#Mengimpor PyTorch Mengimpor PyTorch dan memeriksa versi yang digunakan.

```
import torch
torch.__version__
```

```
'1.13.1+cu116'
```

#Membuat Tensor Skalar adalah bilangan tunggal dan dalam istilah tensor merupakan tensor berdimensi nol.

```
# Scalar
scalar = torch.tensor(7)
scalar
```

```
tensor(7)
```

Kita dapat memeriksa dimensi tensor menggunakan atribut ndim.

```
scalar.ndim
```

```
0
```

Menampilkan nomor dari tensor

```
# Get the Python number within a tensor (only works with one-element tensors)
scalar.item()
```

```
7
```

Vektor adalah tensor berdimensi tunggal tetapi dapat memuat banyak bilangan.

Misalnya, kita dapat memiliki vektor [3, 2] untuk mendeskripsikan [kamar tidur, kamar mandi] di rumah. Atau kita dapat menggunakan [3, 2, 2] untuk mendeskripsikan [kamar tidur, kamar mandi, tempat parkir mobil] di rumah.

```
# Vector
vector = torch.tensor([7, 7])
vector
```

```
tensor([7, 7])
```

Menampilkan jumlah vektor dari kode di atas yaitu 1. Kita dapat mengetahui jumlah dimensi yang dimiliki tensor di PyTorch dengan jumlah tanda kurung siku di luar ([) dan Anda hanya perlu menghitung satu sisinya.

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

```
1
```

Memeriksa bentuk vektor

```
# Check shape of vector
vector.shape

torch.Size([2])
```

Di atas menampilkan torch.Size([2]) yang berarti vektor kita berbentuk [2]. Ini karena dua elemen yang kita tempatkan di dalam tanda kurung siku ([7, 7]). Sekarang mari kita lihat matriksnya.

```
# Matrix
MATRIX = torch.tensor([[7, 8],
                       [9, 10]])
MATRIX

tensor([[ 7,  8],
        [ 9, 10]])
```

MATRIX mempunyai dua dimensi , dihitung dari jumlah tanda kurung siku di sisi luar.

```
# Check number of dimensions
MATRIX.ndim

2
```

Kita mendapatkan hasil torch.Size([2, 2]) karena MATRIX memiliki kedalaman dua elemen dan lebar dua elemen.

```
MATRIX.shape

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

Cara membuat tensor.

```
# Tensor
TENSOR = torch.tensor([[[1, 2, 3],
                        [3, 6, 9],
                        [2, 4, 5]]])
TENSOR

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

Dimensi yang dihasilkan adalah 3 dari kode di atau dihitung dari jumlah tanda kurung sikunya.

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

```
3
```

menghasilkan torch.Size([1, 3, 3]). Dimensinya dari luar ke dalam. Artinya ada 1 dimensi 3 kali 3.

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

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

#Tensor acak

Mulailah dengan angka acak -> lihat data -> perbarui nomor acak -> lihat data -> perbarui nomor acak...

Membuat tensor bilangan acak. Dapat dilakukan menggunakan torch.rand() dan meneruskan parameter ukuran.

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

```
(tensor([[0.6541, 0.4807, 0.2162, 0.6168],
         [0.4428, 0.6608, 0.6194, 0.8620],
         [0.2795, 0.6055, 0.4958, 0.5483]]),
 torch.float32)
```

Fleksibilitas dari torch.rand() adalah kita dapat mengatur ukurannya sesuai keinginan kita.

Misalnya, kita menginginkan tensor acak dalam bentuk gambar umum [224, 224, 3] ([tinggi, lebar, saluran_warna]).

```
# 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)
```

#Nol dan satu

Terkadang kita hanya ingin mengisi tensor dengan nol atau satu. Hal ini sering terjadi dengan masking (seperti menutupi beberapa nilai dalam satu tensor dengan nol agar model tahu untuk tidak mempelajarinya).

Mari kita buat tensor penuh nol dengan torch.zeros() Sekali lagi, parameter ukuran ikut berperan.

```
# Create a tensor of all zeros
zeros = torch.zeros(size=(3, 4))
zeros, zeros.dtype
```

```
(tensor([[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]),
 torch.float32)
```

Kita bisa melakukan hal yang sama untuk membuat tensor semuanya kecuali menggunakan torch.ones() sebagai gantinya.

```
# Create a tensor of all ones
ones = torch.ones(size=(3, 4))
ones, ones.dtype
```

```
(tensor([[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]),
 torch.float32)
```

#Membuat rentang dan tensor sejenisnya

Terkadang kita mungkin menginginkan rentang angka, seperti 1 hingga 10 atau 0 hingga 100.

