Chapter 3

It starts with a tensor

This chapter covers:

- Understanding tensors, the basic data structure in PyTorch
- Indexing and operating on tensors
- Interoperating with NumPy multidimensional arrays
- Moving computations to the GPU for speed

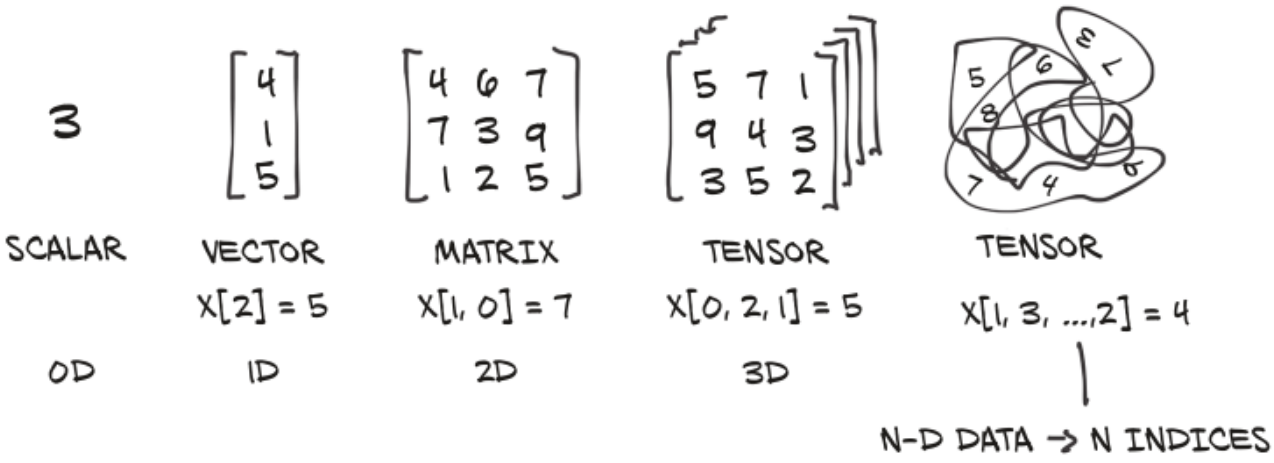
Jupyter Notebook Link (https://github.com/deep-learning-with-pytorch/dlwpt-code/tree/master/p1ch3)

Network deals with information in floating-point numbers

- We need a way to encode real-world data into something digestible by a network
- And a way to decode the output back to something we can understand and use for our purpose

In the context of deep learning, tensors refer to **generalization of vectors and matrices to an arbitrary number of dimensions**, aka **multidimensional arrays**

- The dimensionality of a tensor coincides with the number of indexes used to refer to scalar values within the tensor



NumPy is also one of the most popular multi-dim array library

- seamless interoperability with Numpy
  - `numpy.asarray()`
  - `torch.from_numpy()`
- Therefore provides integration with:
  - Scipy (www.scipy.org)
  - Scikit-learn (https://scikit-learn.org)
  - Pandas (https://pandas.pydata.org)

PyTorch tensors have a few abilities that numpy doesn't

- use GPU
- distribute operations on multiple devices (or machines)

- keep track of the graph of computation that created them

In [5]:

```
print("----Python Lists----")
"""
First take a look at Python list
"""
a = [1.0,2.0,1.0]
print("a[0]:", a[0])
a[2] = 3.0
print("a:",a)

print("\n\n----Pytorch Tensors----")
"""
Constructing out first tensors
"""
import torch

a = torch.ones(3)
print("a:", a)
print("a[1]:", a[1])
print("float(a[1]):", float(a[1]))
a[2] = 2.0
print("a:", a)
```

```
----Python Lists----
a[0]: 1.0
a: [1.0, 2.0, 3.0]


----Pytorch Tensors----
a: tensor([1., 1., 1.])
a[1]: tensor(1.)
float(a[1]): 1.0
a: tensor([1., 1., 2.])
```

