Geoscientific Machine Learning

https://geo-sciml.com

Pankaj K Mishra

Abstract

This book introduces geoscientific machine learning with Julia, combining practical coding workflows with scientific reasoning for Earth-system problems. It develops core machine-learning concepts, then advances to physics-informed and operator-learning methods used in scientific machine learning. The emphasis is on reproducible, computation-ready examples and on connecting data-driven models to governing physics, inverse problems, and geoscience applications.

Table of Contents

Preface **8**

I Julia Programming for Geoscience **9**

1 Getting started with Julia **10**
- 1.1 Install Julia and VS Code . . . 10
- 1.2 Three ways to run Julia code . . . 11
- 1.3 Hello, Julia! . . . 12
- 1.4 Your first calculations . . . 13
- 1.5 Reusing code with functions . . . 14
- 1.6 Making decisions and repeating things . . . 15
- 1.7 Organising data with custom types . . . 16
- 1.8 Getting comfortable with the REPL . . . 17
- 1.9 Everyday patterns . . . 17
- 1.10 Performance habits . . . 18
- 1.11 Customising your Julia setup . . . 19
- 1.12 The `.julia` directory: where packages live . . . 21
- 1.13 Using Julia behind a proxy . . . 22
- 1.14 Reproducible environments . . . 23

2 Files & Data Manipulation **28**
- 2.1 Writing and reading text files . . . 28
- 2.2 Delimited data with `DelimitedFiles` . . . 30
- 2.3 CSV files and DataFrames . . . 31
- 2.4 Working with file paths . . . 36
- 2.5 Searching, copying, and renaming files . . . 37
- 2.6 Geoscientific file formats . . . 41
- 2.7 Clean up . . . 43

3 Plotting & Data Visualisation **44**
- 3.1 The Makie ecosystem . . . 44
- 3.2 Installing CairoMakie . . . 45
- 3.3 Your first plot . . . 46
- 3.4 Scatter plots . . . 47
- 3.5 Line plots with multiple series . . . 49
- 3.6 Error bars . . . 50
- 3.7 Bar charts and histograms . . . 51
- 3.8 Heatmaps and contour plots . . . 53
- 3.9 Multi-panel figures (subplots) . . . 55

3.10 Customising appearance . . . 57
3.11 Log-scale axes . . . 59
3.12 Annotations and markers . . . 60
3.13 Saving figures . . . 61
3.14 Putting it together: a geoscience figure . . . 61
3.15 Geographic maps with GeoMakie . . . 62
3.16 Where to learn more . . . 65

II Neural Networks **66**

4 Neural Networks **67**
4.1 The basic picture . . . 67
4.2 How this part is organized . . . 68
4.3 Notation used in this part . . . 68
4.4 What to watch for . . . 68
4.5 Summary . . . 69

5 Your First Neural Network **70**
5.1 Sauna Satisfaction Predictor . . . 70
5.2 What just happened? . . . 73

6 Building Blocks of Neural Networks **74**
6.1 Notation used in this part . . . 74
6.2 The artificial neuron . . . 75
6.3 Activation functions . . . 75
6.4 Layers . . . 76
6.5 Networks: stacking layers . . . 77
6.6 The loss function . . . 77
6.7 Backpropagation and gradients . . . 78
6.8 Optimizers . . . 78
6.9 Regularization . . . 79
6.10 The training loop . . . 79
6.11 Minimum diagnostics checklist (for geoscience workflows) . . . 79
6.12 Summary . . . 80

7 Multilayer Perceptrons **81**
7.1 Architecture . . . 81
7.2 Universal approximation . . . 81
7.3 Code example: function approximation . . . 82
7.4 Effect of depth and width . . . 84
7.5 When to use multilayer perceptrons . . . 84
7.6 Geoscience applications . . . 85

8 Convolutional Neural Networks **86**
8.1 The convolution operation . . . 86
8.2 Pooling . . . 86
8.3 A typical CNN architecture . . . 87
8.4 Code example: classifying simple seismic image patches . . . 87

8.5 Key CNN architectures . . . 93
8.6 Geoscience applications . . . 93

9 Recurrent Neural Networks **94**
9.1 The simple RNN . . . 94
9.2 Long Short-Term Memory (LSTM) . . . 95
9.3 Gated Recurrent Unit (GRU) . . . 95
9.4 Code example: predicting a synthetic geophysical time series . . . 95
9.5 When to use RNNs . . . 98
9.6 Geoscience applications . . . 98

10 Transformers **100**
10.1 Self-attention . . . 100
10.2 Multi-head attention . . . 100
10.3 The transformer block . . . 101
10.4 Positional encoding . . . 101
10.5 Code example: minimal self-attention . . . 101
10.6 Advantages over RNNs . . . 103
10.7 Geoscience applications . . . 104

11 Graph Neural Networks **105**
11.1 Graphs: a brief reminder . . . 105
11.2 Message passing . . . 105
11.3 Graph Convolutional Network (GCN) . . . 106
11.4 Code example: node classification on a synthetic graph . . . 106
11.5 When to use GNNs . . . 110
11.6 Geoscience applications . . . 111

12 Autoencoders **112**
12.1 Architecture . . . 112
12.2 Variational Autoencoder (VAE) . . . 113
12.3 Code example: denoising autoencoder for geophysical signals . . . 113
12.4 When to use autoencoders . . . 116
12.5 Geoscience applications . . . 116

13 Generative Adversarial Networks **117**
13.1 The adversarial objective . . . 117
13.2 Architecture . . . 118
13.3 Code example: a more stable GAN for a rock-physics crossplot . . . 118
13.4 When to use GANs . . . 123
13.5 Geoscience applications . . . 123

14 Diffusion Models **124**
14.1 The forward noising process . . . 124
14.2 The learned reverse process . . . 125
14.3 Code example: learning a 1D porosity prior with a diffusion model . . . 125
14.4 When to use diffusion models . . . 128
14.5 Geoscience applications . . . 129

15 Flow Matching 130
15.1 The basic idea . . . 130
15.2 Why flow matching is interesting . . . 131
15.3 Code example: transporting Gaussian noise into a bimodal porosity prior . . . 131
15.4 When to use flow matching . . . 134
15.5 Geoscience applications . . . 134

III Scientific Machine Learning 136

16 Inverse Modeling 137
16.1 Forward problems versus inverse problems . . . 137
16.2 Why scientific machine learning helps . . . 138
16.3 The common template . . . 138
16.4 What this part covers . . . 138
16.5 What to keep in mind . . . 139
16.6 Summary . . . 139

17 Physics-Informed Neural Networks 140
17.1 The idea . . . 140
17.2 The PINN loss function . . . 141
17.3 Code example: solving the 1D heat equation . . . 141
17.4 PINN strengths and limitations . . . 144
17.5 Geoscience applications . . . 145

18 Physics-Based ML for Inversion 146
18.1 Why not just use a PINN? . . . 146
18.2 The inversion framework . . . 147
18.3 Implicit neural representations (INRs) . . . 147
18.4 Code example: 1D steady-state diffusion inversion . . . 148
18.5 Why INRs work well for geophysical inversion . . . 152
18.6 Connection to traditional inversion . . . 152
18.7 Geoscience applications . . . 153

19 DeepONet 154
19.1 The operator learning problem . . . 154
19.2 The universal approximation theorem for operators . . . 155
19.3 Architecture: branch and trunk . . . 155
19.4 Variants . . . 156
19.5 Code example: learning the antiderivative operator . . . 156
19.6 Key properties of DeepONet . . . 160
19.7 DeepONet vs FNO . . . 160
19.8 Geoscience applications . . . 161

20 Physics-Informed DeepONet 162
20.1 The key idea . . . 162
20.2 PI-DeepONet loss function . . . 163
20.3 When to use PI-DeepONet . . . 163
20.4 Code example: 1D diffusion–reaction operator . . . 164

20.5 How PI-DeepONet differs from PINN + DeepONet 168
20.6 The training trade-off 168
20.7 Geoscience applications 169

21 Fourier Neural Operator **170**
21.1 From convolution to spectral convolution 170
21.2 FNO architecture 171
21.3 Code example: FNO for the 1D advection equation 172
21.4 Resolution invariance 177
21.5 FNO variants 177
21.6 Comparison with other neural operators 178
21.7 Geoscience applications 178

IV Applications **180**

22 SciML Applications in Geoscience **181**
22.1 Seismology and seismic imaging 181
22.2 Hydrogeology and subsurface flow 182
22.3 Weather and climate science 183
22.4 Geothermal energy 183
22.5 Solid Earth geophysics 184
22.6 Geodynamics and mantle convection 184
22.7 Earthquake science 185
22.8 Cross-cutting themes 185
22.9 Choosing the right SciML approach 186

References **187**

Preface

Welcome to **Geoscientific Machine Learning**!

This book is for geoscience students, researchers, and practitioners who want to get started with scientific machine learning (SciML), where machine learning models are trained using domain knowledge about the problem, most often the physics and numerical structure behind it. It is also for machine learning practitioners who are building new methods and want to apply them to geoscientific problems, a domain with some of the most diverse and challenging datasets in science.

This is a living book hosted at www.geo-sciml.com. The website contains the book source code, and whenever the text or code changes, a new version of the site is rendered and published. Code outputs such as figures, tables, and results are updated as well, and the downloadable PDF is regenerated so it matches the latest version. The full website source can also be downloaded and run locally, for example in VS Code.

Machine learning and AI are evolving rapidly, and it is difficult to estimate how geoscience research and applications will change over the coming years. For that reason, this book is written as a live project rather than a conventional one-time publication. The content will evolve over time, tracking current research questions while keeping a forward-looking view. At the same time, stable and citable checkpoints matter, so PDF snapshots of the book will be published on arXiv, with a permanent archive of the corresponding source code for each snapshot on Zenodo.

The programming language used throughout is Julia (Bezanson et al., 2017). The first part of the book introduces the computational setup and teaches enough Julia to reproduce the results and continue learning with confidence. No prior programming experience is required. The book then covers the core ideas of machine learning and deep learning that underpin modern models, before moving into Scientific machine learning methods and geoscience-focused applications. The emphasis is on methods designed for systems governed by differential equations and physical structure, including physics-informed approaches and operator learning. The goal is to connect data-driven models with scientific reasoning in a way that is practical for real geoscientific problems.

If you find this book useful, please consider citing the current web edition as:

Mishra, P. K. (2026). *Geoscientific Machine Learning* [Online book]. https://geo-sciml.com

```
@book{mishra2026geosciml,
  title={Geoscientific Machine Learning},
  author={Pankaj K Mishra},
  url={https://geo-sciml.com},
  year={2026},
  note={Web edition}
}
```

Part I

Julia Programming for Geoscience

1 Getting started with Julia

You are starting from scratch, and that is perfectly fine. No prior programming experience is assumed.

In this chapter we will install Julia (Bezanson et al., 2017) and VS Code, write our first lines of code, and build up to functions, loops, and custom types. By the end you will be comfortable running Julia interactively, saving scripts, and managing packages in reproducible environments. If you have used another language before (Python, R, MATLAB), you will notice how little setup Julia needs to get going.

1.1 Install Julia and VS Code

You need two things: **Julia** (the language) and **VS Code** (the editor you will write code in). Install them in this order:

1. Install **VS Code** from code.visualstudio.com.
2. Install **Julia** using juliaup, a small tool that manages Julia versions for you:
   - **Windows (PowerShell)**
     ```
     winget install julia -s msstore
     juliaup add release
     ```
   - **macOS (Terminal)**
     ```
     curl -fsSL https://install.julialang.org | sh
     juliaup add release
     ```
   - **Linux (Terminal)**
     ```
     curl -fsSL https://install.julialang.org | sh
     juliaup add release
     ```
3. Open a new terminal and check that Julia is available:
   ```
   julia --version
   ```
   If it prints something like `julia version 1.11.2`, you are ready.

4. Open VS Code, go to the **Extensions** panel (the square icon on the left sidebar, or press Ctrl/Cmd+Shift+X), and search for **Julia**. Install the one published by **julialang**. This single extension gives you syntax highlighting, code execution, an integrated REPL, a debugger, and a plot viewer. That is everything you need to get started.

Reading code blocks in this book

Throughout this book you will see two styles of code block. **Julia code** has a thick coloured bar on the left edge:

```julia
println("I am Julia code")
```

Shell commands, configuration files, and other non-Julia text have a plain border with no coloured bar:

```
echo "I am a shell command"
```

If a block has the coloured side bar, you can type it into the Julia REPL or a `.jl` file. If it does not, it is meant for your terminal, a config file, or is just illustrative output.

1.2 Three ways to run Julia code

Before we start writing code, it helps to know that there are three ways to run Julia. Each is useful in different situations:

1.2.1 1. Inside VS Code (recommended for beginners)

This is the easiest way. Create a file ending in `.jl` (e.g., `test.jl`), type your code, and press **Ctrl+Enter** (Windows/Linux) or **Cmd+Enter** (macOS) to run one line or a selected block. The Julia extension opens a built-in REPL panel at the bottom of VS Code and sends your code there. You see the results immediately, and you can go back and edit your file at any time.

This is what we will use throughout this book. It combines the best of both worlds: your code is saved in a file you can come back to, but you can run pieces of it interactively.

1.2.2 2. Interactively in the REPL (command line)

Open a terminal and type `julia` to start the **REPL** (Read-Eval-Print Loop). You get a `julia>` prompt where you can type expressions and see results instantly:

```
$ julia

               _
   _       _ _(_)_     |
  (_)     | (_) (_)    |  Version 1.11.2
```

```
   _ _   _| |_  __ _   |
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |
 _/ |\__'_|_|_|\__'_|  |
|__/                   |

julia> 2 + 3
5

julia> sqrt(144)
12.0
```

The REPL is great for quick experiments: testing a formula, checking how a function works, or exploring a dataset. But nothing is saved to a file, so it is not ideal for work you want to keep.

1.2.3 3. Running a script from the terminal

If you have a file called `myscript.jl`, you can run the whole thing at once from the terminal:

```
julia myscript.jl
```

Julia reads the file from top to bottom, executes every line, prints any output, and exits. This is how you run production code, batch jobs on a cluster, or automated workflows. You don't see intermediate results, only what the script explicitly prints.

1.2.4 Which one should I use?

Situation	Best option
Learning and exploring	VS Code (Ctrl/Cmd+Enter) or REPL
Developing code you want to keep	VS Code (code is in a file, but you run it interactively)
Quick one-off calculations	REPL
Running a finished analysis on a cluster	`julia myscript.jl`
Automated or scheduled jobs	`julia myscript.jl`

You will naturally switch between these as you get more comfortable. For now, open VS Code and follow along.

1.3 Hello, Julia!

Let's make sure everything works. In VS Code, create a new file called `test.jl` and type the following:

```
println("Hello, Julia!")
x = [1, 2, 3, 4, 5]
println("Sum: ", sum(x))
```

Press **Ctrl+Enter** (Windows/Linux) or **Cmd+Enter** (macOS) to run each line. You should see:

```
Hello, Julia!
Sum: 15
```

Congratulations, you just ran your first Julia program. Everything after this point builds on what you just did.

1.4 Your first calculations

Julia works like a calculator right out of the box. You don't need to import anything or set anything up:

```
1 + 2, sin(1.0)
```

```
(3, 0.8414709848078965)
```

This computed `1 + 2` and the sine of `1.0` (in radians) and returned both results as a pair. The comma groups them into a **tuple**, just a way of showing multiple answers together.

1.4.1 Storing values in variables

In programming, a **variable** is a name you give to a value so you can use it later. Think of it as a sticky note on a number:

```
a = 10
b = 3
a + b, a - b, a * b, a / b
```

```
(13, 7, 30, 3.3333333333333335)
```

Here `a` is a label for `10` and `b` is a label for `3`. The next line asks Julia to add, subtract, multiply, and divide them, all at once.

You can name variables almost anything: `temperature`, `depth_km`, `n_samples`. Use names that describe what the value means. Your future self will thank you.

1.4.2 Collecting values in a list

In real work you almost never deal with just one number. You have a list of measurements, a column of data, a time series. In Julia, a list of values is called an **Array** (or **Vector** when it is one-dimensional). You create one with square brackets:

```
x = [1, 2, 3, 4, 5]
sum(x), length(x)
```

```
(15, 5)
```

`sum(x)` adds up all the values. `length(x)` counts how many there are. No imports, no setup.

1.4.3 What about text?

Not everything is a number. Julia handles text (called **strings**) too. Strings are wrapped in double quotes:

```
name = "Julia"
println("Hello, $name!")
```

The `$` inside a string inserts the value of a variable. This is called **string interpolation**. You can also insert expressions: `"2 + 3 = $(2 + 3)"`.

1.5 Reusing code with functions

If you find yourself doing the same calculation over and over, wrap it in a **function**. A function is like a recipe: you give it inputs (ingredients) and it gives you back a result (the dish).

The simplest way to define a function is the one-liner:

```
f(x) = x^2 + 2x + 1
f(3)
```

```
16
```

This says: "define `f` so that it takes `x`, squares it, adds twice `x`, adds `1`, and returns the result." Now `f(3)` gives `16`, `f(10)` gives `121`, and so on.

For anything longer than one line, use the multi-line form:

```
function kinetic_energy(mass, velocity)
    return 0.5 * mass * velocity^2
end
```

```
kinetic_energy(70.0, 3.0)   # a 70 kg person walking at 3 m/s
```

Functions can also return more than one value. Unpack them into separate variables:

```
function minmax(x)
    return minimum(x), maximum(x)
end

lo, hi = minmax([4, 1, 7, 2])
println("min = $lo, max = $hi")
```

Why functions matter

Julia's compiler makes code inside functions run very fast, often as fast as C or Fortran. Code typed at the top level (outside a function) cannot be optimised the same way. So the habit of wrapping your work in functions is not just about tidiness, it is about speed.

1.6 Making decisions and repeating things

1.6.1 If / else: choosing what to do

Programs often need to choose what to do based on a condition. In Julia, this reads almost like English:

```
x = 5
if x > 0
    println("positive")
elseif x < 0
    println("negative")
else
    println("zero")
end
```

```
positive
```

`println` means "print this line of text." Julia checks the conditions from top to bottom and runs the first one that is true, then skips the rest.

1.6.2 Loops: repeating work

A **loop** lets you repeat something many times without copying and pasting. The most common loop says "for each value in this range, do something":

```julia
for i in 1:5
    println(i^2)
end
```

```
1
4
9
16
25
```

`1:5` means the numbers 1, 2, 3, 4, 5. For each one, Julia squares it and prints the result. In geoscience, you might use a loop to process every station in a survey, every layer in a model, or every time step in a simulation.

1.7 Organising data with custom types

A geoscience measurement is rarely just a single number. It has coordinates, a value, an uncertainty, maybe a timestamp. Julia lets you bundle related values together using a **struct** (short for structure):

```julia
struct Point
    x::Float64
    y::Float64
end

p = Point(1.0, 2.0)
p.x, p.y
```

```
(1.0, 2.0)
```

This creates a new type called `Point` with two fields. `Float64` means "a decimal number with double precision." Once defined, you create a point by writing `Point(1.0, 2.0)` and access its parts with `p.x` and `p.y`.

Why bother with structs?

As your projects grow, structs keep things organised. Instead of passing around loose variables like `lat`, `lon`, `elevation`, you pass a single `Station` object that holds all of them together. Julia's compiler also uses the type information to make your code faster.

1.8 Getting comfortable with the REPL

The **REPL** (Read-Eval-Print Loop) is the interactive prompt you see when you type `julia` in a terminal. It is a great place to try things out quickly.

1.8.1 Built-in modes

The REPL has several modes. You switch by pressing a single key at the start of an empty line:

Key	Mode	What it does
`]`	**Pkg**	Package manager: add, remove, update packages
`;`	**Shell**	Run shell commands without leaving Julia
`?`	**Help**	Look up documentation for any function or type
Backspace	**Julia**	Return to normal Julia mode

Try it: press `?` then type `sum` and hit Enter. Julia shows you the documentation for `sum`, with examples.

1.8.2 Tab completion and Unicode

Press **Tab** to autocomplete names. This works for functions, variables, file paths, and even Unicode:

- Type `pri` then Tab ⟶ `print`
- Type `\alpha` then Tab ⟶ `α`
- Type `\sqrt` then Tab ⟶ `√`

This is how you type mathematical symbols in Julia code. Any LaTeX name works.

1.8.3 Handy shortcuts

- **Semicolons suppress output**: `x = rand(1000);` assigns the array without printing all 1000 numbers.
- **`ans` holds the last result**: after typing `2 + 3`, typing `ans * 10` gives `50`.
- **Underscores in numbers**: write `1_000_000` instead of `1000000` for readability. Julia ignores the underscores.

1.9 Everyday patterns

These are practical patterns you will use constantly.

1.9.1 Broadcasting with the dot `.`

Apply any function or operator to every element of an array by adding a dot:

```
x = [1, 2, 3, 4]
sin.(x)              # sine of each element
x .^ 2               # square each element
x .+ 10              # add 10 to each element
```

This works with your own functions too:

```
f(x) = x^2 + 1
f.([1, 2, 3])        # returns [2, 5, 10]
```

1.9.2 Finding documentation and methods

Use `methods()` to see all versions of a function, and `@which` to find out which version gets called:

```
methods(+)                # all methods for +
@which 1.0 + 2.0          # which method handles Float64 + Float64
```

1.9.3 Timing your code

Use `@time` for a quick measurement. Run it **twice** because the first call includes compilation time:

```
@time sum(rand(1_000_000))     # first call: includes compilation
@time sum(rand(1_000_000))     # second call: actual speed
```

For serious benchmarking, use the BenchmarkTools package:

```
using BenchmarkTools
@btime sum(rand(1_000_000))
```

1.10 Performance habits

These two habits will save you hours of debugging slow code.

1.10.1 Put work inside functions

Code at the top level (the global scope) runs slowly because Julia cannot predict the types of global variables. The same code inside a function runs orders of magnitude faster:

```julia
# Slow — top level
x = rand(1_000_000)
total = 0.0
for val in x
    total += val
end

# Fast — inside a function
function mysum(x)
    total = 0.0
    for val in x
        total += val
    end
    return total
end
mysum(rand(1_000_000))
```

1.10.2 Loop in the right order

Julia stores matrices **column by column** in memory (like Fortran), not row by row (like C or Python). When looping over a matrix, always loop over rows in the inner loop:

```julia
A = rand(1000, 1000)

# Fast — columns in the outer loop, rows in the inner loop
for j in 1:1000       # column
    for i in 1:1000   # row
        A[i, j] += 1
    end
end
```

1.11 Customising your Julia setup

1.11.1 Startup file

If you load the same packages every session, add them to your startup file. Julia runs this file automatically on launch.

The file lives at `~/.julia/config/startup.jl`. Create the directory and file if they don't exist:

```bash
mkdir -p ~/.julia/config
echo 'try using Revise catch end' > ~/.julia/config/startup.jl
```

On Windows (PowerShell):

```
New-Item -ItemType Directory -Force -Path "$env:USERPROFILE\.julia\config"
Set-Content "$env:USERPROFILE\.julia\config\startup.jl" 'try using Revise catch end'
```

> **Revise.jl**
>
> Revise watches your source files and automatically picks up changes without restarting Julia. Install it once with `] add Revise`, then load it from your startup file as shown above. It is one of the most useful Julia packages.

1.11.2 Point VS Code to Julia manually

In most cases VS Code finds Julia automatically. If it does not, go to **Settings → Julia: Executable Path** and enter the path:

- **Windows**

  `%LOCALAPPDATA%\Programs\Julia-*\bin\julia.exe`

  or

  `%USERPROFILE%\.juliaup\bin\julia.exe`

- **macOS**

  `/Applications/Julia-*.app/Contents/Resources/julia/bin/julia`

  or

  `$HOME/.juliaup/bin/julia`

- **Linux**

  `/usr/bin/julia`

  or

  `$HOME/.juliaup/bin/julia`

OS	Default location

1.12 The `.julia` directory: where packages live

When you install Julia and start adding packages, Julia creates a hidden directory called `.julia` in your home folder:

OS	Default location
Windows	`C:\Users\YourName\.julia\`
macOS	`/Users/YourName/.julia/`
Linux	`/home/YourName/.julia/`

This directory is called the **depot**. It stores everything Julia needs:

Subfolder	What's inside
`packages/`	Source code of every package you install
`compiled/`	Precompiled versions of those packages
`artifacts/`	Binary libraries and datasets that packages download
`registries/`	The General registry, Julia's index of all public packages
`logs/`	Build and install logs
`environments/`	Project environments and their `Manifest.toml` files
`config/`	Your `startup.jl` file

1.12.1 Why it grows large

Every package brings its own dependencies, compiled caches, and sometimes large binary artefacts. A plotting library like CairoMakie can pull in hundreds of megabytes. Over time, `.julia` can easily reach **5–20 GB**.

On a personal laptop this is fine. But on **university HPC clusters and shared Linux servers**, your home directory often has a quota, sometimes as little as 5 or 10 GB. If `.julia` fills it up, you will see "disk quota exceeded" errors and Julia will stop installing packages.

1.12.2 Moving `.julia` to a larger disk

Set the `JULIA_DEPOT_PATH` environment variable to point somewhere with more space:

- **Windows (PowerShell, persist permanently)**

  ```
  setx JULIA_DEPOT_PATH "D:\JuliaDepot"
  ```

- **macOS / Linux** — add to `~/.bashrc` or `~/.zshrc`:

```
export JULIA_DEPOT_PATH="$HOME/julia-depot"
```

- **On a cluster** (use scratch or project space):

```
export JULIA_DEPOT_PATH="/scratch/$USER/julia-depot"
```

Already have a large `.julia` in your home?

Move the existing directory first, then set the variable:

```
mv ~/.julia /scratch/$USER/julia-depot
export JULIA_DEPOT_PATH="/scratch/$USER/julia-depot"
```

On Windows (PowerShell):

```
Move-Item "$env:USERPROFILE\.julia" "D:\JuliaDepot"
setx JULIA_DEPOT_PATH "D:\JuliaDepot"
```

1.13 Using Julia behind a proxy

On university and corporate networks, internet traffic often goes through a proxy. Julia needs to know about this to download packages. If you skip this step, `Pkg.add(...)` will hang or time out.

Ask your IT department for the proxy address. Then set it as an environment variable:

Windows (PowerShell, persist permanently)

```
setx HTTP_PROXY  "http://proxy.example.com:8080"
setx HTTPS_PROXY "http://proxy.example.com:8080"
```

Close and reopen your terminal after running `setx`.

macOS / Linux — add to `~/.bashrc` or `~/.zshrc`:

```
export HTTP_PROXY="http://proxy.example.com:8080"
export HTTPS_PROXY="http://proxy.example.com:8080"
```

Then reload:

```
source ~/.zshrc   # or ~/.bashrc
```

If authentication is required, include credentials in the URL:

```
http://username:password@proxy.example.com:8080
```

1.13.1 Verify it works

Open the Julia REPL and try:

```
ENV["HTTP_PROXY"]        # should print your proxy address
using Pkg
Pkg.add("Example")       # should download successfully
```

No-proxy for local addresses

If internal servers should bypass the proxy:

```
export NO_PROXY="localhost,127.0.0.1,.internal.example.com"
```

On Windows (PowerShell):

```
$env:NO_PROXY = "localhost,127.0.0.1,.internal.example.com"
```

1.14 Reproducible environments

Imagine you wrote a script that reads geoscientific data, runs an inversion, and produces a beautiful map. It works perfectly today. Six months later your colleague tries to run it and gets errors everywhere. A package was updated, a function was renamed, a dependency now needs a newer Julia version. This is the *"it works on my machine"* problem, and it is one of the biggest obstacles to reproducible science.

Julia's **built-in package manager** solves this at the language level. Every Julia project can carry two small text files that record *exactly* which packages (and which versions) were used. Anyone, anywhere, can recreate the identical environment with a single command.

1.14.1 What a Julia project looks like

A typical Julia project is just a folder with a few files:

```
MyProject/
├── Project.toml      # what you depend on
├── Manifest.toml     # exact snapshot of every version
└── src/
    └── MyProject.jl  # your code
```

You may also have a `test/`, `docs/`, or `scripts/` folder, but the two `.toml` files are the heart of reproducibility.

1.14.2 `Project.toml`: what your project needs

This file lists the **direct dependencies**, the packages *you* asked for. Here is the `Project.toml` for this very book:

```
[deps]
CSV = "336ed68f-0bac-5ca0-87d4-7b16caf5d00b"
CairoMakie = "13f3f980-e62b-5c42-98c6-ff1f3baf88f0"
DataFrames = "a93c6f00-e57d-5684-b7b6-d8193f3e46c0"
DelimitedFiles = "8bb1440f-4735-579b-a4ab-409b98df4dab"
Statistics = "10745b16-79ce-11e8-11f9-7d13ad32a3b2"
```

Each entry has a name and a **UUID** (a unique identifier so Julia never confuses two packages with the same name). You rarely type UUIDs by hand; the package manager fills them in when you run `Pkg.add("CSV")`.

You can optionally add a `[compat]` section to pin version ranges:

```
[compat]
CSV = "0.10"
DataFrames = "1.6"
julia = "1.10"
```

This tells Julia: "My code is tested with CSV 0.10.x, DataFrames 1.6.x, and Julia 1.10 or newer." It prevents someone from accidentally installing an incompatible future version.

1.14.3 `Manifest.toml`: the exact snapshot

When you install packages, Julia resolves every dependency (and every dependency's dependency) and writes the result into `Manifest.toml`. This file records the **exact version** of every single package in the tree. It can be hundreds of lines long, and that is normal.

Think of it this way:

File	Analogy	You edit it?
`Project.toml`	A recipe's ingredient list ("flour, eggs, sugar")	Yes
`Manifest.toml`	The supermarket receipt with exact brands and batch numbers	No (generated automatically)

Should you commit `Manifest.toml` to Git?

For applications and papers: yes. It guarantees bit-for-bit reproducibility. Anyone who checks out your repo and runs `Pkg.instantiate()` gets exactly the same package versions.
For libraries (reusable packages): usually no. You want downstream users to resolve versions against *their* environment, not yours.

1.14.4 Creating and using environments

Start a new project in any empty folder:

```
using Pkg
Pkg.activate(".")      # use the current folder as the environment
Pkg.add("CSV")         # adds CSV to Project.toml and resolves Manifest.toml
Pkg.add("DataFrames")
```

Or in the REPL's Pkg mode (press]):

```
(@v1.11) pkg> activate .
(MyProject) pkg> add CSV DataFrames
```

Julia creates `Project.toml` and `Manifest.toml` for you.

Reproduce someone else's environment:

```
using Pkg
Pkg.activate(".")       # point to the folder containing their Project.toml
Pkg.instantiate()       # download the exact versions from Manifest.toml
```

That is it. Two commands and you have an identical setup. No guessing, no version conflicts.

1.14.5 The default (global) environment

When you start Julia without activating a project, you are in the **global environment** (shown as `@v1.11` in the REPL prompt). Packages you add here are available everywhere, which is convenient for everyday tools like `Revise` or `BenchmarkTools`. But for actual research work, always create a **project-specific environment** so your results are reproducible.

How other languages handle this

If you come from **Python**, Julia's `Project.toml` is similar to `requirements.txt` or `pyproject.toml`, and `Manifest.toml` is like a `pip freeze` snapshot or a Poetry/PDM lock file. The difference is that Julia's package manager is built into the language, with no separate tool to install.

> If you come from **C++**, reproducible builds are traditionally much harder. You might use CMake with pinned versions, Conan, or vcpkg, but there is no single built-in standard. Julia's approach is closer to Rust's `Cargo.toml` / `Cargo.lock` system.
> In all cases the idea is the same: **separate what you want from what you got**, so someone else can recreate the exact same setup later.

1.14.6 Key Pkg commands explained

Here is what the most common `Pkg` functions actually do:

`Pkg.activate(".")` **Switch into a project environment.** This tells Julia: "use the `Project.toml` in this folder instead of the global one." The REPL prompt changes from `(@v1.11)` to the project name. Nothing is installed yet; you are just pointing Julia at a folder.

`Pkg.add("CSV")` **Add a package.** Writes the package name and UUID into `Project.toml`, resolves a compatible set of versions, downloads everything needed, and records the result in `Manifest.toml`. If `Project.toml` does not exist yet, Julia creates it for you.

`Pkg.instantiate()` **Recreate an environment from existing files.** If a `Manifest.toml` is present, it downloads the *exact* versions listed there. If only a `Project.toml` exists (no manifest), it resolves fresh versions and creates the manifest. This is the command you run after cloning someone else's repository.

`Pkg.precompile()` **Precompile installed packages ahead of time.** Julia turns package code into cached compiled form so the first real run is smoother. This is useful right after `Pkg.add()`, `Pkg.instantiate()`, or `Pkg.update()`, especially if you just changed environments, pulled a fresh repository, or want to avoid a long compile pause when you start working.

`Pkg.resolve()` **Re-resolve versions without adding or removing anything.** Useful after you edit `Project.toml` by hand (for example, adding a `[compat]` entry). It updates `Manifest.toml` to match the current `Project.toml` without downloading new packages.

`Pkg.update()` **Update all packages** to the newest versions that are still compatible with your `[compat]` constraints. This rewrites `Manifest.toml`. Run this periodically to pick up bug fixes and performance improvements.

`Pkg.status()` **List installed packages** and their versions. In Pkg-mode just type `st` or `status`.

`Pkg.pin("CSV")` **Lock a package at its current version** so that `Pkg.update()` will not change it. Useful when you know a newer release breaks your workflow. Unpin with `Pkg.free("CSV")`.

`Pkg.rm("CSV")` **Remove a package** from the project. Deletes it from `Project.toml` and re-resolves the manifest.

`Pkg.gc()` **Garbage-collect.** Removes cached packages that are no longer used by any environment on your machine. Helpful when your `.julia` folder grows large (see the earlier section on the `.julia` directory).

`Pkg.build()` **Re-run build scripts** for installed packages. Occasionally needed after system-level changes (new compiler, updated shared libraries).

1.14.7 Quick reference

Task	REPL Pkg-mode (])	Script
Activate an environment	`activate .`	`Pkg.activate(".")`
Add a package	`add CSV`	`Pkg.add("CSV")`
Remove a package	`rm CSV`	`Pkg.rm("CSV")`
Recreate from lock file	`instantiate`	`Pkg.instantiate()`
Precompile installed packages	`precompile`	`Pkg.precompile()`
Re-resolve after hand edits	`resolve`	`Pkg.resolve()`
See what is installed	`status`	`Pkg.status()`
Update all packages	`update`	`Pkg.update()`
Pin a version	`pin CSV@0.10.14`	`Pkg.pin("CSV", v"0.10.14")`
Unpin	`free CSV`	`Pkg.free("CSV")`
Clean unused caches	`gc`	`Pkg.gc()`
Re-run build scripts	`build`	`Pkg.build()`

Learn more

The official **Pkg documentation** covers everything in detail, including custom registries, private packages, artifacts, and more:
https://pkgdocs.julialang.org/

With these two files and a handful of commands, your work stays reproducible, whether you revisit it next year or share it with a colleague on the other side of the world.

2 Files & Data Manipulation

At this point you know how to install Julia, run code in VS Code or the REPL, write functions, and create reproducible environments with `Project.toml`.

In this chapter we will put those skills to work on something you will do constantly in geoscience: **reading data from files, reshaping it, and writing results back out**. We start with plain text files, move on to delimited numeric data, then to CSV tables with the `DataFrames` ecosystem, and finish with a tour of specialised geoscientific formats (NetCDF, HDF5, SEG-Y, Shapefiles). We create small example files right here in the code so you can run every block and see the results yourself.

If some of the syntax looks unfamiliar at first, especially the `do ... end` pattern or the ⇒ arrow, don't worry. These are common Julia patterns that will become second nature with practice. The important thing is to understand *what* each block does, not to memorise the syntax right now.

2.1 Writing and reading text files

The simplest kind of file is plain text. Let's create one:

```
# Create a small text file
open("stations.txt", "w") do f
    write(f, "Station  Latitude  Longitude  Elevation_m\n")
    write(f, "AAL      64.35     25.75      120\n")
    write(f, "BBR      61.50     23.80      95\n")
    write(f, "CCK      60.17     24.94      15\n")
end
```


What happened here? `open("stations.txt", "w")` creates (or overwrites) a file named `stations.txt`. The `"w"` means "write mode." The `do f ... end` block is Julia's way of saying "while the file is open, call it `f` and do these things, then close it automatically." Each `write` call puts a line of text into the file.

Now let's read it back:

```
content = read("stations.txt", String)
println(content)
```

```
Station  Latitude  Longitude  Elevation_m
AAL      64.35     25.75      120
BBR      61.50     23.80      95
CCK      60.17     24.94      15
```

`read("stations.txt", String)` loads the entire file into a single string. For a small file like this, that is perfectly fine.

2.1.1 Reading line by line

Sometimes you want to process a file one line at a time, for instance to skip a header or to parse each row:

```
open("stations.txt", "r") do f
    for line in eachline(f)
        println("→ ", line)
    end
end
```

```
→ Station  Latitude  Longitude  Elevation_m
→ AAL      64.35     25.75      120
→ BBR      61.50     23.80      95
→ CCK      60.17     24.94      15
```

The `"r"` means “read mode.” `eachline(f)` gives you one line at a time, which is memory-efficient even for very large files.

2.1.2 Splitting lines into columns

In geoscience data files, columns are usually separated by spaces, tabs, or commas. You can split a line into pieces with `split()`:

```
open("stations.txt", "r") do f
    header = readline(f)              # read and skip the header
    for line in eachline(f)
        parts = split(line)           # splits on whitespace by default
        name  = parts[1]
        lat   = parse(Float64, parts[2])
        lon   = parse(Float64, parts[3])
        elev  = parse(Float64, parts[4])
        println("$name is at ($lat, $lon), elevation $elev m")
    end
end
```

```
AAL is at (64.35, 25.75), elevation 120.0 m
BBR is at (61.5, 23.8), elevation 95.0 m
CCK is at (60.17, 24.94), elevation 15.0 m
```

`split(line)` breaks the string into a list of substrings. `parse(Float64, ...)` converts a text string like `"64.35"` into an actual number. This pattern (read a line, split it, parse the pieces) is the bread and butter of working with geoscience text files.

2.2 Delimited data with `DelimitedFiles`

For files with a regular grid of numbers (like gravity measurements or temperature readings), Julia has a built-in module called `DelimitedFiles` that reads the whole file into a matrix in one step.

Let's create a small data file and read it:

```
# Create a space-delimited data file (no header, just numbers)
open("gravity.txt", "w") do f
    write(f, "64.35  25.75  -12.3\n")
    write(f, "64.40  25.80  -11.8\n")
    write(f, "64.45  25.85  -13.1\n")
    write(f, "64.50  25.90  -12.7\n")
    write(f, "64.55  25.95  -14.0\n")
end
```

```
20
```

```
using DelimitedFiles

data = readdlm("gravity.txt")       # reads into a matrix
println("Size: ", size(data))        # 5 rows, 3 columns
```

```
Size: (5, 3)
```

Now you can pull out individual columns:

```
lat = data[:, 1]         # all rows, first column
lon = data[:, 2]         # all rows, second column
gz  = data[:, 3]         # all rows, third column

println("Latitudes: ", lat)
println("Gravity anomalies: ", gz)
```

```
Latitudes: [64.35, 64.4, 64.45, 64.5, 64.55]
Gravity anomalies: [-12.3, -11.8, -13.1, -12.7, -14.0]
```

The `[:, 1]` notation means "all rows, column 1." This is the same notation used in MATLAB and NumPy.

You can also write a matrix back out:

```
# Add 1.0 to all gravity values and save
gz_corrected = gz .+ 1.0
output = hcat(lat, lon, gz_corrected)    # glue columns side by side
writedlm("gravity_corrected.txt", output, '\t')   # tab-separated

# Verify
println(read("gravity_corrected.txt", String))
```

```
64.35   25.75   -11.3
64.4    25.8    -10.8
64.45   25.85   -12.1
64.5    25.9    -11.7
64.55   25.95   -13.0
```

`hcat` stands for "horizontal concatenate". It sticks columns together into a matrix. `writedlm` writes the matrix to a file, using the tab character `'\t'` as the separator.

2.3 CSV files and DataFrames

CSV (Comma-Separated Values) is probably the most common data format in science. Julia has excellent CSV support through the **CSV.jl** and **DataFrames.jl** packages.

> **ℹ Installing packages**
>
> If you haven't installed these packages yet, press `]` in the REPL to enter Pkg mode and type:
>
> ```
> add CSV DataFrames
> ```
>
> Or from normal Julia code: `using Pkg; Pkg.add(["CSV", "DataFrames"])`. You only need to do this once.

2.3.1 Creating a DataFrame

A **DataFrame** is a table: rows and columns, like a spreadsheet. Each column has a name and all values in a column have the same type. If you have used pandas in Python, R's `data.frame`, or even Excel, the concept is the same.

```
using DataFrames

# Create a table of borehole samples
samples = DataFrame(
    borehole  = ["BH-01", "BH-01", "BH-01", "BH-02", "BH-02"],
    depth_m   = [10.0, 20.0, 30.0, 15.0, 25.0],
    rock_type = ["granite", "granite", "gneiss", "granite", "schist"],
    density   = [2.65, 2.68, 2.75, 2.63, 2.71]
)
```

	borehole	depth_m	rock_type	density
	String	Float64	String	Float64
1	BH-01	10.0	granite	2.65
2	BH-01	20.0	granite	2.68
3	BH-01	30.0	gneiss	2.75
4	BH-02	15.0	granite	2.63
5	BH-02	25.0	schist	2.71

Julia displays the table neatly. Each column is a named array, and you can work with them individually or together.

2.3.2 Writing and reading CSV

Let's save this table to a CSV file and read it back:

```
using CSV

# Write to CSV
CSV.write("samples.csv", samples)

# Read it back
samples_loaded = CSV.read("samples.csv", DataFrame)
```

	borehole	depth_m	rock_type	density
	String7	Float64	String7	Float64
1	BH-01	10.0	granite	2.65
2	BH-01	20.0	granite	2.68
3	BH-01	30.0	gneiss	2.75
4	BH-02	15.0	granite	2.63
5	BH-02	25.0	schist	2.71

That's it. One line to write, one line to read. CSV.jl automatically detects column types, handles headers, and deals with missing values.

2.3.3 Selecting columns

Grab one or more columns by name:

```
# One column - returns a vector
samples.depth_m
```

```
5-element Vector{Float64}:
 10.0
 20.0
 30.0
 15.0
 25.0
```

```
# Multiple columns - returns a new DataFrame
select(samples, :borehole, :density)
```

	borehole	density
	String	Float64
1	BH-01	2.65
2	BH-01	2.68
3	BH-01	2.75
4	BH-02	2.63
5	BH-02	2.71

The `:` before a column name makes it a **Symbol**, Julia's way of referring to names. You will see this pattern everywhere in DataFrames.

2.3.4 Filtering rows

Keep only rows that match a condition:

```
# Samples deeper than 20 m
filter(row -> row.depth_m > 20, samples)
```

	borehole	depth_m	rock_type	density
	String	Float64	String	Float64
1	BH-01	30.0	gneiss	2.75
2	BH-02	25.0	schist	2.71

The `row -> row.depth_m > 20` part is an **anonymous function**, a small throwaway function that takes one argument (`row`) and returns `true` or `false`. Don't worry about the syntax too much; think of it as saying "keep rows where depth is greater than 20."

```julia
# Only granite samples
filter(row -> row.rock_type == "granite", samples)
```

	borehole	depth_m	rock_type	density
	String	Float64	String	Float64
1	BH-01	10.0	granite	2.65
2	BH-01	20.0	granite	2.68
3	BH-02	15.0	granite	2.63

2.3.5 Adding and transforming columns

```julia
# Add a new column
samples.weight_kg = samples.density .* 1000.0   # density × 1000
samples
```

	borehole	depth_m	rock_type	density	weight_kg
	String	Float64	String	Float64	Float64
1	BH-01	10.0	granite	2.65	2650.0
2	BH-01	20.0	granite	2.68	2680.0
3	BH-01	30.0	gneiss	2.75	2750.0
4	BH-02	15.0	granite	2.63	2630.0
5	BH-02	25.0	schist	2.71	2710.0

The `.*` (dot-star) applies the multiplication to every row. You saw this broadcasting pattern in Chapter 1.

