# Tensors don't change unless reassigned
tensor
```

```
tensor([1, 2, 3])
```

let's substract a number and this time we will reassign the tensor variable

```
# Subtracted and reassign
tensor = tensor - 10
tensor
```

```
tensor([-9, -8, -7])
```

```
# Add and reassign
tensor = tensor + 10
tensor
```

```
tensor([1, 2, 3])
```

pytorch also has a bunch of built in functions like torch.mul() and torch.add()

```
# Try out PyTorch in-built functions
torch.mul(tensor,22)
```

```
tensor([22, 44, 66])
```

```
torch.add(tensor, 33)
```

```
tensor([34, 35, 36])
```

However it's more common to use the operator symbols like * instead of torch.mul()

```
# Element wise multiplication  (each element multiplies its equivalent, index 0->0, 1->1, 2->2, 3->3)
print(tensor, "*", tensor)
print("Equals: ", tensor * tensor)
```

```
tensor([1, 2, 3]) * tensor([1, 2, 3])
Equals:  tensor([1, 4, 9])
```

Matrix multiplication (is all you need)

One of the most common operations in machine learning and deep learning algorithms (like neural networks) is matrix multiplication.

PyTorch implements matrix multiplication functionality in the torch.matmul() method.

The main two rules for matrix multiplication to remember are:

1. The inner dimensions must match:

- (3, 2) @ (3, 2) won't work
- (2, 3) @ (3, 2) will work
- (3, 2) @ (2, 3) will work

2. The resulting matrix has the shape of the outer dimensions:

- (2, 3) @ (3, 2) -> (2, 2)
- (3, 2) @ (2, 3) -> (3, 3)

Note: "@" in Python is the symbol for matrix multiplication.

Resource: You can see all of the rules for matrix multiplication using torch.matmul() in the PyTorch documentation.

Let's create a tensor and perform element-wise multiplication and matrix multiplication on it.

```
import torch
tensor = torch.tensor([3,4,5])
tensor.shape
```

```
torch.Size([3])
```

The difference between element-wise multiplication and matrix multiplication is the addition of values.

For our tensor variable with values [1, 2, 3]:

Operation Calculation Code

- Element-wise multiplication [1*1, 2*2, 3*3] = [1, 4, 9] tensor * tensor
- Matrix multiplication [1*1 + 2*2 + 3*3] = [14] tensor.matmul(tensor)

```
# Element -wise matrix multiplication
print(tensor)
tensor * tensor
```

```
tensor([3, 4, 5])
tensor([ 9, 16, 25])
```

```
# Matrix multiplication
torch.matmul(tensor,tensor)
```

```
tensor(50)
```

```
# Can also use the "@" symbol for matrix multiplication, through not recommended
tensor @ tensor
```

```
tensor(50)
```

you can do matrix multiplication by hand but it's not recommended
the in-built torch.matmul() method is faster

```
# matrix multiplication by hand
9*9+8*8+4*4
```

```
161
```

```
%%time
# matrix multiplication by hand
# avoid doing operations with for loops at all cost, they are computationally expensive
value = 0
for i in range(len(tensor)):
  value += tensor[i] * tensor[i]
value
```

```
CPU times: user 1.1 ms, sys: 0 ns, total: 1.1 ms
Wall time: 1.2 ms
tensor(50)
```

```
%%time
torch.matmul(tensor,tensor)
```

```
CPU times: user 868 µs, sys: 0 ns, total: 868 µs
Wall time: 875 µs
tensor(50)
```

One of the most common erros in deep learning: shape erros

Because much of deep learning is multiplying and performing operations on matrices and matrices have a strict rule about what shapes and sizes can be combined, one of the most common errors you'll run into in deep learning is shape mismatches.

```
# Shape need to be in the right way
tensor_R =torch.tensor([[3,4],
                        [9,8],
                        [3,2]], dtype=torch.float32)
tensor_M = torch.tensor([[1,2],
                         [4,5],
                         [7,8]], dtype= torch.float32)
torch.matmul(tensor_R,tensor_M)  # this will error
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-69-594505892980> in <cell line: 8>()
      6                         [4,5],
      7                         [7,8]], dtype= torch.float32)
----> 8 torch.matmul(tensor_R,tensor_M)  # this will error

RuntimeError: mat1 and mat2 shapes cannot be multiplied (3x2 and 3x2)
```

SEARCH STACK OVERFLOW

We can make matrix multiplication work between tensor_A and tensor_B by making their inner dimensions match.

One of the ways to do this is with a transpose (switch the dimensions of a given tensor).

You can perform transposes in PyTorch using either:

torch.transpose(input, dim0, dim1) - where input is the desired tensor to transpose and dim0 and dim1 are the dimensions to be swapped.
tensor.T - where tensor is the desired tensor to transpose.