In [6]:

```
"""
Python list or tuple: elements individually allocated in memory
PyTorch tensors or NumPy arrays: typically contiguous memory blocks
"""

points = torch.zeros(6) # Using .zeros is just  way to get an appropriately sized a

print(points)

points[0] = 4.0 # Overwrite the zeros with values we actually want
points[1] = 1.0
points[2] = 5.0
points[3] = 3.0
points[4] = 2.0
points[5] = 1.0

print(points)
```

```
tensor([0., 0., 0., 0., 0., 0.])
tensor([4., 1., 5., 3., 2., 1.])
```

In [13]:

```
# Can also pass python list to constructor
points = torch.tensor([4.0, 1.0, 5.0, 3.0, 2.0, 1.0])
print(points)

# float(tensor value) to get the coordinates
print(float(points[0]), float(points[1]))

#Instantiate 2d tensor from 2d list
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
print(points)

# Getting the shape of tensor
print("Shape of tensor:", points.shape)

# Using .zeros or .ones to initialize the tensor
points = torch.zeros(3, 2)
print(points)

# Use two indices to access element in 2d tensor
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
print(points[0, 1])
print(points[0][1])
print(points[0]) # access first tensor
```

```
tensor([4., 1., 5., 3., 2., 1.])
4.0 1.0
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
Shape of tensor: torch.Size([3, 2])
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
tensor(1.)
tensor(1.)
tensor([4., 1.])
```

In [19]:

```
# Indexing list
some_list = list(range(6))
some_list[:] # all elements
some_list[1:4] # from element 1(inclusive) to element 4 (exclusive)
some_list[1:] # from element 1 to end of list
some_list[:4] # from beginning of list to element 4 (exclusive)
some_list[:-1] # from beginning of list to ONE BEFORE the last element
some_list[1:4:2] # from element 1 inclusive to element 4 exclusive, in steps of 2

# Indexing tensors
"""
added functionality: can use range indexing for EACH of the tensor's dimensions
"""

print(points[1:]) # all rows after first
print(points[1:, :]) # all rows after first, all columns
print(points[1:, 0]) # all rows after first, first column
print(points[None]) # adds dimension of size 1, just like unsqueeze

print(points.storage()) # NOT IN OPENSOURCE TEXTBOOK
```

```
tensor([[5., 3.],
        [2., 1.]])
tensor([[5., 3.],
        [2., 1.]])
tensor([5., 2.])
tensor([[[4., 1.],
         [5., 3.],
         [2., 1.]]])
 4.0
 1.0
 5.0
 3.0
 2.0
 1.0
[torch.FloatStorage of size 6]
```

Dimensions (or axis) of tensors usually index something like pixel locations or color channels.

- Need to remember the ordering of dimensions when indexing into a tensor
- Keeping track of which dimension contains what data can be error-prone

In [31]:

```
img_t = torch.randn(3, 5, 5) # shape [channels, rows, columns]
weights = torch.tensor([0.2126, 0.7152, 0.0722])
batch_t = torch.randn(2, 3, 5, 5) # shape [batch, channels, rows, columns]
"""
For generalizability, count from the back
i.e. batch data has BATCH at 0th position, image data has CHANNEL as 0th position
but always same, counting from the back
"""
img_gray_naive = img_t.mean(-3) # average channel
batch_gray_naive = batch_t.mean(-3) # average channel

print(img_gray_naive.shape, batch_gray_naive.shape)

"""
PyTorch allows us to multiply things that are the same shape
as well as shapes where one operand is of size 1 in a given dimension -> torch.Size
It also appends leading dimensions of size 1 automatically [2,5,5] * [5,5] -> [2,5,
"""
unsqueezed_weights = weights.unsqueeze(-1).unsqueeze_(-1)
print(weights.shape)
print(unsqueezed_weights.shape)
img_weights = (img_t * unsqueezed_weights) # 3,5,5 X 3,1,1 -> 3,5,5
batch_weights = (batch_t * unsqueezed_weights) # 2,3,5,5 * 3,1,1 -> 2,3,5,5
img_gray_weighted = img_weights.sum(-3) # 3,5,5 -> 5,5
batch_gray_weighted = batch_weights.sum(-3) # 2,3,5,5 -> 2,5,5
print(batch_weights.shape, batch_t.shape, unsqueezed_weights.shape)
print(img_gray_weighted.shape, batch_gray_weighted.shape)
```

```
torch.Size([5, 5]) torch.Size([2, 5, 5])
torch.Size([3])
torch.Size([3, 1, 1])
torch.Size([2, 3, 5, 5]) torch.Size([2, 3, 5, 5]) torch.Size([3, 1,
1])
torch.Size([5, 5]) torch.Size([2, 5, 5])
```

