Fourier Neural Operator

Tutorial 5

Operator learning: an FNO in PyTorch that maps initial conditions of the Allen–Cahn equation to solutions at a later time.

Bogdan Raonic · Oct 19, 2026 · Seminar for Applied Mathematics (D-MATH) & ETH AI Center

Outline

01 Operator learning

02 Fourier Neural Operator

03 Implementation in PyTorch

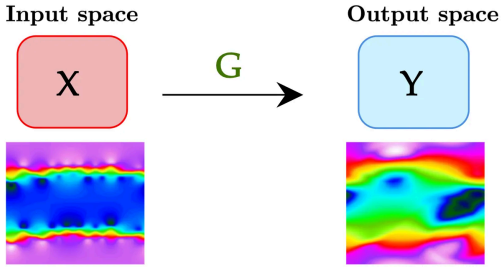
04 Results and common mistakes

05 Practice

01

Operator learning

Operators: functions in, functions out



PINNs: one solution

Tutorials 3 and 4 trained a network for one PDE instance. A new initial condition means training again.

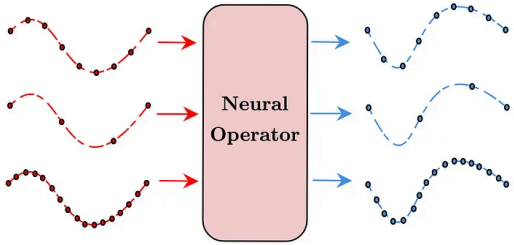
Operators: the whole map

The solution operator G takes an input function, e.g. the initial condition, to the solution.

Learned from data

We train on pairs (input, output) from a numerical solver. Then G is evaluated on new inputs in one forward pass.

Neural operators: any discretization



Functions as samples

On a computer a function is stored as its values on a grid, like the pixels of an image.

Grid independence

A neural operator should accept and return functions on different grids, coarse or fine.

Why it matters

Train on one resolution, evaluate on another. We test this at the end of the tutorial.

The Allen–Cahn equation

PDE

$$u_t = \Delta u - \varepsilon^2 u(u^2 - 1)$$

$x \in [-1, 1]$

OPERATOR

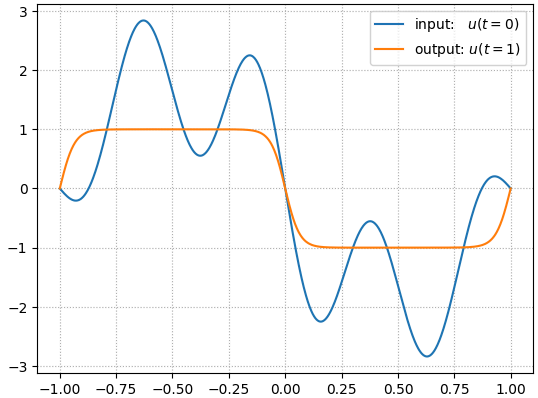
$$\mathcal{G} : u(\cdot, 0) \mapsto u(\cdot, 1)$$

initial condition to the solution at t = 1

DATA

128 train, 32 val, 256 test

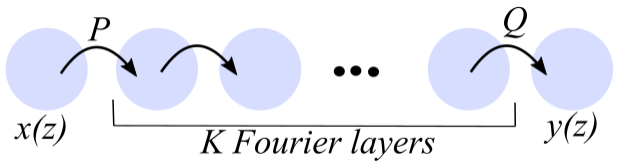
pairs from a solver, provided as .npy files



02

Fourier Neural Operator

FNO: lift, Fourier layers, project



P · Lifting

A pointwise linear layer raises the number of channels: $(u_0(x), x)$ becomes 64 channels.

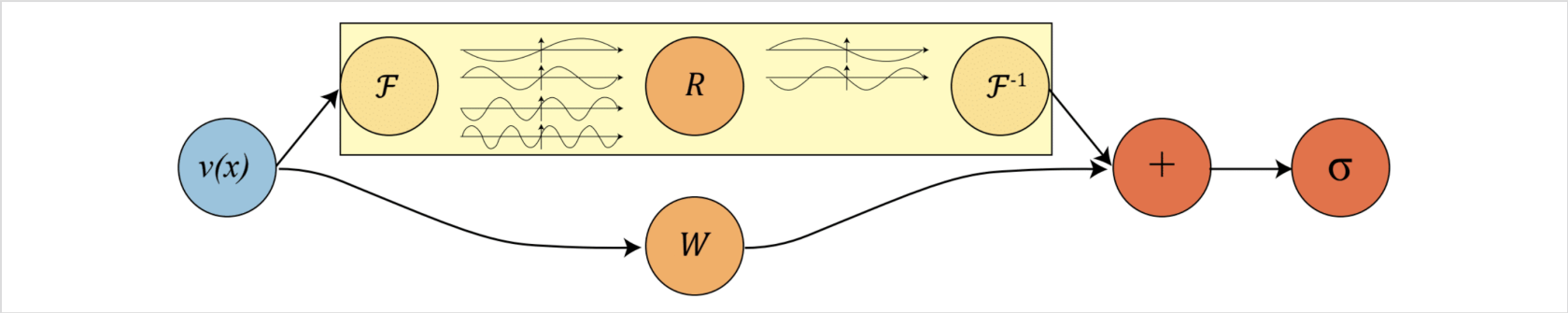
K Fourier layers

The same kind of layer applied K times. Here K = 3, with width 64.

Q · Projection

Pointwise layers bring the channels back down: 64 to 32 to 1, the output u(x, 1).

