

Introduction to PyTorch

Yifei Liu

Tutorial session, STAT8056 2024 Spring

Slides and notebook available at: <https://github.com/yifei-liu-stat/stat8056-intro-pytorch>

For questions/discussions/typos, contact: liu00980@umn.edu



UNIVERSITY OF MINNESOTA
Driven to Discover®

Outline

- Set up working environment
 - Coding platform
 - Practice: create conda environment for your project
- Introduction to PyTorch framework
- Deep learning with PyTorch
- Additional resources



Coding platform

On your local machine

Use Anaconda to manage packages:

- Go to anaconda.com
- Download Anaconda
- Open Anaconda Navigator (GUI)
 - Create an isolated environment for each of your projects.
 - Manage dependencies/versions of packages within the environment.
- Alternatively, one can use Anaconda in command-line style.

Notebook-style coding:

- Good for reports and demos.
- In Anaconda Navigator, install Jupyter Notebook (if not already)
- Launch Jupyter Notebook.

Script-style coding:

- For efficiency and productivity.
- Choose an IDE to start with.
- Take VS Code for example:
 - Download [VS Code](#).
 - Follow this [tutorial](#) to learn how to use Python in Visual Studio Code.



Coding platform

On a remote host/server

Google Colab (for this tutorial)

- Notebook-style cloud computing.
 - <https://colab.research.google.com/>
 - Free version runs up to 12 hrs.
- Free GPU resource:
 - Runtime | Change runtime type | Hardware accelerator | GPU
- Many libraries (numpy, torch, scikit-learn ...) are pre-installed.
- Co-edit/code with collaborators.
- In sync. with your Google Drive.

Resources from MSI:

- Need to be in a research group to use the cluster. [MSI access](#).
- Support Jupyter notebook:
 - Connect to [UMN VPN](#) (off-campus only)
 - Visit notebooks.msi.umn.edu
 - Choose and start a server.
- [Free MSI tutorial \(with recordings\)](#)

Other cloud services:

- [Deep-learning-in-cloud](#)
 - A list of cloud vendors either with free or paid services (with some free credits)



UMN computing resources

MSI: <https://www.msi.umn.edu/>

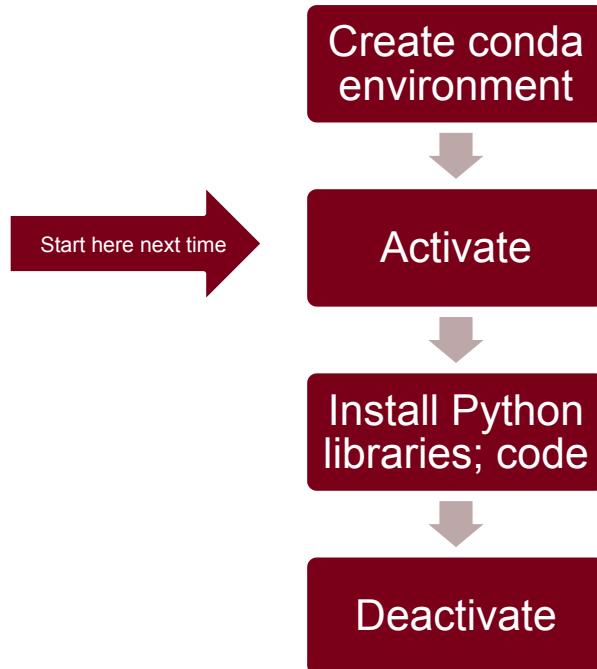
Introduction to Minnesota Supercomputing Institute (MSI)	This tutorial is geared to new MSI users and will provide a highlevel introduction to the facilities and computational resources at MSI.	02/01/2022
Introduction to Linux	This tutorial will provide an introduction to the Linux operating system, with particular attention paid to working from the command line	02/03/2022
Programming with Python	Introduction to fundamentals of programming using the python language.	02/08/2022
Job Submission and Scheduling at MSI	This tutorial will introduce users to MSI supercomputers, and provide an overview of how to submit calculations to the job schedulers	02/10/2022
Interactive Computing at MSI	This two part tutorial will introduce you to the concept of interactive high performance computing, and provide attendees hands-on experience running interactive parallel jobs on the Mesabi HPC	02/15/2022
Data Storage Systems and Data Analysis Workflows for Research	In this tutorial you will learn about the data storage systems available for academic research at the University of Minnesota	02/17/2022
Compiling and Debugging at MSI	This tutorial will help users learn the basics of compiling and debugging their code on MSI systems	02/24/2022
Python for Scientific Computing	This session includes efficient data processing with NumPy and Scipy, data visualization, and techniques for using python to drive parallel supercomputing tasks.	03/01/2022
RNA-Seq Analysis	This tutorial covers the basics of differential expression analysis and touches on other RNA-seq topics such as transcriptome assembly.	03/03/2022
Parallel Computing On Agate	This tutorial will help users learn the basics of parallel computation methods, including strategies for collecting calculations together for parallel execution.	03/31/2022

LATIS: <https://latisresearch.umn.edu/>

Feb. 18th 10:00am-noon	Creating Publication Worthy Visualizations without Code	Registration
Feb. 25th 10:00am-noon	Introduction to Computational Text Analysis	Registration
Mar. 4th 10:00am-noon	Reproducible research practices in Excel (yes, Excel)	Registration
Mar. 9th 10:00am-noon	Data Management in transition: Strategies for when you graduate	Registration
Mar. 18th 10:00am-noon [RESCHEDULED]	Advanced Nvivo	Registration
Mar. 25th 10:00am-noon	Introduction to parallel computing	Registration
April 1st 10:00am-noon	Introduction to SQL and Research Databases	Registration

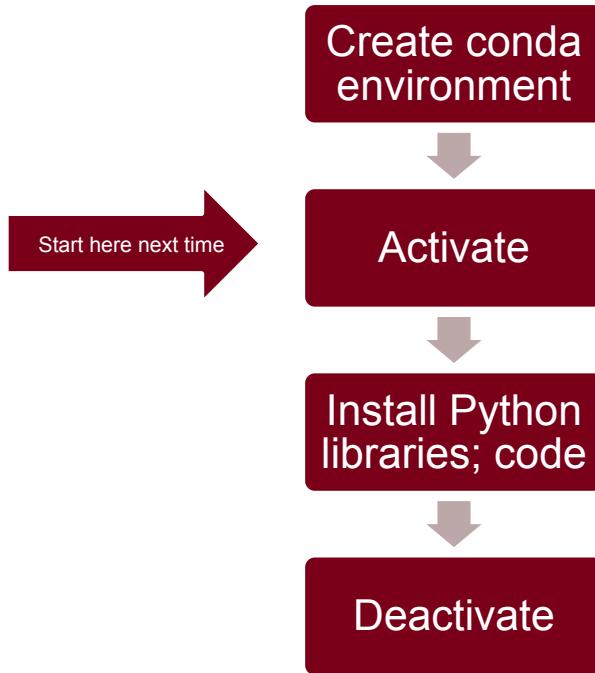


Create a conda environment for your Python project



- An alternative way to Virtualenv
- Manage dependencies (**different versions** of Python and some libraries) **within an isolated environment**.
 - An example: Python library [foolbox](#)
- Key commands:
 - `conda create -n <envname> python=3.8.3`
 - `conda activate <envname>`
 - `conda install <package>=<version>`
 - `conda deactivate`
- Check [Conda Cheat Sheet](#)

A hands-on exercise



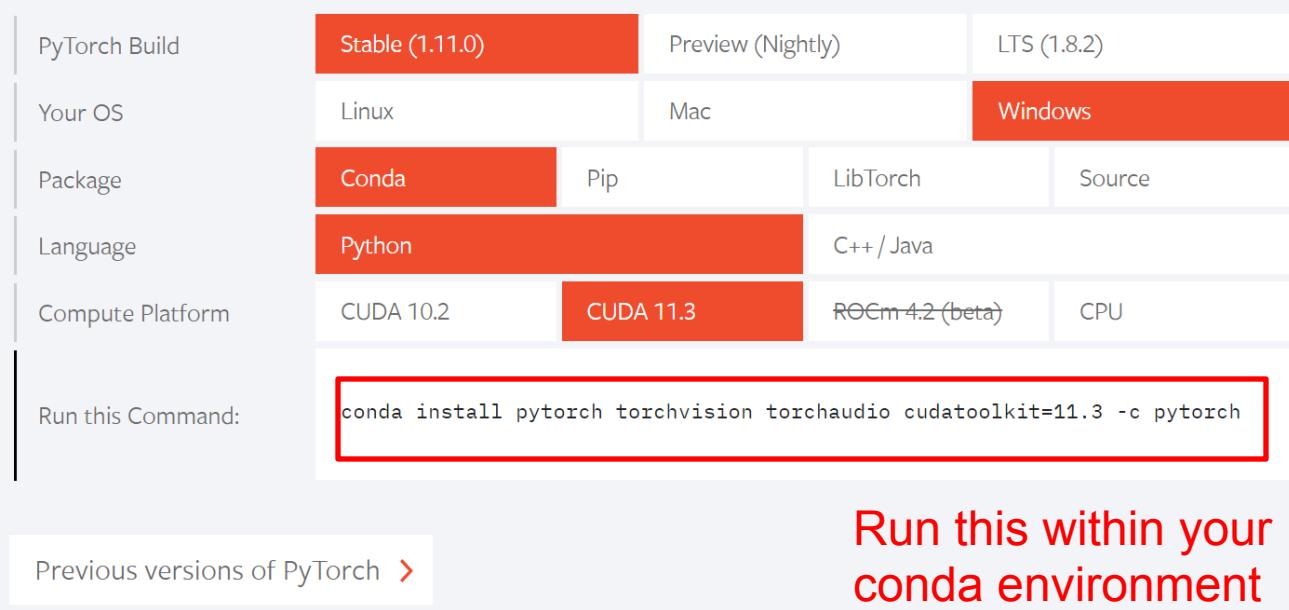
- Make sure Anaconda is installed (either on your local machine or a remote host)
 - Windows: open Anaconda Prompt
 - MacOS/Linux: open terminal
 - Type `conda --version` to check availability
- Create an environment for course project
 - `conda create -n 8056proj python=3.9.13`
 - `conda activate 8056proj`
 - `conda install numpy`
 - `conda install jupyter notebook`
 - Do some coding:
 - Launch Jupyter Notebook: `jupyter notebook`
 - Launch Python terminal: `python`
 - `conda deactivate`

Outline

- Set up working environment
- Introduction to PyTorch framework
 - Tensor, gradient and computation graph
 - Tensor manipulation on GPU
 - Use PyTorch as a general ML framework
- Deep learning with PyTorch
- Additional resources

Installation of PyTorch

- Choose configurations and install PyTorch from <http://pytorch.org/>



The image shows a screenshot of the PyTorch installation configuration interface. It's a grid-based selector with the following rows:

- PyTorch Build:** Stable (1.11.0) (highlighted in orange), Preview (Nightly), LTS (1.8.2).
- Your OS:** Linux, Mac, Windows (highlighted in orange).
- Package:** Conda (highlighted in orange), Pip, LibTorch, Source.
- Language:** Python (highlighted in orange), C++ / Java.
- Compute Platform:** CUDA 10.2, CUDA 11.3 (highlighted in orange), ROCm 4.2 (beta), CPU.

Below the grid, there is a section labeled "Run this Command:" containing the terminal command:

```
conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch
```

A red box highlights this command. At the bottom left, there is a link "Previous versions of PyTorch >". On the right side of the command section, the text "Run this within your conda environment" is displayed in red.

- If you use Google Colab, PyTorch is pre-installed with suitable configurations.

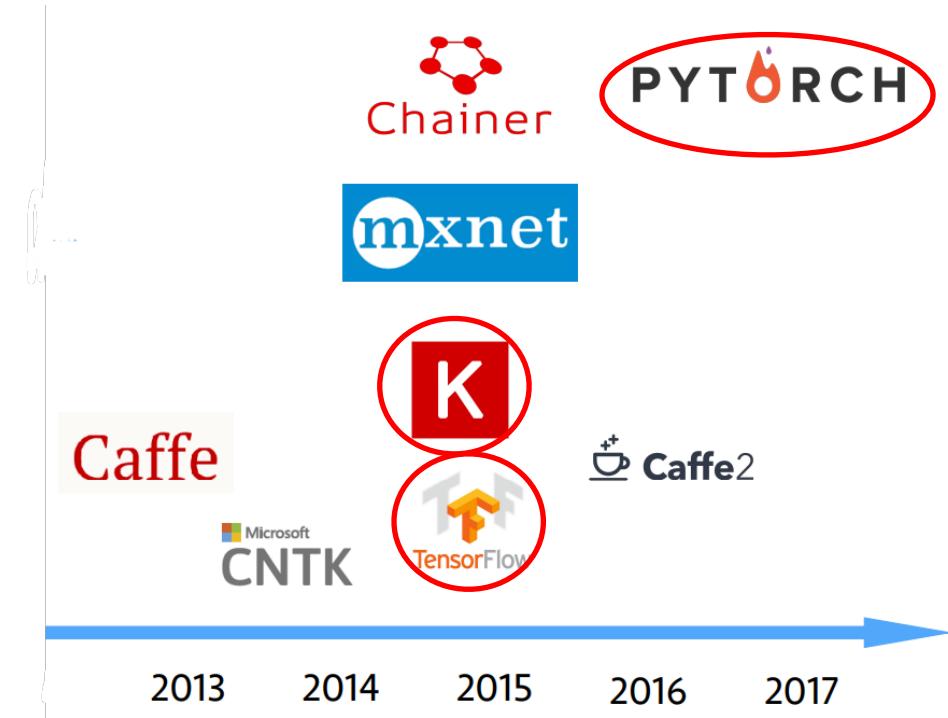
Overview of PyTorch

A fancy version of NumPy but can:

- Trace gradient via autograd
- Accelerate via GPU
- Accommodate a bunch of machine learning functionalities **including but not limited to neural network**

More importantly,

- Easy interface compared to others
- Easy to understand and debug
- Check this [Google Trends](#)



Credit: "Gluon: new MXNet interface to accelerate research"

Overview of PyTorch

Can you guess what this model is?



```
# initialize a training model
train_model = nn.Sequential(
    nn.Linear(5, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 1)
)
train_model
```

↳ Sequential(
 (0): Linear(in_features=5, out_features=256, bias=True)
 (1): ReLU()
 (2): Linear(in_features=256, out_features=128, bias=True)
 (3): ReLU()
 (4): Linear(in_features=128, out_features=1, bias=True)
)

What about this one?

```
model = nn.Sequential(
    nn.Conv2d(1, 32, 3, 1),
    nn.ReLU(),
    nn.Conv2d(32, 64, 3, 1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(0.25),
    nn.Flatten(),
    nn.Linear(9216, 128),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(128, 10),
    nn.LogSoftmax(dim = 1)
)
```



Tensor

What is a tensor?

