Data 188: Introduction to Deep Learning

Intro to PyTorch

Speaker: Eric Kim
Lecture 15 (Week 09)
2026-03-19, Spring 2026. UC Berkeley.

Announcements

- Midterm grades will be released soon!
  - Regrade requests via Gradescope

- HW3 will be released soon. "Intro to Pytorch"
  - Tip: the remaining homework assignments in this course will not use Needle (ie your previous HW0 -> HW2).

Outline

PyTorch demo (notebook)

What is a GPU?

PyTorch demo (notebook)

What is a GPU?

GPU: "Graphics Processing Unit"

A separate hardware device that connects to your computer. Aka "discrete graphics card"

Reason to use them: if you can express your computation in a "GPU-friendly" way, then you can have much higher compute **throughput** than CPUs

- Typical applications: gaming, photo/video editing, video streaming

CPU benefits: flexible, low-latency, accessible (every computer has a CPU!)

GPU benefits: (very) high compute throughput

Pictured: my GeForce RTX 3080 Ti!

What is a GPU?

Two classes of GPUs:

- **Gaming/commodity**. Aka what you buy for your gaming PC.
  - Ex: Nvidia GeForce RTX 4080: >$999
- **Data center**. Aka what big companies like Google/Meta/OpenAI train/serve their DNN models on.
  - Ex: Nvidia H100 GPU: ~$25k per card.
  - Most heavy-duty DNN train machines have 8 GPUS, so ~$200K per machine.
    - Aka Amazon Cloud + Nvidia is making the big bucks right now!

Pictured: Nvidia DGX-2 data center machine [link] at Nvidia's booth for CVPR 2018 (Salt Lake City!). Has 16 Tesla V100 GPUs, likely worth several hundreds of thousands of USD!

Major GPU types

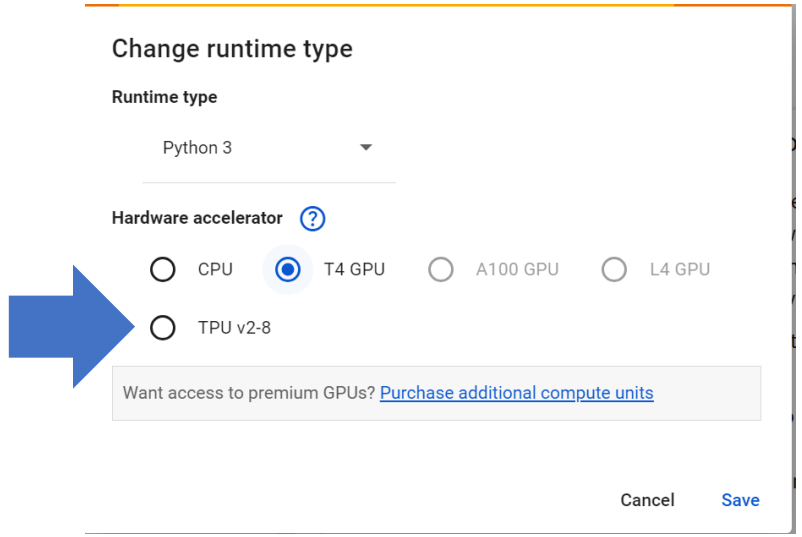
As of 2024: in this course (and ~99% of ML/AI): we use Nvidia GPUs (CUDA, cuDNN).

DNN frameworks like pytorch have excellent support for Nvidia GPUs (CUDA)

TPU ("Tensor Processing Unit"): A special type of "AI accelerator" hardware built by Google [link]

- Ex: Tensorflow, Google papers often use TPUs instead of Nvidia GPUs
- In Colab, you can choose either Nvidia GPU (eg T4) or a TPU.

