18-661: Introduction to ML for Engineers

Pytorch (and How We Got Here)

Spring 2022

Tianshu Huang, ECE – Carnegie Mellon University

Advertisement

Real Time 3D Scene Capture using NeRFs

Description NeRFs, or Neural Radiance Fields, are a method to create a 3D model of an object or scene by learning an implicit representation from a number of training images. Recent work has accelerated NeRF training from hours or even days down to seconds, which could potentially enable using NeRFs for real time scene capture.

Our objective is to explore the potential to use NeRFs for real time scene capture, which may involve modifying existing NeRF architectures, designing novel training pipelines, or integrating LIDAR-based volumetric video capture.

Skills Students should be proficient with Python. It will be helpful to have some exposure to CUDA and be familiar with deep learning frameworks.

Contact tianshu2@andrew.cmu.edu

Outline

- Deep Learning Hardware
- Deep Learning Frameworks
- Pytorch Tutorial
- Pytorch Example

Outline

- Deep Learning Hardware
- Deep Learning Frameworks
- Pytorch Tutorial
- Pytorch Example

Disclaimer: this lecture will not appear on your final exam, though some content, in particular PyTorch, will be used on Homework 7.

Deep Learning Hardware

The Problem with CPUs

Neural networks require lots of parallel computations, but CPUs require instructions to be executed sequentially.

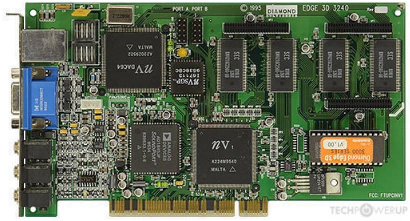
How can we speed up computations?

- More cores: lots of overhead (Intel Xeon Phi, discontinued in 2020)
- More execution units, longer pipeline: requires sophisticated out-of-order execution, branch prediction, etc; doesn't scale
- SIMD instructions (AVX): you still carry around the baggage of the CPU architecture; can't easily make vectors huge

CPUs don't scale, and can only get you so far.

Graphics Cards

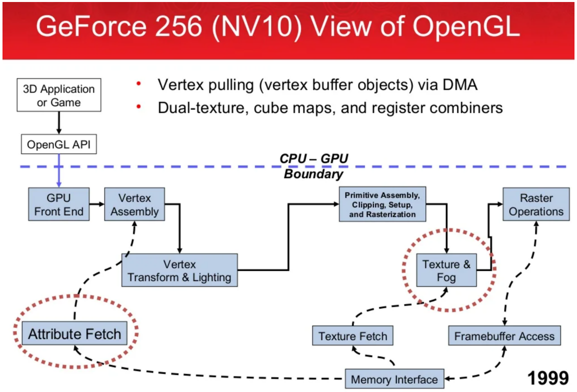
Graphics Cards: a card that connects to a display to show graphics.



Source: https://www.techpowerup.com/gpu-specs/nv1.c2015

Graphical Processing Units

Nvidia GeForce 256 “Transforming and Lighting engine”: compute shaders — pretty much just SIMD code execution!


GeForce 256 (NV10) View of OpenGL

- Vertex pulling (vertex buffer objects) via DMA
- Dual-texture, cube maps, and register combiners

Source: https://www.techspot.com/article/650-history-of-the-gpu/. Slide from SIGGRAPH Asia 2008.

General Purpose GPUs

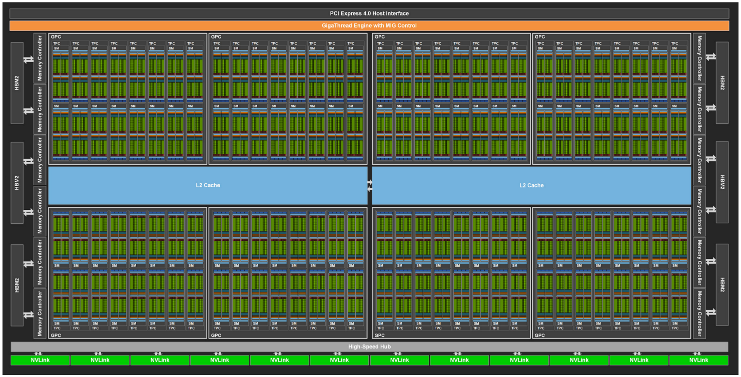
Nvidia and AMD embraced the “General Purpose GPU” paradigm for computer graphics:

- Organization of the GPU into Streaming Multiprocessors (SMs)
- “Nvidia realized that more cores running at a slower speed are more efficient for parallel workloads than fewer cores running at twice the frequency.”

