Fourier Neural Operators: Theory, Implementation, and Practice

A Comprehensive Tutorial with Exercises

Tutorial Document

December 3, 2025

Abstract

This comprehensive tutorial explores Fourier Neural Operators (FNOs), a revolutionary architecture for learning solution operators of partial differential equations. We present the theoretical foundations, explain why FNOs are resolution-invariant, dissect the PyTorch implementation in detail, and provide structured exercises to build your own implementation from scratch. This document is designed for students and practitioners who want to deeply understand both the mathematical elegance and practical implementation of FNOs.

Contents

1 Introduction to Operator Learning

1.1 The Classical Problem

Traditional neural networks excel at learning mappings between finite-dimensional spaces:

$$f : \mathbb{R}^n \to \mathbb{R}^m \tag{1}$$

However, many problems in science and engineering involve learning operators that map between **infinite-dimensional function spaces**:

$$\mathcal{G} : \mathcal{U} \to \mathcal{V} \tag{2}$$

where $\mathcal{U}$ and $\mathcal{V}$ are function spaces.

1.2 PDE Solution Operators

Definition 1.1: PDE Solution Operator

Consider a parametric PDE:

$$\begin{cases} \mathcal{L}(u; a) = f & \text{in } \Omega \\ \mathcal{B}(u) = g & \text{on } \partial\Omega \end{cases} \tag{3}$$

where $\mathcal{L}$ is a differential operator, a represents parameters (e.g., coefficients, initial conditions), f is a source term, and $\mathcal{B}$ represents boundary conditions.
The **solution operator** is:

$$\mathcal{G} : a \mapsto u \tag{4}$$

which maps the parameters to the solution.

Remark 1.1: Why Operator Learning?

Traditional approaches solve PDEs numerically for each new set of parameters, which is computationally expensive. Operator learning aims to learn $\mathcal{G}$ directly from data, enabling:

- Fast predictions for new parameters
- Generalization across different discretizations
- Integration into larger computational pipelines

1.3 The Discretization Challenge

When we work with computers, functions must be discretized. A key challenge is:

Key Point

Traditional neural networks are tied to specific discretizations. If you train on a 64×64 grid, the model cannot handle a 128×128 grid without retraining.

This is where Fourier Neural Operators shine.

2 Fourier Neural Operators: Architecture

2.1 High-Level Overview

The Fourier Neural Operator (FNO) was introduced by Li et al. (2020) in their paper "Fourier Neural Operator for Parametric Partial Differential Equations."

The key innovation is performing convolutions in the **Fourier domain**, which enables:

1. **Global receptive fields** (unlike local convolutions)
2. **Resolution invariance** (works on any discretization)
3. **Efficient computation** (via FFT)

2.2 Architecture Components

An FNO consists of three main components:

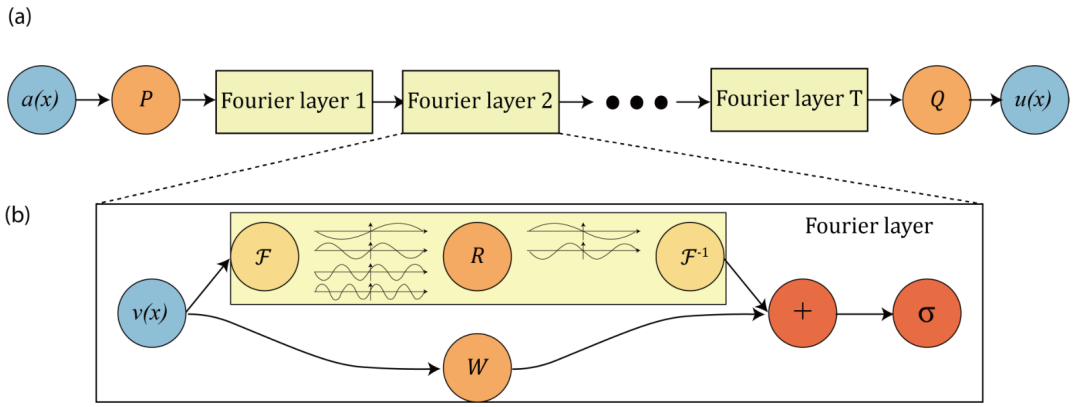

Figure 1: Fourier Neural Operator Architecture. The network consists of a lifting layer P, multiple Fourier layers, and a projection layer Q.

2.2.1 1. Lifting Layer (P)

$$v_0(x) = P(a(x), x) \tag{5}$$

The lifting layer maps the input function (and potentially spatial coordinates) to a higher-dimensional space:

$$P : \mathbb{R}^{d_a} \to \mathbb{R}^d \tag{6}$$

where d_a is the input channel dimension and d is the hidden dimension (width).

Purpose: Increases representation capacity, similar to feature extraction in CNNs.

