Week 4: Neural Network Basics

Structure, Activations, and How Networks Learn from Data

Where Week 4 Fits

- Weeks 1–3 focused on dynamical models, mechanisms, and parameter variation.
- Week 4 introduces the trainable nonlinear map that later appears as f_θ.
- We use a simple regression task ($y = \sin x$) so students can focus on the learning idea itself.

Physics model $\rightarrow$ data-driven model: the building block we need is a flexible trainable function.

Learning Objectives

By the end of this week, students should be able to:

- explain why a linear model can be too limited
- describe the **structure** of a one-hidden-layer MLP
- recognize common **activation functions** (sigmoid, tanh, ReLU)
- read the **forward-pass** equation and **MSE loss**
- follow the **chain rule** through a backprop step
- understand **gradient descent** and the role of the learning rate
- connect this example to later Neural ODE work

Learning Path — Eight Acts

1. **Motivation**: a line cannot fit $\sin x$
2. **Structure**: from one neuron to a full MLP
3. **Activations**: what brings the nonlinearity
4. **Hidden features**: how a layer builds a curve
5. **Training**: loss, backprop, gradient descent, learning rate
6. **Capacity**: architecture vs. over-/under-fitting
7. **Beyond the MLP**: CNN, RNN, Transformer, GNN, ...
8. **Bridges**: Neural ODE and GFM modeling

Concrete example first, then unpack the black box.

Why This Week Matters

Later we will learn a derivative map such as

$$\dot{x} = f_\theta(x) \qquad \text{or} \qquad \dot{x} = f_\theta(x, u)$$

Week 4 answers a simpler question first:

What kind of model can learn a flexible input–output map from data?

The **tool** (neural network) is the same; the **target meaning** will change in Week 5+.

Clean Example 1: A Line vs. a Curved Target

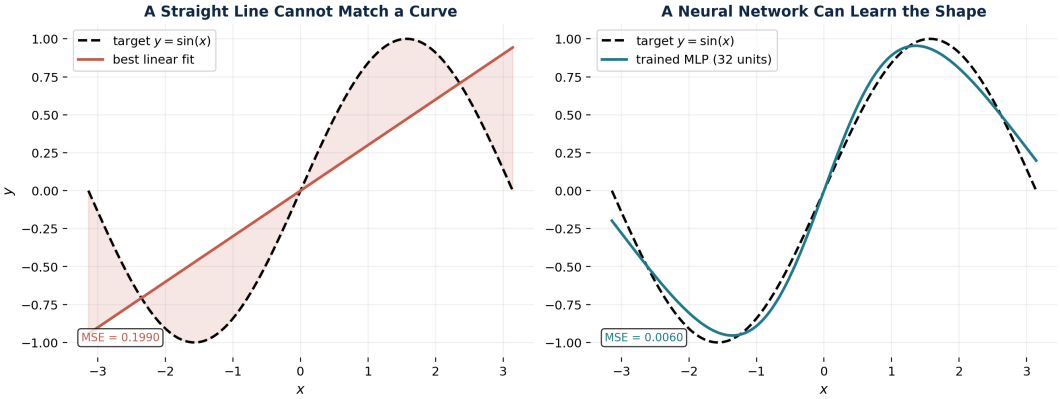
We use the simple target

$$y = \sin(x), \qquad x \in [-\pi, \pi]$$

Two questions to ask **before** looking at the plot:

- what happens if the model is forced to be a **straight line** $\hat{y} = ax + b$?
- what extra freedom does a **nonlinear** model provide?

Linear Model vs. Nonlinear Network



What Students Should Notice

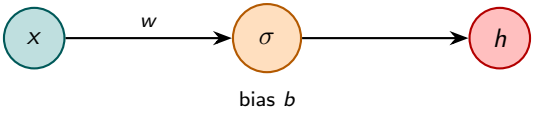
- the straight-line model captures only a rough trend — large error in the middle
- the nonlinear network can bend to match the curved structure
- this is the real **motivation** for using a neural network here

So: how is a "nonlinear network" actually built?

The Building Block: a Single Neuron

A single neuron is a small **affine** transformation followed by a **nonlinearity**:

$$h = \sigma(w\,x + b)$$



- w, b are **learnable parameters** adjusted during training.
- $\sigma(\cdot)$ is the **activation function** (we cover it next).
- Without σ, stacking neurons would still give a linear map.