2.3.6 Grouping and summarising

This is where DataFrames really shine. Group your data by a category and compute summaries:

```julia
using Statistics

# Average density per borehole
gdf = groupby(samples, :borehole)
combine(gdf, :density => mean => :avg_density)
```

	borehole	avg_density
	String	Float64
1	BH-01	2.69333
2	BH-02	2.67

Read this as: "group the table by `:borehole`, then for the `:density` column compute the `mean` and call the result `:avg_density`." The ⇒ arrows connect input ⟶ function ⟶ output name.

```julia
# Multiple summaries at once
combine(
    groupby(samples, :rock_type),
    :density => mean => :avg_density,
    :density => minimum => :min_density,
    :depth_m => maximum => :deepest
)
```

	rock_type	avg_density	min_density	deepest
	String	Float64	Float64	Float64
1	granite	2.65333	2.63	20.0
2	gneiss	2.75	2.75	30.0
3	schist	2.71	2.71	25.0

2.3.7 Joining tables

In real projects your data is often spread across multiple files. Joining brings them together:

```julia
# A second table with borehole coordinates
locations = DataFrame(
    borehole  = ["BH-01", "BH-02", "BH-03"],
    latitude  = [64.35, 64.40, 64.45],
    longitude = [25.75, 25.80, 25.85]
)

# Inner join - only boreholes that appear in both tables
innerjoin(samples, locations, on = :borehole)
```

	borehole	depth_m	rock_type	density	weight_kg	latitude	longitude
	String	Float64	String	Float64	Float64	Float64	Float64
1	BH-01	10.0	granite	2.65	2650.0	64.35	25.75
2	BH-01	20.0	granite	2.68	2680.0	64.35	25.75
3	BH-01	30.0	gneiss	2.75	2750.0	64.35	25.75
4	BH-02	15.0	granite	2.63	2630.0	64.4	25.8
5	BH-02	25.0	schist	2.71	2710.0	64.4	25.8

`innerjoin` keeps only rows where the `:borehole` value exists in both tables. BH-03 has no samples, so it is dropped. If you want to keep all locations even when there are no matching samples, use `leftjoin` or `outerjoin` instead.

2.3.8 Handling missing data

Real data has gaps. Julia represents missing values with a special value called `missing`:

```
# Create data with a gap
obs = DataFrame(
    station = ["A", "B", "C", "D"],
    temp_C  = [12.3, missing, 14.1, 13.7]
)
obs
```

	station	temp_C
	String	Float64?
1	A	12.3
2	B	*missing*
3	C	14.1
4	D	13.7

```
# Drop rows with missing values
dropmissing(obs, :temp_C)
```

	station	temp_C
	String	Float64
1	A	12.3
2	C	14.1
3	D	13.7

```
# Compute mean, skipping missing values
using Statistics
mean(skipmissing(obs.temp_C))
```

```
13.366666666666665
```

`skipmissing` is a wrapper that tells functions to ignore the gaps. Without it, `mean` would return `missing` because any operation involving `missing` propagates the unknown.

2.4 Working with file paths

When your project has many data files, you need to build file paths correctly. Julia's `joinpath` function handles this across operating systems (so you don't have to worry about `/` vs `\`):

```
# Build a path
path = joinpath("data", "field_campaign", "gravity.txt")
println(path)
```

```
data/field_campaign/gravity.txt
```

Other useful path functions:

```
println("Current directory: ", pwd())
println("Files here: ", readdir("."))
```

```
# Check if a file exists before reading
if isfile("samples.csv")
    println("samples.csv exists!")
else
    println("File not found")
end
```

```
samples.csv exists!
```

Organise your data early

Create a `data/` folder in your project from the start. Put raw data in `data/raw/`, processed data in `data/processed/`. This habit will save you from chaos later, especially when you have 50 stations and 3 field campaigns.

2.5 Searching, copying, and renaming files

In practice, geoscientific data rarely arrives in one tidy folder. You might receive a USB drive with hundreds of files scattered across nested directories, including raw measurements from field instruments, GPS logs, photos, metadata files, all mixed together. A common first task is: **find all files of a certain type, rename them consistently, and copy them into a single clean folder.**

Julia has everything you need for this built into the standard library, no extra packages required.

2.5.1 Listing files in a directory

```
# readdir returns the names of everything in a folder
for name in readdir(".")
    println(name)
end
```

This is one of those examples that is more useful when you run it in your own project than when I freeze the output here, because the exact listing depends on whatever happens to be in the working directory at render time.

By default `readdir` gives you the names without the full path. Pass `join = true` to get complete paths:

```
for path in readdir("data/raw", join = true)
    println(path)
end
```

2.5.2 Searching recursively with `walkdir`

`walkdir` is the key function for searching through directories and all their subdirectories. It yields a tuple `(root, dirs, files)` at each level:

```
# Find every .csv file anywhere inside data/
for (root, dirs, files) in walkdir("data")
    for f in files
        if endswith(f, ".csv")
            full_path = joinpath(root, f)
            println(full_path)
        end
    end
end
```

You can wrap this into a reusable function:

```
"""
    find_files(dir, ext)

Recursively find all files with extension `ext` in `dir` and its subdirectories.
Returns a vector of full paths.
"""
function find_files(dir, ext)
    results = String[]
    for (root, dirs, files) in walkdir(dir)
        for f in files
            if endswith(f, ext)
                push!(results, joinpath(root, f))
            end
        end
    end
    return results
end
```

```
Main.Notebook.find_files
```

```
# Example: find all .dat files under a field campaign folder
dat_files = find_files("data/raw", ".dat")
```

```
println("Found $(length(dat_files)) .dat files")
```

2.5.3 Extracting parts of a file path

Julia has several functions for splitting a path into its components:

```
example_path = joinpath("surveys", "2024", "gravity", "station_042.csv")

println("Directory:  ", dirname(example_path))
println("Filename:   ", basename(example_path))
println("Name only:  ", splitext(basename(example_path))[1])
println("Extension:  ", splitext(basename(example_path))[2])
println("Split path: ", splitpath(example_path))
```

```
Directory:  surveys/2024/gravity
Filename:   station_042.csv
Name only:  station_042
Extension:  .csv
Split path: ["surveys", "2024", "gravity", "station_042.csv"]
```

2.5.4 Copying and renaming files

Suppose you have collected EM data from multiple campaigns and the instrument saved files with unhelpful names like `meas_001.dat`. You want to copy them into a single `processed/` folder, renaming each file to include the station name and date.

Here is a complete worked example. We first create a fake directory tree so you can run everything:

```
# --- Set up a fake directory tree ---
for station in ["EM01", "EM02", "EM03"]
    dir = joinpath("fake_survey", station)
    mkpath(dir)    # like mkdir -p: creates all intermediate directories
    for i in 1:3
        filepath = joinpath(dir, "meas_$(lpad(i, 3, '0')).dat")
        write(filepath, "fake data for $station measurement $i\n")
    end
end
println("Created fake survey tree:")
for (root, dirs, files) in walkdir("fake_survey")
    for f in files
        println("  ", joinpath(root, f))
    end
end
```

```
Created fake survey tree:
  fake_survey/EM01/meas_001.dat
  fake_survey/EM01/meas_002.dat
  fake_survey/EM01/meas_003.dat
  fake_survey/EM02/meas_001.dat
  fake_survey/EM02/meas_002.dat
  fake_survey/EM02/meas_003.dat
  fake_survey/EM03/meas_001.dat
  fake_survey/EM03/meas_002.dat
  fake_survey/EM03/meas_003.dat
```

Now let's search for all `.dat` files, rename them to include the station name, and copy them into a clean output folder:

```julia
# --- Find, rename, and copy ---
output_dir = "collected_data"
mkpath(output_dir)

for (root, dirs, files) in walkdir("fake_survey")
    for f in files
        if endswith(f, ".dat")
            # The station name is the immediate parent folder
            station = basename(root)

            # Build a new name: EM01_meas_001.dat
            new_name = station * "_" * f

            src = joinpath(root, f)
            dst = joinpath(output_dir, new_name)

            cp(src, dst, force = true)   # copy; force=true overwrites if exists
        end
    end
end

# Check the result
println("Files in $(output_dir):")
for f in sort(readdir(output_dir))
    println("  ", f)
end
```

```
Files in collected_data:
  EM01_meas_001.dat
  EM01_meas_002.dat
  EM01_meas_003.dat
  EM02_meas_001.dat
```

```
EM02_meas_002.dat
EM02_meas_003.dat
EM03_meas_001.dat
EM03_meas_002.dat
EM03_meas_003.dat
```

2.5.5 Key functions at a glance

Function	What it does
`readdir(dir)`	List contents of a directory
`readdir(dir, join=true)`	Same, but returns full paths
`walkdir(dir)`	Recursively walk through all subdirectories
`mkpath(dir)`	Create a directory (and parents), like `mkdir -p`
`cp(src, dst)`	Copy a file
`mv(src, dst)`	Move or rename a file
`rm(path)`	Delete a file
`rm(path, recursive=true)`	Delete a directory and everything inside it
`isfile(path)`	Check if a path points to an existing file
`isdir(path)`	Check if a path points to an existing directory
`joinpath(parts...)`	Build a path from pieces (cross-platform)
`basename(path)`	Filename from a path
`dirname(path)`	Directory from a path
`splitext(name)`	Split into `(name, ".ext")`
`splitpath(path)`	Split into a vector of path components

2.6 Geoscientific file formats

Geoscience uses several specialised binary formats to store large, multidimensional datasets efficiently. Julia has packages for all the common ones. Here is a brief overview. Detailed usage will be added in future updates of this chapter.

2.6.1 NetCDF

NetCDF (Network Common Data Form) is the standard format for climate, ocean, and atmospheric data. Use the **NCDatasets.jl** package:

```julia
using NCDatasets

# Read a NetCDF file
ds = NCDataset("temperature.nc")
temp = ds["temperature"][:, :, :]    # read the full 3D array
lat  = ds["latitude"][:]
lon  = ds["longitude"][:]
```

```
close(ds)

# Or use the do-block pattern (auto-closes)
NCDataset("temperature.nc") do ds
    temp = ds["temperature"][:, :, :]
    println("Shape: ", size(temp))
end
```

```
# Write a NetCDF file
NCDataset("output.nc", "c") do ds
    defDim(ds, "x", 10)
    defDim(ds, "y", 20)
    v = defVar(ds, "elevation", Float64, ("x", "y"))
    v[:, :] = rand(10, 20)
end
```

2.6.2 HDF5

HDF5 (Hierarchical Data Format) is widely used in geophysics, remote sensing, and seismology. Use **HDF5.jl**:

```
using HDF5

# Write
h5open("model.h5", "w") do f
    f["resistivity"] = rand(100, 100, 50)
    f["depth"] = collect(0.0:0.5:24.5)
end

# Read
h5open("model.h5", "r") do f
    rho   = read(f, "resistivity")
    depth = read(f, "depth")
    println("Model size: ", size(rho))
end
```

2.6.3 SEG-Y

SEG-Y is the seismic industry standard. Use **SegyIO.jl**:

```
using SegyIO

# Read a SEG-Y file
```

```julia
block = segy_read("seismic_line.segy")
traces = block.data           # matrix of traces
headers = block.traceheaders   # trace header information
```

2.6.4 Shapefiles and GeoJSON

For vector geospatial data (points, polygons, map boundaries), use **Shapefile.jl** or **GeoJSON.jl**:

```julia
using Shapefile

table = Shapefile.Table("geology_map.shp")
# Access as a DataFrame
using DataFrames
df = DataFrame(table)
```

> ⚠ These examples are not executed
>
> The geoscientific format examples above use `eval: false` style. They show the code patterns but don't run here because they require actual data files. When you have your own NetCDF, HDF5, or SEG-Y files, copy the pattern and replace the file name.

> ❗ Want more formats or deeper coverage?
>
> This section will grow as the book develops. If there is a specific geoscience format or workflow you need (GRIB, GeoTIFF, ASDF, miniSEED, EDI, ModEM, or anything else) please open an issue on GitHub and describe your use case. Your request will help us prioritise what to add next.

2.7 Clean up

Let's remove the temporary files we created in this chapter:

```julia
for f in ["stations.txt", "gravity.txt", "gravity_corrected.txt", "samples.csv"]
    isfile(f) && rm(f)
end
println("Cleaned up.")
```

```
Cleaned up.
```

3 Plotting & Data Visualisation

At this point you know how to write Julia code, work with functions and data structures, and read data from files into arrays and DataFrames.

In this chapter we will learn how to **visualise** that data. We start from the very basics, a single line on a pair of axes, and build up to the kinds of plots geoscientists use every day: scatter plots with error bars, borehole logs, contour maps, heatmaps, multi-panel figures, and more. By the end you will be able to turn any dataset into a publication-ready figure and save it as PNG, PDF, or SVG.

3.1 The Makie ecosystem

Julia has several plotting libraries. In this book we use **Makie**, a modern, high-performance plotting ecosystem designed for scientific visualisation. Makie is split into **backends**. You write the same plotting code, but choose a backend depending on where you want the output:

Backend	Package	Best for
CairoMakie	`CairoMakie.jl`	Static figures (PNG, PDF, SVG) for papers, reports, this book
GLMakie	`GLMakie.jl`	Interactive, GPU-accelerated windows for rotating 3-D models, exploring large datasets on your screen
WGLMakie	`WGLMakie.jl`	Interactive plots in a web browser or Jupyter/Pluto notebook

> **!** CairoMakie vs GLMakie: which one do I use?
>
> Think of Makie as one plotting language with different "printers":
>
> - **CairoMakie** renders to a file (PNG, PDF, SVG). There is no window. You call `save("fig.pdf", fig)` and get a crisp vector graphic ready for a journal. It works everywhere, including headless servers and HPC clusters with no display. This is what we use throughout this book.
> - **GLMakie** opens an **interactive window** on your screen, powered by your GPU. You can pan, zoom, and rotate 3-D scenes with the mouse, perfect for exploring a velocity model, a point cloud, or a seismicity catalogue before you decide which view to export. You need a display and a working OpenGL driver.
> - **WGLMakie** does the same interactive thing, but inside a **web browser**. It is the best choice for Jupyter or Pluto notebooks.

The key point: **your plotting code does not change**. You just swap `using CairoMakie` for `using GLMakie` (or `using WGLMakie`). So learn the API once, and pick the backend that fits the task.

For this book we use **CairoMakie** because it produces crisp vector graphics and does not need a GPU. When you start exploring 3-D geological models or large point clouds interactively, switch to **GLMakie**. The plotting code stays the same, only the `using` line changes.

What about Plots.jl and Plotly?

Plots.jl is another popular Julia plotting package with a different API. It supports multiple backends (GR, Plotly, PGFPlotsX, etc.) through a single interface. If you prefer Plotly-style interactive HTML plots, you can use `Plots.jl` with the `plotlyjs()` backend, or use **PlotlyJS.jl** directly.
We chose Makie for this book because its composability (building complex figures from simple pieces) and performance with large datasets make it a natural fit for geoscientific work. But there is no wrong choice — explore both and use what feels right for your workflow.

GMT.jl: the Generic Mapping Tools for Julia

If you work with geographic or bathymetric data, you should know about **GMT.jl**, a Julia wrapper around the legendary Generic Mapping Tools (GMT). GMT has been the gold standard for map-making in the earth sciences for over 30 years. The Julia wrapper gives you the full power of GMT (coastlines, topographic relief, geodetic projections, gravity grids, focal mechanisms) from within Julia.
GMT.jl is not covered in this chapter because it has its own extensive documentation and a different API from Makie. But if your work involves publication-quality maps with coastlines, plate boundaries, or bathymetry, it is absolutely worth exploring:
https://www.generic-mapping-tools.org/GMT.jl/stable/

3.2 Installing CairoMakie

Before we can plot anything, we need to add CairoMakie to our project environment. Press `]` in the REPL to enter Pkg mode:

```
pkg> add CairoMakie
```

Or from normal Julia code:

```julia
using Pkg; Pkg.add("CairoMakie")
```

You only need to do this once per environment. CairoMakie is already in this book's `Project.toml`, so if you cloned the book repository and ran `Pkg.instantiate()`, you are all set.

First load is slow, that is normal

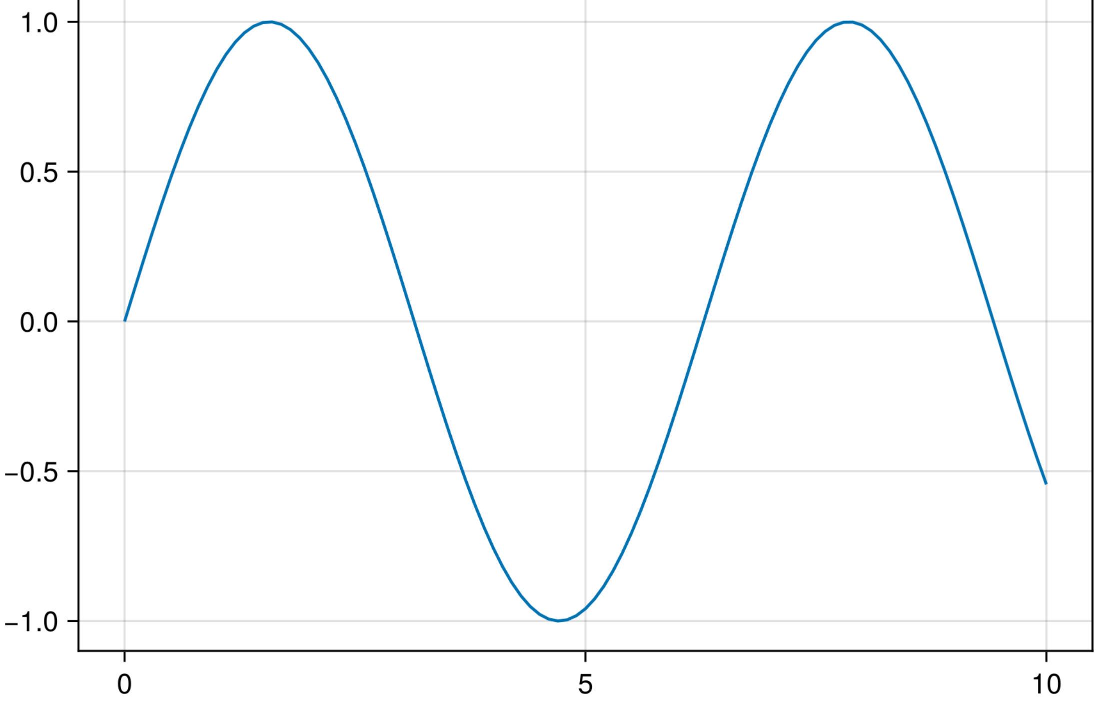
The very first time you run `using CairoMakie` in a fresh Julia session, it takes a while (sometimes 20 to 40 seconds) because Julia compiles the package. Subsequent calls in the same session are instant. This is Julia's "time to first plot" (TTFP). It has improved enormously over recent versions and continues to get faster.

```
using CairoMakie
CairoMakie.activate!(type = "png")
```

3.3 Your first plot

Let's start with the simplest possible plot, a line:

```
x = 0:0.1:10
y = sin.(x)

lines(x, y)
```

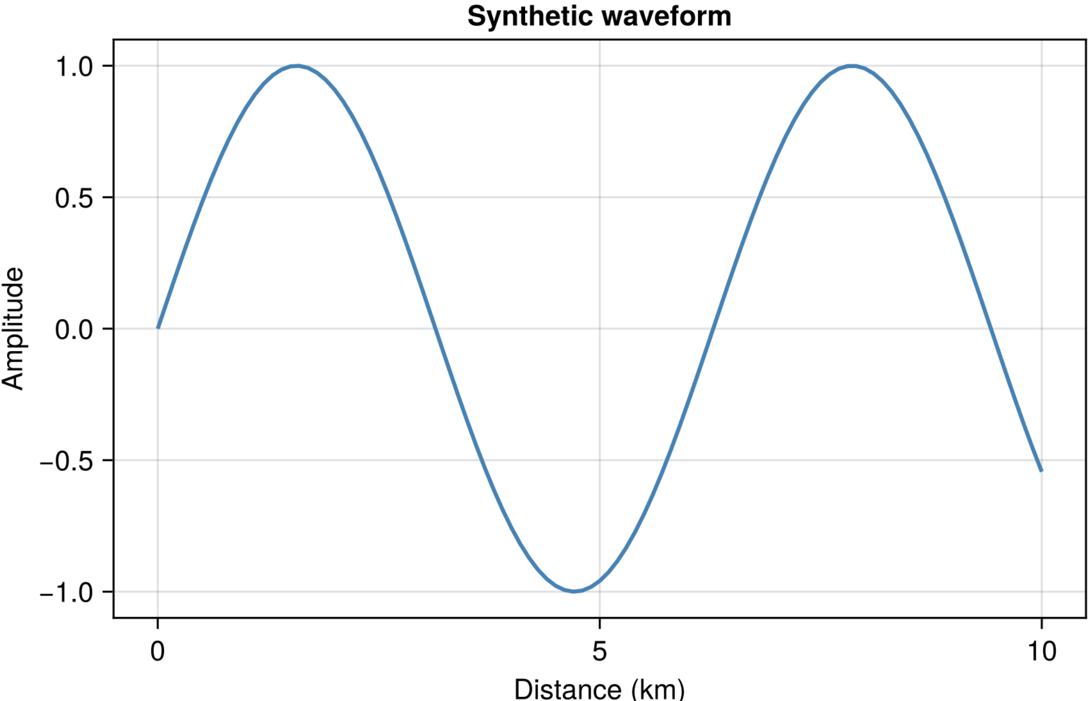


That is it. `lines(x, y)` creates a figure, adds an axis, and draws a line through the points. Makie returns a `FigureAxisPlot` object that renders automatically.

3.3.1 Adding labels and a title

A plot without labels is just a pretty picture. Always label your axes:

```
fig, ax, plt = lines(x, sin.(x),
    axis = (
        xlabel = "Distance (km)",
        ylabel = "Amplitude",
        title  = "Synthetic waveform"
    ),
    linewidth = 2,
    color = :steelblue
)
fig
```

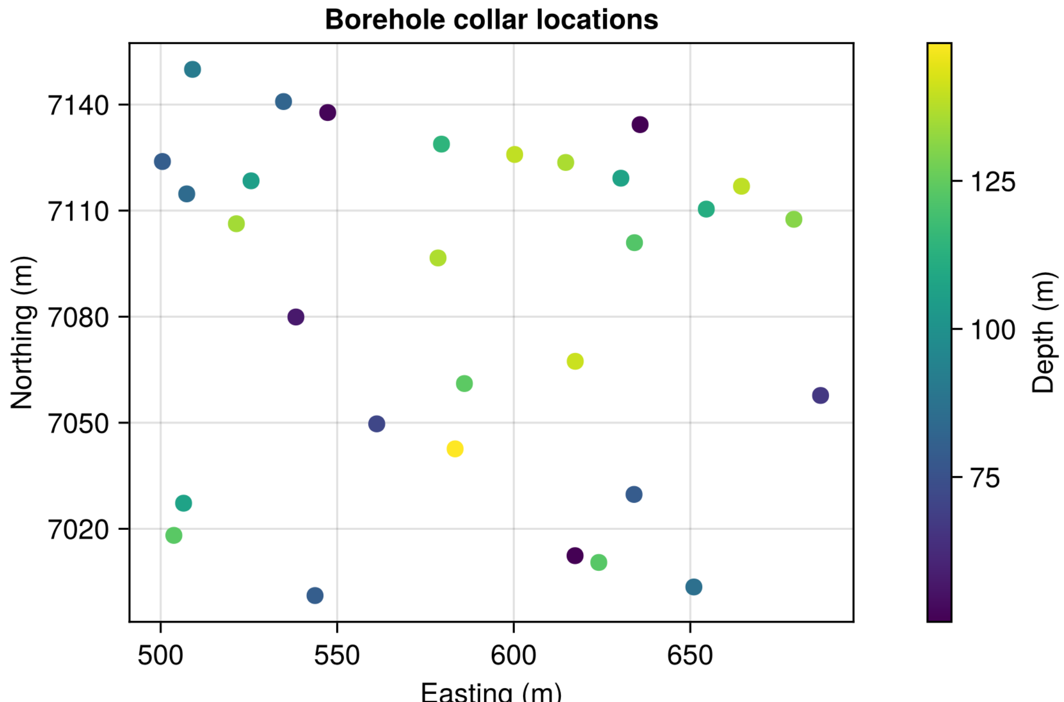


The `fig, ax, plt = ...` pattern gives you handles to the **figure**, the **axis**, and the **plot object**. You will use these handles a lot when building more complex figures.

3.4 Scatter plots

Scatter plots are the workhorse of geoscience: plotting measurement locations, geochemical concentrations, geophysical anomalies, or any point data.

```
# Synthetic borehole collar locations
n = 30
easting  = 500 .+ 200 .* rand(n)
northing = 7000 .+ 150 .* rand(n)
depth    = 50 .+ 100 .* rand(n)

fig, ax, plt = scatter(easting, northing,
    color = depth,
    colormap = :viridis,
    markersize = 12,
    axis = (
        xlabel = "Easting (m)",
        ylabel = "Northing (m)",
        title  = "Borehole collar locations",
        aspect = DataAspect()
    )
)
Colorbar(fig[1, 2], plt, label = "Depth (m)")
fig
```



A few things to notice:

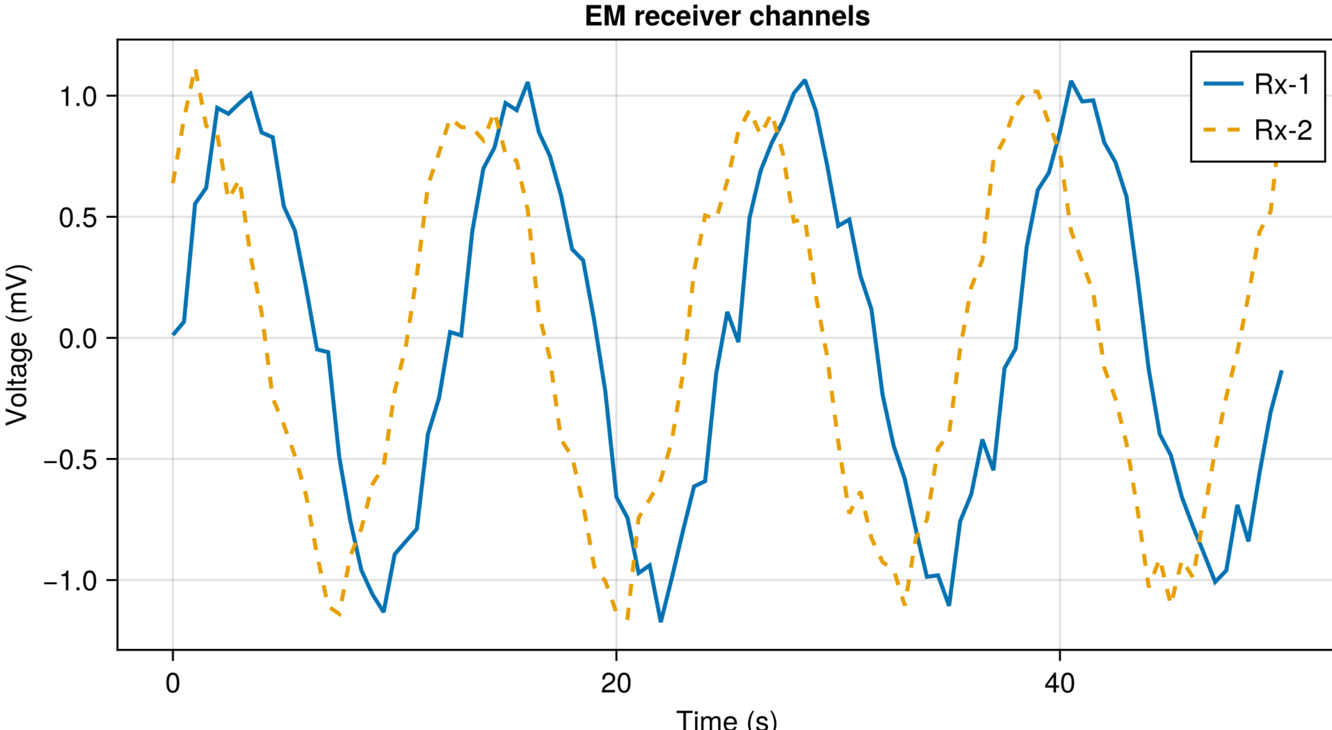
- **`color = depth`** maps a data vector to colour, just like you would colour-code sample values on a field map.

- **`colormap = :viridis`** picks a perceptually uniform colourmap. Other good choices for geoscience: `:turbo`, `:RdBu` (diverging, great for anomalies), `:terrain`, `:deep`.
- **`Colorbar(...)`** adds a colour legend. The position `fig[1, 2]` means "row 1, column 2 of the figure layout."
- **`aspect = DataAspect()`** keeps the x and y scales equal — essential when plotting spatial coordinates.

3.5 Line plots with multiple series

Geoscientists often compare several datasets on the same axes: different field campaigns, model predictions versus observations, or time series from multiple stations.

```
time = 0:0.5:50             # time in seconds
signal_1 = sin.(0.5 .* time) .+ 0.1 .* randn(length(time))
signal_2 = sin.(0.5 .* time .+ π/3) .+ 0.1 .* randn(length(time))

fig = Figure(size = (700, 400))
ax = Axis(fig[1, 1],
    xlabel = "Time (s)",
    ylabel = "Voltage (mV)",
    title  = "EM receiver channels"
)
lines!(ax, time, signal_1, label = "Rx-1", linewidth = 2)
lines!(ax, time, signal_2, label = "Rx-2", linewidth = 2, linestyle = :dash)
axislegend(ax, position = :rt)
fig
```

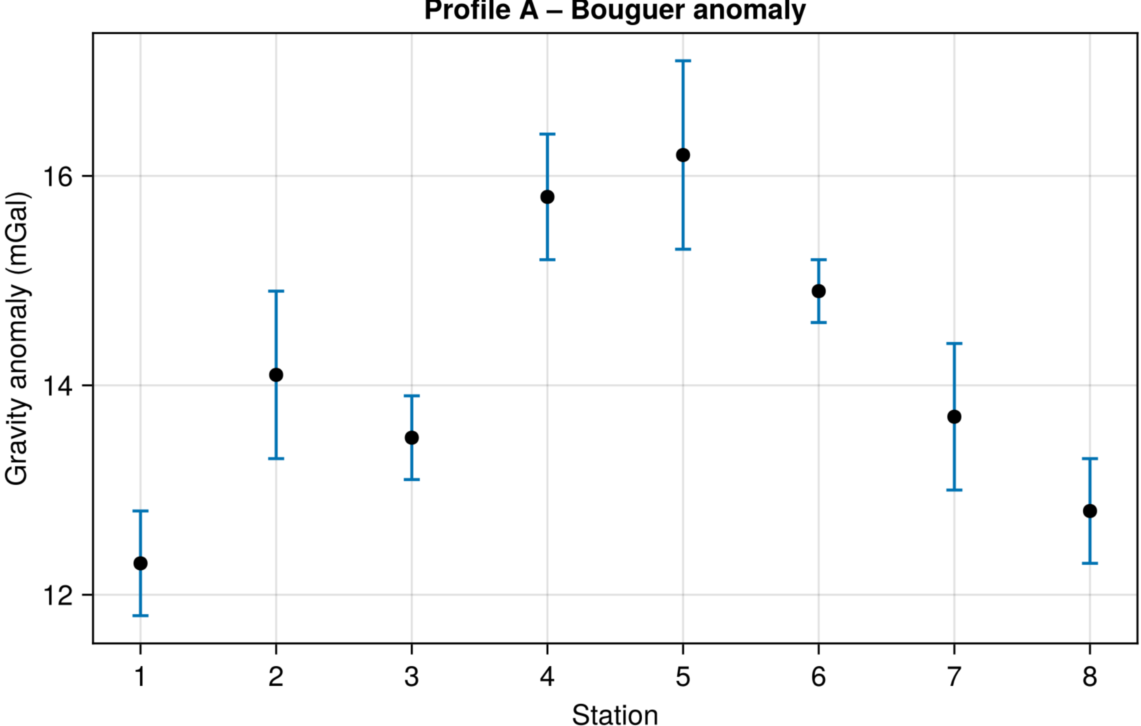


Notice the `!` in `lines!` and `axislegend!`-style calls. The exclamation mark means "add to an existing axis" rather than creating a new one. This is a consistent Makie convention.

3.6 Error bars

Field measurements always have uncertainty. Use `errorbars!` to show it:

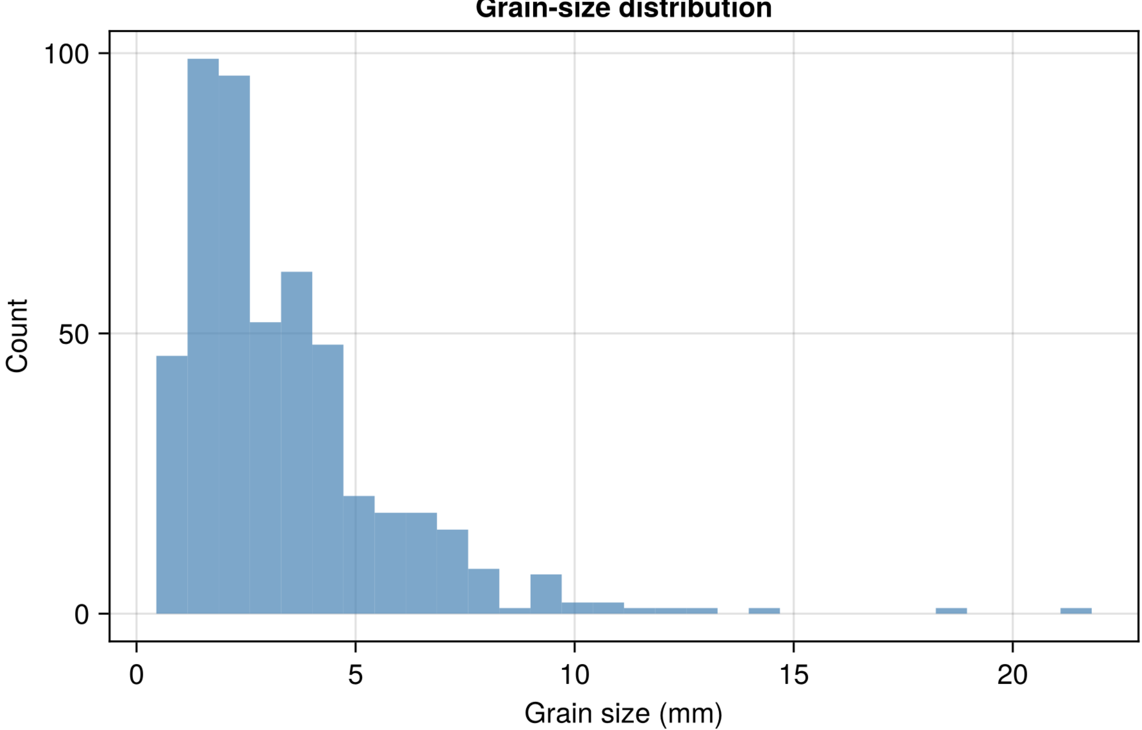
```
stations = 1:8
gravity  = [12.3, 14.1, 13.5, 15.8, 16.2, 14.9, 13.7, 12.8]
uncert   = [0.5,  0.8,  0.4,  0.6,  0.9,  0.3,  0.7,  0.5]

fig = Figure(size = (600, 400))
ax = Axis(fig[1, 1],
    xlabel = "Station",
    ylabel = "Gravity anomaly (mGal)",
    title  = "Profile A – Bouguer anomaly",
    xticks = 1:8
)
errorbars!(ax, stations, gravity, uncert, whiskerwidth = 8)
scatter!(ax, stations, gravity, markersize = 10, color = :black)
fig
```



3.7 Bar charts and histograms

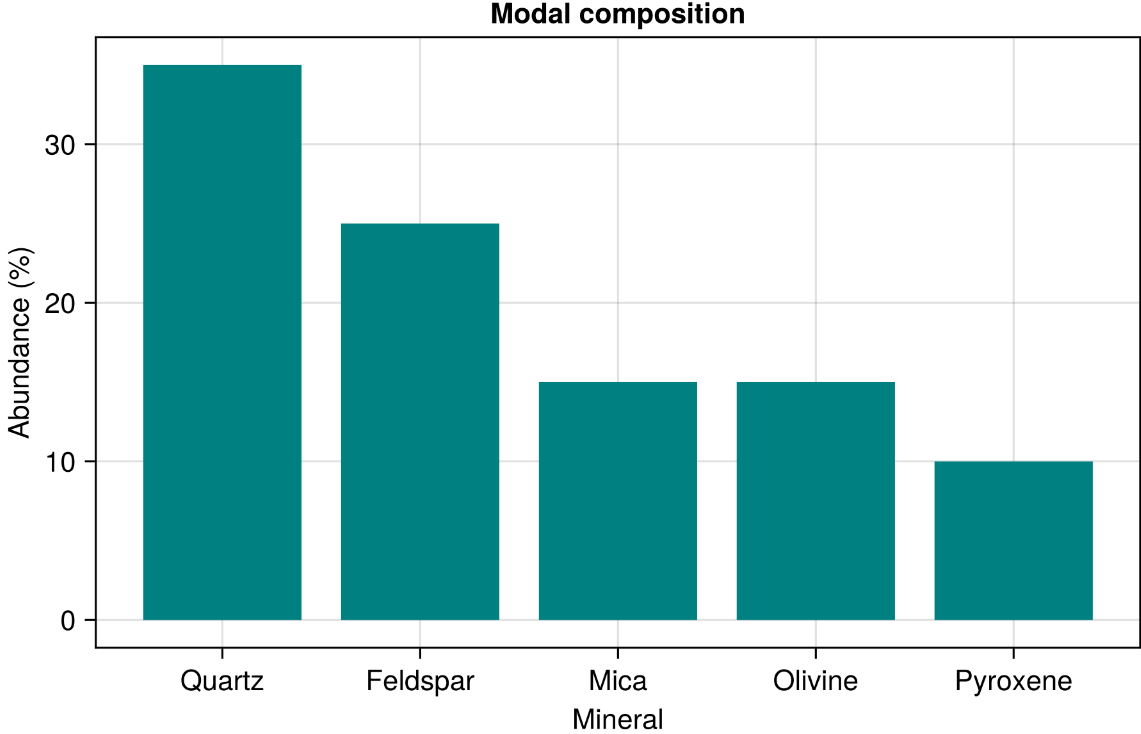
3.7.1 Histogram of sample data

```
# Simulated grain sizes (log-normal distribution)
grain_sizes = exp.(randn(500) .* 0.6 .+ 1.0)

fig = Figure(size = (600, 400))
ax = Axis(fig[1, 1],
    xlabel = "Grain size (mm)",
    ylabel = "Count",
    title  = "Grain-size distribution"
)
hist!(ax, grain_sizes, bins = 30, color = (:steelblue, 0.7))
fig
```



3.7.2 Bar chart

```
minerals  = ["Quartz", "Feldspar", "Mica", "Olivine", "Pyroxene"]
abundance = [35, 25, 15, 15, 10]

fig = Figure(size = (600, 400))
ax = Axis(fig[1, 1],
    xlabel = "Mineral",
    ylabel = "Abundance (%)",
    title  = "Modal composition",
    xticks = (1:5, minerals)
)
barplot!(ax, 1:5, abundance, color = :teal)
fig
```



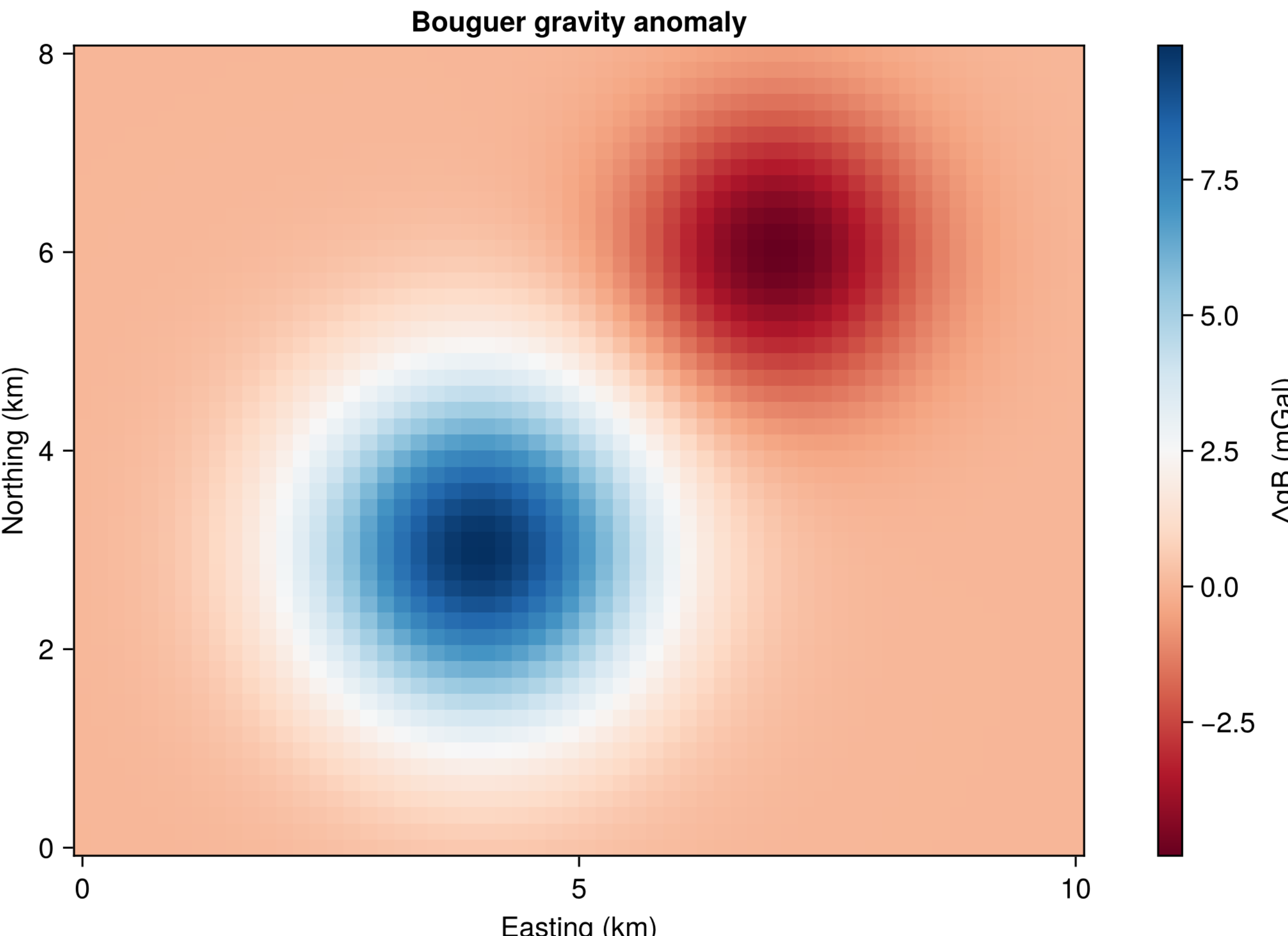
3.8 Heatmaps and contour plots

Gridded data is everywhere in geoscience: gravity grids, magnetic surveys, temperature fields, elevation models. Makie handles these with `heatmap` and `contour`/`contourf`.