Anda dapat menggunakan torch.arange(start, end, step) untuk melakukannya.

Di mana: • start = awal rentang (misalnya 0) • akhir = akhir rentang (misalnya 10) • langkah = berapa banyak langkah di antara setiap nilai (misalnya 1)

```
# Use torch.arange(), torch.range() is deprecated
zero_to_ten_deprecated = torch.range(0, 10) # Note: this may return an
error in the future

# Create a range of values 0 to 10
zero_to_ten = torch.arange(start=0, end=10, step=1)
zero_to_ten
```

```
/tmp/ipykernel_3695928/193451495.py:2: UserWarning: torch.range is
deprecated and will be removed in a future release because its
behavior is inconsistent with Python's range builtin. Instead, use
torch.arange, which produces values in [start, end).
  zero_to_ten_deprecated = torch.range(0, 10) # Note: this may return
an error in the future

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

Terkadang kita mungkin menginginkan satu tensor jenis tertentu dengan bentuk yang sama dengan tensor lainnya.

Misalnya tensor yang semuanya nol dengan bentuk yang sama dengan tensor sebelumnya.

Untuk melakukannya, Anda dapat menggunakan torch.zeros_like(input) atau torch.ones_like(input) yang mengembalikan tensor yang diisi dengan nol atau satu dalam bentuk yang sama dengan inputnya.

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

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

#Tipe data tensor

Ada banyak tipe data tensor berbeda yang tersedia di PyTorch.

Ada yang khusus untuk CPU dan ada pula yang lebih baik untuk GPU.

Mengenal mana yang membutuhkan waktu.

Umumnya jika kita melihat torch.cuda di mana pun, tensor digunakan untuk GPU (karena GPU Nvidia menggunakan perangkat komputasi yang disebut CUDA)

Jenis yang paling umum (dan umumnya default) adalah torch.float32 atau torch.float.

Ini disebut sebagai "titik mengambang 32-bit".

Namun ada juga floating point 16-bit (torch.float16 atau torch.half) dan floating point 64-bit (torch.float64 atau torch.double).

Dan yang lebih membingungkan lagi, ada juga bilangan bulat 8-bit, 16-bit, 32-bit, dan 64-bit.

Mari kita lihat cara membuat beberapa tensor dengan tipe data tertentu. Kita dapat melakukannya menggunakan parameter dtype.

```
# Default datatype for tensors is float32
float_32_tensor = torch.tensor([3.0, 6.0, 9.0],
                               dtype=None, # defaults to None, which
is torch.float32 or whatever datatype is passed
                               device=None, # defaults to None, which
uses the default tensor type
                               requires_grad=False) # if True,
operations performed on the tensor are recorded

float_32_tensor.shape, float_32_tensor.dtype, float_32_tensor.device
```

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

Selain masalah bentuk (bentuk tensor tidak cocok), dua masalah paling umum lainnya yang akan Anda temui di PyTorch adalah masalah tipe data dan perangkat.

Misalnya, salah satu tensornya adalah torch.float32 dan yang lainnya adalah torch.float16 (PyTorch sering kali menyukai tensor dengan format yang sama).

Atau salah satu tensor Anda ada di CPU dan yang lainnya ada di GPU (PyTorch menyukai penghitungan antar tensor berada di perangkat yang sama).

Untuk saat ini mari buat tensor dengan dtype=torch.float16.

```
float_16_tensor = torch.tensor([3.0, 6.0, 9.0],
                               dtype=torch.float16) # torch.half would
also work

float_16_tensor.dtype
```

```
torch.float16
```

#Mendapatkan informasi dari tensor

Setelah Anda membuat tensor (atau orang lain atau modul PyTorch telah membuatnya untuk Anda), Anda mungkin ingin mendapatkan beberapa informasi dari tensor tersebut. adalah torch.float32 atau tipe data apa pun yang diteruskan Kita akan melihat lebih banyak pembicaraan tentang perangkat ini nanti.

Kami telah melihat ini sebelumnya, tetapi tiga atribut paling umum yang ingin Anda ketahui tentang tensor adalah:

• bentuk - apa bentuk tensornya? (beberapa operasi memerlukan aturan bentuk tertentu) • dtype - tipe data apa yang menyimpan elemen dalam tensor? • perangkat - di perangkat apa tensor disimpan? (biasanya GPU atau CPU)

Mari buat tensor acak dan cari tahu detailnya.