2.2.2 2. Fourier Layers

The core of the FNO. Each layer computes:

$$v_{t+1}(x) = \sigma\left(W v_t(x) + (\mathcal{K} v_t)(x)\right) \tag{7}$$

where:

- W is a local linear transformation (implemented as 1D convolution)
- $\mathcal{K}$ is the **integral kernel operator** (implemented in Fourier space)
- σ is an activation function

2.2.3 3. Projection Layer (Q)

$$u(x) = Q(v_T(x)) \tag{8}$$

Maps from the high-dimensional hidden space back to the output:

$$Q : \mathbb{R}^d \to \mathbb{R}^{d_u} \tag{9}$$

2.3 The Fourier Layer in Detail

This is where the magic happens. Let's understand each step:

Algorithm 1 Fourier Layer Forward Pass

Input: $v_t \in \mathbb{R}^{B \times d \times n}$ where B is batch size, d is channels, n is spatial resolution
Step 1: Apply FFT to transform to frequency domain
$\hat{v}_t = \mathcal{F}(v_t) \in \mathbb{C}^{B \times d \times n/2+1}$
Step 2: Apply learnable linear transformation on lower frequency modes
$\hat{w}_t = R_\phi \hat{v}_t$ where R_ϕ acts only on modes $k \leq k_{\max}$
Step 3: Apply inverse FFT to return to physical space
$w_t = \beth(\hat{w}_t) \in \mathbb{R}^{B \times d \times n}$
Step 4: Add skip connection with activation
$v_{t+1} = \sigma(w_t + W v_t)$
Return: v_{t+1}

Definition 2.1: Integral Kernel Operator in Fourier Space

The kernel operator can be written as:

$$(\mathcal{K}v)(x) = \int_\Omega \kappa(x, y) v(y) dy \tag{10}$$

By the convolution theorem, this becomes a pointwise multiplication in Fourier space:

$$\widehat{(\mathcal{K}v)}(k) = \hat{\kappa}(k) \cdot \hat{v}(k) \tag{11}$$

where $\hat{\kappa}(k)$ are learnable complex weights parameterizing the kernel in frequency space.

3 Why FNO is Resolution Invariant

This is one of the most important properties of FNOs. Let's understand it deeply.

3.1 The Problem with Traditional CNNs

Traditional CNNs operate in the **spatial domain**:

- Weights are tied to specific grid locations
- Number of parameters depends on resolution
- Model cannot handle different resolutions without retraining

3.2 The FNO Solution

FNOs operate in the **frequency domain**:

Theorem 3.1: Resolution Invariance of FNO

Let $v \in L^2(\Omega)$ be a function and $v^{(n)} \in \mathbb{R}^n$ be its discretization on n grid points. The FNO operator $\mathcal{G}_\theta$ satisfies:

$$\lim_{n\to\infty} \left\| \mathcal{G}_\theta(v^{(n)}) - \mathcal{G}_\theta(v) \right\|_{L^2} = 0 \tag{12}$$

Furthermore, for two different discretizations $v^{(n_1)}$ and $v^{(n_2)}$:

$$\mathcal{G}_\theta(v^{(n_1)}) \approx \mathcal{G}_\theta(v^{(n_2)}) \tag{13}$$

up to discretization error, without any retraining.

Intuition. The key insight is that FNO learns in the frequency domain:

1. **Truncation in Frequency Space:** FNO only parameterizes the first $k_{\max}$ Fourier modes. These modes represent the large-scale structure of the function.
2. **Resolution Independence:** The Fourier modes are intrinsic to the function, not the discretization. Whether you discretize with $n = 64$ or $n = 128$ points, the first 16 Fourier modes represent the same mathematical content.
3. **Automatic Interpolation:** The inverse FFT automatically handles different resolutions. Given the same Fourier coefficients, it produces the correct values on any grid.
4. **Zero-Padding:** For higher resolutions, modes beyond $k_{\max}$ are implicitly zero, which is a natural assumption for smooth functions.

□

3.3 Mathematical Foundation

Let's formalize this. For a function $v : \Omega \to \mathbb{R}$ where $\Omega = [0, 1]$, the Fourier series expansion is:

$$v(x) = \sum_{k\in\mathbb{Z}} \hat{v}(k) e^{2\pi i k x} \tag{14}$$

The FNO parametrizes a truncated version:

$$v_{\text{FNO}}(x) = \sum_{|k|\le k_{\max}} \hat{v}(k) e^{2\pi i k x} \tag{15}$$

This truncation is:

- **Independent of discretization:** The same $k_{\max}$ modes work for any resolution
- **A natural regularization:** Assumes the solution is dominated by low-frequency components (which is true for most smooth PDEs)
- **Computationally efficient:** Fewer parameters than full resolution

Remark 3.1: Practical Implication

You can train on coarse grids and test on fine grids!
For example:

- Train on 64×64 grid (cheaper)
- Test on 256×256 grid (more accurate)
- No retraining needed!

The model will generalize because it learned the operator in the continuous function space, not tied to a specific discretization.

4 Implementation Details and PyTorch Tricks

Now let's dive deep into the implementation, understanding every design choice and PyTorch trick.