In [33]:

```
"""
The above methods get messy quickly ㄹㅇㅋㅋ
For sake of efficiency: PyTorch function einsum
einsum :: specifies an indexing mini-language (https://rockt.github.io/2018/04/30/e
"""
img_gray_weighted_fancy = torch.einsum('...chw,c->...hw', img_t, weights)
batch_gray_weighted_fancy = torch.einsum('...chw,c->...hw', batch_t, weights)
batch_gray_weighted_fancy.shape

# summarizing unnamed things using "..."
# Seems OK to learn how einsum works, for the sake of readability
```

Out[33]:

```
torch.Size([2, 5, 5])
```

In [46]:

```
#The above are still error-prone, so introduced NAMED TENSORS in PyTorch1.3
# > are considered to be experimental, subject to change. Don't use for important s

weights_named = torch.tensor([0.2126, 0.7152, 0.0722], names=['channels'])
print(weights_named)


#When adding names : use "refine_names"
#Can also overwrite or drop(by passing in None) existing names

img_named = img_t.refine_names(..., 'channels', 'rows', 'columns')
batch_named = batch_t.refine_names(..., 'channels', 'rows', 'columns')
batch_changed = batch_t.refine_names(..., None, None, 'columns')
print("\nimg named:", img_named.shape, img_named.names)
print("batch named:", batch_named.shape, batch_named.names)
print("batch changed:", batch_changed.shape, batch_changed.names)


# Can align names too
# "align_as" returns tensor with missing dimensions added and existing ones permute

weights_aligned = weights_named.align_as(img_named)
print("\naligned weights:",weights_aligned.shape, weights_aligned.names)


# Functions accepting dimension arguments (eg sum) also take named dimensions

gray_named = (img_named * weights_aligned).sum('channels')
print("\nSum taking names:", gray_named.shape, gray_named.names)

# Trying to combine dimensions with different names returns an error
# To use tensors outside functions that operate on named tensors, need to drop name

gray_plain = gray_named.rename(None)
print("\nDropped names:",gray_plain.shape, gray_plain.names)

# The book sticks to unnamed tensors, given the experimental nature of this feature
```

```
tensor([0.2126, 0.7152, 0.0722], names=('channels',))

img named: torch.Size([3, 5, 5]) ('channels', 'rows', 'columns')
batch named: torch.Size([2, 3, 5, 5]) (None, 'channels', 'rows', 'colu
mns')
batch changed: torch.Size([2, 3, 5, 5]) (None, None, None, 'columns')

aligned weights: torch.Size([3, 1, 1]) ('channels', 'rows', 'columns')

Sum taking names: torch.Size([5, 5]) ('rows', 'columns')

Dropped names: torch.Size([5, 5]) (None, None)
```

Tensor Element Types

Using the standard Python numeric types can be suboptimal:

- Numbers in python are objects (heavier, requires more bits)
- Lists in Python are meant for sequential collections of objects

    - no operations for dot product, or summing vectors together.
- The python interpreter is slow compared to optimized, compiled code.