```
# 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}") # will
default to CPU
```

```
tensor([[0.4688, 0.0055, 0.8551, 0.0646],
        [0.6538, 0.5157, 0.4071, 0.2109],
        [0.9960, 0.3061, 0.9369, 0.7008]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

#Operasi dasar

Mari kita mulai dengan beberapa operasi dasar, penjumlahan (+), pengurangan (-), perkalian ganda (*).

Mereka bekerja seperti yang Anda kira.

```
# Create a tensor of values and add a number to it
tensor = torch.tensor([1, 2, 3])
tensor + 10
```

```
tensor([11, 12, 13])
```

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

```
tensor([10, 20, 30])
```

Perhatikan bagaimana nilai tensor di atas tidak menjadi tensor([110, 120, 130]), hal ini karena nilai di dalam tensor tidak berubah kecuali jika ditetapkan ulang.

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

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

Mari kita kurangi sebuah angka dan kali ini kita akan menetapkan ulang variabel tensornya.

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

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

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

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

PyTorch juga memiliki banyak fungsi bawaan seperti torch.mul() (kependekan dari perkalian) dan torch.add() untuk melakukan operasi dasar.

```
# Can also use torch functions
torch.multiply(tensor, 10)
```

```
tensor([10, 20, 30])
```

```
# Original tensor is still unchanged
tensor
```

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

Namun, lebih umum menggunakan simbol operator seperti * daripada torch.mul()

```
# Element-wise multiplication (each element multiplies its equivalent,
index 0->0, 1->1, 2->2)
```

```
print(tensor, "*", tensor)
print("Equals:", tensor * tensor)
```

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

#Perkalian matriks (yang Anda perlukan)

Mari kita membuat tensor dan melakukan perkalian berdasarkan elemen dan perkalian matriks di atasnya.

```
import torch
tensor = torch.tensor([1, 2, 3])
tensor.shape
```

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

```
# Element-wise matrix multiplication
tensor * tensor
```

```
tensor([1, 4, 9])
```

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

```
tensor(14)
```

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

```
tensor(14)
```

Metode torch.matmul() bawaan lebih cepat.

```
%%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 773 µs, sys: 0 ns, total: 773 µs
Wall time: 499 µs
```

```
tensor(14)
```

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

```
CPU times: user 146 µs, sys: 83 µs, total: 229 µs
Wall time: 171 µs

tensor(14)
```

#Salah satu kesalahan paling umum dalam pembelajaran mendalam (kesalahan bentuk)