4.1 The SpectralConv1d Layer

This is the heart of the FNO. Let's examine it piece by piece:

```python
class SpectralConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, modes1):
        super(SpectralConv1d, self).__init__()

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.modes1 = modes1  # Number of Fourier modes to keep

        # Initialize weights with proper scaling
        self.scale = (1 / (in_channels * out_channels))
        self.weights1 = nn.Parameter(
            self.scale * torch.rand(
                in_channels, out_channels,
                self.modes1,
                dtype=torch.cfloat  # Complex numbers!
            )
        )
```

Listing 1: SpectralConv1d Implementation

4.1.1 PyTorch Trick #1: Complex Parameters

Key Point

`dtype=torch.cfloat` creates complex-valued parameters. This is crucial because Fourier coefficients are complex numbers: $\hat{v}(k) \in \mathbb{C}$.
PyTorch's autograd automatically handles complex gradients!

4.1.2 PyTorch Trick #2: Einstein Summation

```python
def compl_mul1d(self, input, weights):
    # (batch, in_channel, x), (in_channel, out_channel, x)
```

```
    # -> (batch, out_channel, x)
    return torch.einsum("bix,iox->box", input, weights)
```

Listing 2: Complex Multiplication via einsum

Remark 4.1: Why einsum?

`torch.einsum` performs Einstein summation notation:

- **Elegant:** Describes the operation mathematically
- **Flexible:** Works with arbitrary tensor dimensions
- **Efficient:** Optimized by PyTorch backend

The notation `"bix,iox->box"` means:

$$\text{output}[b, o, x] = \sum_i \text{input}[b, i, x] \cdot \text{weights}[i, o, x] \tag{16}$$

This is a batched matrix multiplication over the channel dimension, for each frequency x.

4.1.3 The Forward Pass

```
def forward(self, x):
    batchsize = x.shape[0]
    # x.shape = [batch, channels, spatial_dim]

    # Step 1: FFT to frequency domain
    x_ft = torch.fft.rfft(x)
    # x_ft.shape = [batch, channels, spatial_dim//2 + 1]

    # Step 2: Multiply relevant Fourier modes
    out_ft = torch.zeros(
        batchsize, self.out_channels,
        x.size(-1) // 2 + 1,
        device=x.device, dtype=torch.cfloat
    )
    out_ft[:, :, :self.modes1] = self.compl_mul1d(
        x_ft[:, :, :self.modes1],
        self.weights1
    )

    # Step 3: Inverse FFT back to physical space
    x = torch.fft.irfft(out_ft, n=x.size(-1))

    return x
```

Listing 3: Forward Pass of SpectralConv1d

4.1.4 PyTorch Trick #3: Real FFT (rfft)

Key Point

`torch.fft.rfft` is used instead of `torch.fft.fft` because:

- Input is real-valued
- Due to symmetry: $\hat{v}(-k) = \overline{\hat{v}(k)}$
- Only need to store positive frequencies
- Output size: $n/2 + 1$ instead of n (saves memory)
- Inverse: `torch.fft.irfft` reconstructs real signal

4.1.5 PyTorch Trick #4: Mode Truncation

```
out_ft[:, :, :self.modes1] = self.compl_mul1d(...)
```

This implements the low-pass filter:

- Only transform the first `modes1` frequencies
- Higher frequencies remain zero (implicit in `out_ft` initialization)
- This is the resolution invariance mechanism!

4.2 The Complete FNO1d Model

```
class FNO1d(nn.Module):
    def __init__(self, modes, width):
        super(FNO1d, self).__init__()

        self.modes1 = modes
        self.width = width

        # Lifting: R^2 -> R^width
        self.linear_p = nn.Linear(2, self.width)

        # Three Fourier layers
        self.spect1 = SpectralConv1d(self.width, self.width, self.
    modes1)
        self.spect2 = SpectralConv1d(self.width, self.width, self.
    modes1)
        self.spect3 = SpectralConv1d(self.width, self.width, self.
    modes1)

        # Skip connections (1x1 convolutions)
        self.lin0 = nn.Conv1d(self.width, self.width, 1)
        self.lin1 = nn.Conv1d(self.width, self.width, 1)
        self.lin2 = nn.Conv1d(self.width, self.width, 1)

        # Projection: R^width -> R^32 -> R^1
        self.linear_q = nn.Linear(self.width, 32)
        self.output_layer = nn.Linear(32, 1)

        self.activation = torch.nn.Tanh()
```

```
    def fourier_layer(self, x, spectral_layer, conv_layer):
        return self.activation(spectral_layer(x) + conv_layer(x))

    def forward(self, x):
        # x.shape = [batch, spatial, 2]  (includes grid points!)

        # Lift
        x = self.linear_p(x)  # [batch, spatial, width]
        x = x.permute(0, 2, 1)  # [batch, width, spatial]

        # Fourier layers
        x = self.fourier_layer(x, self.spect1, self.lin0)
        x = self.fourier_layer(x, self.spect2, self.lin1)
        x = self.fourier_layer(x, self.spect3, self.lin2)

        # Project
        x = x.permute(0, 2, 1)  # [batch, spatial, width]
        x = self.activation(self.linear_q(x))  # [batch, spatial, 32]
        x = self.output_layer(x)  # [batch, spatial, 1]

        return x
```

Listing 4: Full FNO1d Implementation

4.2.1 PyTorch Trick #5: Including Grid Coordinates

Key Point

The input has dimension 2: `[u(x), x]`
Why include x?

- Provides positional information
- Helps with non-periodic boundary conditions
- Allows the model to learn position-dependent transformations

This is different from standard CNNs where position is implicit in the convolution kernel locations.