Stacking Neurons: One-Hidden-Layer MLP

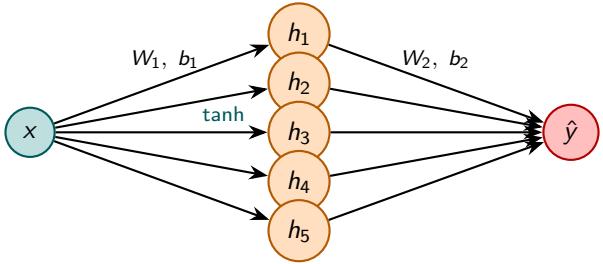
Put many neurons in a layer, then combine their outputs:

$$\hat{y} = W_2 \tanh(W_1 x + b_1) + b_2$$

- **Layer 1** (W_1, b_1): creates hidden features $h_j = \tanh(W_{1,j} x + b_{1,j})$.
- **Activation** (tanh): bends each feature into a nonlinear shape.
- **Layer 2** (W_2, b_2): linearly combines features into the prediction.

This is the smallest network that can represent curved functions.

Network Architecture: One Hidden Layer



(27 more)

Layer	Shape	Parameters
W_1, b_1	$1 \times 32,\ 1 \times 32$	64
W_2, b_2	$32 \times 1,\ 1 \times 1$	33
Total		**97**

Why We Need a Nonlinear Activation

Suppose we drop σ and stack two linear layers:

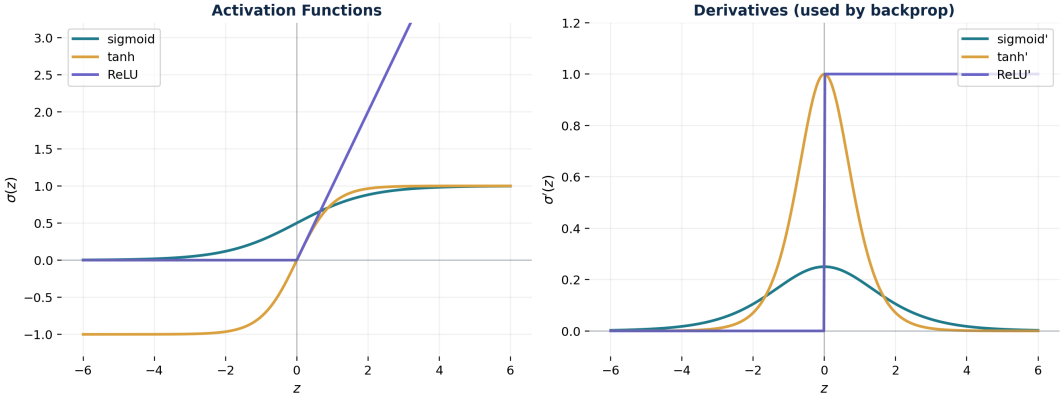
$$\hat{y} = W_2\left(W_1 x + b_1\right) + b_2 = \underbrace{\left(W_2 W_1\right)}_{W'} x + \underbrace{\left(W_2 b_1 + b_2\right)}_{b'}$$

No matter how many layers we stack, the result is still $\hat{y} = W'x + b'$ — a **single linear map**.

Key insight

Depth is useless without nonlinearity. The activation function $\sigma(\cdot)$ is what gives the network its **expressive power**.

Common Activation Functions



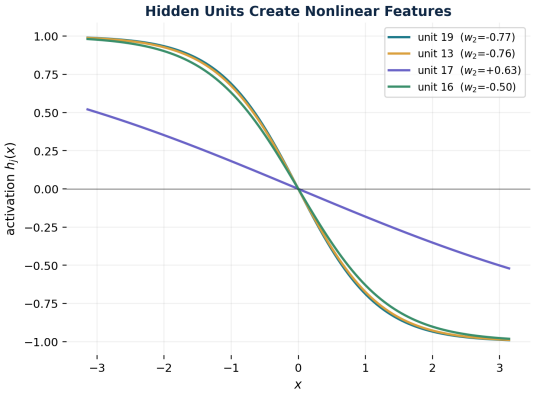
Left: the activations themselves. Right: their derivatives — these enter the backprop formulas.