Karena sebagian besar pembelajaran mendalam adalah mengalikan dan melakukan operasi pada matriks, dan matriks memiliki aturan ketat tentang bentuk dan ukuran apa yang dapat digabungkan, salah satu kesalahan paling umum yang akan Anda temui dalam pembelajaran mendalam adalah ketidakcocokan bentuk.

```
# Shapes need to be in the right way
tensor_A = torch.tensor([[1, 2],
                         [3, 4],
                         [5, 6]], dtype=torch.float32)

tensor_B = torch.tensor([[7, 10],
                         [8, 11],
                         [9, 12]], dtype=torch.float32)

torch.matmul(tensor_A, tensor_B) # (this will error)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call
last)
/home/daniel/code/pytorch/pytorch-course/pytorch-deep-learning/00_pyto
rch_fundamentals.ipynb Cell 75 in <cell line: 10>()
      <a href='vscode-notebook-cell://ssh-remote
%2B7b22686f73744e616d65223a22544954414e2d525458227d/home/daniel/code/
pytorch/pytorch-course/pytorch-deep-learning/
00_pytorch_fundamentals.ipynb#Y134sdnNjb2RlLXJlbW90ZQ%3D%3D?
line=1'>2</a> tensor_A = torch.tensor([[1, 2],
      <a href='vscode-notebook-cell://ssh-remote
%2B7b22686f73744e616d65223a22544954414e2d525458227d/home/daniel/code/
pytorch/pytorch-course/pytorch-deep-learning/
00_pytorch_fundamentals.ipynb#Y134sdnNjb2RlLXJlbW90ZQ%3D%3D?
line=2'>3</a>                             [3, 4],
      <a href='vscode-notebook-cell://ssh-remote
%2B7b22686f73744e616d65223a22544954414e2d525458227d/home/daniel/code/
pytorch/pytorch-course/pytorch-deep-learning/
00_pytorch_fundamentals.ipynb#Y134sdnNjb2RlLXJlbW90ZQ%3D%3D?
line=3'>4</a>                             [5, 6]], dtype=torch.float32)
      <a href='vscode-notebook-cell://ssh-remote
%2B7b22686f73744e616d65223a22544954414e2d525458227d/home/daniel/code/
pytorch/pytorch-course/pytorch-deep-learning/
00_pytorch_fundamentals.ipynb#Y134sdnNjb2RlLXJlbW90ZQ%3D%3D?
line=5'>6</a> tensor_B = torch.tensor([[7, 10],
      <a href='vscode-notebook-cell://ssh-remote
```

```
%2B7b22686f73744e616d65223a22544954414e2d525458227d/home/daniel/code/
pytorch/pytorch-course/pytorch-deep-learning/
00_pytorch_fundamentals.ipynb#Y134sdnNjb2RlLXJlbW90ZQ%3D%3D?
line=6'>7</a>                                [8, 11],
      <a href='vscode-notebook-cell://ssh-remote
%2B7b22686f73744e616d65223a22544954414e2d525458227d/home/daniel/code/
pytorch/pytorch-course/pytorch-deep-learning/
00_pytorch_fundamentals.ipynb#Y134sdnNjb2RlLXJlbW90ZQ%3D%3D?
line=7'>8</a>                                [9, 12]], dtype=torch.float32)
---> <a href='vscode-notebook-cell://ssh-remote
%2B7b22686f73744e616d65223a22544954414e2d525458227d/home/daniel/code/
pytorch/pytorch-course/pytorch-deep-learning/
00_pytorch_fundamentals.ipynb#Y134sdnNjb2RlLXJlbW90ZQ%3D%3D?
line=9'>10</a> torch.matmul(tensor_A, tensor_B)

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

Kita dapat membuat perkalian matriks berfungsi antara tensor_A dan tensor_B dengan mencocokkan dimensi dalamnya.

Salah satu cara untuk melakukannya adalah dengan transpose (mengganti dimensi tensor tertentu).

Anda dapat melakukan transpos di PyTorch menggunakan:

• torch.transpose(input, dim0, dim1) - dengan input adalah tensor yang diinginkan untuk ditransposisi dan dim0 serta dim1 adalah dimensi yang akan ditukar. Traceback (panggilan terakhir • tensor.T - di mana tensor adalah tensor yang diinginkan untuk diubah urutannya.

```
# View tensor_A and tensor_B
print(tensor_A)
print(tensor_B)

tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
tensor([[ 7., 10.],
        [ 8., 11.],
        [ 9., 12.]])

# View tensor_A and tensor_B.T
print(tensor_A)
print(tensor_B.T)

tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
tensor([[ 7.,  8.,  9.],
        [10., 11., 12.]])
```

```
# The operation works when tensor_B is transposed
print(f"Original shapes: tensor_A = {tensor_A.shape}, tensor_B =
{tensor_B.shape}\n")
print(f"New shapes: tensor_A = {tensor_A.shape} (same as above),
tensor_B.T = {tensor_B.T.shape}\n")
print(f"Multiplying: {tensor_A.shape} * {tensor_B.T.shape} <- inner
dimensions match\n")
print("Output:\n")
output = torch.matmul(tensor_A, tensor_B.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([[ 27.,  30.,  33.],
        [ 61.,  68.,  75.],
        [ 95., 106., 117.]])

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

You can also use `torch.mm()` which is a short for `torch.matmul()`.

