ETH zürich

CAMLab

Tutorial 1: Function approximation with Pytorch

AI in Science and Engineering 2026HS

Shizheng Wen | Sep 21, 2026

Seminar for Applied Mathematics (D-MATH) & ETH AI Center

ETH AI CENTER

SAM Seminar for Applied Mathematics

Outline

01 Logistics

02 Pytorch Crash Course

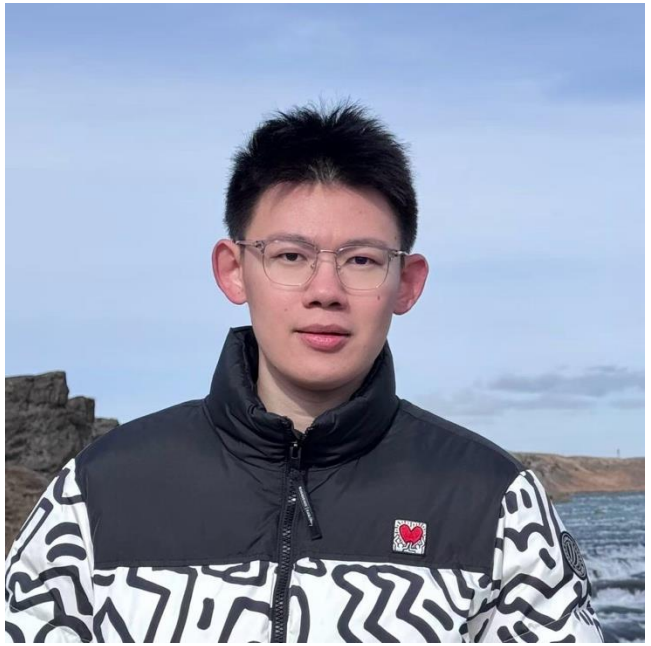
03 Practice

04 Preview

Outline

01 Logistics

TA Team



Shizheng Wen

unstructured neural operator

Differentiable programming

Transfer learning

Bogdan Raonic

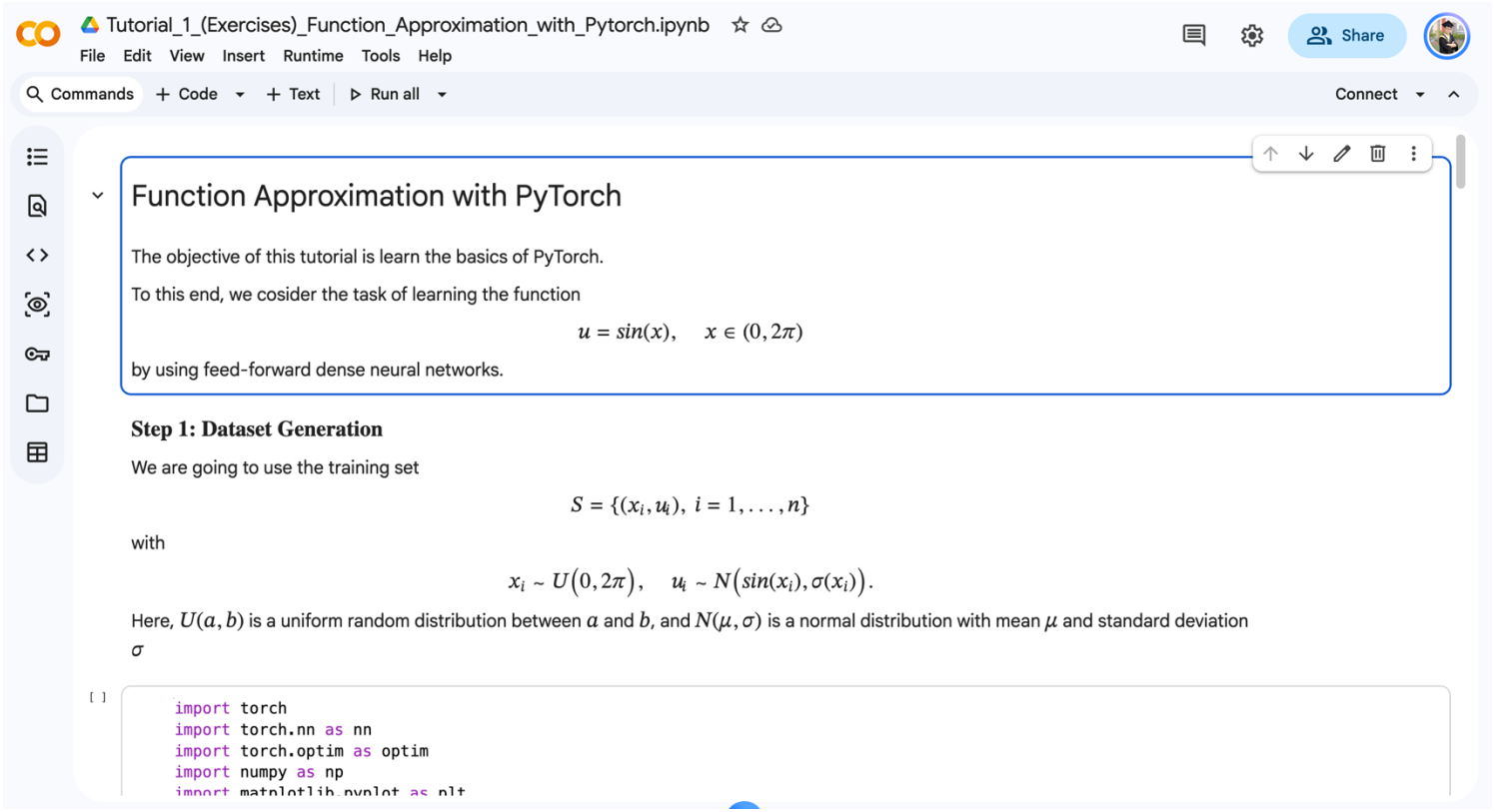
structured neural operator

Physics informed neural network

Diffusion model

Teaching format

- Going over the basic concept of the necessary knowledge
- Practice of Jupiter notebook on Colab.
- Questions (no questions about grading project!)