To fully use **efficient low-level implementations** of numerical data structures and related operations on them, the objects within a tensor must all be numbers of the **same type**

`dtype` argument to tensor constructors (also in Numpy) supports:

- `torch.float32` or `torch.float` : 32-bit floating-point : **DEFAULY DATA TYPE**
- `torch.float64` or `torch.double` : 64-bit, double-precision floating-point
- `torch.float16` or `torch.half` : 16-bit, half-precision floating-point
    - supported in recent GPUs, much faster at a cost of a little lower accuracy
- `torch.int8` : signed 8-bit integers
- `torch.uint8` : unsigned 8-bit integers
- `torch.int16` or torch.short : signed 16-bit integers
- `torch.int32` or torch.int : signed 32-bit integers
- `torch.int64` or torch.long : signed 64-bit integers
- `torch.bool` : Boolean

Tensors can be used as indexes in other tensors

- expects this tensor to have 64-bit integer data type
- creating tensor with integers as arguments create 64-bit integer tensor by default
    - ex) `torch.tensor([2, 2])`

Therefore, will mostly use float32 and int64

Predicates on tensors (ex. `points > 1.0`) produce `bool` tensors indicating whether each individual element satisfiefs the condition.

In [48]:

```
double_points = torch.ones(10, 2, dtype=torch.double)
short_points = torch.tensor([[1, 2], [3, 4]], dtype=torch.short)
print(short_points.dtype)

# Can also cast the output of a tensor creator function
double_points = torch.zeros(10, 2).double()
short_points = torch.ones(10, 2).short()

# or the more convenient 'to' method
double_points = torch.zeros(10, 2).to(torch.double)
short_points = torch.ones(10, 2).to(dtype=torch.short)
```

```
torch.int16
```

The tensor API

Worth taking a look at tensor operations that PyTorch offers

- would be of little use to list them all here
- just get a general feel for the API
- more specifics in the [online documentation (http://pytorch.org/docs)](http://pytorch.org/docs)

In [50]:

```
a = torch.ones(3, 2)
a_t = torch.transpose(a,0,1)
a_t2 = a.transpose(0,1)
print(a.shape, a_t.shape, a_t2.shape)
```

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

Tensor operations (check more in document) are divided into groups:

- Creation ops
  - for constructing a tensor.
  - ex) `ones` and `from_numpy`
- Indexing, Slicing, Joining, Mutating ops
  - functions for changing the shape, stride, or content of a tensor
  - ex) `transpose`
- Math ops
  - pointwise ops : obtains a new tensor by applying function to each element separately (`abs` , `cos`)
  - reduction ops: aggregate values by iterating through tensors (`mean` , `std` , `norm`)
  - comparison ops: evaluate numerical predicates over tensors (`equal` , `max`)
  - spectral ops: transforming in and operating in the frequency domain (`stft` , `hamming_window`) <-- ??
  - other operations: special functions operating on vectors (`cross` , `trace`)
  - BLAS and LAPACK operations: Basic Linear Algebra Subprograms (vec-vec, vec-mat, mat-mat operations)
- Random sampling
  - generate values by drawing randomly from probability distributions
  - ex) `randn` , `normal`
- Serialization
  - saving and loading tensors
  - ex) `load` , `save`
- Parallelism
  - controlling number of threads for parallel CPU execution
  - ex) `set_num_threads`

In [51]:

```
# Tensor is managed in a contiguous manner through `torch.Storage`
# REGARDLESS OF SHAPE

points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points.storage()
```