Source: `https://www.techspot.com/article/659-history-of-the-gpu-part-4/`

Modern GPU Architecture

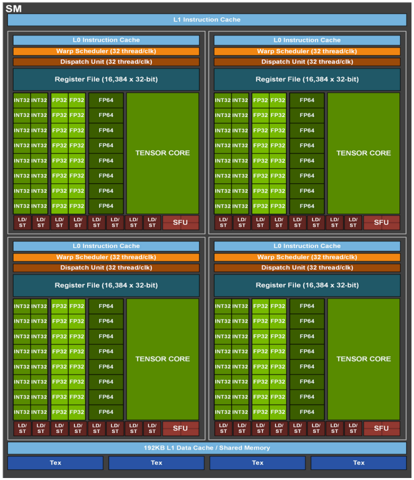
Nvidia RTX A100: $\approx$ \$30000; 108 SMs



PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

Source: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

Streaming Multiprocessor


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

We are executing a dense layer with batch size 256 and 4096 hidden units $\implies 2^{20}$ parallel.

1. Spawn 2^{20} threads.
2. Split threads into 4096 blocks of 256 threads.
3. Each SM gets assigned a block, and divides it into 8 warps of 32 threads.
4. These warps are sent to Warp Sechedulers that execute the instructions using 16 int32 units, 16 fp32 units, 8 fp64 units, and 1 tensor core.

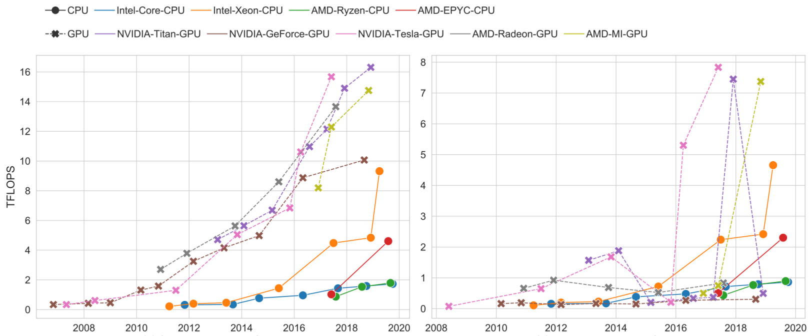
Total data parallelism: **6912**

GPU vs CPU

Device	Current Price	Floating Point	Power (TDP)
Nvidia A100 80GB	$\approx$ \$30000	156 TFlops	400W
Nvidia RTX A6000	$\approx$ \$6000	38.7 TFlops	300W
AMD EPYC 7713	$\approx$ \$7000	4.1 TFlops	225W
Nvidia RTX 3090	$\approx$ \$2000	35.6 TFlops	350W
Nvidia GTX 970	\$150 used	3.9 TFlops	150W

Specs from https://www.techpowerup.com/. Prices reflect current market prices as of March 2022.

GPU vs CPU



(a) Single-precision performance. (b) Double-precision performance.

Source: https://arxiv.org/pdf/1911.11313.pdf

More than just "More Cores"

Tensor Cores for 4x4 "Generalized Matrix Multiply":
GEMM$(A, B, C) = \mathbf{AB} + \mathbf{C}$. For example, if we multiply 2 8x8 matrices:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix},$$

where $A_{11}B_{11} + A_{12}B_{21} = \text{GEMM}(A_{11}, B_{11}, \text{GEMM}(A_{12}, B_{12}, 0))$.

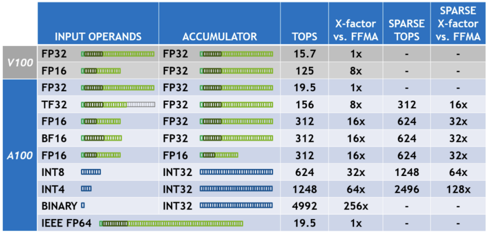
More than just “More Cores”

Tensor Cores for 4x4 “Generalized Matrix Multiply”:
GEMM$(A, B, C) = \mathbf{AB} + \mathbf{C}$. For example, if we multiply 2 8x8 matrices:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix},$$

where $A_{11}B_{11} + A_{12}B_{21} = \text{GEMM}(A_{11}, B_{11}, \text{GEMM}(A_{12}, B_{12}, 0))$.