```
# View tensor_R and tensor_M
print(tensor_R)
print(tensor_M)
```

```
tensor([[3., 4.],
        [9., 8.],
        [3., 2.]])
tensor([[1., 2.],
        [4., 5.],
        [7., 8.]])
```

```
# View tensor_R and tensor_M.T
print(tensor_R)
print(tensor_M.T)
```

```
tensor([[3., 4.],
        [9., 8.],
        [3., 2.]])
tensor([[1., 4., 7.],
        [2., 5., 8.]])
```

```
# The operation works when tensor_B is transposed

print(f"Original shapes: tensor_A = {tensor_R.shape}, tensor_B = {tensor_M.shape}\n")
print(f"New shapes: tensor_A = {tensor_R.shape} (same as above), tensor_B.T = {tensor_M.T.shape}\n")
print(f"Multiplying: {tensor_R.shape} * {tensor_M.T.shape} <- inner dimensions match\n")
print("Output:\n")
output = torch.matmul(tensor_R, tensor_M.T)
print(output)
print(f"\nOutput shape: {output.shape}")
```

```
Original shapes: tensor_A = torch.Size([3, 2]), tensor_B = torch.Size([3, 2])

New shapes: tensor_A = torch.Size([3, 2]) (same as above), tensor_B.T = torch.Size([2, 3])

Multiplying: torch.Size([3, 2]) * torch.Size([2, 3]) <- inner dimensions match

Output:

tensor([[ 11.,  32.,  53.],
        [ 25.,  76., 127.],
        [  7.,  22.,  37.]])

Output shape: torch.Size([3, 3])
```

we can also use torch.mm() whic is a short for torch.matmul(_)

```
# torch.mm() is a shortcut for matmul
torch.mm(tensor_R, tensor_M.T)
```

```
tensor([[ 11.,  32.,  53.],
        [ 25.,  76., 127.],
        [  7.,  22.,  37.]])
```

Without the transpose, the rules of matrix mulitplication aren't fulfilled and we get an error like above.

> Note: A matrix multiplication like this is also referred to as the dot product of two matrice

Neural networks are full of matrix multiplications and dot products.

The torch.nn.Linear() module (we'll see this in action later on), also known as a feed-forward layer or fully connected layer, implements a matrix multiplication between an input x and a weights matrix A.
y = x . R.T + b

Where:

- x is the input to the layer (deep learning is a stack of layers like torch.nn.Linear() and others on top of each other).
- A is the weights matrix created by the layer, this starts out as random numbers that get adjusted as a neural network learns to better represent patterns in the data (notice the "T", that's because the weights matrix gets transposed).
- Note: You might also often see W or another letter like X used to showcase the weights matrix.
- b is the bias term used to slightly offset the weights and inputs.
- y is the output (a manipulation of the input in the hopes to discover patterns in it).
- This is a linear function (you may have seen something like
  y = mx + b

  in high school or elsewhere), and can be used to draw a straight line!

Let's play around with a linear layer.
ry changing the values of in_features and out_features below and see what happens.

Do you notice anything to do with the shapes?

```
tensor_R
```

```
tensor([[3., 4.],
        [9., 8.],
        [3., 2.]])
```

```
# Since the linear layer starts with a random weights matrix, let's make it reproducible
torch.manual_seed(42)
# this uses matrix multiplication
linear = torch.nn.Linear(in_features=2,out_features=6) # in_features = matches inner dimention of input
# out_feature = describes outer value
x = tensor_R
output = linear(x)
print(f"Input shape: {x.shape}\n")
print(f"Output:\n{output}\n\noutput shape: {output.shape}")
```

```
Input shape: torch.Size([3, 2])

Output:
tensor([[ 4.4919,  2.1970,  0.4469,  0.5285,  0.3401,  2.4777],
        [10.0831,  3.8013,  0.0881,  0.1240,  2.0052,  6.6948],
        [ 3.3181,  0.8979,  0.1615, -0.3021,  1.3776,  2.2130]],
       grad_fn=<AddmmBackward0>)

output shape: torch.Size([3, 6])
```

Question: What happens if you change in_features from 2 to 3 above? Does it error? How could you change the shape of the input (x) to accomodate to the error? Hint: what did we have to do to tensor_B above?

If you've never done it before, matrix multiplication can be a confusing topic at first.

But after you've played around with it a few times and even cracked open a few neural networks, you'll notice it's everywhere.

Remember, matrix multiplication is all you need.
When you start digging into neural network layers and building your own, you'll find matrix multiplications everywhere.

Finding the min, max, mean, sum, etc (tensor aggregation)

- now we seen a few ways to manipulate tensors, let's run through a few ways to aggregate them (go from more values to less values)
- first we will create a tensor and then find the max, min, mean and sum of it.