Out[51]:

```
 4.0
 1.0
 5.0
 3.0
 2.0
 1.0
[torch.FloatStorage of size 6]
```

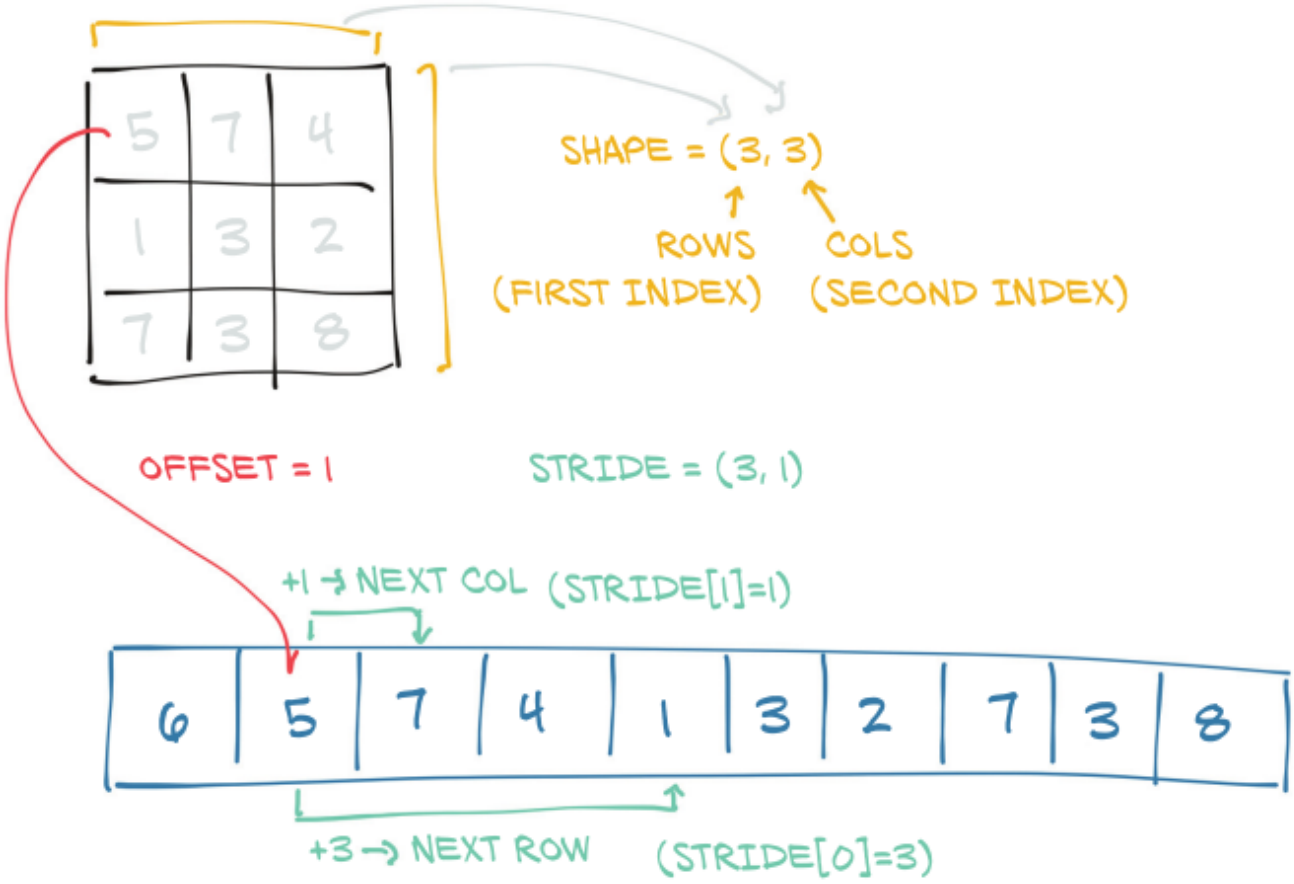
In [54]:

```
# In-place operations
# ALL methods with a trailing underscore ("_") in their name is in place
# Others return a new tensor
a = torch.ones(3,2)

a.zero_()
print(a)
```

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

Tensor metadata: Size, offset and stride

Information needed by tensors to index into a storage



Accessing an element i, j in a 2D tensor results in accessing the `storage_offset + stride[0] * i + stride[1] * j` element in the storage.

- Storage offset ususally 0
- If a tensor is a **view of a storage created to hold a larger tensor**, the offset might be a positive value.