AMD is trying to enter the AI/ML market too (IMO, difficult to break in at the moment)



Colab: T4 is Nvidia GPU, "TPU" is Google TPU

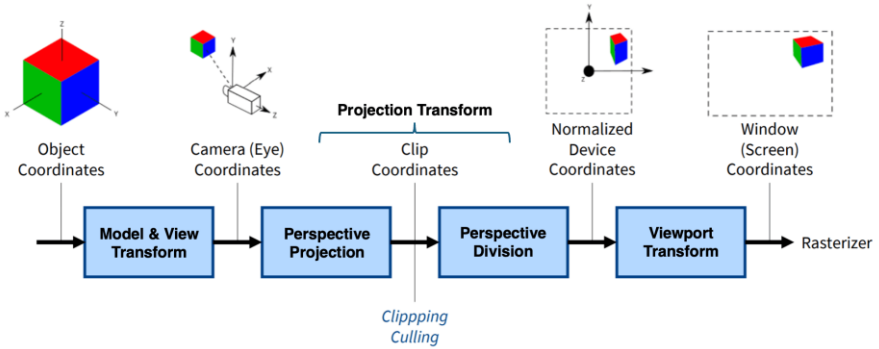
Original GPU motivation

Original intent: accelerate graphics processing, eg for computer graphics (CGI, Pixar) and video games (real-time graphics)

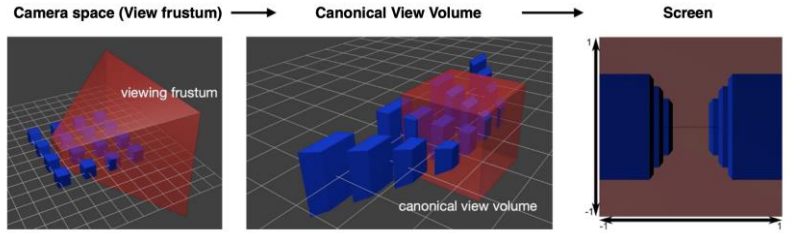
- Interested? Take CS 184! [link] Computer graphics and computer vision have a healthy relationship, lots of interesting overlap

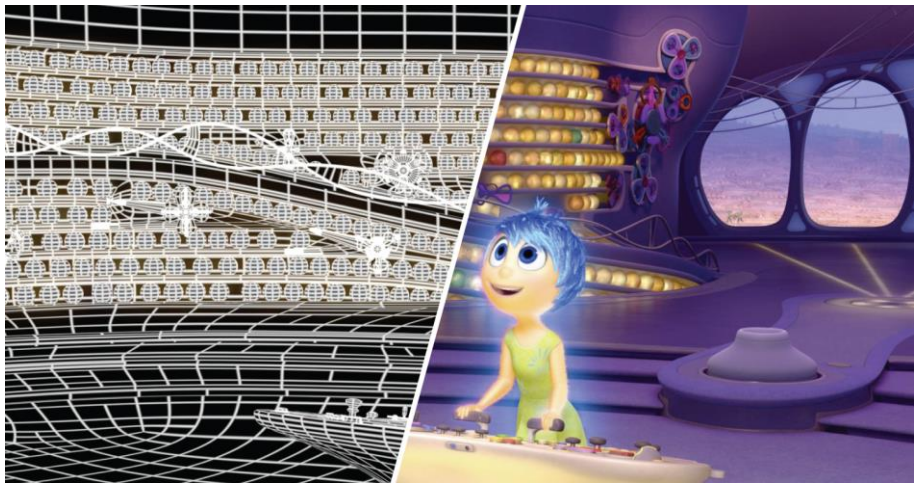
Motivation: graphics ultimately boils down to matrices and vectors. Need to accelerate matrix/vector computation, as the CPU wasn't enough back then (and still isn't now)

- Ex: a 3D point is represented as a 3D* vector, rotations/translations/scales are represented as 3x3 matrices.



$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$





* actually, we represent 3D points as a 4D vector [x, y, z, 1] ("homogenous" coordinates), and transformation matrices as 4x4 matrices, for good reasons (projective geometry).

https://leeyngdo.github.io/blog/computer-graphics/2024-02-29-graphics-pipeline/
https://vitaminac.github.io/Matrices-in-Computer-Graphics/
https://sciencebehindpixar.org/pipeline/rendering

What do GPUs excel at?

GPUs can do a LOT of parallel computation, much more so than CPUs!

- CPU: Typically has 4-16 cores (up to 8-32 active threads with hyperthreading)
- GPU: A Tesla P100 GPU has 56 "Streaming Multiprocesesors", each with 2048 active threads (up to 114688 active threads!)

If your computation can be easily parallelized, then GPUs are very good

Fortunately, nearly all DNN code falls under this category!