```
# create a tensor
x = torch.arange(0,200,20)
x
```

```
tensor([  0,  20,  40,  60,  80, 100, 120, 140, 160, 180])
```

let's perform some aggregation

```
print(f"Minimum: {x.min()}")
print(f"Maximum: {x.max()}")
# print(f"Mean: {x,mean()}") this will give error
print(f"Mean: {x.type(torch.float32).mean()}")
print(f"Sum: {x.sum()}")
```

```
Minimum: 0
Maximum: 180
Mean: 90.0
Sum: 900
```

> Note: You may find some methods such as torch.mean() require tensors to be in torch.float32 (the most common) or another specific datatype, otherwise the operation will fail.

```
# we can also dothe same as above with torch methods
torch.max(x), torch.min(x), torch.mean(x.type(torch.float32)), torch.sum(x)
```

```
(tensor(180), tensor(0), tensor(90.), tensor(900))
```

```
# find the min
torch.min(x), x.min()
```

```
(tensor(0), tensor(0))
```

```
# find the max
torch.max(x), x.max()
```

```
(tensor(180), tensor(180))
```

```
# find the mean
torch.mean(x.type(torch.float32)), x.type(torch.float32).mean()
```

```
(tensor(90.), tensor(90.))
```

```
# Find sum
torch.sum(x), x.sum()
```

```
(tensor(900), tensor(900))
```

Finding the positional min and max

You can also find the index of a tensor where the max or minimum occurs with torch.argmax() and torch.argmin() respectively.

This is helpful incase you just want the position where the highest (or lowest) value is and not the actual value itself

```
# create a tensor
tensor = torch.arange(20,200,20)
print(f"Tensor: {tensor}")

# Returns index of max and min values
print(f"Index where max values occurs: {tensor.argmax()}")
print(f"Index where min vlaue occurs: {tensor.argmin()}")
```

```
Tensor: tensor([ 20,  40,  60,  80, 100, 120, 140, 160, 180])
Index where max values occurs: 8
Index where min vlaue occurs: 0
```

```
tensor[8]
```

```
tensor(180.)
```

```
tensor[0]
```

```
tensor(20.)
```

Change tensor datatype

As mentioned, a common issue with deep learning operations is having your tensors in different datatypes.

If one tensor is in torch.float64 and another is in torch.float32, you might run into some errors.

But there's a fix.

You can change the datatypes of tensors using torch.Tensor.type(dtype=None) where the dtype parameter is the datatype you'd like to use.

First we'll create a tensor and check it's datatype (the default is torch.float32).

```
# Create a tensor and check its datatype
tensor = torch.arange(20.,200.,20.)
tensor.dtype
```

```
torch.float32
```

```
# Create a float16 tensor
tensor_float16 = tensor.type(torch.float16)
tensor_float16
```

```
tensor([ 20.,  40.,  60.,  80., 100., 120., 140., 160., 180.],
       dtype=torch.float16)
```

