UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE of ENGINEERING

Introduction to ML
Lecture-15: Pytroch and Tensors

Hamed Tabkhi

Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

Deep learning competitive landscape

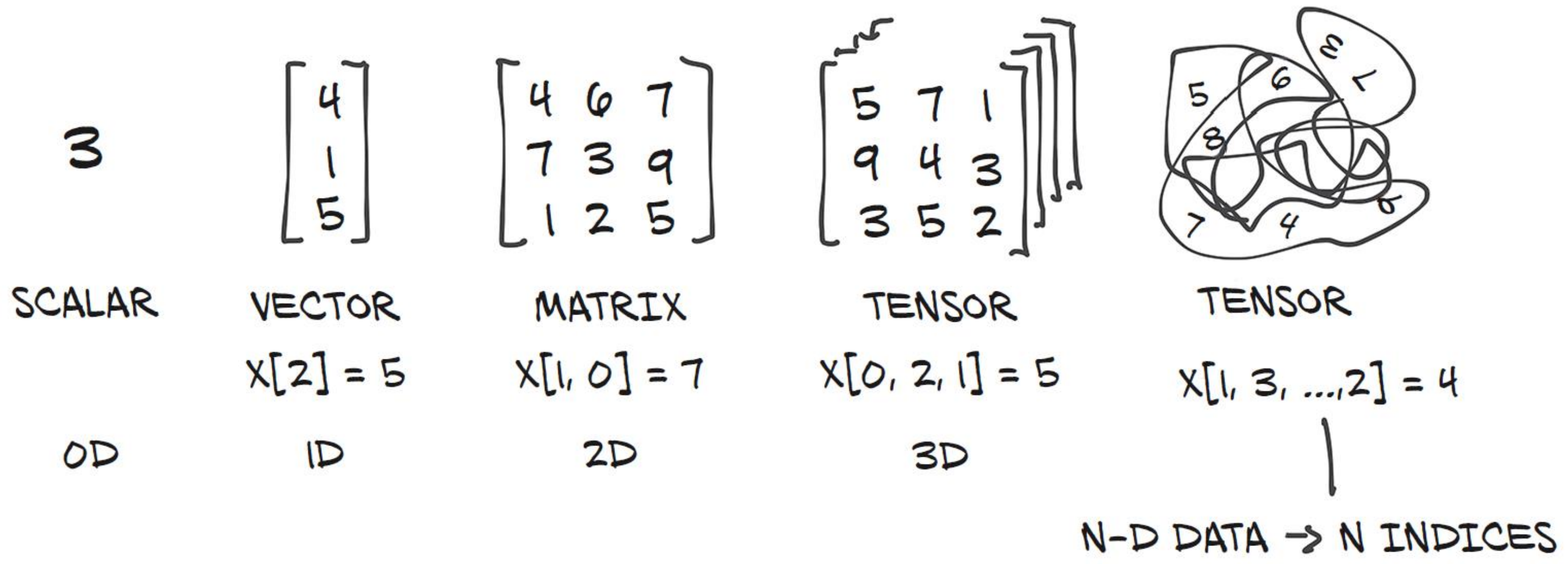
TensorFlow:
– Consumed Keras entirely, promoting it to a first-class API
– Provided an immediate-execution "eager mode" that is somewhat similar to how PyTorch approaches computation
– Released TF 2.0 with eager mode by default

PyTorch:
– Consumed Caffe2 for its backend
– Replaced most of the low-level code reused from the Torch project
– Added support for ONNX, a vendor-neutral model description and exchange format
– Added a delayed-execution "graph mode" runtime called *TorchScript*

PyTorch for deep learning

- PyTorch is a library for Python programs that facilitates building deep learning projects.
- It emphasizes flexibility and allows deep learning models to be expressed in idiomatic Python.
- This approachability and ease of use found early adopters in the research community, and in the years since its first release.
- It has grown into one of the most prominent deep learning tools across a broad range of applications.
- It provides accelerated computation using graphical processing units (GPUs).
- PyTorch provides facilities that support numerical optimization on generic mathematical expressions, which deep learning uses for training.
- PyTorch has been equipped with a high-performance C++ runtime that can be used to deploy models for inference without relying on Python, and can be used for designing and training models in C++.

What is a Tensor



- In the context of deep learning, tensors refer to the generalization of vectors and matrices to an arbitrary number of dimensions.
- Another name for the same concept is ***multidimensional*** *array*.

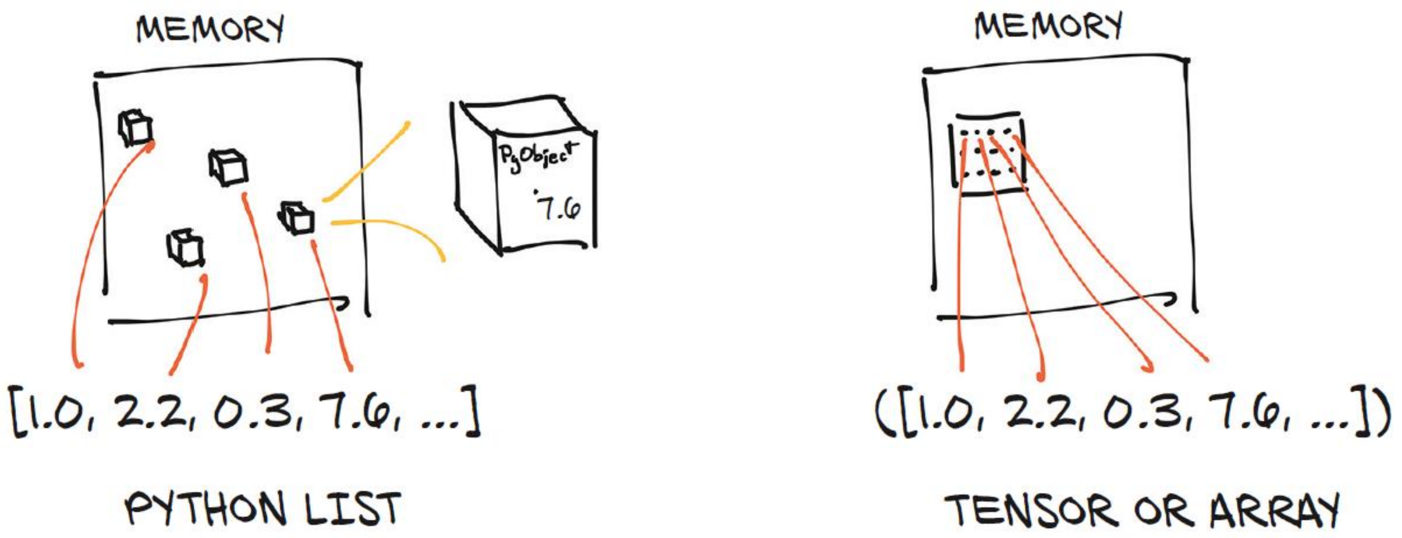
Pytorch Tensor and NumPy

- PyTorch is not the only library that deals with multidimensional arrays.
- NumPy (https://numpy.org/) is by far the most popular multidimensional array library, it has now arguably become ***the lingua franca*** of data science.
- PyTorch features seamless interoperability with NumPy, which brings with it first-class integration with the rest of the scientific libraries in Python, such as SciPy (www.scipy.org), Scikit-learn (https://scikit-learn.org), and Pandas (https://pandas.pydata.org).