4.2.2 PyTorch Trick #6: The permute Dance

Notice the `permute` operations:

```
x = x.permute(0, 2, 1)  # Before Fourier layers
...
x = x.permute(0, 2, 1)  # After Fourier layers
```

Why?

- `nn.Linear` expects: `[batch, spatial, features]`
- `SpectralConv1d` expects: `[batch, features, spatial]`
- `torch.fft.rfft` operates on the last dimension

This is a common pattern in neural operator implementations.

4.2.3 PyTorch Trick #7: Skip Connections via 1D Convolution

```
self.lin0 = nn.Conv1d(self.width, self.width, 1)
```

A 1D convolution with kernel size 1 is equivalent to:

- Pointwise linear transformation
- Applies same weights at every spatial location
- More efficient than `nn.Linear` for spatial data
- Natural fit for the skip connection $Wv_t(x)$

4.3 Training Details

```
optimizer = Adam(fno.parameters(), lr=0.001, weight_decay=1e-5)
scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer, step_size=50, gamma=0.5
)

loss_fn = nn.MSELoss()

for epoch in range(epochs):
    for input_batch, output_batch in training_loader:
        optimizer.zero_grad()

        # Forward pass
        pred = fno(input_batch).squeeze(2)

        # Compute loss
        loss = loss_fn(pred, output_batch)

        # Backward pass
        loss.backward()
        optimizer.step()

    scheduler.step()
```

Listing 5: Training Loop

4.3.1 PyTorch Trick #8: Squeeze Operation

```
pred = fno(input_batch).squeeze(2)
```

The model outputs shape `[batch, spatial, 1]`, but the target has shape `[batch, spatial]`. The `squeeze(2)` removes the singleton dimension.

4.3.2 PyTorch Trick #9: Weight Decay as Regularization

```
Adam(fno.parameters(), lr=0.001, weight_decay=1e-5)
```

Weight decay adds L2 regularization:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{MSE}} + \lambda \sum_i \theta_i^2 \tag{17}$$

This prevents overfitting, especially important for operator learning where training data is limited.

5 Example: Allen-Cahn Equation

The notebook demonstrates FNO on the Allen-Cahn equation:

$$u_t = \Delta u - \epsilon^{-2} u(u^2 - 1), \quad (x, t) \in \mathbb{R} \times \mathbb{R}_{>0} \tag{18}$$

The operator to learn:

$$\mathcal{G} : u(\cdot, t = 0) \mapsto u(\cdot, t = 1) \tag{19}$$

5.1 Data Processing

```
x_data = torch.from_numpy(np.load("AC_data_input.npy")).type(torch.
    float32)
y_data = torch.from_numpy(np.load("AC_data_output.npy")).type(torch.
    float32)

# Swap columns to get [u(x), x] format
temporary_tensor = torch.clone(x_data[:, :, 0])
x_data[:, :, 0] = x_data[:, :, 1]
x_data[:, :, 1] = temporary_tensor
```

Listing 6: Data Loading

Data Format:

- `x_data`: shape `[n_samples, n_points, 2]` containing $[u_0(x), x]$
- `y_data`: shape `[n_samples, n_points]` containing $u_1(x)$

5.2 Resolution Invariance Test

The notebook demonstrates resolution invariance by subsampling:

```
subsample = 10
input_test_n = input_function_test[idx, ::subsample]
output_test_n = output_function_test[idx, ::subsample]

# Model still works!
output_pred = fno(input_test_n)
```

Listing 7: Testing on Coarser Grid

Key Point

The same trained model works on:

- Full resolution ($n = 1024$ points)
- Subsampled ($n = 102$ points, every 10th point)
- Any other resolution!

This is impossible with standard CNNs.

6 Extension to 2D: SpectralConv2d

For 2D problems, the extension is natural:

```python
class SpectralConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, modes1, modes2):
        super(SpectralConv2d, self).__init__()

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.modes1 = modes1  # Modes in first dimension
        self.modes2 = modes2  # Modes in second dimension

        self.scale = (1 / (in_channels * out_channels))

        # Need two weight matrices for 2D!
        self.weights1 = nn.Parameter(
            self.scale * torch.rand(
                in_channels, out_channels,
                self.modes1, self.modes2,
                dtype=torch.cfloat
            )
        )
        self.weights2 = nn.Parameter(
            self.scale * torch.rand(
                in_channels, out_channels,
                self.modes1, self.modes2,
                dtype=torch.cfloat
            )
        )

    def compl_mul2d(self, input, weights):
        # (batch, in_ch, x, y), (in_ch, out_ch, x, y)
        # -> (batch, out_ch, x, y)
        return torch.einsum("bixy,ioxy->boxy", input, weights)

    def forward(self, x):
        batchsize = x.shape[0]

        # 2D FFT
        x_ft = torch.fft.rfft2(x)

        # Multiply modes in four corners
        out_ft = torch.zeros(
            batchsize, self.out_channels,
            x.size(-2), x.size(-1) // 2 + 1,
            dtype=torch.cfloat, device=x.device
        )

        # Upper modes
        out_ft[:, :, :self.modes1, :self.modes2] = \
            self.compl_mul2d(
                x_ft[:, :, :self.modes1, :self.modes2],
                self.weights1
            )

        # Lower modes (due to periodicity)
        out_ft[:, :, -self.modes1:, :self.modes2] = \
            self.compl_mul2d(
                x_ft[:, :, -self.modes1:, :self.modes2],
                self.weights2
            )
```

```
59
60          # Inverse 2D FFT
61          x = torch.fft.irfft2(out_ft, s=(x.size(-2), x.size(-1)))
62
63          return x
```

Listing 8: SpectralConv2d

Remark 6.1: Why Two Weight Matrices in 2D?