```
# Create a int8 tensor
tensor_int8 = tensor.type(torch.int8)
tensor_int8
```

```
tensor([  20,   40,   60,   80,  100,  120, -116,  -96,  -76],
       dtype=torch.int8)
```

> Note: Different datatypes can be confusing to begin with. But think of it like this, the lower the number (e.g. 32, 16, 8), the less precise a computer stores the value. And with a lower amount of storage, this generally results in faster computation and a smaller overall model. Mobile-based neural networks often operate with 8-bit integers, smaller and faster to run but less accurate than their float32 counterparts.

Reshaping, stacking, squeezing and unsqueezing

you'll want to reshape or change the dimensions of your tensors without actually changing the values inside them.

- Reshaping - reshapes an input tensor to a defined shape
- View - Return a view of an input tensor of certain shape but keep the same memory as the original tensor
- Stacking - combine multiple tensors on top of each other (vstack) or side by side (hstack)
- Squeeze - removes all 1 dimensions from a tensor
- Unsqueeze - add a 1 dimension to a target tensor
- Permute - Return a view of the input with dimensions permuted (swapped) in a certain way

To do so, some popular methods are:
Method One-line description

- torch.reshape(input, shape) Reshapes input to shape (if compatible), can also use torch.Tensor.reshape().
- Tensor.view(shape) Returns a view of the original tensor in a different shape but shares the same data as the original tensor.
- torch.stack(tensors, dim=0) Concatenates a sequence of tensors along a new dimension (dim), all tensors must be same size.
- torch.squeeze(input) Squeezes input to remove all the dimenions with value 1.
  torch.unsqueeze(input, dim) Returns input with a dimension value of 1 added at dim.
- torch.permute(input, dims) Returns a view of the original input with its dimensions permuted (rearranged) to dims.
  Why do any of these?

Because deep learning models (neural networks) are all about manipulating tensors in some way. And because of the rules of matrix multiplication, if you've got shape mismatches, you'll run into errors. These methods help you make sure the right elements of your tensors are mixing with the right elements of other tensors.

```
# Create a tensor
import torch
x = torch.arange(2.,9.)
x, x.shape
```

```
(tensor([2., 3., 4., 5., 6., 7., 8.]), torch.Size([7]))
```

```
# Now let's add extra dimention with torch.reshape()
# Add an extra dimention
x_reshaped = x.reshape(1,7)
x_reshaped, x_reshaped.shape
```

```
(tensor([[2., 3., 4., 5., 6., 7., 8.]]), torch.Size([1, 7]))
```

```
# Now let's add extra dimention with torch.reshape()
# Add an extra dimention
x_reshaped = x.reshape(7,1)
x_reshaped, x_reshaped.shape
```

```
(tensor([[2.],
         [3.],
         [4.],
         [5.],
         [6.],
         [7.],
         [8.]]),
 torch.Size([7, 1]))
```

```
# Now let's change the view with torch.view()
# Change the view (keeps same data as original but changes view)
y = x.view(1,7)
y, y.shape
```

```
(tensor([[2., 3., 4., 5., 6., 7., 8.]]), torch.Size([1, 7]))
```

Remember though, changing the view of a tensor with torch.view() really only creates a new view of the same tensor.

So changing the view changes the original tensor too.

```
# Changing y changes x
y[:,0] = 5
y, x
```

```
(tensor([[5., 3., 4., 5., 6., 7., 8.]]), tensor([5., 3., 4., 5., 6., 7., 8.]))
```

```
#stack our new tensor on top of itself five times, we could do so with torch.stack()
# Stack tensors on top of each other
x_stacked = torch.stack([x,x,x,x], dim=0)
x_stacked
```

```
tensor([[5., 3., 4., 5., 6., 7., 8.],
        [5., 3., 4., 5., 6., 7., 8.],
        [5., 3., 4., 5., 6., 7., 8.],
        [5., 3., 4., 5., 6., 7., 8.]])
```

```
x_stacked = torch.stack([x,x,x,x], dim=1)
x_stacked
```

```
tensor([[5., 5., 5., 5.],
        [3., 3., 3., 3.],
        [4., 4., 4., 4.],
        [5., 5., 5., 5.],
        [6., 6., 6., 6.],
        [7., 7., 7., 7.],
        [8., 8., 8., 8.]])
```

```
# you can use torch.squeeze() (I remember this as squeezing the tensor to only have dimensions over 1).
```

```
print(f"Previous tensor: {x_reshaped}")
print(f"Previous shape: {x_reshaped.shape}")

# Remove extra dimentions from x_reshaped
x_squeezed = x_reshaped.squeeze()
print(f"\nNew tensor: {x_squeezed}")
print(f"New shape: {x_squeezed.shape}")
```

```
Previous tensor: tensor([[5.],
        [3.],
        [4.],
        [5.],
        [6.],
        [7.],
        [8.]])
Previous shape: torch.Size([7, 1])

New tensor: tensor([5., 3., 4., 5., 6., 7., 8.])
New shape: torch.Size([7])
```

```
# And to do the reverse of torch.squeeze() you can use torch.unsqueeze() to add a dimension value of 1 at a specific index.
print(f"Previous tensor: {x_squeezed}")
print(f"Previous shape: {x_squeezed.shape}")

## Add an extra dimension with unsqueeze
x_unsqueezed = x_squeezed.unsqueeze(dim=0)
print(f"\nNew tensor: {x_unsqueezed}")
print(f"New shape: {x_unsqueezed.shape}")
```

```
Previous tensor: tensor([5., 3., 4., 5., 6., 7., 8.])
Previous shape: torch.Size([7])

New tensor: tensor([[5., 3., 4., 5., 6., 7., 8.]])
New shape: torch.Size([1, 7])
```

You can also rearrange the order of axes values with torch.permute(input, dims), where the input gets turned into a view with new dims.