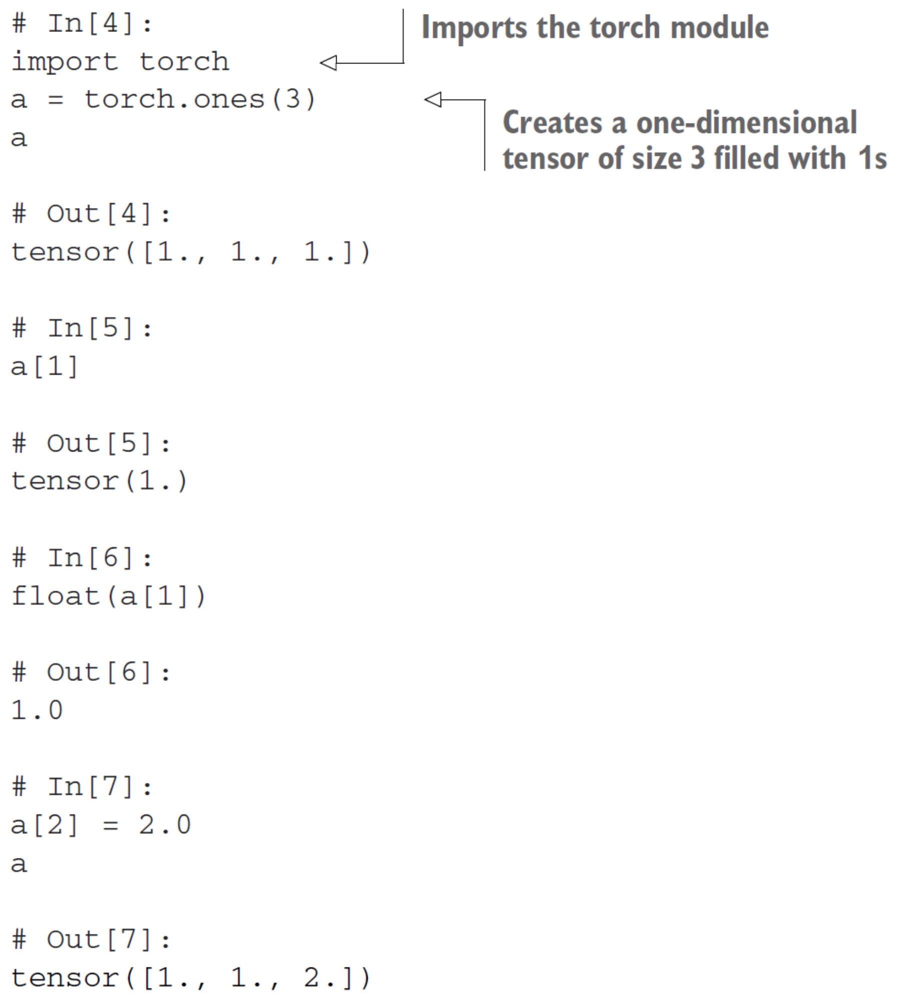
- Compared to NumPy arrays, PyTorch tensors have a few superpowers:
  1. Ability to perform very fast operations on graphical processing units (GPUs)
  2. Distribute operations on multiple devices or machines
  3. Keep track of the graph of computations

The essence of Tensors

- Python lists are collections of Python objects that are individually allocated in memory.
- PyTorch tensors or NumPy arrays, on the other hand, are views over (typically) contiguous memory blocks containing *unboxed* C numeric types rather than Python objects.



Constructing our first Tensor



```
# In[4]:
import torch
a = torch.ones(3)
a

# Out[4]:
tensor([1., 1., 1.])

# In[5]:
a[1]

# Out[5]:
tensor(1.)

# In[6]:
float(a[1])

# Out[6]:
1.0

# In[7]:
a[2] = 2.0
a

# Out[7]:
tensor([1., 1., 2.])
```

Imports the torch module

Creates a one-dimensional tensor of size 3 filled with 1s

- After importing the `torch` module, we call a function that creates a (one-dimensional) tensor of size 3 filled with the value `1.0`. We can access an element using its zero-based index or assign a new value to it.

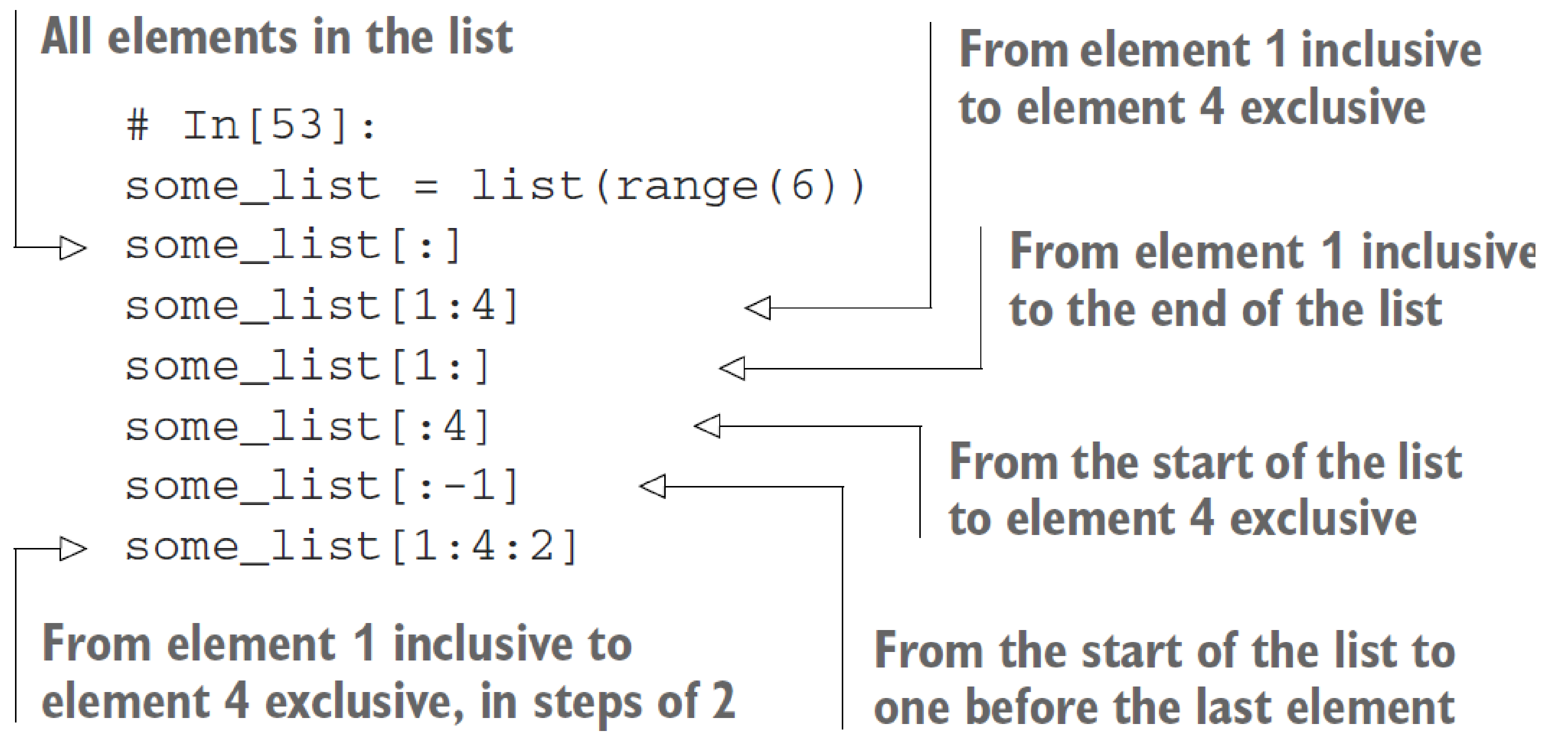
Another example

```
# In[14]:
    points = torch.zeros(3, 2)
    Points

# Out[13]:
    tensor([[0., 0.],
    [0., 0.],
    [0., 0.]])

# In[14]:
    points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0,
    1.0]])
    Points

# Out[14]:
    tensor([[4., 1.],
    [5., 3.],
    [2., 1.]])
```

```
# In[15]:
    points[0, 1]

# Out[15]:
    tensor(1.)

# In[16]:
    points[0]

# Out[16]:
    tensor([4., 1.])
```

Range Indexing

Indexing in List in Python: Granularity is elements

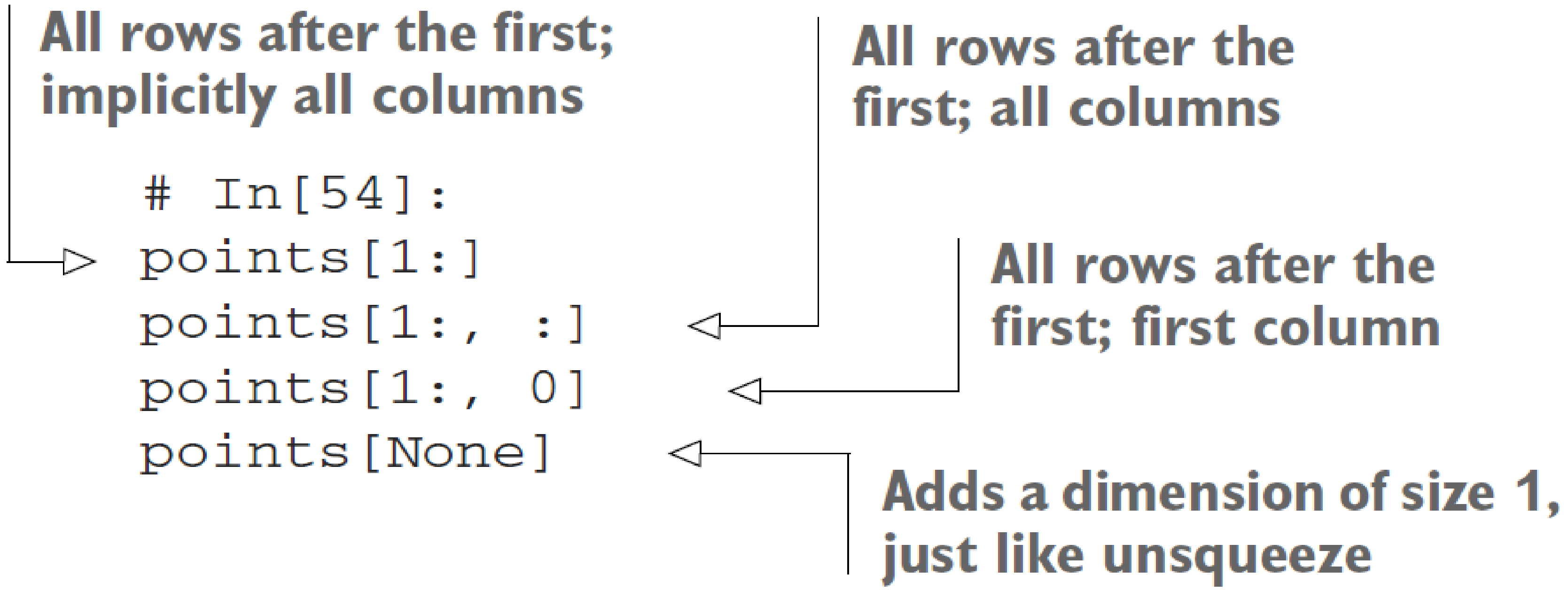


```
# In[53]:
some_list = list(range(6))
some_list[:]
some_list[1:4]
some_list[1:]
some_list[:4]
some_list[:-1]
some_list[1:4:2]
```