- Matrix/vector calculations (Linear, Conv2d, Relu, Softmax, etc.)

(2017) "Tensor Cores" [link] are an Nvidia hardware feature that further accelerates certain DNN operations, particularly for lower-precision datatypes like float16 ("mixed precision training")

GPUs for ML

AlexNet (2012) was an early (successful!) example of training a ConvNet on GPU hardware

GPU acceleration caught on: DNN libraries began offering "first class" GPU support

Caffe (2014): Developed at UC Berkeley. [link]

Caffe2 (2017): Developed at Facebook. [link]

- Note: basically deprecated in favor of pytorch

Tensorflow (2015): Developed at Google [link]

Pytorch (2016+): Developed at Facebook [link]

As of 2024, Pytorch and Tensorflow are the top DNN frameworks in use at both industry and academia. Personally: I prefer pytorch, but each has their strengths and weaknesses.

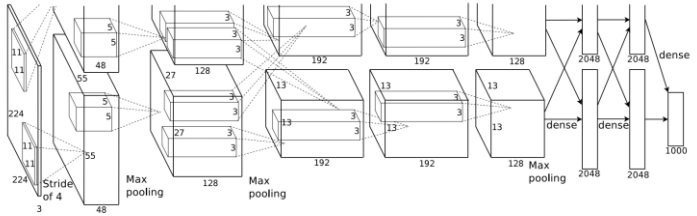


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Caffe2

TensorFlow

PyTorch

Nvidia: CUDA

CUDA: Compute Unified Device Architecture

Low-level library that tells the GPU how to compute your desired code.

CUDA code is basically C code

Main idea: you write CUDA code that expresses your computation in a "parallelizable" way, to effectively utilize the GPU's many execution threads

- Writing performant parallel code is an art! 99% of the time ML devs don't have to worry about this