- A multidimensional array
- Some examples:
 - 0D-tensor: scalar
 - 1D-tensor: vector
 - 2D-tensor: matrix; non-RGB image
 - 3D-tensor: RGB image (3 channels)
 - 4D-tensor: one RGB video clip
 - 5D-tensor: a collection of RGB video clips
 - 6D-tensor: ???
- Tensor in PyTorch:
 - Just like arrays in Numpy
 - `t.numpy()` and `torch.from_numpy(a)`

np.array versus torch.tensor

- Similar object creations:
 - All-ones, all-zeros, identity matrix, random...
 - Check [PyTorch tensor creations](#)
- Similar math operations:
 - Indexing, slicing, reshape, transpose, tensor product, element-wise operation...
 - Check [PyTorch tensor math operations](#)

```
import numpy as np
myarray = np.ones(3)
print(myarray)
print(myarray + 1.5)
```

[1. 1. 1.]
[2.5 2.5 2.5]

```
import torch
mytensor = torch.ones(3)
print(mytensor)
print(mytensor + 1.5)
```

tensor([1., 1., 1.])
tensor([2.5000, 2.5000, 2.5000])



Tensor (cont.)

- Enable gradient tracing:
 - Upon any tensor creation, set `requires_grad = True`
 - For any existent tensor `t`, call `t.requires_grad_(True)`
- Calculate gradient:
 - Do some calculation from `t` and get `f`
 - Call `f.backward()` for calculation
 - Call `t.grad` to access the gradient
- Disable gradient tracing:
 - (Permanently) `t = t.detach()`
 - (Temporarily) with `torch.no_grad()`:



```
# create tensors, and enable gradient tracing
c = torch.tensor(1.)
t = torch.tensor(3., requires_grad = True)
print("Tensor c:", c)
print("Tensor t:", t)

# do some calculation
s = t**2
f = 3 * s + c

# calculate the gradient of f w.r.t. t
f.backward()

# access the gradient
print("Gradient df/dt:", t.grad)
```

```
Tensor c: tensor(1.)
Tensor t: tensor(3., requires_grad=True)
Gradient df/dt: tensor(18.)
```



Dynamic computation graph

t

c



```
# create tensors, and enable gradient tracing
c = torch.tensor(1.)
t = torch.tensor(3., requires_grad = True)
print("Tensor c:", c)
print("Tensor t:", t)

# do some calculation
s = t**2
f = 3 * s + c

# calculate the gradient of f w.r.t. t
f.backward()

# access the gradient
print("Gradient df/dt:", t.grad)
```

Tensor c: tensor(1.)
Tensor t: tensor(3., requires_grad=True)
Gradient df/dt: tensor(18.)



Dynamic computation graph



```
# create tensors, and enable gradient tracing
c = torch.tensor(1.)
t = torch.tensor(3., requires_grad = True)
print("Tensor c:", c)
print("Tensor t:", t)

# do some calculation
s = t**2
f = 3 * s + c

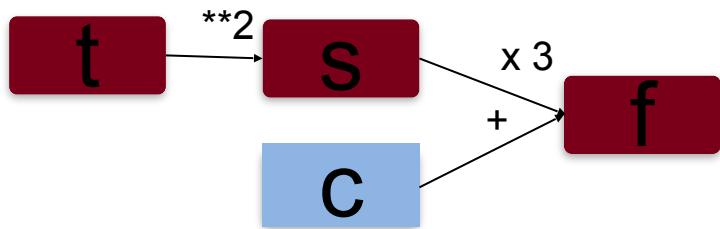
# calculate the gradient of f w.r.t. t
f.backward()

# access the gradient
print("Gradient df/dt:", t.grad)
```

Tensor c: tensor(1.)
Tensor t: tensor(3., requires_grad=True)
Gradient df/dt: tensor(18.)



Dynamic computation graph



```
# create tensors, and enable gradient tracing
c = torch.tensor(1.)
t = torch.tensor(3., requires_grad = True)
print("Tensor c:", c)
print("Tensor t:", t)

# do some calculation
s = t**2
f = 3 * s + c

# calculate the gradient of f w.r.t. t
f.backward()

# access the gradient
print("Gradient df/dt:", t.grad)
```

Tensor c: tensor(1.)
Tensor t: tensor(3., requires_grad=True)
Gradient df/dt: tensor(18.)



Dynamic computation graph



The graph structure is destroyed
Once you call `.backward()`



```
# create tensors, and enable gradient tracing
c = torch.tensor(1.)
t = torch.tensor(3., requires_grad = True)
print("Tensor c:", c)
print("Tensor t:", t)

# do some calculation
s = t**2
f = 3 * s + c

# calculate the gradient of f w.r.t. t
f.backward()

# access the gradient
print("Gradient df/dt:", t.grad)
```

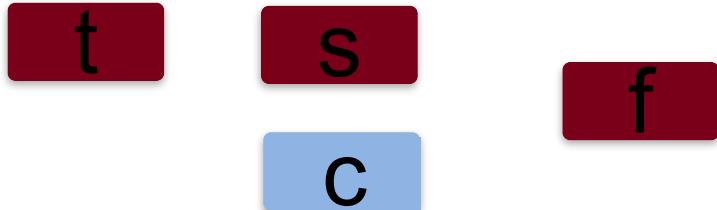
```
Tensor c: tensor(1.)
Tensor t: tensor(3., requires_grad=True)
Gradient df/dt: tensor(18.)
```



Dynamic computation graph



Access the gradient using `t.grad`



```
# create tensors, and enable gradient tracing
c = torch.tensor(1.)
t = torch.tensor(3., requires_grad = True)
print("Tensor c:", c)
print("Tensor t:", t)

# do some calculation
s = t**2
f = 3 * s + c

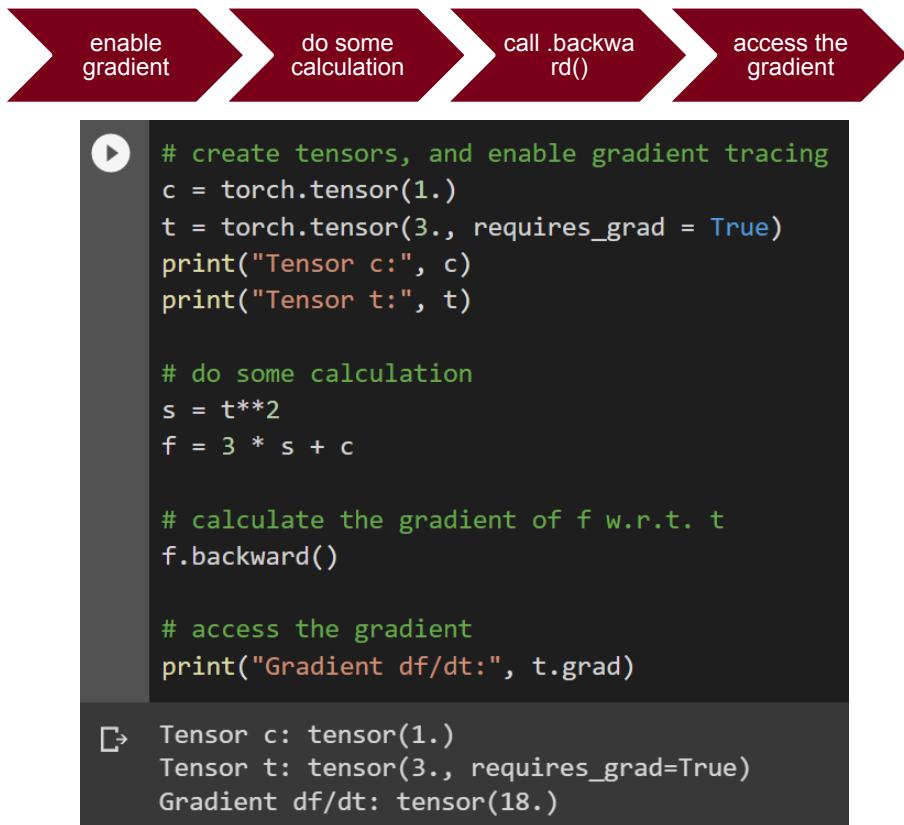
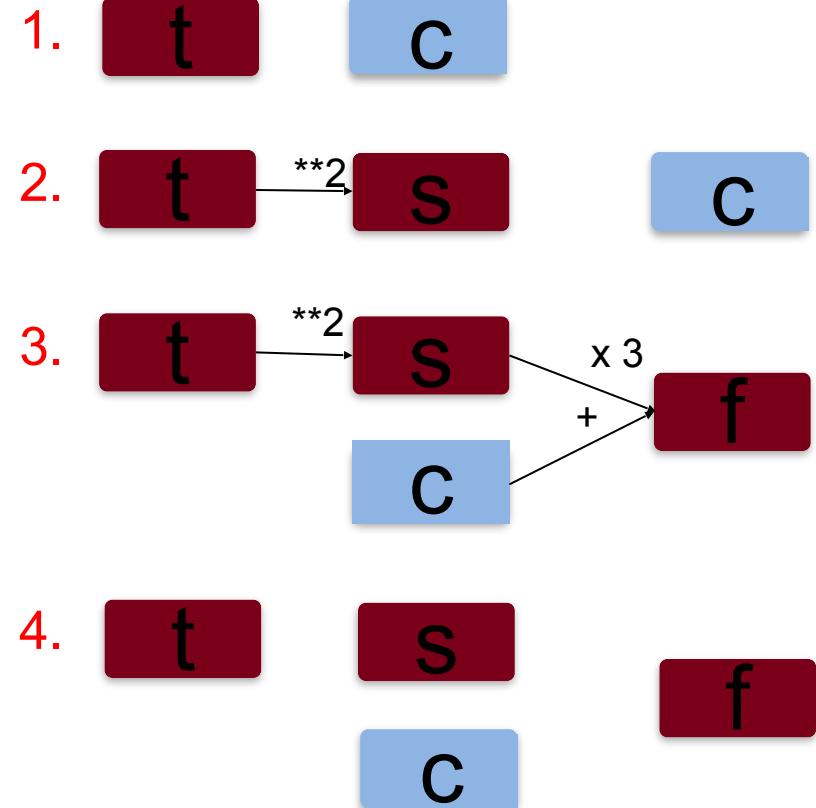
# calculate the gradient of f w.r.t. t
f.backward()

# access the gradient
print("Gradient df/dt:", t.grad)
```

```
Tensor c: tensor(1.)
Tensor t: tensor(3., requires grad=True)
Gradient df/dt: tensor(18.)
```



Dynamic computation graph



Why should we understand PyTorch computation graph?