3.8.1 Heatmap

```
# Synthetic gravity anomaly on a 2-D grid
x_grid = range(0, 10, length = 60)
y_grid = range(0, 8,  length = 50)
gz = [10 * exp(-((xi - 4)^2 + (yi - 3)^2) / 3) -
       5 * exp(-((xi - 7)^2 + (yi - 6)^2) / 2)
      for xi in x_grid, yi in y_grid]

fig = Figure(size = (700, 500))
ax = Axis(fig[1, 1],
    xlabel = "Easting (km)",
    ylabel = "Northing (km)",
    title  = "Bouguer gravity anomaly",
    aspect = DataAspect()
)
```

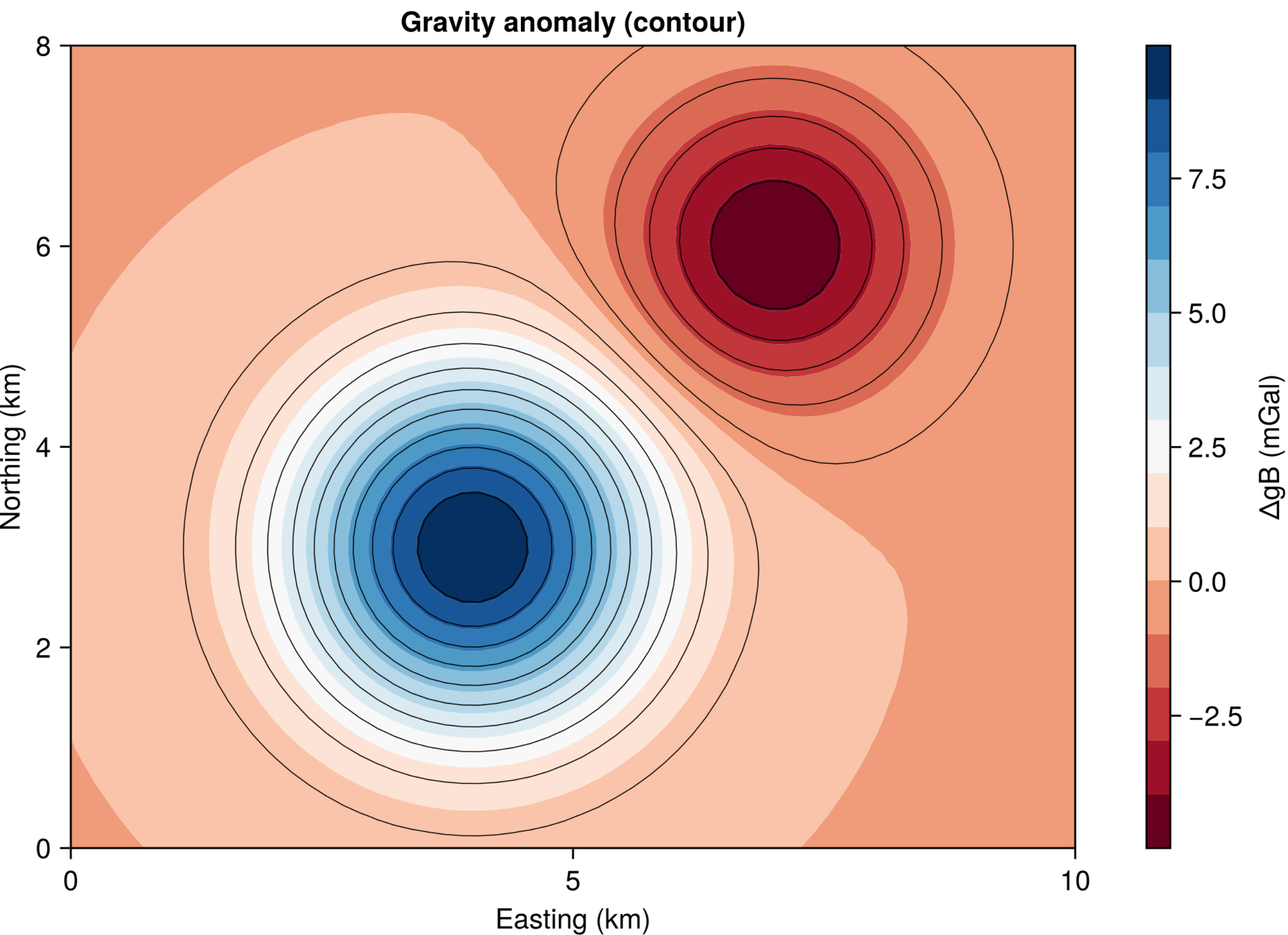
```
hm = heatmap!(ax, x_grid, y_grid, gz, colormap = :RdBu)
Colorbar(fig[1, 2], hm, label = "ΔgB (mGal)")
fig
```



3.8.2 Filled contour plot

```
fig = Figure(size = (700, 500))
ax = Axis(fig[1, 1],
    xlabel = "Easting (km)",
    ylabel = "Northing (km)",
    title  = "Gravity anomaly (contour)",
    aspect = DataAspect()
)
co = contourf!(ax, x_grid, y_grid, gz, levels = 15, colormap = :RdBu)
contour!(ax, x_grid, y_grid, gz, levels = 15, color = :black, linewidth = 0.5)
Colorbar(fig[1, 2], co, label = "ΔgB (mGal)")
```

```
fig
```



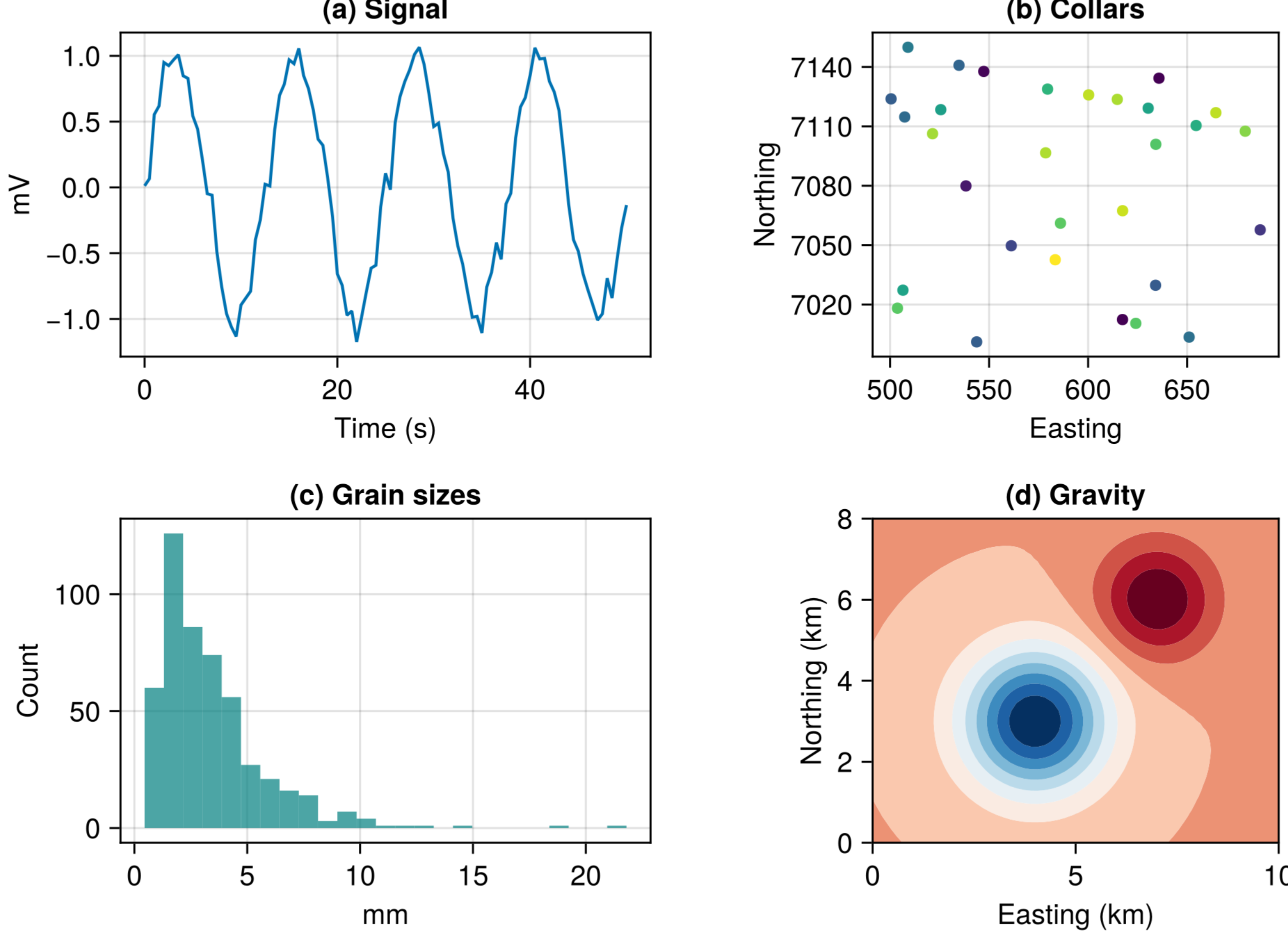
Overlaying `contour!` lines on a `contourf!` fill is a classic geophysics style. The thin black lines make it easier to read precise values.

3.9 Multi-panel figures (subplots)

Scientific papers often require several panels in one figure. Makie uses a **grid layout** system: `fig[row, col]` places content anywhere on the grid.

```
fig = Figure(size = (700, 500))

# Panel a — line plot
ax1 = Axis(fig[1, 1], title = "(a) Signal", xlabel = "Time (s)", ylabel = "mV")
lines!(ax1, time, signal_1)
```

```
# Panel b — scatter
ax2 = Axis(fig[1, 2], title = "(b) Collars",
    xlabel = "Easting", ylabel = "Northing", aspect = DataAspect())
scatter!(ax2, easting, northing, color = depth, colormap = :viridis, markersize = 8)

# Panel c — histogram
ax3 = Axis(fig[2, 1], title = "(c) Grain sizes", xlabel = "mm", ylabel = "Count")
hist!(ax3, grain_sizes, bins = 25, color = (:teal, 0.7))

# Panel d — contour
ax4 = Axis(fig[2, 2], title = "(d) Gravity", xlabel = "Easting (km)",
    ylabel = "Northing (km)", aspect = DataAspect())
contourf!(ax4, x_grid, y_grid, gz, levels = 12, colormap = :RdBu)

fig
```



You can span rows or columns for insets and wide panels — see the Makie layout tutorial for advanced arrangements.

3.10 Customising appearance

3.10.1 Fonts and themes

Makie ships with several built-in themes. You can also override individual settings:

```
with_theme(theme_minimal()) do
    fig = Figure(size = (600, 400))
    ax = Axis(fig[1, 1],
        xlabel = "Frequency (Hz)",
        ylabel = "Power spectral density",
        title  = "Minimal theme example",
        xscale = log10,
        yscale = log10
    )
    freqs = 10 .^ range(-1, 3, length = 200)
    psd   = 1 ./ freqs .+ 0.01 .* abs.(randn(200))
    lines!(ax, freqs, psd, linewidth = 2)
    fig
end
```

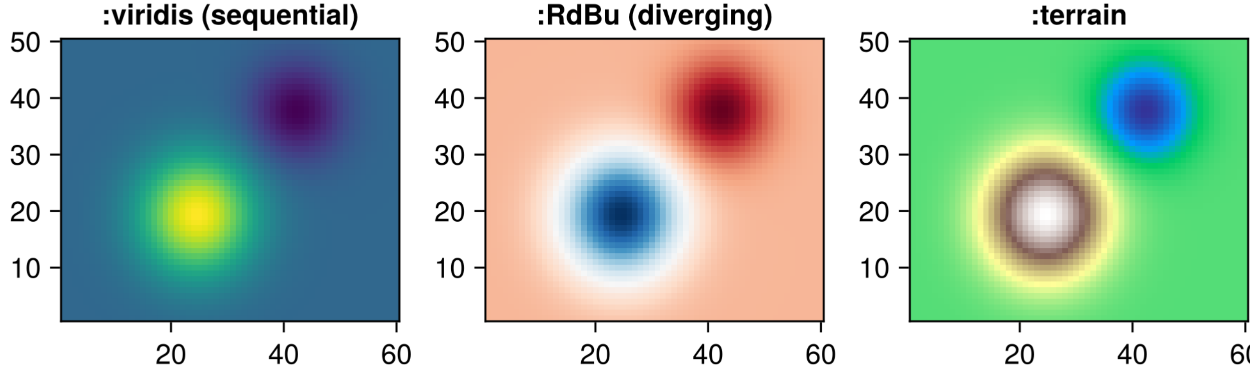
Other themes to try: `theme_dark()`, `theme_light()`, `theme_black()`, `theme_ggplot2()`.

3.10.2 Colourmaps

Choosing the right colourmap matters, especially in geoscience where a bad colourmap can hide features or mislead the reader. Some guidelines:

Data type	Good choices	Avoid
Sequential (elevation, depth, concentration)	`:viridis`, `:cividis`, `:deep`, `:thermal`	`:jet` (perceptual jumps)
Diverging (anomalies centred on zero)	`:RdBu`, `:BrBG`, `:PiYG`	Any sequential map
Categorical (lithology, fault type)	`:tab10`, `:Set1`	Continuous maps
Terrain / bathymetry	`:terrain`, `:topo`	`:hot`, `:gray`

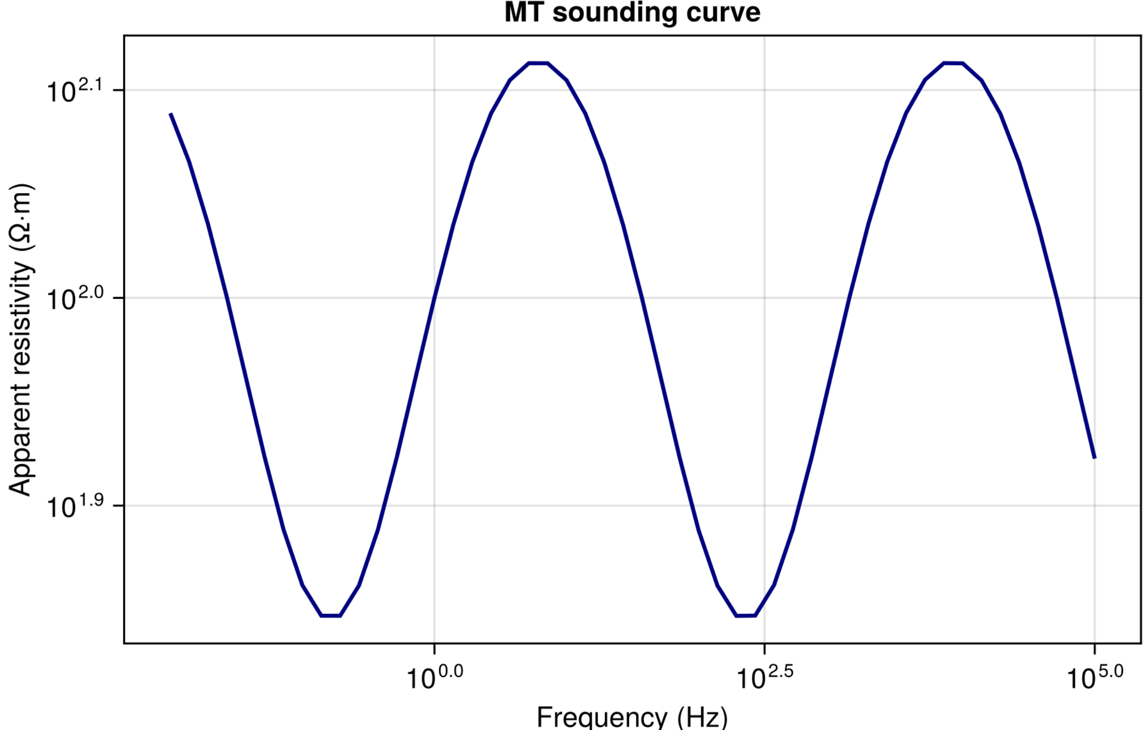
```
fig = Figure(size = (650, 280))

ax1 = Axis(fig[1, 1], title = ":viridis (sequential)", aspect = DataAspect())
heatmap!(ax1, gz, colormap = :viridis)

ax2 = Axis(fig[1, 2], title = ":RdBu (diverging)", aspect = DataAspect())
heatmap!(ax2, gz, colormap = :RdBu)

ax3 = Axis(fig[1, 3], title = ":terrain", aspect = DataAspect())
heatmap!(ax3, gz, colormap = :terrain)

fig
```



3.11 Log-scale axes

Geophysical data often spans many orders of magnitude: resistivity, frequency, spectral power. Use `xscale = log10` or `yscale = log10`:

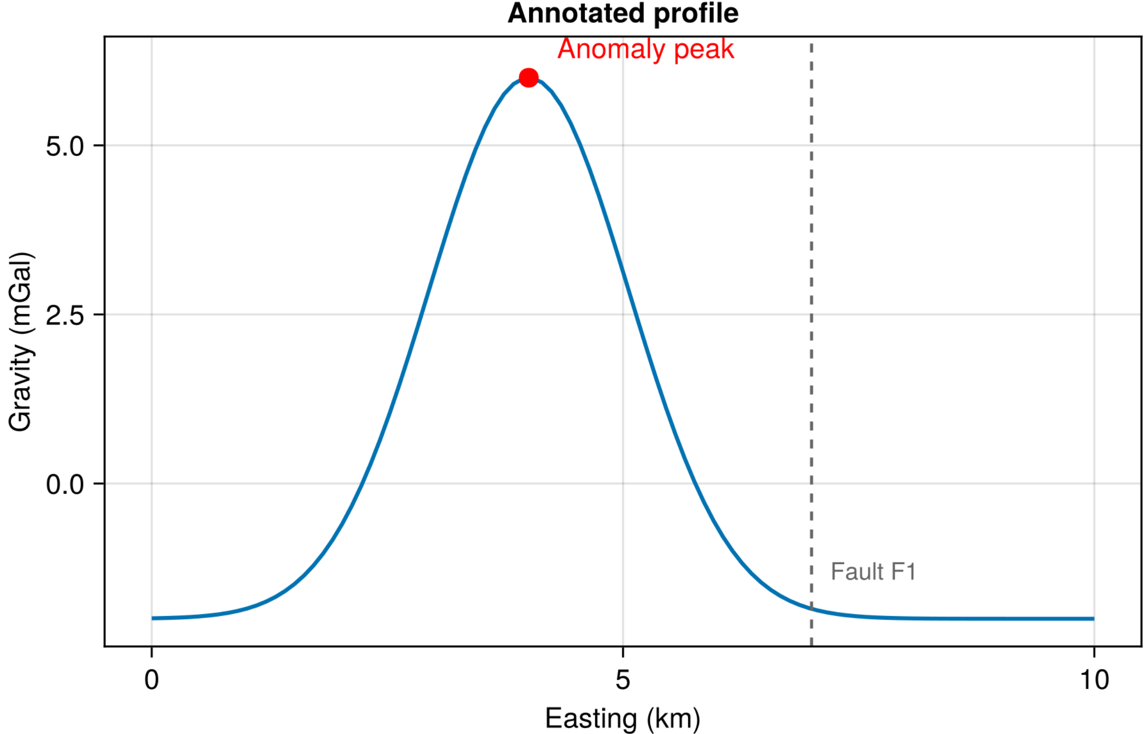
```
freq = 10 .^ range(-2, 5, length = 50)
apparent_res = 100 .* (1 .+ 0.3 .* sin.(log10.(freq) .* 2))

fig = Figure(size = (600, 400))
ax = Axis(fig[1, 1],
    xlabel = "Frequency (Hz)",
    ylabel = "Apparent resistivity (Ω·m)",
    title  = "MT sounding curve",
    xscale = log10,
    yscale = log10
)
lines!(ax, freq, apparent_res, linewidth = 2, color = :navy)
fig
```



3.12 Annotations and markers

Labelling key features on a plot (a fault, an anomaly peak, a sample location) is something you will do often:

```
fig = Figure(size = (600, 400))
ax = Axis(fig[1, 1],
    xlabel = "Easting (km)", ylabel = "Gravity (mGal)",
    title  = "Annotated profile"
)
profile_x = range(0, 10, length = 100)
profile_g = 8 .* exp.(-((profile_x .- 4) ./ 1.5).^2) .- 2
lines!(ax, profile_x, profile_g, linewidth = 2)

# Mark the peak
scatter!(ax, [4.0], [6.0], markersize = 14, color = :red)
text!(ax, 4.3, 6.2, text = "Anomaly peak", fontsize = 14, color = :red)

# Mark a fault
vlines!(ax, [7.0], color = :grey40, linestyle = :dash, linewidth = 1.5)
text!(ax, 7.2, -1.5, text = "Fault F1", fontsize = 12, color = :grey40)

fig
```



3.13 Saving figures

Use `save` to export your figure. The file extension determines the format:

```
save("gravity_profile.png", fig)              # raster (300 dpi by default)
save("gravity_profile.pdf", fig)              # vector, best for papers
save("gravity_profile.svg", fig)              # vector, good for web
save("gravity_profile.png", fig, px_per_unit = 3)  # high-res raster (900 dpi)
```

PDF for papers, PNG for presentations

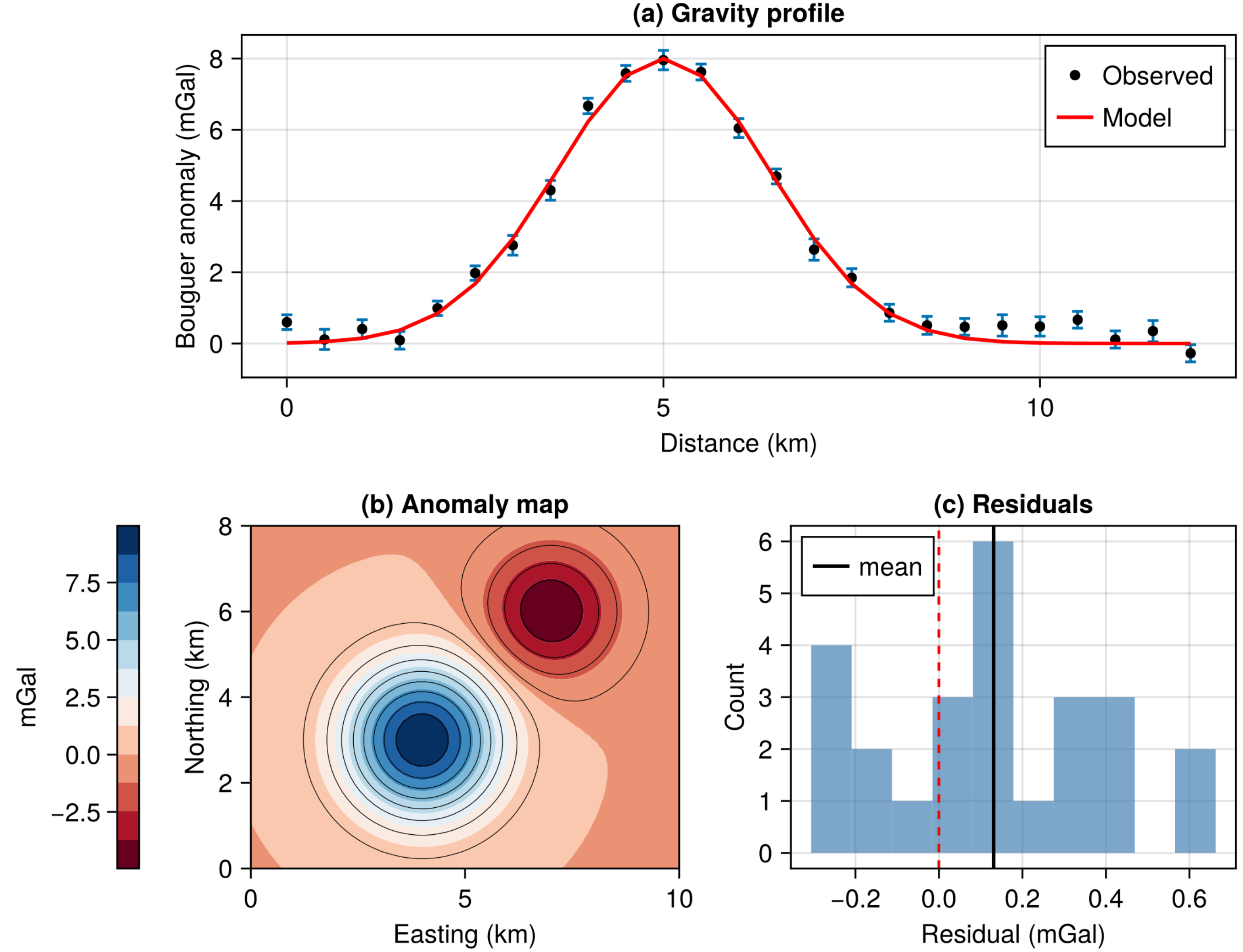
For journal submissions, **PDF** or **SVG** gives you lossless vector graphics that look sharp at any zoom. For slides and posters, **PNG at high resolution** (`px_per_unit = 3` or higher) is more portable.

3.14 Putting it together: a geoscience figure

Let's combine several techniques into a single multi-panel figure typical of a geophysical report:

```
using Statistics

fig = Figure(size = (700, 550))

# --- Panel a: profile with error bars ---
ax1 = Axis(fig[1, 1:2], title = "(a) Gravity profile",
    xlabel = "Distance (km)", ylabel = "Bouguer anomaly (mGal)")
dist   = range(0, 12, length = 25)
obs    = 8 .* exp.(-((dist .- 5) ./ 2).^2) .+ 0.3 .* randn(25)
uncert = 0.2 .+ 0.1 .* rand(25)
errorbars!(ax1, collect(dist), obs, uncert, whiskerwidth = 6)
scatter!(ax1, collect(dist), obs, markersize = 8, color = :black, label = "Observed")
model_g = 8 .* exp.(-((dist .- 5) ./ 2).^2)
lines!(ax1, collect(dist), model_g, linewidth = 2, color = :red, label = "Model")
axislegend(ax1, position = :rt)

# --- Panel b: contour map ---
ax2 = Axis(fig[2, 1], title = "(b) Anomaly map",
    xlabel = "Easting (km)", ylabel = "Northing (km)",
    aspect = DataAspect())
co = contourf!(ax2, x_grid, y_grid, gz, levels = 12, colormap = :RdBu)
contour!(ax2, x_grid, y_grid, gz, levels = 12, color = :black, linewidth = 0.4)
Colorbar(fig[2, 0], co, label = "mGal", flipaxis = false)

# --- Panel c: histogram of residuals ---
residuals = obs .- model_g
```

```
ax3 = Axis(fig[2, 2], title = "(c) Residuals",
    xlabel = "Residual (mGal)", ylabel = "Count")
hist!(ax3, residuals, bins = 10, color = (:steelblue, 0.7))
vlines!(ax3, [0.0], color = :red, linestyle = :dash)
vlines!(ax3, [mean(residuals)], color = :black, linewidth = 2, label = "mean")
axislegend(ax3, position = :lt)

fig
```



3.15 Geographic maps with GeoMakie

So far every plot has used plain numeric axes — “Easting (km)” and “Northing (km).” But geoscientists often need to plot data on a **map** with coastlines, country borders, and a proper geographic projection. That is

what **GeoMakie.jl** gives you.

GeoMakie provides a `GeoAxis`, a drop-in replacement for Makie's `Axis` that understands coordinate reference systems (CRS) and projections. You feed it longitude/latitude data and tell it which projection you want; it handles the transformation and draws graticules automatically.

3.15.1 Installing GeoMakie

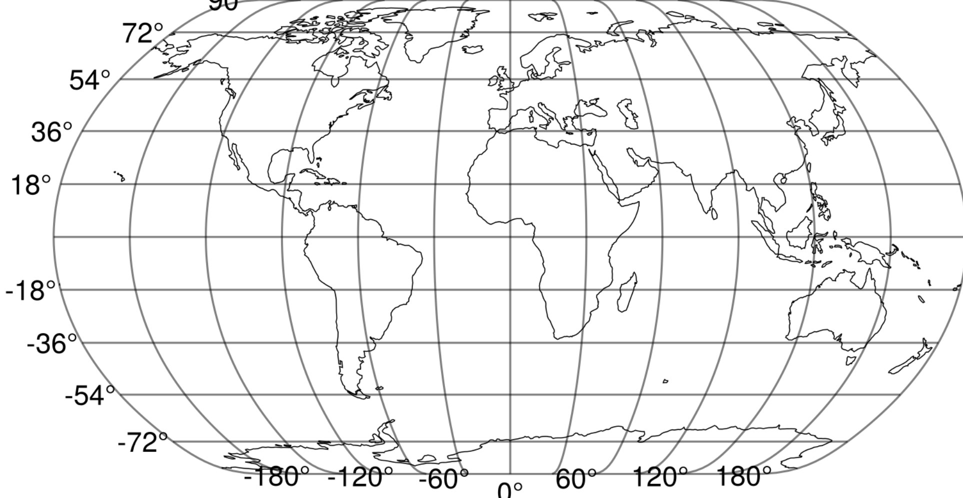
```
pkg> add GeoMakie
```

Or: `using Pkg; Pkg.add("GeoMakie")`. GeoMakie pulls in Proj.jl (for map projections) and NaturalEarth (for coastlines and country boundaries).

```julia
using GeoMakie
```

3.15.2 A world map in three lines

```julia
fig = Figure()
ga = GeoAxis(fig[1, 1], dest = "+proj=natearth", title = "Natural Earth projection")
lines!(ga, GeoMakie.coastlines(), color = :black, linewidth = 0.5)
fig
```



`GeoAxis` accepts any PROJ-string through the `dest` keyword. The default source CRS is WGS84 (longitude/latitude). Try replacing `"+proj=natearth"` with `"+proj=ortho"` for an orthographic globe view, or `"+proj=moll"` for a Mollweide equal-area projection.

3.15.3 Overlaying polygons and raster data

GeoMakie integrates with **GeoInterface.jl**, so you can plot any GeoInterface-compatible geometry: shapefiles, GeoJSON, geological unit polygons, fault traces, concession boundaries:

```julia
using GeoMakie, GeoJSON

# Load a GeoJSON of geological units (hypothetical file)
geol = GeoJSON.read("geology_units.geojson")

fig = Figure()
ga = GeoAxis(fig[1, 1], dest = "+proj=utm +zone=18 +south")
poly!(ga, geol, color = :lightgrey, strokecolor = :black, strokewidth = 0.5)
fig
```

Common projections for geoscience

Region / Use case	PROJ string	Notes
Global equal-area	`+proj=moll`	Mollweide, good for thematic world maps
Global for visuals	`+proj=natearth`	Natural Earth, pleasant for presentations
Globe view	`+proj=ortho +lon_0=0 +lat_0=30`	Orthographic, looks like a photo from space
Local field survey	`+proj=utm +zone=35 +datum=WGS84`	Metric grid, precise local coordinates
Continental	`+proj=lcc +lat_1=30 +lat_2=60 +lon_0=10`	Lambert Conformal Conic, preserves shape
Polar	`+proj=stere +lat_0=90 +lon_0=0`	Stereographic, Arctic/Antarctic maps

> You can browse all available projections at https://proj.org/operations/projections/.

3.15.4 Learn more about GeoMakie

- **Documentation & gallery**: https://geo.makie.org/
- **Source code**: https://github.com/MakieOrg/GeoMakie.jl

3.16 Where to learn more

- **Makie documentation**: https://docs.makie.org/ with tutorials, gallery, and the full API reference.
- **Beautiful Makie**: https://beautiful.makie.org/, a gallery of polished example figures you can copy and adapt.
- **Makie colour maps**: the `ColorSchemes.jl` package gives you access to hundreds of colour maps. Browse them at https://juliagraphics.github.io/ColorSchemes.jl/stable/.

Part II

Neural Networks

4 Neural Networks

At this point you know how to read files, work with arrays and tables, and make figures. That is enough to start asking a more interesting question: can we build a model that learns a pattern directly from data instead of us writing the rule by hand?

That is the role of a **neural network**. A neural network is just a layered function with many adjustable parameters. During training, those parameters are nudged until the network maps inputs to outputs in a useful way. The idea is simple. What makes neural networks powerful is that with enough data, enough nonlinearity, and a sensible training setup, they can approximate very complicated relationships.

For geoscience, this matters because many problems involve patterns that are real but hard to write down explicitly: seismic facies, weather evolution, surrogate models for PDE solvers, or relationships between sparse observations and hidden structure in the subsurface. Neural networks are not magic shortcuts around physics, but they are flexible function approximators that become extremely useful when combined with scientific knowledge.

4.1 The basic picture

At the smallest scale, a neuron takes an input vector $\mathbf{x}$, forms a weighted sum, adds a bias, and applies a nonlinear activation:

$$z = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

One neuron is not very interesting. A layer of neurons can learn several features at once, and a stack of layers can build increasingly abstract representations. In practice, training a network always comes back to the same loop:

1. Make a prediction from the input.
2. Compare that prediction with the target.
3. Measure the mismatch with a loss function.
4. Adjust the parameters to reduce the loss.

That is the whole game. The rest of this part is about understanding what each of those steps really means.

4.2 How this part is organized

The next chapter gives you a first end-to-end example before any serious theory. That is intentional. It is easier to understand the moving parts once you have seen the whole workflow in action.

After that, we slow down and unpack the pieces one by one:

- Your First Neural Network shows the full workflow: data, model, training, and prediction.
- Building Blocks of Neural Networks explains neurons, layers, losses, gradients, and optimizers.
- The remaining chapters first cover major architecture families used throughout modern machine learning: multilayer perceptrons, convolutional networks, recurrent networks, transformers, and graph neural networks. The last group of chapters then shifts to generative modeling: autoencoders, GANs, diffusion models, and flow matching.

The goal is not to turn you into a deep-learning theorist. The goal is to make the models legible enough that when they reappear later in scientific machine learning, they feel like tools you understand rather than black boxes.

4.3 Notation used in this part

To keep the mathematics readable, this part uses a few conventions consistently:

- Bold symbols such as $\mathbf{x}$ and $\mathbf{h}$ denote vectors.
- Superscripts in parentheses, such as $\mathbf{h}^{(l)}$, index layers or network depth.
- Subscripts, such as $\mathbf{h}_t$, usually index time steps, samples, or nodes.
- $\mathcal{L}$ denotes a loss function, while subscripts such as $\mathcal{L}_{\text{data}}$ or $\mathcal{L}_{\text{bc}}$ identify particular pieces of that loss.

These are not universal rules across all textbooks, but they will keep the notation in this book steady from chapter to chapter.

4.4 What to watch for

When you first learn neural networks, it is tempting to focus on architecture names and package syntax. Those matter, but the deeper questions are simpler:

- What is the input, and what is the output?
- What exactly is being learned?
- What loss is being minimized?
- What assumptions are hidden in the data split, the architecture, and the training loop?

If you keep asking those questions, most neural-network papers become much easier to read.

4.5 Summary

Neural networks are flexible parameterized functions trained from data. In geoscience they become especially useful when they are paired with domain structure, physical constraints, and careful evaluation. The next chapter starts with a concrete example so you can see the workflow before we dissect the machinery.

5 Your First Neural Network

Let's skip the theory and train a neural network end to end. You don't need to understand how it works yet. The goal of this chapter is to show you the *feel* of a complete machine-learning workflow: load data, build a model, train it, and use it to predict something. Everything you see here will be explained properly in the chapters that follow.

5.1 Sauna Satisfaction Predictor

In an alternate universe, the Geological Survey of Finland maintains sauna visitor logs. For each visit, three inputs are recorded:

- **Sauna temperature** (°C)
- **Outside temperature** (°C)
- **Minutes since last coffee**

After the session, the visitor assigns a **Satisfaction Score** between 0 and 1. We will train a small neural network to predict this score from the three inputs.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf
```

5.1.1 Step 1: Load and prepare the data

Read the data from a CSV file and normalize the inputs to zero mean and unit variance. This is standard practice: neural networks learn faster when inputs are on similar scales.

```
lines = readlines("sauna_log.txt")
data = reduce(hcat, [parse.(Float32, split(l, ",")) for l in lines[2:end]])

X_raw = data[1:3, :]
Y     = data[4:4, :]

# Normalize inputs
mu  = mean(X_raw, dims = 2)
sig = std(X_raw, dims = 2)
X   = (X_raw .- mu) ./ sig

# Split into train and test
rng = Xoshiro(123)
```

```
n_train = Int(round(0.8 * size(X, 2)))
idx = randperm(rng, size(X, 2))

X_train = X[:, idx[1:n_train]]
Y_train = Y[:, idx[1:n_train]]
X_test  = X[:, idx[n_train+1:end]]
Y_test  = Y[:, idx[n_train+1:end]]

@printf "Train: %d  Test: %d\n" size(X_train, 2) size(X_test, 2)
```

```
Train: 160  Test: 40
```

5.1.2 Step 2: Build the model

Define a small neural network with one hidden layer of 8 neurons. Don't worry about what `Dense`, `tanh`, or `Chain` mean yet—we will cover all of that shortly.

```
model = Chain(
    Dense(3 => 8, tanh),
    Dense(8 => 1)
)

# Initialize parameters and state
ps, st = Lux.setup(rng, model)

# Define loss function (mean squared error)
function mse_loss(model, ps, st, data)
    x, y_true = data
    y_pred, st_new = model(x, ps, st)
    loss = mean((y_pred .- y_true) .^ 2)
    return loss, st_new, ()
end
```

```
mse_loss (generic function with 1 method)
```

5.1.3 Step 3: Train

Feed the training data through the network 500 times (epochs), adjusting the parameters each time to reduce the prediction error.

```
opt = Adam(0.01f0)
function train_model(model, ps, st, data; epochs = 500, lr = 0.01f0)
    tstate = Training.TrainState(model, ps, st, Adam(lr))
```

```
    for epoch in 1:epochs
        _, loss, _, tstate = Training.single_train_step!(
            AutoZygote(), mse_loss, data, tstate
        )
        if epoch == 1 || epoch % 100 == 0
            @printf "Epoch %4d  MSE = %.6f\n" epoch loss
        end
    end
    return tstate
end

tstate = train_model(model, ps, st, (X_train, Y_train))

# Evaluate on test data
Y_pred_test, _ = model(X_test, tstate.parameters, tstate.states)
test_mse = mean((Y_pred_test .- Y_test) .^ 2)
test_mae = mean(abs.(Y_pred_test .- Y_test))

# Baseline: always predict the training-set mean score
baseline = fill(mean(Y_train), size(Y_test))
baseline_mae = mean(abs.(baseline .- Y_test))

@printf "Test MSE: %.6f  Test MAE: %.4f\n" test_mse test_mae
@printf "Baseline MAE (predict mean): %.4f\n" baseline_mae
```

```
Epoch    1  MSE = 0.298768
Epoch  100  MSE = 0.021929
Epoch  200  MSE = 0.008098
Epoch  300  MSE = 0.006308
Epoch  400  MSE = 0.005393
Epoch  500  MSE = 0.005107
Test MSE: 0.006323  Test MAE: 0.0615
Baseline MAE (predict mean): 0.2668
```

If the network's test MAE is clearly lower than the baseline MAE, it is learning meaningful structure instead of only predicting the average. In geoscience terms, that means the model is extracting relationships from the inputs (temperature context and coffee delay) rather than memorizing noise.

5.1.4 Step 4: Predict

Use the trained model to score a few specific sauna scenarios:

```
scenarios = [
    (sauna=80, outside=-20, coffee=10, label="Perfect: 80°C, -20°C, fresh coffee"),
    (sauna=65, outside=5,   coffee=100, label="Poor: lukewarm, mild, stale coffee"),
```

```
    (sauna=95, outside=-25, coffee=30, label="Extreme: 95°C, deep winter"),
]

function predict_score(s, model, ps, st, mu, sig)
    x = Float32.([(s.sauna - mu[1]) / sig[1],
                  (s.outside - mu[2]) / sig[2],
                  (s.coffee - mu[3]) / sig[3]])
    pred, _ = model(reshape(x, 3, 1), ps, st)
    return pred[1]
end

for s in scenarios
    sc = predict_score(s, model, tstate.parameters, tstate.states, mu, sig)
    @printf "%-35s %.2f\n" s.label sc
end
```

```
Perfect: 80°C, -20°C, fresh coffee  1.04
Poor: lukewarm, mild, stale coffee  0.07
Extreme: 95°C, deep winter          0.41
```

5.2 What just happened?

In about 30 lines of code you:

1. **Loaded data** from a file and split it into training and test sets.
2. **Built a model** — a small neural network.
3. **Trained it** — the computer adjusted thousands of numbers inside the model so that the predictions got closer and closer to the real scores.
4. **Used it** — fed in new inputs and got predictions back.

You don't yet know *why* this works. The next chapters will explain every part: what a neuron is, what the loss function does, how gradients flow backward through the network, and why the optimizer matters. But the overall workflow — data ⟶ model ⟶ train ⟶ predict — stays exactly the same for every neural network in this book.

This chapter deletes `sauna_log.txt` in a final hidden cleanup step so the temporary example file does not linger in the project directory after rendering. If you stop midway through the workflow or run only selected code blocks, remove it manually:

```
rm sauna_log.txt
```

On PowerShell:

```
Remove-Item sauna_log.txt
```

6 Building Blocks of Neural Networks

Key references

The modern understanding of neural networks rests on a small number of milestone contributions:

- **The perceptron** — the first trainable artificial neuron (Rosenblatt, 1958).
- **Backpropagation** — the algorithm that makes training deep networks practical (Rumelhart et al., 1986).
- **ReLU activations** — simple non-linearity that solved the vanishing-gradient problem for deep networks (Nair & Hinton, 2010).
- **Adam optimizer** — an adaptive learning-rate method that became the default for most deep-learning work (Kingma & Ba, 2015).
- **Dropout** — a regularization technique that prevents overfitting by randomly zeroing neurons during training (Srivastava et al., 2014).
- **Batch normalization** — stabilizes and accelerates training by normalizing layer inputs (Ioffe & Szegedy, 2015).
- **Deep learning review** — an authoritative overview of the entire field (LeCun et al., 2015).

In the previous chapter you trained a neural network without knowing what was going on inside. Now we unpack every piece.

A **feedforward neural network** is the simplest network topology: information moves from the input layer to the output layer with no recurrence, feedback loop, or hidden state. When this feedforward architecture is built by stacking several dense layers, it is usually called a **multilayer perceptron** (MLP). Later chapters will revisit that architecture in more detail, but this is the basic meaning of "feedforward" throughout the book.

6.1 Notation used in this part

Before we get into the details, it helps to keep a small notation guide in mind. We will use the same conventions throughout the neural-network chapters:

Symbol	Meaning	Convention
$\mathbf{x}$, $\mathbf{h}$, $\hat{\mathbf{y}}$	Vectors	Bold lowercase symbols denote vectors.
$W^{(l)}$, $\mathbf{b}^{(l)}$	Weights and bias at layer l	Superscripts in parentheses index layer depth.
$\mathbf{h}_t$	Hidden state at time step t	Subscripts usually index time, samples, or nodes.

Symbol	Meaning	Convention
$\mathbf{h}_i^{(k)}$	Node i representation at GNN layer k	Subscript for node, superscript for layer.
$\mathscr{L}$	Total loss	Use subscripts such as $\mathscr{L}_{\text{data}}$ or $\mathscr{L}_{\text{bc}}$ for components.
$\hat{y}$	Generic prediction	In inversion chapters we may switch to symbols such as d^{pred} and d^{obs} when the data meaning matters.

These choices are not the only valid ones, but keeping them fixed makes the later chapters easier to read.

6.2 The artificial neuron

A biological neuron receives signals, integrates them, and fires if the total exceeds a threshold. An artificial neuron follows the same idea in simplified form. Given an input vector $\mathbf{x} \in \mathbb{R}^n$, a neuron computes:

$$z = \sigma\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

where:

- $\mathbf{w} \in \mathbb{R}^n$ is the **weight vector** — one weight per input.
- $b \in \mathbb{R}$ is the **bias** — a constant shift.
- σ is a **generic activation function** — a non-linear function applied to the weighted sum.

When we mean a specific activation, we will write it explicitly, for example $\tanh(\cdot)$ or $\text{ReLU}(\cdot)$.

The weights and bias are the *learnable parameters*. Training a neural network means finding values of $\mathbf{w}$ and b (across all neurons) that make the network's output match the desired target.

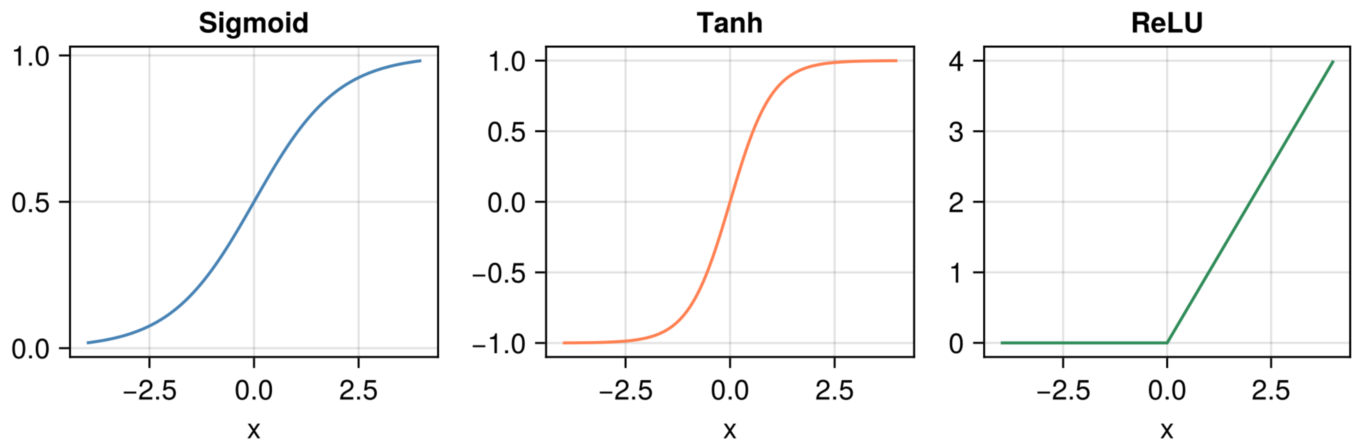
6.3 Activation functions

Without an activation function, stacking many layers would still produce a linear mapping — no matter how deep the network. The activation introduces non-linearity, which gives the network the ability to learn complex patterns. Common choices include:

Function	Formula	Notes
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	Squashes output to $(0, 1)$. Used in early networks.
Tanh	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	Centered at zero; range $(-1, 1)$.

Function	Formula	Notes
ReLU	$\mathrm{ReLU}(x) = \max(0, x)$	Default for most modern networks (Nair & Hinton, 2010).
Leaky ReLU	$\max(\alpha x, x)$ for small α	Avoids "dead neurons" where ReLU outputs zero permanently.