Fourier layer: a convolution in frequency space



01 · FFT

Transform each channel to Fourier coefficients with torch.fft.rfft.

02 · R on low modes

Keep the lowest 16 modes and multiply them by learned complex weights. Discard the rest.

03 · Inverse FFT, + W, σ

Back to physical space, add a pointwise linear skip W v(x), apply tanh. The operation is global.

03

Implementation

SpectralConv1d: FFT, multiply, inverse FFT

```
def forward(self, x):
    # x: (batch, channels, grid points)
    x_ft = torch.fft.rfft(x)

    out_ft = torch.zeros(batchsize, self.out_channels,
        x.size(-1) // 2 + 1, dtype=torch.cfloat)
    out_ft[:, :, :self.modes1] = self.compl_mul1d(
        x_ft[:, :, :self.modes1], self.weights1)

    return torch.fft.irfft(out_ft, n=x.size(-1))

# (b, in, k), (in, out, k) -> (b, out, k)
torch.einsum("bix,iox->box", input, weights)
```

rfft

Real input, so only the non-negative frequencies are kept: N/2 + 1 coefficients.

Truncation

Only the first modes1 = 16 coefficients get weights. Higher modes are set to zero.

einsum

Per mode, mixes the input channels into output channels with a complex matrix.

FNO1d: lifting, three Fourier layers, projection

```
self.linear_p = nn.Linear(2, self.width)   # (u0, x)
self.spect1 = SpectralConv1d(width, width, modes)
self.lin0 = nn.Conv1d(width, width, 1)      # W
self.linear_q = nn.Linear(self.width, 32)

def fourier_layer(self, x, spectral, conv):
    return self.activation(spectral(x) + conv(x))
def forward(self, x):
    x = self.linear_p(x).permute(0, 2, 1)
    x = self.fourier_layer(x, self.spect1, self.lin0)
    ...  # spect2, spect3 the same way
    x = self.activation(self.linear_q(x.permute(0, 2, 1)))
    return self.output_layer(x)
```

Grid as input

Two input channels: the initial condition $u_0(x)$ and the coordinate x.

Conv1d, kernel 1

The skip W is a pointwise linear map, the same at every grid point.

permute

Linear acts on the last axis, Conv1d and the FFT expect channels first.

Training: plain supervised regression

```
fno = FNO1d(modes=16, width=64)
optimizer = Adam(fno.parameters(), lr=0.001)
scheduler = StepLR(optimizer, step_size=50, gamma=0.5)
l = torch.nn.MSELoss()

for epoch in range(epochs):
    for input_batch, output_batch in training_set:
        optimizer.zero_grad()
        pred = fno(input_batch).squeeze(2)
        loss = l(pred, output_batch)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

No physics in the loss

Unlike a PINN, only data: MSE between predicted and solver outputs.

Mini-batches of 10

The same loop as in Tutorials 1 and 2, with Adam and a step learning-rate schedule.

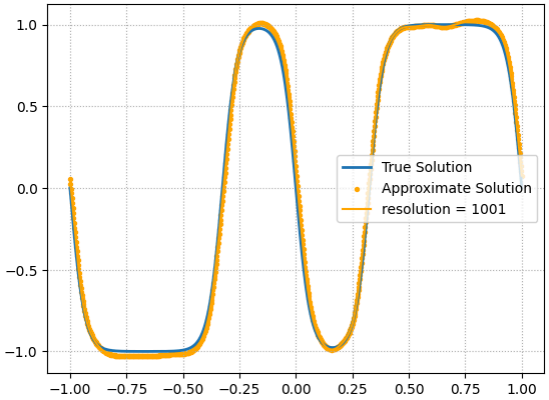
epochs = 10

Set low for class. The notebook suggests 250 for a fully trained model.

04

Results

Result: a test sample

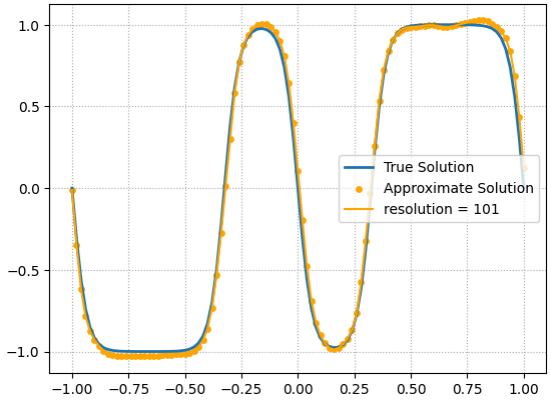


4.2%

relative L2 error on test sample 12, after 10 epochs

The input was never seen in training. One forward pass, no solver.

Resolution invariance: 10× coarser grid



4.9%

relative L2 error on the same sample, every 10th grid point

Same trained weights, 101 points instead of the training grid. The Fourier weights act on modes, not on grid points.

Common mistakes: FNO

WHAT YOU SEE	CAUSE	FIX
Shape error in the spectral layer	Channels last instead of first	permute to (batch, channels, grid)
Error: modes out of range	modes > N/2 + 1 on a coarse grid	Keep modes below N/2 + 1
Validation error looks too small	Sum over val batches divided by len(testing_set)	Divide by len(validation_set)
Weights stay real, error is high	dtype missing in the weight tensor	Use dtype=torch.cfloat

05

Practice

Practice: an FNO on Colab

- Check link on the Moodle.
- Train for 250 epochs and compare the test error.
- Change the number of modes and the width.
- Try other subsampling factors for resolution invariance.