All elements in the list
From element 1 inclusive to element 4 exclusive
From element 1 inclusive to the end of the list
From the start of the list to element 4 exclusive
From the start of the list to one before the last element
From element 1 inclusive to element 4 exclusive, in steps of 2

Range Indexing

Indexing Tensor in Pytorch: Granularity is Column and Row



All rows after the first; implicitly all columns

All rows after the first; all columns

All rows after the first; first column

Adds a dimension of size 1, just like unsqueeze

```
# In[54]:
points[1:]
points[1:, :]
points[1:, 0]
points[None]
```

The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Tensors and Mean

- imagine that we have a 3D tensor like `img_t and` and we want to convert it to grayscale

```
img_t = torch.randn(3, 5, 5) # shape [channels, rows, columns]
```

https://pytorch.org/docs/stable/generated/torch.randn.html

- here we have a batch of 2:

```
batch_t = torch.randn(2, 3, 5, 5) # shape [batch, channels, rows, columns]
```

Tensors and Mean

- The lazy, unweighted mean can thus be written as follows:

```
img_gray_naive = img_t.mean(-3)
batch_gray_naive = batch_t.mean(-3)
img_gray_naive.shape, batch_gray_naive.shape


# Out[4]:
(torch.Size([5, 5]), torch.Size([2, 5, 5]))
```

- So sometimes the RGB channels are in dimension 0, and sometimes they are in dimension 1. But we can generalize by counting from the end: they are always in dimension –3, the third from the end.

  https://pytorch.org/docs/stable/generated/torch.mean.html

Tensors and Weighted Mean

- PyTorch will allow us to multiply things that are the same shape, as well as shapes where one operand is of size 1 in a given dimension.
- It also appends leading dimensions of size 1 automatically. This is a feature called *broadcasting*.
- `Example: batch_t` of shape (2, 3, 5, 5) is multiplied by `unsqueezed_weights` of shape (3,1, 1), resulting in a tensor of shape (2, 3, 5, 5), from which we can then sum the third dimension from the end (the three channels):

```
weights = torch.tensor([0.2126, 0.7152, 0.0722])
unsqueezed_weights = weights.unsqueeze(-1).unsqueeze_(-1)
unsqueezed_weights.shape
#Out [5]:
torch.Size([3, 1, 1])
```

https://pytorch.org/docs/stable/generated/torch.unsqueeze.html

Tensors and Weighted Mean

```
img_weights = (img_t * unsqueezed_weights)
batch_weights = (batch_t * unsqueezed_weights)
img_gray_weighted = img_weights.sum(-3)
batch_gray_weighted = batch_weights.sum(-3)
batch_weights.shape, batch_t.shape, unsqueezed_weights.shape
# Out[6]:
(torch.Size([2, 3, 5, 5]), torch.Size([2, 3, 5, 5]), torch.Size([3, 1,
1]))
```

Pytorch Broadcasting:
https://pytorch.org/docs/stable/notes/broadcasting.html

Data Types for Tensors

`torch.float32` or `torch.float`: 32-bit floating-point

`torch.float64` or `torch.double`: 64-bit, double-precision floating-point

`torch.float16` or `torch.half`: 16-bit, half-precision floating-point

`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

Data Types for Tensors

- Computations happening in neural networks are typically executed with 32-bit floating-point precision.
- Higher precision, like 64-bit, will not buy improvements in the accuracy of a model and will require more memory and computing time.
- The 16-bit floating-point, half-precision data type is not present natively in standard CPUs, but it is offered on modern GPUs.
- It is possible to switch to half-precision to decrease the footprint of a neural network model if needed, with a minor impact on accuracy.

Data Types' attributes

- we can specify the proper `dtype` as an argument to the constructor.

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

- We can find out about the `dtype` for a tensor by accessing the corresponding attribute:

```
short_points.dtype
torch.int16
```

- We can find out about the `dtype` for a tensor by accessing the corresponding attribute:
- `short_points.dtype`
- `torch.int16`

Data Types' attributes

- We can also cast the output of a tensor creation function to the right type using the corresponding casting method, such as

```
double_points = torch.zeros(10, 2).double()

short_points = torch.ones(10, 2).short()
```

- or:

```
double_points = torch.zeros(10, 2).to(torch.double)

short_points = torch.ones(10, 2).to(dtype=torch.short)
```

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Tensors’ view of Storage

- # In[17]:

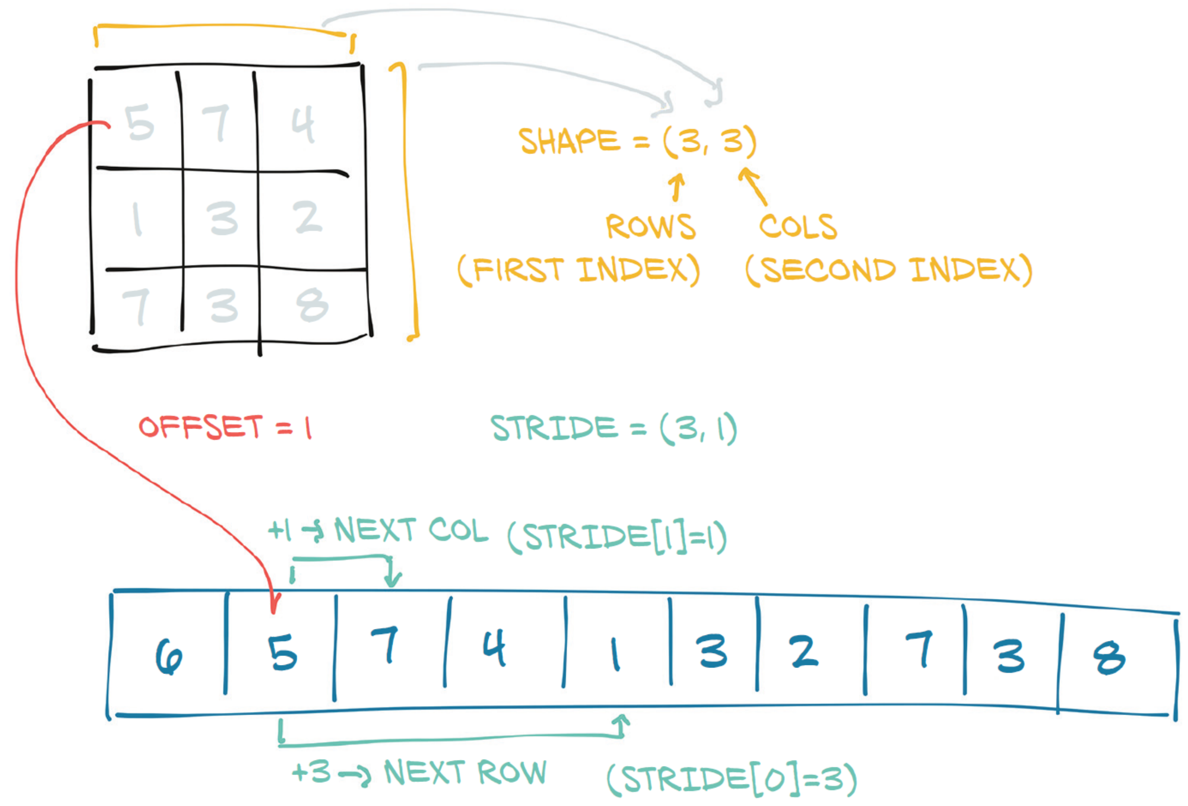
```
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points.storage()
```

- # Out[17]:

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

Tensors Offset and Stride

- The storage offset is the index in the storage corresponding to the first element in the tensor.
- The stride is the number of elements in the storage that need to be skipped over to obtain the next element along each dimension.