Anda juga dapat menggunakan torch.mm() yang merupakan kependekan dari torch.matmul().

```
# torch.mm is a shortcut for matmul
torch.mm(tensor_A, tensor_B.T)
```

```
tensor([[ 27.,  30.,  33.],
        [ 61.,  68.,  75.],
        [ 95., 106., 117.]])
```

```
# Since the linear layer starts with a random weights matrix, let's
make it reproducible (more on this later)
torch.manual_seed(42)
# This uses matrix multiplication
linear = torch.nn.Linear(in_features=2, # in_features = matches inner
dimension of input
                         out_features=6) # out_features = describes
outer value
x = tensor_A
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([[2.2368, 1.2292, 0.4714, 0.3864, 0.1309, 0.9838],
        [4.4919, 2.1970, 0.4469, 0.5285, 0.3401, 2.4777],
        [6.7469, 3.1648, 0.4224, 0.6705, 0.5493, 3.9716]],
       grad_fn=<AddmmBackward0>)

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

#Menemukan min, maks, mean, jumlah, dll (agregasi)

Sekarang kita telah melihat beberapa cara untuk memanipulasi tensor, mari kita lihat beberapa cara untuk menggabungkannya (mulai dari nilai yang lebih banyak ke nilai yang lebih kecil).

Pertama kita akan membuat tensor lalu mencari nilai maks, min, rata-rata, dan jumlahnya.

```
# Create a tensor
x = torch.arange(0, 100, 10)
x
```

```
tensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Sekarang mari kita lakukan beberapa agregasi.

```
print(f"Minimum: {x.min()}")
print(f"Maximum: {x.max()}")
# print(f"Mean: {x.mean()}") # this will error
print(f"Mean: {x.type(torch.float32).mean()}") # won't work without
float datatype
print(f"Sum: {x.sum()}")
```

```
Minimum: 0
Maximum: 90
Mean: 45.0
Sum: 450
```

Catatan: Anda mungkin menemukan beberapa metode seperti torch.mean() mengharuskan tensor berada di torch.float32 (yang paling umum) atau tipe data spesifik lainnya, jika tidak, operasi akan gagal.

Anda juga dapat melakukan hal yang sama seperti di atas dengan metode obor.

```
torch.max(x), torch.min(x), torch.mean(x.type(torch.float32)),
torch.sum(x)
```

```
(tensor(90), tensor(0), tensor(45.), tensor(450))
```

#Posisi min/maks

Anda juga dapat menemukan indeks tensor yang nilai maks atau minimumnya muncul dengan torch.argmax() dan torch.argmin() masing-masing.

Ini berguna jika Anda hanya menginginkan posisi dengan nilai tertinggi (atau terendah) dan bukan nilai sebenarnya (kita akan melihatnya di bagian selanjutnya saat menggunakan fungsi aktivasi softmax).

```
# Create a tensor
tensor = torch.arange(10, 100, 10)
print(f"Tensor: {tensor}")

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

```
Tensor: tensor([10, 20, 30, 40, 50, 60, 70, 80, 90])
Index where max value occurs: 8
Index where min value occurs: 0
```

#Ubah tipe data tensor

Pertama kita akan membuat tensor dan memeriksa tipe datanya (defaultnya adalah torch.float32).

```
# Create a tensor and check its datatype
tensor = torch.arange(10., 100., 10.)
tensor.dtype
```

```
torch.float32
```

Sekarang kita akan membuat tensor lain sama seperti sebelumnya tetapi mengubah tipe datanya menjadi torch.float16.

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

```
tensor([10., 20., 30., 40., 50., 60., 70., 80., 90.],
dtype=torch.float16)
```

Dan kita bisa melakukan hal serupa untuk membuat tensor torch.int8.

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

```
tensor([10, 20, 30, 40, 50, 60, 70, 80, 90], dtype=torch.int8)
```

#Membentuk kembali

Pertama, kita akan membuat tensor.

```
# Create a tensor
import torch
x = torch.arange(1., 8.)
x, x.shape
```

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

Sekarang mari tambahkan dimensi ekstra dengan torch.reshape().

```
# Add an extra dimension
x_reshaped = x.reshape(1, 7)
x_reshaped, x_reshaped.shape
```

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

Kita juga bisa mengubah tampilan dengan torch.view().