In [60]:

```
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1] # subtensor -> changing subtensor changes original tensor
second_point.storage_offset()
```

Out[60]:

```
2
```

In [56]:

```
second_point.size()
```

Out[56]:

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

In [57]:

```
second_point.shape
```

Out[57]:

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

In [58]:

```
points.stride()
```

Out[58]:

```
(2, 1)
```

In [61]:

```
# changes to subtensor affects the original tensor
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1]
second_point[0] = 10.0
points
```

Out[61]:

```
tensor([[ 4.,  1.],
        [10.,  3.],
        [ 2.,  1.]])
```

In [62]:

```
# ..which might not be desirable, so we can clone the subtensor instead:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1].clone()
second_point[0] = 10.0
points
```

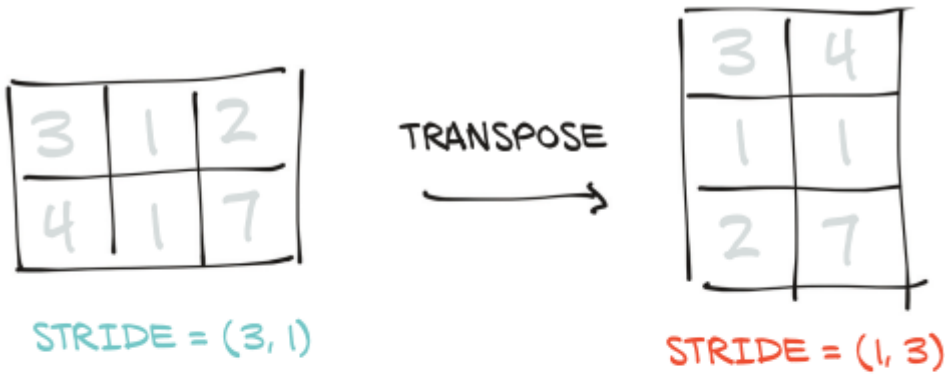
Out[62]:

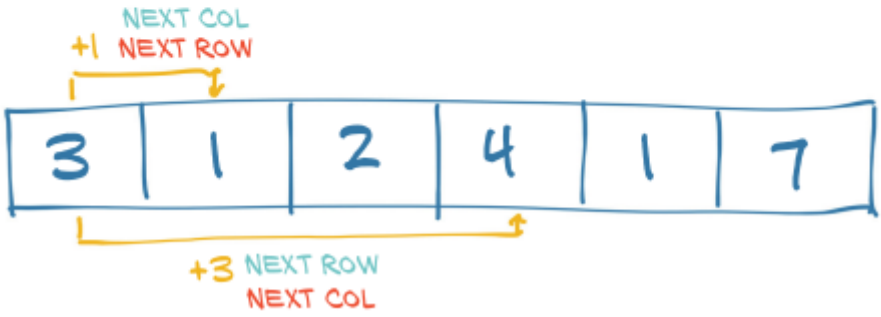
```
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

In [66]:

```
# Transposing
# "t" function : shorthand for transpose for 2D tensors

points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
print(points)

points_t = points.t()
print(points_t)

# Verify that the two sensors share the same storage
print(id(points.storage()) == id(points_t.storage()))
assert(id(points.storage()) == id(points_t.storage()))

# Verify they differ in only shape and stride
print(points.stride(), points_t.stride())
```

```
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
tensor([[4., 5., 2.],
        [1., 3., 1.]])
True
(2, 1) (1, 2)
```





In [85]:

```
# Transposing in higher dimensions
some_t = torch.ones(3, 4, 5)
transpose_t = some_t.transpose(0, 2)
print(some_t.shape, transpose_t.shape)
print(some_t.stride(), transpose_t.stride())

"""
CONTIGUOUS
A tensor whose values are laid out in the storage starting from the rightmost
dimension onward (that is, moving along rows for a 2D tensor) 핵심: "ALONG ROWS"
"""
pass # 이거 안해주면 위의 주석이 print됨
```