- Essentially, PyTorch is using chain rule to calculate the gradient
- A computation graph defines how the chain rule applies to your calculation
- Common MISTAKES:
 - Call `.backward()` when there is no graph
 - Retrieve gradient of non-leaf nodes
 - Gradient accumulation. To solve this problem, call `t.grad.zero_()` before building the second graph

Illustration of gradient accumulation

```
# first back propagation
t = torch.tensor(3., requires_grad = True)
s = t**2
s.backward()
print("Gradient ds/dt:", t.grad)

# second back propagation
f = 5 * t + 1
f.backward()
print("Gradient df/dt (without emptying t.grad):", t.grad)
```

```
↳ Gradient ds/dt: tensor(6.)
Gradient df/dt (without emptying t.grad) tensor(11.)
```

should be 5

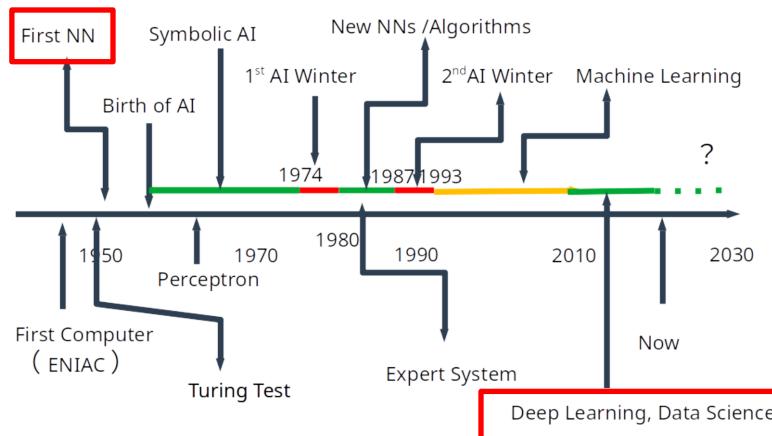
```
t.grad.zero_() # zero out the gradient
f = 5 * t + 1
f.backward()
print("Gradient df/dt (after emptying t.grad):", t.grad)
```

```
↳ Gradient df/dt (after emptying t.grad): tensor(5.)
```



Calculation on GPU

- Neural network dates back to 1950s, but became popular **only recently (last decade)** due to huge improvements in computation



Credit: CSCI 5980/8980: Think Deep Learning offered by Ju Sun, UMN

- GPU allows faster **large-scale matrix multiplication** (or tensor product)
- Use GPU on PyTorch:
 - `torch.cuda.is_available()` # GPU?
 - `t = t.cuda()` # move to GPU
 - `t = t.cpu()` # move to CPU
 - `t.device` # cpu or cuda: index
 - The transfer time can be long
 - Make sure all tensor manipulations are performed on the same device
- Advanced: [train on multiple GPUs](#)

GPU Acceleration on Apple Silicon

PyTorch: [Install PyTorch](#)

PyTorch:
[Introducing Accelerated PyTorch Training on Mac](#)

Apple:
[Accelerated PyTorch Training on Mac](#)

NOTE: Latest PyTorch requires Python 3.8 or later. For more details, see Python section below.

The screenshot shows the PyTorch build configuration interface. It includes dropdown menus for PyTorch Build (Stable 2.2.0), Your OS (Mac selected), Package (Conda selected), Language (Python selected), and Compute Platform (CUDA 11.8 selected). A command line at the bottom shows the conda install command.

PyTorch Build	Stable (2.2.0)	Preview (Nightly)		
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	CUDA 11.8	CUDA 12.1	ROcm 5.7	Default

Run this Command:
conda install pytorch::pytorch torchvision torchaudio -c pytorch

Previously
– device_name = "cuda"
– torch.cuda.is_available()

A terminal window displays the following Python code:

```
# GPU acceleration on Apple Silicon
d = 3000
device_name = "mps"

## Check whether GPU acceleration is available
torch.backends.mps.is_available()

## Create tensor on GPU
A = torch.rand(d, d, device = device_name)

## Move tensor to GPU
B = torch.rand(d, d)
B = B.to(device_name)

## Calculation
begin = time.time()
C = torch.mm(A, B)
print(f"GPU time (torch): {time.time() - begin} s")
```



Time comparison: matrix multiplication on CPU and GPU

- We want to perform matrix multiplication $C = AB$ where both A and B are 3000×3000 matrices
- Enable GPU on [Google Colab](#) (free!)
 1. Runtime
 2. Change runtime type
 3. Hardware Accelerator: GPU
 4. Runtime -> Restart Runtime
- Much faster on GPU for large problem!

```
▶ np.random.seed(8056)

d = 3000

# C = A B with numpy on CPU
A = np.random.rand(d, d)
B = np.random.rand(d, d)
begin = time.time()
C = A.dot(B)
print(f"CPU time (numpy): {time.time() - begin} s")

# C = A B with torch on CPU
A_t_cpu = torch.tensor(A)
B_t_cpu = torch.tensor(B)
begin = time.time()
C = torch.mm(A_t_cpu, B_t_cpu)
print(f"CPU time (torch): {time.time() - begin} s")

# C = A B with torch on GPU
# make sure GPU is available
A_t_gpu = torch.tensor(A).cuda()
B_t_gpu = torch.tensor(B).cuda()
begin = time.time()
C = torch.mm(A_t_gpu, B_t_gpu)
print(f"GPU time (torch): {time.time() - begin} s")

CPU time (numpy): 1.0066237449645996 s
CPU time (torch): 0.8800492286682129 s
GPU time (torch): 0.0038247108459472656 s
```



GPU not always better

- Calculation on GPU should be optimized smartly since
 - GPU has **limited memory** compared to CPU
 - Transfer/overhead time from CPU to GPU can be large

```
d = 3000
begin = time.time()
A = torch.rand((d, d)).cuda()
B = torch.rand((d, d)).cuda()
print(f"Time from CPU to GPU: {time.time() - begin} s")
begin = time.time()
C = torch.mm(A, B)
print(f"GPU time (torch): {time.time() - begin} s")
```

Time from CPU to GPU: 0.15282464027404785 s
GPU time (torch): 0.0008263587951660156 s

NVIDIA-SMI 440.100			Driver Version: 440.100		CUDA Version: 10.2		
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
	Fan	Temp			Pwr:Usage/Cap	Memory-Usage	GPU-Util
0	TITAN RTX	Off	00000000:1A:00.0 Off		13556MiB / 24220MiB	61%	N/A
1	TITAN RTX	Off	00000000:1B:00.0 Off		18798MiB / 24220MiB	60%	N/A
2	TITAN RTX	Off	00000000:3D:00.0 Off		13616MiB / 24220MiB	62%	N/A
3	TITAN RTX	Off	00000000:3E:00.0 Off		20962MiB / 24220MiB	61%	N/A
4	TITAN RTX	Off	00000000:88:00.0 Off		13590MiB / 24220MiB	92%	N/A
5	TITAN RTX	Off	00000000:89:00.0 Off		18832MiB / 24220MiB	61%	N/A
6	TITAN RTX	Off	00000000:B1:00.0 Off		13650MiB / 24220MiB	90%	N/A
7	TITAN RTX	Off	00000000:B2:00.0 Off		20996MiB / 24220MiB	91%	N/A



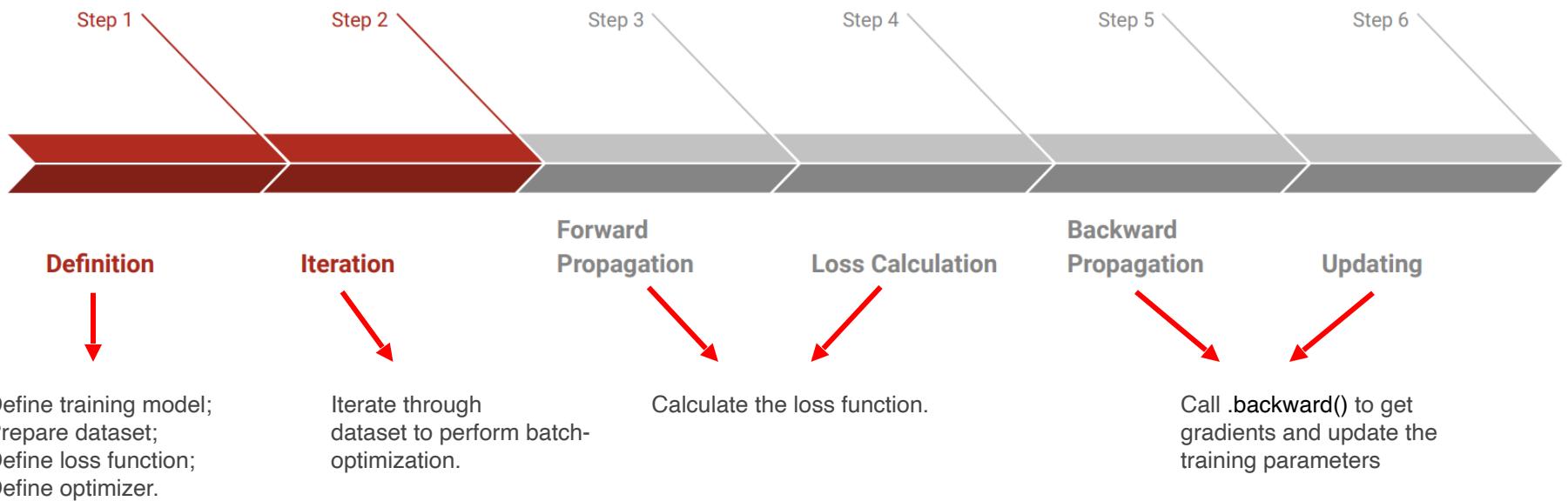
ML in PyTorch

- PyTorch provides a framework for general ML modeling **including but not limited to deep learning**
- What is a common ML pipeline?
 - Prepare your dataset (batches)
 - Choose/design a training model
 - Choose a loss/objective function
 - Optimization (calculation; gradient)
 - Evaluation and inference
- The go-to optimization method: SGD
 - Instead of using gradient calculated from all training samples, we **only use the gradient from a randomly chosen sample**
- Practical one: (Mini-) Batch SGD
 - Use gradient from **a batch of samples**
 - **Batch size:** # samples in one batch
 - **Epoch:** a full pass of all training samples
 - Special cases:
 - Batch size = 1: vanilla SGD
 - Batch size = n: GD



ML in PyTorch

Credit: HPRC Short Course by Jian Tao, TAMU



Linear regression with SGD

– The naive way

- The true model: $y = 2 * x + 1$
- Some other setups:
 - $n = 2000$, $x \sim \text{Uniform}(0, 1)$
 - Number of epochs: 10
 - Batch size: 200 (10 batches per epoch)
 - Learning rate: 0.05
 - Initialize both intercept and slope with $\text{Uniform}(0, 1)$



```
n = 2000
x = torch.rand(n)

# create dataset with true model
a0 = 2
b0 = 1
y = a0 * x + b0

# set up optimization parameter of SGD
a = torch.rand(1, requires_grad = True)
b = torch.rand(1, requires_grad = True)
nepochs = 10
batch_size = 200
lr = 0.5
```





```
for epoch in range(nepochs):
    for batch in range(round(n / batch_size)):
        start = batch * batch_size
        end = start + batch_size
        # perform update on a batch
        x_batch = x[start:end]
        y_batch = y[start:end]

        # build computation graph
        y_hat = a * x_batch + b
        myloss = torch.mean((y_batch - y_hat)**2)

        # gradient calculation
        myloss.backward()

        # SGD update
        with torch.no_grad():
            a -= lr * a.grad
            b -= lr * b.grad

        # avoid gradient accumulation
        a.grad.zero_()
        b.grad.zero_()

    print(f"Epoch: {epoch + 1} / {nepochs}")
    print(f"MSE: {myloss: .2e}; a: {a.item():.4f}; b: {b.item():.4f}")
```

Outer iteration
of SGD (epoch)

Linear regression with SGD – The naive way

Output:

```
↳ Epoch: 1 / 10
MSE: 3.69e-02; a: 1.4460; b: 1.2822
Epoch: 2 / 10
MSE: 9.24e-03; a: 1.7226; b: 1.1413
Epoch: 3 / 10
MSE: 2.32e-03; a: 1.8611; b: 1.0707
Epoch: 4 / 10
MSE: 5.81e-04; a: 1.9305; b: 1.0354
Epoch: 5 / 10
MSE: 1.46e-04; a: 1.9652; b: 1.0177
Epoch: 6 / 10
MSE: 3.65e-05; a: 1.9826; b: 1.0089
Epoch: 7 / 10
MSE: 9.15e-06; a: 1.9913; b: 1.0044
Epoch: 8 / 10
MSE: 2.29e-06; a: 1.9956; b: 1.0022
Epoch: 9 / 10
MSE: 5.75e-07; a: 1.9978; b: 1.0011
Epoch: 10 / 10
MSE: 1.44e-07; a: 1.9989; b: 1.0006
```





```
for epoch in range(nepochs):
    for batch in range(round(n / batch_size)):
        start = batch * batch_size
        end = start + batch_size
        # perform update on a batch
        x_batch = x[start:end]
        y_batch = y[start:end]

        # build computation graph
        y_hat = a * x_batch + b
        myloss = torch.mean((y_batch - y_hat)**2)

        # gradient calculation
        myloss.backward()

        # SGD update
        with torch.no_grad():
            a -= lr * a.grad
            b -= lr * b.grad

        # avoid gradient accumulation
        a.grad.zero_()
        b.grad.zero_()

    print(f"Epoch: {epoch + 1} / {nepochs}")
    print(f"MSE: {myloss: .2e}; a: {a.item():.4f}; b: {b.item():.4f}")
```

Inner iteration
of SGD (batch)