```
# Change view (keeps same data as original but changes view)
# See more: https://stackoverflow.com/a/54507446/7900723
z = x.view(1, 7)
z, z.shape
```

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

Namun ingat, mengubah tampilan tensor dengan torch.view() sebenarnya hanya akan menghasilkan tampilan baru. Jadi mengubah tampilan juga akan mengubah tensor aslinya.

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

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

Jika kita ingin menumpuk tensor baru di atasnya sebanyak lima kali, kita dapat melakukannya dengan torch.stack().

```
# Stack tensors on top of each other
x_stacked = torch.stack([x, x, x, x], dim=0) # try changing dim to
dim=1 and see what happens
x_stacked
```

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

Bagaimana kalau menghapus semua dimensi tunggal dari tensor?

Untuk melakukannya, Anda dapat menggunakan torch.squeeze() (Saya ingat ini sebagai dimensi lebih dari 1).

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

# Remove extra dimension 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., 2., 3., 4., 5., 6., 7.]])
Previous shape: torch.Size([1, 7])

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

Dan untuk melakukan kebalikan dari torch.squeeze() Anda dapat menggunakan torch.unsqueeze() untuk menambahkan nilai dimensi 1 pada indeks tertentu.

```
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., 2., 3., 4., 5., 6., 7.])
Previous shape: torch.Size([7])

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

Anda juga dapat mengatur ulang urutan nilai sumbu dengan torch.permute(input, 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) # shifts axis 0->1, 1->2, 2->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])
```

#Pengindeksan (memilih data dari tensor)

Terkadang Anda ingin memilih data tertentu dari tensor (misalnya, hanya kolom pertama atau baris kedua).

Untuk melakukannya, Anda dapat menggunakan pengindeksan.

```
# Create a tensor
import torch
x = torch.arange(1, 10).reshape(1, 3, 3)
x, x.shape
```

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

Nilai pengindeksan menjadi dimensi luar -> dimensi dalam (lihat tanda kurung siku).

```
# 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 bracket: {x[0][0][0]}")
```

```
First square bracket:
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
Second square bracket: tensor([1, 2, 3])
Third square bracket: 1
```

Anda juga dapat menggunakan : untuk menentukan "semua nilai dalam dimensi ini" dan kemudian menggunakan koma (,) untuk menambahkan dimensi lain.

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

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

```
# Get all values of 0th & 1st dimensions but only index 1 of 2nd dimension
x[:, :, 1]
```

```
tensor([[2, 5, 8]])
```

```
# Get all values of the 0 dimension but only the 1 index value of the 1st and 2nd dimension
x[:, 1, 1]
```

```
tensor([5])
```

```
# Get index 0 of 0th and 1st dimension and all values of 2nd dimension
x[0, 0, :] # same as x[0][0]
```

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

#Tensor PyTorch & NumPy

Dua metode utama yang ingin Anda gunakan untuk NumPy ke PyTorch (dan kembali lagi) adalah:

• torch.from_numpy(ndarray) - Array NumPy -> Tensor PyTorch. • torch.Tensor.numpy() - Tensor PyTorch -> array NumPy.

Mari kita mencobanya.

```
# 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))
```

Karena kami menetapkan ulang tensor di atas, jika Anda mengubah tensor, arraynya akan tetap sama.

```
# 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))
```

Dan jika Anda ingin beralih dari tensor PyTorch ke array NumPy, Anda dapat memanggil tensor.numpy()

```
# Tensor to NumPy array
tensor = torch.ones(7) # create a tensor of ones with dtype=float32
numpy_tensor = tensor.numpy() # will be dtype=float32 unless changed
tensor, numpy_tensor
```

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

Dan aturan yang sama berlaku seperti di atas, jika Anda mengubah tensor asli, numpy_tensor baru tetap sama.

```
# 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))
```

#Reproduksibilitas (mencoba mengambil yang acak dari yang acak)

Kita akan mulai dengan membuat dua tensor acak, karena keduanya acak, Anda pasti mengira keduanya berbeda, bukan?