Data Types specifically for deep learning:

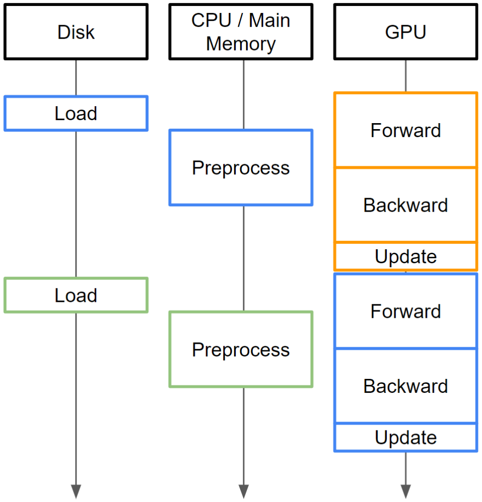


	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA	SPARSE TOPS	SPARSE X-factor vs. FFMA
V100	FP32	FP32	15.7	1x	-	-
	FP16	FP32	125	8x	-	-
A100	FP32	FP32	19.5	1x	-	-
	TF32	FP32	156	8x	312	16x
	FP16	FP32	312	16x	624	32x
	BF16	FP32	312	16x	624	32x
	FP16	FP16	312	16x	624	32x
	INT8	INT32	624	32x	1248	64x
	INT4	INT32	1248	64x	2496	128x
	BINARY	INT32	4992	256x	-	-
	IEEE FP64		19.5	1x	-	-

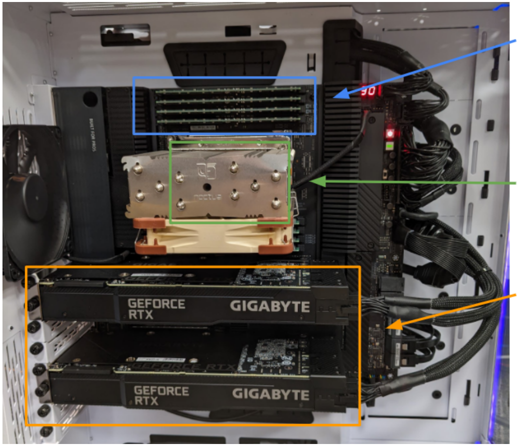
Source: https://jonathan-hui.medium.com/ai-chips-a100-gpu-with-nvidia-ampere-architecture-3034ed685e6e

A Typical Deep Learning Pipeline

- The CPU is usually used for data preprocessing only.
- All parameter and gradient computations take place on the GPU
- Data loading and preprocessing should be pipelined to avoid impacting runtime.



A Typical Deep Learning Machine



System memory: dataset

CPU: preprocessing

GPU: everything else

Batch-parallel training
(GPU0 gets the first half,
GPU1 gets the second half)

An Atypical Deep Learning Machine

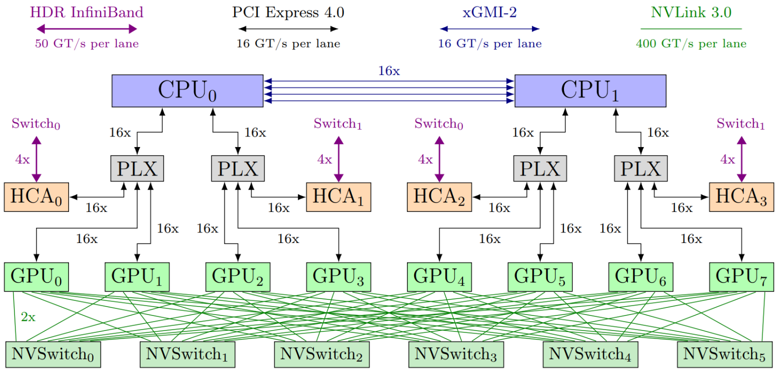


Figure 2: Architecture diagram of a single training node.

We trained GPT-NeoX-20B on twelve Supermicro AS-4124GO-NART servers, each with eight NVIDIA A100-SXM4-40GB GPUs and configured with two AMD EPYC 7532 CPUs. All GPUs can directly access the InfiniBand switched fabric through one of four ConnectX-6 HCAs for GPUDirect RDMA. Two NVIDIA MQM8700-HS2R switches—connected by 16 links—compose the spine of this InfiniBand network, with one link per node CPU socket connected to each switch. Figure 2 shows a simplified overview of a node as configured for training.