```
using CairoMakie

x = -4:0.01:4
fig = Figure(size = (700, 250))
ax1 = Axis(fig[1, 1], title = "Sigmoid", xlabel = "x")
lines!(ax1, x, 1 ./ (1 .+ exp.(-x)), color = :steelblue)
ax2 = Axis(fig[1, 2], title = "Tanh", xlabel = "x")
lines!(ax2, x, tanh.(x), color = :coral)
ax3 = Axis(fig[1, 3], title = "ReLU", xlabel = "x")
lines!(ax3, x, max.(0, x), color = :seagreen)
fig
```



6.4 Layers

A **layer** is a collection of neurons that operate in parallel on the same input. In a **dense** (fully connected) layer, every neuron receives every input:

$$\mathbf{h}^{(l)} = \sigma\left(W^{(l)}\,\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

where $W^{(l)} \in \mathbb{R}^{m \times n}$ is the weight matrix for layer l (m neurons, n incoming features) and $\mathbf{b}^{(l)} \in \mathbb{R}^{m}$ is the corresponding bias vector.

In Lux.jl, creating a dense layer with 3 inputs and 8 outputs using the `tanh` activation looks like:

```
using Lux, Random

layer = Dense(3 => 8, tanh)
rng = Xoshiro(0)
ps, st = Lux.setup(rng, layer)
println("Weight matrix size: ", size(ps.weight))
println("Bias vector size:   ", size(ps.bias))
```

```
Weight matrix size: (8, 3)
Bias vector size:   (8,)
```

6.5 Networks: stacking layers

A neural network is formed by **chaining layers** so that the output of one layer becomes the input to the next. In Lux.jl this is done with `Chain`:

```
model = Chain(
    Dense(3 => 16, relu),   # hidden layer 1
    Dense(16 => 8, relu),   # hidden layer 2
    Dense(8 => 1)            # output layer (no activation → raw value)
)
ps, st = Lux.setup(rng, model)
```

```
((layer_1 = (weight = Float32[-0.87449217 -1.2761252 1.4936523; -1.495204 1.6102042 1.9952867; … ; 0.20966864
1.5356417; 0.9057038 -1.7467937 0.40091395], bias = Float32[0.24175844, 0.15477172, 0.16122879, 0.068802044, -
0.3278107, 0.07800663, 0.06683914, -0.18069106, -0.5477161, -0.08528596, 0.35881144, 0.14086951, 0.49927178, 0
0.7281545 0.22887067 … 0.66682464 0.666083; -0.03448241 -0.06876608 … -0.7251021 0.14132917; … ; 0.6025773 0.
0.7467699 0.062139012 … -0.48436588 0.24338031], bias = Float32[0.17736006, 0.13691956, 0.048085272, -
0.04162532, 0.16139144, -0.09218615, -0.052222192, 0.20891446]), layer_3 = (weight = Float32[-
0.5780716 -0.5690776 … 0.004986225 -0.13891284], bias = Float32[-0.038554586])), (layer_1 = NamedTuple(), laye
```

The number of layers and the number of neurons per layer are **hyperparameters** — choices made by the user, not learned from data.

6.6 The loss function

The **loss function** (or cost function) measures how far the network's predictions are from the true targets. Training tries to minimize this value. Common choices:

- **Mean Squared Error (MSE)** — for regression: $\mathscr{L} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$

- **Cross-Entropy** — for classification: $\mathscr{L} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)\right]$

In the chapters ahead, when the loss has several pieces, we will write them as named components such as $\mathscr{L}_{\text{data}}$, $\mathscr{L}_{\text{pde}}$, or $\mathscr{L}_{\text{bc}}$ rather than leaving them unnamed inside one long equation.

The choice of loss function depends on the problem. Regression tasks almost always use MSE; classification tasks use cross-entropy.

6.7 Backpropagation and gradients

Training requires knowing **how each parameter affects the loss**. This information is captured by the **gradient** — the partial derivative of the loss with respect to each parameter.

The **backpropagation** algorithm (Rumelhart et al., 1986) computes these gradients efficiently using the chain rule of calculus, working backward from the loss through each layer to the parameters:

$$\frac{\partial \mathscr{L}}{\partial w_{ij}} = \frac{\partial \mathscr{L}}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}$$

In Julia, automatic differentiation libraries such as **Zygote.jl** handle this for you. You write the forward computation; Zygote computes the gradients automatically.

6.8 Optimizers

Once we have the gradients, we need a rule for updating the parameters. The simplest approach is **gradient descent**:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathscr{L}$$

where η is the **learning rate**. A small η means slow but stable progress; a large η learns faster but risks overshooting.

In practice, more sophisticated optimizers are used:

- **SGD with momentum** — accumulates a running average of past gradients to smooth updates.
- **Adam** (Kingma & Ba, 2015) — adapts the learning rate individually for each parameter. It is the most widely used optimizer and a good default.
- **RMSProp** — scales updates by a running average of squared gradients; often effective for noisy sequence problems.
- **AdamW** — Adam with decoupled weight decay; often preferred in modern deep-learning training because regularization behaves more predictably.
- **L-BFGS** — a quasi-Newton optimizer useful in some scientific ML settings (including some PINN workflows) for final fine-tuning after Adam.

6.8.1 Learning-rate schedules

In practice, keeping a fixed learning rate for all epochs is rarely optimal. Common schedules are:

- **Step decay**: reduce η by a factor every N epochs.
- **Cosine decay**: gradually lower η with a cosine schedule.
- **Warmup + decay**: start with a small η, increase for a few epochs, then decay.

A good practical pattern is: start with Adam/AdamW at a moderate learning rate, then reduce the learning rate once validation loss plateaus.

6.9 Regularization

Deep networks can **overfit**: they memorize the training data instead of learning general patterns. Regularization techniques reduce this risk:

- **Dropout** (Srivastava et al., 2014) — during training, randomly sets a fraction of neuron outputs to zero. At test time, all neurons are active but their outputs are scaled. This forces the network to not rely on any single neuron.
- **Batch normalization** (Ioffe & Szegedy, 2015) — normalizes the input to each layer across the current mini-batch. This stabilizes training and acts as a mild regularizer.
- **Weight decay** — adds a penalty proportional to the squared weights to the loss, discouraging large parameter values.
- **Early stopping** — monitors validation loss during training and stops when it begins to increase.

6.10 The training loop

Putting it all together, training a neural network follows this loop:

1. **Forward pass** — feed input through the network to get a prediction.
2. **Compute loss** — compare prediction to true target.
3. **Backward pass** — compute gradients of the loss with respect to all parameters.
4. **Update parameters** — apply the optimizer rule.
5. **Repeat** for many epochs (full passes through the training data).

Each iteration over a subset (**mini-batch**) of the data is one **step**. One full pass through the entire dataset is one **epoch**. The code example in the previous chapter follows exactly this loop.

6.11 Minimum diagnostics checklist (for geoscience workflows)

Before trusting any neural-network result, check the following:

1. **Separate splits**: train, validation, and test sets must be separate in time/space where relevant.
2. **Baseline comparison**: compare against a simple baseline (mean predictor, persistence model, linear regression).

3. **Generalization gap**: if train loss is low but validation/test loss is high, you are overfitting.
4. **Domain sanity check**: predictions should respect basic geoscientific constraints (ranges, trends, known physics).
5. **Error by regime**: report errors by important regimes (e.g., depth interval, facies class, season, tectonic setting), not only one global metric.

These checks are often more important than squeezing out a small gain in one metric.

6.12 Summary

Concept	Role
Neuron	Weighted sum → activation
Layer	Collection of neurons
Network	Chain of layers
Loss	Measures prediction error
Gradient	Direction to improve parameters
Backpropagation	Efficient gradient computation
Optimizer	Parameter update rule
Regularization	Prevents overfitting

7 Multilayer Perceptrons

Key references

- **Universal approximation** — a multilayer perceptron with a single hidden layer can approximate any continuous function to arbitrary accuracy, given enough neurons (Cybenko, 1989; Hornik et al., 1989).
- **Backpropagation** — the training algorithm that made multilayer networks practical (Rumelhart et al., 1986).
- **Deep learning review** — comprehensive overview of multilayer perceptrons and deep architectures (LeCun et al., 2015).

A **multilayer perceptron** (MLP) is the standard dense feedforward network introduced in the previous chapter. Here the focus is on what that architecture can represent, how it is trained in practice, and when it is a good baseline choice.

7.1 Architecture

An MLP consists of:

1. An **input layer** — one node per feature (not a computation layer, just the data entry point).
2. One or more **hidden layers** — dense layers with activation functions.
3. An **output layer** — produces the final prediction.

Every neuron in one layer is connected to every neuron in the next layer, which is why these are called **fully connected** or **dense** layers.

For an MLP with L hidden layers, the computation at layer l is:

$$\mathbf{h}^{(l)} = \sigma\left(W^{(l)}\,\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

where $\mathbf{h}^{(0)} = \mathbf{x}$ is the input and the final layer output $\mathbf{h}^{(L+1)} = \hat{\mathbf{y}}$ is the prediction.

7.2 Universal approximation

The **universal approximation theorem** (Cybenko, 1989; Hornik et al., 1989) states that a feedforward network with a single hidden layer containing a sufficient number of neurons can approximate any continuous function on a compact set to arbitrary precision. This is a powerful existence result, but it does not tell you *how many* neurons you need or *how* to find the weights. In practice, deeper (more layers) but narrower MLPs tend to generalize better than a single massive hidden layer.

7.3 Code example: function approximation

Let's use an MLP to approximate the function $f(x) = \sin(2\pi x)\, e^{-x^2}$ from noisy samples.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)

# Generate training data
n = 200
x_data = Float32.(range(-2, 2, length = n))
f_true(x) = sin(2π * x) * exp(-x^2)
y_data = f_true.(x_data) .+ 0.05f0 .* randn(rng, Float32, n)

X = reshape(x_data, 1, :)
Y = reshape(y_data, 1, :)

# Train/test split
idx = randperm(rng, n)
n_train = Int(round(0.8 * n))
tr = idx[1:n_train]
te = idx[n_train+1:end]

X_train, Y_train = X[:, tr], Y[:, tr]
X_test,  Y_test  = X[:, te], Y[:, te]
```

```
(Float32[1.839196 1.8994975 … 0.110552765 -0.4924623], [-0.04691443233346553 0.06493792605102766 … 0.64
0.11029157439538681])
```

```
# Build a 2-hidden-layer MLP
model = Chain(
    Dense(1 => 32, relu),
    Dense(32 => 32, relu),
    Dense(32 => 1)
)

ps, st = Lux.setup(rng, model)

function mse_loss(model, ps, st, data)
    x, y = data
    ŷ, st_new = model(x, ps, st)
    loss = mean((ŷ .- y) .^ 2)
    return loss, st_new, ()
end
```

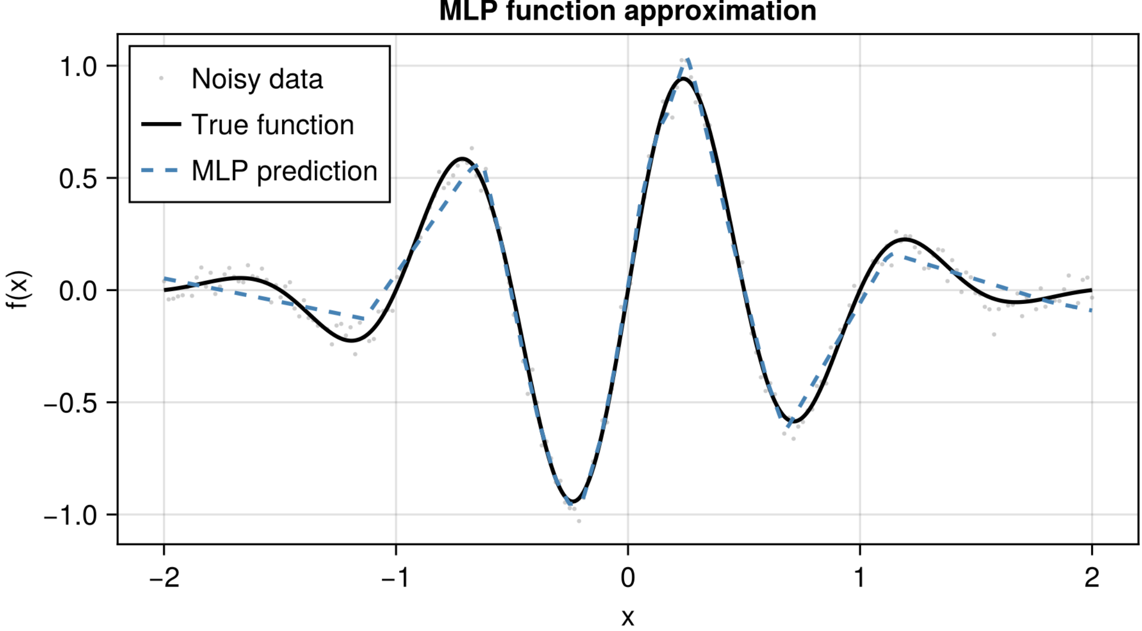
```
mse_loss (generic function with 1 method)
```

```julia
opt = Adam(0.005f0)
function train_model(model, ps, st, data; epochs = 1000, lr = 0.005f0)
    tstate = Training.TrainState(model, ps, st, Adam(lr))
    for epoch in 1:epochs
        _, loss, _, tstate = Training.single_train_step!(
            AutoZygote(), mse_loss, data, tstate
        )
        if epoch == 1 || epoch % 200 == 0
            @printf "Epoch %4d  MSE = %.6f\n" epoch loss
        end
    end
    return tstate
end

tstate = train_model(model, ps, st, (X_train, Y_train))

# Holdout evaluation
Y_test_pred, _ = model(X_test, tstate.parameters, tstate.states)
test_mse = mean((Y_test_pred .- Y_test) .^ 2)
@printf "Holdout test MSE = %.6f\n" test_mse
```

```
Epoch    1  MSE = 10.610252
Epoch  200  MSE = 0.050187
Epoch  400  MSE = 0.027555
Epoch  600  MSE = 0.015669
Epoch  800  MSE = 0.007704
Epoch 1000  MSE = 0.004761
Holdout test MSE = 0.004282
```

```julia
# Plot the result
x_fine = Float32.(range(-2, 2, length = 500))
X_fine = reshape(x_fine, 1, :)
Y_pred, _ = model(X_fine, tstate.parameters, tstate.states)

fig = Figure(size = (600, 350))
ax = Axis(fig[1, 1], xlabel = "x", ylabel = "f(x)",
       title = "MLP function approximation")
scatter!(ax, x_data, y_data, markersize = 3, color = (:gray, 0.4),
         label = "Noisy data")
lines!(ax, x_fine, f_true.(x_fine), color = :black, linewidth = 2,
       label = "True function")
lines!(ax, x_fine, vec(Y_pred), color = :steelblue, linewidth = 2,
    linestyle = :dash, label = "MLP prediction")
```

```
axislegend(ax, position = :lt)
fig
```



Interpretation tip: use the holdout test MSE as the primary quality indicator. A smooth fit that looks good visually can still overfit noisy samples; a separate test set is your safeguard.

7.4 Effect of depth and width

The universal approximation theorem guarantees that a *wide enough* single hidden layer can approximate any function. In practice:

- **Wider** layers (more neurons) increase capacity but can overfit.
- **Deeper** networks (more layers) learn hierarchical features and often generalize better with fewer total parameters.
- **Very deep** networks are harder to train due to vanishing gradients — residual connections (K. He et al., 2016) and normalization help.

A useful rule of thumb: start with 2–3 hidden layers of moderate width (32–128 neurons) and adjust based on performance.

7.5 When to use multilayer perceptrons

MLPs are the default starting point whenever:

- The input is a **fixed-size feature vector** (e.g., geophysical measurements at a station).

- There is **no spatial or temporal structure** that you want the architecture to exploit.
- You need a **simple, fast, interpretable baseline** before trying more complex architectures.

For data with spatial structure (images, grids), convolutional networks are usually better. For sequential data (time series), recurrent networks or transformers are preferred. The MLP remains the building block inside most of these architectures.

7.6 Geoscience applications

Multilayer perceptrons have been widely used in geoscience as flexible function approximators:

- **Seismic picking and trace editing** — early feedforward networks were used for first-break refraction picking and seismic trace cleaning (McCormack et al., 1993).
- **Velocity analysis and moveout correction** — MLPs were applied to automate NMO correction and velocity estimation in seismic processing (Calderón-Macías et al., 1998).
- **Reservoir characterization** — feedforward networks were used for reservoir and seismic characterization from waveform-derived attributes (An et al., 2001; An & Moon, 2005).
- **Lithology classification** — borehole lithology inference from downhole logs was an early and influential classification use case (Benaouda et al., 1999).
- **Petrophysical prediction** — porosity and permeability estimation from well logs is a classic MLP regression task in rock physics and reservoir studies (Huang et al., 1996; Huang & Williamson, 1997).
- **Thermal-property estimation** — MLPs were also used to predict thermal conductivity from geophysical well logs (Goutorbe et al., 2006).
- **Geophysical inversion** — MLPs can serve as surrogate forward models, mapping model parameters to predicted data. Once trained, they replace expensive physics-based simulations and can be embedded inside iterative inversion schemes (Lopez-Alvis et al., 2019).
- **Overview** — Bergen et al. (2019) and Reichstein et al. (2019) provide broad reviews of machine learning across the geosciences, including many regression and classification problems for which MLPs are natural baselines.

The MLP is rarely the final architecture for production geoscience workflows, but it is almost always the first model you should try. If an MLP solves the problem, the extra complexity of deeper architectures is unnecessary.

8 Convolutional Neural Networks

Key references

- **LeNet** — the first successful convolutional neural network for image recognition (LeCun et al., 1989).
- **AlexNet** — deep CNN that won the ImageNet competition and launched the deep-learning era (Krizhevsky et al., 2012).
- **U-Net** — encoder-decoder architecture with skip connections for dense prediction (Ronneberger et al., 2015).
- **ResNet** — residual connections enabling very deep networks (100+ layers) (K. He et al., 2016).

A **convolutional neural network** (CNN) exploits the spatial structure in data — the fact that nearby pixels or grid cells tend to be related. Instead of connecting every input to every neuron, a CNN slides small learned filters across the data, detecting local patterns such as edges, textures, and shapes. This makes CNNs far more parameter-efficient than feedforward networks for image-like data.

8.1 The convolution operation

In a CNN, a **filter** (or **kernel**) is a small weight matrix, typically 3×3 or 5×5. The filter slides across the input and at each position computes a dot product between the filter weights and the local patch of input values. This produces a **feature map** — a new grid where each cell represents how strongly that local pattern was detected at that position.

For a 2D input $\mathbf{X}$ and a filter $\mathbf{K}$ of size $k \times k$, the convolution at position (i, j) is:

$$(\mathbf{X} * \mathbf{K})_{i,j} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \mathbf{K}_{m,n} \cdot \mathbf{X}_{i+m,\, j+n}$$

A convolutional layer applies many such filters in parallel, each learning to detect a different pattern.

8.2 Pooling

After convolution, **pooling** layers reduce the spatial size of the feature maps, keeping only the most important information. The most common type is **max pooling**, which takes the maximum value in each small window (e.g., 2×2). Pooling reduces computation, provides some translation invariance, and increases the receptive field of deeper layers.

8.3 A typical CNN architecture

A CNN usually alternates convolutional and pooling layers, progressively reducing spatial resolution while increasing the number of feature channels:

1. **Input** — e.g., a 28×28 single-channel image.
2. **Conv ⟶ ReLU ⟶ Pool** — repeated 2–3 times.
3. **Flatten** — reshape the 2D feature maps into a 1D vector.
4. **Dense layers** — one or two fully connected layers for the final prediction.

8.4 Code example: classifying simple seismic image patches

We use a standard 2D CNN to classify tiny synthetic seismic-style images into three classes: layered horizons, a faulted horizon pattern, and a dome-like structure. This is easier to read than the previous texture example because the three patterns are visually distinct and the problem matches the usual CNN story: take an image as input, return one class label as output.

The problem formulation is simple: each input is one 32×32 grayscale image patch, and the target is one of three structural classes. The network output is a vector of three class scores. After a softmax, those scores become class probabilities, and the largest probability gives the predicted class.

This is still a toy problem. Real seismic interpretation is not this clean. But for a first CNN example, it is useful because we can clearly see what the network is trying to separate and we can directly check whether the predictions match the visible pattern in the image.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)

class_names = ["Layered", "Faulted", "Dome"]

function gaussian2d(x, y, μx, μy, σx, σy)
    exp.(-0.5f0 .* (((x .- μx) ./ σx) .^ 2 .+ ((y .- μy) ./ σy) .^ 2))
end

# Generate a small synthetic seismic-style image patch.
function make_seismic_patch(rng, n = 32)
    class_id = rand(rng, 1:3)
    x = Float32.(range(-1, 1, length = n))
    y = Float32.(range(-1, 1, length = n))
    xx = repeat(reshape(x, n, 1), 1, n)
    yy = repeat(reshape(y, 1, n), n, 1)

    image = zeros(Float32, n, n)

    if class_id == 1
        # Layered reflectors.
```

```
        image .= 0.50f0 .+ 0.22f0 .* sin.(Float32(7.0) .* Float32(pi) .* (yy .+ 0.04f0 .* sin.(Float
        image .+= 0.015f0 .* randn(rng, Float32, n, n)
    elseif class_id == 2
        # Faulted reflectors with a visible offset.
        shifted_yy = yy .+ 0.22f0 .* (xx .> 0.08f0)
        image .= 0.50f0 .+ 0.22f0 .* sin.(Float32(7.0) .* Float32(pi) .* shifted_yy)
        image .-= 0.12f0 .* gaussian2d(xx, yy, 0.08f0, 0.0f0, 0.03f0, 0.85f0)
        image .+= 0.015f0 .* randn(rng, Float32, n, n)
    else
        # Dome-like reflector geometry.
        dome = yy .+ 0.55f0 .* exp.(-((xx ./ 0.42f0) .^ 2))
        image .= 0.50f0 .+ 0.22f0 .* sin.(Float32(7.0) .* Float32(pi) .* dome)
        image .+= 0.015f0 .* randn(rng, Float32, n, n)
    end

    image = Float32.(clamp.(image, 0.03f0, 0.98f0))
    patch = reshape(image, n, n, 1)
    y = zeros(Float32, 3)
    y[class_id] = 1.0f0
    return patch, y, class_id
end

# Create a labelled image dataset
n_samples = 900
patches = zeros(Float32, 32, 32, 1, n_samples)    # (height, width, channels, batch)
labels = zeros(Float32, 3, n_samples)

for i in 1:n_samples
    x, y, _ = make_seismic_patch(rng)
    patches[:, :, :, i] .= x
    labels[:, i] .= y
end

# Train/test split
idx = randperm(rng, n_samples)
n_train = Int(round(0.8 * n_samples))
tr = idx[1:n_train]
te = idx[n_train+1:end]

X_train, Y_train = patches[:, :, :, tr], labels[:, tr]
X_test,  Y_test  = patches[:, :, :, te], labels[:, te]
```

```
(Float32[0.51531047 0.28371927 … 0.72622246 0.5048716; 0.51920027 0.2755116 … 0.6919893 0.4924293; … ;
```

The input tensor has shape (`height, width, channels, batch`), and the label for each patch is a one-hot vector with three entries. So this is just standard three-class image classification with geoscience-flavored

patterns.

```
# Build a small 2D CNN classifier
model = Chain(
    Conv((5, 5), 1 => 8, relu; pad = SamePad()),
    MaxPool((2, 2)),
    Conv((3, 3), 8 => 16, relu; pad = SamePad()),
    MaxPool((2, 2)),
    WrappedFunction(x -> reshape(x, :, size(x, 4))),
    Dense(16 * 8 * 8 => 24, relu),
    Dense(24 => 3)
)

ps, st = Lux.setup(rng, model)

function softmax_cols(x)
    x_shift = x .- maximum(x, dims = 1)
    ex = exp.(x_shift)
    ex ./ sum(ex, dims = 1)
end

function cross_entropy_loss(model, ps, st, data)
    x, y = data
    logits, st_new = model(x, ps, st)
    ŷ = softmax_cols(logits)
    ε = 1.0f-7
    loss = -mean(sum(y .* log.(ŷ .+ ε), dims = 1))
    return loss, st_new, ()
end

function predicted_classes(probabilities)
    [findmax(probabilities[:, i])[2] for i in axes(probabilities, 2)]
end

function true_classes(labels)
    [findmax(labels[:, i])[2] for i in axes(labels, 2)]
end
```

```
true_classes (generic function with 1 method)
```

```
function train_model(model, ps, st, data; epochs = 240, lr = 0.0015f0)
    tstate = Training.TrainState(model, ps, st, Adam(lr))
    for epoch in 1:epochs
        _, loss, _, tstate = Training.single_train_step!(
            AutoZygote(), cross_entropy_loss, data, tstate
        )
```

```
        if epoch == 1 || epoch % 60 == 0
            @printf "Epoch %3d  cross-entropy = %.4f\n" epoch loss
        end
    end
    return tstate
end

tstate = train_model(model, ps, st, (X_train, Y_train))

# Holdout evaluation
test_logits, _ = model(X_test, tstate.parameters, tstate.states)
test_prob = softmax_cols(test_logits)
test_loss = -mean(sum(Y_test .* log.(test_prob .+ 1.0f-7), dims = 1))
test_acc = mean(predicted_classes(test_prob) .== true_classes(Y_test))
@printf "Holdout cross-entropy: %.4f  Accuracy: %.3f\n" test_loss test_acc
```

```
Epoch   1  cross-entropy = 2.7561
Epoch  60  cross-entropy = 0.0423
Epoch 120  cross-entropy = 0.0014
Epoch 180  cross-entropy = 0.0003
Epoch 240  cross-entropy = 0.0001
Holdout cross-entropy: 0.0001  Accuracy: 1.000
```

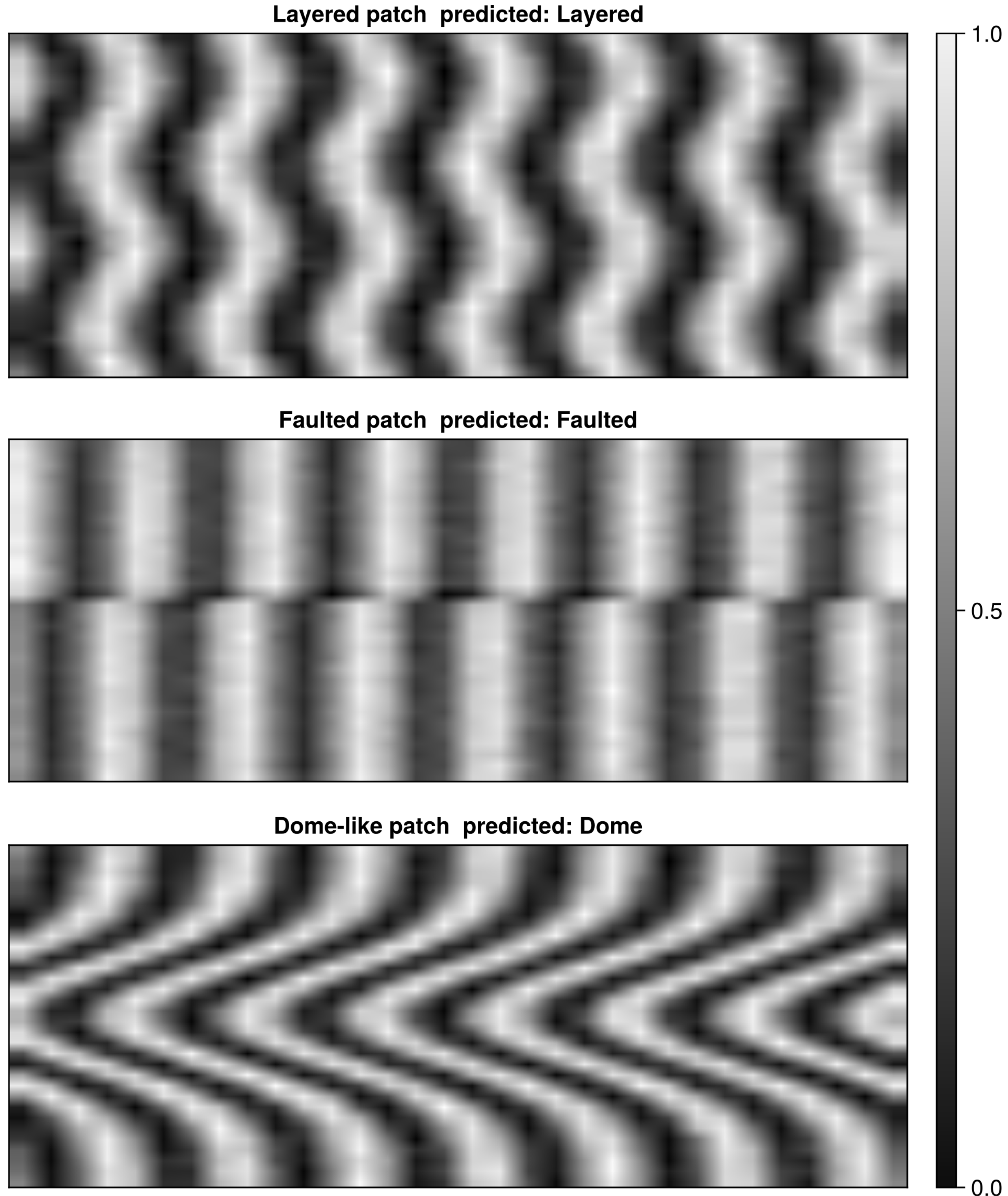
The output above reports two things. The cross-entropy tells us how confident the network is on the correct class, while the accuracy tells us how often it predicts the right class on unseen patches. For a teaching example like this one, we want both numbers to show that the CNN has learned the visible image pattern instead of just memorizing the training set.

```
confusion = zeros(Int, 3, 3)
for (true_class, pred_class) in zip(true_classes(Y_test), predicted_classes(test_prob))
    confusion[true_class, pred_class] += 1
end

for i in 1:3
    class_acc = confusion[i, i] / sum(confusion[i, :])
    @printf "%s accuracy: %.3f\n" class_names[i] class_acc
end
```

```
Layered accuracy: 1.000
Faulted accuracy: 1.000
Dome accuracy: 1.000
```

Those per-class accuracies are useful because a single overall accuracy can hide one weak class. If one seismic pattern is consistently confused with another, it will show up here even when the mean score still looks good.

```
# Visualize one image patch from each class
function sample_with_label(seed, wanted_label)
    rng_local = Xoshiro(seed)
    while true
        x, y, class_id = make_seismic_patch(rng_local)
        class_id == wanted_label && return x, y, class_id
    end
end

patch_layered, _, _ = sample_with_label(99, 1)
patch_faulted, _, _ = sample_with_label(199, 2)
patch_dome, _, _ = sample_with_label(299, 3)

prob_layered, _ = model(reshape(patch_layered, 32, 32, 1, 1), tstate.parameters, tstate.states)
prob_faulted, _ = model(reshape(patch_faulted, 32, 32, 1, 1), tstate.parameters, tstate.states)
prob_dome, _ = model(reshape(patch_dome, 32, 32, 1, 1), tstate.parameters, tstate.states)

prob_layered = softmax_cols(prob_layered)
prob_faulted = softmax_cols(prob_faulted)
prob_dome = softmax_cols(prob_dome)

fig = Figure(size = (640, 760))

ax1 = Axis(fig[1, 1], title = "Layered patch  predicted: $(class_names[findmax(prob_layered[:, 1])[2]])")
image!(ax1, permutedims(dropdims(patch_layered, dims = 3), (2, 1)))
hidedecorations!(ax1)

ax2 = Axis(fig[2, 1], title = "Faulted patch  predicted: $(class_names[findmax(prob_faulted[:, 1])[2]])")
image!(ax2, permutedims(dropdims(patch_faulted, dims = 3), (2, 1)))
hidedecorations!(ax2)

ax3 = Axis(fig[3, 1], title = "Dome-like patch  predicted: $(class_names[findmax(prob_dome[:, 1])[2]])")
image!(ax3, permutedims(dropdims(patch_dome, dims = 3), (2, 1)))
hidedecorations!(ax3)

Colorbar(fig[1:3, 2], limits = (0, 1), colormap = :grays)
fig
```



In the plot, each panel is one input image and the title shows the predicted class. That gives a direct visual check that the network output matches the pattern a human reader would also identify.

8.5 Key CNN architectures

Several landmark architectures expanded the capabilities of CNNs:

- **ResNet** (K. He et al., 2016) — introduces skip connections that add the input of a block to its output, enabling training of very deep networks (100+ layers) without vanishing gradients.
- **U-Net** (Ronneberger et al., 2015) — an encoder-decoder architecture with skip connections at each resolution level, originally designed for biomedical image segmentation. Widely adopted in geoscience for dense prediction tasks.

8.6 Geoscience applications

CNNs are the dominant architecture for geoscience tasks involving gridded spatial data:

- **Seismic fault detection** — Wu et al. (2019) trained a 3D CNN (FaultSeg3D) on synthetic seismic volumes to segment faults in 3D, demonstrating that CNNs can detect complex fault geometries directly from seismic data.
- **Earthquake detection** — Perol et al. (2018) developed ConvQuake, a CNN that detects and locates earthquakes directly from raw seismic waveforms, outperforming traditional detection methods in noisy environments.
- **Seismic waveform classification and first-break picking** — Yuan et al. (2020) used a CNN for waveform classification and first-break picking, showing how convolutional models can detect local seismic patterns directly from traces.
- **Remote sensing** — land-use classification, mineral mapping, and change detection from satellite and airborne imagery are natural CNN applications, as the data is inherently image-like.
- **Seismic image interpretation** — 2D CNNs can classify local image patterns such as layered structure, faults, or dome-like geometry from small patches, as demonstrated in the code example above.

The key insight is: **whenever your geoscience data lives on a regular grid, a CNN is likely a good starting point**. The spatial weight sharing built into convolutions matches the physics of spatially correlated earth properties.

9 Recurrent Neural Networks

Key references

- **Simple RNN** — the idea that a network can process sequences by maintaining a hidden state (Elman, 1990).
- **LSTM** — Long Short-Term Memory, which solved the vanishing-gradient problem for sequences and enabled learning over hundreds of time steps (Hochreiter & Schmidhuber, 1997).
- **GRU** — Gated Recurrent Unit, a simplified variant of LSTM with comparable performance (Cho et al., 2014).
- **ConvLSTM** — combining convolutional and recurrent structures for spatiotemporal prediction (Shi et al., 2015).

Feedforward and convolutional networks process each input independently. But many geoscience datasets are **sequential**: seismograms, well-log curves, climate records, and satellite time series all have a natural ordering in time (or depth). A **recurrent neural network** (RNN) is designed for this: it processes a sequence one step at a time, maintaining a hidden state that carries information from earlier steps to later ones.

9.1 The simple RNN

At each time step t, a simple RNN receives the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$, and produces a new hidden state:

$$\mathbf{h}_t = \tanh\left(W_h\, \mathbf{h}_{t-1} + W_x\, \mathbf{x}_t + \mathbf{b}\right)$$

Here the subscript t indexes time, while the subscripts on W_h and W_x identify the role of each matrix: hidden-to-hidden and input-to-hidden, respectively. This is the same convention introduced earlier in the part: superscripts are reserved for layer depth, and subscripts mark time or semantic roles.

The hidden state acts as the network's *memory*. The output at each step can be read from $\mathbf{h}_t$ directly or passed through an additional dense layer.

Problem: In practice, simple RNNs struggle to learn long-range dependencies because gradients either vanish (shrink to zero) or explode (grow unboundedly) when propagated backward through many time steps.

9.2 Long Short-Term Memory (LSTM)

The **LSTM** (Hochreiter & Schmidhuber, 1997) solves the vanishing-gradient problem by introducing a **cell state** $\mathbf{c}_t$ alongside the hidden state $\mathbf{h}_t$, controlled by three learned gates:

- **Forget gate** $\mathbf{f}_t$ — decides what information to discard from the cell state.
- **Input gate** $\mathbf{i}_t$ — decides what new information to store.
- **Output gate** $\mathbf{o}_t$ — decides what part of the cell state to expose.

As before, the subscript t denotes the time step. The different letter subscripts on the weight matrices indicate the gate they belong to.

$$\begin{aligned}
\mathbf{f}_t &= \sigma\big(W_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f\big) \\
\mathbf{i}_t &= \sigma\big(W_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i\big) \\
\tilde{\mathbf{c}}_t &= \tanh\big(W_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c\big) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\
\mathbf{o}_t &= \sigma\big(W_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o\big) \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}$$

The cell state can carry information unchanged through many time steps, and the gates learn to open and close during training, allowing the network to decide what to remember and what to forget.

9.3 Gated Recurrent Unit (GRU)

The **GRU** (Cho et al., 2014) is a simplification of the LSTM that merges the cell state and hidden state into a single state vector, using two gates instead of three:

$$\begin{aligned}
\mathbf{r}_t &= \sigma\big(W_r[\mathbf{h}_{t-1}, \mathbf{x}_t]\big) \\
\mathbf{z}_t &= \sigma\big(W_z[\mathbf{h}_{t-1}, \mathbf{x}_t]\big) \\
\tilde{\mathbf{h}}_t &= \tanh\big(W_h[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t]\big) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t
\end{aligned}$$

GRUs have fewer parameters than LSTMs and train faster, while producing similar results on many tasks.

9.4 Code example: predicting a synthetic geophysical time series

We generate a synthetic oscillating signal (simulating a geophysical measurement with periodic and trend components) and train an LSTM to predict the next value given the recent past.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)
```

```
# Generate a synthetic time series: trend + oscillation + noise
t = Float32.(0:0.05:10)
signal = 0.3f0 .* t .+ sin.(2π .* 0.5f0 .* t) .+ 0.3f0 .* sin.(2π .* 1.3f0 .* t) .+
         0.15f0 .* randn(rng, Float32, length(t))

# Standardize before training so the recurrent model does not spend capacity on scale alone
μ_signal = mean(signal)
σ_signal = std(signal)
signal_scaled = (signal .- μ_signal) ./ σ_signal

# Create input/output pairs using a sliding window
window = 20
n_pairs = length(signal_scaled) - window
X_seq = zeros(Float32, 1, window, n_pairs)  # (features, time_steps, batch)
Y_seq = zeros(Float32, 1, n_pairs)          # (features, batch)

for i in 1:n_pairs
    X_seq[1, :, i] = signal_scaled[i:i+window-1]
    Y_seq[1, i]    = signal_scaled[i+window]
end

# Train/test split (chronological split to respect time ordering)
n_train = Int(round(0.8 * n_pairs))
X_train, Y_train = X_seq[:, :, 1:n_train], Y_seq[:, 1:n_train]
X_test,  Y_test  = X_seq[:, :, n_train+1:end], Y_seq[:, n_train+1:end]
```

```
(Float32[0.24319372 -0.057659 … 0.91191345 1.0251187;;; -0.057659 -0.13387857 … 1.0251187 1.3189939;;;
0.13387857 -0.17540371 … 1.3189939 1.5677433;;; … ;;; 0.85227734 0.96867204 … 0.4017068 0.8487579;;; 0
```

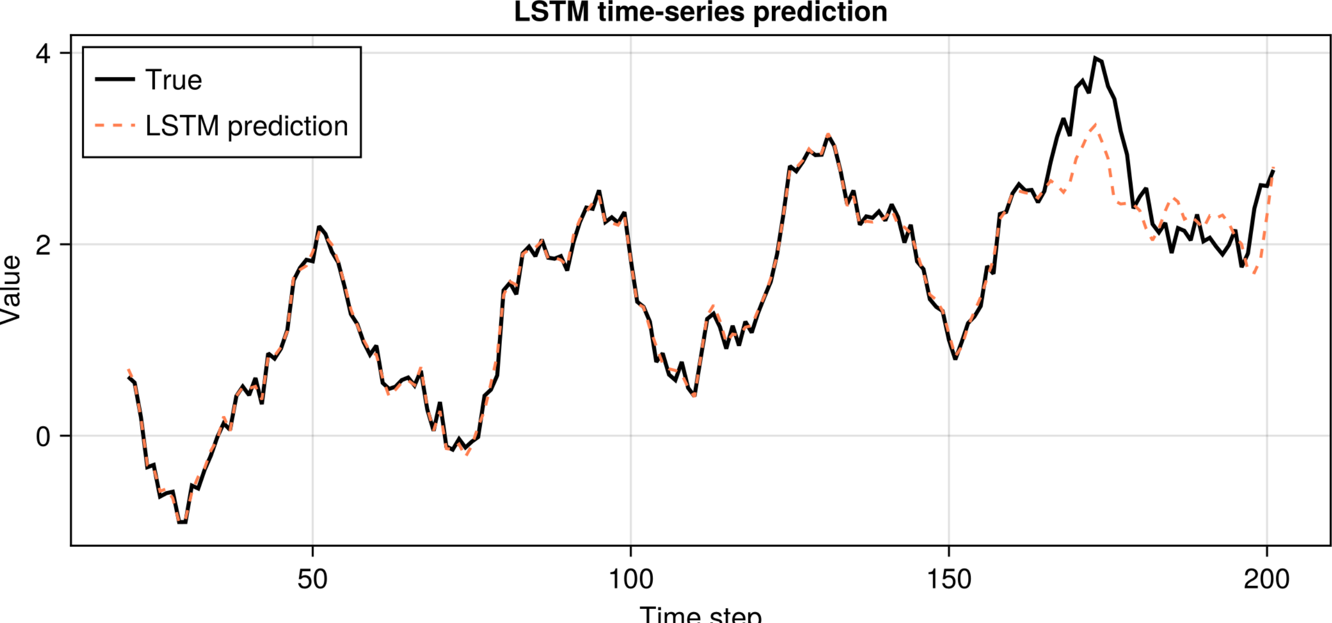
```
# Build an LSTM model that reads the sequence, then maps the final hidden state to a prediction
model = Chain(
    Recurrence(LSTMCell(1 => 32)),
    Dense(32 => 1)
)

ps, st = Lux.setup(rng, model)

function mse_loss(model, ps, st, data)
    x, y = data
    ŷ, st_new = model(x, ps, st)
    loss = mean((ŷ .- y) .^ 2)
    return loss, st_new, ()
end
```

```
mse_loss (generic function with 1 method)
```

```
function train_model(model, ps, st, data; epochs = 600, lr = 0.003f0)
    tstate = Training.TrainState(model, ps, st, Adam(lr))
    for epoch in 1:epochs
        _, loss, _, tstate = Training.single_train_step!(
            AutoZygote(), mse_loss, data, tstate
        )
        if epoch == 1 || epoch % 120 == 0
            @printf "Epoch %3d  MSE = %.6f\n" epoch loss
        end
    end
    return tstate
end

tstate = train_model(model, ps, st, (X_train, Y_train))

# Holdout evaluation
Y_test_pred, _ = model(X_test, tstate.parameters, tstate.states)
test_mse = mean(((σ_signal .* Y_test_pred .+ μ_signal) .- (σ_signal .* Y_test .+ μ_signal)) .^ 2)
@printf "Holdout test MSE = %.6f\n" test_mse
```

```
Epoch   1  MSE = 0.895788
Epoch 120  MSE = 0.024068
Epoch 240  MSE = 0.011021
Epoch 360  MSE = 0.006385
Epoch 480  MSE = 0.004051
Epoch 600  MSE = 0.003718
Holdout test MSE = 0.233183
```

```
# Predict and plot
Y_pred, _ = model(X_seq, tstate.parameters, tstate.states)
Y_pred = σ_signal .* Y_pred .+ μ_signal

fig = Figure(size = (700, 350))
ax = Axis(fig[1, 1], xlabel = "Time step", ylabel = "Value",
          title = "LSTM time-series prediction")
lines!(ax, window+1:length(signal), signal[window+1:end],
       color = :black, label = "True", linewidth = 2)
lines!(ax, window+1:length(signal), vec(Y_pred),
       color = :coral, label = "LSTM prediction", linestyle = :dash)
axislegend(ax, position = :lt)
fig
```



9.5 When to use RNNs

RNNs are the natural choice when:

- Data has a **sequential or temporal structure** (time series, depth-indexed logs).
- **Order matters** — shuffling the data would destroy information.
- You need to **capture dependencies** between earlier and later parts of a sequence.

