The size of current state-of-the-art networks makes computation a critical issue, in particular for training and optimizing meta-parameters.

Although they were historically developed for mass-market real-time CGI, the highly parallel architecture of GPUs is extremely fitting to signal processing and high dimension linear algebra.

Their use is instrumental in the success of deep-learning (Raina et al., 2009; Ciresan et al., 2010; Krizhevsky et al., 2012; Shi et al., 2016).
A standard NVIDIA GTX 1080 has 2,560 single-precision computing cores clocked at 1.6GHz, and delivers a peak performance of \( \approx 9 \) TFlops.

The precise structure of a GPU memory and how its cores communicate with it is a complicated topic that we will not cover here.

François Fleuret
Deep learning / 6.6. Using GPUs

(Shi et al., 2016)
The current standard to program a GPU is through the CUDA (“Compute Unified Device Architecture”) model, defined by NVIDIA.

Alternatives are OpenCL, backed by several CPU/DSP manufacturers, and more recently AMD’s HIP/ROCm.

Google developed its own line of processors for deep learning dubbed TPU (“Tensor Processing Unit”) which offer excellent flops/watt performance.

In practice, as of today (23.03.2020), NVIDIA hardware remains the default choice for deep learning, and CUDA is the reference framework in use.

From a practical perspective, libraries interface the framework (e.g. PyTorch) with the “computational backend” (e.g. CPU or GPU)

- BLAS (“Basic Linear Algebra Subprograms”): vector/matrix products, and the cuBLAS implementation for NVIDIA GPUs,
- LAPACK (“Linear Algebra Package”): linear system solving, Eigen-decomposition, etc.
- cuDNN (“NVIDIA CUDA Deep Neural Network library”) computations specific to deep-learning on NVIDIA GPUs.
Using GPUs in PyTorch

The use of the GPUs in PyTorch is done by creating or copying tensors into their memory.

Operations on tensors in a device’s memory are done by the said device.
As for the type, the device can be specified to the creation operations as a device, or as a string that will implicitly be converted to a device.

```python
>>> x = torch.zeros(10, 10)
>>> x.device
device(type='cpu')

>>> x = torch.zeros(10, 10, device = torch.device('cuda'))
>>> x.device
device(type='cuda', index=0)

>>> x = torch.zeros(10, 10, device = torch.device('cuda:1'))
>>> x.device
device(type='cuda', index=1)

>>> x = torch.zeros(10, 10, device = 'cuda:0')
>>> x.device
device(type='cuda', index=0)
```

The `torch.Tensor.to(device)` returns a clone on the specified device if the tensor is not already there or returns the tensor itself if it was already there.

The argument device can be either a string, or a device.

Alternatives are `torch.Tensor.cuda([gpu_id])` and `torch.Tensor.cpu()`.

⚠️ Moving data between the CPU and the GPU memories is far slower than moving it inside the GPU memory.
```python
>>> u = torch.tensor([1, 2, 3])
>>> u.device
device(type='cpu')
>>> v = u.to('cuda') # copy of u
>>> v
tensor([1, 2, 3], device='cuda:0')
>>> v[0] = 5
>>> u
tensor([1, 2, 3])
>>> w = u.to('cpu') # this is u itself
>>> w
tensor([1, 2, 3])
>>> w[0] = 5
>>> u
tensor([5, 2, 3])

>>> m = torch.empty(10, 10).normal_()
>>> m.device
device(type='cpu')
>>> x = torch.empty(10, 100).normal_()
>>> q = m@x
>>> q.device
device(type='cpu')
>>> m = m.to('cuda')
>>> x = x.to('cuda')
>>> q = m@x # This is done on GPU (#0)
>>> q.device
device(type='cuda', index=0)
```
Since operations maintain the types and devices of the tensors, you generally do not need to worry about making your code generic regarding these aspects.

To explicitly create new tensors you can use a tensor’s `new_*()` methods.

```python
generate_code_torch_tensor_creation()
```

Apart from `copy_()`, operations cannot mix different tensor types or devices:

```python
generate_code_tensor_type_mismatch()
```

Similarly if multiple GPUs are available, cross-GPUs operations are not allowed by default, with the exception of `copy_()`.
The method `torch.Module.to(device)` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the specified device.

⚠️ Although they do not have a “_” in their names, these Module operations make changes in-place.

The method `torch.cuda.is_available()` returns a Boolean value indicating if a GPU is available, so a typical GPU-friendly code would start with

```python
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
```

and then have some `device = device` in some places, and/or

```python
model.to(device)
criterion.to(device)
train_input, train_target = train_input.to(device), train_target.to(device)
test_input, test_target = test_input.to(device), test_target.to(device)
```
A very simple way to leverage multiple GPUs is to wrap the model in a \texttt{nn.DataParallel}.

The forward of \texttt{nn.DataParallel(my\_module)} will

1. split the input mini-batch along the first dimension in as many mini-batches as there are GPUs,
2. send them to the forwards of clones of \texttt{my\_module} located on each GPU,
3. concatenate the results.

And it is (of course!) autograd-compliant.
If we define a simple module to printout the calls to forward.

```python
class Dummy(nn.Module):
    def __init__(self, m):
        super().__init__()
        self.m = m

    def forward(self, x):
        print('Dummy.forward', x.size(), x.device)
        return self.m(x)
```

```python
x = torch.empty(50, 10).normal_
model = Dummy(nn.Linear(10, 5))

print('On CPU')
y = model(x)

x = x.to('cuda')
model.to('cuda')

print('On GPU w/o nn.DataParallel')
y = model(x)

print('On GPU w/ nn.DataParallel')
parallel_model = nn.DataParallel(model)
y = parallel_model(x)
```

will print, on a machine with two GPUs:

```
On CPU
Dummy.forward torch.Size([50, 10]) cpu
On GPU w/o nn.DataParallel
Dummy.forward torch.Size([50, 10]) cuda:0
On GPU w/ nn.DataParallel
Dummy.forward torch.Size([25, 10]) cuda:0
dummy.forward torch.Size([25, 10]) cuda:1
```
References