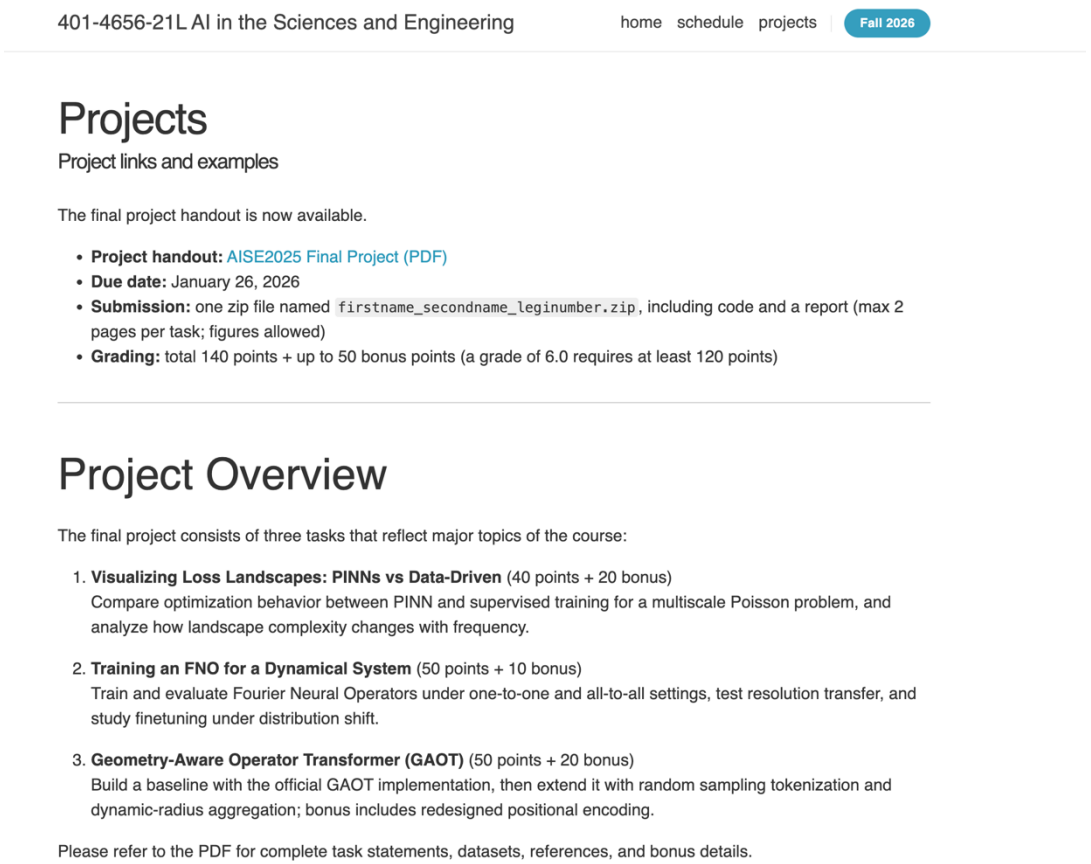
Syllabus

- You are flexible to join and choose you unfamiliar part.

• Function Approximation with Pytorch	- Shizheng
• Cross Validation and Intro to CNNs	- Shizheng
• PINN Training	- Bogdan
• Fourier Neural Operator	- Bogdan
• Convolutional Neural Operator	- Bogdan
• Time-Dependent CNO	- Bogdan
• Vison Transformer for solving PDEs	- Bogdan
• Graph Neural Network for soving PDEs	- Shizheng
• Geometry Aware Operator Transformer	- Shizheng
• Score-Based Diffusion Models	- Bogdan
• Differentiable Programming	- Shizheng
• Transfer Learning	- Shizheng

Final Project

- The course is project-based grading.
- The project will be release into mid of December, and the DDL is at the end of January.