The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Tensors Offset

```
# In[21]:
    points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
    second_point = points[1]
    second_point.storage_offset()
# Out[21]:
    2
```

- The resulting tensor has offset 2 in the storage (since we need to skip the first point, which has two items).

```
#In[24]:
    points.stride()
# Out[24]:
    (2, 1)
```

- The stride is a tuple indicating the number of elements in the storage that have to be skipped when the index is increased by 1 in each dimension.

Tensor APIs

- *Creation ops*—Functions for constructing a tensor, like `ones` and `from_numpy`
- *Indexing, slicing, joining, mutating ops*—Functions for changing the shape, stride, or content of a tensor, like `transpose`
- *Random sampling*—Functions for generating values by drawing randomly from probability distributions, like `randn` and `normal`
- *Serialization*—Functions for saving and loading tensors, like `load` and `save`
- *Parallelism*—Functions for controlling the number of threads for parallel CPU execution, like `set_num_threads`

The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Tensor APIs Math Operations

Math ops are Functions for manipulating the content of the tensor through computations:

- *Pointwise ops*—Functions for obtaining a new tensor by applying a function to each element independently, like `abs` and `cos`
- *Reduction ops*—Functions for computing aggregate values by iterating through tensors, like `mean`, `std`, and `norm`
- *Comparison ops*—Functions for evaluating numerical predicates over tensors, like `equal` and `max`
- *Spectral ops*—Functions for transforming in and operating in the frequency domain, like `stft` and `hamming_window`
- *Other operations*—Special functions operating on vectors, like `cross`, or matrices, like `trace`
- *BLAS and LAPACK operations*—Functions following the Basic Linear Algebra Subprograms (BLAS) specification for scalar, vector-vector, matrix-vector, and matrix-matrix operations

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Tensor APIs

- At this point, we know what PyTorch tensors are and how they work under the hood.
- The vast majority of operations on and between tensors are available in the `torch` module

```
# In[71]:
    a = torch.ones(3, 2)
    a_t = torch.transpose(a, 0, 1)
    a.shape, a_t.shape

# Out[71]:
    (torch.Size([3, 2]), torch.Size([2, 3]))
```

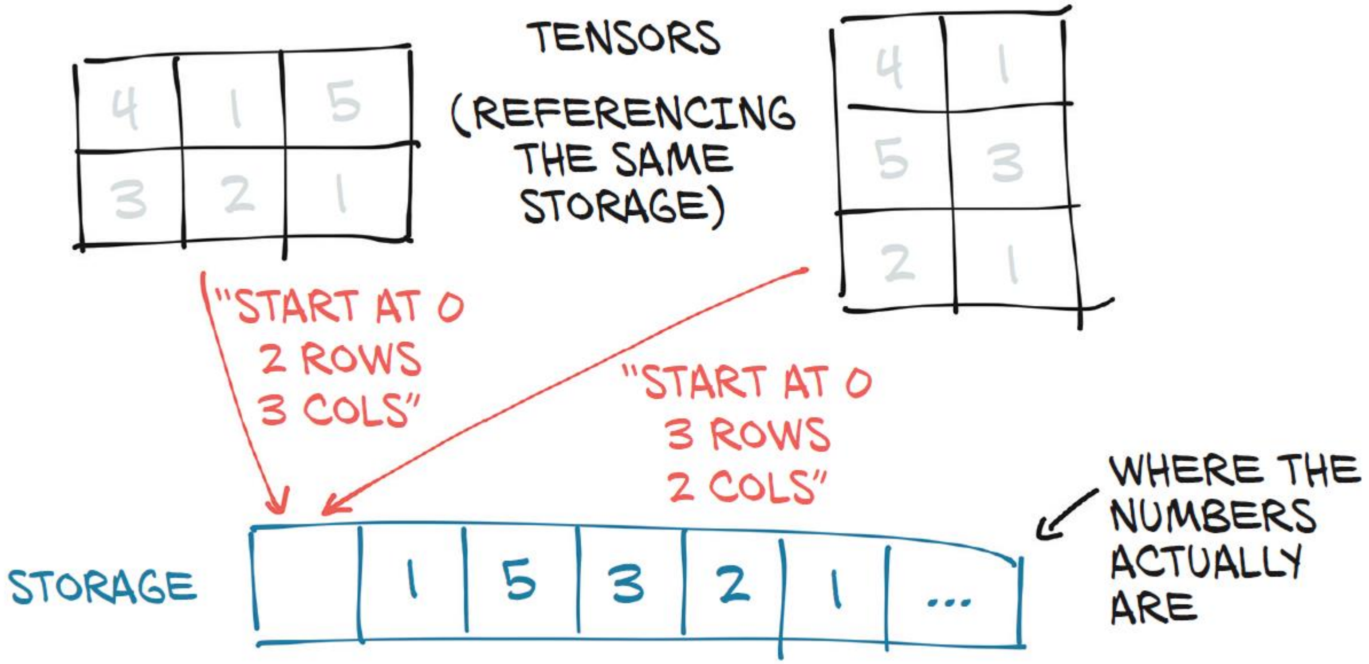
- or as a method of the `a` tensor:

```
# In[72]:
    a = torch.ones(3, 2)
    a_t = a.transpose(0, 1)
    a.shape, a_t.shape

# Out[72]:
    (torch.Size([3, 2]), torch.Size([2, 3]))
```

See here: https://pytorch.org/docs/stable/generated/torch.transpose.html?highlight=transpose#torch.transpose

Tensors' view of Storage

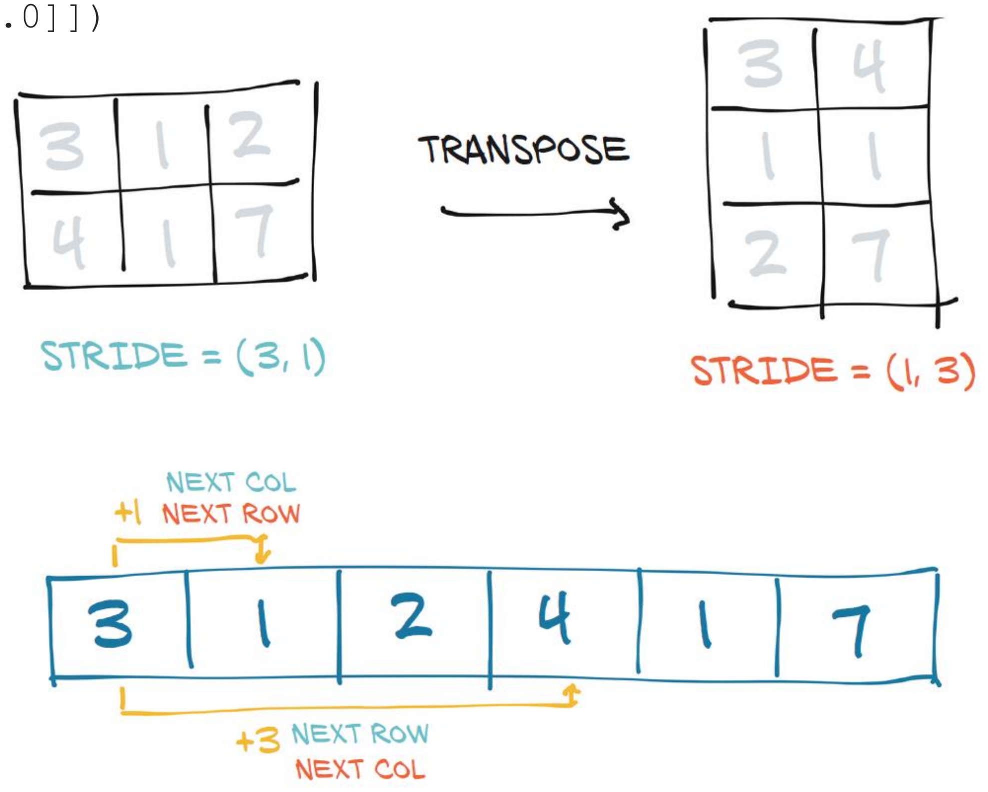
- Values in tensors are allocated in contiguous chunks of memory managed by `torch.Storage` instances.
- A storage is a one-dimensional array of numerical data: that is, a contiguous block of memory containing numbers of a given type
- Multiple tensors can index the same storage even if they index into the data differently.
- The layout of a storage is always one-dimensional, regardless of the dimensionality of any and all tensors that might refer to it.