For very long sequences (thousands of steps), **transformers** (next chapter) often outperform RNNs because they can attend to any part of the sequence without passing information step by step.

9.6 Geoscience applications

Recurrent networks have been applied to a wide range of sequential geoscience problems:

- **Sequence modeling in seismology** — recurrent and hybrid sequence models are widely used for waveform analysis and event interpretation. W. Zhu & Beroza (2019) is a closely related deep-learning benchmark for seismic arrival picking, though PhaseNet itself is primarily convolutional rather than recurrent.
- **Climate and weather forecasting** — Ham et al. (2019) used a CNN-LSTM hybrid to forecast the El Niño–Southern Oscillation (ENSO) up to 18 months ahead, significantly outperforming physics-based dynamical models.
- **Precipitation nowcasting** — Shi et al. (2015) introduced the ConvLSTM, combining convolutional and LSTM operations to predict radar echo sequences, a spatiotemporal forecasting task.

- **Machine learning in geoscience overview** — Dramsch (2020) provides a comprehensive review of 70 years of machine learning in the geosciences, covering many recurrent-network applications in seismology, well-log analysis, and geophysical signal processing.

10 Transformers

> Key references
>
> - **Attention is all you need** — the paper that introduced the transformer architecture (Vaswani et al., 2017).
> - **BERT** — bidirectional transformer pre-training that reshaped NLP (Devlin et al., 2019).
> - **Vision Transformer (ViT)** — showed that transformers can match or beat CNNs on image classification (Dosovitskiy et al., 2021).

The **transformer** is the architecture behind large language models and, increasingly, behind state-of-the-art models in vision, time-series forecasting, and scientific computing. Unlike RNNs, which process sequences one step at a time, transformers process the entire sequence at once using a mechanism called **self-attention** that lets each element directly attend to every other element.

10.1 Self-attention

The core idea: for each element in the input sequence, compute how much attention it should pay to every other element, then produce an output that is a weighted combination of the values.

Given an input sequence $\mathbf{X} \in \mathbb{R}^{n \times d}$ (n tokens, d features), three linear projections produce:

$$\mathbf{Q} = \mathbf{X}W_Q, \quad \mathbf{K} = \mathbf{X}W_K, \quad \mathbf{V} = \mathbf{X}W_V$$

where $\mathbf{Q}$ (queries), $\mathbf{K}$ (keys), and $\mathbf{V}$ (values) are matrices. The attention output is:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

The softmax row gives the **attention weights** — how much each position attends to every other position. The $\sqrt{d_k}$ scaling prevents the dot products from becoming too large.

10.2 Multi-head attention

Instead of computing a single attention, the transformer uses **multi-head attention**: it runs h parallel attention functions with different learned projections, then concatenates and projects the results:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)\, W_O$$

Each head can learn to focus on different types of relationships (e.g., one head might attend to nearby positions, another to distant ones).

10.3 The transformer block

A single transformer block consists of:

1. **Multi-head self-attention** — with a residual connection and layer normalization.
2. **Feed-forward network** — two dense layers with a non-linearity, also with a residual connection and layer normalization.

$$\mathbf{z} = \text{LayerNorm}(\mathbf{x} + \text{MultiHeadAttn}(\mathbf{x}))$$
$$\mathbf{y} = \text{LayerNorm}(\mathbf{z} + \text{FFN}(\mathbf{z}))$$

Stacking many such blocks gives the transformer its depth.

10.4 Positional encoding

Self-attention is **permutation-invariant** — it does not know the order of the input. To inject sequence order, transformers add a **positional encoding** to the input embeddings. The original paper used sinusoidal functions:

$$PE_{(pos,2i)} = \sin\left(pos/10000^{2i/d}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(pos/10000^{2i/d}\right)$$

This gives each position a unique signature that the network can learn to interpret.

10.5 Code example: minimal self-attention

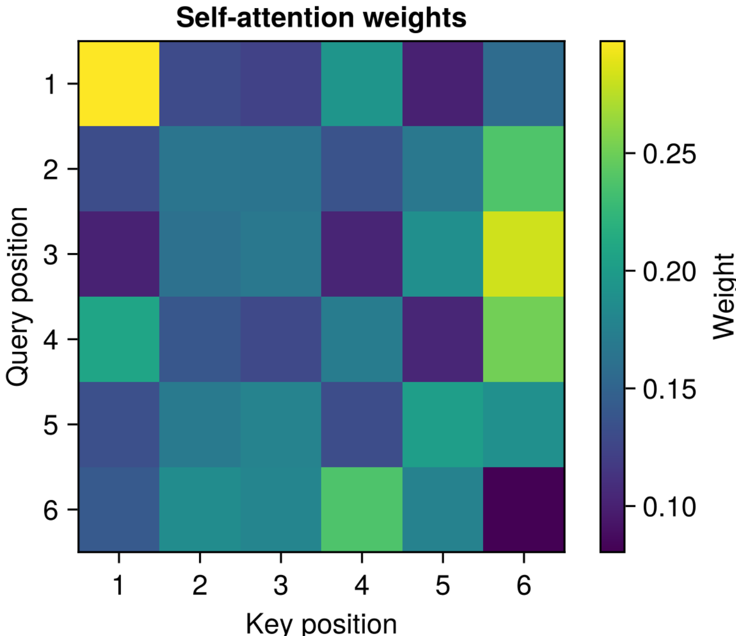
We implement a minimal self-attention mechanism from scratch to illustrate the core computation. This is not meant for production use, but shows exactly what happens inside the attention layer.

```
using LinearAlgebra, Random, CairoMakie

rng = Xoshiro(42)

#-----sequence setup------
seq_len, d_model = 6, 4
X = randn(rng, Float32, seq_len, d_model)

#-----learned projections------
dₖ = d_model
W_Q = randn(rng, Float32, d_model, dₖ) .* 0.5f0
```

```
W_K = randn(rng, Float32, d_model, dₖ) .* 0.5f0
W_V = randn(rng, Float32, d_model, dₖ) .* 0.5f0

Q = X * W_Q
K = X * W_K
V = X * W_V

#-----attention weights------
scores = Q * K' ./ sqrt(Float32(dₖ))

#-----row-wise softmax------
function row_softmax(S)
    exp_S = exp.(S .- maximum(S, dims = 2))
    return exp_S ./ sum(exp_S, dims = 2)
end

attn_weights = row_softmax(scores)

#-----weighted output------
output = attn_weights * V

println("Attention weights (each row sums to 1):")
for i in 1:seq_len
    println("  Token $i → ", round.(attn_weights[i, :], digits = 3))
end
```

```
Attention weights (each row sums to 1):
  Token 1 → Float32[0.298, 0.13, 0.123, 0.194, 0.1, 0.156]
  Token 2 → Float32[0.131, 0.164, 0.164, 0.136, 0.168, 0.238]
  Token 3 → Float32[0.1, 0.16, 0.167, 0.102, 0.188, 0.282]
  Token 4 → Float32[0.208, 0.139, 0.128, 0.171, 0.103, 0.252]
  Token 5 → Float32[0.133, 0.169, 0.177, 0.131, 0.201, 0.189]
  Token 6 → Float32[0.142, 0.185, 0.179, 0.237, 0.176, 0.08]
```

```
# Visualize the attention pattern
fig = Figure(size = (400, 350))
ax = Axis(fig[1, 1], title = "Self-attention weights",
          xlabel = "Key position", ylabel = "Query position",
          xticks = 1:seq_len, yticks = 1:seq_len,
          yreversed = true)
hm = heatmap!(ax, 1:seq_len, 1:seq_len, attn_weights',
              colormap = :viridis)
Colorbar(fig[1, 2], hm, label = "Weight")
fig
```



How to read this plot:

- Each **row** corresponds to one query token.
- Bright cells indicate which key positions that query attends to most.
- Rows should sum to 1 (a probability distribution).

This chapter demonstrates the mechanics of attention, not end-to-end benchmark performance. In real geoscience transformer models, evaluation should include holdout skill metrics (e.g., MAE/RMSE, event detection F1, or forecast skill scores) and comparisons against RNN/CNN baselines.

10.6 Advantages over RNNs

Feature	RNN	Transformer
Long-range dependencies	Difficult (vanishing gradients)	Direct attention
Parallelization	Sequential (slow)	Fully parallel
Memory cost	$O(n)$ per step	$O(n^2)$ for full attention
Positional awareness	Built-in (sequential processing)	Requires positional encoding

Transformers scale better to long sequences and large datasets, which explains their dominance in modern AI. For very long sequences, efficient variants (sparse attention, linear attention) reduce the $O(n^2)$ cost.

10.7 Geoscience applications

Transformers are rapidly entering geoscience, especially for tasks where long-range dependencies and large-scale data matter:

- **Earthquake detection** — Mousavi et al. (2020) introduced the Earthquake Transformer (EQTransformer), which uses attention mechanisms to simultaneously detect earthquakes and pick seismic phases from continuous waveform data, achieving superior performance over CNN and RNN baselines.
- **Weather forecasting** — Bi et al. (2023) (Pangu-Weather) is a prominent example of the recent shift toward attention-heavy learned weather models for medium-range forecasting. By contrast, FourCastNet (Pathak et al., 2022) is an important fast weather model built on Fourier neural operators rather than a transformer, so it fits more naturally with the operator-learning material later in the book.
- **Earth observation** — Vision transformers are being applied to satellite imagery for land-cover classification, change detection, and environmental monitoring, following the success of ViT (Dosovitskiy et al., 2021) in computer vision.

The transformer is increasingly the architecture of choice for problems involving long sequences, multimodal data, or large-scale pre-training in geoscience.

11 Graph Neural Networks

Key references

- **The GNN model** — the foundational framework for neural networks that operate on graph-structured data (Scarselli et al., 2009).
- **Graph Convolutional Networks (GCN)** — spectral-domain convolution on graphs using a first-order Chebyshev approximation (Kipf & Welling, 2017).
- **Message Passing Neural Networks (MPNN)** — a unifying framework that describes most GNN variants as message-passing operations between nodes (Gilmer et al., 2017).

All neural networks we have seen so far assume a specific data structure: vectors (feedforward), grids (CNNs), or sequences (RNNs/transformers). But many real-world datasets are naturally represented as **graphs** — collections of nodes connected by edges. In geoscience, examples include:

- **Sensor networks** — seismic stations, weather stations, or boreholes at irregular locations.
- **Geological meshes** — unstructured finite-element meshes for geological modeling.
- **Molecular structures** — mineral crystal structures or fluid molecules.
- **River networks** — hydrological drainage basins as directed graphs.

A **graph neural network** (GNN) operates directly on graph-structured data, learning representations that respect the connectivity structure.

11.1 Graphs: a brief reminder

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of:

- **Nodes** $\mathcal{V} = \{v_1, \dots, v_n\}$, each with a feature vector $\mathbf{x}_i$.
- **Edges** $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, representing connections between nodes. Edges can also carry features.

The connectivity is described by an **adjacency matrix** $A \in \{0, 1\}^{n \times n}$, where $A_{ij} = 1$ if there is an edge from node i to node j.

11.2 Message passing

Most GNNs follow the **message-passing** paradigm (Gilmer et al., 2017). At each layer, every node:

1. **Gathers messages** from its neighbors.
2. **Aggregates** them (e.g., sum, mean, or max).
3. **Updates** its own feature vector based on the aggregated message and its current state.

Formally, at layer k:

$$\mathbf{h}_i^{(k)} = \phi\left(\mathbf{h}_i^{(k-1)}, \bigoplus_{j \in \mathcal{N}(i)} \psi\left(\mathbf{h}_i^{(k-1)}, \mathbf{h}_j^{(k-1)}, \mathbf{e}_{ij}\right)\right)$$

Here the notation is worth stating explicitly: the subscript i identifies the node, while the superscript (k) identifies the message-passing layer. That keeps graph notation aligned with the rest of the book, where superscripts in parentheses track layer depth.

where:

- $\mathcal{N}(i)$ is the set of neighbors of node i.
- ψ is the **message function** (how to compute a message from a neighbor).
- $\bigoplus$ is the **aggregation function** (sum, mean, or max over all messages).
- ϕ is the **update function** (how to combine the aggregated message with the node's current state).

After K layers of message passing, each node's representation has been informed by nodes up to K hops away.

11.3 Graph Convolutional Network (GCN)

The **GCN** (Kipf & Welling, 2017) is a popular and simple GNN variant. The layer-wise update rule is:

$$H^{(k+1)} = \sigma\left(\tilde{D}^{-1/2}\, \tilde{A}\, \tilde{D}^{-1/2}\, H^{(k)}\, W^{(k)}\right)$$

In matrix form, $H^{(k)}$ collects all node features at layer k row by row, while $W^{(k)}$ is the learnable weight matrix for that layer.

where $\tilde{A} = A + I$ (adjacency with self-loops), $\tilde{D}$ is the degree matrix of $\tilde{A}$, $H^{(k)}$ is the feature matrix at layer k, $W^{(k)}$ is the learnable weight matrix, and σ is an activation function.

In plain language: each node averages the features of its neighbors (including itself), then applies a linear transformation and a non-linearity.

11.4 Code example: node classification on a synthetic graph

We build a simple graph where node features are noisy versions of a class label, and the graph structure encodes which nodes are likely to share the same class. The GNN learns to denoise the labels using the graph structure.

```
using Random, LinearAlgebra, Statistics, Printf, CairoMakie, Zygote

rng = Xoshiro(42)

# Create a synthetic graph: two clusters
n_nodes = 40
```

```
n_class1 = 20

# Node features: 2D, with class-dependent mean
features = zeros(Float32, n_nodes, 2)
labels = zeros(Int, n_nodes)
for i in 1:n_nodes
    if i <= n_class1
        features[i, :] = [1.0f0, 0.0f0] .+ 0.5f0 .* randn(rng, Float32, 2)
        labels[i] = 1
    else
        features[i, :] = [0.0f0, 1.0f0] .+ 0.5f0 .* randn(rng, Float32, 2)
        labels[i] = 2
    end
end

# Adjacency: higher probability of edges within the same class
A = zeros(Float32, n_nodes, n_nodes)
for i in 1:n_nodes
    for j in i+1:n_nodes
        p = labels[i] == labels[j] ? 0.3 : 0.05
        if rand(rng) < p
            A[i, j] = 1.0f0
            A[j, i] = 1.0f0
        end
    end
end
```

```
# Implement a simple 2-layer GCN manually
# Normalized adjacency with self-loops
A_hat = A + I(n_nodes)
D_hat = Diagonal(vec(sum(A_hat, dims = 2)))
D_inv_sqrt = Diagonal(1.0f0 ./ sqrt.(diag(D_hat)))
A_norm = D_inv_sqrt * A_hat * D_inv_sqrt

# Parameters
d_in, d_hidden, d_out = 2, 8, 2
W1 = 0.5f0 .* randn(rng, Float32, d_in, d_hidden)
W2 = 0.5f0 .* randn(rng, Float32, d_hidden, d_out)

# Forward pass
function gcn_forward(X, A_norm, W1, W2)
    H1 = max.(A_norm * X * W1, 0)       # GCN layer 1 + ReLU
    H2 = A_norm * H1 * W2               # GCN layer 2 (logits)
    return H2
end
```

```
# Softmax
function softmax_rows(X)
    eX = exp.(X .- maximum(X, dims = 2))
    return eX ./ sum(eX, dims = 2)
end

# One-hot targets
targets = zeros(Float32, n_nodes, 2)
for i in 1:n_nodes
    targets[i, labels[i]] = 1.0f0
end

# Node split for evaluation
perm = randperm(rng, n_nodes)
n_train = Int(round(0.7 * n_nodes))
train_idx = perm[1:n_train]
test_idx  = perm[n_train+1:end]
```

```
12-element Vector{Int64}:
  7
 40
 11
  5
 13
  9
  8
 23
 33
 37
 38
 39
```

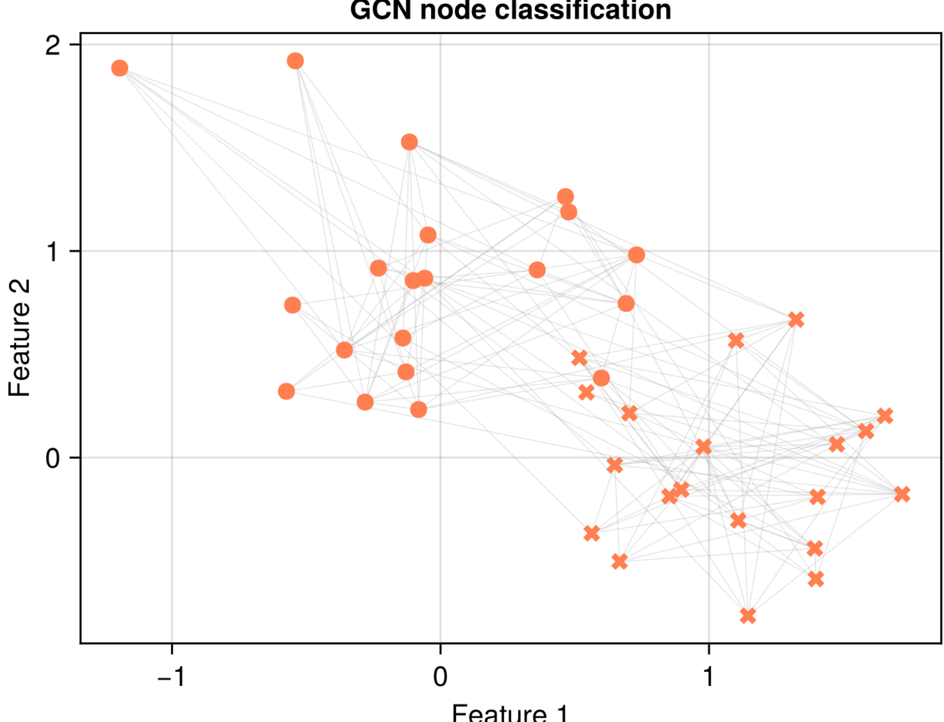
```
# Train the GCN
lr = 0.05f0

for epoch in 1:200
    function train_loss(W1_, W2_)
        logits = gcn_forward(features, A_norm, W1_, W2_)
        probs = softmax_rows(logits)
        p_tr = probs[train_idx, :]
        t_tr = targets[train_idx, :]
        return -mean(sum(t_tr .* log.(p_tr .+ 1.0f-7), dims = 2))
    end

    loss = train_loss(W1, W2)
```

```
    grad_W1, grad_W2 = Zygote.gradient(train_loss, W1, W2)

    W1 .-= lr .* grad_W1
    W2 .-= lr .* grad_W2

    if epoch == 1 || epoch % 50 == 0
        logits = gcn_forward(features, A_norm, W1, W2)
        probs = softmax_rows(logits)
        preds = argmax.(eachrow(probs))
        train_acc = mean(preds[train_idx] .== labels[train_idx])
        test_acc = mean(preds[test_idx] .== labels[test_idx])
        @printf "Epoch %3d  Loss: %.4f  Train acc: %.1f%%  Test acc: %.1f%%\n" epoch loss 100*train_acc 100
    end
end
```

```
Epoch   1  Loss: 0.8375  Train acc: 50.0%  Test acc: 50.0%
Epoch  50  Loss: 0.6482  Train acc: 50.0%  Test acc: 50.0%
Epoch 100  Loss: 0.5825  Train acc: 50.0%  Test acc: 50.0%
Epoch 150  Loss: 0.5477  Train acc: 50.0%  Test acc: 50.0%
Epoch 200  Loss: 0.5272  Train acc: 50.0%  Test acc: 50.0%
```

```
# Visualize the graph with predicted labels
logits = gcn_forward(features, A_norm, W1, W2)
preds = argmax.(eachrow(softmax_rows(logits)))

# Layout: use feature coordinates
fig = Figure(size = (500, 400))
ax = Axis(fig[1, 1], title = "GCN node classification",
          xlabel = "Feature 1", ylabel = "Feature 2")

# Draw edges
for i in 1:n_nodes
    for j in i+1:n_nodes
        if A[i, j] > 0
            lines!(ax, [features[i, 1], features[j, 1]],
                      [features[i, 2], features[j, 2]],
                   color = (:gray, 0.2), linewidth = 0.5)
        end
    end
end

# Draw nodes colored by predicted class
colors = [p == 1 ? :steelblue : :coral for p in preds]
markers = [l == p ? :circle : :xcross for (l, p) in zip(labels, preds)]
scatter!(ax, features[:, 1], features[:, 2],
```

```
        color = colors, marker = markers, markersize = 12)

fig
```



In the plot, circles are correctly classified nodes and crosses are misclassified ones. The GNN uses graph connectivity (which nodes are connected) together with node features to make predictions — even when the features alone are noisy and overlapping.

11.5 When to use GNNs

GNNs are the right choice when:

- Data lives on an **irregular structure** (not a regular grid or sequence).
- **Relationships between entities** are important (sensor networks, molecular graphs).
- You want to make predictions about **nodes**, **edges**, or the **entire graph**.
- The graph structure itself carries information (e.g., which stations are nearby).

If your data is on a regular grid, a CNN is simpler and usually sufficient. If your data is sequential, use an RNN or transformer. GNNs fill the gap for irregular, relational data.

11.6 Geoscience applications

- **Weather forecasting** — Lam et al. (2023) introduced GraphCast, a graph-neural-network-based model for medium-range global weather prediction. It represents the atmosphere as a graph on a multi-resolution mesh, enabling accurate 10-day forecasts at $0.25°$ resolution while running orders of magnitude faster than traditional numerical weather models.
- **Hydrological networks** — Stanev et al. (2021) applied GNNs to hydrological coherence analysis using remotely sensed water-level data, leveraging the natural graph structure of river networks and gauge station connectivity.
- **Molecular and mineral property prediction** — in geochemistry, GNNs can predict properties of minerals and fluids directly from their molecular graph structures, following the same approach used successfully for drug discovery and materials science (Gilmer et al., 2017).

12 Autoencoders

Key references

- **Deep autoencoders** — training deep networks to learn compressed representations for dimensionality reduction (Hinton & Salakhutdinov, 2006).
- **Variational autoencoders (VAE)** — a probabilistic framework that combines autoencoders with Bayesian inference to generate new samples (Kingma & Welling, 2014).

This chapter starts the generative-model block of the neural-networks part. The previous chapters focused mostly on architectures for prediction, classification, or representation learning on specific data structures. From here through GANs, diffusion models, and flow matching, the emphasis shifts toward models that learn a data distribution well enough to reconstruct, sample, or generate new realizations.

An **autoencoder** is a neural network trained to reconstruct its own input. This may sound pointless — why predict something you already have? The key is that the network is forced through a **bottleneck**: a narrow hidden layer with far fewer neurons than the input dimension. To reconstruct the input from this compressed representation, the network must learn which features are essential and which are noise.

12.1 Architecture

An autoencoder has two parts:

1. **Encoder** f_θ: maps the high-dimensional input $\mathbf{x}$ to a low-dimensional **latent code** $\mathbf{z}$:

$$\mathbf{z} = f_\theta(\mathbf{x}), \quad \mathbf{z} \in \mathbb{R}^d, \quad d \ll \dim(\mathbf{x})$$

2. **Decoder** g_ϕ: reconstructs the input from the latent code:

$$\hat{\mathbf{x}} = g_\phi(\mathbf{z})$$

The network is trained to minimize the **reconstruction loss**:

$$\mathscr{L} = \|\mathbf{x} - \hat{\mathbf{x}}\|^2$$

After training, the encoder provides a learned compression (useful for dimensionality reduction, denoising, and feature extraction) and the decoder can generate data from latent codes.

12.2 Variational Autoencoder (VAE)

A standard autoencoder maps each input to a single point in latent space. The **variational autoencoder** (Kingma & Welling, 2014) instead maps each input to a *distribution* — a mean μ and variance σ^2 — and samples from that distribution:

$$\mathbf{z} = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

The VAE loss combines reconstruction accuracy with a regularization term that keeps the latent distributions close to a standard normal:

$$\mathcal{L}_{\text{VAE}} = \underbrace{\|\mathbf{x} - \hat{\mathbf{x}}\|^2}_{\text{reconstruction}} + \underbrace{D_{\text{KL}}\big(q(\mathbf{z}|\mathbf{x}) \,\|\, p(\mathbf{z})\big)}_{\text{KL divergence}}$$

This gives the VAE a smooth, continuous latent space that can be sampled to **generate new data**.

12.3 Code example: denoising autoencoder for geophysical signals

We train a simple autoencoder to denoise a 1D signal — a common preprocessing task in geophysics.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)

# Generate clean signals and noisy versions
function make_signal(rng, n = 64)
    t = Float32.(range(0, 2π, length = n))
    clean = sin.(t) .+ 0.5f0 .* sin.(3 .* t)
    phase = 2π * rand(rng, Float32)
    amp   = 0.5f0 + rand(rng, Float32)
    clean = amp .* sin.(t .+ phase) .+ (amp * 0.5f0) .* sin.(3 .* t .+ phase)
    return clean
end

n_samples = 200
sig_len = 64
clean_data = zeros(Float32, sig_len, n_samples)
noisy_data = zeros(Float32, sig_len, n_samples)

for i in 1:n_samples
    c = make_signal(rng, sig_len)
    clean_data[:, i] = c
    noisy_data[:, i] = c .+ 0.3f0 .* randn(rng, Float32, sig_len)
end
```

```
# Train/test split
idx = randperm(rng, n_samples)
n_train = Int(round(0.8 * n_samples))
tr = idx[1:n_train]
te = idx[n_train+1:end]

X_train, Y_train = noisy_data[:, tr], clean_data[:, tr]
X_test,  Y_test  = noisy_data[:, te], clean_data[:, te]
```

```
(Float32[0.029786032 1.286609 … -0.45210695 0.4506653; -0.29807103 1.1513951 … 0.43684977 0.42317572; …
0.5844341 0.09730356; -0.5292163 0.82614696 … 0.4062874 0.15289801], Float32[0.09621503 1.3667396 … 0.0
0.081914954; -0.08477017 1.5328605 … 0.2656388 0.15320125; … ; 0.27371335 1.1510824 … -
0.10672952 -0.31406265; 0.09621489 1.3667397 … 0.08092139 -0.08191477])
```

```
# Autoencoder: encoder compresses to latent_dim, decoder reconstructs
latent_dim = 8

encoder = Chain(
    Dense(sig_len => 32, relu),
    Dense(32 => latent_dim, relu)
)

decoder = Chain(
    Dense(latent_dim => 32, relu),
    Dense(32 => sig_len)
)

autoencoder = Chain(encoder, decoder)

ps, st = Lux.setup(rng, autoencoder)

function ae_loss(model, ps, st, data)
    noisy, clean = data
    reconstructed, st_new = model(noisy, ps, st)
    loss = mean((reconstructed .- clean) .^ 2)
    return loss, st_new, ()
end
```

```
ae_loss (generic function with 1 method)
```

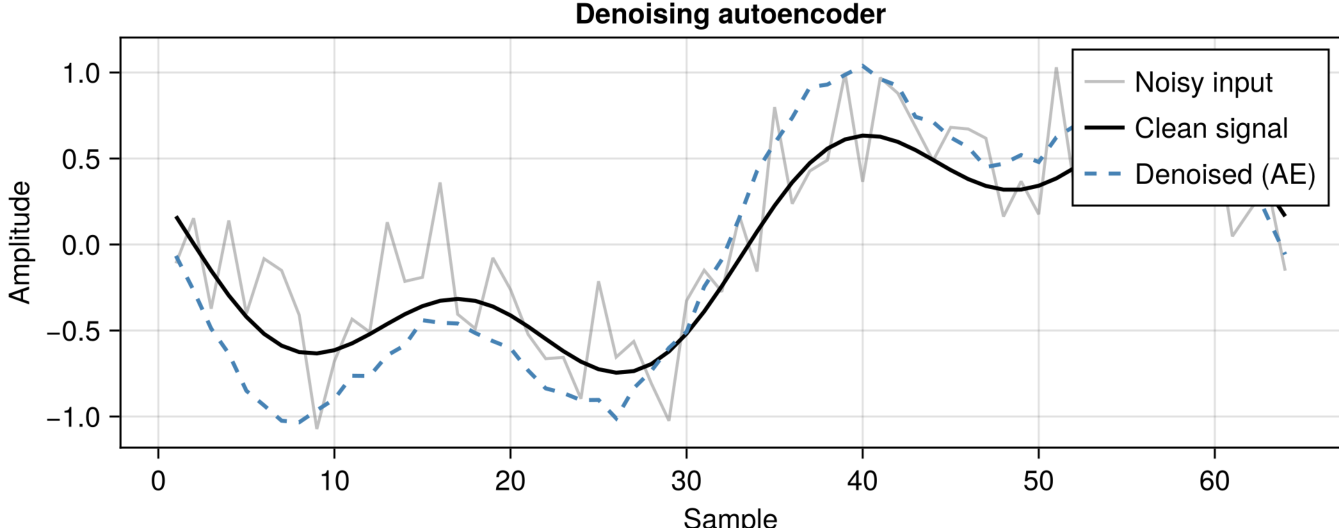
```
function train_model(model, ps, st, data; epochs = 500, lr = 0.003f0)
    tstate = Training.TrainState(model, ps, st, Adam(lr))
    for epoch in 1:epochs
```

```
        _, loss, _, tstate = Training.single_train_step!(
            AutoZygote(), ae_loss, data, tstate
        )
        if epoch == 1 || epoch % 100 == 0
            @printf "Epoch %4d  MSE = %.6f\n" epoch loss
        end
    end
    return tstate
end

tstate = train_model(autoencoder, ps, st, (X_train, Y_train))

# Holdout reconstruction quality
Y_test_pred, _ = autoencoder(X_test, tstate.parameters, tstate.states)
test_mse = mean((Y_test_pred .- Y_test) .^ 2)
@printf "Holdout reconstruction MSE = %.6f\n" test_mse
```

```
Epoch    1  MSE = 6.554671
Epoch  100  MSE = 0.291780
Epoch  200  MSE = 0.066843
Epoch  300  MSE = 0.047740
Epoch  400  MSE = 0.040926
Epoch  500  MSE = 0.037677
Holdout reconstruction MSE = 0.173980
```

```
# Test on a new signal
test_clean = make_signal(Xoshiro(99), sig_len)
test_noisy = test_clean .+ 0.3f0 .* randn(Xoshiro(99), Float32, sig_len)

denoised, _ = autoencoder(reshape(test_noisy, sig_len, 1),
                          tstate.parameters, tstate.states)

fig = Figure(size = (700, 300))
ax = Axis(fig[1, 1], xlabel = "Sample", ylabel = "Amplitude",
          title = "Denoising autoencoder")
lines!(ax, test_noisy, color = (:gray, 0.5), label = "Noisy input")
lines!(ax, test_clean, color = :black, linewidth = 2, label = "Clean signal")
lines!(ax, vec(denoised), color = :steelblue, linewidth = 2,
       linestyle = :dash, label = "Denoised (AE)")
axislegend(ax, position = :rt)
fig
```

```
┌ Warning: Mixed-Precision `matmul_cpu_fallback!` detected and Octavian.jl cannot be used for this set of inpu
└ @ LuxLib.Impl C:\Users\pmishra\.julia\packages\LuxLib\ZJ3gh\src\impl\matmul.jl:194
```



12.4 When to use autoencoders

Task	Autoencoder variant
Dimensionality reduction	Standard AE
Denoising	Denoising AE (train on noisy → clean pairs)
Anomaly detection	Train on normal data; high reconstruction error = anomaly
Generative modeling	VAE (sample from latent space to create new data)
Feature learning	Use encoder output as features for downstream tasks

12.5 Geoscience applications

- **Seismic denoising** — autoencoders trained to map noisy seismic traces to clean versions, effectively learning the noise characteristics of the acquisition system.
- **Geological model compression** — high-dimensional 3D geological property models can be compressed to a low-dimensional latent space using autoencoders, making inversion and uncertainty quantification computationally feasible.
- **Anomaly detection in monitoring data** — autoencoders trained on normal operating data (e.g., from geothermal wells or mining sensors) flag anomalies when reconstruction error exceeds a threshold.
- **Subsurface modeling** — Lopez-Alvis et al. (2019) used deep autoencoder-based approaches for inverse modeling of subsurface transport, where the autoencoder compresses the parameter space before inversion.
- **Overview** — Bergen et al. (2019) discusses the role of representation learning and dimensionality reduction, of which autoencoders are a central tool, across geoscience disciplines.

13 Generative Adversarial Networks

Key references

- **GANs** — the original framework introducing the adversarial training paradigm: a generator network learns to produce realistic data by competing against a discriminator network (Goodfellow et al., 2014).
- **DCGAN** — deep convolutional GAN, which established stable architectures and training practices for image generation (Radford et al., 2016).

A **generative adversarial network** (GAN) sits inside a short block of chapters on generative models. The previous chapters mostly focused on architectures used for prediction, classification, or representation learning on different data structures. In this part of the book, autoencoders, GANs, diffusion models, and flow matching are grouped together because they all learn data distributions and produce new samples, even though their training mechanisms differ substantially.

A **generative adversarial network** (GAN) consists of two neural networks that are trained simultaneously in competition:

- The **generator** G takes random noise $\mathbf{z}$ and produces a synthetic sample $G(\mathbf{z})$.
- The **discriminator** D receives either a real sample or a generated sample and tries to classify it as real or fake.

Training alternates between improving the discriminator (so it better distinguishes real from fake) and improving the generator (so its fakes become more convincing). The result is a generator that can produce realistic, novel samples from the data distribution.

13.1 The adversarial objective

The GAN training objective is a minimax game:

$$\min_G \max_D \ \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\big[\log D(\mathbf{x})\big] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\big[\log\big(1 - D(G(\mathbf{z}))\big)\big]$$

At convergence, the generator produces samples that the discriminator cannot distinguish from real data. In practice, GANs can be difficult to train: the generator and discriminator must be balanced, and training can be unstable.

13.2 Architecture

The original GAN used feedforward networks for both G and D. The **DCGAN** (Radford et al., 2016) established best practices for using convolutional architectures:

- **Generator**: uses transposed convolutions to upsample noise into an image.
- **Discriminator**: uses standard convolutions to classify images as real or fake.
- Batch normalization, LeakyReLU in the discriminator, and ReLU in the generator improve stability.

13.3 Code example: a more stable GAN for a rock-physics crossplot

The earlier GAN version had the right idea but the training was too unstable. Here we keep the same geoscience target, synthetic porosity-velocity points, but switch to a more standard Wasserstein-style setup with a critic and weight clipping. That is still simple enough for a chapter example, but it is much less likely to collapse onto only one cluster.

The problem formulation is different from the CNN chapter. We are not classifying anything here. The generator takes random noise as input and produces a synthetic (ϕ, v_p) point. The critic looks at real and generated points and learns to score which ones are more realistic. After training, the output we care about is the cloud of generated samples and whether it matches the real crossplot shape.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)

# Generate synthetic porosity-velocity pairs for two rock populations.
function make_crossplot_sample(rng)
    if rand(rng) < 0.5f0
        # High-porosity, lower-velocity population
        ϕ = clamp(0.28f0 + 0.025f0 * randn(rng, Float32), 0.18f0, 0.36f0)
        vₚ = 3.15f0 - 1.35f0 * (ϕ - 0.28f0) + 0.05f0 * randn(rng, Float32)
    else
        # Tight, lower-porosity, higher-velocity population
        ϕ = clamp(0.085f0 + 0.015f0 * randn(rng, Float32), 0.03f0, 0.14f0)
        vₚ = 4.55f0 - 0.85f0 * (ϕ - 0.085f0) + 0.04f0 * randn(rng, Float32)
    end
    return Float32[ϕ, vₚ]
end

n_real = 512
real_data = zeros(Float32, 2, n_real)
for i in 1:n_real
    real_data[:, i] = make_crossplot_sample(rng)
end

μ_cross = mean(real_data, dims = 2)
```

```
σ_cross = std(real_data, dims = 2)
real_data_scaled = (real_data .- μ_cross) ./ σ_cross
```

```
2×512 Matrix{Float32}:
 -1.08862   -1.06215  -0.863432   1.17695  …  -1.0941     0.593133  -0.912383
  0.942664   0.98521   1.04727   -1.1457       1.00004   -1.05781    1.00444
```

```
# Generator: noise → synthetic porosity-velocity pair
latent_dim = 4
generator = Chain(
    Dense(latent_dim => 48, relu),
    Dense(48 => 48, relu),
    Dense(48 => 2)
)

# Critic: porosity-velocity pair → realism score
critic = Chain(
    Dense(2 => 48, leakyrelu),
    Dense(48 => 48, leakyrelu),
    Dense(48 => 1)
)

ps_g, st_g = Lux.setup(rng, generator)
ps_c, st_c = Lux.setup(rng, critic)
```

```
((layer_1 = (weight = Float32[0.095236704 -0.26362634; -0.48108375 -0.13117667; … ; 0.44177726 0.6421332; 0.6
0.12413696], bias = Float32[-0.553358, -0.19415233, 0.111818284, -0.13330719, -0.14170487, 0.0057655205, -
0.22624613, 0.6107927, -0.5914383, 0.14539988  …  -0.31601107, 0.46229714, -0.6209592, -
0.1444649, -0.2987946, -0.1347729, -0.5750111, -0.5215596, 0.5276568, 0.6253284]), layer_2 = (weight = Float3
0.05611117 … 0.12109106 -0.113850355; -0.028847044 -0.11505275 … 0.034280352 -0.13108106; … ; 0.019708203 0.0
0.06890222; 0.027860826 -0.10359403 … -0.12468299 -0.12424605], bias = Float32[0.110834286, -
0.13745223, -0.013768459, -0.13376565, 0.0061205155, -0.09815026, 0.07609777, -0.07705173, 0.08112589, -
0.09739101  …  -0.015835546, 0.029057994, 0.08935784, -0.0706621, -0.061476413, 0.015290982, 0.13887009, -
0.05602218, -0.028790316, -0.10544139]), layer_3 = (weight = Float32[0.09919259 0.1722236 … 0.14712217 -
0.07780698], bias = Float32[0.08056794])), (layer_1 = NamedTuple(), layer_2 = NamedTuple(), layer_3 = NamedTu
```

```
function sample_batch(rng, data, batch_size)
    idx = rand(rng, 1:size(data, 2), batch_size)
    data[:, idx]
end

function clip_params(x, clip_value)
    if x isa NamedTuple
```

```
        names = fieldnames(typeof(x))
        values = map(name -> clip_params(getfield(x, name), clip_value), names)
        return NamedTuple{names}(Tuple(values))
    elseif x isa AbstractArray
        return clamp.(x, -clip_value, clip_value)
    else
        return x
    end
end

function train_wgan(generator, critic, ps_g, st_g, ps_c, st_c;
                    epochs = 700, batch_size = 128, critic_steps = 4,
                    lr_g = 0.00025f0, lr_c = 0.00025f0, clip_value = 0.03f0)
    opt_state_g = Optimisers.setup(RMSProp(lr_g), ps_g)
    opt_state_c = Optimisers.setup(RMSProp(lr_c), ps_c)

    for epoch in 1:epochs
        critic_loss = 0.0f0
        for _ in 1:critic_steps
            x_real = sample_batch(rng, real_data_scaled, batch_size)
            z = randn(rng, Float32, latent_dim, batch_size)

            (c_loss, st_c_new), c_grads = Zygote.withgradient(ps_c) do pc
                fake, _ = generator(z, ps_g, st_g)
                real_score, st_c1 = critic(x_real, pc, st_c)
                fake_score, st_c2 = critic(fake, pc, st_c1)
                loss = mean(fake_score) - mean(real_score)
                return loss, st_c2
            end

            opt_state_c, ps_c = Optimisers.update(opt_state_c, ps_c, c_grads[1])
            ps_c = clip_params(ps_c, clip_value)
            st_c = st_c_new
            critic_loss = c_loss
        end

        z = randn(rng, Float32, latent_dim, batch_size)
        (g_loss, st_g_new), g_grads = Zygote.withgradient(ps_g) do pg
            fake, st_g_ = generator(z, pg, st_g)
            fake_score, _ = critic(fake, ps_c, st_c)
            loss = -mean(fake_score)
            return loss, st_g_
        end

        opt_state_g, ps_g = Optimisers.update(opt_state_g, ps_g, g_grads[1])
        st_g = st_g_new
```

```
        if epoch == 1 || epoch % 140 == 0
            @printf "Epoch %3d  critic loss: %.4f  generator loss: %.4f\n" epoch critic_loss g_loss
        end
    end

    return ps_g, st_g, ps_c, st_c
end

ps_g, st_g, ps_c, st_c = train_wgan(generator, critic, ps_g, st_g, ps_c, st_c)
```