```
# Create tensor with specific shape
x_original = torch.rand(size=(224,224,3))

# Permute the original tensor to rearrange the axis order
x_permuted = x_original.permute(2,0,1)

print(f"Previous shape: {x_original.shape}")
print(f"New shape: {x_permuted.shape}")
```

```
Previous shape: torch.Size([224, 224, 3])
New shape: torch.Size([3, 224, 224])
```

```
# Create tensor with specific shape
x_original = torch.rand(size=(224,224,3))

# Permute the original tensor to rearrange the axis order
x_permuted = x_original.permute(2,1,0)

print(f"Previous shape: {x_original.shape}")
print(f"New shape: {x_permuted.shape}")
```

```
Previous shape: torch.Size([224, 224, 3])
New shape: torch.Size([3, 224, 224])
```

```
# Create tensor with specific shape
x_original = torch.rand(size=(224,224,3))

# Permute the original tensor to rearrange the axis order
x_permuted = x_original.permute(0,2,1)

print(f"Previous shape: {x_original.shape}")
print(f"New shape: {x_permuted.shape}")
```

> Note: Because permuting returns a view (shares the same data as the original), the values in the permuted tensor will be the same as the original tensor and if you change the values in the view, it will change the values of the original.

Indexing (selecting data from tensors)

Sometimes you'll want to select specific data from tensors (for example, only the first column or second row).

To do so, you can use indexing

```
#  Create a tensor
import torch
x = torch.arange(2,11).reshape(1,3,3)
x, x.shape
```

```
(tensor([[[ 2,  3,  4],
          [ 5,  6,  7],
          [ 8,  9, 10]]]),
 torch.Size([1, 3, 3]))
```

ndexing values goes outer dimension -> inner dimension (check out the square brackets).

```
# let's index bracket by bracket
print(f"First square bracket: \n {x[0]}")
print(f"Second square bracket: {x[0][0]}")
print(f"Third square bracke: {x[0][0][0]}")
```

```
First square bracket:
 tensor([[ 2,  3,  4],
        [ 5,  6,  7],
        [ 8,  9, 10]])
Second square bracket: tensor([2, 3, 4])
Third square bracke: 2
```

you can also use : to specify "all values in this dimentions" and then use a comma(,) to add another dimention.

```
# get all values of 0th dimention and the 0 index of 1st dimension
x[:,0]
```

```
tensor([[2, 3, 4]])
```

```
# get all vlues of oth and 1th dimentions but only index 1 of 2nd dimension\
x[:,:,1]
```

```
tensor([[3, 6, 9]])
```

```
# Get all values of the o dimention but only the 1 index values of the 1st and 2nd dimention
x[:,1,1]
```

```
tensor([6])
```

```
# Get index 0 of 0th and 1st dimention and alll values of 2nd dimention
x[0,0,:]
```

```
tensor([2, 3, 4])
```

**Pytorch tensors & Numpy

- since Numpy is a popular Python numerical computing library, PyTorch has funcionality to interact with it nicely The two main methods you'll want to use for NumPy to PyTorch (and back again) are:
- torch.from_numpy(ndarray) - NumPy array -> PyTorch tensor.
- torch.Tensor.numpy() - PyTorch tensor -> NumPy array.

```
# Numpy array to tensor
import torch
import numpy as np
array = np.arange(1.0,8.0)
tensor = torch.from_numpy(array)
array, tensor
```

```
(array([1., 2., 3., 4., 5., 6., 7.]),
 tensor([1., 2., 3., 4., 5., 6., 7.], dtype=torch.float64))
```

Note: By default, NumPy arrays are created with the datatype float64 and if you convert it to a PyTorch tensor, it'll keep the same datatype (as above).

However, many PyTorch calculations default to using float32.

So if you want to convert your NumPy array (float64) -> PyTorch tensor (float64) -> PyTorch tensor (float32), you can use tensor = torch.from_numpy(array).type(torch.float32).

Because we reassigned tensor above, if you change the tensor, the array stays the same.