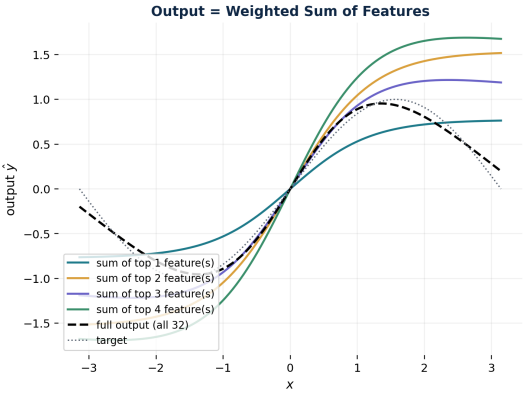
Sigmoid, Tanh, ReLU — Side by Side

	Sigmoid	**Tanh**	**ReLU**
Formula	$\dfrac{1}{1+e^{-z}}$	$\dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$\max(0, z)$
Range	$(0, 1)$	$(-1, 1)$	$[0, \infty)$
Derivative	$\sigma(z)\big(1-\sigma(z)\big)$	$1-\tanh^2(z)$	$\mathbf{1}[z > 0]$
Zero-centered?	No	Yes	No
Vanishing gradient?	Yes (large $\lvert z\rvert$)	Yes (large $\lvert z\rvert$)	No (for $z > 0$)
Typical use	output probabilities	classic hidden layer	default for deep nets

We use **tanh** for this week's sine-regression task because it is smooth, zero-centered, and well-behaved on small networks.

Hidden Features Build the Output





How To Read the Hidden Layer

- each hidden unit produces one intermediate nonlinear response
- the output layer adds **weighted versions** of those responses
- the final curve is built from a combination of simpler pieces

A network learns **what features to build** *and* **how to combine them** — *both at the same time.*

Loss: Measuring Prediction Error

For regression we use **mean squared error** (MSE):

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}_i - y_i\right)^2$$

- $\theta = \{W_1, b_1, W_2, b_2\}$ — all learnable parameters.
- L is a single number: smaller = better fit.
- Goal of training: **find θ that minimizes L.**

Training is an optimization problem in parameter space.

The Training Loop — Big Picture First

Four steps, repeated many times:

1. **Forward pass**: compute $\hat{y}$ from the current θ.
2. **Loss**: compute $L = \mathrm{MSE}(\hat{y}, y)$.
3. **Backward pass**: compute $\nabla_\theta L$.
4. **Update**: $\theta \leftarrow \theta - \eta \, \nabla_\theta L$.

Steps 1, 2, 4 are easy to write down. **Step 3 is the mystery**: how do we compute the gradient of the loss w.r.t. every parameter when there are thousands of them?

Answer: **backpropagation** — *the chain rule applied systematically.*

Backpropagation: The Chain Rule Applied

Forward pass (compute prediction and cache intermediates):

$$z_1 = W_1 x + b_1 \quad \rightarrow \quad h = \tanh(z_1) \quad \rightarrow \quad \hat{y} = W_2 h + b_2 \quad \rightarrow \quad L = \tfrac{1}{N}\|\hat{y} - y\|^2$$

Backward pass (propagate gradient from L back to parameters):

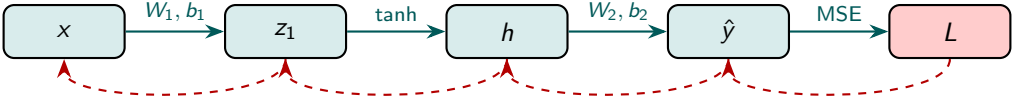
$$\frac{\partial L}{\partial \hat{y}} = \tfrac{2}{N}(\hat{y} - y) \;\Rightarrow\; \frac{\partial L}{\partial W_2} = h^\top \frac{\partial L}{\partial \hat{y}}, \quad \frac{\partial L}{\partial b_2} = \sum \tfrac{\partial L}{\partial \hat{y}}$$

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial \hat{y}} W_2^\top \;\Rightarrow\; \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial h} \odot \left(1 - \tanh^2 z_1\right)$$

$$\frac{\partial L}{\partial W_1} = x^\top \frac{\partial L}{\partial z_1}, \qquad \frac{\partial L}{\partial b_1} = \sum \tfrac{\partial L}{\partial z_1}$$

Each arrow is one application of the chain rule.