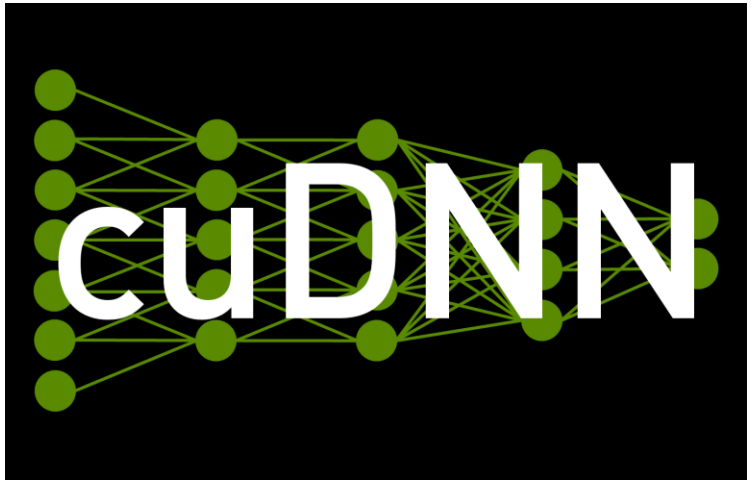
```
__global__
void add(int n, float * x, float * y) {
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

Pictured: element-wise vector addition CUDA code. Vector-addition is easily parallelizable, so we can chunk up the input arrays ("blocks") and dispatch each block to different GPU execution threads to process in parallel!

Nvidia cuDNN

cuDNN: an Nvidia library built on top of CUDA to provide high-performance DNN kernels (like optimized linear forward/backward, conv2d fwd/bkwd, etc)

DNN frameworks like pytorch, tensorflow ultimately compile down to CUDA code (often via cuDNN calls) which is what actually runs on your GPU!



Pytorch + GPUs

Pytorch makes it easy to use GPUs in your pytorch code!

Terminology: all Tensors live on a `torch.device` [link]

- By default: CPU device: `torch.device("cpu")`
- GPU (cuda): torch.device("cuda")
  - (If your machine has multiple GPUs) torch.device("cuda:0"), torch.device("cuda:1"), ...

Main principle: when doing an operation involving two tensors, both tensors must be on the same device!

Tensor(CPU) + Tensor(CPU): OK!
Tensor(GPU) + Tensor(GPU): OK!

Tensor(CPU) + Tensor(GPU): ERROR!

Note: in this course, we'll primarily focus on Nvidia GPU acceleration ("cuda", via `torch.cuda`). In 2026 there exist other accelerators (eg Apple MPS, TPU XLA, AMD ROCm, Intel XPU), and a new `torch.accelerator` API.

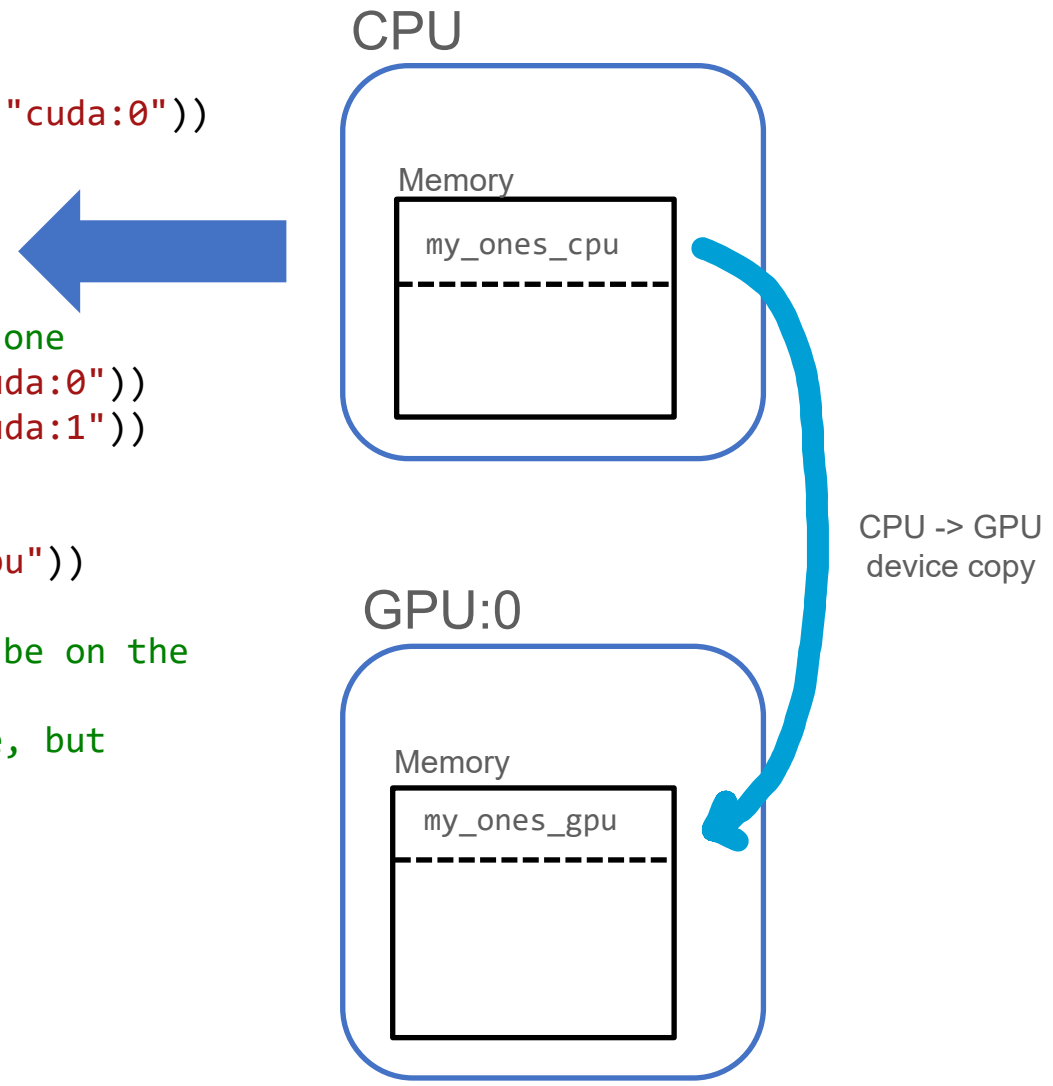
Pytorch example code: Tensor devices

```
# create tensor directly on cuda device
my_zeros_cuda = torch.zeros(size=[2, 3], device=torch.device("cuda:0"))
# create tensor on CPU, but move it to GPU (CPU -> GPU copy)
my_ones_cpu = torch.ones(size=[2, 4])
my_ones_cuda = my_ones_cpu.to(device=torch.device("cuda:0"))