Linear regression with SGD – The naive way

Output:

```
↳ Epoch: 1 / 10
MSE: 3.69e-02; a: 1.4460; b: 1.2822
Epoch: 2 / 10
MSE: 9.24e-03; a: 1.7226; b: 1.1413
Epoch: 3 / 10
MSE: 2.32e-03; a: 1.8611; b: 1.0707
Epoch: 4 / 10
MSE: 5.81e-04; a: 1.9305; b: 1.0354
Epoch: 5 / 10
MSE: 1.46e-04; a: 1.9652; b: 1.0177
Epoch: 6 / 10
MSE: 3.65e-05; a: 1.9826; b: 1.0089
Epoch: 7 / 10
MSE: 9.15e-06; a: 1.9913; b: 1.0044
Epoch: 8 / 10
MSE: 2.29e-06; a: 1.9956; b: 1.0022
Epoch: 9 / 10
MSE: 5.75e-07; a: 1.9978; b: 1.0011
Epoch: 10 / 10
MSE: 1.44e-07; a: 1.9989; b: 1.0006
```





```
for epoch in range(nepochs):
    for batch in range(round(n / batch_size)):
        start = batch * batch_size
        end = start + batch_size
        # perform update on a batch
        x_batch = x[start:end]
        y_batch = y[start:end]

        # build computation graph
        y_hat = a * x_batch + b
        myloss = torch.mean((y_batch - y_hat)**2)

        # gradient calculation
        myloss.backward()

        # SGD update
        with torch.no_grad():
            a -= lr * a.grad
            b -= lr * b.grad

        # avoid gradient accumulation
        a.grad.zero_()
        b.grad.zero_()

    print(f"Epoch: {epoch + 1} / {nepochs}")
    print(f"MSE: {myloss:.2e}; a: {a.item():.4f}; b: {b.item():.4f}")
```

1. Use a batch of data to build a computation graph

Linear regression with SGD – The naive way

Output:

```
↳ Epoch: 1 / 10
MSE: 3.69e-02; a: 1.4460; b: 1.2822
Epoch: 2 / 10
MSE: 9.24e-03; a: 1.7226; b: 1.1413
Epoch: 3 / 10
MSE: 2.32e-03; a: 1.8611; b: 1.0707
Epoch: 4 / 10
MSE: 5.81e-04; a: 1.9305; b: 1.0354
Epoch: 5 / 10
MSE: 1.46e-04; a: 1.9652; b: 1.0177
Epoch: 6 / 10
MSE: 3.65e-05; a: 1.9826; b: 1.0089
Epoch: 7 / 10
MSE: 9.15e-06; a: 1.9913; b: 1.0044
Epoch: 8 / 10
MSE: 2.29e-06; a: 1.9956; b: 1.0022
Epoch: 9 / 10
MSE: 5.75e-07; a: 1.9978; b: 1.0011
Epoch: 10 / 10
MSE: 1.44e-07; a: 1.9989; b: 1.0006
```





```
for epoch in range(nepochs):
    for batch in range(round(n / batch_size)):
        start = batch * batch_size
        end = start + batch_size
        # perform update on a batch
        x_batch = x[start:end]
        y_batch = y[start:end]

        # build computation graph
        y_hat = a * x_batch + b
        myloss = torch.mean((y_batch - y_hat)**2)

        # gradient calculation
        myloss.backward()

        # SGD update
        with torch.no_grad():
            a -= lr * a.grad
            b -= lr * b.grad

        # avoid gradient accumulation
        a.grad.zero_()
        b.grad.zero_()

    print(f"Epoch: {epoch + 1} / {nepochs}")
    print(f"MSE: {myloss:.2e}; a: {a.item():.4f}; b: {b.item():.4f}")
```

2. Calculate gradient based on the computation graph

Linear regression with SGD – The naive way

Output:

```
Epoch: 1 / 10
MSE: 3.69e-02; a: 1.4460; b: 1.2822
Epoch: 2 / 10
MSE: 9.24e-03; a: 1.7226; b: 1.1413
Epoch: 3 / 10
MSE: 2.32e-03; a: 1.8611; b: 1.0707
Epoch: 4 / 10
MSE: 5.81e-04; a: 1.9305; b: 1.0354
Epoch: 5 / 10
MSE: 1.46e-04; a: 1.9652; b: 1.0177
Epoch: 6 / 10
MSE: 3.65e-05; a: 1.9826; b: 1.0089
Epoch: 7 / 10
MSE: 9.15e-06; a: 1.9913; b: 1.0044
Epoch: 8 / 10
MSE: 2.29e-06; a: 1.9956; b: 1.0022
Epoch: 9 / 10
MSE: 5.75e-07; a: 1.9978; b: 1.0011
Epoch: 10 / 10
MSE: 1.44e-07; a: 1.9989; b: 1.0006
```





```
for epoch in range(nepochs):
    for batch in range(round(n / batch_size)):
        start = batch * batch_size
        end = start + batch_size
        # perform update on a batch
        x_batch = x[start:end]
        y_batch = y[start:end]

        # build computation graph
        y_hat = a * x_batch + b
        myloss = torch.mean((y_batch - y_hat)**2)

        # gradient calculation
        myloss.backward()

        # SGD update
        with torch.no_grad():
            a -= lr * a.grad
            b -= lr * b.grad

        # avoid gradient accumulation
        a.grad.zero_()
        b.grad.zero_()

print(f"Epoch: {epoch + 1} / {nepochs}")
print(f"MSE: {myloss:.2e}; a: {a.item():.4f}; b: {b.item():.4f}")
```

3. Update parameters and zero out gradients

Linear regression with SGD – The naive way

Output:

```
↳ Epoch: 1 / 10
MSE: 3.69e-02; a: 1.4460; b: 1.2822
Epoch: 2 / 10
MSE: 9.24e-03; a: 1.7226; b: 1.1413
Epoch: 3 / 10
MSE: 2.32e-03; a: 1.8611; b: 1.0707
Epoch: 4 / 10
MSE: 5.81e-04; a: 1.9305; b: 1.0354
Epoch: 5 / 10
MSE: 1.46e-04; a: 1.9652; b: 1.0177
Epoch: 6 / 10
MSE: 3.65e-05; a: 1.9826; b: 1.0089
Epoch: 7 / 10
MSE: 9.15e-06; a: 1.9913; b: 1.0044
Epoch: 8 / 10
MSE: 2.29e-06; a: 1.9956; b: 1.0022
Epoch: 9 / 10
MSE: 5.75e-07; a: 1.9978; b: 1.0011
Epoch: 10 / 10
MSE: 1.44e-07; a: 1.9989; b: 1.0006
```



Linear regression with SGD – The PyTorch way

Built-in functionalities

- Dataset preparation:
 - [torch.utils.data](#)
 - Check [Datasets & DataLoaders](#)
- Define training model:
 - Check [torch.nn](#) for all kinds of components to build your own model
- Optimization algorithms:
 - Check [torch.optim](#) for various opt methods

```
▶ from torch import nn, optim
    from torch.utils.data import Dataset, DataLoader
```

```
▶ class SimpleLinear(Dataset):
    def __init__(self, n, a, b):
        self.n = n
        self.x = torch.rand(n, 1)
        self.y = a * self.x + b

    def __len__(self):
        return self.n

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

# create a training model
# (cont.) also initializes a and b
mymodel = nn.Sequential(nn.Linear(1, 1))

# prepare dataset and dataloaders
mydata = SimpleLinear(n = 2000, a = 2, b = 1)
mydataloader = DataLoader(mydata, batch_size = 200)

# set up optimization with SGD
criterion = nn.MSELoss()
optimizer = optim.SGD(mymodel.parameters(), lr = 0.5)
```



Linear regression with SGD – The PyTorch way

Prepare our datasets

- Dataset (a Python class)
 - `__init__`: initialize the dataset
 - `__len__`: sample size of the dataset
 - `__getitem__`: fetch a sample with idx
 - As the input of `DataLoader` function
- `DataLoader` (a Python function)
 - Split the dataset into batches
 - Check [this](#) for more advanced usages
 - Have some tricks to reduce overhead time when transferring data to GPU

```
▶ class SimpleLinear(Dataset):  
    def __init__(self, n, a, b):  
        self.n = n  
        self.x = torch.rand(n, 1)  
        self.y = a * self.x + b  
  
    def __len__(self):  
        return self.n  
  
    def __getitem__(self, idx):  
        return self.x[idx], self.y[idx]  
  
# create a training model  
# (cont.) also initializes a and b  
mymodel = nn.Sequential(nn.Linear(1, 1))  
  
# prepare dataset and dataloaders  
mydata = SimpleLinear(n = 2000, a = 2, b = 1)  
mydataloader = DataLoader(mydata, batch_size = 200)  
  
# set up optimization with SGD  
criterion = nn.MSELoss()  
optimizer = optim.SGD(mymodel.parameters(), lr = 0.5)
```



Linear regression with SGD – The PyTorch way

Create the training model:

- nn.Sequential
 - Build a model with sequential operations
 - nn.Linear(m, n)
 - A **linear operator** of shape $n \times m$
 - A **bias vector** of shape $n \times 1$ (default)
 - All parameters are **initialized automatically upon creation**
 - Check [torch.nn](#) for other operations

```
▶ class SimpleLinear(Dataset):  
    def __init__(self, n, a, b):  
        self.n = n  
        self.x = torch.rand(n, 1)  
        self.y = a * self.x + b  
  
    def __len__(self):  
        return self.n  
  
    def __getitem__(self, idx):  
        return self.x[idx], self.y[idx]  
  
# create a training model  
# (cont.) also initializes a and b  
mymodel = nn.Sequential(nn.Linear(1, 1))  
  
# prepare dataset and dataloaders  
mydata = SimpleLinear(n = 2000, a = 2, b = 1)  
mydataloader = DataLoader(mydata, batch_size = 200)  
  
# set up optimization with SGD  
criterion = nn.MSELoss()  
optimizer = optim.SGD(mymodel.parameters(), lr = 0.5)
```



Linear regression with SGD – The PyTorch way

Set up optimization:

- The loss function
- The optimization method
- Optimization parameters

```
▶ class SimpleLinear(Dataset):  
    def __init__(self, n, a, b):  
        self.n = n  
        self.x = torch.rand(n, 1)  
        self.y = a * self.x + b  
  
    def __len__(self):  
        return self.n  
  
    def __getitem__(self, idx):  
        return self.x[idx], self.y[idx]  
  
# create a training model  
# (cont.) also initializes a and b  
mymodel = nn.Sequential(nn.Linear(1, 1))  
  
# prepare dataset and dataloaders  
mydata = SimpleLinear(n = 2000, a = 2, b = 1)  
mydataloader = DataLoader(mydata, batch_size = 200)  
  
# set up optimization with SGD  
criterion = nn.MSELoss()  
optimizer = optim.SGD(mymodel.parameters(), lr = 0.5)
```



Linear regression with SGD

– The PyTorch way



```
nepochs = 10
for epoch in range(nepochs):
    for x_batch, y_batch in mydataloader:
        # build computation graph
        yhat = mymodel(x_batch)
        myloss = criterion(y_batch, yhat)

        Build computation graph
        # optimization
        optimizer.zero_grad() # zero out gradient
        myloss.backward() # back propagation
        optimizer.step() # update parameters
```

Output:

```
↳ Epoch: 1 / 10
MSE: 3.52e-02; a: 1.3904; b: 1.3368
Epoch: 2 / 10
MSE: 9.21e-03; a: 1.6882; b: 1.1722
Epoch: 3 / 10
MSE: 2.41e-03; a: 1.8406; b: 1.0881
Epoch: 4 / 10
MSE: 6.30e-04; a: 1.9185; b: 1.0450
Epoch: 5 / 10
MSE: 1.65e-04; a: 1.9583; b: 1.0230
Epoch: 6 / 10
MSE: 4.31e-05; a: 1.9787; b: 1.0118
Epoch: 7 / 10
MSE: 1.13e-05; a: 1.9891; b: 1.0060
Epoch: 8 / 10
MSE: 2.95e-06; a: 1.9944; b: 1.0031
Epoch: 9 / 10
MSE: 7.71e-07; a: 1.9971; b: 1.0016
Epoch: 10 / 10
MSE: 2.02e-07; a: 1.9985; b: 1.0008
```



Linear regression with SGD

– The PyTorch way



```
nepochs = 10
for epoch in range(nepochs):
    for x_batch, y_batch in mydataloader:
        # build computation graph
        yhat = mymodel(x_batch)
        myloss = criterion(y_batch, yhat)

        # optimization
        optimizer.zero_grad() # zero out gradient
        myloss.backward() # back propagation
        optimizer.step() # update parameters
```

The whole
optimization part

Output:

```
↳ Epoch: 1 / 10
MSE: 3.52e-02; a: 1.3904; b: 1.3368
Epoch: 2 / 10
MSE: 9.21e-03; a: 1.6882; b: 1.1722
Epoch: 3 / 10
MSE: 2.41e-03; a: 1.8406; b: 1.0881
Epoch: 4 / 10
MSE: 6.30e-04; a: 1.9185; b: 1.0450
Epoch: 5 / 10
MSE: 1.65e-04; a: 1.9583; b: 1.0230
Epoch: 6 / 10
MSE: 4.31e-05; a: 1.9787; b: 1.0118
Epoch: 7 / 10
MSE: 1.13e-05; a: 1.9891; b: 1.0060
Epoch: 8 / 10
MSE: 2.95e-06; a: 1.9944; b: 1.0031
Epoch: 9 / 10
MSE: 7.71e-07; a: 1.9971; b: 1.0016
Epoch: 10 / 10
MSE: 2.02e-07; a: 1.9985; b: 1.0008
```



Linear regression with SGD

– Comparison of two ways

The PyTorch way

```
nepochs = 10
for epoch in range(nepochs):
    for x_batch, y_batch in mydataloader:
        # build computation graph
        yhat = mymodel(x_batch)
        myloss = criterion(y_batch, yhat)

        # optimization
        optimizer.zero_grad() # zero out gradient
        myloss.backward() # back propagation
        optimizer.step() # update parameters
```



Agnostic to datasets, models,
losses and optimization methods!

