UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE of ENGINEERING

Introduction to ML
Lecture-11: Pytroch and Tensors

Hamed Tabkhi

Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

Repository:

https://github.com/deep-learning-with-pytorch/dlwpt-code

Book: Deep Learning with Pytorch (pdf is on Canvas page of the course)

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

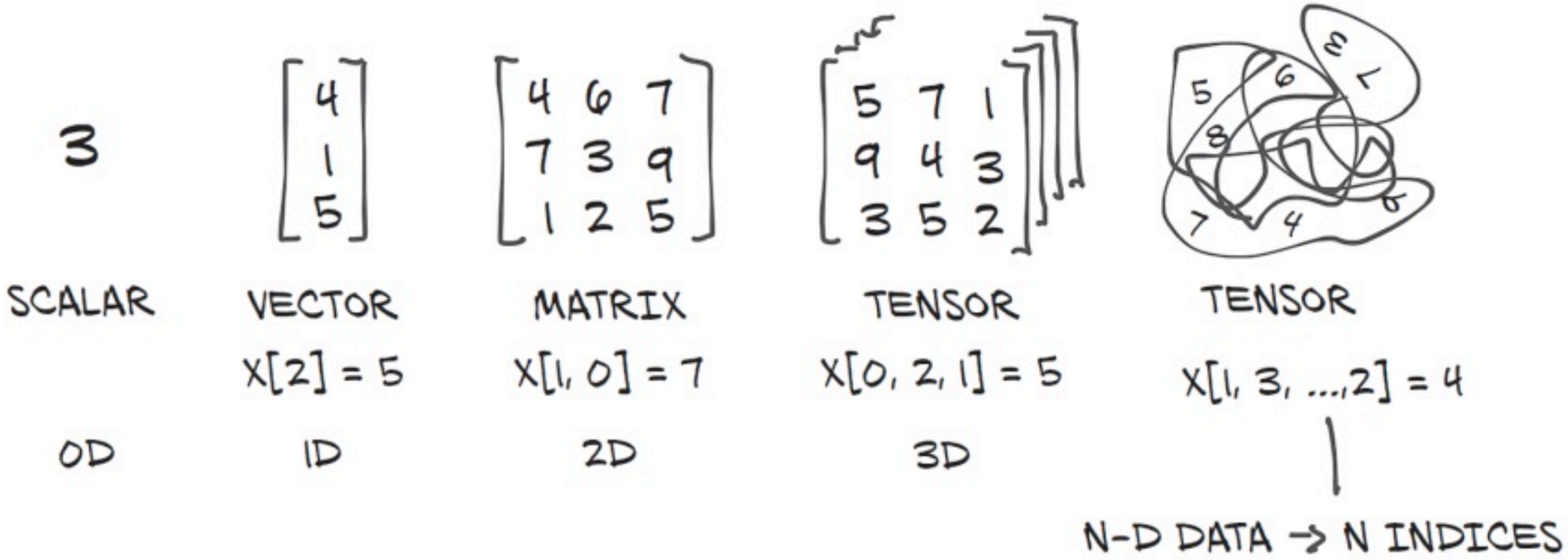
– Added support for ONNX, a vendor-neutral model description and exchange format

– 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*.

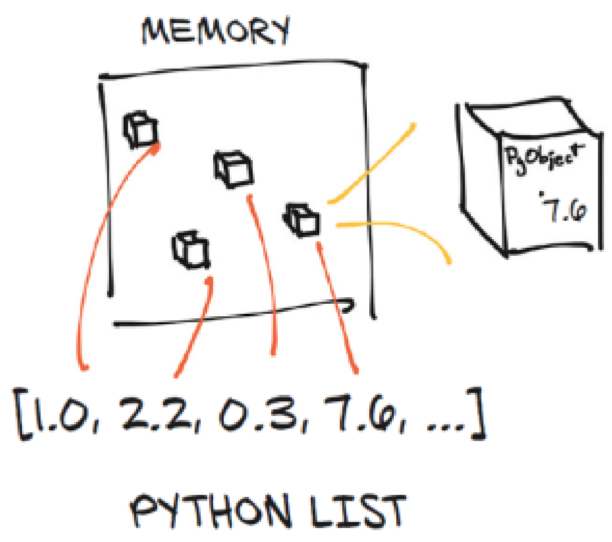
Pytorch Tensor and NumPy

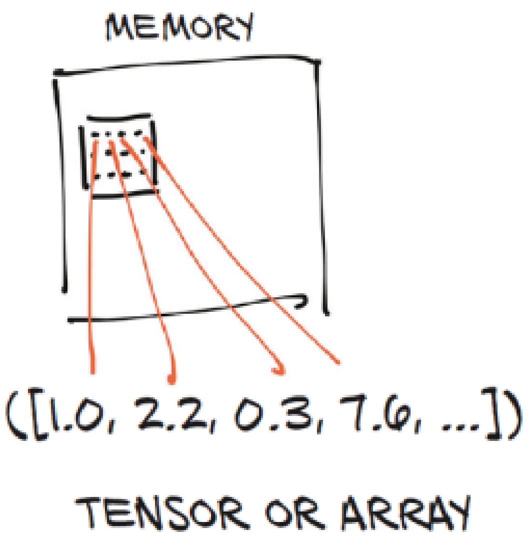
- PyTorch is not the only library that deals with multidimensional arrays.
- NumPy (https://numpy.org/) is by far the most popular multidimensional array library, it has now arguably become **the *lingua franca*** of data science.
- PyTorch features seamless interoperability with NumPy, which brings 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.