# if you have multiple gpus, can send a tensor to a specific one
my_ones_cuda_device0 = my_ones_cpu.to(device=torch.device("cuda:0"))
my_ones_cuda_device1 = my_ones_cpu.to(device=torch.device("cuda:1"))

# finally, can send a tensor from GPU back to CPU
my_ones_cuda_to_cpu = my_ones_cuda.to(device=torch.device("cpu"))

# Beware: when doing ops between two tensors, they both most be on the
same device!
# RuntimeError: Expected all tensors to be on the same device, but
found at least two devices, cuda:0 and cpu!
my_ones_cpu + my_ones_cuda  # ERROR
```



Pytorch example code: module.to(device)

```python
import torch, torch.nn as nn
class Autoencoder(nn.Module):
    def __init__(self, out_channels_first: int = 16):
        super().__init__()
        self.encoder = nn.Sequential( # like the Composition layer you built
            nn.Conv2d(1, out_channels_first, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(out_channels_first, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7),
            nn.ReLU(),
            nn.ConvTranspose2d(32, out_channels_first, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(out_channels_first, 1, 3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )
    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

def print_statedict_info(module: torch.nn.Module):
    for param_key, param_val in module.state_dict().items():
        print(f"param_key={param_key}: {param_val.shape}, device={param_val.device}")

model_a = Autoencoder()
print("CPU module")
print_statedict_info(model_a)
model_a_gpu = model_a.to(device=torch.device("cuda:0"))
print("GPU module")
print_statedict_info(model_a_gpu)
```

```
(venv) PS C:\Users\Eric\teaching\data_c182_fall2024\src\lectures\lecture19>
python .\module_gpu_demo.py
CPU module
param_key=encoder.0.weight: torch.Size([16, 1, 3, 3]), device=cpu
param_key=encoder.0.bias: torch.Size([16]), device=cpu
param_key=encoder.2.weight: torch.Size([32, 16, 3, 3]), device=cpu
param_key=encoder.2.bias: torch.Size([32]), device=cpu
param_key=encoder.4.weight: torch.Size([64, 32, 7, 7]), device=cpu
param_key=encoder.4.bias: torch.Size([64]), device=cpu
param_key=decoder.0.weight: torch.Size([64, 32, 7, 7]), device=cpu
param_key=decoder.0.bias: torch.Size([32]), device=cpu
param_key=decoder.2.weight: torch.Size([32, 16, 3, 3]), device=cpu
param_key=decoder.2.bias: torch.Size([16]), device=cpu
param_key=decoder.4.weight: torch.Size([16, 1, 3, 3]), device=cpu
param_key=decoder.4.bias: torch.Size([1]), device=cpu
GPU module
param_key=encoder.0.weight: torch.Size([16, 1, 3, 3]), device=cuda:0
param_key=encoder.0.bias: torch.Size([16]), device=cuda:0
param_key=encoder.2.weight: torch.Size([32, 16, 3, 3]), device=cuda:0
param_key=encoder.2.bias: torch.Size([32]), device=cuda:0
param_key=encoder.4.weight: torch.Size([64, 32, 7, 7]), device=cuda:0
param_key=encoder.4.bias: torch.Size([64]), device=cuda:0
param_key=decoder.0.weight: torch.Size([64, 32, 7, 7]), device=cuda:0
param_key=decoder.0.bias: torch.Size([32]), device=cuda:0
param_key=decoder.2.weight: torch.Size([32, 16, 3, 3]), device=cuda:0
param_key=decoder.2.bias: torch.Size([16]), device=cuda:0
param_key=decoder.4.weight: torch.Size([16, 1, 3, 3]), device=cuda:0
param_key=decoder.4.bias: torch.Size([1]), device=cuda:0
```

Takeaway: model.to(device=device) copies all of the model parameters to the target device (eg all weight/bias params)

GPU memory: limited resource

Beware: GPUs have a limited amount of memory. Exceeding GPU memory will lead to your train run being killed with a "GPU out of memory" error!

Question: when training a DNN model, what are the main uses of GPU memory?

- **Model weights.** Ex: Linear's weight/bias parameters.
- **Intermediate activations.** If your model has N Conv2d's, then there will be N activation feature maps that pytorch has to keep track of that uses up GPU memory!
  - Scales linearly with your batch_size!
- **Gradients.** Calculated during backwards()
- **Additional optimizer state.** Ex: Adam requires `2*num_model_params` additional values (gradient moving avg, gradient magnitude moving avg)

Training model on GPU: Change 1/2

Taking a CPU pytorch training code and migrating it to the GPU is (often) very easy, a one-line(s) change!

Change 1: Move model (eg its model parameters) to the GPU device