```
# Change the array, keep the tensor
array = array + 1
array, tensor
```

```
(array([2., 3., 4., 5., 6., 7., 8.]),
 tensor([1., 2., 3., 4., 5., 6., 7.], dtype=torch.float64))
```

```
# Tensor to Numpy array
tensor = torch.ones(7)
numpy_tensor = tensor.numpy()
tensor, numpy_tensor
```

```
(tensor([1., 1., 1., 1., 1., 1., 1.]),
 array([1., 1., 1., 1., 1., 1., 1.], dtype=float32))
```

```
# Change the tensor, keep the array the same
tensor = tensor + 1
tensor, numpy_tensor
```

```
(tensor([2., 2., 2., 2., 2., 2., 2.]),
 array([1., 1., 1., 1., 1., 1., 1.], dtype=float32))
```

Reproducibility (trying to take the random out of random)

As you learn more about neural networks and machine learning, you'll start to discover how much randomness plays a part.

Well, pseudorandomness that is. Because after all, as they're designed, a computer is fundamentally deterministic (each step is predictable) so the randomness they create are simulated randomness (though there is debate on this too, but since I'm not a computer scientist, I'll let you find out more yourself).

How does this relate to neural networks and deep learning then?

We've discussed neural networks start with random numbers to describe patterns in data (these numbers are poor descriptions) and try to improve those random numbers using tensor operations (and a few other things we haven't discussed yet) to better describe patterns in data.

In short:

start with random numbers -> tensor operations -> try to make better (again and again and again)

Although randomness is nice and powerful, sometimes you'd like there to be a little less randomness.

Why?

So you can perform repeatable experiments.

For example, you create an algorithm capable of achieving X performance.

And then your friend tries it out to verify you're not crazy.

How could they do such a thing?

That's where reproducibility comes in. In other words, can you get the same (or very similar) results on your computer running the same code as I get on mine?

Let's see a brief example of reproducibility in PyTorch.

We'll start by creating two random tensors, since they're random, you'd expect them to be different right?

```
import torch

# Create two random tensors
random_tensor_A = torch.rand(3,4)
random_tensor_B = torch.rand(3,4)

print(f"Tensor A:\n{random_tensor_A}\n")
print(f"Tensor B:\n{random_tensor_B}\n")
print(f"Does Tensor A equal Tensor B? (anywhere)")
random_tensor_A == random_tensor_B
```

```
Tensor A:
tensor([[0.8660, 0.2001, 0.7926, 0.1140],
        [0.6855, 0.0086, 0.3588, 0.9211],
        [0.8627, 0.8223, 0.5248, 0.5881]])

Tensor B:
tensor([[0.2746, 0.3791, 0.2975, 0.6450],
        [0.0163, 0.9160, 0.5888, 0.8403],
        [0.4761, 0.7140, 0.4644, 0.6362]])

Does Tensor A equal Tensor B? (anywhere)
```

```
tensor([[False, False, False, False],
        [False, False, False, False],
        [False, False, False, False]])
```

ust as you might've expected, the tensors come out with different values.

But what if you wanted to created two random tensors with the same values.

As in, the tensors would still contain random values but they would be of the same flavour.

That's where torch.manual_seed(seed) comes in, where seed is an integer (like 42 but it could be anything) that flavours the randomness.

Let's try it out by creating some more flavoured random tensors.

import torch

```
import torch
import random

# # Set the random seed
RANDOM_SEED=42 # try changing this to different values and see what happens to the numbers below
torch.manual_seed(seed=RANDOM_SEED)
random_tensor_C = torch.rand(3, 4)

# Have to reset the seed every time a new rand() is called
# Without this, tensor_D would be different to tensor_C
torch.random.manual_seed(seed=RANDOM_SEED) # try commenting this line out and seeing what happens
random_tensor_D = torch.rand(3, 4)

print(f"Tensor C:\n{random_tensor_C}\n")
print(f"Tensor D:\n{random_tensor_D}\n")
print(f"Does Tensor C equal Tensor D? (anywhere)")
random_tensor_C == random_tensor_D
```

```
Tensor C:
tensor([[0.8823, 0.9150, 0.3829, 0.9593],
        [0.3904, 0.6009, 0.2566, 0.7936],
        [0.9408, 0.1332, 0.9346, 0.5936]])

Tensor D:
tensor([[0.8823, 0.9150, 0.3829, 0.9593],
        [0.3904, 0.6009, 0.2566, 0.7936],
        [0.9408, 0.1332, 0.9346, 0.5936]])

Does Tensor C equal Tensor D? (anywhere)
tensor([[True, True, True, True],
        [True, True, True, True],
        [True, True, True, True]])
```

✓ 0s completed at 12:08 PM