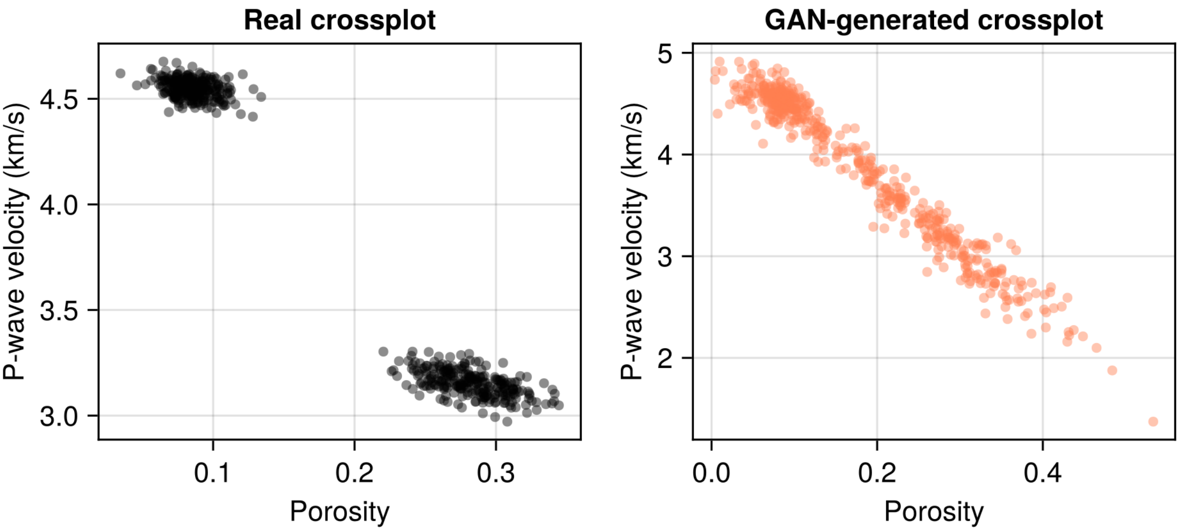
```
Epoch   1  critic loss: -0.0010  generator loss: -0.0293
Epoch 140  critic loss: -0.0033  generator loss: -0.0304
Epoch 280  critic loss: -0.0037  generator loss: -0.0350
Epoch 420  critic loss: -0.0027  generator loss: -0.0358
Epoch 560  critic loss: -0.0030  generator loss: -0.0353
Epoch 700  critic loss: -0.0027  generator loss: -0.0356
```

```
((layer_1 = (weight = Float32[-1.6170778 1.438317 -1.4887156 -0.799762; 0.20049846 -
0.25750396 -0.63647604 -1.4251372; … ; -0.822932 -1.4251498 1.1302763 -1.6529914; -1.0376585 0.26558796 -
0.42028984 1.1279557], bias = Float32[0.11374632, -0.20868967, 0.26450476, -0.36825222, 0.1826905, -
0.36717394, 0.16281371, 0.36403623, 0.4234871, 0.2853451  …  -0.49563542, -0.0097132465, -
0.17194575, 0.12402804, 0.23683684, -0.36284924, 0.24442586, -0.1958218, 0.42159754, -
0.42089146]), layer_2 = (weight = Float32[-0.22104508 0.2708321 … 0.062846504 -0.41944122; 0.22122528 0.18381
0.39359152 -0.16378614; … ; -0.32657 0.2787427 … -0.008122357 -0.069475085; 0.028553978 0.04858848 … -
0.20754774 0.2562649], bias = Float32[0.07640082, -0.023688277, 0.20423998, 0.067435145, -
0.025763195, 0.025153305, 0.030307252, -0.17817481, 0.25480497, -0.122916736  …  0.039165087, 0.09899269, 0.0
0.09638943, -0.09809221, 0.111562625, 0.2355382, 0.09990677, -0.04035821, -0.020403877]), layer_3 = (weight =
0.24831717 -0.095159106 … -0.1811671 0.066351555; 0.0873096 0.18436377 … 0.12655254 -
0.095087424], bias = Float32[0.03987667, 0.018674318])), (layer_1 = NamedTuple(), layer_2 = NamedTuple(), laye
0.03; -0.03 0.03; … ; -0.008723305 0.024386626; 0.01966317 -0.009388254], bias = Float32[0.008675049, -
0.017944403, -0.03, -0.017168913, -0.017156893, 0.026852814, -0.017958269, 0.027898777, 0.008676276, 0.03  …
0.01795752, -0.029777953, -0.01857011, 0.008671484, 0.0086770365, -0.029778061, -0.03, 0.027442524]), layer_2
0.03 -0.028011717; 0.029960053 0.008583459 … -0.03 -0.028023977; … ; 0.029960053 0.008583459 … -
0.03 -0.028023977; -0.029960053 -0.008583461 … 0.03 0.028023977], bias = Float32[0.03, -
0.03, 0.019332852, -0.03, 0.03, -0.03, 0.021491326, -0.028429843, 0.019329412, -0.03  …  -
0.026209906, 0.01933217, 0.03, -0.03, -0.03, 0.03, 0.03, -0.03, -0.03, -0.028429843]), layer_3 = (weight = Fl
0.03], bias = Float32[0.03])), (layer_1 = NamedTuple(), layer_2 = NamedTuple(), layer_3 = NamedTuple()))
```

```
# Generate synthetic crossplot samples
z_test = randn(rng, Float32, latent_dim, n_real)
fake_points, _ = generator(z_test, ps_g, st_g)
fake_points = fake_points .* σ_cross .+ μ_cross

# Simple distribution checks (not sufficient alone, but useful sanity checks)
@printf "Real porosity mean/std: %.3f / %.3f\n" mean(real_data[1, :]) std(real_data[1, :])
```

```
@printf "Fake porosity mean/std: %.3f / %.3f\n" mean(fake_points[1, :]) std(fake_points[1, :])
@printf "Real velocity mean/std: %.3f / %.3f\n" mean(real_data[2, :]) std(real_data[2, :])
@printf "Fake velocity mean/std: %.3f / %.3f\n" mean(fake_points[2, :]) std(fake_points[2, :])

fig = Figure(size = (620, 330))
ax1 = Axis(fig[1, 1], title = "Real crossplot",
           xlabel = "Porosity", ylabel = "P-wave velocity (km/s)")
scatter!(ax1, real_data[1, :], real_data[2, :], color = (:black, 0.45), markersize = 7)

ax2 = Axis(fig[1, 2], title = "GAN-generated crossplot",
           xlabel = "Porosity", ylabel = "P-wave velocity (km/s)")
scatter!(ax2, fake_points[1, :], fake_points[2, :], color = (:coral, 0.45), markersize = 7)

Label(fig[0, :], "GAN: Real vs Generated Rock-Physics Crossplots",
      fontsize = 16)
fig
```

```
Real porosity mean/std: 0.181 / 0.101
Fake porosity mean/std: 0.180 / 0.111
Real velocity mean/std: 3.866 / 0.700
Fake velocity mean/std: 3.867 / 0.738
```



The printed means and standard deviations are only a quick sanity check. The real test is the scatter plot: if the generated cloud matches the two-cluster structure, then the GAN is at least learning the gross geometry of the data distribution. If it collapses to one cluster or one narrow streak, the training has failed even if one or two scalar summary numbers look acceptable.

13.4 When to use GANs

GANs excel at **generating realistic samples** from a learned distribution. They are particularly useful when:

- You need to **augment** limited training data with realistic synthetic examples.
- You want to **sample** from a complex distribution (e.g., geological models consistent with observations).
- You need to **transfer styles** or transform between domains (e.g., converting sketches to realistic geological cross-sections).

GANs can be harder to train than other generative models (VAEs, diffusion models). Mode collapse (the generator produces only a few types of output) and training instability are common challenges.

13.5 Geoscience applications

- **Porous media reconstruction** — Mosser et al. (2017) used DCGANs to generate 3D micro-CT-scale porous media samples that match the statistical properties of real rock samples, enabling rapid generation of representative geological volumes for flow simulation.
- **Geostatistical inversion** — Laloy et al. (2018) used a spatial GAN as a geological prior for inverse problems, generating training-image-consistent geological models during inversion. This allows the inversion to stay within geologically realistic model space.
- **Stochastic seismic inversion** — Mosser et al. (2020) combined GANs with seismic inversion, using the generator as a geological prior to produce multiple subsurface models consistent with both seismic data and geological knowledge.
- **Geological facies modeling** — GANs have been used to generate realistic 2D and 3D geological facies models that honor well-data and geological constraints, as an alternative to traditional geostatistical simulation.

14 Diffusion Models

Key references

- **Thermodynamic diffusion models** — the early formulation that introduced gradual noising and learned reversal as a generative strategy (Sohl-Dickstein et al., 2015).
- **DDPM** — the paper that made diffusion models practical and widely adopted by training a network to predict injected noise (Ho et al., 2020).
- **Score-based diffusion** — the stochastic-differential-equation view that connects diffusion models, score matching, and continuous-time sampling (Y. Song et al., 2021).

A **diffusion model** learns to generate data by reversing a gradual corruption process. We begin with a clean sample $\mathbf{x}_0$, add small amounts of Gaussian noise over many steps until the sample becomes almost pure noise, and then train a neural network to undo that corruption one step at a time.

Conceptually, diffusion models are attractive because they replace one hard generative jump with many easy denoising subproblems. Instead of asking a network to directly map random noise to a realistic geological or physical sample, we ask it a sequence of simpler questions: "given a slightly corrupted sample, what noise was added?" or equivalently "how should this point move to become a little cleaner?"

14.1 The forward noising process

In a discrete diffusion model, the forward process defines a Markov chain:

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}\left(\sqrt{1-\beta_t}\,\mathbf{x}_{t-1},\ \beta_t I\right)$$

where β_t is a small variance schedule. Repeating this many times produces progressively noisier states $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T$.

An especially useful closed-form expression is:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\epsilon, \qquad \epsilon \sim \mathcal{N}(0, I)$$

with $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$. This says that every noisy sample is just a weighted combination of the clean sample and Gaussian noise.

14.2 The learned reverse process

The reverse process is parameterized by a neural network, often written as $\epsilon_\theta(\mathbf{x}_t, t)$, that predicts the noise contained in a noisy sample. Training minimizes a simple mean-squared error:

$$\mathcal{L}(\theta) = \mathbb{E}_{\mathbf{x}_0, t, \epsilon}\left[\|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|_2^2\right]$$

This objective is one of the main reasons diffusion models are stable: they reduce generative modeling to supervised regression on synthetic noise-corruption pairs.

At sampling time, we start from Gaussian noise and repeatedly apply the learned denoising update from $t = T$ back to $t = 1$. Each reverse step is small, but together they transform noise into a sample drawn from the learned distribution.

14.3 Code example: learning a 1D porosity prior with a diffusion model

We use a tiny diffusion model to learn a bimodal 1D porosity distribution representing two simplified rock populations. The point of the example is not image generation; it is to show the core workflow on a distribution that is easy to visualize. The network input is a noisy porosity value together with the diffusion time step, and the output is the predicted injected noise.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)

function sample_porosity(rng, n)
    values = zeros(Float32, n)
    for i in 1:n
        if rand(rng) < 0.55f0
            values[i] = clamp(0.27f0 + 0.025f0 * randn(rng, Float32), 0.16f0, 0.36f0)
        else
            values[i] = clamp(0.09f0 + 0.015f0 * randn(rng, Float32), 0.03f0, 0.14f0)
        end
    end
    return reshape(values, 1, :)
end

n_data = 768
x0_data = sample_porosity(rng, n_data)
μ_data = mean(x0_data)
σ_data = std(x0_data)
x0_scaled = (x0_data .- μ_data) ./ σ_data

n_steps = 40
β = Float32.(range(1.0f-4, 0.03f0, length = n_steps))
α = 1.0f0 .- β
```

```
ᾱ = accumulate(*, α)
```

```
40-element Vector{Float32}:
 0.9999
 0.9990334
 0.9974016
 0.9950079
 0.991857
 0.9879557
 0.9833123
 0.9779369
 0.9718411
 0.9650382
 ⋮
 0.67936
 0.6626251
 0.6457944
 0.6288961
 0.61195785
 0.5950066
 0.57806873
 0.56116986
 0.54433477
```

```
# Tiny MLP: noisy porosity and normalized time step → predicted noise
denoiser = Chain(
    Dense(2 => 32, relu),
    Dense(32 => 32, relu),
    Dense(32 => 1)
)

ps, st = Lux.setup(rng, denoiser)
opt_state = Optimisers.setup(Adam(0.005f0), ps)

function diffusion_loss(ps, x0_batch, t_idx, ε)
    ᾱ_t = reshape(ᾱ[t_idx], 1, :)
    x_t = sqrt.(ᾱ_t) .* x0_batch .+ sqrt.(1 .- ᾱ_t) .* ε
    t_feature = reshape(Float32.(t_idx ./ n_steps), 1, :)
    inputs = vcat(x_t, t_feature)
    ε̂ , _ = denoiser(inputs, ps, st)
    return mean((ε̂  .- ε) .^ 2)
end
```

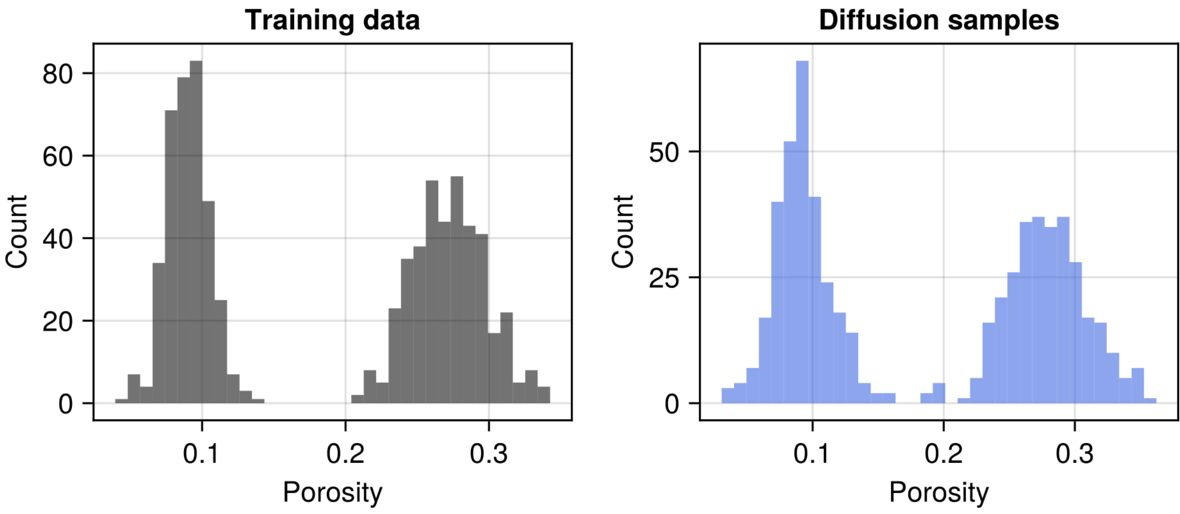
```
diffusion_loss (generic function with 1 method)
```

```
batch_size = 128

for epoch in 1:400
    idx = rand(rng, 1:size(x0_scaled, 2), batch_size)
    x0_batch = x0_scaled[:, idx]
    t_idx = rand(rng, 1:n_steps, batch_size)
    ε = randn(rng, Float32, 1, batch_size)

    loss, grads = Zygote.withgradient(ps) do p
        diffusion_loss(p, x0_batch, t_idx, ε)
    end

    opt_state, ps = Optimisers.update(opt_state, ps, grads[1])

    if epoch == 1 || epoch % 100 == 0
        @printf "Epoch %3d  diffusion loss = %.6f\n" epoch loss
    end
end
```

```
Epoch   1  diffusion loss = 1.627333
Epoch 100  diffusion loss = 0.544971
Epoch 200  diffusion loss = 0.417677
Epoch 300  diffusion loss = 0.594249
Epoch 400  diffusion loss = 0.632698
```

```
# Reverse sampling from Gaussian noise
function predict_noise(x, step, ps)
    t_feature = fill(Float32(step / n_steps), 1, size(x, 2))
    inputs = vcat(x, t_feature)
    ε̂ , _ = denoiser(inputs, ps, st)
    return ε̂
end

n_samples = 600
x = randn(rng, Float32, 1, n_samples)

for step in n_steps:-1:1
    ε̂  = predict_noise(x, step, ps)
    α_t = α[step]
    β_t = β[step]
    ᾱ_t = ᾱ[step]
    z = step > 1 ? randn(rng, Float32, size(x)) : zeros(Float32, size(x))
    x = (x .- (β_t / sqrt(1.0f0 - ᾱ_t)) .* ε̂ ) ./ sqrt(α_t)
    if step > 1
        x .+= sqrt(β_t) .* z
    end
```

```
end

generated_porosity = clamp.(x .* σ_data .+ μ_data, 0.0f0, 0.4f0)

fig = Figure(size = (620, 320))
ax1 = Axis(fig[1, 1], title = "Training data",
           xlabel = "Porosity", ylabel = "Count")
hist!(ax1, vec(x0_data), bins = 35, color = (:black, 0.55))

ax2 = Axis(fig[1, 2], title = "Diffusion samples",
           xlabel = "Porosity", ylabel = "Count")
hist!(ax2, vec(generated_porosity), bins = 35, color = (:royalblue, 0.6))

Label(fig[0, :], "Diffusion model: learned 1D porosity prior", fontsize = 16)
fig
```



The generated histogram should reproduce the two main porosity modes, even if it is imperfect. That is enough to make the core point: diffusion models learn a distribution by repeatedly solving small denoising problems rather than a single direct generation problem.

14.4 When to use diffusion models

Diffusion models are especially attractive when:

- You care about **sample quality and distributional realism** more than single-shot speed.
- The target distribution is **multimodal**, so a single deterministic prediction would be misleading.
- You want a **learned prior** that can later be conditioned on observations in an inverse problem.

- Training stability matters more than the fastest possible sampling.

Their main drawback is sampling cost: a diffusion model typically needs many denoising steps to generate one sample. That is often acceptable for offline uncertainty quantification, but it can be limiting inside large inverse loops unless acceleration tricks are used.

14.5 Geoscience applications

- **Seismic interpolation and denoising** — diffusion models are well matched to reconstructing missing or corrupted wavefield content because they learn realistic structure rather than only pointwise averages.
- **Learned priors for inverse problems** — diffusion sampling can generate ensembles of geologically plausible subsurface models that are later conditioned on travel-time, waveform, or reservoir observations.
- **Geomodel and facies generation** — diffusion models can represent multimodal geological uncertainty for channels, facies maps, and permeability fields without collapsing to a single realization.
- **Remote sensing and Earth observation** — cloud removal, gap filling, downscaling, and super-resolution are natural conditional-generation tasks for diffusion models on spatial fields.
- **Uncertainty-aware surrogate workflows** — because diffusion models generate ensembles rather than point estimates, they are attractive wherever geoscience decisions need uncertainty bands rather than one best model.
- **Overview** — these applications fit the broader shift toward probabilistic, data-driven geoscience modeling reviewed in Bergen et al. (2019) and Dramsch (2020).

15 Flow Matching

Key references

- **Neural ODEs** — the continuous-time viewpoint where a neural network parameterizes a vector field and data evolves through an ordinary differential equation (R. T. Q. Chen et al., 2018).
- **Continuous normalizing flows** — scalable continuous-time generative flows that connect latent variables to data through learned dynamics (Grathwohl et al., 2019).
- **Flow matching** — the modern training objective that learns a transport vector field directly from simple reference paths between noise and data (Lipman et al., 2023).

A **flow-matching model** learns a continuous vector field that transports samples from a simple base distribution, usually Gaussian noise, to the target data distribution. Instead of adding noise and then reversing it as in diffusion models, flow matching learns the velocity field of a probability flow.

This perspective is useful because it turns generative modeling into a transport problem. If we know how samples should move at every time $t \in [0, 1]$, then generation becomes a matter of integrating an ordinary differential equation from noise to data.

15.1 The basic idea

Suppose $\mathbf{x}_0 \sim p_0$ is a simple reference sample and $\mathbf{x}_1 \sim p_1$ is a target data sample. We define an interpolation path between them, for example the straight-line path

$$\mathbf{x}_t = (1 - t)\,\mathbf{x}_0 + t\,\mathbf{x}_1, \qquad t \in [0, 1].$$

For this path, the ideal velocity is simply

$$\mathbf{u}_t = \frac{d\mathbf{x}_t}{dt} = \mathbf{x}_1 - \mathbf{x}_0.$$

Flow matching trains a neural network $\mathbf{v}_\theta(\mathbf{x}, t)$ to predict that velocity from points sampled along the path. A simple objective is

$$\mathscr{L}(\theta) = \mathbb{E}_{\mathbf{x}_0, \mathbf{x}_1, t}\left[\|\mathbf{v}_\theta(\mathbf{x}_t, t) - \mathbf{u}_t\|_2^2\right].$$

After training, generation solves the ODE

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}_\theta(\mathbf{x}, t), \qquad \mathbf{x}(0) \sim p_0.$$

The learned dynamics push the base noise distribution toward the data distribution.

15.2 Why flow matching is interesting

Compared with diffusion models, flow matching often gives a cleaner conceptual picture for inverse problems and conditional generation:

- the model is a **deterministic transport map** rather than a stochastic reverse Markov chain,
- sampling can use standard ODE solvers,
- and the conditioning logic fits naturally with transport from a prior toward an observation-consistent posterior.

In practice, diffusion and flow matching are closely related. Both learn time-dependent transformations from simple noise to complex data. The difference is mostly in whether the learned process is framed as denoising a stochastic corruption or integrating a deterministic flow.

15.3 Code example: transporting Gaussian noise into a bimodal porosity prior

We reuse the same synthetic porosity distribution as in the diffusion chapter, but now train a velocity network directly. The input is a point on the interpolation path together with the time t, and the output is the velocity that should move that point toward the target distribution.

```julia
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)

function sample_porosity(rng, n)
    values = zeros(Float32, n)
    for i in 1:n
        if rand(rng) < 0.55f0
            values[i] = clamp(0.27f0 + 0.025f0 * randn(rng, Float32), 0.16f0, 0.36f0)
        else
            values[i] = clamp(0.09f0 + 0.015f0 * randn(rng, Float32), 0.03f0, 0.14f0)
        end
    end
    return reshape(values, 1, :)
end

n_data = 768
x1_data = sample_porosity(rng, n_data)
μ_data = mean(x1_data)
σ_data = std(x1_data)
x1_scaled = (x1_data .- μ_data) ./ σ_data
```

```
1×768 Matrix{Float32}:
 -1.16637  0.702224  -1.03697  -0.924098  …  1.36255  -1.12927  -0.992214
```

```julia
# Tiny MLP: point on the path and time → velocity
velocity_net = Chain(
    Dense(2 => 32, tanh),
    Dense(32 => 32, tanh),
    Dense(32 => 1)
)

ps, st = Lux.setup(rng, velocity_net)
opt_state = Optimisers.setup(Adam(0.005f0), ps)

function flow_matching_loss(ps, x0_batch, x1_batch, t_batch)
    x_t = (1 .- t_batch) .* x0_batch .+ t_batch .* x1_batch
    u_t = x1_batch .- x0_batch
    inputs = vcat(x_t, t_batch)
    v̂ , _ = velocity_net(inputs, ps, st)
    return mean((v̂  .- u_t) .^ 2)
end
```

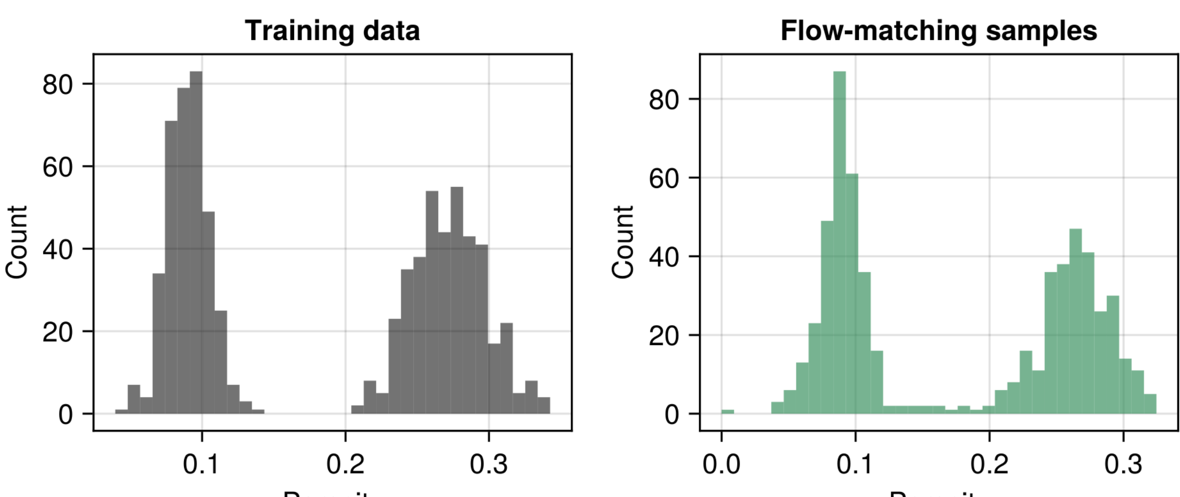
```
flow_matching_loss (generic function with 1 method)
```

```julia
batch_size = 128

for epoch in 1:400
    idx = rand(rng, 1:size(x1_scaled, 2), batch_size)
    x1_batch = x1_scaled[:, idx]
    x0_batch = randn(rng, Float32, 1, batch_size)
    t_batch = rand(rng, Float32, 1, batch_size)

    loss, grads = Zygote.withgradient(ps) do p
        flow_matching_loss(p, x0_batch, x1_batch, t_batch)
    end

    opt_state, ps = Optimisers.update(opt_state, ps, grads[1])

    if epoch == 1 || epoch % 100 == 0
        @printf "Epoch %3d  flow-matching loss = %.6f\n" epoch loss
    end
end
```

```
Epoch   1  flow-matching loss = 2.732322
Epoch 100  flow-matching loss = 1.153513
```

```
Epoch 200  flow-matching loss = 1.656029
Epoch 300  flow-matching loss = 1.500055
Epoch 400  flow-matching loss = 1.196840
```

```
# Sample by integrating the learned ODE from t = 0 to t = 1
function velocity_prediction(x, t, ps)
    t_feature = fill(Float32(t), 1, size(x, 2))
    inputs = vcat(x, t_feature)
    v̂ , _ = velocity_net(inputs, ps, st)
    return v̂
end

n_samples = 600
x = randn(rng, Float32, 1, n_samples)
n_solver_steps = 60
dt = 1.0f0 / n_solver_steps

for step in 0:n_solver_steps-1
    t = step * dt
    x .+= dt .* velocity_prediction(x, t, ps)
end

generated_porosity = clamp.(x .* σ_data .+ μ_data, 0.0f0, 0.4f0)

fig = Figure(size = (620, 320))
ax1 = Axis(fig[1, 1], title = "Training data",
           xlabel = "Porosity", ylabel = "Count")
hist!(ax1, vec(x1_data), bins = 35, color = (:black, 0.55))

ax2 = Axis(fig[1, 2], title = "Flow-matching samples",
           xlabel = "Porosity", ylabel = "Count")
hist!(ax2, vec(generated_porosity), bins = 35, color = (:seagreen, 0.65))

Label(fig[0, :], "Flow matching: transport from Gaussian noise to porosity prior", fontsize = 16)
fig
```



The learned histogram should approximate the two porosity modes again, but the sampling mechanism is now different: there is no reverse noise-removal chain. We simply integrate a learned velocity field from noise to data.

15.4 When to use flow matching

Flow matching is especially appealing when:

- You want a **continuous-time transport view** of generation.
- You expect to reuse the model inside **conditioning, inversion, or data-assimilation** workflows.
- Deterministic ODE-based sampling is easier to reason about than stochastic reverse diffusion.
- You care about **faster generation**, because well-trained flows can often be sampled with fewer solver steps than diffusion models need denoising steps.

The main tradeoff is that the model must learn a good global velocity field. If that field is poor, ODE integration can drift into unrealistic regions of state space.

15.5 Geoscience applications

- **Posterior transport for inverse problems** — flow matching is a natural way to move samples from a simple prior toward complex posterior ensembles in ill-posed geophysical inference.
- **Conditional geological generation** — deterministic transport maps can generate facies, permeability, or structural models conditioned on sparse control data while preserving multimodality.
- **Fast sampling inside iterative workflows** — because flow models can often use relatively few solver steps, they are attractive when generation must be repeated many times inside inversion or uncertainty-propagation loops.

- **Data assimilation and field reconstruction** — flow matching provides a clean framework for transporting coarse or noisy fields toward high-resolution, observation-consistent realizations.
- **Bridging latent priors and physics-aware models** — the transport viewpoint fits naturally with later scientific machine learning chapters, where priors, constraints, and inverse objectives are combined.
- **Overview** — flow matching is newer than GANs and diffusion in geoscience, but it is well aligned with the uncertainty-aware and inverse-problem-driven perspective emphasized in Bergen et al. (2019) and Dramsch (2020).

Part III

Scientific Machine Learning

16 Inverse Modeling

Up to this point, the book has focused on neural networks as flexible function approximators. In many geoscience problems, though, the real task is not just to predict an output from an input. The real task is to recover something hidden: a velocity model from seismic data, a conductivity field from electromagnetic measurements, or a forcing term from sparse observations.

That is an **inverse problem**. We observe consequences and work backward to infer the cause.

Inverse problems are central to geoscience because the Earth is mostly inaccessible. We rarely observe the subsurface, the mantle, or the full atmospheric state directly. Instead, we measure indirect signals and try to reconstruct the underlying system. This is difficult for two reasons: the governing physics is often expensive to solve, and the inverse map is usually unstable, non-unique, or both.

Scientific machine learning becomes useful exactly here. It gives us ways to combine flexible neural representations with known physical structure, so that learning is guided by more than data alone.

16.1 Forward problems versus inverse problems

A **forward problem** starts with a model and predicts observations. For example, if you know the seismic velocity model, you can simulate travel times or waveforms.

An **inverse problem** goes the other way. You start with observations and try to infer the model that could have produced them.

In compact notation:

$$extforward: \quad m \longmapsto d$$

$$extinverse: \quad d \longmapsto m$$

where m denotes the unknown model parameters and d denotes the observed data.

The forward map is usually the easier direction. The inverse map is harder because several different models may explain the same data, and small errors in the data can produce large changes in the recovered model. That is why inverse problems need regularization, prior knowledge, or physical constraints.

16.2 Why scientific machine learning helps

Classical inversion already uses physics. That part is not new. What SciML changes is the representation and the optimization strategy.

Instead of representing the unknown model on a fixed mesh with many independent cells, we can represent it with a neural network. Instead of training only against labeled pairs, we can also penalize PDE residuals, boundary conditions, and other scientific structure. Instead of learning a single solution, we can sometimes learn an entire operator that maps one function to another.

This leads to three recurring ideas in the chapters that follow:

- **Physics-informed training**: the governing equation becomes part of the loss.
- **Implicit neural representations**: the unknown field is represented as a continuous neural function.
- **Operator learning**: the model learns a mapping between whole functions, not just between finite-dimensional vectors.

These are different tools, not competing slogans. The right one depends on whether you want to solve one PDE instance, recover unknown parameters, or accelerate many repeated forward solves.

16.3 The common template

Despite the variety of methods, most scientific machine-learning problems in this part fit the same high-level template:

1. Represent an unknown field, state, or operator with trainable parameters θ.
2. Use observations, physical constraints, or both to define a loss $\mathscr{L}(\theta)$.
3. Optimize the parameters so the model matches the data while respecting the governing structure.

Written abstractly,

$$heta^{\star} = \arg\min_{\theta} \mathscr{L}(\theta)$$

The details change from chapter to chapter, but the pattern stays recognizable.

16.4 What this part covers

The next five chapters develop the main SciML ideas used throughout current geoscience work:

- Physics-Informed Neural Networks shows how to solve a PDE by training a network to satisfy the governing equation.
- Physics-Based ML for Inversion focuses on recovering unknown physical-property fields from data while keeping the physics in the loop.
- DeepONet introduces operator learning through a branch-trunk architecture.
- Physics-Informed DeepONet combines operator learning with PDE constraints.

- Fourier Neural Operator moves operator learning into Fourier space for fast surrogate modeling on regular grids.

Together, these chapters cover the main conceptual shift from standard machine learning to scientific machine learning: we are no longer learning only from examples. We are learning in a setting where equations, geometry, conservation laws, and physical plausibility all matter.

16.5 What to keep in mind

It is easy to get distracted by new acronyms in this area. A better filter is to keep asking four practical questions:

- What object is being learned: a solution, a material property, or an operator?
- Where does the supervision come from: data, physics, or both?
- What assumptions are built into the parameterization?
- How will the result be checked against scientific reality, not just training loss?

Those questions will keep the chapter sequence grounded, and they matter much more than whether a method has a fashionable name.

16.6 Summary

Inverse problems sit at the center of geoscience because we infer hidden structure from indirect observations. Scientific machine learning does not replace the underlying physics. It gives us new ways to represent unknowns, impose constraints, and make repeated forward and inverse calculations practical. The next chapter starts with the most direct example: PINNs.

17 Physics-Informed Neural Networks

Key references

- **Early neural-network PDE solvers** — using neural networks to solve ODEs and PDEs by embedding the equation in the loss function (Lagaris et al., 1998).
- **Physics-Informed Neural Networks (PINNs)** — the foundational framework that popularized training neural networks with PDE residuals as soft constraints (Raissi et al., 2019).
- **PINN review** — comprehensive review of physics-informed machine learning covering PINNs, operator learning, and hybrid methods (Karniadakis et al., 2021).
- **DeepXDE** — a widely used library for PINNs that formalized many practical training strategies (Lu, Meng, et al., 2021).
- **Gradient pathologies** — understanding and mitigating the training difficulties unique to PINNs (S. Wang, Teng, et al., 2021).
- **Neural ODEs** — treating the forward pass of a neural network as an ODE, connecting deep learning and differential equations (R. T. Q. Chen et al., 2018).
- **Universal differential equations** — embedding neural networks inside differential equations to learn unknown physics (Rackauckas et al., 2020).

In the previous part of this book we treated neural networks as function approximators: given input–output pairs, the network learns to map one to the other. **Physics-informed neural networks** (PINNs) take a fundamentally different approach. Instead of (or in addition to) learning from data, the network is trained to satisfy a known physical law — typically a partial differential equation (PDE). The physics itself becomes part of the loss function.

17.1 The idea

Suppose we want to solve the PDE:

$$\mathcal{N}[u](\mathbf{x}, t) = 0, \qquad \mathbf{x} \in \Omega, \quad t \in [0, T]$$

with boundary conditions $\mathcal{B}[u] = 0$ on $\partial\Omega$ and initial conditions $u(\mathbf{x}, 0) = u_0(\mathbf{x})$.

A PINN approximates the solution $u(\mathbf{x}, t)$ with a neural network $u_\theta(\mathbf{x}, t)$, where θ denotes the trainable parameters. The network takes coordinates $(\mathbf{x}, t)$ as input and outputs the predicted solution value.

The key insight is that because the network is differentiable, we can compute $\frac{\partial u_\theta}{\partial t}$, $\frac{\partial^2 u_\theta}{\partial x^2}$, etc. using automatic differentiation. We then define the **PDE residual**:

$$r_\theta(\mathbf{x}, t) = \mathcal{N}[u_\theta](\mathbf{x}, t)$$

and train the network to make this residual zero everywhere.

17.2 The PINN loss function

The total loss combines three named components:

$$\mathscr{L}(\theta) = \lambda_r \mathscr{L}_r(\theta) + \lambda_b \mathscr{L}_{\mathrm{bc}}(\theta) + \lambda_0 \mathscr{L}_0(\theta)$$

with

$$\mathscr{L}_r(\theta) = \frac{1}{N_r} \sum_{i=1}^{N_r} |r_\theta(\mathbf{x}_i^r, t_i^r)|^2$$

$$\mathscr{L}_{\mathrm{bc}}(\theta) = \frac{1}{N_b} \sum_{j=1}^{N_b} |\mathscr{B}[u_\theta](\mathbf{x}_j^b, t_j^b)|^2$$

$$\mathscr{L}_0(\theta) = \frac{1}{N_0} \sum_{k=1}^{N_0} |u_\theta(\mathbf{x}_k^0, 0) - u_0(\mathbf{x}_k^0)|^2$$

where:

- $\{(\mathbf{x}_i^r, t_i^r)\}$ are **collocation points** sampled inside the domain where the PDE must hold.
- $\{(\mathbf{x}_j^b, t_j^b)\}$ are points on the boundary.
- $\{(\mathbf{x}_k^0)\}$ are points at $t = 0$.
- $\lambda_r, \lambda_b, \lambda_0$ are weighting factors.

If observational data is also available, a fourth term $\mathscr{L}_{\mathrm{data}}$ can be added, making the PINN a hybrid physics-data model.

17.3 Code example: solving the 1D heat equation

We solve the heat equation $\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$ on $x \in [0, 1]$, $t \in [0, 1]$ with initial condition $u(x, 0) = \sin(\pi x)$ and boundary conditions $u(0, t) = u(1, t) = 0$. The exact solution is $u(x, t) = e^{-\alpha \pi^2 t} \sin(\pi x)$.