```
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([[0.8016, 0.3649, 0.6286, 0.9663],
        [0.7687, 0.4566, 0.5745, 0.9200],
        [0.3230, 0.8613, 0.0919, 0.3102]])

Tensor B:
tensor([[0.9536, 0.6002, 0.0351, 0.6826],
        [0.3743, 0.5220, 0.1336, 0.9666],
        [0.9754, 0.8474, 0.8988, 0.1105]])

Does Tensor A equal Tensor B? (anywhere)

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

Mari kita mencobanya dengan membuat lebih banyak lagi

```
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]])
```

#Menjalankan tensor pada GPU (dan membuat komputasi lebih cepat)

Mendapatkan GPU

```
!nvidia-smi
```

```
Sat Jan 21 08:34:23 2023
+-----------------------------------------------------------------------------
--------+
| NVIDIA-SMI 515.48.07    Driver Version: 515.48.07    CUDA Version:
11.7     |
|-------------------------------+----------------------
+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util
Compute M. |
|                               |                      |
MIG M. |
|
===============================+======================+==============
=======|
|   0  NVIDIA TITAN RTX    On   | 00000000:01:00.0 Off |
```

```
N/A |
| 40%   30C    P8     7W / 280W |    177MiB / 24576MiB |      0%
Default |
|                                |                      |
N/A |
+-------------------------------+----------------------
+----------------------+


+-----------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                  GPU
Memory |
|        ID   ID
Usage      |
|
=======================================================================
=======|
|    0   N/A  N/A      1061      G   /usr/lib/xorg/Xorg
53MiB |
|    0   N/A  N/A   2671131      G   /usr/lib/xorg/Xorg
97MiB |
|    0   N/A  N/A   2671256      G   /usr/bin/gnome-shell
9MiB |
+-----------------------------------------------------------------------
--------+
```

Menjalankan PyTorch di GPU

Anda dapat menguji apakah PyTorch memiliki akses ke GPU menggunakan torch.cuda.is_available().

```
# Check for GPU
import torch
torch.cuda.is_available()

True
```

Mari buat variabel perangkat untuk menyimpan jenis perangkat yang tersedia.

```
# Set device type
device = "cuda" if torch.cuda.is_available() else "cpu"
device

'cuda'
```

Anda dapat menghitung jumlah GPU yang dapat diakses PyTorch menggunakan torch.cuda.device_count().

```
# Count number of devices
torch.cuda.device_count()
```

```
1
```

Menempatkan tensor (dan model) pada GPU

Mari kita coba membuat tensor dan meletakkannya di GPU (jika tersedia).

```
# Create tensor (default on CPU)
tensor = torch.tensor([1, 2, 3])

# Tensor not on GPU
print(tensor, tensor.device)

# Move tensor to GPU (if available)
tensor_on_gpu = tensor.to(device)
tensor_on_gpu
```

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

tensor([1, 2, 3], device='cuda:0')
```

Memindahkan tensor kembali ke CPU

Mari kita coba menggunakan metode torch.Tensor.numpy() pada tensor_on_gpu kita

```
# If tensor is on GPU, can't transform it to NumPy (this will error)
tensor_on_gpu.numpy()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)
/home/daniel/code/pytorch/pytorch-course/pytorch-deep-learning/00_pyto
rch_fundamentals.ipynb Cell 157 in <cell line: 2>()
      <a href='vscode-notebook-cell://ssh-remote
%2B7b22686f73744e616d65223a22544954414e2d525458227d/home/daniel/code/
pytorch/pytorch-course/pytorch-deep-learning/
00_pytorch_fundamentals.ipynb#Y312sdnNjb2RlLXJlbW90ZQ%3D%3D?
line=0'>1</a> # If tensor is on GPU, can't transform it to NumPy (this
will error)
----> <a href='vscode-notebook-cell://ssh-remote
%2B7b22686f73744e616d65223a22544954414e2d525458227d/home/daniel/code/
pytorch/pytorch-course/pytorch-deep-learning/
00_pytorch_fundamentals.ipynb#Y312sdnNjb2RlLXJlbW90ZQ%3D%3D?
line=1'>2</a> tensor_on_gpu.numpy()

TypeError: can't convert cuda:0 device type tensor to numpy. Use
Tensor.cpu() to copy the tensor to host memory first.
```

Ini menyalin tensor ke memori CPU sehingga dapat digunakan dengan CPU

```
# Instead, copy the tensor back to cpu
tensor_back_on_cpu = tensor_on_gpu.cpu().numpy()
tensor_back_on_cpu
```

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

Cara di atas mengembalikan salinan tensor GPU di memori CPU sehingga tensor asli masih di GPU.

```
tensor_on_gpu
```

```
tensor([1, 2, 3], device='cuda:0')
```