Forward and Backward Pass — Summary



- **Forward** (solid, teal): compute prediction and loss.
- **Backward** (dashed, red): propagate $\partial L/\partial(\cdot)$ from the loss back to every parameter.
- Cached intermediates (z_1, h) let us reuse work from the forward pass.

Gradient Descent: The Update Rule

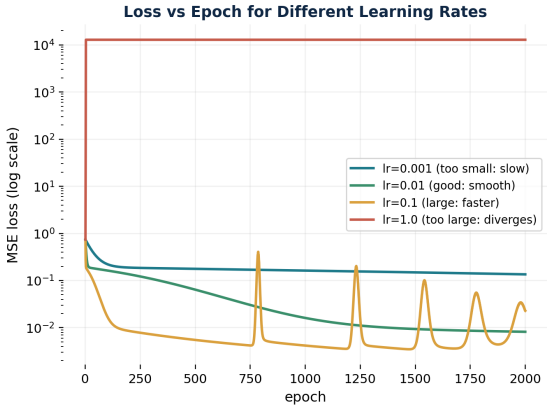
Once backprop gives us $\nabla_\theta L$, apply it to every parameter:

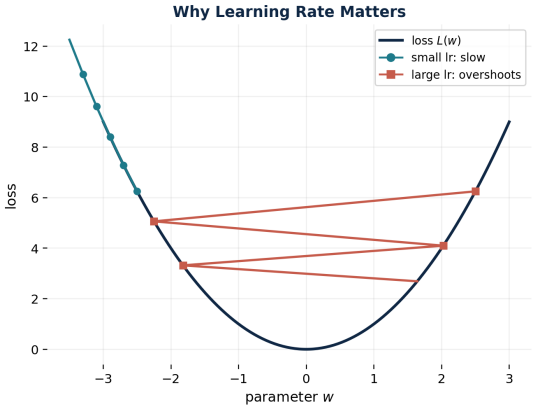
$$W_1 \leftarrow W_1 - \eta \frac{\partial L}{\partial W_1}, \qquad b_1 \leftarrow b_1 - \eta \frac{\partial L}{\partial b_1}$$

$$W_2 \leftarrow W_2 - \eta \frac{\partial L}{\partial W_2}, \qquad b_2 \leftarrow b_2 - \eta \frac{\partial L}{\partial b_2}$$

- One full pass over the data = one **epoch**.
- Repeating many epochs drives the loss downward.
- The **learning rate** η controls how big each step is — the last tuning knob we need to discuss.

Learning Rate: Too Small, Just Right, Too Large





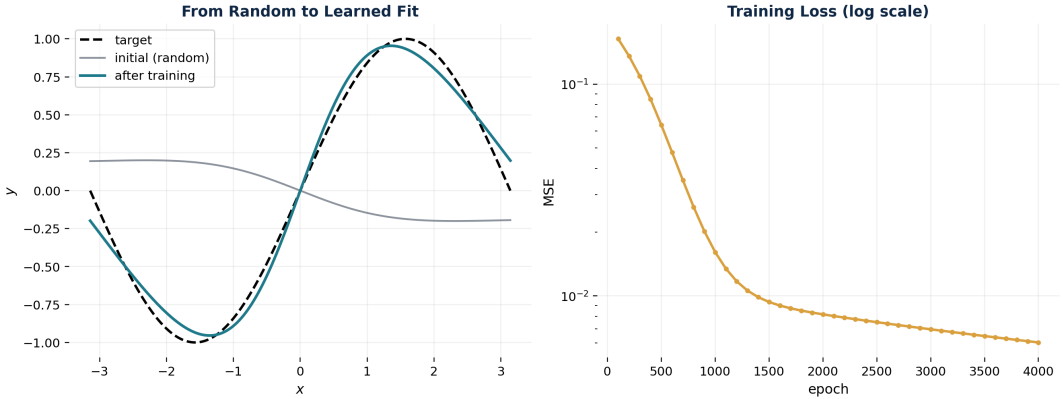
Left: loss curves on the same task for $\eta \in \{0.001, 0.01, 0.1, 1.0\}$. Right: schematic of step size on a quadratic loss surface.