```
# Create model (on CPU first, by default)
net = Net(hidden_num_chans=model_hidden_num_chans)

# Move model to GPU (if available to pytorch)
device_gpu_maybe = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")

print(f"(pre net.to) GPU max_memory_allocated: {torch.cuda.max_memory_allocated() / 1e6} MB")
net_gpu_maybe = net.to(device=device_gpu_maybe)
print(f"(post net.to) GPU max_memory_allocated: {torch.cuda.max_memory_allocated() / 1e6} MB")
```

```
(pre net.to) conv1.weight.device: cpu
(pre net.to) GPU max_memory_allocated: 0.0 MB
(post net.to) GPU max_memory_allocated: 0.481792 MB
(post net.to) conv1.weight.device: cuda:0
```

Huzzah, our layers are on the GPU!

Note: `torch.cuda.max_memory_allocated()` tells us how much GPU memory we've used. This tells us that our model parameters takes up 0.48 MB of GPU memory. Neat!

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#train-the-network

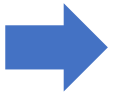
Training model on GPU: Change 2/2

Change 2: move all model inputs (including targets/labels!) to the GPU before calling forward

```
def train_model(model: torch.nn.Module, optimizer, criterion, trainloader, num_epochs: int, device:
torch.device) -> torch.Tensor:
    for epoch in range(num_epochs):  # loop over the dataset multiple times
        running_loss = 0.0
        for ind_batch, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            # Note: dataloader outputs inputs, labels as CPU tensors
            inputs, labels = data
            inputs = inputs.to(device=device)
            labels = labels.to(device=device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
```



`device` is our GPU device (or CPU device if we don't have a GPU!)

Since `model` is on GPU, and `inputs, labels` are on GPU, we won't have errors like "mismatch device"

Implementation tip: this code is "device agnostic", in that it works for both CPU and GPU contexts (just pass in device=torch.device("cpu") or device=torch.device("cuda:0")). This is the way!

Demo: pytorch CPU vs GPU (single GPU)

Demo: gpu_train_example.py

```
(venv) PS
C:\Users\Eric\teaching\data_c182_fall2024\src\lectures\le
cture19> python .\gpu_train_example.py
Files already downloaded and verified
Files already downloaded and verified
batchsize=64, model_hidden_num_chans=128,
num_dataloader_workers=2
(CPU) Begin training (120078 model params)
(CPU) Finished Training (33.91305661201477 secs,
1474.3584033733462 imgs/sec)
(pre net.to) GPU max_memory_allocated: 0.0 MB
(post net.to) GPU max_memory_allocated: 0.481792 MB
(GPU) Begin training (120078 model params)
(post train_model) GPU max_memory_allocated: 96.306688 MB
(GPU) Finished Training (9.31104588508606 secs,
5369.966018542273 imgs/sec)
```

Device	**Batchsize**	**Train throughput**
CPU	64	**1474 imgs/sec**
GPU	64	**5369 imgs/sec**

Demo: pytorch CPU vs GPU (single GPU)

Interestingly: the GPU doesn't always outperform the CPU! (here it does, but there are settings where it doesn't)

A few rules of thumb:

- Model should be big enough
- Batchsize needs to be big enough too!

Reason: too small model/batchsize means you spend most of your time doing CPU<->GPU communication, leading to poor GPU utilization

GPU's ideal computing mode: operate on large data "all at once" (aka large batches), rather than small data one-at-a-time.

Device	**Batchsize**	**Train throughput**
CPU	64	**1474 imgs/sec**
GPU	64	**5369 imgs/sec**
CPU	4	**1350 imgs/sec**
GPU	4	**1633 imgs/sec**
CPU	2	**860 imgs/sec**
GPU	2	**937 imgs/sec**