401-4656-21L AI in the Sciences and Engineering home schedule projects Fall 2026

Projects

Project links and examples

The final project handout is now available.

- **Project handout:** AISE2025 Final Project (PDF)
- **Due date:** January 26, 2026
- **Submission:** one zip file named `firstname_secondname_leginumber.zip`, including code and a report (max 2 pages per task; figures allowed)
- **Grading:** total 140 points + up to 50 bonus points (a grade of 6.0 requires at least 120 points)

Project Overview

The final project consists of three tasks that reflect major topics of the course:

1. **Visualizing Loss Landscapes: PINNs vs Data-Driven** (40 points + 20 bonus)
   Compare optimization behavior between PINN and supervised training for a multiscale Poisson problem, and analyze how landscape complexity changes with frequency.
2. **Training an FNO for a Dynamical System** (50 points + 10 bonus)
   Train and evaluate Fourier Neural Operators under one-to-one and all-to-all settings, test resolution transfer, and study finetuning under distribution shift.
3. **Geometry-Aware Operator Transformer (GAOT)** (50 points + 20 bonus)
   Build a baseline with the official GAOT implementation, then extend it with random sampling tokenization and dynamic-radius aggregation; bonus includes redesigned positional encoding.

Please refer to the PDF for complete task statements, datasets, references, and bonus details.

Outline

02 Pytorch crash course

Five pieces of PyTorch for this course

- Twenty minutes of slides, then you write the code yourself.

01

Tensor

An array that can live on the CPU or the GPU

02

Autograd

Exact derivatives, computed for you

03

nn.Module

A model together with its parameters

04

Optimizer

The rule that updates the parameters

05

Training loop

Five lines that tie everything together

Tensors: shape, type and device

```
import torch

# 100 points in [0, 1], shape (100,)
x = torch.linspace(0, 1, 100)

# add a feature axis, shape (100, 1)
x = x.unsqueeze(-1)
print(x.shape, x.dtype, x.device)

# move to the GPU if there is one
use_gpu = torch.cuda.is_available()
device = "cuda" if use_gpu else "cpu"
x = x.to(device)
```

Shape

Layers expect (batch, features). One input number per sample means (N, 1), not (N,).

Type

The default is float32. NumPy gives float64, so convert when you load data.

Device

The model and the data must be on the same device.

Autograd: exact derivatives for free

```
# ask PyTorch to track x
x = torch.tensor(2.0, requires_grad=True)

# every operation is recorded
y = x**3 + 2*x

# chain rule, from y back to x
y.backward()

print(x.grad)
# tensor(14.)
# check: 3 * 2**2 + 2 = 14
```

Computation graph

PyTorch records each operation while your code runs.

backward()

Walks the graph in reverse and applies the chain rule.

.grad

The result is stored on the tensor. It adds up if you call backward() again.

Input derivatives: a preview of PINNs

```
x = torch.linspace(0, 1, 50).unsqueeze(-1)
x.requires_grad_(True)

# later: u = model(x)
u = torch.sin(3 * x)

du_dx = torch.autograd.grad(
    u, x,
    grad_outputs=torch.ones_like(u),
    create_graph=True,
)[0]

# du_dx equals 3 * cos(3 * x)
```

Derivative in x

Not with respect to the weights, but with respect to the input of the network.

create_graph=True

Keeps the result differentiable, so you can take second derivatives and train on them.

Why it matters

This is how a PINN builds the PDE residual in the PINN tutorial.

nn.Module: a model and its parameters

```
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, width=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, width), nn.Tanh(),
            nn.Linear(width, width), nn.Tanh(),
            nn.Linear(width, 1),
        )

    def forward(self, x):
        return self.net(x)
```

__init__

Create the layers here. Their weights are registered as parameters.

forward

Says how input becomes output. Call model(x), never model.forward(x).

No backward to write

Autograd builds it from forward.

Loss and optimizer: the training setup

```
model = MLP().to(device)

# what to minimize
loss_fn = nn.MSELoss()

# how to minimize it
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=1e-3,
)
```

Loss

One number that says how wrong the model is. For regression, the mean squared error.

Optimizer

Adam is a safe first choice. It needs the list of parameters.

Learning rate

The step size. Usually the first thing to tune.

Training loop: five lines, every time

- The same five lines in every tutorial. Only the model and loss change.