```
torch.Size([3, 4, 5]) torch.Size([5, 4, 3])
(20, 5, 1) (1, 5, 20)
```

In [86]:

```
# Some tensor operations only work on contiguous tensors (ex "view")
# points is contiguous, but its transpose is not
print(points.is_contiguous())
print(points_t.is_contiguous())
```

```
True
False
```

In [88]:

```
# can obtain a new contiguous tensor using "contiguous" method
print(points_t.storage())
print(points_t.stride())

points_t_cont = points_t.contiguous()
print(points_t)
print(points_t_cont) # 생김새는 같음
print(points_t_cont.stride())
print(points_t_cont.storage()) # has been reshuffled
```

```
 4.0
 1.0
 5.0
 3.0
 2.0
 1.0
[torch.FloatStorage of size 6]
(1, 2)
tensor([[4., 5., 2.],
        [1., 3., 1.]])
tensor([[4., 5., 2.],
        [1., 3., 1.]])
(3, 1)
 4.0
 5.0
 2.0
 1.0
 3.0
 1.0
[torch.FloatStorage of size 6]
```

Moving tensors to GPU

Every PyTorch tensor can be transferred to (one of) the GPU(s) in order to perform faster computations

The device attribute

```
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cud
a')
# Creates a tensor on GPU
points_gpu = points.to(device='cuda')
# Copies a tensor to GPU
points_gpu = points.to(device='cuda:0')
# When there are multiple GPUs

# Shorthand methods "cpu" and "cuda"
points_gpu = points.cuda() # defaults to GPU index 0
points_gpu = points.cuda(0)
points_cpu = points_gpu.cpu()
```

All further calculations on the tensor are carried out on the GPU

Numpy Interoperability

```
# Torch tensor to numpy
points = torch.ones(3, 4)
points_np = points.numpy()
points_np # returns a numpy array

# numpy to torch tensor
points = torch.from_numpy(points_np)
```

Note: How the data is stored under the hood is SEPARATE from the tensor API discussed above

Any implementation that meets the contract of that API can be considered a tensor

- Gives birth to other types of tensors:
  - Specific to certain devices (like Google TPUs)
  - Sparse Tensors (for sparse entries)
  - Quantized tensors

The usual tensors are called **dense** or **strided** tensors

Serializing (Loading and Saving) Tensors

If the data is valuable, will want to save it to file and load it back at some point.

```
# Saving our "points" tensor to an "ourpoints.t" file
torch.save(points, '../data/p1ch3/ourpoints.t')

# or
with open('../data/p1ch3/ourpoints.t','wb') as f:
    torch.save(points, f)

# Loading back our tensor
points = torch.load('../data/p1ch3/ourpoints.t')

# or
with open('../data/p1ch3/ourpoints.t','rb') as f:
    points = torch.load(f)

# Saving model
checkpoint = {
    'model': model.state_dict()
}
torch.save(checkpoint, os.path.join(dirname, 'model.pt'))

# Loading saved model
checkpoint = torch.load(os.path.join(dirname, 'model.pt'))
model.load_state_dict(checkpoint['model'])
```

Serializing to HDF5 with h5py

Need to save tensors interoperably when introducing PyTorch into existing systems

HDF5

- portable, widely supported format for representing serialized multidim arrays
- organized in a nested key-value dictionary
- Python support through `h5py` library (Link (www.h5py.org))

```
# Then can save by converting to Numpy array
import h5py
f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'w')
dset = f.create_dataset('coords', data=points.numpy()) #coords is just a ke
y into the HDF5 file
f.close()

# Can have other keys
# Can also have nested keys

# Loading just the last two points in our dataset:
f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'r')
dset = f['coords']
last_points = dset[-2:] # data stays on disk (NOT loaded into memory) until
this is called

# can do this straightaway
last_points = torch.from_numpy(dset[-2:])
f.close()
```