The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Example: Transposing without Copying

```
# In[30]:
        points = torch.tensor([[3.0, 1.0, 2.0], [4.0, 1.0, 7.0]])
# In[31]:
        points_t = points.t()
        id(points.storage()) == id(points_t.storage())
# Out[32]:
        True#
In[33]:
        points.stride()
# Out[33]:
        (3, 1)
# In[34]:
        points_t.stride()
# Out[34]:
        (1, 3)
```



In our case, `points` is contiguous, while its transpose is not

https://pytorch.org/docs/stable/generated/torch.t.html

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Another Example

- # In[30]:
```
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points_t = points.t()
points.stride()
```
- # Out[33]:
```
(2, 1)
```
- # In[34]:
```
points_t.stride()
```
- # Out[34]:
```
(1, 2)
```

- # In[39]:
```
points.is_contiguous()
```
- # Out[39]:
```
True
```
- # In[40]:
```
points_t.is_contiguous()
```
- # Out[40]:
```
False
```

In our case, `points` is contiguous, while its transpose is not

The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Example: Transposing higher dimensions

- Transposing in PyTorch is not limited to matrices. We can transpose a multidimensional array by specifying the two dimensions along which transposing (flipping shape and stride) should occur:

```
# In[35]:
       some_t = torch.ones(3, 4, 5)
       transpose_t = some_t.transpose(0, 2)
       some_t.shape
# Out[35]:
       torch.Size([3, 4, 5])
# In[36]:
       transpose_t.shape
# Out[36]:
       torch.Size([5, 4, 3])
# In[37]:
       some_t.stride()
# Out[37]:
       (20, 5, 1)
# In[38]:
       transpose_t.stride()
# Out[38]:
       (1, 5, 20)
```

Contagious Tensors

- Some tensor operations in PyTorch only work on contiguous tensors, such as `view`.
- In that case, PyTorch will throw an informative exception and require us to call `contiguous` explicitly. It's worth noting that calling `contiguous` will do nothing (and will not hurt performance) if the tensor is already contiguous.

```
points_t_cont = points_t.contiguous()
points_t_cont
```

- `# Out[44]:`

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

- `# In[45]:`

```
points_t_cont.stride()
```

- `# Out[45]:`

```
(3, 1)
```

The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

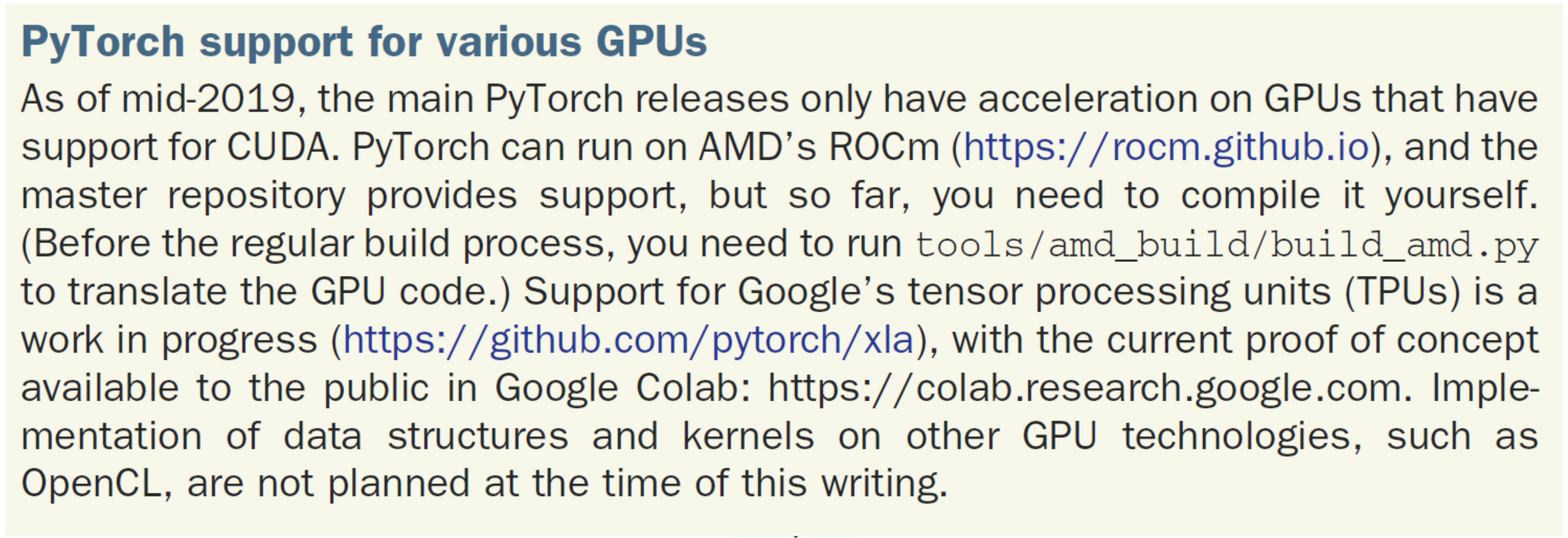
Contagious Tensors

- `# In[46]:`
  ```
  points_t_cont.storage()
  ```
- `# Out[46]:`
  ```
  4.0
  5.0
  2.0
  1.0
  3.0
  1.0
  [torch.FloatStorage of size 6]
  ```

- Notice that the storage has been reshuffled in order for elements to be laid out row-by-row in the new storage.
- The stride has been changed to reflect the new layout.

GPU-Specific Runtime for Handling Tensors

- PyTorch tensors also can be stored on a different kind of processor: a graphics processing unit (GPU).
- Every PyTorch tensor can be transferred to (one of) the GPU(s) in order to perform massively parallel, fast computations.
- All operations that will be performed on the tensor will be carried out using GPU-specific routines that come with PyTorch.



PyTorch support for various GPUs

As of mid-2019, the main PyTorch releases only have acceleration on GPUs that have support for CUDA. PyTorch can run on AMD's ROCm (https://rocm.github.io), and the master repository provides support, but so far, you need to compile it yourself. (Before the regular build process, you need to run `tools/amd_build/build_amd.py` to translate the GPU code.) Support for Google's tensor processing units (TPUs) is a work in progress (https://github.com/pytorch/xla), with the current proof of concept available to the public in Google Colab: https://colab.research.google.com. Implementation of data structures and kernels on other GPU technologies, such as OpenCL, are not planned at the time of this writing.

Tensors GPU Attribute

- We create a new tensor that has the same numerical data, but stored in the RAM of the GPU, rather than in regular system RAM.

```
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda’)
```

- We could instead copy a tensor created on the CPU onto the GPU using the `to` method:

```
      points_gpu = points.to(device='cuda’)
```

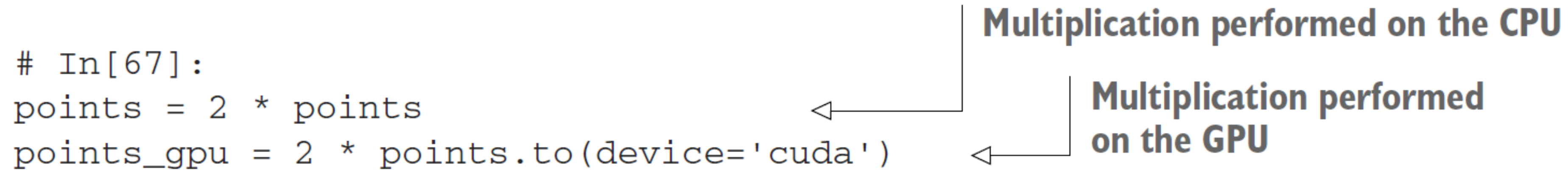
- Now that the data is stored locally on the GPU, we’ll start to see the speedups mentioned earlier when performing mathematical operations on the tensor.

Tensors GPU Attribute

- At this point, any operation performed on the tensor, such as multiplying all elements by a constant, is carried out on the GPU.
- In almost all cases, CPU- and GPU-based tensors expose the same user-facing API, making it much easier to write code that is agnostic to where, exactly, the heavy number crunching is running.
- If our machine has more than one GPU, we can also decide on which GPU we allocate the tensor by passing a zero-based integer identifying the GPU on the machine.

```
points_gpu = points.to(device='cuda:0')
```

Tensors GPU Attribute



```
# In[67]:
points = 2 * points
points_gpu = 2 * points.to(device='cuda')
```

Multiplication performed on the CPU

Multiplication performed on the GPU

- Note that the `points_gpu` tensor is not brought back to the CPU once the result has been computed.
- Here's what happened in this line:
  1 The `points` tensor is copied to the GPU.
  2 A new tensor is allocated on the GPU and used to store the result of the multiplication.
  3 A handle to that GPU tensor is returned.
- Therefore, if we also add a constant to the result

```
points_gpu = points_gpu + 4
```

- The addition is still performed on the GPU, and no information flows to the CPU (unless we print or access the resulting tensor).