The naive way

```
for epoch in range(nepochs):
    for batch in range(round(n / batch_size)):
        start = batch * batch_size
        end = start + batch_size
        # perform update on a batch
        x_batch = x[start:end]
        y_batch = y[start:end]

        # build computation graph
        y_hat = a * x_batch + b
        myloss = torch.mean((y_batch - y_hat)**2)

        # gradient calculation
        myloss.backward()

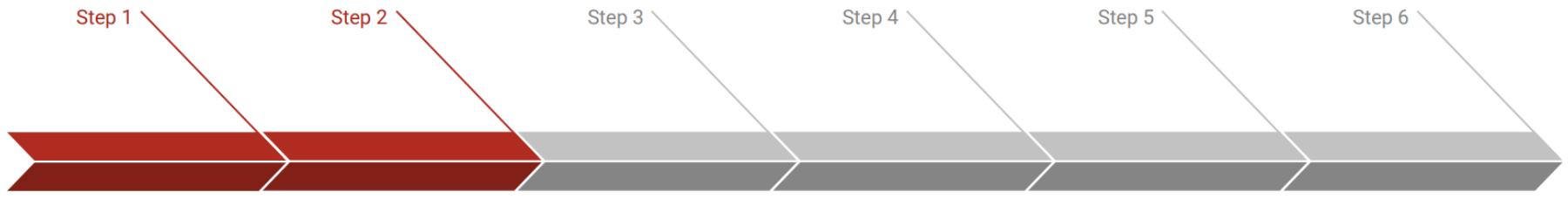
        # SGD update
        with torch.no_grad():
            a -= lr * a.grad
            b -= lr * b.grad

        # avoid gradient accumulation
        a.grad.zero_()
        b.grad.zero_()
```



ML workflow in PyTorch

Credit: HPRC Short Course by Jian Tao, TAMU



Definition

- Define training model using `torch.nn`;
- Define Dataset using `Dataset` class;
- Define loss function;
- Define optimizer with `torch.optim`

Iteration

- Iterate through dataset using `DataLoader` function; (to perform batch-optimization)

Forward Propagation

```
# forward pass  
yhat = mymodel(inputs)  
myloss = criterion(labels, yhat)
```

Loss Calculation

```
# zero out the gradient  
optimizer.zero_grad()  
# backward propagation  
myloss.backward()  
# update the model  
optimizer.step()
```

Backward Propagation

Updating



Outline

- Set up working environment
- Introduction to PyTorch framework
- Deep learning with PyTorch
 - Regression using ReLU neural net
 - Image classification with CNN
- Additional resources



Recap

- Use **Anaconda** to manage multiple Python projects at the same time:
 - Ensure compacity between libraries;
 - Use it with GUI or from a shell.
- Use Google Colab (cloud service) for notebook-style coding:
 - One free GPU available;
 - Runs up to 12 hours;
 - In sync with Google Drive;
 - Collaborate with your teammates.
- DL framework PyTorch:
 - User-friendly interface;
 - Gradient tracing via computation graph;
 - GPU acceleration.
- General ML/DL workflow:
 - Prepare your dataset (batches);
 - Choose/design a training model;
 - Choose a loss/objective function;
 - Optimization (calculation; gradient);
 - Evaluation and inference.



Recap:

Linear regression with SGD

– The PyTorch way

Prepare our datasets

- Dataset (a Python class)
 - `__init__`: initialize the dataset
 - `__len__`: sample size of the dataset
 - `__getitem__`: fetch a sample with idx
 - As the input of `DataLoader` function
- `DataLoader` (a Python function)
 - Split the dataset into batches
 - Check [this](#) for more advanced usages
 - Have some tricks to reduce overhead time when transferring data to GPU

```
class SimpleLinear(Dataset):  
    def __init__(self, n, a, b):  
        self.n = n  
        self.x = torch.rand(n, 1)  
        self.y = a * self.x + b  
  
    def __len__(self):  
        return self.n  
  
    def __getitem__(self, idx):  
        return self.x[idx], self.y[idx]  
  
# create a training model  
# (cont.) also initializes a and b  
mymodel = nn.Sequential(nn.Linear(1, 1))  
  
# prepare dataset and dataloaders  
mydata = SimpleLinear(n = 2000, a = 2, b = 1)  
mydataloader = DataLoader(mydata, batch_size = 200)  
  
# set up optimization with SGD  
criterion = nn.MSELoss()  
optimizer = optim.SGD(mymodel.parameters(), lr = 0.5)
```



Recap:

Linear regression with SGD – The PyTorch way

Create the training model:

- **nn.Sequential**
 - Build a model with sequential operations
 - **nn.Linear(m, n)**
 - A **linear operator** of shape $n \times m$
 - A **bias vector** of shape $n \times 1$ (default)
 - All parameters are **initialized automatically upon creation**
 - Check [torch.nn](#) for other operations



```
class SimpleLinear(Dataset):  
    def __init__(self, n, a, b):  
        self.n = n  
        self.x = torch.rand(n, 1)  
        self.y = a * self.x + b  
  
    def __len__(self):  
        return self.n  
  
    def __getitem__(self, idx):  
        return self.x[idx], self.y[idx]  
  
# create a training model  
# (cont.) also initializes a and b  
mymodel = nn.Sequential(nn.Linear(1, 1))  
  
# prepare dataset and dataloaders  
mydata = SimpleLinear(n = 2000, a = 2, b = 1)  
mydataloader = DataLoader(mydata, batch_size = 200)  
  
# set up optimization with SGD  
criterion = nn.MSELoss()  
optimizer = optim.SGD(mymodel.parameters(), lr = 0.5)
```



Recap: Linear regression with SGD – The PyTorch way

Set up optimization:

- The loss function
- The optimization method
- Optimization parameters

```
▶ class SimpleLinear(Dataset):  
    def __init__(self, n, a, b):  
        self.n = n  
        self.x = torch.rand(n, 1)  
        self.y = a * self.x + b  
  
    def __len__(self):  
        return self.n  
  
    def __getitem__(self, idx):  
        return self.x[idx], self.y[idx]  
  
# create a training model  
# (cont.) also initializes a and b  
mymodel = nn.Sequential(nn.Linear(1, 1))  
  
# prepare dataset and dataloaders  
mydata = SimpleLinear(n = 2000, a = 2, b = 1)  
mydataloader = DataLoader(mydata, batch_size = 200)  
  
# set up optimization with SGD  
criterion = nn.MSELoss()  
optimizer = optim.SGD(mymodel.parameters(), lr = 0.5)
```



Recap:

Linear regression with SGD – The PyTorch way



```
nepochs = 10
for epoch in range(nepochs):
    for x_batch, y_batch in mydataloader:
        # build computation graph
        yhat = mymodel(x_batch)
        myloss = criterion(y_batch, yhat)

        Build computation graph
        # optimization
        optimizer.zero_grad() # zero out gradient
        myloss.backward() # back propagation
        optimizer.step() # update parameters
```

Recap:

Linear regression with SGD – The PyTorch way



```
nepochs = 10
for epoch in range(nepochs):
    for x_batch, y_batch in mydataloader:
        # build computation graph
        yhat = mymodel(x_batch)
        myloss = criterion(y_batch, yhat)

        # optimization
        optimizer.zero_grad() # zero out gradient
        myloss.backward() # back propagation
        optimizer.step() # update parameters
```

The whole
optimization part



Recap:

Linear regression with SGD

– Comparison of two ways

The PyTorch way

```
nepochs = 10
for epoch in range(nepochs):
    for x_batch, y_batch in mydataloader:
        # build computation graph
        yhat = mymodel(x_batch)
        myloss = criterion(y_batch, yhat)

        # optimization
        optimizer.zero_grad() # zero out gradient
        myloss.backward() # back propagation
        optimizer.step() # update parameters
```



Agnostic to datasets, models,
losses and optimization methods!

The naive way

```
for epoch in range(nepochs):
    for batch in range(round(n / batch_size)):
        start = batch * batch_size
        end = start + batch_size
        # perform update on a batch
        x_batch = x[start:end]
        y_batch = y[start:end]

        # build computation graph
        y_hat = a * x_batch + b
        myloss = torch.mean((y_batch - y_hat)**2)

        # gradient calculation
        myloss.backward()

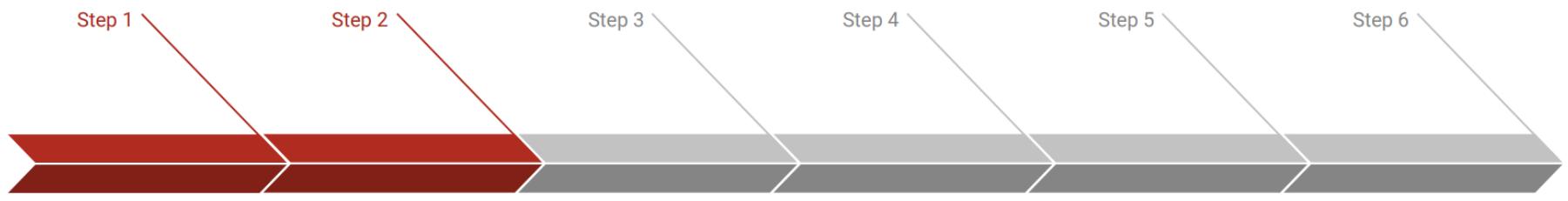
        # SGD update
        with torch.no_grad():
            a -= lr * a.grad
            b -= lr * b.grad

        # avoid gradient accumulation
        a.grad.zero_()
        b.grad.zero_()
```



Recap: ML workflow in PyTorch

Credit: HPRC Short Course by Jian Tao, TAMU



Definition

- Define training model using `torch.nn`;
- Define Dataset using `Dataset` class;
- Define loss function;
- Define optimizer with `torch.optim`

Iteration

- Iterate through dataset using `DataLoader` function; (to perform batch-optimization)

Forward Propagation

```
# forward pass  
yhat = mymodel(inputs)  
myloss = criterion(labels, yhat)
```

Loss Calculation

```
# zero out the gradient  
optimizer.zero_grad()  
# backward propagation  
myloss.backward()  
# update the model  
optimizer.step()
```

Backward Propagation

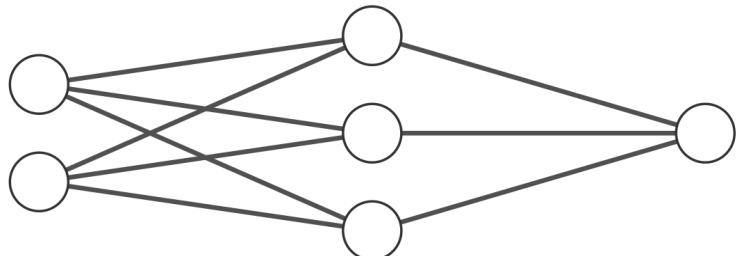
Updating



From linear model to shallow ReLU network

Usage: `nn.Linear(in_features, out_features)`

Shallow ReLU neural net



Input Layer $\in \mathbb{R}^2$

Hidden Layer $\in \mathbb{R}^3$

Output Layer $\in \mathbb{R}^1$

$$\text{ReLU}nn2: \mathbf{x} \in \mathbb{R}^2 \mapsto \mathbf{w}_2^\top (\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)_+ + b_2 \in \mathbb{R}$$

$$\mathbf{W}_1 \in \mathbb{R}^{3 \times 2}, \mathbf{b}_1 \in \mathbb{R}^3, \mathbf{w}_2 \in \mathbb{R}^3, b_2 \in \mathbb{R}$$

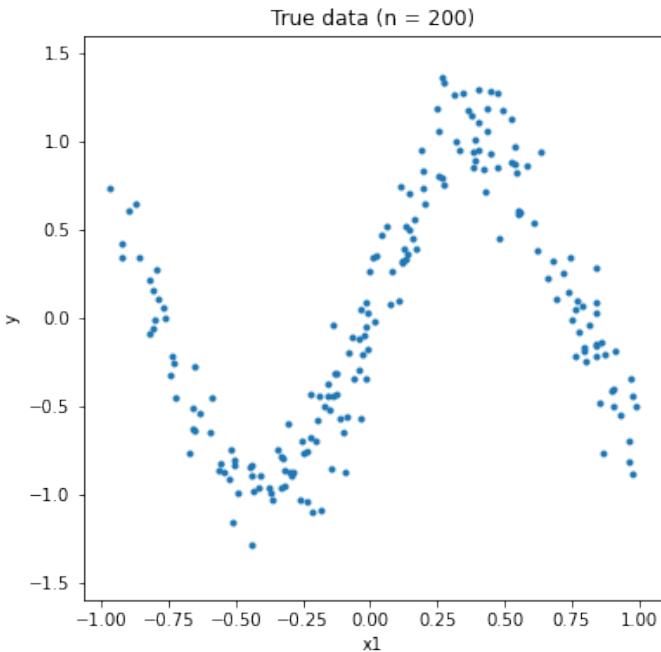
```
# shallow relu net
ReLU(nn2 = nn.Sequential(
    nn.Linear(2, 3),
    nn.ReLU(),
    nn.Linear(3, 1)
)
for param in ReLU(nn2).parameters():
    print(type(param.data), param.shape)
```

```
→ <class 'torch.Tensor'> torch.Size([3, 2])
→ <class 'torch.Tensor'> torch.Size([3])
→ <class 'torch.Tensor'> torch.Size([1, 3])
→ <class 'torch.Tensor'> torch.Size([1])
```



Fit a ReLU neural network to Sine function

What dose the data look like?



$$y = 4 \sin(x_1) + \mathcal{N}(0, 1) \text{ where } \mathbf{x} \sim \text{Unif}([-1, 1]^5)$$

What is our training model?