Due to the symmetry of the FFT:

- Upper-left corner: $k_1 \geq 0, k_2 \geq 0$ (weights1)
- Lower-left corner: $k_1 < 0, k_2 \geq 0$ (weights2)
- Right half is redundant (real signal symmetry)

This efficiently represents all necessary Fourier modes while respecting symmetries.

7 Exercises: Building Your Own FNO

These exercises will guide you through implementing your own Fourier Neural Operator from scratch. Work through them sequentially—each builds on the previous ones.

7.1 Exercise Set 1: Understanding Fourier Transforms

Exercise 1.1: Manual FFT

Goal: Understand what FFT computes.
Task:

a) Create a simple sinusoidal function: $f(x) = \sin(2\pi \cdot 3x) + 0.5\sin(2\pi \cdot 7x)$, sampled at $n = 128$ points on $[0, 1]$.

b) Compute its FFT using `numpy.fft.fft` or `torch.fft.fft`.

c) Plot the magnitude of the Fourier coefficients. You should see peaks at frequencies 3 and 7.

d) Reconstruct the signal using inverse FFT and verify it matches the original.

Implementation Hints:

```
import numpy as np
import matplotlib.pyplot as plt

n = 128
x = np.linspace(0, 1, n, endpoint=False)
f = np.sin(2*np.pi*3*x) + 0.5*np.sin(2*np.pi*7*x)

# Your code here
```

Questions to Answer:

- What do the Fourier coefficients represent physically?
- Why do we see peaks at specific frequencies?
- What happens to the Fourier transform if you change n?

Exercise 1.2: Mode Truncation

Goal: Understand the effect of keeping only low-frequency modes.
Task:

a) Load a non-trivial function (e.g., a noisy signal or use the Allen-Cahn data).

b) Compute its FFT.

c) Set all Fourier coefficients beyond the first $k_{\max} = 8, 16, 32$ modes to zero.

d) Compute the inverse FFT for each case and plot the reconstructed signals.

e) Compute the relative L2 error for each truncation level.

Expected Observation: Higher $k_{\max}$ preserves more details. But for smooth functions, even low $k_{\max}$ gives good approximations.
Connection to FNO: This is exactly what FNO does! It learns a transformation of the low-frequency modes.

Exercise 1.3: Resolution Invariance Test

Goal: See firsthand how Fourier modes are resolution-invariant.
Task:

a) Create a function at resolution $n = 64$: $f(x) = x^2$.

b) Compute first 8 Fourier coefficients.

c) Upsample the function to $n = 128$ (just evaluate at more points).

d) Compute first 8 Fourier coefficients of the upsampled function.

e) Compare the coefficients. They should be nearly identical!

Understanding: The Fourier coefficients represent the intrinsic function, not the discretization.

7.2 Exercise Set 2: Implementing SpectralConv1d

Exercise 2.1: Complex Parameter Initialization

Goal: Learn to work with complex PyTorch tensors.
Task:

a) Create a complex tensor:

```
weights = nn.Parameter(
    torch.randn(8, 16, 10, dtype=torch.cfloat)
)
```

b) Print its shape, dtype, and a few values.

c) Perform a simple operation: multiply by i (imaginary unit).

d) Take the real part, imaginary part, and absolute value.

e) Compute its gradient using a dummy loss function.

Verify: PyTorch can backpropagate through complex operations!

Exercise 2.2: Einstein Summation Practice

Goal: Master `torch.einsum` for the complex multiplication.
Task:

a) Create two random tensors:

```
input = torch.randn(4, 8, 10)    # [batch, in_ch, freq]
weights = torch.randn(8, 16, 10) # [in_ch, out_ch, freq]
```

b) Implement the multiplication using:

- Manual loops
- `torch.matmul` (think carefully about dimensions!)
- `torch.einsum("bix,iox->box", ...)`

c) Verify all three methods give the same result.

d) Time each method. Which is fastest?

Expected Learning: `einsum` is both elegant and efficient.

Exercise 2.3: Complete SpectralConv1d Implementation

Goal: Implement the complete spectral convolution layer.
Task: Fill in the following skeleton:

```
class SpectralConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, modes):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.modes = modes

        # TODO: Initialize weights
        # Hint: Use dtype=torch.cfloat
        # Use Xavier initialization scaled appropriately

    def compl_mul1d(self, input, weights):
        # TODO: Implement complex multiplication
        # Hint: Use torch.einsum
        pass

    def forward(self, x):
        # TODO: Implement forward pass
        # 1. Apply rfft
        # 2. Multiply first 'modes' frequencies
        # 3. Apply irfft
        pass
```

Test Your Implementation:

```
layer = SpectralConv1d(in_channels=4, out_channels=8, modes=8)
x = torch.randn(2, 4, 64)  # [batch, channels, spatial]
output = layer(x)

assert output.shape == (2, 8, 64)
print("Test passed!")
```

Debug Checklist:

- Are shapes correct after each operation?
- Did you handle the complex dtype correctly?
- Does the output have the right shape?
- Can you backpropagate through it?