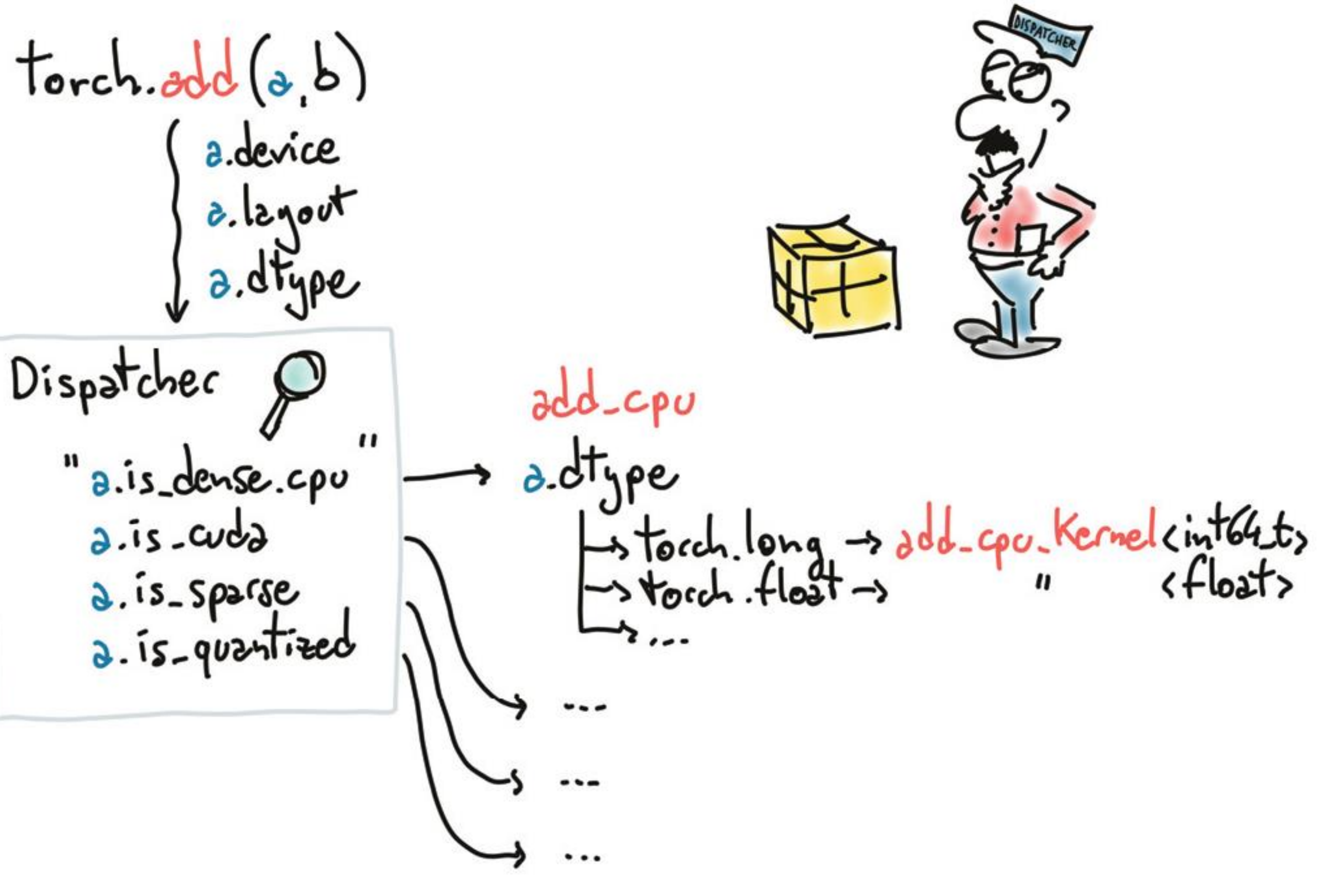
Tensors GPU Attribute

- In order to move the tensor back to the CPU, we need to provide a `cpu` argument to the `to` method, `points_cpu = points_gpu.to(device='cpu')`
- We can also use the shorthand methods `cpu` and `cuda` instead of the `to` method to
- Achieve the same goal:

```
points_gpu = points.cuda()
points_gpu = points.cuda(0)
points_cpu = points_gpu.cpu()
```

- It's also worth mentioning that by using the `to` method, we can change the placement and the data type simultaneously by providing both `device` and `dtype` as arguments.

PyTorch dispatcher



The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

NumPy Interoperability

- Pytorch offers zero-copy interoperability with NumPy arrays
- NumPy is ubiquity in the Python data science ecosystem.
- NumPy due to its ubiquity in the Python data science ecosystem.
- We can take advantage of the huge swath of functionality in the wider Python ecosystem that has built up around the NumPy array type.

NumPy Interoperability

```
# In[54]:
    points = torch.ones(3, 4)
    points_np = points.numpy()
    points_np
```

-
  ```
  # Out[55]:
    array([[1., 1., 1., 1.],
    [1., 1., 1., 1.],
    [1., 1., 1., 1.]], dtype=float32)
  ```
- It will return a NumPy multidimensional array of the right size, shape, and numerical type.
- Interestingly, the returned array shares the same underlying buffer with the tensor storage.
- This means the `numpy` method can be effectively executed at basically no cost, as long as the data sits in CPU RAM.
- **It also means modifying the NumPy array will lead to a change in the originating tensor.**
- **If the tensor is allocated on the GPU, PyTorch will make a copy of the content of the tensor into a NumPy array allocated on the CPU.**

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

NumPy Interoperability

- Conversely, we can obtain a PyTorch tensor from a NumPy array this way

```
points = torch.from_numpy(points_np)
```

- It will use the same buffer-sharing strategy we just described.

Note: While the default numeric type in PyTorch is 32-bit floating-point, for NumPy it is 64-bit. We usually want to use 32-bit floating-points, so we need to make sure we have tensors of `dtype torch .float` after converting.

Saving Tensors to a File

- Creating a tensor on the fly is all well and good, but if the data inside is valuable, we will want to save it to a file and load it back at some point.
- We don't want to have to retrain a model from scratch every time we start running our program!
- PyTorch uses `pickle` under the hood to serialize the tensor object, plus dedicated serialization code for the storage.

```
# In[57]:
  torch.save(points, '../data/p1ch3/ourpoints.t')
```

- As an alternative, we can pass a file descriptor in instead of the filename:

```
# In[58]:
  with open('../data/p1ch3/ourpoints.t','wb') as f:
  torch.save(points, f)
```

Loading from a file to a Tensor

- Loading our points back is similarly a one-liner
- `# In[59]:`

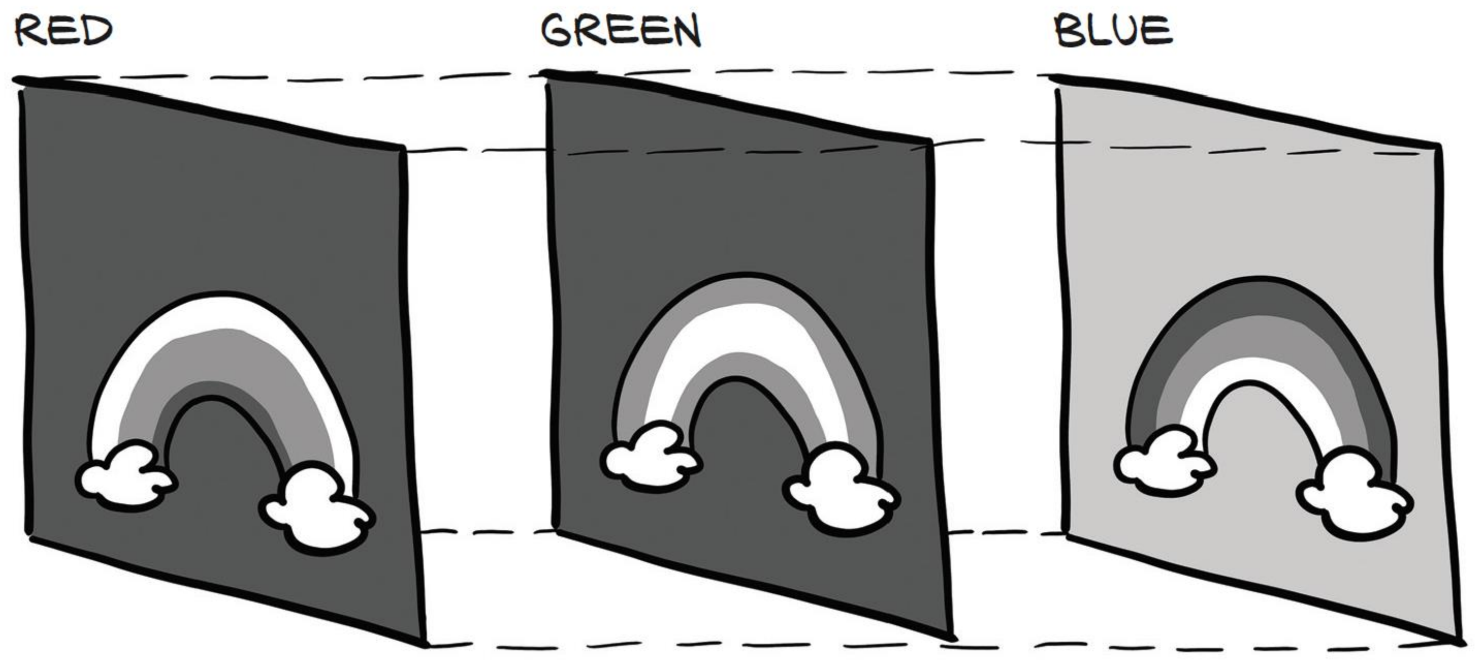
```
points = torch.load('../data/p1ch3/ourpoints.t’)
```

- or, equivalently,

```
# In[60]:
with open('../data/p1ch3/ourpoints.t','rb') as f:
points = torch.load(f)
```

World as a Floating-Point Number

Our first job as deep learning practitioners is to encode heterogeneous, real-world data into a tensor of floating-point numbers, ready for consumption by a neural network.