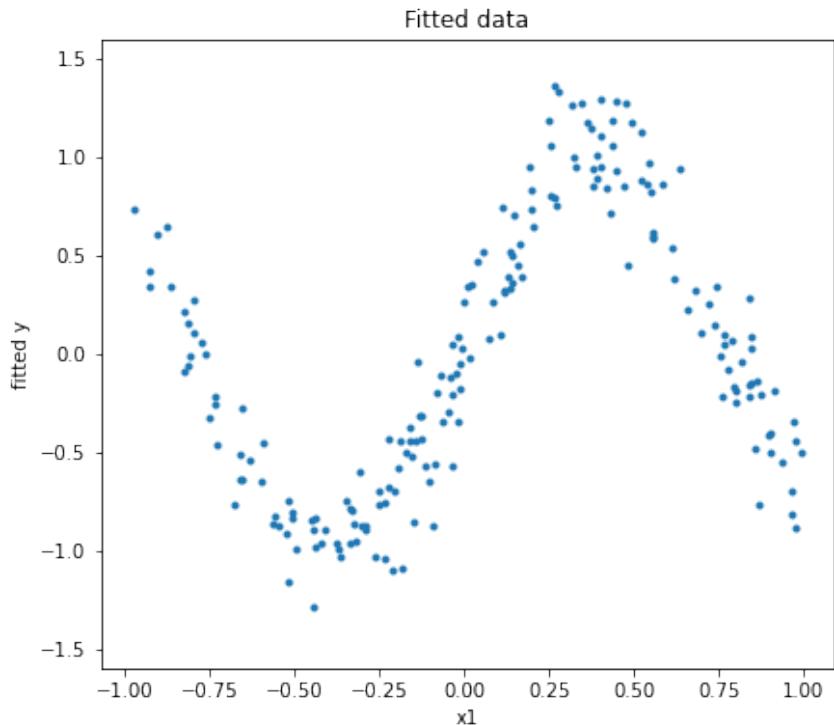
```
# initialize a training model
train_model = nn.Sequential(
    nn.Linear(5, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 1)
)
train_model
```

↳ Sequential(
 (0): Linear(in_features=5, out_features=256, bias=True)
 (1): ReLU()
 (2): Linear(in_features=256, out_features=128, bias=True)
 (3): ReLU()
 (4): Linear(in_features=128, out_features=1, bias=True)
)

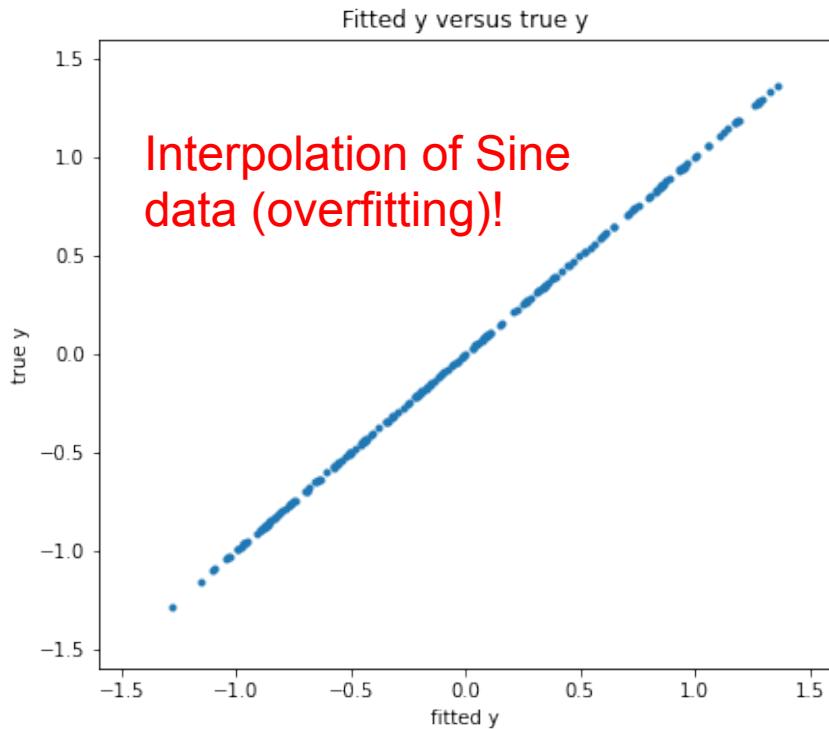
- A two-layer ReLU neural network with 256 hidden units in the first layer, and 128 hidden units in the second layer



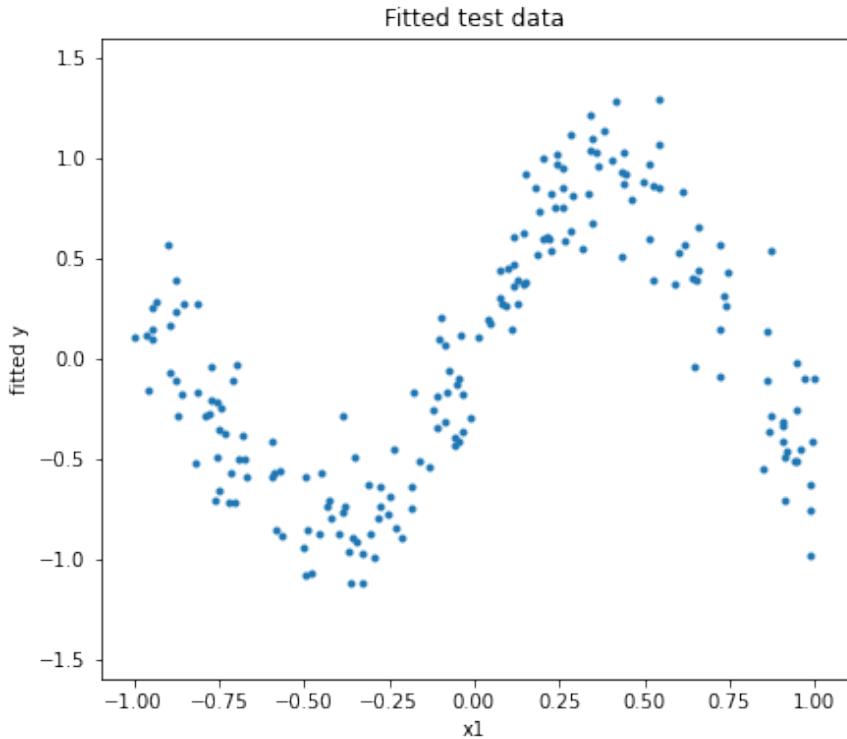
Fit a ReLU neural network to Sine function



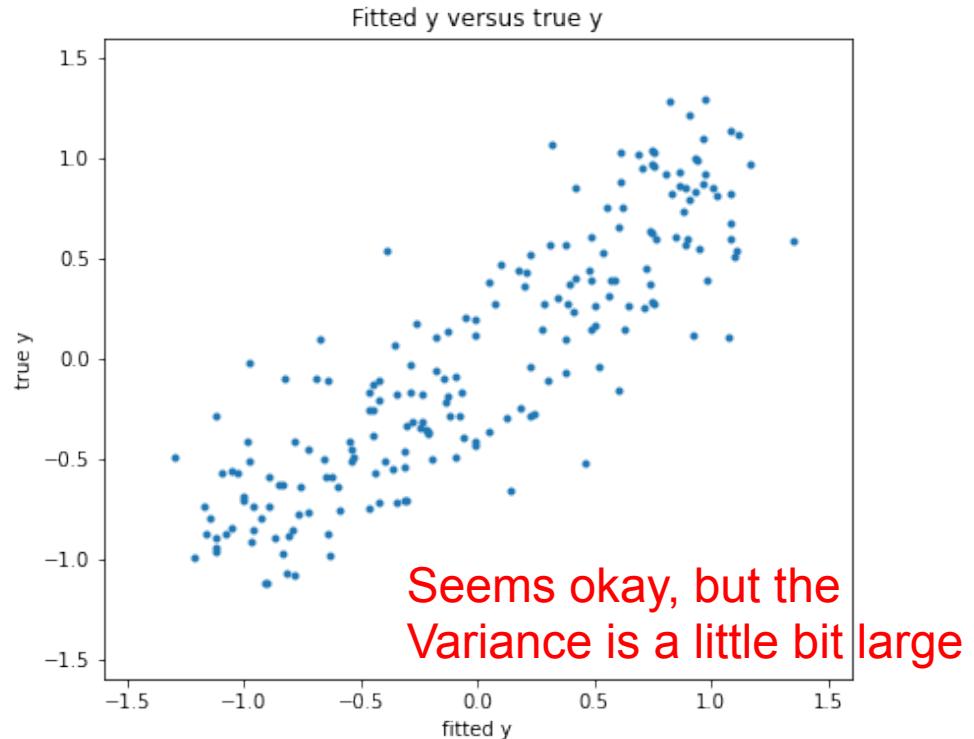
Trained model on **TRAINING** data



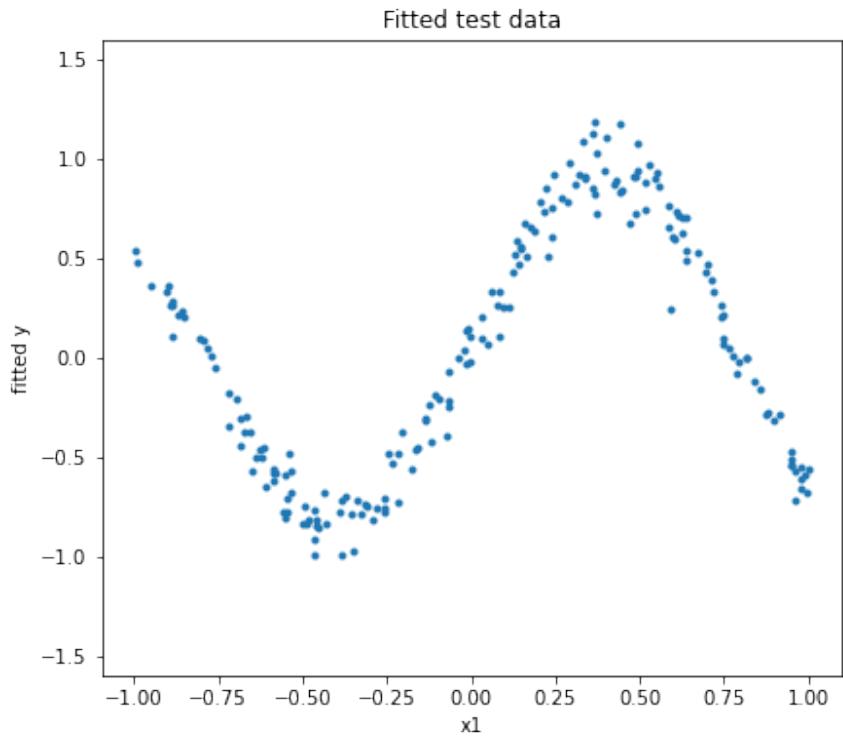
Fit a ReLU neural network to Sine function



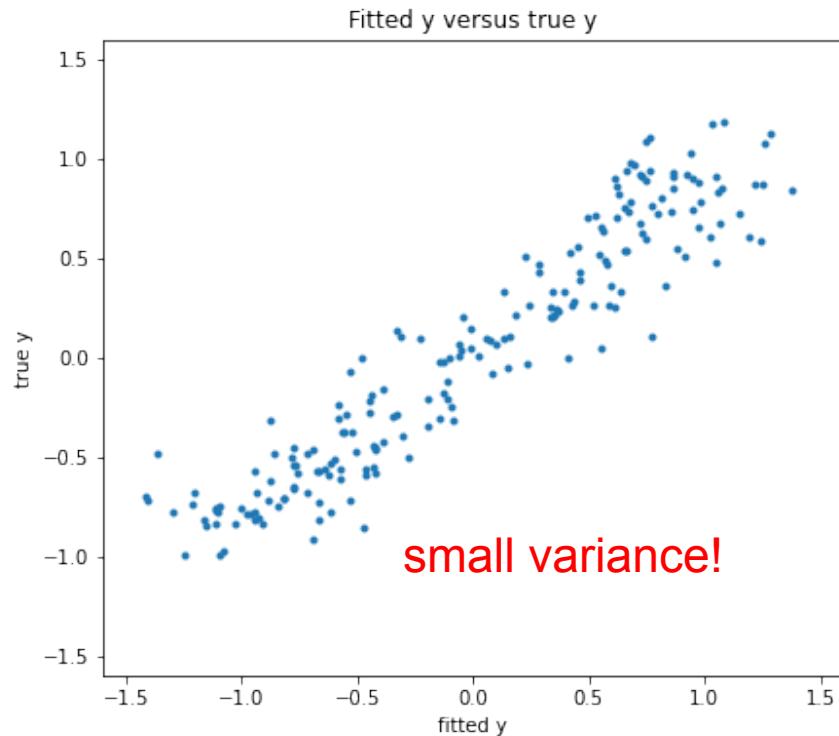
Trained model on TESTING data



Fit a ReLU neural network to Sine function



Trained model on **TESTING** data and with **REGULARIZATION**



Code: `optimizer = optim.Adam(train_model.parameters(), lr = 0.001, weight_decay = 0.01)`



Regularization of neural net

- To achieve better generalization

- Explicit regularization:

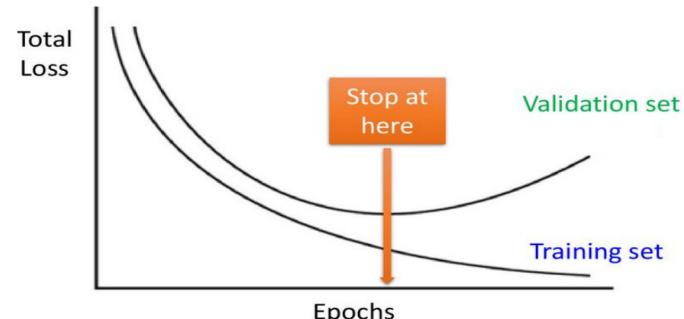
$$\min_{\mathbf{W}} \frac{1}{m} \sum_{i=1}^m \ell(\mathbf{y}_i, \text{DNN}_{\mathbf{W}}(\mathbf{x}_i)) + \lambda \Omega(\mathbf{W})$$

- Norms of weight matrices
- Norms of gradients / Jacobians
- ...