7.3 Exercise Set 3: Building the Full FNO

Exercise 3.1: Lifting and Projection Layers

Goal: Implement the input/output transformations.
Task:

a) Implement a lifting layer that maps from input channels (2, including grid) to width (32):

```
class LiftingLayer(nn.Module):
    def __init__(self, in_dim, width):
        # TODO: Initialize Linear layer

    def forward(self, x):
        # TODO: Apply transformation
        # Input: [batch, spatial, in_dim]
        # Output: [batch, spatial, width]
```

b) Implement a projection layer that maps from width back to output dimension (1):

```
class ProjectionLayer(nn.Module):
    def __init__(self, width, out_dim):
        # TODO: Can use multiple Linear layers
        # TODO: Include activation functions
```

c) Test both layers with dummy data.

Exercise 3.2: Fourier Layer Module

Goal: Combine SpectralConv1d with skip connections.
Task: Implement a single Fourier layer:

```
class FourierLayer(nn.Module):
    def __init__(self, width, modes):
        super().__init__()
        # TODO: Initialize SpectralConv1d
        # TODO: Initialize Conv1d for skip connection
        # TODO: Initialize activation function

    def forward(self, x):
        # TODO: Implement v' =   (K(v) + W(v))
        # Remember: SpectralConv1d expects [batch, ch, spatial]
        pass
```

Test:

```
layer = FourierLayer(width=32, modes=16)
x = torch.randn(4, 32, 128)
output = layer(x)
assert output.shape == x.shape
```

Exercise 3.3: Complete FNO1d

Goal: Assemble all components into the full model.
Task: Complete this architecture:

```
class FNO1d(nn.Module):
    def __init__(self, modes, width, n_layers=4):
        super().__init__()

        self.modes = modes
        self.width = width
        self.n_layers = n_layers

        # TODO: Initialize lifting layer
        # TODO: Initialize n_layers Fourier layers
        # Hint: Use nn.ModuleList
        # TODO: Initialize projection layer

    def forward(self, x):
        # x shape: [batch, spatial, 2] with [u(x), x]

        # TODO: Apply lifting
        # TODO: Permute to [batch, width, spatial]
        # TODO: Apply each Fourier layer sequentially
        # TODO: Permute back to [batch, spatial, width]
        # TODO: Apply projection

        return output  # Shape: [batch, spatial, 1]
```

Verification:

```
model = FNO1d(modes=16, width=32, n_layers=4)
x = torch.randn(2, 128, 2)
output = model(x)

assert output.shape == (2, 128, 1)
print(f"Model has {sum(p.numel() for p in model.parameters())}
    parameters")
```

7.4 Exercise Set 4: Training and Evaluation

Exercise 4.1: Data Preparation

Goal: Prepare data in the correct format.
Task:

a) Generate synthetic training data:

- Input: Random functions + grid
- Output: Transformed functions (apply some simple operator, e.g., derivative, smoothing)

b) Format as:

```
# Input: [n_samples, n_points, 2]  # [u(x), x]
# Output: [n_samples, n_points]
```

c) Create PyTorch DataLoaders with appropriate batch size.

d) Visualize a few training examples.

Bonus: Try loading and preprocessing the Allen-Cahn data.

Exercise 4.2: Training Loop

Goal: Train your FNO model.
Task: Implement a complete training loop:

```python
def train_fno(model, train_loader, val_loader, epochs=100):
    # TODO: Initialize optimizer (Adam)
    # TODO: Initialize learning rate scheduler
    # TODO: Initialize loss function (MSELoss)

    for epoch in range(epochs):
        # Training phase
        model.train()
        train_loss = 0
        for inputs, targets in train_loader:
            # TODO: Forward pass
            # TODO: Compute loss
            # TODO: Backward pass
            # TODO: Optimizer step

        # Validation phase
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for inputs, targets in val_loader:
                # TODO: Compute validation loss

        # TODO: Step learning rate scheduler
        # TODO: Print progress

    return model
```

Monitor:

- Training loss should decrease
- Validation loss should decrease (watch for overfitting!)
- Relative L2 error

Exercise 4.3: Resolution Invariance Experiment

Goal: Verify resolution invariance empirically.
Task:

a) Train your FNO on data at resolution $n = 64$.

b) Test on data at resolutions: $n = 32, 64, 128, 256$.

c) For each resolution, compute:

- Prediction error
- Inference time
- Memory usage

d) Create plots showing:

- Error vs. resolution
- Qualitative comparisons (plot predictions)

Expected Result: Error should remain roughly constant across resolutions (up to discretization effects). This proves resolution invariance!
Bonus: Compare with a standard CNN trained on the same data. Show that CNN fails on different resolutions.