Loading Images

- There are plenty of ways to load images in Python

```
# In[2]:
import imageio
img_arr = imageio.imread('../data/p1ch4/image-dog/bobby.jpg')
img_arr.shape

# Out[2]:
(720, 1280, 3)
```

- At this point, `img` is a NumPy array-like object with three dimensions: two spatial
- dimensions, width and height; and a third dimension corresponding to the red,
- green, and blue channels.

`Note:` TorchVision is another module in Pytorch for interacting with images

Changing the Layout

- We can use the tensor’s `permute` method with the old dimensions for each new dimension to get to an appropriate layout.
- Given an input tensor $H \times W \times C$ as obtained previously, we get a proper layout by having channel 2 first and then channels 0 and 1:

$H \times W \times C$ -> $C \times H \times W$

```
# In[3]:
img = torch.from_numpy(img_arr)
out = img.permute(2, 0, 1)
```

NOTE: Note that this operation does not make a copy of the tensor data. Instead, `out` uses the same underlying storage as `img` and only plays with the size and stride information at the tensor level.

Changing the Layout

- To create a dataset of multiple images to use as an input for our neural networks, we store the images in a batch along the first dimension to obtain an $N \times C \times H \times W$ tensor.

```
# In[4]:

batch_size = 3

batch = torch.zeros(batch_size, 3, 256, 256, dtype=torch.uint8)
```

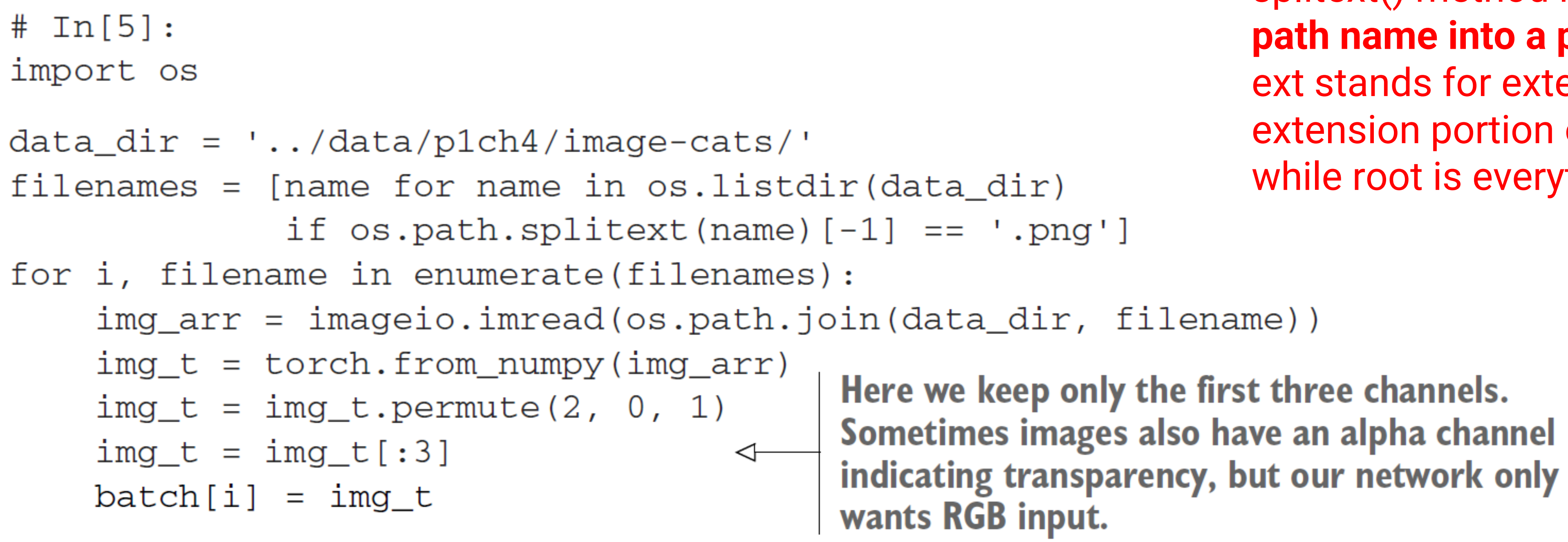
- This indicates that our batch will consist of three RGB images 256 pixels in height and 256 pixels in width.
- Notice the type of the tensor: we're expecting each color to be represented as an 8-bit integer, as in most photographic formats from standard consumer cameras.

Changing the Layout

- We can now load all PNG images from an input directory and store them in the tensor:

enumerate() is a built-in Python function. The enumerate() function **allows you to loop over an iterable object and keep track of how many iterations have occurred**.

splitext() method is **used to split the path name into a pair root and ext**. Here, ext stands for extension and has the extension portion of the specified path while root is everything except ext part.



```
# In[5]:
import os

data_dir = '../data/p1ch4/image-cats/'
filenames = [name for name in os.listdir(data_dir)
                if os.path.splitext(name)[-1] == '.png']
for i, filename in enumerate(filenames):
    img_arr = imageio.imread(os.path.join(data_dir, filename))
    img_t = torch.from_numpy(img_arr)
    img_t = img_t.permute(2, 0, 1)
    img_t = img_t[:3]
    batch[i] = img_t
```

Here we keep only the first three channels. Sometimes images also have an alpha channel indicating transparency, but our network only wants RGB input.

OS— Miscellaneous operating system interfaces
https://docs.python.org/3/library/os.html

The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Normalizing the Data

- Neural networks exhibit the best training performance when the input data ranges roughly from 0 to 1, or from -1 to 1 (this is an effect of how their building blocks are defined).
- We need to cast a tensor to floating-point and normalize the values of the pixels.
- Casting to floating-point is easy, but normalization is trickier, as it depends on what range of the input, we decide should lie between 0 and 1 (or -1 and 1).

- One possibility is to just divide the values of the pixels by 255 (the maximum representable number in 8-bit unsigned):

```
# In[6]:

batch = batch.float()

batch /= 255.0
```

Normalizing the Data

- Another possibility is to compute the mean and standard deviation of the input data and scale it so that the output has zero mean and unit standard deviation across each channel:

```
# In[7]:
n_channels = batch.shape[1]
for c in range(n_channels):
mean = torch.mean(batch[:, c])
std = torch.std(batch[:, c])
batch[:, c] = (batch[:, c] - mean) / std
```

NOTE: Here, we normalize just a single batch of images.

It is a good practice to compute the mean and standard deviation on all the training data in advance

Tabular Data

- The simplest form of data we'll encounter on a machine learning job is sitting in a spreadsheet, CSV file, or database
  - There's no meaning to the order in which samples appear in the table: such a table is a collection of independent samples.
  - Time series, in which samples are related by a time dimension.
- Tabular data is typically not homogeneous: different columns don't have the same type.
  - We might have a column showing the weight of apples and another encoding their color in a label.
- PyTorch tensors, on the other hand, are homogeneous.
  - Information in PyTorch is typically encoded as a number, typically floating-point (though integer types and Boolean are supported as well).
  - This numeric encoding is deliberate, since neural networks are mathematical entities that take real numbers

Numeral Values

1. Continues Values
2. Ordinal Values
3. Categorical Values

Continues Values

- These are the most intuitive when represented as numbers.
- They are strictly ordered, and a difference between various values has a strict meaning.
  - Stating that package A is 2 kilograms heavier than package B
  - Package B came from 100 miles farther away than A
- If you're counting or measuring something with units, it's probably a continuous value.
- The literature divides continuous values further:
  - It makes sense to say something is twice as heavy or three times farther away, so those values are said to be on a **ratio scale**.
  - The time of day, on the other hand, does have the notion of difference, but it is not reasonable to claim that 6:00 is twice as late as 3:00; so time of day only offers an **interval scale**.