Aside: Why not AMD?

AMD GPUs are not usually used, and for the most part cannot be used with deep learning frameworks!

- No hardware optimization for deep learning (Tensor Cores, ML-specific data types)
- No software support (CUDA, cuDNN, etc)
- Poor community adoption due to poor historical performance

Deep Learning Frameworks

Before Pytorch

What we used to do:

- LeNet, 1989: custom compiled code (most likely C or Fortran)
- AlexNet, 2012: custom CUDA code
- Early Deep Learning “Boom”: early frameworks such as Caffe, Theano
- Recent deep learning: Tensorflow vs Pytorch

Modern Frameworks

Tensorflow: the first “Modern” deep learning framework.

- (TF 1) Build, compile, then execute compute graph
- (TF 2) This is too annoying, let’s add “eager execution” instead

Modern Frameworks

Tensorflow: the first “Modern” deep learning framework.

- (TF 1) Build, compile, then execute compute graph
- (TF 2) This is too annoying, let’s add “eager execution” instead

Pytorch: easier to use than tensorflow

- Optimize overhead for eager execution, and don’t worry about compiling graphs
- ... Maybe we still want that performance of graph execution

Modern Frameworks

Tensorflow: the first “Modern” deep learning framework.

- (TF 1) Build, compile, then execute compute graph
- (TF 2) This is too annoying, let’s add “eager execution” instead

Pytorch: easier to use than tensorflow

- Optimize overhead for eager execution, and don’t worry about compiling graphs
- ... Maybe we still want that performance of graph execution

JAX: built from the ground up to use a JIT approach

- Much more intuitive than Pytorch and Tensorflow
- New and not yet mature, missing a lot of tooling

Choosing a Framework: Pytorch vs Tensorflow vs JAX

When to use...

- JAX: you work for Google, or have connections at Google.

Choosing a Framework: Pytorch vs Tensorflow vs JAX

When to use...

- JAX: you work for Google, or have connections at Google.
- Tensorflow: you have TPUs or want to easily deploy your model using Tensorflow Lite.

Choosing a Framework: Pytorch vs Tensorflow vs JAX

When to use...

- JAX: you work for Google, or have connections at Google.
- Tensorflow: you have TPUs or want to easily deploy your model using Tensorflow Lite.
- Pytorch: everyone else.

Choosing a Framework: Pytorch vs Tensorflow vs JAX

When to use...

- JAX: you work for Google, or have connections at Google.
- Tensorflow: you have TPUs or want to easily deploy your model using Tensorflow Lite.
- Pytorch: everyone else.

... though if you use high level APIs such as Flax, Keras, torch.nn, there is little difference between the frameworks.

Pytorch

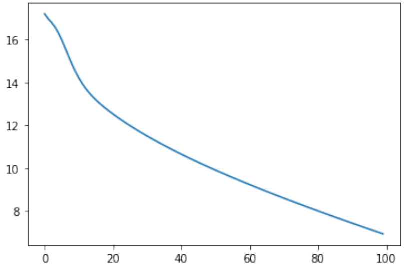
Pytorch

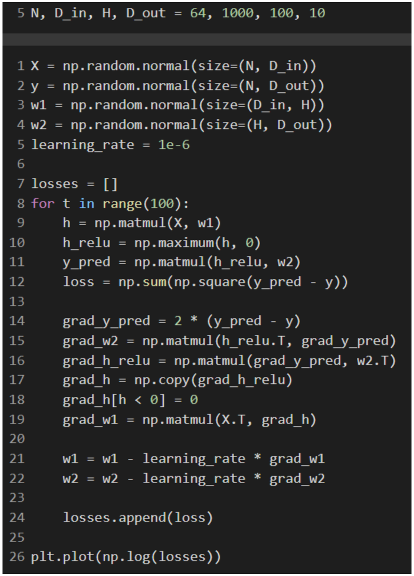
Three main components:

- GPU Computation (on `torch.Tensor`), which can use eager execution (default) or graph execution (`torch.jit`)
- Automatic Differentiation (`torch.autograd`)
- High level neural network API (`torch.nn`, `torch.optim`, `torch.utils.data`)