Reading the Learning Rate Plot

- η **too small** (10^{-3}): loss decreases, but **slowly** — takes many epochs to make progress.
- η **well tuned** (10^{-2} here): smooth, steady decrease.
- η **moderately large** (10^{-1}): still converges, sometimes faster.
- η **too large** (1.0): loss **oscillates or diverges** — the update overshoots the minimum.

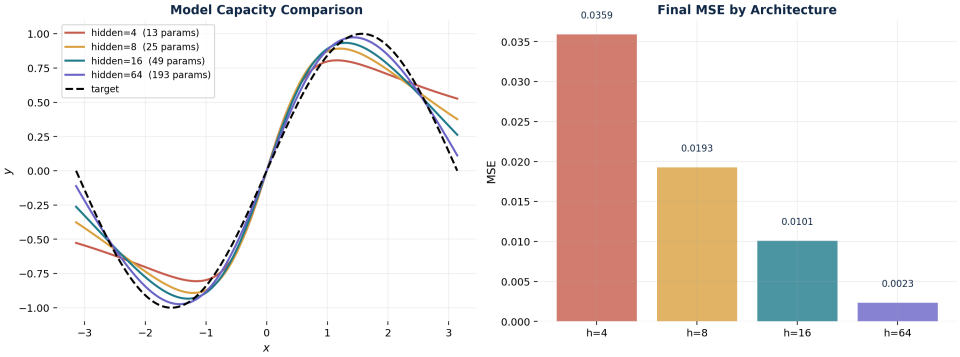
Practical tuning: try a few values on a log scale (e.g. $10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}$*)*
and watch the loss curve.

From Random Prediction to Learned Fit



Left: initial random prediction vs. trained fit. Right: loss curve over epochs.

Model Capacity Comparison



Capacity, Underfitting, and Overfitting

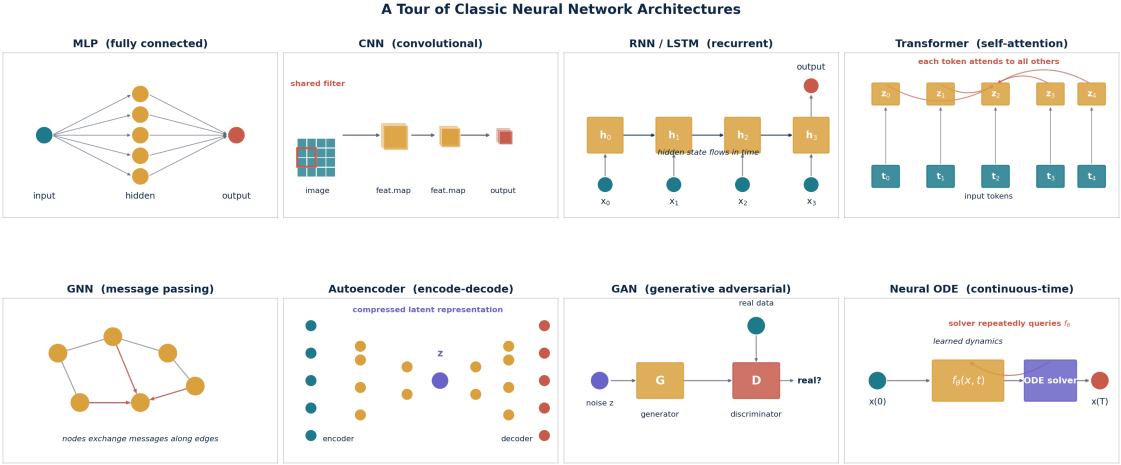
- **Too few hidden units**: cannot bend enough $\Rightarrow$ *underfitting*.
- **Well-chosen width**: matches the target shape.
- **Too many hidden units + small dataset**: risk of fitting noise $\Rightarrow$ *overfitting*.

Three knobs that control the capacity–data trade-off:

- architecture (width, depth)
- amount of training data
- regularization (weight decay, early stopping — later weeks)

For Week 4 the lesson is simple: **model structure matters**.

Beyond the MLP: A Tour of Classic Architectures



The MLP is the starting point. Different data structures led to specialized architectures — all still trained with backprop + gradient descent.