7.5 Exercise Set 5: Advanced Topics

Exercise 5.1: Implement SpectralConv2d

Goal: Extend to 2D problems.
Task:

a) Implement `SpectralConv2d` following the 1D version.

b) Key differences:

- Use `torch.fft.rfft2` and `torch.fft.irfft2`
- Need two weight matrices (upper and lower modes)
- Einstein notation: `"bixy,ioxy->boxy"`

c) Test on 2D data (e.g., images or 2D PDE solutions).

Skeleton:

```
class SpectralConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, modes1, modes2):
        # TODO: Two weight matrices

    def compl_mul2d(self, input, weights):
        # TODO: 2D complex multiplication

    def forward(self, x):
        # TODO: rfft2, multiply corners, irfft2
```

Exercise 5.2: Hyperparameter Tuning

Goal: Understand the impact of hyperparameters.
Task: Systematically vary and test:

- Number of modes: $k_{\max} = 8, 16, 32$
- Width: $d = 16, 32, 64, 128$
- Number of layers: $L = 2, 3, 4, 5$
- Activation functions: Tanh, ReLU, GELU

For each configuration:

a) Count parameters

b) Measure training time per epoch

c) Record final validation error

Questions:

- What is the optimal trade-off between accuracy and efficiency?
- Does more always mean better?
- Which hyperparameters matter most?

Exercise 5.3: Apply to a Real PDE

Goal: Solve a PDE of your choice.
Suggested PDEs:

- Burgers' equation: $u_t + uu_x = \nu u_{xx}$
- Heat equation: $u_t = \alpha u_{xx}$
- Wave equation: $u_{tt} = c^2 u_{xx}$
- Navier-Stokes (2D)

Pipeline:

a) Generate or obtain training data

b) Train FNO to learn the solution operator

c) Test on unseen parameters

d) Compare with traditional numerical solver (accuracy and speed)

Report:

- Problem description
- Data generation procedure
- Model architecture and training
- Results and visualizations
- Comparison with baseline

7.6 Exercise Set 6: Understanding Limitations

Exercise 6.1: When FNO Fails

Goal: Understand the limitations.
Task: Test FNO on:

a) Discontinuous functions (e.g., step functions, shocks)

b) High-frequency functions (fast oscillations)

c) Functions with different boundary conditions than training

d) Extrapolation beyond the training distribution

Questions:

- Where does FNO struggle?
- Why does mode truncation cause issues with discontinuities?
- How could you modify FNO to handle these cases?

Exercise 6.2: Comparison with Other Architectures

Goal: Contextualize FNO in the operator learning landscape.
Task:

a) Implement a simple baseline:

- Fully connected network
- CNN
- U-Net

b) Train all models on the same task.

c) Compare:

- Parameter count
- Training time
- Inference time
- Accuracy on same resolution
- Accuracy on different resolutions (resolution invariance)

Create a table summarizing results.
Expected Finding: FNO shines in resolution invariance but may not always be the most accurate on a fixed resolution.

8 Advanced Theoretical Topics

8.1 Universal Approximation for Operators

Theorem 8.1: Universal Approximation (Informal)

Under mild conditions, neural operators (including FNO) can approximate any continuous operator between function spaces arbitrarily well, given sufficient width and depth.

This extends the classical universal approximation theorem from functions to operators!

8.2 Convergence Analysis

The approximation error of FNO can be decomposed as:

$$\text{Total Error} = \underbrace{\text{Approximation Error}}_{\text{Model capacity}} + \underbrace{\text{Discretization Error}}_{\text{Grid resolution}} + \underbrace{\text{Training Error}}_{\text{Optimization}} \tag{20}$$

Key Insights:

- **Approximation error** decreases with more modes and layers
- **Discretization error** is intrinsic to the problem, not the architecture
- **Training error** can be minimized with proper optimization

FNO's resolution invariance means the approximation error is independent of the discretization!

8.3 Connection to Green's Functions

The integral kernel operator learned by FNO can be interpreted as a learned Green's function:

$$(\mathcal{K}v)(x) = \int_{\Omega} G(x, y)v(y)dy \tag{21}$$

where $G(x, y)$ is implicitly represented through its Fourier modes.
This connects FNO to classical mathematical physics!

9 Implementation Best Practices

9.1 Memory Efficiency

- **Gradient Checkpointing:** For very deep FNOs, use activation checkpointing to trade compute for memory.
- **Mixed Precision:** FNO works well with FP16 training:

```
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()

with autocast():
    output = model(input)
    loss = loss_fn(output, target)

scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

- **Reduced Modes During Training:** Start with fewer modes, increase later.

9.2 Numerical Stability

- **Weight Initialization:** Proper initialization of complex weights matters:

```
# Use Xavier initialization scaled for complex numbers
std = np.sqrt(2.0 / (in_channels * out_channels))
weights = nn.Parameter(
    std * torch.randn(..., dtype=torch.cfloat)
)
```

- **Normalization:** Consider layer normalization or instance normalization between Fourier layers.
- **Residual Connections:** The skip connection Wv_t is crucial for training stability.

9.3 Debugging Checklist

When your FNO isn't working:

1. **Check shapes:** Print tensor shapes after each operation
2. **Verify FFT:** Ensure rfft/irfft reconstructs the input
3. **Inspect Fourier modes:** Are the learned weights reasonable?
4. **Test on simple data:** Can it fit a single example?

5. **Check gradients:** Use `torch.autograd.gradcheck`
6. **Visualize predictions:** Plot actual vs predicted
7. **Monitor losses:** Both MSE and relative L2 error
8. **Check for NaNs:** Add assertions throughout

10 Further Resources and Extensions

10.1 Recommended Papers

1. **Original FNO Paper:** Li, Z., et al. (2020). "Fourier Neural Operator for Parametric Partial Differential Equations." *arXiv:2010.08895*
2. **Survey:** Kovachki, N., et al. (2021). "Neural Operator: Learning Maps Between Function Spaces." *arXiv:2108.08481*
3. **Improvements:** Various papers on factorized FNO, adaptive FNO, etc.

10.2 Code Repositories

- Official: https://github.com/neuraloperator/neuraloperator
- PyTorch implementations with tutorials
- Pre-trained models available

10.3 Related Architectures

- **DeepONet:** Branch-Trunk architecture
- **Graph Neural Operators:** For irregular meshes
- **Transformer-based Operators:** Attention in function space
- **Physics-Informed FNOs:** Incorporating PDE constraints

10.4 Open Problems

- **Handling Discontinuities:** Better representations for shocks
- **Adaptive Resolution:** Learned multi-scale representations
- **Uncertainty Quantification:** Bayesian neural operators
- **Multi-Physics:** Coupled operators for complex systems

11 Conclusion

You now have a comprehensive understanding of Fourier Neural Operators:

- **Theory:** Why they work (Fourier analysis, operator theory)
- **Architecture:** How they're designed (lifting, Fourier layers, projection)
- **Resolution Invariance:** The key innovation (mode truncation, frequency domain learning)

- **Implementation:** PyTorch tricks and best practices
- **Practice:** Through hands-on exercises

Key Point

The Central Insight of FNOs:
By learning in the frequency domain and truncating to low modes, FNOs:

1. Learn operators in continuous function space
2. Are independent of discretization
3. Generalize across resolutions
4. Efficiently capture global dependencies

This is a paradigm shift from pixel-based to physics-based learning!

11.1 Next Steps

1. Work through all exercises systematically
2. Implement FNO from scratch
3. Apply to your own PDE problem
4. Read the original papers
5. Explore the official codebase
6. Consider extensions and improvements

Remember: The best way to understand FNO is to implement it yourself. Use this tutorial as your guide, but don't hesitate to experiment and explore!

A Appendix A: Mathematical Background

A.1 Fourier Series Review

For a periodic function $f : [0, L] \to \mathbb{R}$:

$$f(x) = \sum_{k=-\infty}^{\infty} c_k e^{2\pi i k x/L} \tag{22}$$

where the Fourier coefficients are:

$$c_k = \frac{1}{L} \int_0^L f(x) e^{-2\pi i k x/L} dx \tag{23}$$

A.2 Discrete Fourier Transform

For discrete samples $\{f_n\}_{n=0}^{N-1}$:

$$\hat{f}_k = \sum_{n=0}^{N-1} f_n e^{-2\pi i k n/N}, \quad k = 0, \ldots, N-1 \tag{24}$$

The FFT computes this in $O(N \log N)$ time instead of $O(N^2)$.

A.3 Convolution Theorem

Convolution in physical space is multiplication in Fourier space:

$$\widehat{(f * g)} = \hat{f} \cdot \hat{g} \tag{25}$$

This is why FNO is so efficient!

B Appendix B: PyTorch FFT Functions

B.1 Real FFT Functions

```
# 1D real FFT
torch.fft.rfft(input, n=None, dim=-1, norm=None)
# Returns complex tensor of size (..., n//2 + 1)

# 1D inverse real FFT
torch.fft.irfft(input, n=None, dim=-1, norm=None)
# Returns real tensor of size (..., n)

# 2D real FFT
torch.fft.rfft2(input, s=None, dim=(-2,-1), norm=None)

# 2D inverse real FFT
torch.fft.irfft2(input, s=None, dim=(-2,-1), norm=None)
```

Key Points:

- `n` specifies output length (useful for padding/truncation)
- `norm` can be 'forward', 'backward', or 'ortho'
- Real FFT exploits symmetry: $\hat{f}(-k) = \overline{\hat{f}(k)}$
- Output is complex even though input is real

C Appendix C: Common Errors and Solutions

Error	Solution
Shape mismatch in FFT	Ensure input has correct dimensions. FFT operates on last dimension.
Complex dtype mismatch	Check all operations preserve `torch.cfloat` dtype.
Mode index out of bounds	Ensure `modes` $\leq$ `n_points//2 + 1`
NaN in training	Check learning rate, weight initialization, and gradient norms.
Model doesn't learn	Verify data preprocessing, check if problem is learnable, try simpler task first.
Resolution invariance fails	Ensure you're using rfft/irfft correctly and modes are set appropriately.

Acknowledgments

This tutorial is based on the seminal work of Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Their paper "Fourier Neural Operator for Parametric Partial Differential Equations" revolutionized operator learning.