- Miscellaneous: A neural net structure can also induce regularization. E.g. The convolutional neural net (CNN)

- Implicit regularization:

- The regularization that is not built in the objective function (the loss)
- Regularization induced by **an optimization algorithm**: SGD tends to find a solution with small norm (regularized solution)
- Early stopping; batch normalization; dropout



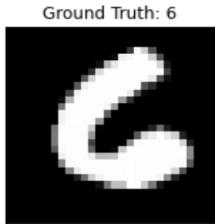
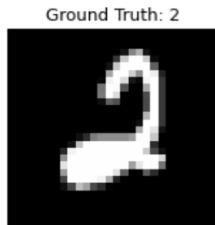
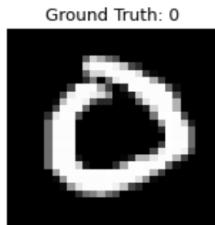
Credit: CSCI 5980/8980: Think Deep Learning offered by Ju Sun, UMN



Image classification using CNN

Dataset: MNIST

- One of the benchmark datasets of deep learning for image classification
- Classifying digits as 0, 1, ..., 9



Our CNN model

```
model = nn.Sequential(  
    nn.Conv2d(1, 32, 3, 1),  
    nn.ReLU(),  
    nn.Conv2d(32, 64, 3, 1),  
    nn.ReLU(),  
    nn.MaxPool2d(2),  
    nn.Dropout(0.25),  
    nn.Flatten(),  
    nn.Linear(9216, 128),  
    nn.ReLU(),  
    nn.Dropout(0.5),  
    nn.Linear(128, 10),  
    nn.LogSoftmax(dim = 1)  
)
```

Image classification using CNN

Our CNN structure with activation functions, dropout regularization in between layers.

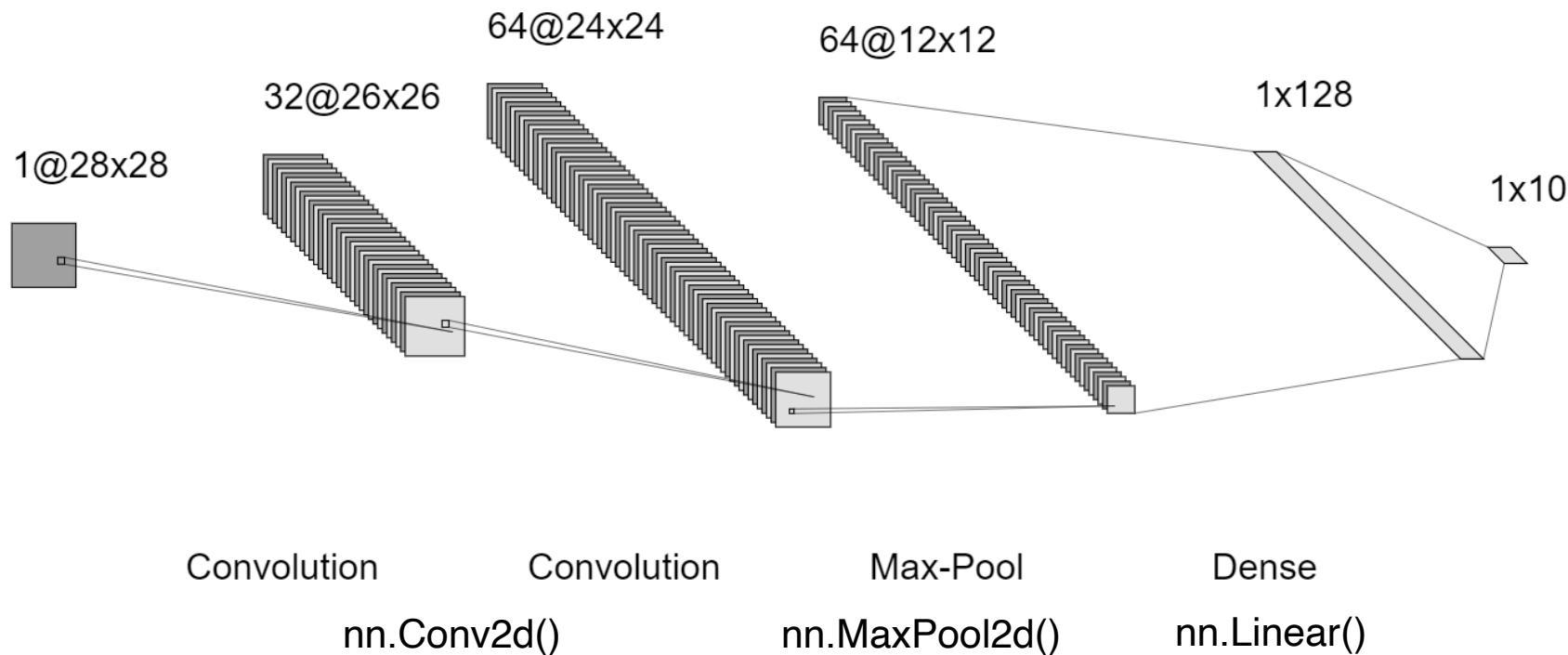
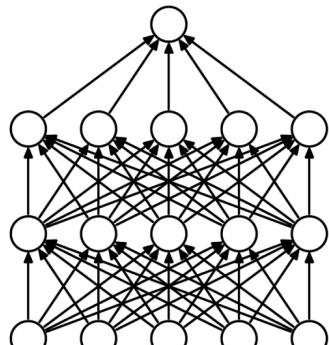


Image classification using CNN

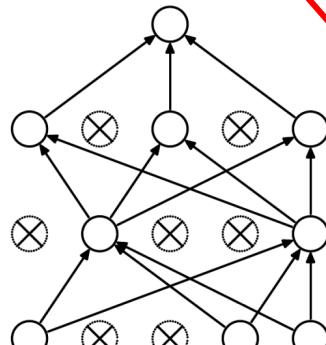
Our CNN model

Dropout for regularization:

- randomly kills inner neurons with some probability p



(a) Standard Neural Net



(b) After applying dropout.

```
model = nn.Sequential(  
    nn.Conv2d(1, 32, 3, 1),  
    nn.ReLU(),  
    nn.Conv2d(32, 64, 3, 1),  
    nn.ReLU(),  
    nn.MaxPool2d(2),  
    nn.Dropout(0.25),  
    nn.Flatten(),  
    nn.Linear(9216, 128),  
    nn.ReLU(),  
    nn.Dropout(0.5),  
    nn.Linear(128, 10),  
    nn.LogSoftmax(dim = 1)  
)
```

Credit: Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1), 1929-1958.

[live demo](#), [Implement from scratch](#)

Image classification using CNN

The total number of trainable parameters is 1,199,882!

Layer (type)	Input Shape	Param #	Tr. Param #
<hr/>			
Conv2d-1	[1, 1, 28, 28]	320	320
ReLU-2	[1, 32, 26, 26]	0	0
Conv2d-3	[1, 32, 26, 26]	18,496	18,496
ReLU-4	[1, 64, 24, 24]	0	0
MaxPool2d-5	[1, 64, 24, 24]	0	0
Dropout-6	[1, 64, 12, 12]	0	0
Flatten-7	[1, 64, 12, 12]	0	0
Linear-8	[1, 9216]	1,179,776	1,179,776
ReLU-9	[1, 128]	0	0
Dropout-10	[1, 128]	0	0
Linear-11	[1, 128]	1,290	1,290
LogSoftmax-12	[1, 10]	0	0
<hr/>			
Total params: 1,199,882			
Trainable params: 1,199,882			
Non-trainable params: 0			
<hr/>			



Check GPU availability

CPU

```
 # check device availability  
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  
print(device)  
  
cpu
```

GPU

```
 # check device availability  
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  
print(device)  
  
 cuda
```



Download and preprocess the MNIST training / testing dataset

Other transforms:
cropping, translation,
rotation, padding...



```
# define preprocessing transforms for images
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)))
])
```

```
# import MNIST dataset
train_data = datasets.MNIST('../data', train = True, download = True, transform = transform)
test_data = datasets.MNIST('../data', train = False, transform = transform)

# prepare dataset loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size = 256, shuffle = True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size = 256)
```



Download and preprocess the MNIST training / testing dataset

Most popular benchmark datasets can be loaded via [torchvision library](#)



```
# define preprocessing transforms for images
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)))
])
```

Training size: 60,000; Test size: 10,000

```
# import MNIST dataset
train_data = datasets.MNIST('../data', train = True, download = True, transform = transform)
test_data = datasets.MNIST('../data', train = False, transform = transform)
```

```
# prepare dataset loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size = 256, shuffle = True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size = 256)
```



and don't forget to prepare DataLoader for training



Define a training utils function

```
# a utils function for training
def train(model, device, train_loader, optimizer, epoch):
    model.train() # enable dropout
    correct = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        # transfer batches of data to specified device
        data, target = data.to(device), target.to(device)

        # build computation graph
        output = model(data)
        loss = F.nll_loss(output, target)

        # the optimization part
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # count the correctly classified cases
        pred = output.argmax(dim = 1, keepdim = True) # get the index of the max log-probability
        correct += pred.eq(target.view_as(pred)).sum().item()

    print(f"Epoch {epoch + 1}:")
    print(f"Training accuracy: {100. * correct / len(train_loader.dataset): .2f} %")
```

Similar as before,
nothing new



Train our CNN



```
# the training
model = myCNN() # initialize our model
model = model.to(device)
optimizer = optim.Adam(model.parameters(), lr = 0.005)

nepochs = 10
for epoch in range(nepochs):
    train(model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
```

```
Epoch 1: Training accuracy: 93.75 %; Testing accuracy: 9847/10000 (98%)
Epoch 2: Training accuracy: 97.94 %; Testing accuracy: 9856/10000 (99%)
Epoch 3: Training accuracy: 98.33 %; Testing accuracy: 9891/10000 (99%)
Epoch 4: Training accuracy: 98.72 %; Testing accuracy: 9883/10000 (99%)
Epoch 5: Training accuracy: 98.74 %; Testing accuracy: 9890/10000 (99%)
Epoch 6: Training accuracy: 98.85 %; Testing accuracy: 9888/10000 (99%)
Epoch 7: Training accuracy: 98.88 %; Testing accuracy: 9879/10000 (99%)
Epoch 8: Training accuracy: 99.07 %; Testing accuracy: 9906/10000 (99%)
Epoch 9: Training accuracy: 99.14 %; Testing accuracy: 9909/10000 (99%)
Epoch 10: Training accuracy: 99.24 %; Testing accuracy: 9893/10000 (99%)
```



Save / load the trained model: state dictionary method

Good for evaluation
and inference

- Save the model: only save weights/biases of the model.



```
# save parameter values
torch.save(model.state_dict(), "model_dict.pt")
```

- Load the model: initialize model, then load state dictionary.

```
# When one wants to use saved model next time:
# 1. initialize the same CNN model
model_new = myCNN()
# 2. load parameter values
model_new.load_state_dict(torch.load("model_dict.pt"))
model_new.eval()
```

Save / load the trained model: checkpoint method

Good for
resuming training

- Save the model: save BOTH weights/biases of the model and configurations of the optimizer.

```
# save checkpoint for resuming training
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict()
}, "model_checkpoint.pt")
```

Save / load the trained model: checkpoint method

Good for
resuming training

- Load the model: initialize BOTH model and **optimizer**, then load their state dictionaries.