Architecture Zoo: Key Ideas at a Glance

Architecture	Data type	Key idea	Typical applications
MLP	tabular / vectors	fully connected layers	regression, small classification (this week)
CNN	images / grids	shared local convolutional filters	image classification, computer vision
RNN / LSTM	sequences	hidden state carried in time	time series, speech, early NLP
Transformer	sequences / sets	self-attention	modern NLP, LLMs, vision transformers
GNN	graphs	message passing between nodes	molecules, power grids, social networks
Autoencoder	any (unsupervised)	encode–decode bottleneck	compression, denoising, anomaly detection
GAN / Diffusion	any (generative)	match data distribution	image/audio generation, inverse problems
Neural ODE	continuous-time	layers $\rightarrow$ ODE solver	dynamics (Week 5+), physics-informed

What Each Architecture Adds on Top of the MLP

- **CNN**: replaces dense weights with *shared local filters*. Exploits spatial locality; drastically fewer parameters for images.
- **RNN / LSTM / GRU**: adds a *recurrent hidden state* so the network can remember context across time steps.
- **Transformer**: *attention* lets every token look at every other token directly. Scales to long sequences; foundation of modern LLMs.
- **GNN**: generalizes convolutions to *graphs* via message passing — well suited to power-system topologies.
- **Autoencoder / VAE**: learns a *compressed latent representation* without labels. Useful for pre-training and dimensionality reduction.
- **GAN / Diffusion**: trains a *generator* against a learned objective instead of labeled targets.
- **Neural ODE**: reinterprets network depth as *continuous time* — the bridge we take next week.

All still use backprop + gradient descent. The architecture is what changes.

Bridge 1: Toward Neural ODE

This week we learned a generic map

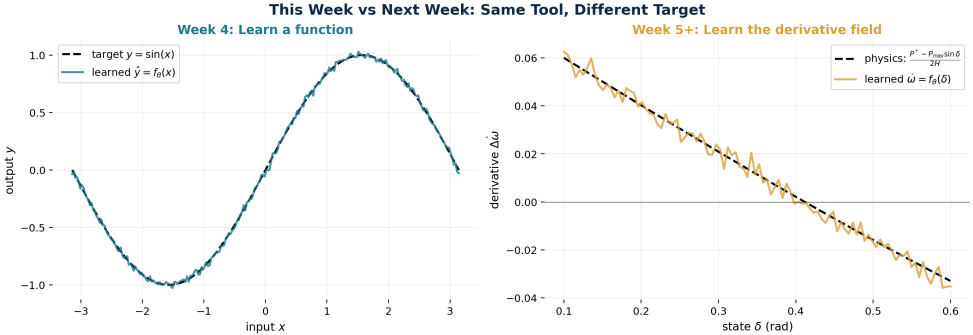
$$x \mapsto y$$

Next week we change the *meaning* of the target:

$$x \mapsto \dot{x} \qquad \text{or} \qquad (x, u) \mapsto \dot{x}$$

- The **model class** (MLP + backprop + gradient descent) is identical.
- The **target** becomes a derivative field instead of a curve.
- Wrapping the MLP inside an ODE solver gives a **Neural ODE**.

Bridge 2: Back to GFM Modeling



- **This week:** learn $x \mapsto y = \sin(x)$ (generic regression).
- **Next week:** learn $(\delta, \Delta\omega) \mapsto (\dot{\delta}, \dot{\Delta\omega})$ (derivative field).
- Same tool, different target meaning — connects to the swing equation from Weeks 2–3.

Mini Exercise

Compare and discuss:

- the straight-line fit vs. the trained neural-network fit
- what the hidden layer is contributing (look at the feature plot)
- hidden sizes 4, 16, and 64 — where does each fail?
- replace tanh with ReLU in the notebook; how does the fit change?
- try $\eta \in \{10^{-4}, 10^{-2}, 1.0\}$ — match the behavior on the learning-rate slide

Key Takeaways

1. A neural network is a **trainable nonlinear function approximator**: affine layers + activation functions.
2. The **activation function** is what gives depth its power — without it, the network collapses to a linear map.
3. **Training** is an optimization problem; the 4-step loop (forward $\rightarrow$ loss $\rightarrow$ backward $\rightarrow$ update) is the heart of it.
4. **Backpropagation** is the chain rule applied end-to-end to compute all parameter gradients efficiently.
5. A well-tuned **learning rate** is essential — too small wastes time, too large diverges.
6. **Capacity** (width/depth) controls what shapes the network can represent — too little underfits, too much risks overfitting.
7. This is the conceptual foundation for later **Neural ODE** models.