```julia
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)
α = 0.01f0   # thermal diffusivity

# Exact solution for comparison
u_exact(x, t) = exp(-α * π^2 * t) * sin(π * x)
```

```
# Sample collocation points (interior), boundary points, and initial points
n_interior = 1000
n_bc = 200
n_ic = 200

# Interior points: random (x, t) in (0,1) × (0,1)
x_int = rand(rng, Float32, 1, n_interior)
t_int = rand(rng, Float32, 1, n_interior)

# Boundary: x = 0 and x = 1
t_bc = rand(rng, Float32, 1, n_bc)
x_bc_0 = zeros(Float32, 1, n_bc)
x_bc_1 = ones(Float32, 1, n_bc)

# Initial condition: t = 0
x_ic = rand(rng, Float32, 1, n_ic)
u_ic = sin.(π .* x_ic)
```

```
# Build the PINN: input (x, t) -> output u
model = Chain(
    Dense(2 => 32, tanh),
    Dense(32 => 32, tanh),
    Dense(32 => 32, tanh),
    Dense(32 => 1)
)

ps, st = Lux.setup(rng, model)
```

```
#-----PINN loss------
function pinn_loss(model, ps, st, data)
    x_i, t_i, x_b0, x_b1, t_b, x_ic_, u_ic_ = data
    λᵣ = 1.0f0
    λᵦ = 10.0f0
    λ₀ = 10.0f0

    #-----PDE residual------
    function u_pred(xt)
        input = reshape(xt, 2, 1)
        out, _ = model(input, ps, st)
        return out[1]
    end

    residuals = zeros(Float32, n_interior)
    for i in 1:n_interior
```

```
        xt = Float32[x_i[i], t_i[i]]
        grad1 = Zygote.gradient(x -> u_pred(x), xt)[1]
        du_dt = grad1[2]
        d2u_dx2 = Zygote.gradient(x -> Zygote.gradient(y -> u_pred(y), x)[1][1], xt)[1][1]
        residuals[i] = du_dt - α * d2u_dx2
    end

    residual = reshape(residuals, 1, :)
    loss_pde = mean(residual .^ 2)

    #-----boundary loss------
    bc_input_0 = vcat(x_b0, t_b)
    bc_input_1 = vcat(x_b1, t_b)
    u_b0, _ = model(bc_input_0, ps, st)
    u_b1, _ = model(bc_input_1, ps, st)
    loss_bc = mean(u_b0 .^ 2) + mean(u_b1 .^ 2)

    #-----initial-condition loss------
    ic_input = vcat(x_ic_, zeros(Float32, 1, size(x_ic_, 2)))
    u_0, _ = model(ic_input, ps, st)
    loss_ic = mean((u_0 .- u_ic_) .^ 2)

    total = λᵣ * loss_pde + λᵦ * loss_bc + λ₀ * loss_ic
    return total, st, ()
end
```

```
# Train the PINN
opt = Adam(0.001f0)
tstate = Training.TrainState(model, ps, st, opt)

data = (x_int, t_int, x_bc_0, x_bc_1, t_bc, x_ic, u_ic)

for epoch in 1:500
    _, loss, _, tstate = Training.single_train_step!(
        AutoZygote(), pinn_loss, data, tstate
    )
    if epoch == 1 || epoch % 100 == 0
        @printf "Epoch %4d  Loss = %.6f\n" epoch loss
    end
end
```

```
# Compare PINN solution to exact solution
nx, nt = 50, 50
xs = Float32.(range(0, 1, length = nx))
ts = Float32.(range(0, 1, length = nt))
```

```
U_pinn = zeros(Float32, nx, nt)
U_true = zeros(Float32, nx, nt)

for (j, tj) in enumerate(ts)
    input = vcat(reshape(xs, 1, :), fill(tj, 1, nx))
    pred, _ = model(input, tstate.parameters, tstate.states)
    U_pinn[:, j] = vec(pred)
    U_true[:, j] = u_exact.(xs, tj)
end

fig = Figure(size = (700, 300))
ax1 = Axis(fig[1, 1], title = "PINN solution", xlabel = "x", ylabel = "t")
heatmap!(ax1, xs, ts, U_pinn, colormap = :viridis)
ax2 = Axis(fig[1, 2], title = "Exact solution", xlabel = "x", ylabel = "t")
heatmap!(ax2, xs, ts, U_true, colormap = :viridis)
ax3 = Axis(fig[1, 3], title = "Absolute error", xlabel = "x", ylabel = "t")
heatmap!(ax3, xs, ts, abs.(U_pinn .- U_true), colormap = :hot)
fig
```

17.4 PINN strengths and limitations

Strengths:

- **Mesh-free** — no need to build a computational mesh. The network is evaluated at arbitrary points.
- **Inverse problems** — if some physical parameters are unknown, they can be made trainable alongside the network weights. The PINN then simultaneously solves the PDE and infers the unknown parameters.
- **Data assimilation** — observational data can be incorporated by adding a data-fitting term to the loss, making PINNs natural for problems where sparse data and physics must both be honored.

Limitations:

- **Training difficulty** — PINNs can be hard to train, especially for stiff PDEs, multi-scale problems, or high-frequency solutions. Gradient pathologies and loss balancing are active research challenges (S. Wang, Teng, et al., 2021).
- **Computational cost** — computing second-order derivatives through automatic differentiation is expensive. For problems where a traditional numerical solver is efficient, PINNs may not be competitive.
- **Generalization** — a trained PINN solves **one specific instance** of a PDE (one set of boundary/initial conditions). To solve the same PDE with different conditions, you must retrain. This is where **neural operators** (next chapter) offer an advantage.

17.5 Geoscience applications

PINNs have gained significant traction in geoscience because many problems naturally combine sparse observational data with known physical laws:

- **Seismic wave propagation** — Waheed et al. (2021) developed PINNeik, a PINN that solves the eikonal equation for seismic travel-time computation with a mesh-free, differentiable formulation. C. Song et al. (2021) extended PINNs to solve the frequency-domain acoustic wave equation for anisotropic media.
- **Seismic travel times** — Smith et al. (2021) trained EikoNet, a deep neural network constrained by the eikonal equation, to predict seismic travel times in complex 3D velocity models.
- **Seismic inversion** — Yang & Ma (2019) used deep learning for full-waveform inversion, embedding physical constraints to improve velocity model estimation from seismic data.
- **Subsurface flow** — Y. Zhu et al. (2019) applied physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification in subsurface transport problems, demonstrating that PINNs can work without labeled data when the governing PDE is known.
- **Geophysics overview** — Alkhalifah et al. (2022) provides a review of machine-learning methods for geophysics with emphasis on physics-informed approaches, covering seismic imaging, inversion, and interpretation.
- **Weather and climate** — Kashinath et al. (2021) presents case studies of physics-informed machine learning for weather and climate modeling, demonstrating how physical constraints improve forecast accuracy and physical consistency.

18 Physics-Based ML for Inversion

Key references

- **Implicit neural representations (INRs)** — coordinate-based networks that represent continuous signals, originally popularized for 3D scene reconstruction (Mildenhall et al., 2022; Sitzmann et al., 2020).
- **Fourier feature encodings** — overcoming spectral bias of MLPs by mapping inputs through Fourier features, enabling learning of high-frequency detail (Tancik et al., 2020).
- **PINNs for inversion** — embedding PDE constraints in the loss to simultaneously solve forward and inverse problems (Raissi et al., 2019).
- **Physics-informed review** — broad treatment of physics-informed ML including PDE-constrained optimization and hybrid approaches (Karniadakis et al., 2021).
- **Self-adaptive PINNs** — learning the loss-weighting factors alongside the network parameters to balance PDE residual and data terms (McClenny & Braga-Neto, 2023).

In the previous chapter we used PINNs to **solve** a PDE: the network represented the solution field and was trained purely from the governing equation. Here we tackle a harder and more practically important problem: **geophysical inversion**. The goal is to recover an unknown physical property (e.g. subsurface conductivity, seismic velocity, or density) from measured data, while honoring a known PDE that relates the property to the observations.

The idea is simple but powerful: represent the unknown property field as a **neural network** (an implicit neural representation, or INR), and train it by minimizing the PDE residual together with a data-misfit term.

18.1 Why not just use a PINN?

A standard PINN parameterizes the **solution** $u(\mathbf{x})$ as a network and trains it to satisfy the PDE. In an inverse problem, both the solution u and the material property $m(\mathbf{x})$ are unknown. We observe noisy data d^{obs} at a few measurement locations, and we know the PDE:

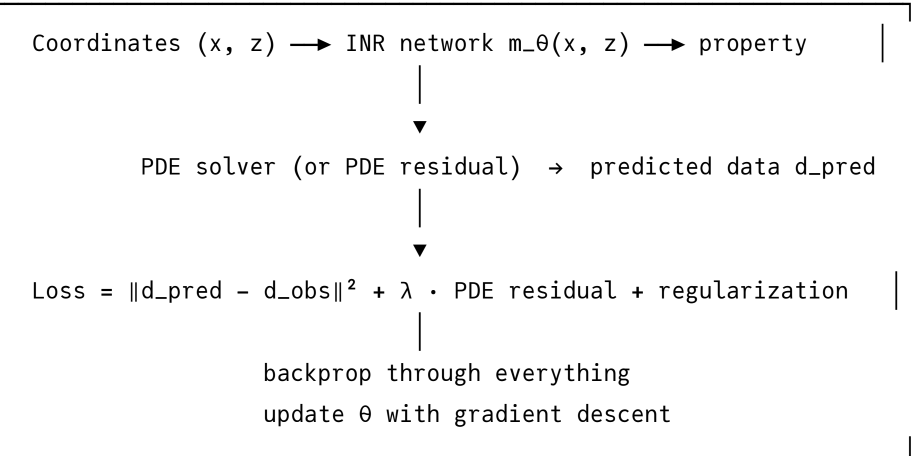
$$\mathcal{N}[u; m] = 0 \quad \text{in } \Omega$$

A physics-based ML inversion parameterizes the **model** $m_\theta(\mathbf{x})$ as a neural network and computes the PDE solution (or its residual) as part of the training loop. This approach has two key advantages:

1. **Regularization through architecture** — the network implicitly regularizes the solution because it can only represent smooth functions (spectral bias), eliminating the need for explicit Tikhonov-style regularization.

2. **Mesh-free, continuous representation** — the recovered model is a continuous function that can be queried at any point, not tied to a fixed grid.

18.2 The inversion framework



The total loss is:

$$\mathscr{L}(\theta) = \mathscr{L}_{\text{data}}(\theta) + \lambda_r \mathscr{L}_r(\theta)$$

where

$$\mathscr{L}_{\text{data}}(\theta) = \frac{1}{N_d} \sum_{i=1}^{N_d} \left(d_i^{\text{pred}}(\theta) - d_i^{\text{obs}}\right)^2$$

$$\mathscr{L}_r(\theta) = \frac{1}{N_c} \sum_{j=1}^{N_c} \left|\mathcal{N}[u; m_\theta](\mathbf{x}_j)\right|^2$$

In this chapter I use d^{obs} and d^{pred} instead of y and $\hat{y}$ because the distinction between measured and predicted data is part of the geophysical setup, not just generic regression notation.

18.3 Implicit neural representations (INRs)

An INR is simply a coordinate-based MLP:

$$m_\theta : \mathbb{R}^d \to \mathbb{R}, \quad (x_1, \ldots, x_d) \mapsto m_\theta(x_1, \ldots, x_d)$$

The network takes spatial coordinates as input and outputs the physical property at that location. By construction, the output is a continuous, differentiable function of position.

18.3.1 Spectral bias and positional encoding

Plain MLPs suffer from **spectral bias** — they learn low-frequency components first and struggle with high-frequency detail (Tancik et al., 2020). For geophysical models with sharp boundaries or fine-scale structure, this is a problem.

Fourier feature encoding addresses this by mapping input coordinates through sinusoidal functions before passing them to the network:

$$\gamma(\mathbf{x}) = \left[\sin(2\pi\mathbf{B}\mathbf{x}),\ \cos(2\pi\mathbf{B}\mathbf{x})\right]$$

where $\mathbf{B} \in \mathbb{R}^{m\times d}$ is a random matrix (sampled once and fixed). The frequency scale of $\mathbf{B}$ controls the spatial resolution the network can represent.

18.4 Code example: 1D steady-state diffusion inversion

We solve a toy inverse problem: recover a spatially varying diffusion coefficient $\kappa(x)$ from noisy observations of the solution field $u(x)$.

Forward problem (steady-state diffusion):

$$-\frac{d}{dx}\left[\kappa(x)\,\frac{du}{dx}\right] = f(x), \quad x \in [0, 1]$$

with $u(0) = 0$, $u(1) = 0$ and a known source $f(x) = \sin(2\pi x)$.

Inverse problem: Given noisy measurements of u at a few sensor locations, recover $\kappa(x)$.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)
```

```
# True diffusion coefficient (what we want to recover)
κ_true(x) = 1.0f0 + 0.5f0 * sin(2π * x)

# Source term
f_source(x) = sin(2π * x)

# Generate "observed" data by solving the ODE numerically (finite differences)
nx_fd = 200
x_fd = Float32.(range(0, 1, length = nx_fd))
dx_fd = x_fd[2] - x_fd[1]

# Build FD system: -d/dx[κ du/dx] = f
# Using central differences
A = zeros(Float32, nx_fd, nx_fd)
```

```
rhs = zeros(Float32, nx_fd)

A[1, 1] = 1.0f0   # u(0) = 0
A[end, end] = 1.0f0  # u(1) = 0

for i in 2:nx_fd-1
    κ_plus  = 0.5f0 * (κ_true(x_fd[i]) + κ_true(x_fd[i+1]))
    κ_minus = 0.5f0 * (κ_true(x_fd[i]) + κ_true(x_fd[i-1]))
    A[i, i-1] = -κ_minus / dx_fd^2
    A[i, i]   = (κ_plus + κ_minus) / dx_fd^2
    A[i, i+1] = -κ_plus / dx_fd^2
    rhs[i] = f_source(x_fd[i])
end

u_fd = A \ rhs

# Sample noisy observations at sparse locations
n_obs = 20
obs_idx = sort(rand(rng, 2:nx_fd-1, n_obs))
x_obs = x_fd[obs_idx]
u_obs = u_fd[obs_idx] .+ 0.005f0 .* randn(rng, Float32, n_obs)
```

```
# Plot the true solution and observations
fig = Figure(size = (700, 250))
ax1 = Axis(fig[1, 1], xlabel = "x", ylabel = "u(x)", title = "Solution field")
lines!(ax1, x_fd, u_fd, color = :black, linewidth = 2, label = "True u(x)")
scatter!(ax1, x_obs, u_obs, color = :red, markersize = 8, label = "Observations")
axislegend(ax1, position = :rt)

ax2 = Axis(fig[1, 2], xlabel = "x", ylabel = "κ(x)", title = "Diffusion coefficient")
lines!(ax2, x_fd, κ_true.(x_fd), color = :black, linewidth = 2, label = "True κ(x)")
axislegend(ax2, position = :rb)
fig
```

```
#-----κ representation------
n_features = 32
B_matrix = randn(rng, Float32, n_features, 1) .* 4.0f0  # frequency scale

function fourier_encode(x, B)
    projected = B * reshape(x, 1, :)  # (n_features, n_points)
    return vcat(sin.(2π .* projected), cos.(2π .* projected))  # (2*n_features, n_points)
end

#-----κ-network------
```

```
κ_model = Chain(
    Dense(2 * n_features => 64, tanh),
    Dense(64 => 64, tanh),
    Dense(64 => 1)
)

#-----u-network------
u_model = Chain(
    Dense(2 * n_features => 64, tanh),
    Dense(64 => 64, tanh),
    Dense(64 => 1)
)

ps_κ, st_κ = Lux.setup(rng, κ_model)
ps_u, st_u = Lux.setup(rng, u_model)
```

```
#-----combined loss------
function inversion_loss(ps_κ, ps_u, st_κ, st_u)
    λᵣ = 1.0f0
    λᵦ = 10.0f0

    # Evaluate κ and u at collocation points
    n_colloc = 100
    x_c = Float32.(range(0.01, 0.99, length = n_colloc))
    x_c_enc = fourier_encode(x_c, B_matrix)

    # Predict κ at collocation points (enforce positivity with softplus)
    κ_pred_raw, _ = κ_model(x_c_enc, ps_κ, st_κ)
    κ_pred = Lux.NNlib.softplus.(κ_pred_raw)  # ensure κ > 0

    # Predict u at collocation points
    u_pred_c, _ = u_model(x_c_enc, ps_u, st_u)

    # PDE residual via AD: -d/dx[κ du/dx] - f = 0
    # We compute du/dx and d(κ du/dx)/dx using finite differences on the
    # collocation grid (dense enough for accuracy), keeping AD for the networks
    δx = x_c[2] - x_c[1]
    u_vec = vec(u_pred_c)
    κ_vec = vec(κ_pred)

    # du/dx at half-points
    du_dx = diff(u_vec) ./ δx  # (n_colloc - 1,)
    # κ at half-points
    κ_half = 0.5f0 .* (κ_vec[1:end-1] .+ κ_vec[2:end])
    # flux: κ * du/dx
    flux = κ_half .* du_dx
```

```
    # d(flux)/dx at interior points
    d_flux_dx = diff(flux) ./ δx  # (n_colloc - 2,)

    f_vals = f_source.(x_c[2:end-1])
    pde_residual = -d_flux_dx .- f_vals
    loss_pde = mean(pde_residual .^ 2)

    # Data misfit: compare u predictions at observation locations
    x_obs_enc = fourier_encode(x_obs, B_matrix)
    u_pred_obs, _ = u_model(x_obs_enc, ps_u, st_u)
    loss_data = mean((vec(u_pred_obs) .- u_obs) .^ 2)

    # Boundary conditions: u(0) = 0, u(1) = 0
    x_bc = Float32[0.0, 1.0]
    x_bc_enc = fourier_encode(x_bc, B_matrix)
    u_bc, _ = u_model(x_bc_enc, ps_u, st_u)
    loss_bc = mean(u_bc .^ 2)

    total = loss_data + λᵣ * loss_pde + λᵦ * loss_bc
    return total
end
```

```
# Train both networks jointly
opt_κ = Adam(0.001f0)
opt_u = Adam(0.001f0)
opt_state_κ = Optimisers.setup(opt_κ, ps_κ)
opt_state_u = Optimisers.setup(opt_u, ps_u)

for epoch in 1:2000
    (loss_val), grads = Zygote.withgradient(ps_κ, ps_u) do pκ, pu
        inversion_loss(pκ, pu, st_κ, st_u)
    end

    opt_state_κ, ps_κ = Optimisers.update(opt_state_κ, ps_κ, grads[1])
    opt_state_u, ps_u = Optimisers.update(opt_state_u, ps_u, grads[2])

    if epoch == 1 || epoch % 400 == 0
        @printf "Epoch %4d  Loss = %.6f\n" epoch loss_val
    end
end
```

```
# Compare recovered κ to the true one
x_plot = Float32.(range(0, 1, length = 200))
x_plot_enc = fourier_encode(x_plot, B_matrix)
```

```
κ_recovered_raw, _ = κ_model(x_plot_enc, ps_κ, st_κ)
κ_recovered = vec(Lux.NNlib.softplus.(κ_recovered_raw))

u_recovered, _ = u_model(x_plot_enc, ps_u, st_u)
u_recovered = vec(u_recovered)

fig = Figure(size = (700, 300))
ax1 = Axis(fig[1, 1], xlabel = "x", ylabel = "κ(x)",
           title = "Recovered diffusion coefficient")
lines!(ax1, x_plot, κ_true.(x_plot), color = :black, linewidth = 2,
       label = "True κ(x)")
lines!(ax1, x_plot, κ_recovered, color = :steelblue, linewidth = 2,
       linestyle = :dash, label = "INR recovered")
axislegend(ax1, position = :rb)

ax2 = Axis(fig[1, 2], xlabel = "x", ylabel = "u(x)",
           title = "Recovered solution field")
lines!(ax2, x_fd, u_fd, color = :black, linewidth = 2, label = "True u(x)")
lines!(ax2, x_plot, u_recovered, color = :steelblue, linewidth = 2,
       linestyle = :dash, label = "INR recovered")
scatter!(ax2, x_obs, u_obs, color = :red, markersize = 6, label = "Observations")
axislegend(ax2, position = :rt)
fig
```

18.5 Why INRs work well for geophysical inversion

1. **Implicit regularization** — the network architecture itself acts as a regularizer. Smooth activation functions (like `tanh`) and limited network width naturally produce smooth model outputs without requiring explicit penalty terms.
2. **Spectral bias as a feature** — for geophysical inverse problems (which are ill-posed), the MLP's tendency to learn low frequencies first is actually beneficial: it recovers the large-scale structure first, then gradually adds detail, mimicking multi-scale inversion strategies.
3. **Continuous representation** — unlike grid-based inversion, the recovered model is defined everywhere in the domain. This makes it easy to evaluate at arbitrary resolution, compute derivatives, or interface with mesh-free PDE solvers.
4. **Fourier features for control** — by tuning the frequency scale of the Fourier feature encoding, you directly control the maximum spatial frequency the network can represent, providing explicit control over the model's resolution.

18.6 Connection to traditional inversion

The INR approach is closely related to classical regularized inversion:

	Classical inversion	INR-based inversion
Unknown	Model vector $\mathbf{m} \in \mathbb{R}^M$	Network weights θ
Regularization	Explicit: $\|\nabla m\|^2$, TV, etc.	Implicit: architecture + spectral bias
Resolution	Fixed by mesh	Continuous, adaptive
Gradient	Adjoint-state method	Automatic differentiation
Constraints	Hard to impose smoothly	Natural via activation functions

18.7 Geoscience applications

INR-based inversion is gaining traction across geoscience:

- **Gravity and magnetic inversion** — INRs parameterize 3D density or susceptibility models and train with the forward gravity/magnetic kernel, replacing voxel-based parameterizations with continuous neural fields.
- **Seismic full-waveform inversion** — the velocity model is represented as an INR, and the wave equation residual provides the physics constraint (Rasht-Behesht et al., 2022).
- **Electrical resistivity tomography** — the conductivity distribution is learned as an INR trained with the governing Poisson equation and electrode measurements.
- **Geothermal reservoir characterization** — physics-informed INRs have been applied to recover permeability fields from temperature and pressure observations (Sun et al., 2023).

Looking ahead

This chapter combined a single PDE with a single set of observations. In practice, geophysical inversions involve multiple data types (e.g., gravity + seismic) and complex 3D geometries. The INR framework extends naturally to these settings by adding more physics terms to the loss and using higher-dimensional coordinate inputs.

19 DeepONet

Key references

- **DeepONet** — the foundational neural operator based on the universal approximation theorem for operators, using a branch-trunk architecture (Lu, Jin, et al., 2021).
- **Universal approximation of operators** — the mathematical theorem guaranteeing that a two-sub-network architecture can approximate any continuous nonlinear operator (T. Chen & Chen, 1995).
- **DeepXDE** — a library for PINNs and DeepONets that formalized many practical training strategies (Lu, Meng, et al., 2021).
- **Neural operator survey** — a comprehensive mathematical treatment of neural operators and their approximation properties (Kovachki et al., 2023).
- **Neural operators for science** — review of neural operators accelerating scientific simulations (Azizzadenesheli et al., 2024).

In the PINN chapter we trained a network to solve **one instance** of a PDE. Change the boundary conditions, initial conditions, or source term, and you must retrain from scratch. **DeepONet** (Deep Operator Network) solves a fundamentally more ambitious problem: it learns the **solution operator** — the mapping from input functions to output functions — so that after training, it can instantly predict solutions for *any* new input without retraining.

19.1 The operator learning problem

In many PDE problems, the solution u depends on an input function a (a forcing term, initial condition, boundary condition, or material property):

$$\mathcal{N}[u; a] = 0 \quad \Longrightarrow \quad u = \mathcal{G}(a)$$

where $a \in \mathcal{A}$ is an input function, $u \in \mathcal{U}$ is the corresponding output function, and $\mathcal{G} : \mathcal{A} \to \mathcal{U}$ is the **solution operator**. Writing $\mathcal{G}(a)(y)$ means: first apply the operator to the whole input function a, then evaluate the resulting output function at the query location y. A traditional solver computes $\mathcal{G}(a)$ one a at a time. DeepONet learns $\mathcal{G}_\theta \approx \mathcal{G}$ from data — pairs $\{(a_i, u_i)\}_{i=1}^N$ — so that inference on new inputs is a single forward pass.

19.2 The universal approximation theorem for operators

T. Chen & Chen (1995) proved that a network with two sub-networks can approximate any continuous nonlinear operator to arbitrary accuracy. This theorem provides the mathematical foundation for DeepONet.

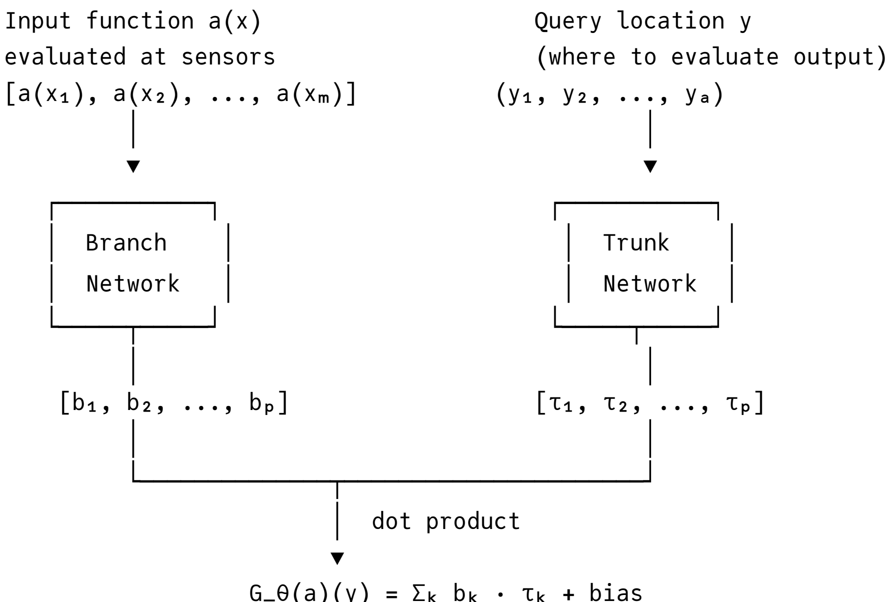
Theorem (informal): For any continuous operator $\mathcal{G} : \mathcal{A} \to \mathcal{U}$ and any $\epsilon > 0$, there exist neural networks (branch and trunk) such that:

$$\left|\mathcal{G}(a)(y) - \mathcal{G}_\theta(a)(y)\right| < \epsilon$$

for all input functions $a \in \mathcal{A}$ and all query locations y.

19.3 Architecture: branch and trunk

DeepONet decomposes the operator approximation into two learnable components:



$$\mathcal{G}_\theta(a)(y) = \sum_{k=1}^{p} b_k(a) \cdot \tau_k(y) + b_0$$

- **Branch network** — takes the input function a sampled at m fixed sensor locations $\{x_1, \dots, x_m\}$ and outputs p coefficients $\{b_1, \dots, b_p\}$. Think of these as the "expansion coefficients" of the output in a learned basis.
- **Trunk network** — takes the query location y (where we want to evaluate the output) and produces p basis functions $\{\tau_1, \dots, \tau_p\}$.

The final output is the inner product of the branch and trunk outputs. This is elegant: the branch encodes *what* the input function looks like, while the trunk learns *where* to produce the output. In other words, the branch depends on the input function a, while the trunk depends on the query location y.

19.4 Variants

- **Unstacked DeepONet** — single branch, single trunk (described above). The most common variant.
- **Stacked DeepONet** — multiple independent branch-trunk pairs summed together.
- **POD-DeepONet** — the trunk basis functions are replaced by Proper Orthogonal Decomposition (POD) modes computed from the training data, making the trunk data-driven rather than learned.

19.5 Code example: learning the antiderivative operator

We train a DeepONet to learn the antiderivative operator:

$$\mathscr{G}(a)(y) = \int_0^y a(s)\,ds$$

Given a function $a(x)$, the operator returns its antiderivative evaluated at any query point y.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)
```

```
# Data generation: random functions and their antiderivatives
n_sensors = 50        # fixed sensor locations for branch input
n_query = 50          # query points per sample
n_train = 2000
n_test = 200

x_sensors = Float32.(range(0, 1, length = n_sensors))
y_query = Float32.(range(0, 1, length = n_query))
dx_sensor = x_sensors[2] - x_sensors[1]

function random_function(rng)
    # Random sum of sinusoids
    n_modes = rand(rng, 2:5)
    coeffs = randn(rng, Float32, n_modes)
    freqs = Float32.(rand(rng, 1:6, n_modes))
    phases = 2π .* rand(rng, Float32, n_modes)
    function f(x)
        val = 0.0f0
        for j in 1:n_modes
```

```
            val += coeffs[j] * sin(2π * freqs[j] * x + phases[j])
        end
        return val
    end
    return f
end

function generate_data(rng, n_samples)
    A = zeros(Float32, n_sensors, n_samples)        # branch inputs
    Y = zeros(Float32, 1, n_query * n_samples)       # query locations
    G = zeros(Float32, 1, n_query * n_samples)       # true outputs

    for i in 1:n_samples
        f = random_function(rng)
        # Evaluate at sensors
        a_vals = f.(x_sensors)
        A[:, i] = a_vals

        # Compute antiderivative at query points via trapezoidal rule
        for (j, yj) in enumerate(y_query)
            # Integrate from 0 to yj
            n_int = max(2, round(Int, yj / dx_sensor) + 1)
            x_int = Float32.(range(0, yj, length = n_int))
            f_int = f.(x_int)
            integral = sum(0.5f0 * (f_int[1:end-1] .+ f_int[2:end]) .* diff(x_int))

            idx = (i - 1) * n_query + j
            Y[1, idx] = yj
            G[1, idx] = integral
        end
    end
    return A, Y, G
end

A_train, Y_train, G_train = generate_data(rng, n_train)
A_test, Y_test, G_test = generate_data(Xoshiro(123), n_test)
```

```
# Build DeepONet: Branch + Trunk
p = 64   # output dimension of both branch and trunk

branch_net = Chain(
    Dense(n_sensors => 128, relu),
    Dense(128 => 128, relu),
    Dense(128 => p)
)
```

```
trunk_net = Chain(
    Dense(1 => 128, relu),
    Dense(128 => 128, relu),
    Dense(128 => p)
)

ps_branch, st_branch = Lux.setup(rng, branch_net)
ps_trunk, st_trunk = Lux.setup(rng, trunk_net)
bias = zeros(Float32, 1)

ps_all = (branch = ps_branch, trunk = ps_trunk, bias = bias)
st_all = (branch = st_branch, trunk = st_trunk)
```

```
# Forward pass: DeepONet output
function deeponet_forward(A, Y, ps, st, n_samples, n_q)
    # Branch: process all input functions
    b_out, st_b = branch_net(A, ps.branch, st.branch)  # (p, n_samples)

    # Trunk: process all query locations
    t_out, st_t = trunk_net(Y, ps.trunk, st.trunk)      # (p, n_samples * n_q)

    # For each query point, dot-product the branch output of its sample
    # with the trunk output
    # Reshape branch output to match queries
    b_expanded = repeat(b_out, 1, n_q)  # (p, n_samples * n_q) — each sample's
    # coefficients repeated n_q times
    # This is a simplification: we need to properly tile
    b_tiled = similar(t_out)
    for i in 1:n_samples
        idx_range = (i-1)*n_q+1 : i*n_q
        b_tiled[:, idx_range] .= b_out[:, i]
    end

    # Dot product + bias
    output = sum(b_tiled .* t_out, dims = 1) .+ ps.bias  # (1, n_samples * n_q)

    st_new = (branch = st_b, trunk = st_t)
    return output, st_new
end

# Loss function
function deeponet_loss(ps, st)
    pred, st_new = deeponet_forward(A_train, Y_train, ps, st, n_train, n_query)
    loss = mean((pred .- G_train) .^ 2)
    return loss, st_new
end
```

```
# Training loop
opt = Adam(0.001f0)
opt_state = Optimisers.setup(opt, ps_all)

for epoch in 1:1000
    (loss_val, st_new), grads = Zygote.withgradient(ps_all) do ps
        deeponet_loss(ps, st_all)
    end
    opt_state, ps_all = Optimisers.update(opt_state, ps_all, grads[1])
    st_all = st_new

    if epoch == 1 || epoch % 200 == 0
        # Test error
        pred_test, _ = deeponet_forward(A_test, Y_test, ps_all, st_all, n_test, n_query)
        test_mse = mean((pred_test .- G_test) .^ 2)
        @printf "Epoch %4d  Train MSE = %.6f  Test MSE = %.6f\n" epoch loss_val test_mse
    end
end
```

```
# Visualize predictions on test samples
fig = Figure(size = (700, 500))
n_show = 4

for i in 1:n_show
    f_test = A_test[:, i]
    idx_range = (i-1)*n_query+1 : i*n_query

    true_vals = vec(G_test[:, idx_range])
    pred_vals_all, _ = deeponet_forward(
        reshape(f_test, :, 1),
        reshape(Y_test[:, idx_range], 1, :),
        ps_all, st_all, 1, n_query
    )
    pred_vals = vec(pred_vals_all)

    row = (i - 1) ÷ 2 + 1
    col = (i - 1) % 2 + 1
    ax = Axis(fig[row, col], xlabel = "y", ylabel = "∫₀ʸ a(s) ds",
              title = "Test sample $i")
    lines!(ax, y_query, true_vals, color = :black, linewidth = 2, label = "Exact")
    lines!(ax, y_query, pred_vals, color = :steelblue, linewidth = 2,
           linestyle = :dash, label = "DeepONet")
    if i == 1
        axislegend(ax, position = :lt, labelsize = 10)
    end
end
```

```
Label(fig[0, :], "DeepONet: learning the antiderivative operator", fontsize = 14)
fig
```

19.6 Key properties of DeepONet

19.6.1 Strengths

- **Mathematical foundation** — grounded in the universal approximation theorem for operators, providing theoretical guarantees on expressiveness.
- **Flexible geometry** — unlike FNO (which requires regular grids for the FFT), DeepONet works with arbitrary input sensor locations and query points. This is crucial for geoscience data, which is rarely on regular grids.
- **Modular design** — the branch and trunk are independent networks that can be customized separately. For example, the branch could be a CNN if the input is an image, or an RNN if the input is a time series.
- **Point-wise evaluation** — the trunk network produces output at individual query points, making it efficient for problems where you only need the solution at specific locations.

19.6.2 Limitations

- **Fixed sensor locations** — the branch network requires the input function to be evaluated at the *same* fixed sensor locations for all samples. This can be restrictive if data comes from different measurement configurations.
- **Data hungry** — DeepONet typically requires more training data than FNO for the same accuracy on problems with regular grids, because it doesn't exploit spatial structure the way spectral methods do.
- **Scaling** — for high-dimensional output functions, the number of trunk basis functions p may need to be large, increasing the computational cost.

19.7 DeepONet vs FNO

Feature	DeepONet	FNO
Architecture	Branch + Trunk	Spectral convolution layers
Grid requirements	Arbitrary (irregular OK)	Regular grid (for FFT)
Theoretical basis	Universal approximation theorem	Kernel integral operators
Resolution invariance	Through trunk network	Through Fourier representation
Best for	Irregular data, point queries	Regular-grid data, global patterns

Feature	DeepONet	FNO
Data efficiency	Moderate	Higher (exploits spatial structure)

19.8 Geoscience applications

DeepONet is particularly well-suited for geoscience problems because field data is rarely on regular grids:

- **Subsurface flow surrogates** — DeepONet can learn the mapping from permeability fields to pressure/saturation solutions, enabling real-time uncertainty quantification for reservoir management.
- **Seismic wave modeling** — given a velocity model as input, DeepONet predicts the wavefield at arbitrary receiver locations, replacing expensive wave-equation solves during inversion.
- **Well log prediction** — the branch network ingests spatially irregular borehole measurements while the trunk evaluates predictions at arbitrary subsurface locations.
- **Climate downscaling** — DeepONet maps coarse-resolution climate model output to fine-resolution fields, handling the irregular observation network naturally.

> Next: Physics-Informed DeepONet
>
> A powerful extension of DeepONet incorporates PDE constraints into the training loss, dramatically reducing the amount of training data needed. We explore this in the next chapter.

20 Physics-Informed DeepONet

Key references

- **Physics-Informed DeepONets** — the foundational paper combining DeepONet with PDE residual constraints, enabling operator learning with little or no labeled data (S. Wang, Wang, et al., 2021).
- **DeepONet** — the underlying architecture based on the universal approximation theorem for operators (Lu, Jin, et al., 2021).
- **PI-DeepONet for fracture mechanics** — application of physics-informed operator learning to complex multi-physics problems (Goswami et al., 2023).
- **Reliable extrapolation** — improving neural operator generalization through physics constraints and sparse observations (M. Wang et al., 2023).
- **Physics-informed ML review** — comprehensive treatment of PINNs, neural operators, and hybrid approaches (Karniadakis et al., 2021).

The DeepONet chapter showed how to learn a solution operator from input–output function pairs. But generating those pairs requires running a traditional PDE solver many times — which can be expensive. **Physics-Informed DeepONet** (PI-DeepONet) (S. Wang, Wang, et al., 2021) eliminates or reduces this requirement by embedding the PDE directly into the DeepONet training loss, just as PINNs embed the PDE into a single-instance solver.

20.1 The key idea

Recall the DeepONet prediction:

$$\mathcal{G}_\theta(a)(y) = \sum_{k=1}^{p} b_k(a) \cdot \tau_k(y) + b_0$$

In a standard DeepONet, we train by minimizing the data loss:

$$\mathcal{L}_{\text{data}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{Q} \left| \mathcal{G}_\theta(a_i)(y_j) - u_i(y_j) \right|^2$$

where the u_i are computed by a PDE solver. In PI-DeepONet, we replace (or supplement) this with a **PDE residual loss**: since the DeepONet output is differentiable with respect to the query location y, we can compute $\frac{\partial \mathcal{G}_\theta}{\partial y}$, $\frac{\partial^2 \mathcal{G}_\theta}{\partial y^2}$, etc. via automatic differentiation and penalize violations of the PDE.

20.2 PI-DeepONet loss function

$$\mathcal{L}(\theta) = \lambda_r \mathcal{L}_r(\theta) + \lambda_{\text{ic}} \mathcal{L}_{\text{ic}}(\theta)$$

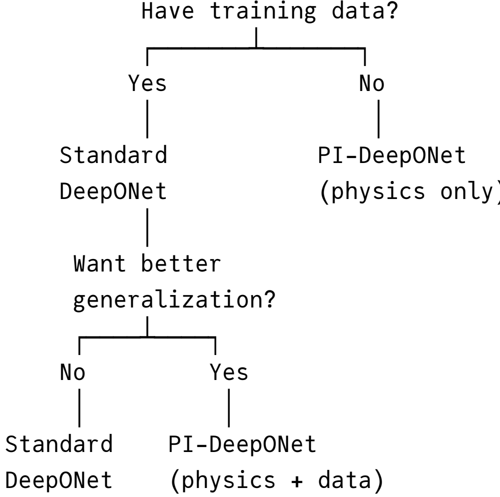
with

$$\mathcal{L}_r(\theta) = \frac{1}{N \cdot N_c} \sum_{i=1}^{N} \sum_{j=1}^{N_c} \left| \mathcal{N}[\mathcal{G}_\theta(a_i)](y_j^c) \right|^2$$

$$\mathcal{L}_{\text{ic}}(\theta) = \frac{1}{N \cdot N_{\text{ic}}} \sum_{i=1}^{N} \sum_{k=1}^{N_{\text{ic}}} \left| \mathcal{G}_\theta(a_i)(y_k^{\text{ic}}) - g_i(y_k^{\text{ic}}) \right|^2$$

The crucial difference from a PINN: the loss is evaluated over **many different input functions** a_i simultaneously, so the network learns an operator, not just one solution. And the crucial difference from a standard DeepONet: no PDE solver is needed to generate training data — the physics is enforced directly.

20.3 When to use PI-DeepONet



PI-DeepONet is most valuable when:

1. **No training data** — you know the PDE but can't afford to run a solver thousands of times.
2. **Limited data** — you have a few solver runs and want to supplement with physics.
3. **Extrapolation** — physics constraints improve generalization beyond the training distribution.

20.4 Code example: 1D diffusion–reaction operator

We train a PI-DeepONet to learn the solution operator for the 1D diffusion–reaction equation:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} - k\,u + a(x), \quad x \in [0,1], \quad t \in [0,0.5]$$

with $u(x,0) = 0$, $u(0,t) = u(1,t) = 0$, and varying source term $a(x)$.

The operator maps $a(x) \mapsto u(x,t)$ at a chosen final time T. We train the PI-DeepONet **without any solver data** — purely from the PDE.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie

rng = Xoshiro(42)

#-----physical parameters------
D = 0.01f0   # diffusion coefficient
k = 1.0f0    # reaction rate
T_final = 0.5f0
```

```
# Sensor locations for the branch input (fixed)
n_sensors = 30
x_sensors = Float32.(range(0, 1, length = n_sensors))

# Generate random source functions a(x)
function random_source(rng)
    n_modes = rand(rng, 2:4)
    coeffs = 0.5f0 .* randn(rng, Float32, n_modes)
    freqs = Float32.(rand(rng, 1:4, n_modes))
    phases = 2π .* rand(rng, Float32, n_modes)
    function a(x)
        val = 0.0f0
        for j in 1:n_modes
            val += coeffs[j] * sin(2π * freqs[j] * x + phases[j])
        end
        return val
    end
    return a
end

# Sample source functions and evaluate at sensors
n_funcs = 200   # number of different source functions per batch
function sample_batch(rng, n)
    A = zeros(Float32, n_sensors, n)
    funcs = []
    for i in 1:n
```

```
        f = random_source(rng)
        A[:, i] = f.(x_sensors)
        push!(funcs, f)
    end
    return A, funcs
end
```

```
# Build the PI-DeepONet
p = 64   # basis dimension

branch_net = Chain(
    Dense(n_sensors => 128, tanh),
    Dense(128 => 128, tanh),
    Dense(128 => p)
)

# Trunk input: (x, t) → p basis functions
trunk_net = Chain(
    Dense(2 => 128, tanh),
    Dense(128 => 128, tanh),
    Dense(128 => p)
)

ps_branch, st_branch = Lux.setup(rng, branch_net)
ps_trunk, st_trunk = Lux.setup(rng, trunk_net)
bias_init = zeros(Float32, 1)

ps = (branch = ps_branch, trunk = ps_trunk, bias = bias_init)
st = (branch = st_branch, trunk = st_trunk)
```

```
# DeepONet forward for a single (a, y) pair — needed for AD through y
function deeponet_single(a_sensors, y_xt, ps, st)
    # a_sensors: (n_sensors,) — branch input
    # y_xt: (2,) — trunk input [x, t]
    b_out, _ = branch_net(reshape(a_sensors, :, 1), ps.branch, st.branch)  # (p, 1)
    t_out, _ = trunk_net(reshape(y_xt, :, 1), ps.trunk, st.trunk)          # (p, 1)
    output = sum(b_out .* t_out) + ps.bias[1]
    return output
end
```

```
# PI-DeepONet loss: PDE residual + IC + BC
function pi_deeponet_loss(ps, st)
    A_batch, funcs = sample_batch(rng, n_funcs)
```

```
n_colloc = 30   # collocation points per function
loss_pde = 0.0f0
loss_ic = 0.0f0
loss_bc = 0.0f0
n_pde_total = 0
n_ic_total = 0
n_bc_total = 0

for i in 1:n_funcs
    a_i = A_batch[:, i]
    source_fn = funcs[i]

    # Collocation points in interior
    x_c = rand(rng, Float32, n_colloc) .* 0.98f0 .+ 0.01f0
    t_c = rand(rng, Float32, n_colloc) .* T_final .* 0.98f0 .+ 0.01f0 * T_final

    for j in 1:n_colloc
        xt = Float32[x_c[j], t_c[j]]

        # u prediction
        u_val = deeponet_single(a_i, xt, ps, st)

        # du/dt and d²u/dx² via AD
        grad_xt = Zygote.gradient(y -> deeponet_single(a_i, y, ps, st), xt)[1]
        du_dx = grad_xt[1]
        du_dt = grad_xt[2]

        # d²u/dx² — gradient of du/dx w.r.t. x
        d2u_dx2 = Zygote.gradient(
            y -> Zygote.gradient(z -> deeponet_single(a_i, z, ps, st), y)[1][1],
            xt
        )[1][1]

        # PDE: du/dt - D * d²u/dx² + k*u - a(x) = 0
        residual = du_dt - D * d2u_dx2 + k * u_val - source_fn(x_c[j])
        loss_pde += residual^2
        n_pde_total += 1
    end

    # Initial condition: u(x, 0) = 0
    n_ic_pts = 10
    for j in 1:n_ic_pts
        x_ic = rand(rng, Float32)
        xt_ic = Float32[x_ic, 0.0f0]
        u_ic = deeponet_single(a_i, xt_ic, ps, st)
        loss_ic += u_ic^2
        n_ic_total += 1
```

```
        end

        # Boundary conditions: u(0,t) = 0, u(1,t) = 0
        n_bc_pts = 5
        for j in 1:n_bc_pts
            t_bc = rand(rng, Float32) * T_final
            u_bc0 = deeponet_single(a_i, Float32[0.0f0, t_bc], ps, st)
            u_bc1 = deeponet_single(a_i, Float32[1.0f0, t_bc], ps, st)
            loss_bc += u_bc0^2 + u_bc1^2
            n_bc_total += 2
        end
    end

    total = loss_pde / n_pde_total +
            10.0f0 * loss_ic / n_ic_total +
            10.0f0 * loss_bc / n_bc_total
    return total, st
end
```

```
# Training loop (this is slow due to per-point AD — reduce iterations for demo)
opt = Adam(0.001f0)
opt_state = Optimisers.setup(opt, ps)

for epoch in 1:200
    (loss_val, st_new), grads = Zygote.withgradient(ps) do p
        pi_deeponet_loss(p, st)
    end
    opt_state, ps = Optimisers.update(opt_state, ps, grads[1])
    st = st_new

    if epoch == 1 || epoch % 50 == 0
        @printf "Epoch %3d  Loss = %.6f\n" epoch loss_val
    end
end
```

```
# Evaluate on new source functions (not seen during training)
fig = Figure(size = (700, 500))
n_show = 4
nx_eval = 50

for i in 1:n_show
    rng_test = Xoshiro(2000 + i)
    f_test = random_source(rng_test)
    a_test = f_test.(x_sensors)
```

```
    x_eval = Float32.(range(0, 1, length = nx_eval))
    u_pred_T = zeros(Float32, nx_eval)

    for j in 1:nx_eval
        xt = Float32[x_eval[j], T_final]
        u_pred_T[j] = deeponet_single(a_test, xt, ps, st)
    end

    row = (i - 1) ÷ 2 + 1
    col = (i - 1) % 2 + 1
    ax = Axis(fig[row, col], xlabel = "x",
              title = "Source $i — u(x, T=$T_final)")
    lines!(ax, x_sensors, a_test .* 0.1f0, color = (:gray, 0.4),
           label = "a(x) × 0.1")
    lines!(ax, x_eval, u_pred_T, color = :steelblue, linewidth = 2,
           label = "PI-DeepONet u(x,T)")
    if i == 1
        axislegend(ax, position = :rt, labelsize = 10)
    end
end

Label(fig[0, :],
      "PI-DeepONet: diffusion-reaction — trained without solver data",
      fontsize = 14)
fig
```

20.5 How PI-DeepONet differs from PINN + DeepONet

	PINN	DeepONet	PI-DeepONet
Learns	One PDE solution	Solution operator	Solution operator
Training data	None (physics only)	Input–output pairs from solver	None or few (physics + optional data)
Generalizes to new inputs?	No (retrain)	Yes	Yes
PDE knowledge used?	Yes (residual loss)	No	Yes (residual loss)
Solver needed?	No	Yes	No (or optional)

20.6 The training trade-off

PI-DeepONet is more expensive per training step than standard DeepONet because it requires computing PDE derivatives via AD for each collocation point. However, it saves the *offline cost* of generating training data with a PDE solver. The break-even depends on:

- **Solver cost** — if the PDE is cheap to solve (e.g., 1D Poisson), generate data and use standard DeepONet. If expensive (e.g., 3D wave equation), PI-DeepONet saves significant compute.
- **Number of input functions** — PI-DeepONet scales well because it can sample random input functions during training without ever solving the PDE.

20.7 Geoscience applications

PI-DeepONet is especially valuable in geoscience where:

- **Forward solvers are expensive** — seismic wave propagation in 3D, coupled multiphysics simulations (thermo-hydro-mechanical), and full Navier-Stokes for mantle convection all have high per-solve costs. PI-DeepONet avoids running these solvers thousands of times.
- **Physics is well known** — geophysical PDEs (wave equation, diffusion equation, Darcy flow) are well established, making physics-informed training reliable.
- **Generalization is needed** — during uncertainty quantification or Bayesian inversion, the operator must be evaluated for thousands of different parameter configurations. A PI-DeepONet trained once can provide all of these evaluations instantly.