Plain Numpy

Numpy implementation of regression using a neural network with 2 layers, trained with Gradient Descent





```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
X = np.random.normal(size=(N, D_in))
y = np.random.normal(size=(N, D_out))
w1 = np.random.normal(size=(D_in, H))
w2 = np.random.normal(size=(H, D_out))
learning_rate = 1e-6

losses = []
for t in range(100):
    h = np.matmul(X, w1)
    h_relu = np.maximum(h, 0)
    y_pred = np.matmul(h_relu, w2)
    loss = np.sum(np.square(y_pred - y))

    grad_y_pred = 2 * (y_pred - y)
    grad_w2 = np.matmul(h_relu.T, grad_y_pred)
    grad_h_relu = np.matmul(grad_y_pred, w2.T)
    grad_h = np.copy(grad_h_relu)
    grad_h[h < 0] = 0
    grad_w1 = np.matmul(X.T, grad_h)

    w1 = w1 - learning_rate * grad_w1
    w2 = w2 - learning_rate * grad_w2

    losses.append(loss)

plt.plot(np.log(losses))
```

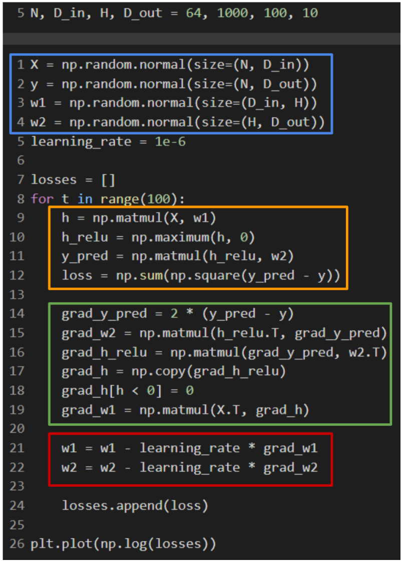
Plain Numpy

Ordinary Arrays on CPU

Forward Pass

Backward Pass

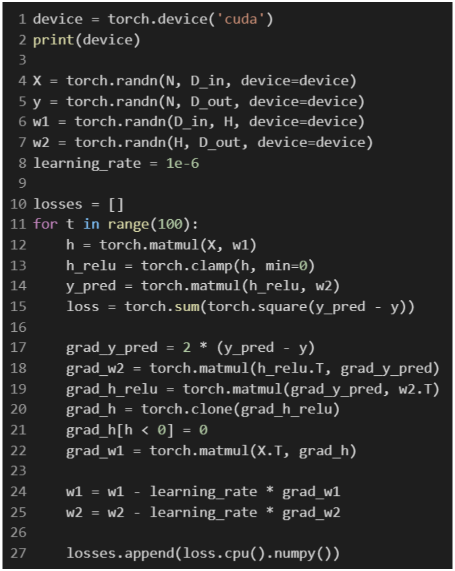
Parameter Update



```
N, D_in, H, D_out = 64, 1000, 100, 10

X = np.random.normal(size=(N, D_in))
y = np.random.normal(size=(N, D_out))
w1 = np.random.normal(size=(D_in, H))
w2 = np.random.normal(size=(H, D_out))
learning_rate = 1e-6

losses = []
for t in range(100):
    h = np.matmul(X, w1)
    h_relu = np.maximum(h, 0)
    y_pred = np.matmul(h_relu, w2)
    loss = np.sum(np.square(y_pred - y))

    grad_y_pred = 2 * (y_pred - y)
    grad_w2 = np.matmul(h_relu.T, grad_y_pred)
    grad_h_relu = np.matmul(grad_y_pred, w2.T)
    grad_h = np.copy(grad_h_relu)
    grad_h[h < 0] = 0
    grad_w1 = np.matmul(X.T, grad_h)

    w1 = w1 - learning_rate * grad_w1
    w2 = w2 - learning_rate * grad_w2

    losses.append(loss)

plt.plot(np.log(losses))
```