Ordinal Values

- The strict ordering, we have with continuous values remains, but the fixed relationship between values no longer applies.
  - A good example of this is ordering a small, medium, or large drink, with small mapped to the value 1, medium 2, and large 3. The large drink is bigger than the medium, in the same way that 3 is bigger than 2, but it doesn’t tell us anything about *how much* bigger.
  - If we were to convert our 1, 2, and 3 to the actual volumes (say, 8, 12, and 24 fluid ounces), then they would switch to being contentious values.
- It’s important to remember that we can’t “do math” on the values outside of ordering them; trying to average large = 3 and small = 1 does *not* result in a medium drink!

Categorical Values

- Have neither ordering nor numerical meaning to their values.
- These are often just enumerations of possibilities assigned arbitrary numbers.
- Assigning water to 1, coffee to 2, soda to 3, and milk to 4 is a good example.
- There’s no real logic to placing water first and milk last; they simply need distinct values to differentiate them.
- We could assign coffee to 10 and milk to –3, and there would be no significant change.

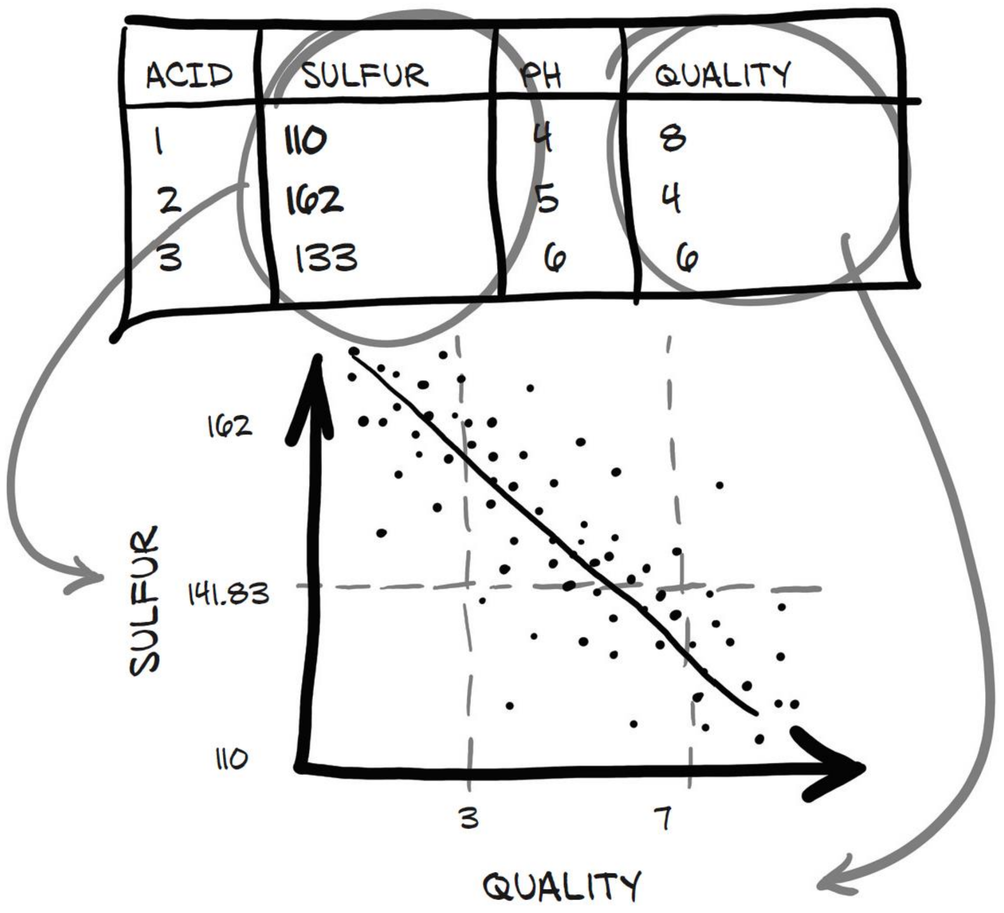
Presenting Tabular Data

- The dataset for white wines can be downloaded here: http://mng.bz/90Ol.
- A copy of the dataset on the Deep Learning with PyTorch Git repository, under data/p1ch4/tabular-wine.

- The file contains a comma-separated collection of values organized in 12 columns preceded by a header line containing the column names.
- The first 11 columns contain values of chemical variables, and the last column contains the sensory quality score from 0 (very bad) to 10 (excellent).

A possible machine learning task on this dataset is predicting the quality score from chemical characterization alone.

Wine Dataset



ACID	SULFUR	PH	QUALITY
1	110	4	8
2	162	5	4
3	133	6	6

Loading a Wine Data Tensor

- we can load the CSV file (containing Wine Dataset) a using Python and then turn it into a PyTorch tensor.

```
# In[2]:
import csv
wine_path = "../data/p1ch4/tabular-wine/winequality-white.csv"
wineq_numpy = np.loadtxt(wine_path, dtype=np.float32, delimiter=";",
                         skiprows=1)
wineq_numpy

# Out[2]:
array([[ 7.  ,  0.27,  0.36, ...,  0.45,  8.8 ,  6.  ],
       [ 6.3 ,  0.3 ,  0.34, ...,  0.49,  9.5 ,  6.  ],
       [ 8.1 ,  0.28,  0.4 , ...,  0.44, 10.1 ,  6.  ],
       ...,
       [ 6.5 ,  0.24,  0.19, ...,  0.46,  9.4 ,  6.  ],
       [ 5.5 ,  0.29,  0.3 , ...,  0.38, 12.8 ,  7.  ],
       [ 6.  ,  0.21,  0.38, ...,  0.32, 11.8 ,  6.  ]], dtype=float32)
```

Loading a Wine Data Tensor

and proceed to convert the NumPy array to a PyTorch tensor:

```
# In[4]:
wineq = torch.from_numpy(wineq_numpy)

wineq.shape, wineq.dtype

# Out[4]:
(torch.Size([4898, 12]), torch.float32)
```

Representing the Scores

```
# In[5]:
data = wineq[:, :-1]
data, data.shape
```

Selects all rows and all columns except the last

```
# Out[5]:
(tensor([[ 7.00,  0.27,  ...,  0.45,  8.80],
         [ 6.30,  0.30,  ...,  0.49,  9.50],
         ...,
         [ 5.50,  0.29,  ...,  0.38, 12.80],
         [ 6.00,  0.21,  ...,  0.32, 11.80]]), torch.Size([4898, 11]))

# In[6]:
target = wineq[:, -1]
target, target.shape
```

Selects all rows and the last column

```
# Out[6]:
(tensor([6., 6.,  ..., 7., 6.]), torch.Size([4898]))
```

The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Integer Vector

- If we want to transform the `target` tensor in a tensor of labels, as an integer vector of scores:

```
# In[7]:
target = wineq[:, -1].long()
target


# Out[7]:
tensor([6, 6, ..., 7, 6])
```

One Hot Encoding

- Encode each of the 10 scores in a vector of 10 elements, with all elements set to 0 but one, at a different index for each score.
- This way, a score of 1 could be mapped onto the vector `(1,0,0,0,0,0,0,0,0,0)`, a score of 5 onto `(0,0,0,0,1,0,0,0,0,0)`, and so on.
- Note that the fact that the score corresponds to the index of the nonzero element is purely incidental: we could shuffle the assignment, and nothing would change from a classification standpoint.
- We can achieve one-hot encoding using the `scatter_` method, which fills the tensor with values from a source tensor along the indices provided as arguments.
- One-hot encoding is also appropriate for quantitative scores when fractional values in between integer scores, like 2.4, make no sense for the application—for when the score is either *this* or *that*.

One Hot Encoding

```
# In[8]:
target_onehot = torch.zeros(target.shape[0], 10)
target_onehot.scatter_(1, target.unsqueeze(1),
1.0)

# Out[8]:
tensor([[0., 0., ..., 0., 0.],
[0., 0., ..., 0., 0.],
...,
[0., 0., ..., 0., 0.],
[0., 0., ..., 0., 0.]])
```

The arguments for `scatter_` are as follows:

- The dimension along which the following two arguments are specified
- A column tensor indicating the indices of the elements to scatter
- A tensor containing the elements to scatter or a single scalar to scatter (1, in this case)

Scatter Operator

```
target_onehot.scatter_(1, target.unsqueeze(1), 1.0)
```

- Its name ends with an underscore. This is a convention in PyTorch that indicates the method will not return a new tensor, but will instead modify the tensor in place.
- For each row, it takes the index of the target label (which coincides with the score in our case) and use it as the column index to set the value.
- The end result is a tensor encoding categorical information.
- The second argument of `scatter_`, the index tensor, is required to have the same
- number of dimensions as the tensor we scatter into.
- Since `target_onehot` has two dimensions (4,898 × 10), we need to add an extra dummy dimension to `target` using `unsqueeze`.