Specific applications include:

- **Seismic waveform modeling** — learn the mapping from velocity models to seismograms, with the wave equation enforced during training (Rasht-Behesht et al., 2022).
- **Groundwater flow** — learn the mapping from hydraulic conductivity fields to pressure heads, constrained by Darcy's law and the groundwater flow equation (Q. He et al., 2020).
- **Fracture mechanics** — PI-DeepONet has been applied to predict stress intensity factors and crack propagation under varying loading conditions (Goswami et al., 2023).

> **i** Practical tip
>
> When training PI-DeepONet, start with strong weights on the initial/boundary condition losses ($\lambda_{ic} = \lambda_{bc} = 10$–$100$) and a moderate weight on the PDE residual ($\lambda_r = 1$). The network must first learn to satisfy the boundary conditions before the PDE residual becomes meaningful.

21 Fourier Neural Operator

Key references

- **Fourier Neural Operator (FNO)** — learning solution operators in Fourier space, enabling resolution-invariant surrogate models for PDEs (Li et al., 2021).
- **Neural operator survey** — a comprehensive mathematical treatment of neural operators and their approximation properties (Kovachki et al., 2023).
- **FourCastNet** — global weather prediction using adaptive Fourier neural operators (Pathak et al., 2022).
- **U-FNO** — enhanced FNO with U-Net-style skip connections for multiphase flow in subsurface reservoirs (Wen et al., 2022).
- **FNO on general geometries** — extending FNO to handle irregular domains via learned deformations (Li et al., 2023).
- **Neural operators for science** — review of neural operators accelerating scientific simulations (Azizzadenesheli et al., 2024).

The DeepONet learned operators using a branch-trunk decomposition. The **Fourier Neural Operator** (FNO) (Li et al., 2021) takes a completely different approach: it performs learning directly in **Fourier space**, where global spatial patterns are captured efficiently and the learned operator is inherently resolution-invariant.

FNO has become arguably the most successful neural operator architecture, powering applications from weather forecasting to fluid dynamics to seismic inversion.

21.1 From convolution to spectral convolution

A standard convolution applies a local kernel:

$$(K * v)(x) = \int K(x - y)\, v(y)\, dy$$

This has a **local receptive field** — each output point depends only on nearby inputs. For PDEs, where information propagates globally (e.g., waves, diffusion), stacking many convolutional layers is needed to capture long-range interactions.

By the **convolution theorem**, convolution in physical space is multiplication in Fourier space:

$$\mathcal{F}[K * v] = \mathcal{F}[K] \cdot \mathcal{F}[v]$$

The FNO leverages this: instead of learning a spatial kernel, it learns a **spectral filter** — a weight tensor that multiplies the Fourier coefficients directly. This gives each layer a **global receptive field** while keeping the computation efficient through the FFT.

21.2 FNO architecture

An FNO consists of three stages:

21.2.1 1. Lifting layer

A point-wise linear layer that maps the input from its native dimension to a higher-dimensional representation:

$$v^{(0)}(x) = Pa(x) + \mathbf{q}$$

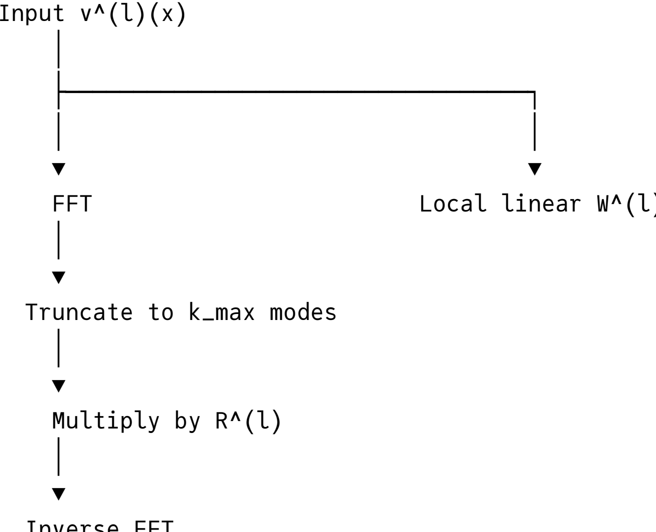
where $P \in \mathbb{R}^{d_v \times d_a}$ and $a(x)$ is the input function at location x.

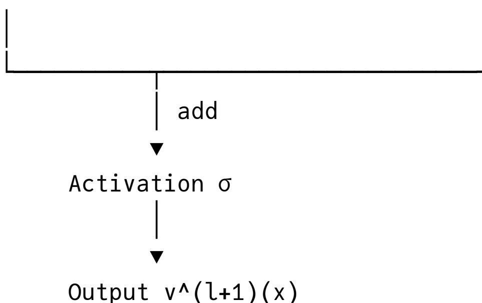
21.2.2 2. Fourier layers (repeated L times)

Each Fourier layer applies:

$$v^{(l+1)}(x) = \sigma\Big(\underbrace{W^{(l)}\, v^{(l)}(x)}_{\text{local linear}} + \underbrace{\mathcal{F}^{-1}\big[R^{(l)} \cdot \mathcal{F}[v^{(l)}]\big](x)}_{\text{spectral convolution}}\Big)$$

The spectral convolution keeps only the lowest $k_{\max}$ Fourier modes and sets the rest to zero. The learned weight tensor $R^{(l)} \in \mathbb{C}^{d_v \times d_v \times k_{\max}}$ acts as a frequency-dependent linear transformation. Here the superscript (l) plays the same role as elsewhere in the book: it indexes layer depth.





21.2.3 3. Projection layer

A point-wise linear layer that maps back to the output dimension:

$$u(x) = Qv^{(L)}(x) + r$$

21.2.4 Why this works

- **Global receptive field** — each Fourier layer sees the entire spatial domain, not just a local neighborhood.
- **Resolution invariance** — the spectral representation is independent of the grid resolution. A model trained on a 64×64 grid can be evaluated on a 256×256 grid by zero-padding the Fourier modes.
- **Efficiency** — the FFT costs $O(n \log n)$, making the spectral convolution as fast as spatial convolution for typical grid sizes.
- **Mode truncation as regularization** — keeping only $k_{\max}$ modes implicitly smooths the learned operator, preventing overfitting to high-frequency noise.

21.3 Code example: FNO for the 1D advection equation

We train a simple FNO to learn the solution operator for the 1D advection equation:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \quad u(x, 0) = u_0(x)$$

The exact solution is a rightward shift: $u(x, t) = u_0(x - ct)$. The operator maps $u_0 \mapsto u(\cdot, T)$.

```
using Lux, Random, Optimisers, Zygote, Statistics, Printf, CairoMakie, FFTW

rng = Xoshiro(42)
```

```
# Generate training data: advection with periodic BCs
nx = 64
c = 1.0f0
T_final = 0.5f0
```

```
dx = 1.0f0 / nx
x_grid = Float32.(range(0, 1 - dx, length = nx))

function generate_pair(rng)
    # Random initial condition: sum of sinusoids
    n_modes = rand(rng, 2:5)
    u0 = zeros(Float32, nx)
    for _ in 1:n_modes
        k = rand(rng, 1:6)
        a = 0.5f0 * randn(rng, Float32)
        ϕ = 2π * rand(rng, Float32)
        u0 .+= a .* sin.(2π .* k .* x_grid .+ ϕ)
    end
    # Exact solution at t = T: shift by c*T (periodic)
    shift = round(Int, c * T_final / dx)
    uT = circshift(u0, -shift)
    return u0, uT
end

n_train = 500
n_test = 100
U0_train = zeros(Float32, nx, n_train)
UT_train = zeros(Float32, nx, n_train)

for i in 1:n_train
    u0, uT = generate_pair(rng)
    U0_train[:, i] = u0
    UT_train[:, i] = uT
end

U0_test = zeros(Float32, nx, n_test)
UT_test = zeros(Float32, nx, n_test)
for i in 1:n_test
    u0, uT = generate_pair(Xoshiro(9000 + i))
    U0_test[:, i] = u0
    UT_test[:, i] = uT
end
```

```
# Spectral convolution layer (custom Lux layer)
struct SpectralConv <: Lux.AbstractLuxLayer
    in_channels::Int
    out_channels::Int
    modes::Int
end

function Lux.initialparameters(rng::AbstractRNG, l::SpectralConv)
```

```
    scale = 1.0f0 / (l.in_channels * l.out_channels)
    R_real = scale .* randn(rng, Float32, l.out_channels, l.in_channels, l.modes)
    R_imag = scale .* randn(rng, Float32, l.out_channels, l.in_channels, l.modes)
    return (R_real = R_real, R_imag = R_imag)
end

Lux.initialstates(::AbstractRNG, ::SpectralConv) = NamedTuple()

function (l::SpectralConv)(x, ps, st)
    # x: (nx, channels, batch)
    nx_in = size(x, 1)
    batch = size(x, 3)

    # FFT along spatial dimension
    x_ft = rfft(x, 1)  # (nx÷2+1, channels, batch)

    # Multiply by learned weights for first `modes` frequencies
    R = complex.(ps.R_real, ps.R_imag)  # (out_ch, in_ch, modes)

    out_ft = zeros(ComplexF32, size(x_ft, 1), l.out_channels, batch)
    for b in 1:batch
        for k in 1:l.modes
            for o in 1:l.out_channels
                for ic in 1:l.in_channels
                    out_ft[k, o, b] += R[o, ic, k] * x_ft[k, ic, b]
                end
            end
        end
    end

    # Inverse FFT
    out = irfft(out_ft, nx_in, 1)
    return out, st
end
```

```
# Build FNO: lift → spectral layers → project
modes = 12
width = 16

lift = Dense(1 => width)

spectral1 = SpectralConv(width, width, modes)
skip1 = Dense(width => width)

spectral2 = SpectralConv(width, width, modes)
skip2 = Dense(width => width)
```

```
spectral3 = SpectralConv(width, width, modes)
skip3 = Dense(width => width)

project = Dense(width => 1)

ps_lift, st_lift = Lux.setup(rng, lift)
ps_s1, st_s1 = Lux.setup(rng, spectral1)
ps_sk1, st_sk1 = Lux.setup(rng, skip1)
ps_s2, st_s2 = Lux.setup(rng, spectral2)
ps_sk2, st_sk2 = Lux.setup(rng, skip2)
ps_s3, st_s3 = Lux.setup(rng, spectral3)
ps_sk3, st_sk3 = Lux.setup(rng, skip3)
ps_proj, st_proj = Lux.setup(rng, project)

ps_all = (lift = ps_lift,
          s1 = ps_s1, sk1 = ps_sk1,
          s2 = ps_s2, sk2 = ps_sk2,
          s3 = ps_s3, sk3 = ps_sk3,
          proj = ps_proj)
st_all = (lift = st_lift,
          s1 = st_s1, sk1 = st_sk1,
          s2 = st_s2, sk2 = st_sk2,
          s3 = st_s3, sk3 = st_sk3,
          proj = st_proj)

function fno_forward(x, ps, st)
    # x: (nx, 1, batch)
    nx_in, _, batch = size(x)

    # Lift to higher dimension
    x_flat = reshape(x, 1, nx_in * batch)
    v, st_l = lift(x_flat, ps.lift, st.lift)
    v = reshape(v, nx_in, width, batch)

    # Fourier layer 1
    v_spec, st_s1 = spectral1(v, ps.s1, st.s1)
    v_skip = reshape(v, width, nx_in * batch)
    v_local, st_sk1 = skip1(v_skip, ps.sk1, st.sk1)
    v_local = reshape(v_local, nx_in, width, batch)
    v = relu.(v_spec .+ v_local)

    # Fourier layer 2
    v_spec2, st_s2 = spectral2(v, ps.s2, st.s2)
    v_skip2 = reshape(v, width, nx_in * batch)
    v_local2, st_sk2 = skip2(v_skip2, ps.sk2, st.sk2)
    v_local2 = reshape(v_local2, nx_in, width, batch)
```

```
    v = relu.(v_spec2 .+ v_local2)

    # Fourier layer 3
    v_spec3, st_s3 = spectral3(v, ps.s3, st.s3)
    v_skip3 = reshape(v, width, nx_in * batch)
    v_local3, st_sk3 = skip3(v_skip3, ps.sk3, st.sk3)
    v_local3 = reshape(v_local3, nx_in, width, batch)
    v = relu.(v_spec3 .+ v_local3)

    # Project back
    v_flat = reshape(v, width, nx_in * batch)
    out, st_p = project(v_flat, ps.proj, st.proj)
    out = reshape(out, nx_in, 1, batch)

    st_new = (lift = st_l,
              s1 = st_s1, sk1 = st_sk1,
              s2 = st_s2, sk2 = st_sk2,
              s3 = st_s3, sk3 = st_sk3,
              proj = st_p)
    return out, st_new
end
```

```
# Training loop
opt = Adam(0.001f0)
opt_state = Optimisers.setup(opt, ps_all)

X_train = reshape(U0_train, nx, 1, n_train)
Y_train = reshape(UT_train, nx, 1, n_train)
X_test = reshape(U0_test, nx, 1, n_test)
Y_test = reshape(UT_test, nx, 1, n_test)

for epoch in 1:500
    (loss, st_new), grads = Zygote.withgradient(ps_all) do ps
        pred, st_ = fno_forward(X_train, ps, st_all)
        l = mean((pred .- Y_train) .^ 2)
        (l, st_)
    end
    opt_state, ps_all = Optimisers.update(opt_state, ps_all, grads[1])
    st_all = st_new

    if epoch == 1 || epoch % 100 == 0
        pred_test, _ = fno_forward(X_test, ps_all, st_all)
        test_mse = mean((pred_test .- Y_test) .^ 2)
        @printf "Epoch %3d  Train MSE = %.6f  Test MSE = %.6f\n" epoch loss test_mse
    end
end
```

```
# Test on new initial conditions
n_show = 4
fig = Figure(size = (700, 500))

for i in 1:n_show
    u0_test, uT_test = generate_pair(Xoshiro(5000 + i))
    x_in = reshape(u0_test, nx, 1, 1)
    pred, _ = fno_forward(x_in, ps_all, st_all)

    row = (i - 1) ÷ 2 + 1
    col = (i - 1) % 2 + 1
    ax = Axis(fig[row, col], title = "Test $i",
              xlabel = "x", ylabel = "u")
    lines!(ax, x_grid, u0_test, color = (:gray, 0.5), label = "u₀")
    lines!(ax, x_grid, uT_test, color = :black, linewidth = 2, label = "Exact u(T)")
    lines!(ax, x_grid, vec(pred), color = :steelblue, linewidth = 2,
           linestyle = :dash, label = "FNO prediction")
    if i == 1
        axislegend(ax, position = :rt, labelsize = 10)
    end
end

Label(fig[0, :], "FNO: advection equation — generalizing to new initial conditions",
      fontsize = 14)
fig
```

21.4 Resolution invariance

One of FNO's most remarkable properties is **resolution invariance** (also called zero-shot super-resolution). Because the learned operator acts on Fourier modes rather than grid points, a model trained at one resolution can be evaluated at a different resolution:

1. **Train** on a 64-point grid.
2. **Evaluate** on a 256-point grid by zero-padding the spectral weights to the new resolution.

This works because the Fourier coefficients have the same physical meaning regardless of discretization — they represent the same spatial frequencies. In practice, FNO maintains good accuracy when evaluating at 2–4× the training resolution, with degradation beyond that.

21.5 FNO variants

The original FNO has inspired many extensions:

Variant	Key idea	Use case
FNO-2D/3D	Multi-dimensional spectral convolution	2D/3D PDEs
U-FNO (Wen et al., 2022)	U-Net-style skip connections between Fourier layers	Multiphase subsurface flow
Geo-FNO (Li et al., 2023)	Learned input deformation for irregular domains	Geophysics, complex geometries
AFNO (Adaptive FNO) (Pathak et al., 2022)	Token mixing via adaptive Fourier layers	Global weather forecasting
Factorized FNO	Factorize multi-dimensional spectral weights	Reduced memory for 3D problems

21.6 Comparison with other neural operators

Feature	FNO	DeepONet	PI-DeepONet
Core mechanism	Spectral convolution	Branch-trunk decomposition	Branch-trunk + PDE loss
Grid requirement	Regular grid	Arbitrary	Arbitrary
Resolution invariance	Native (via Fourier)	Via trunk re-evaluation	Via trunk re-evaluation
Training data	Input-output pairs	Input-output pairs	Physics (+ optional data)
Global receptive field	Per layer	Per forward pass	Per forward pass
Best suited for	Regular-grid PDEs	Irregular sensors	Limited/no solver data

21.7 Geoscience applications

FNO is transforming geoscience workflows where repeated forward modeling is the computational bottleneck:

- **Weather forecasting** — FourCastNet (Pathak et al., 2022) used adaptive Fourier neural operators for global weather prediction, producing 10-day forecasts in seconds. Pangu-Weather (Bi et al., 2023) achieved competitive medium-range forecasting with 3D neural architectures.
- **Seismic inversion surrogates** — FNO-based surrogates replace expensive wave-equation solves during seismic full-waveform inversion, achieving orders-of-magnitude speedup (Yin et al., 2023).
- **Subsurface multiphase flow** — U-FNO (Wen et al., 2022) was applied to CO_2 storage simulations, learning the mapping from injection scenarios to pressure and saturation fields for real-time reservoir management.
- **Seismic wave simulation** — FNO learns the mapping from velocity models to wavefields, enabling rapid scenario testing for seismic hazard assessment (C. Song & Alkhalifah, 2023).
- **General geometries** — Geo-FNO (Li et al., 2023) handles the irregular domains common in Earth science (topography, coastlines, geological boundaries) through learned coordinate deformations.

When to choose FNO

FNO excels when your data lives on **regular grids** and you need **fast, resolution-invariant** operator evaluation. If your data is on irregular sensors or you lack training data, consider DeepONet or PI-DeepONet instead.

Part IV

Applications

22 SciML Applications in Geoscience

This chapter surveys the rapidly growing literature on **scientific machine learning** (SciML) applied to geoscience problems. Unlike the previous chapters — which introduced methods with worked code examples — here we focus on *what has been done* and *where the field is heading*, organized by geoscience sub-discipline.

The methods covered in this survey correspond directly to the tools introduced earlier in this book: PINNs, physics-based ML inversion, DeepONet, PI-DeepONet, and FNO.

22.1 Seismology and seismic imaging

Seismology is one of the most active areas for SciML adoption, driven by the computational cost of wave-equation solves and the abundance of observational data.

22.1.1 PINNs for wave propagation

- Waheed et al. (2021) developed **PINNeik**, a PINN that solves the eikonal equation for seismic travel-time computation. The mesh-free formulation handles complex velocity models naturally and avoids the grid artifacts of fast-marching methods.
- Smith et al. (2021) trained **EikoNet**, a deep neural network constrained by the eikonal equation, to predict seismic travel times in 3D heterogeneous velocity models. Once trained, it provides travel times at arbitrary source–receiver pairs without re-solving.
- C. Song et al. (2021) extended PINNs to solve the **frequency-domain acoustic wave equation** for anisotropic (VTI) media, demonstrating that PINNs can handle the additional complexity of directional velocity dependence.
- Rasht-Behesht et al. (2022) applied PINNs to **full waveform modeling and inversion**, showing that PDE-constrained neural networks can simultaneously recover velocity models and wavefields from sparse seismic observations.
- C. Song & Alkhalifah (2023) used Fourier-feature-enhanced PINNs to simulate **multifrequency seismic wavefields**, overcoming the spectral bias that limits standard PINNs for high-frequency wave solutions.

22.1.2 Neural operators for seismic modeling

- Yin et al. (2023) developed **learned surrogates** based on neural operators for multiphysics-based seismic inverse problems, replacing expensive wave-equation solves with fast neural operator evaluations during gradient-based inversion.
- FNO-based surrogates have been trained to map velocity models to seismograms, enabling **rapid scenario testing** for seismic hazard assessment where thousands of forward simulations are needed.

22.1.3 Seismic inversion

- Yang & Ma (2019) applied deep learning to **full-waveform inversion (FWI)**, using neural networks to directly map seismic data to velocity models while embedding physical constraints to improve the inversion quality.
- Implicit neural representations (INRs) have been used to parameterize velocity models during FWI, providing **implicit regularization** through the network architecture and eliminating the need for hand-tuned regularization parameters.

22.2 Hydrogeology and subsurface flow

Groundwater modeling and subsurface flow simulation are natural targets for SciML because the governing equations (Darcy's law, Richards' equation) are well established but computationally expensive for heterogeneous media.

22.2.1 PINNs for groundwater modeling

- Q. He et al. (2020) applied PINNs to **multiphysics data assimilation** in subsurface transport, demonstrating that physics-informed training can recover contaminant concentration fields from sparse monitoring well data while honoring the advection-diffusion equation.
- Y. Zhu et al. (2019) developed **physics-constrained deep learning** for high-dimensional surrogate modeling and uncertainty quantification in subsurface transport, showing that PINNs can work without labeled data when the governing PDE is known.

22.2.2 Neural operator surrogates for reservoir simulation

- Wen et al. (2022) introduced **U-FNO**, an enhanced Fourier neural operator with U-Net-style skip connections, applied to **multiphase flow in CO_2 storage** simulations. The surrogate learned the mapping from injection scenarios to pressure and saturation fields, enabling real-time reservoir management decisions.
- Neural operators have been trained as surrogates for **history matching** in reservoir engineering, where the operator maps permeability fields to production data, enabling Bayesian inversion with thousands of forward evaluations that would be infeasible with traditional simulators.

- M. Wang et al. (2023) demonstrated **reliable extrapolation** of neural operators informed by physics for CO_2 sequestration modeling, showing that physics constraints improve generalization beyond the training distribution.

22.3 Weather and climate science

Global weather prediction has emerged as a flagship application for neural operators, with several models now rivaling or exceeding traditional numerical weather prediction (NWP) systems.

22.3.1 Neural operator weather models

- Pathak et al. (2022) developed **FourCastNet**, using adaptive Fourier neural operators (AFNO) for global weather prediction. The model produces 10-day forecasts in **seconds** rather than the hours required by traditional NWP systems, at comparable accuracy for many variables.
- Bi et al. (2023) introduced **Pangu-Weather**, a 3D neural architecture that achieved competitive medium-range weather forecasting. The model was trained on 39 years of ERA5 reanalysis data and demonstrated skillful forecasts up to 7 days.
- Lam et al. (2023) developed **GraphCast**, a graph-neural-network-based weather model that outperformed ECMWF's HRES operational system on most metrics, marking a milestone in data-driven weather prediction.

22.3.2 Physics-informed climate modeling

- Kashinath et al. (2021) presented case studies of **physics-informed machine learning for weather and climate modeling**, demonstrating how embedding physical constraints (conservation laws, symmetries) improves forecast accuracy and ensures physical consistency of predictions.
- Physics-informed approaches have been applied to **climate downscaling** — using neural operators to map coarse-resolution climate model output to fine-resolution fields while respecting physical constraints like energy conservation.

22.4 Geothermal energy

Geothermal reservoir characterization and management require solving coupled thermo-hydro-mechanical (THM) equations that are computationally demanding.

- Sun et al. (2023) applied PINNs to **geothermal reservoir simulation**, using physics-informed training to model subsurface temperature and pressure distributions from sparse borehole measurements. The PINN naturally combines the heat equation with observational data, producing physically consistent predictions between measurement points.

- Neural operator surrogates have been explored for **geothermal uncertainty quantification**, where the mapping from subsurface property fields to temperature/pressure responses must be evaluated thousands of times during Bayesian inversion.

22.5 Solid Earth geophysics

22.5.1 Gravity and magnetic inversion

- INR-based parameterizations have been applied to **3D gravity inversion**, where the density distribution is represented as a neural field and trained by minimizing the misfit between predicted and observed gravity anomalies. The network architecture provides implicit regularization, eliminating the need for Tikhonov-style penalty terms.
- Physics-informed approaches to potential field inversion exploit the **Laplace/Poisson equation** as an additional constraint, improving the depth resolution of recovered density models.

22.5.2 Electromagnetic methods

- PINNs have been applied to **magnetotelluric (MT) forward modeling**, solving Maxwell's equations in the frequency domain for layered and 2D conductivity structures.
- Neural operator surrogates for **electromagnetic induction** problems enable rapid evaluation of forward responses during probabilistic inversion of controlled-source electromagnetic (CSEM) data.

22.6 Geodynamics and mantle convection

- PINNs have been applied to solve the **Stokes equations** for mantle convection, where the high computational cost of traditional finite element solvers limits the number of simulations that can be run for parameter studies.
- Neural operators have been explored as surrogates for **mantle convection simulations**, learning the mapping from rheological parameters and boundary conditions to flow fields and temperature distributions.

22.7 Earthquake science

- Mousavi et al. (2020) developed **Earthquake Transformer (EQTransformer)**, an attention-based deep learning model for simultaneous earthquake detection and phase picking. While not strictly a SciML method (it is data-driven), it demonstrates the power of neural architectures for seismological signal processing.
- PINNs have been applied to **earthquake source characterization**, where the wave equation is used as a physics constraint to invert for source parameters (location, mechanism, rupture history) from seismogram recordings.
- Moseley et al. (2020) used deep learning for **fast simulation of seismic waves** in complex media, training neural networks as fast surrogates for finite-difference wave propagation codes.

22.8 Cross-cutting themes

Several themes emerge across these applications:

22.8.1 1. The speed–accuracy trade-off

Neural surrogates (FNO, DeepONet) sacrifice some accuracy for dramatic speedups — typically 10^3–$10^6\times$ faster than traditional solvers. This trade-off is acceptable when the surrogate is used within an outer loop (inversion, UQ) where thousands of forward evaluations are needed.

22.8.2 2. Physics as regularization

In data-scarce geoscience settings, physics-informed losses (PINNs, PI-DeepONet) act as powerful regularizers, preventing the network from producing physically impossible predictions. This is especially valuable in subsurface characterization where direct measurements are sparse and expensive.

22.8.3 3. Mesh-free flexibility

PINNs and INRs naturally handle the irregular geometries common in geoscience — topography, coastlines, fault surfaces, borehole trajectories — without requiring mesh generation, which is often the most time-consuming part of traditional numerical modeling.

22.8.4 4. Differentiable pipelines

Because the entire SciML pipeline is differentiable (from model parameters through the PDE solve to predictions), gradient-based optimization for inversion and optimal experimental design becomes straightforward. This is a fundamental advantage over traditional solvers that require adjoint formulations for gradient computation.

22.8.5 5. Foundation models for Earth science

An emerging trend is the development of **foundation models** — large neural operators pre-trained on massive simulation or reanalysis datasets — that can be fine-tuned for specific tasks. Weather prediction (FourCastNet, Pangu-Weather, GraphCast) is leading this trend, with analogous efforts underway for ocean modeling, seismology, and climate science.

22.9 Choosing the right SciML approach

Problem type	Recommended approach	Key advantage
Solve one PDE instance	**PINN**	No training data, mesh-free
Recover model parameters from data	**Physics-ML inversion (INR)**	Implicit regularization
Repeated forward solves (regular grid)	**FNO**	Fast inference, resolution-invariant
Repeated forward solves (irregular data)	**DeepONet**	Flexible geometry
Operator learning without solver data	**PI-DeepONet**	Physics replaces training data
Real-time forecasting at scale	**FNO / AFNO**	Sub-second global predictions

A rapidly evolving field

The SciML landscape is changing fast. New architectures, training strategies, and geoscience applications appear regularly. The references in this chapter provide entry points into the literature as of 2025; readers are encouraged to track developments through venues such as *Journal of Computational Physics*, *Nature Machine Intelligence*, *Geophysical Journal International*, and the ICLR/NeurIPS workshops on AI for science.

References

Alkhalifah, T., Song, C., Waheed, U. bin, & Hao, Q. (2022). Machine learning for data-driven geophysics. *Surveys in Geophysics*, *43*, 191–218. https://doi.org/10.1007/s10712-021-09681-9

An, P., & Moon, W. M. (2005). Reservoir characterization using feedforward neural networks. *SEG Technical Program Expanded Abstracts*, 258–262. https://doi.org/10.1190/1.1822454

An, P., Moon, W. M., & Kalantzis, F. (2001). Reservoir characterization using seismic waveform and feedforword neural networks. *Geophysics*, *66*(5). https://doi.org/10.1190/1.1487090

Azizzadenesheli, K., Kovachki, N., Li, Z., Liu-Schiaffini, M., Kossaifi, J., & Anandkumar, A. (2024). Neural operators for accelerating scientific simulations and design. *Nature Reviews Physics*, *6*, 320–328. https://doi.org/10.1038/s42254-024-00712-5

Benaouda, D., Wadge, G., Whitmarsh, R. B., Rothwell, R. G., & MacLeod, C. (1999). Inferring the lithology of borehole rocks by applying neural network classifiers to downhole logs: An example from the ocean drilling program. *Geophysical Journal International*, *136*(2), 477–491. https://doi.org/10.1046/j.1365-246X.1999.00746.x

Bergen, K. J., Johnson, P. A., Hoop, M. V. de, & Beroza, G. C. (2019). Machine learning for data-driven discovery in solid earth geoscience. *Science*, *363*(6433). https://doi.org/10.1126/science.aau0323

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Bi, K., Xie, L., Zhang, H., Chen, X., Gu, X., & Tian, Q. (2023). Accurate medium-range global weather forecasting with 3D neural networks. *Nature*, *619*, 533–538. https://doi.org/10.1038/s41586-023-06185-3

Calderón-Macías, C., Sen, M. K., & Stoffa, P. L. (1998). Automatic NMO correction and velocity estimation by a feedforward neural network. *Geophysics*, *63*(5). https://doi.org/10.1190/1.1444465

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018). Neural ordinary differential equations. *Advances in Neural Information Processing Systems*, *31*.

Chen, T., & Chen, H. (1995). Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, *6*(4), 911–917. https://doi.org/10.1109/72.392253

Cho, K., Merriënboer, B. van, Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1724–1734. https://doi.org/10.3115/v1/D14-1179

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303–314. https://doi.org/10.1007/BF02551274

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171–4186.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. *Proceedings of the International Conference on Learning Representations (ICLR)*.

Dramsch, J. S. (2020). 70 years of machine learning in geoscience in review. *Advances in Geophysics*, *61*, 1–55. https://doi.org/10.1016/bs.agph.2020.08.002

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*(2), 179–211. https://doi.org/10.1207/s15516709cog1402_1

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 1263–1272.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets. *Advances in Neural Information Processing Systems*, *27*.

Goswami, S., Yin, M., Yu, Y., & Karniadakis, G. E. (2023). Physics-informed DeepONets for modeling fracture mechanics. *Computer Methods in Applied Mechanics and Engineering*, *406*, 115852. https://doi.org/10.1016/j.cma.2023.115852

Goutorbe, B., Lucazeau, F., & Bonneville, A. (2006). Using neural networks to predict thermal conductivity from geophysical well logs. *Geophysical Journal International*, *166*(1), 115–125. https://doi.org/10.1111/j.1365-246X.2006.02924.x

Grathwohl, W., Chen, R. T. Q., Bettencourt, J., Sutskever, I., & Duvenaud, D. (2019). FFJORD: Free-form continuous dynamics for scalable reversible generative models. *Proceedings of the International Conference on Learning Representations (ICLR)*.

Ham, Y.-G., Kim, J.-H., & Luo, J.-J. (2019). Deep learning for multi-year ENSO forecasts. *Nature*, *573*, 568–572. https://doi.org/10.1038/s41586-019-1559-7

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778. https://doi.org/10.1109/CVPR.2016.90

He, Q., Barajas-Solano, D., Tartakovsky, G., & Tartakovsky, A. M. (2020). Physics-informed neural networks for multiphysics data assimilation with application to subsurface transport. *Advances in Water Resources*, *141*, 103610. https://doi.org/10.1016/j.advwatres.2020.103610

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, *313*(5786), 504–507. https://doi.org/10.1126/science.1127647

Ho, J., Jain, A., & Abbeel, P. (2020). Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, *33*, 6840–6851.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

Huang, Z., Shimeld, J., Williamson, M., & Katsube, J. (1996). Permeability prediction with artificial neural network modeling in the venture gas field, offshore eastern canada. *Geophysics*, *61*(2), 422–436. https://doi.org/10.1190/1.1443970

Huang, Z., & Williamson, M. A. (1997). Determination of porosity and permeability in reservoir intervals by artificial neural network modelling, offshore eastern canada. *Petroleum Geoscience*, *3*(3), 245–254. https://doi.org/10.1144/petgeo.3.3.245

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 448–456.

Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S., & Yang, L. (2021). Physics-informed machine learning. *Nature Reviews Physics*, *3*, 422–440. https://doi.org/10.1038/s42254-021-00314-5

Kashinath, K., Mustafa, M., Albert, A., Wu, J., Jiang, C., Esmaeilzadeh, S., Azizzadenesheli, K., Wang, R., Chattopadhyay, A., Singh, A., et al. (2021). Physics-informed machine learning: Case studies for weather and climate modelling. *Philosophical Transactions of the Royal Society A*, *379*(2194), 20200093. https://doi.org/10.1098/rsta.2020.0093

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1412.6980

Kingma, D. P., & Welling, M. (2014). Auto-encoding variational bayes. *Proceedings of the 2nd International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1312.6114

Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. *Proceedings of the 5th International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1609.02907

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhatt, K., Stuart, A., & Anandkumar, A. (2023). Neural operator: Learning maps between function spaces with applications to PDEs. *Journal of Machine Learning Research*, *24*(89), 1–97.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, *25*.

Lagaris, I. E., Likas, A., & Fotiadis, D. I. (1998). Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, *9*(5), 987–1000. https://doi.org/10.1109/72.712178

Laloy, E., Hérault, R., Jacques, D., & Linde, N. (2018). Training-image based geostatistical inversion using a spatial generative adversarial neural network. *Water Resources Research*, *54*(1), 381–406. https://doi.org/10.1002/2017WR022148

Lam, R., Sanchez-Gonzalez, A., Willson, M., Wirnsberger, P., Fortunato, M., Alet, F., Ravuri, S., Ewalds, T., Eaton-Rosen, Z., Hu, W., et al. (2023). Learning skillful medium-range global weather forecasting. *Science*, *382*(6677), 1416–1421. https://doi.org/10.1126/science.adi2336

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*, 436–444. https://doi.org/10.1038/nature14539

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, *1*(4), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

Li, Z., Huang, D. Z., Liu, B., & Anandkumar, A. (2023). Fourier neural operator with learned deformations for PDEs on general geometries. *Journal of Machine Learning Research*, *24*(388), 1–26.

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhatt, K., Stuart, A., & Anandkumar, A. (2021). Fourier neural operator for parametric partial differential equations. *Proceedings of the International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2010.08895

Lipman, Y., Chen, R. T. Q., Ben-Hamu, H., Nickel, M., & Le, M. (2023). Flow matching for generative modeling. *Proceedings of the International Conference on Learning Representations (ICLR)*.

Lopez-Alvis, J., Hermans, T., Nguyen, F., Caterina, D., & Haugerud, A. J. (2019). Deep-learning-based inverse modeling approaches: A subsurface transport example. *Water Resources Research*, *55*(8), 6305–6327. https://doi.org/10.1029/2018WR024638

Lu, L., Jin, P., Pang, G., Zhang, Z., & Karniadakis, G. E. (2021). Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, *3*, 218–229. https://doi.org/10.1038/s42256-021-00302-5

Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2021). DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, *63*(1), 208–228. https://doi.org/10.1137/19M1274067

McClenny, L. D., & Braga-Neto, U. M. (2023). Self-adaptive physics-informed neural networks. *Journal of Computational Physics*, *474*, 111722. https://doi.org/10.1016/j.jcp.2022.111722

McCormack, M. D., Zaucha, D. E., & Dushek, D. W. (1993). First-break refraction event picking and seismic data trace editing using neural networks. *Geophysics*, *58*(1). https://doi.org/10.1190/1.1443352

Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., & Ng, R. (2022). NeRF: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, *65*(1), 99–106. https://doi.org/10.1145/3503250

Moseley, B., Markham, A., & Nissen-Meyer, T. (2020). Deep learning for fast simulation of seismic waves in complex media. *Solid Earth*, *11*, 1527–1549. https://doi.org/10.5194/se-11-1527-2020

Mosser, L., Dubrule, O., & Blunt, M. J. (2017). Reconstruction of three-dimensional porous media using generative adversarial neural networks. *Physical Review E*, *96*(4), 043309. https://doi.org/10.1103/PhysRevE.96.043309

Mosser, L., Dubrule, O., & Blunt, M. J. (2020). Stochastic seismic waveform inversion using generative adversarial networks as a geological prior. *Mathematical Geosciences*, *52*, 53–79. https://doi.org/10.1007/s11004-019-09832-6

Mousavi, S. M., Ellsworth, W. L., Zhu, W., Chuber, L. Y., & Beroza, G. C. (2020). Earthquake transformer – an attentive deep-learning model for simultaneous earthquake detection and phase picking. *Nature Communications*, *11*(3952). https://doi.org/10.1038/s41467-020-17591-w

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 807–814.

Pathak, J., Subramanian, S., Harrington, P., Raja, S., Chattopadhyay, A., Mardani, M., Kurth, T., Hall, D., Li, Z., Azizzadenesheli, K., Hassanzadeh, P., Kashinath, K., & Anandkumar, A. (2022). FourCastNet: A global data-driven high-resolution weather forecasting model using adaptive fourier neural operators. *arXiv Preprint arXiv:2202.11214*.

Perol, T., Gharbi, M., & Denolle, M. (2018). Convolutional neural network for earthquake detection and location. *Science Advances*, *4*(2), e1700578. https://doi.org/10.1126/sciadv.1700578

Rackauckas, C., Ma, Y., Martensen, J., Warner, C., Zubov, K., Supekar, R., Skinner, D., Ramadhan, A., & Edelman, A. (2020). Universal differential equations for scientific machine learning. *arXiv Preprint arXiv:2001.04385*.

Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. *Proceedings of the International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1511.06434

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

Rasht-Behesht, M., Huber, C., Shukla, K., & Karniadakis, G. E. (2022). Physics-informed neural networks (PINNs) for wave propagation and full waveform inversions. *Journal of Geophysical Research: Solid Earth*, *127*(5), e2021JB023120. https://doi.org/10.1029/2021JB023120

Reichstein, M., Camps-Valls, G., Stevens, B., Jung, M., Denzler, J., Carvalhais, N., & Prabhat. (2019). Deep learning and process understanding for data-driven earth system science. *Nature*, *566*, 195–204. https://doi.org/10.1038/s41586-019-0912-1

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 234–241. https://doi.org/10.1007/978-3-319-24574-4_28

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408. https://doi.org/10.1037/h0042519

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*, 533–536. https://doi.org/10.1038/323533a0

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, *20*(1), 61–80. https://doi.org/10.1109/TNN.2008.2005605

Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W., & Woo, W. (2015). Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *Advances in Neural Information Processing Systems*, *28*.

Sitzmann, V., Martel, J. N. P., Bergman, A. W., Lindell, D. B., & Wetzstein, G. (2020). Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, *33*, 7462–7473.

Smith, J. D., Azizzadenesheli, K., & Ross, Z. E. (2021). EikoNet: Solving the eikonal equation with deep

neural networks. *IEEE Transactions on Geoscience and Remote Sensing*, *59*(12), 10685–10696. https://doi.org/10.1109/TGRS.2020.3039165

Sohl-Dickstein, J., Weiss, E. A., Maheswaranathan, N., & Ganguli, S. (2015). Deep unsupervised learning using nonequilibrium thermodynamics. *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2256–2265.

Song, C., & Alkhalifah, T. (2023). Simulating seismic multifrequency wavefields with the fourier feature physics-informed neural network. *Geophysical Journal International*, *232*(3), 1503–1514. https://doi.org/10.1093/gji/ggac399

Song, C., Alkhalifah, T., & Waheed, U. bin. (2021). Solving the frequency-domain acoustic VTI wave equation using physics-informed neural networks. *Geophysical Journal International*, *225*(2), 846–859. https://doi.org/10.1093/gji/ggab010

Song, Y., Sohl-Dickstein, J., Kingma, D. P., Kumar, A., Ermon, S., & Poole, B. (2021). Score-based generative modeling through stochastic differential equations. *Proceedings of the International Conference on Learning Representations (ICLR)*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(1), 1929–1958.

Stanev, E. V., Schulz-Stellenfleth, J., & Grayek, S. (2021). Hydrological coherence of remotely sensed water level data products with graph neural networks. *Journal of Hydrology*, *603*, 126923. https://doi.org/10.1016/j.jhydrol.2021.126923

Sun, Q., Burghardt, J., & Bhatt, D. (2023). Physics-informed neural networks for geothermal reservoir simulation. *Geothermics*, *109*, 102644. https://doi.org/10.1016/j.geothermics.2023.102644

Tancik, M., Srinivasan, P. P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., Ramamoorthi, R., Barron, J. T., & Ng, R. (2020). Fourier features let networks learn high frequency functions in low dimensional domains. *Advances in Neural Information Processing Systems*, *33*, 7537–7547.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*.

Waheed, U. bin, Haghighat, E., Alkhalifah, T., Song, C., & Hao, Q. (2021). PINNeik: Eikonal solution using physics-informed neural networks. *Computers & Geosciences*, *155*, 104833. https://doi.org/10.1016/j.cageo.2021.104833

Wang, M., Wang, S., & Perdikaris, P. (2023). Reliable extrapolation of deep neural operators informed by physics or sparse observations. *Computer Methods in Applied Mechanics and Engineering*, *412*, 116064. https://doi.org/10.1016/j.cma.2023.116064

Wang, S., Teng, Y., & Perdikaris, P. (2021). Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, *43*(5), A3055–A3081. https://doi.org/10.1137/20M1318043

Wang, S., Wang, H., & Perdikaris, P. (2021). Learning the solution operator of parametric partial differential equations with physics-informed DeepONets. *Science Advances*, *7*(40), eabi8605. https://doi.org/10.1126/sciadv.abi8605

Wen, G., Li, Z., Azizzadenesheli, K., Anandkumar, A., & Benson, S. M. (2022). U-FNO – an enhanced fourier neural operator-based deep-learning model for multiphase flow. *Advances in Water Resources*, *163*, 104180. https://doi.org/10.1016/j.advwatres.2022.104180

Wu, X., Liang, L., Shi, Y., & Fomel, S. (2019). FaultSeg3D: Using synthetic data sets to train an end-to-end convolutional neural network for 3D seismic fault segmentation. *Geophysics*, *84*(3), IM35–IM45. https://doi.org/10.1190/geo2018-0646.1

Yang, F., & Ma, J. (2019). Deep-learning inversion: A next-generation seismic velocity model building method. *Geophysics*, *84*(4), R583–R599. https://doi.org/10.1190/geo2018-0249.1

Yin, Z., Orozco, R., Louboutin, M., & Herrmann, F. J. (2023). Solving multiphysics-based inverse problems with learned surrogates and constraints. *Advanced Modeling and Simulation in Engineering Sciences*, *10*,

14. https://doi.org/10.1186/s40323-023-00252-0

Yuan, S., Liu, J., Wang, S., Wang, T., & Shi, P. (2020). Seismic waveform classification and first-break picking using convolution neural networks. *IEEE Geoscience and Remote Sensing Letters*, *17*(8), 1408–1412. https://doi.org/10.1109/LGRS.2019.2948601

Zhu, W., & Beroza, G. C. (2019). PhaseNet: A deep-neural-network-based seismic arrival-time picking method. *Geophysical Journal International*, *216*(1), 261–273. https://doi.org/10.1093/gji/ggy423

Zhu, Y., Zabaras, N., Koutsourelakis, P.-S., & Perdikaris, P. (2019). Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. *Journal of Computational Physics*, *394*, 56–81. https://doi.org/10.1016/j.jcp.2019.05.024