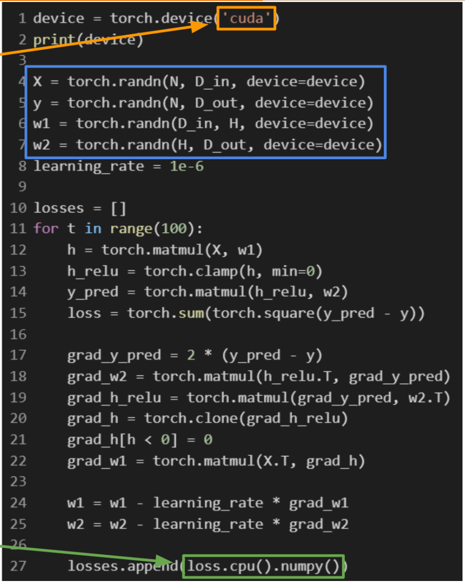
GPU Computation

Replace numpy with torch (with a few exceptions where the function names change)



```
device = torch.device('cuda')
print(device)

X = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)
learning_rate = 1e-6

losses = []
for t in range(100):
    h = torch.matmul(X, w1)
    h_relu = torch.clamp(h, min=0)
    y_pred = torch.matmul(h_relu, w2)
    loss = torch.sum(torch.square(y_pred - y))

    grad_y_pred = 2 * (y_pred - y)
    grad_w2 = torch.matmul(h_relu.T, grad_y_pred)
    grad_h_relu = torch.matmul(grad_y_pred, w2.T)
    grad_h = torch.clone(grad_h_relu)
    grad_h[h < 0] = 0
    grad_w1 = torch.matmul(X.T, grad_h)

    w1 = w1 - learning_rate * grad_w1
    w2 = w2 - learning_rate * grad_w2

    losses.append(loss.cpu().numpy())
```

GPU Computation



CUDA = Nvidia GPU

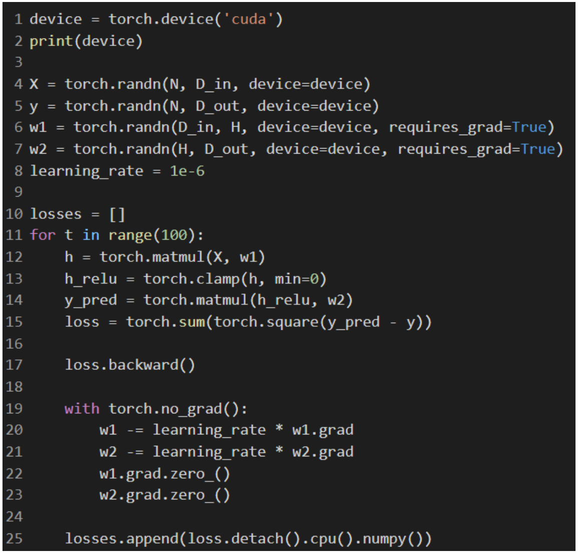
Arrays sent to GPU

Send the loss back to the CPU

```
device = torch.device('cuda')
print(device)

X = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)
learning_rate = 1e-6

losses = []
for t in range(100):
    h = torch.matmul(X, w1)
    h_relu = torch.clamp(h, min=0)
    y_pred = torch.matmul(h_relu, w2)
    loss = torch.sum(torch.square(y_pred - y))

    grad_y_pred = 2 * (y_pred - y)
    grad_w2 = torch.matmul(h_relu.T, grad_y_pred)
    grad_h_relu = torch.matmul(grad_y_pred, w2.T)
    grad_h = torch.clone(grad_h_relu)
    grad_h[h < 0] = 0
    grad_w1 = torch.matmul(X.T, grad_h)

    w1 = w1 - learning_rate * grad_w1
    w2 = w2 - learning_rate * grad_w2

    losses.append(loss.cpu().numpy())
```

Automatic Differentiation

Pytorch's Automatic Differentiation uses a "gradient tape" which records all operations made on tensors marked with `requires_grad`



```python
device = torch.device('cuda')
print(device)

X = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, requires_grad=True)
learning_rate = 1e-6

losses = []
for t in range(100):
    h = torch.matmul(X, w1)
    h_relu = torch.clamp(h, min=0)
    y_pred = torch.matmul(h_relu, w2)
    loss = torch.sum(torch.square(y_pred - y))

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()

    losses.append(loss.detach().cpu().numpy())
```