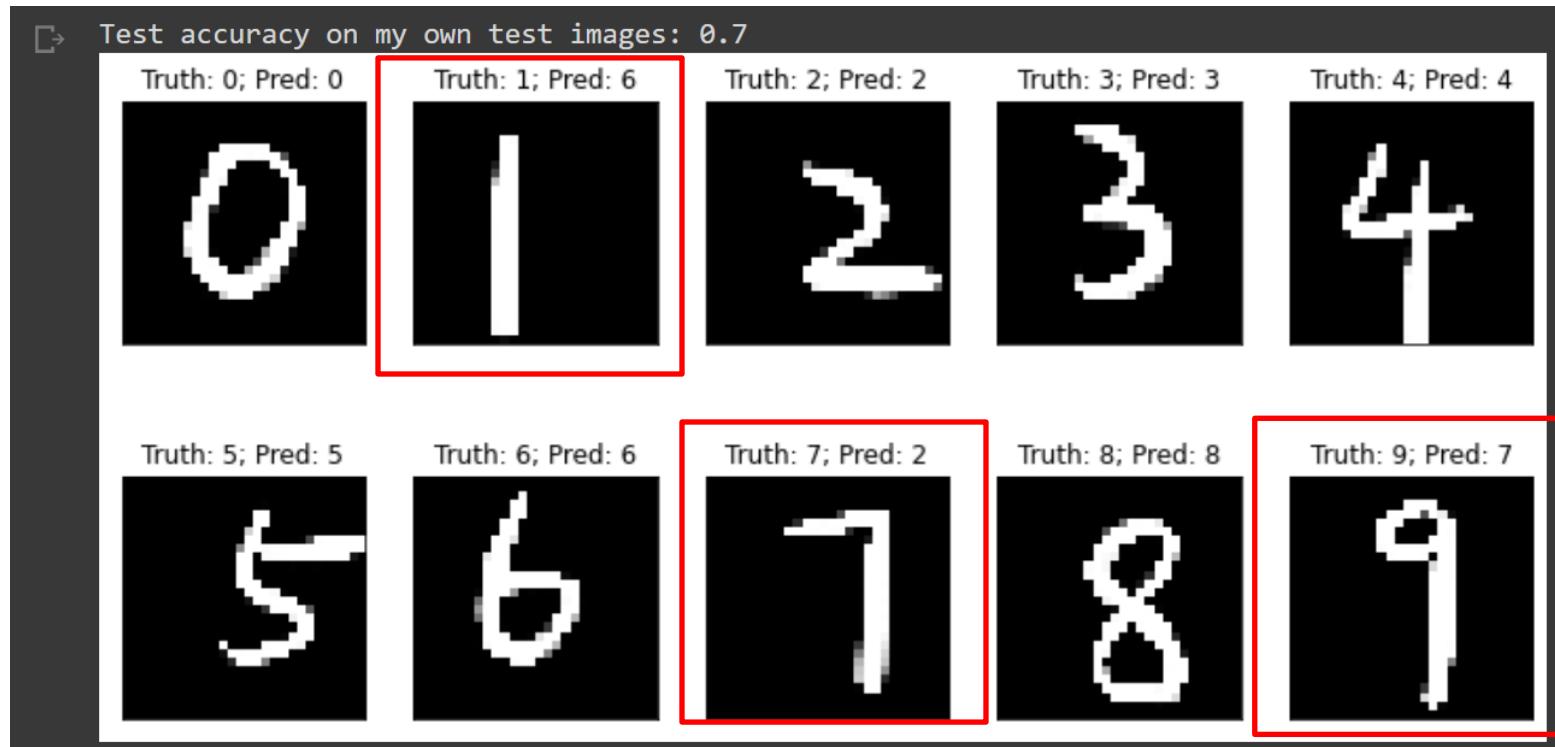
```
# When one wants to use saved model next time:  
# 1. initialize class and optimizer  
model_new = myCNN()  
optimizer_new = optim.Adam(model_new.parameters(), lr = 0.005)  
# 2. load checkpoints for both model and optimizer  
checkpoint = torch.load("model_checkpoint.pt")  
model_new.load_state_dict(checkpoint['model_state_dict'])  
optimizer_new.load_state_dict(checkpoint['optimizer_state_dict'])  
epoch = checkpoint['epoch']  
# 3. do whatever you like :)  
# - either - evaluate the mode and do some inference  
model_new.eval()  
# - or - resume training from your last checkpoint  
model_new.train()
```

Check [saving and loading models](#) for more details



Robustness issue of CNN

Underperformed test accuracy.
Solution: data augmentation!



Use data augmentation to improve generalization

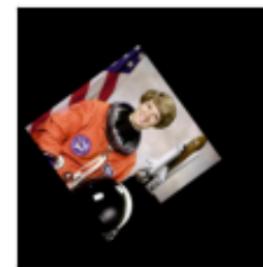
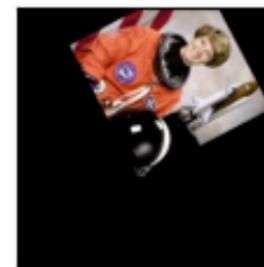
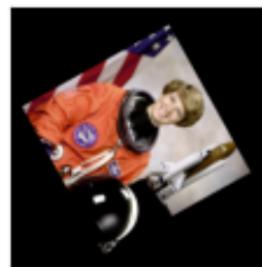
Key idea: Add some “noises” to augment your training data so that the trained model is more robust to new test data.

Original image



change colors

Original image



change spatial information

Implement data augmentation in PyTorch

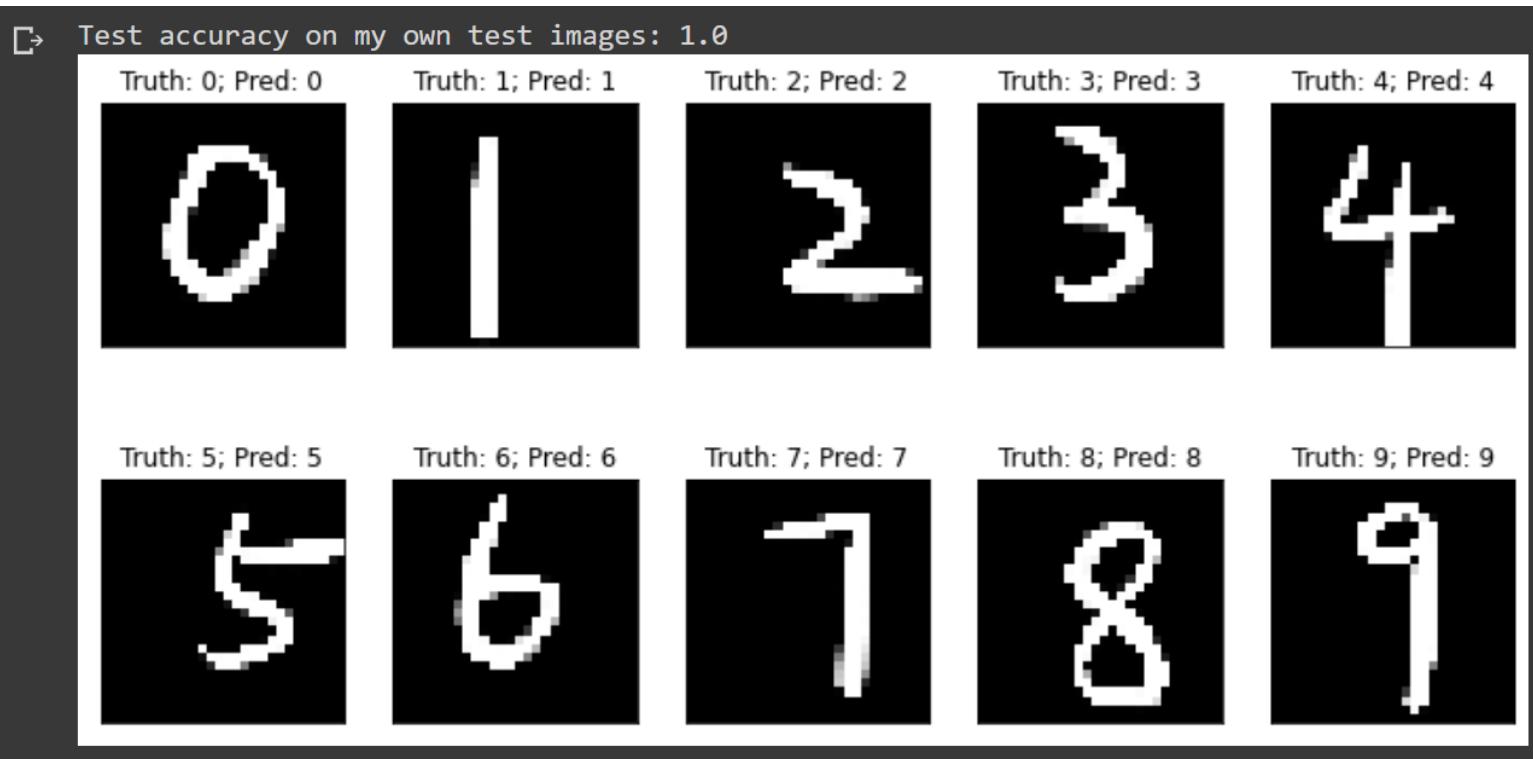
- Simply add more augmentation transforms when preparing your Dataset.

```
transform = transforms.Compose([
    transforms.ColorJitter(brightness = 0.05, contrast = 0.05),
    transforms.RandomAffine(degrees = 10, translate = (0.1,0.1), scale = (0.9, 1.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)))
])
```

- Check [Transforming and Augmenting Images](#) for more details.

CNN trained with data augmentation

Test accuracy 1.0;
Correctly classified 1, 7 and 9;
Better generalization!



Robustness issue of CNN

Solution: adversarial training!



$$+ .007 \times$$



=



x

“panda”

57.7% confidence

$$\text{sign}(\nabla_x J(\theta, x, y))$$

“nematode”

8.2% confidence

$$\epsilon \text{sign}(\nabla_x J(\theta, x, y)) +$$

“gibbon”

99.3 % confidence

Credit: Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.

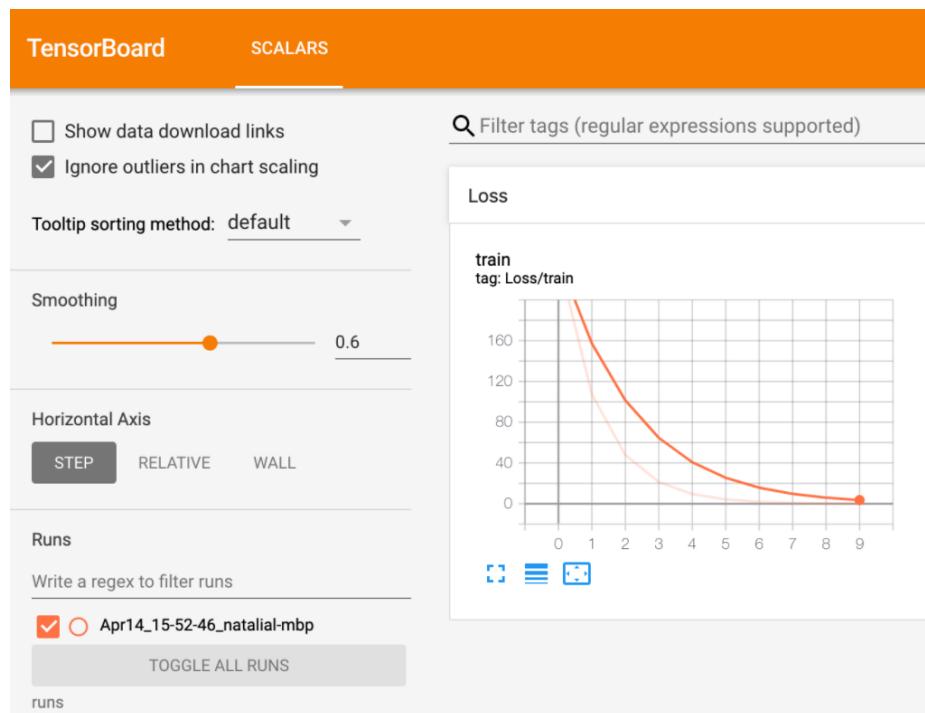


Outline

- Set up working environment
- Introduction to PyTorch framework
- Deep learning with PyTorch
- Other resources

Visualization with TensorBoard

- Monitor your training:
 - Metrics such as loss and accuracy
 - The model graph
 - Histograms, images and much more
- Some resources:
 - [How to use TensorBoard with PyTorch](#)
 - [Visualizing models, data, and training with TensorBoard](#)
 - [PyTorch TensorBoard support](#)



Use pretrained models from HuggingFace



The AI community building the future.

Build, train and deploy state of the art models powered by the reference open source in machine learning.

Star 84,670

- HuggingFace: <https://huggingface.co/>
- Check their [model hub](#).
- Play with their [transformers](#) library.

- Thousands of pre-trained models:
 - **Text**: text classification, question answering, summarization, translation, text generation;
 - **Images**: image classification, object detection, segmentation, generation;
 - **Audio**: speech recognition, audio classification
- Quick APIs for download and fine-tuning.
- Backed by popular framework like PyTorch.



Transformers

build failing license Apache-2.0 website online release v4.26.1 Contributor Covenant V2.0 adopted DOI 10.5281/zenodo.7391177

English | 简体中文 | 繁體中文 | 한국어 | Español | 日本語 | हिन्दी



Tutorials and courses:

Books and tutorials:

- [Dive into Deep Learning](#) (livebook)
- [Deep Learning](#) by Ian Goodfellow
- [Official PyTorch tutorial](#)
- [Deep learning with Python](#) (livebook)
- [UvA DL Notebooks](#)

Courses:

- [DL/ML tutorial](#) by Hung-Yi Lee
- [Deep learning](#) course by Yann LeCun
- [Deep learning with Pytorch](#)
- [Stanford STAT385 series](#)
- [Think Deep Learning](#) by Ju Sun





UNIVERSITY OF MINNESOTA

Driven to Discover®

Crookston Duluth Morris Rochester Twin Cities

The University of Minnesota is an equal opportunity educator and employer.