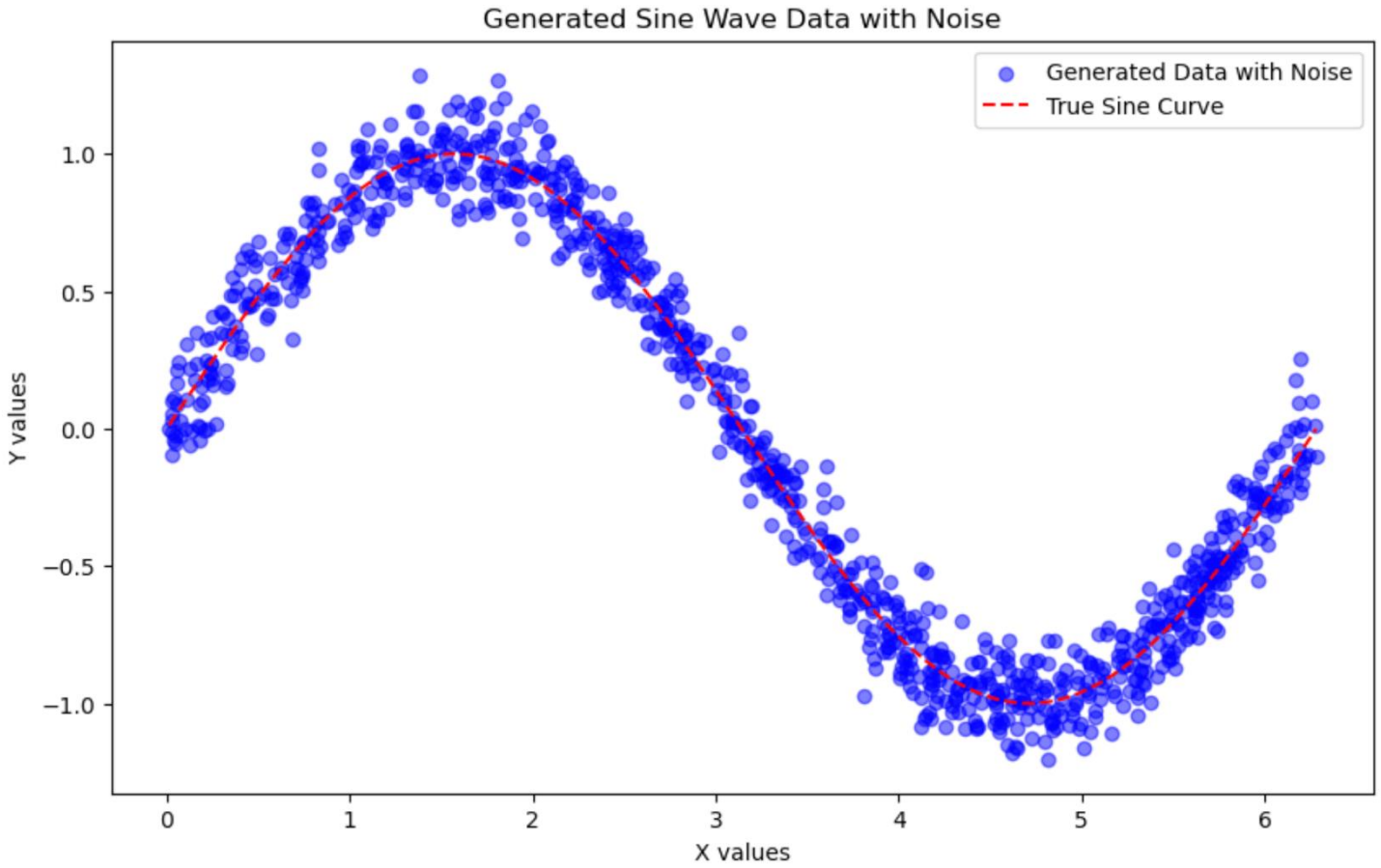
```
for epoch in range(2000):
    optimizer.zero_grad()       # 1. clear old gradients
    y_pred = model(x)           # 2. forward pass
    loss = loss_fn(y_pred, y)   # 3. compute the loss
    loss.backward()             # 4. backward pass
    optimizer.step()            # 5. update the weights

    if epoch % 200 == 0:
        print(epoch, loss.item())
```

Common mistakes: four bugs everyone meets

What you see	Cause	Fix
The loss jumps or does not go down	zero_grad() is missing, so gradients add up	Call it at the start of every step
The loss looks fine, but the fit is wrong	Shapes (N,) and (N, 1) broadcast to (N, N)	Print the shapes, use unsqueeze(-1)
Error: tensors on different devices	The model is on the GPU, the data on the CPU	Move both with .to(device)
Error: cannot call numpy() on a tensor that requires grad	You plot a tensor that is still in the graph	Use .detach().cpu() first

Jupiter on Colab – Fit a 1D function with an MLP

- Check link on the Moodle.



Next week

- Cross validation and CNN
- Feel free to leave if you have no further questions