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)
```

Tensors' view of Storage

- # In[17]:

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

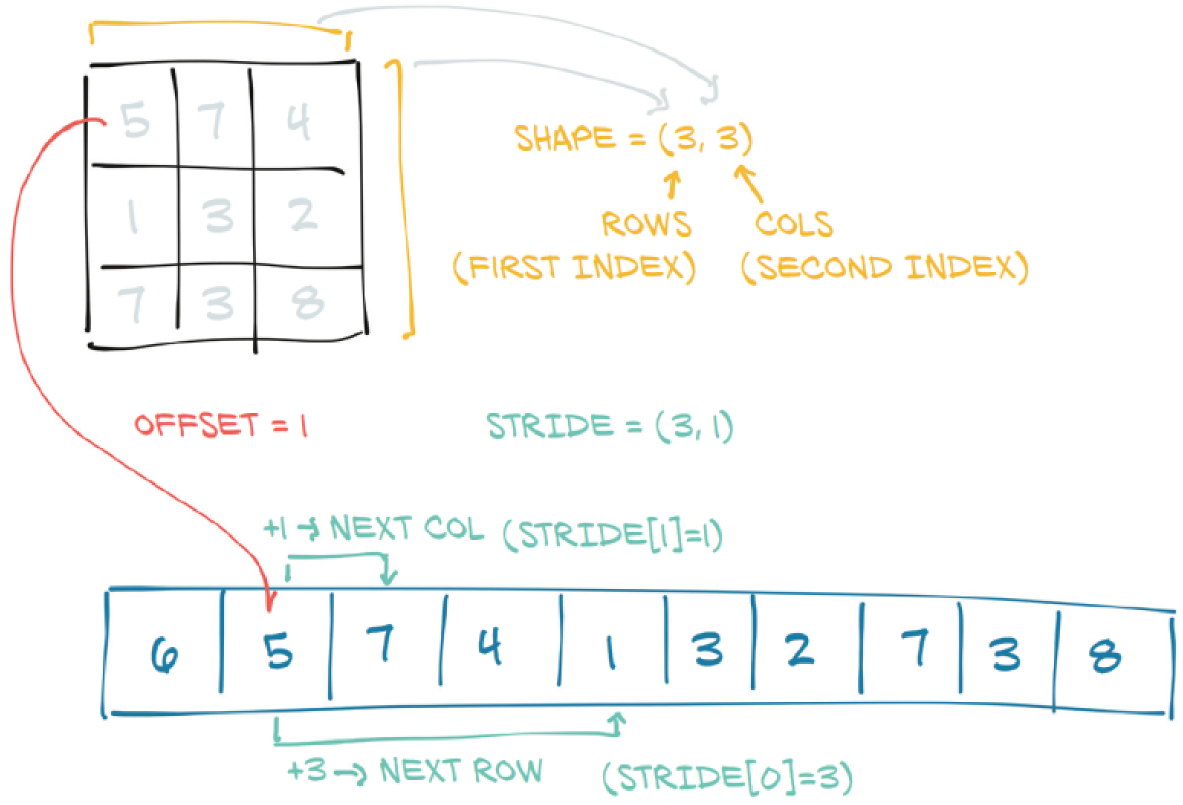
- # Out[17]:

```
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.



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`

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

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:

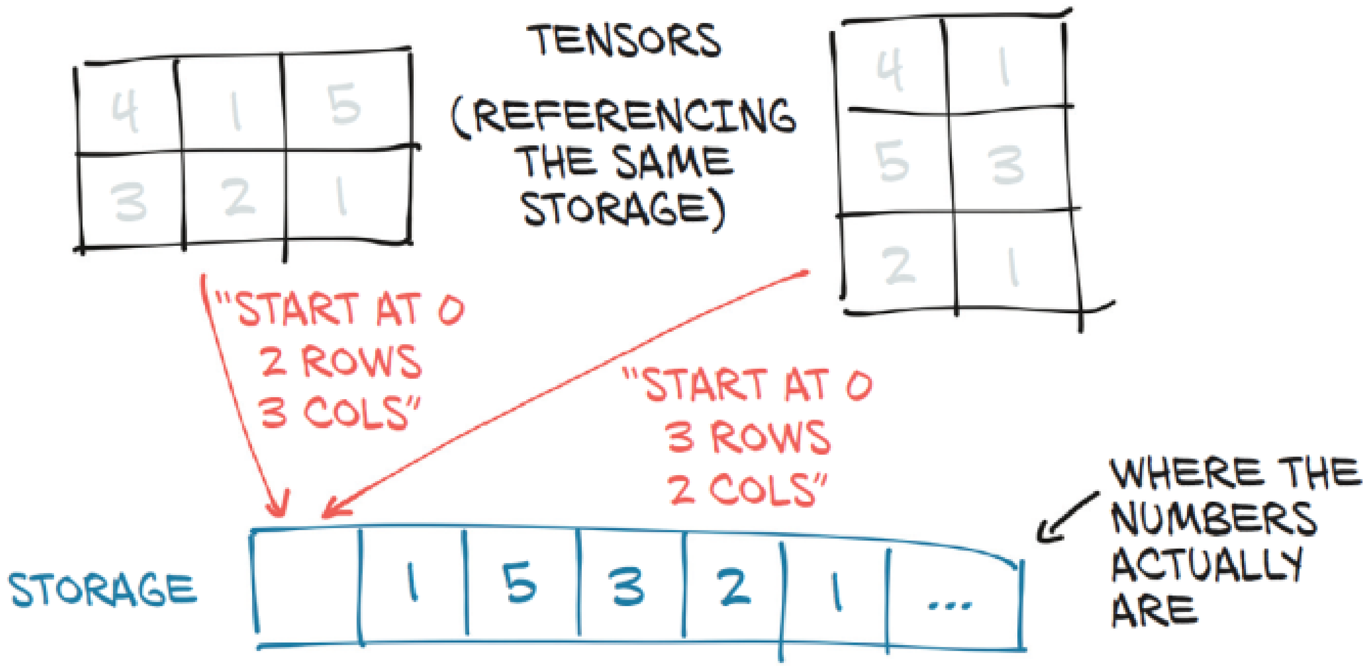
```
# In[72]:
    a = torch.ones(3, 2)
    a_t = a.transpose(0, 1)
    a.shape, a_t.shape

# 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

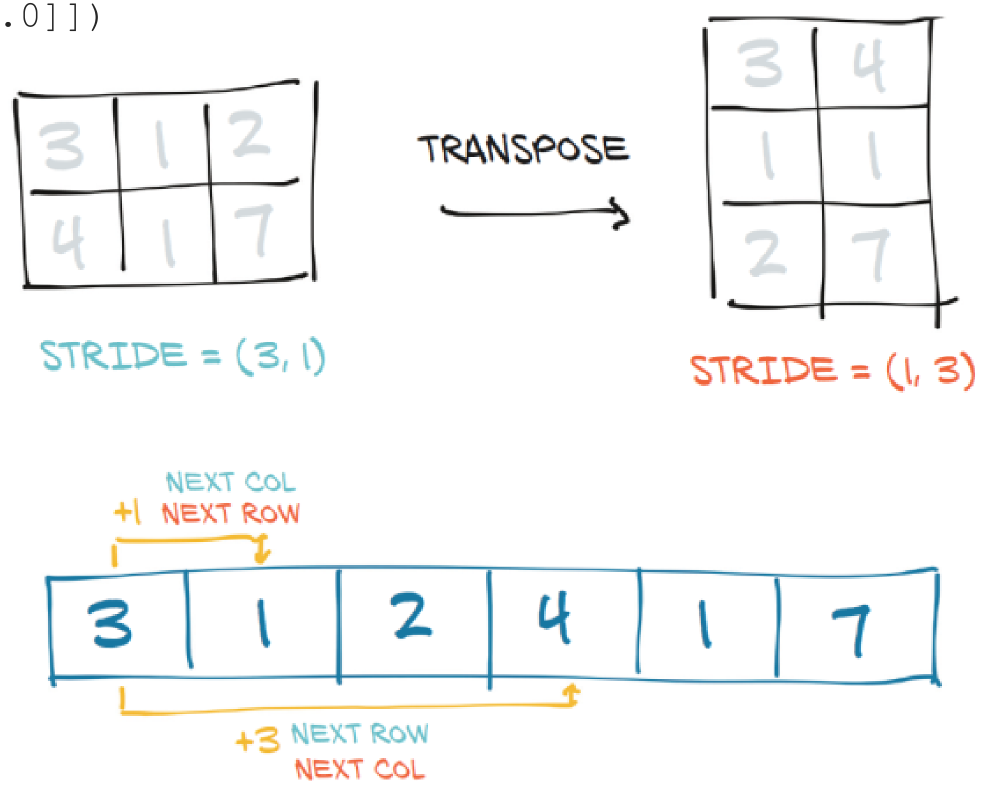
Tensors' view of Storage

- 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.



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

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.

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.**

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)
  ```
  ```