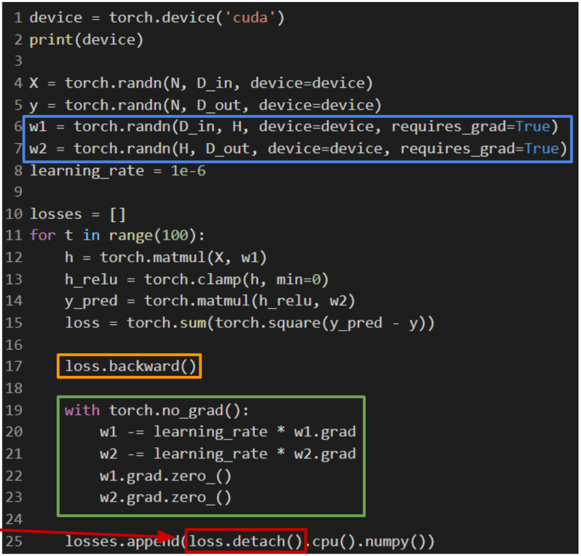
Automatic Differentiation

Record gradients for the weights

… and it's like magic

Don't track gradients when updating params

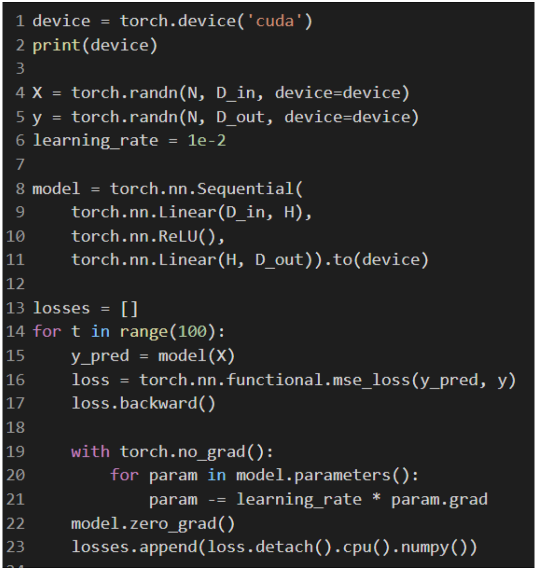
Don't send gradient to CPU



```
device = torch.device('cuda')
print(device)

X = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, requires_grad=True)
learning_rate = 1e-6

losses = []
for t in range(100):
    h = torch.matmul(X, w1)
    h_relu = torch.clamp(h, min=0)
    y_pred = torch.matmul(h_relu, w2)
    loss = torch.sum(torch.square(y_pred - y))

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()

    losses.append(loss.detach().cpu().numpy())
```

High Level API: `torch.nn`

Instead of manually constructing each layer, activation, initialization, etc, use pre-constructed layers



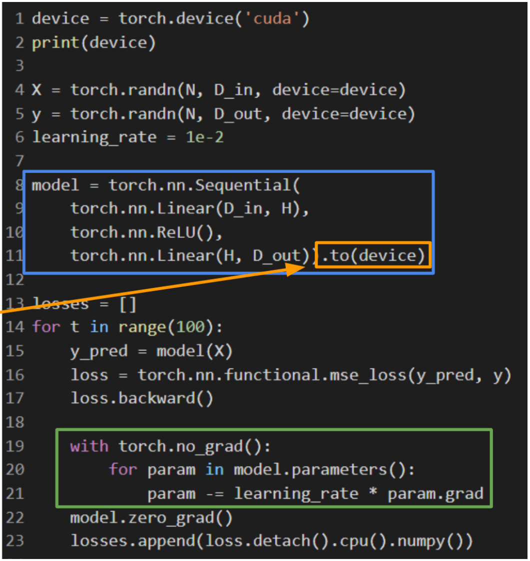
```python
device = torch.device('cuda')
print(device)

X = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
learning_rate = 1e-2

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)).to(device)

losses = []
for t in range(100):
    y_pred = model(X)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
    losses.append(loss.detach().cpu().numpy())
```

High Level API: `torch.nn`

You will implement something similar in HW5 Q3

Send model to GPU

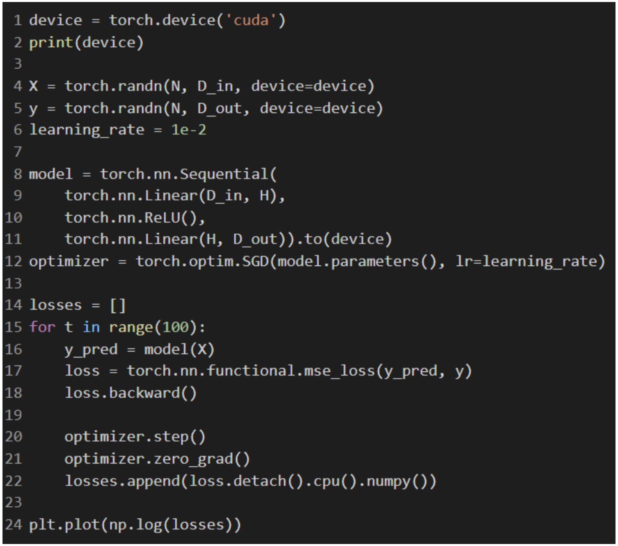
The model provides a convenient way to iterate over its parameters



```python
device = torch.device('cuda')
print(device)

X = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
learning_rate = 1e-2

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)).to(device)

losses = []
for t in range(100):
    y_pred = model(X)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
    losses.append(loss.detach().cpu().numpy())
```

High Level API: `torch.optim`

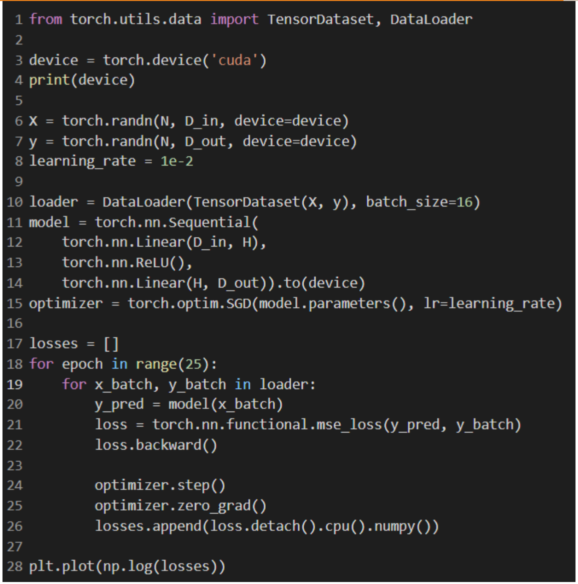
Let's get rid of that last bit of handwritten training code . . .



```python
device = torch.device('cuda')
print(device)

X = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
learning_rate = 1e-2

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

losses = []
for t in range(100):
    y_pred = model(X)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
    losses.append(loss.detach().cpu().numpy())

plt.plot(np.log(losses))
```

High Level API: `torch.utils.data`

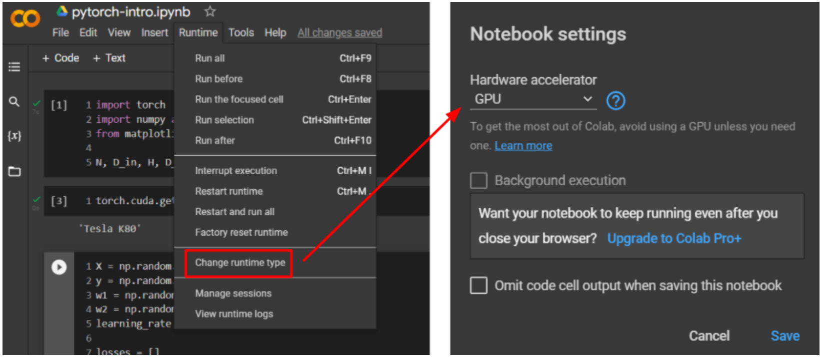
Use `DataLoader` to pipeline data loading and preprocessing



```python
from torch.utils.data import TensorDataset, DataLoader

device = torch.device('cuda')
print(device)

X = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
learning_rate = 1e-2

loader = DataLoader(TensorDataset(X, y), batch_size=16)
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

losses = []
for epoch in range(25):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred, y_batch)
        loss.backward()

        optimizer.step()
        optimizer.zero_grad()
        losses.append(loss.detach().cpu().numpy())

plt.plot(np.log(losses))
```

Pytorch Example

Google Colab



```
[1] import torch
    import numpy a
    from matplotl

    N, D_in, H, D
```

```
[3] torch.cuda.get

'Tesla K80'
```

```
X = np.random
y = np.random
w1 = np.rando
w2 = np.rando
learning_rate

losses = []
```

Notebook settings

Hardware accelerator
GPU

To get the most out of Colab, avoid using a GPU unless you need one. Learn more

☐ Background execution

Want your notebook to keep running even after you close your browser? Upgrade to Colab Pro+

☐ Omit code cell output when saving this notebook

Recitation & HW5

HW5 Release on Monday

- Start early!

Recitation:

- Deeper dive into the Pytorch Example
- Backpropagation Walkthrough
- Vectorization, Numerical Stability, and Debugging Tricks