Neural Networks Basics

Lecture 03 — PHYG004, 2026 Spring

Prof. Young Woo Choi

Department of Physics, Sogang University

3/11/2026

Outline

Part 1: Introduction to Machine Learning

- What is ML? Programming with data vs explicit rules
- Four key components and their physics parallels
- Supervised learning: regression and classification
- Loss functions with physics connections

Part 2: From Linear Models to Neural Networks

- Linear regression, the single neuron, why nonlinearity matters
- MLPs: architecture, parameter counting, universal approximation
- Backpropagation and `jax.grad`
- SGD, initialization, and training in JAX/Flax
- Overfitting, underfitting, and generalization

Part 3: Wrap-up and Hands-on Preview

Part 1

Introduction to Machine Learning

Based on D2L.ai Ch. 1, adapted for physicists

What is Machine Learning?

Traditional Programming

- Human writes **explicit rules**
- Input + Rules $\longrightarrow$ Output
- Example: Classify Ising phases by computing $\langle |m| \rangle$ and comparing to threshold

Machine Learning

- Machine **learns rules from data**
- Input + Output $\longrightarrow$ Rules
- Example: Feed spin configurations + labels $\longrightarrow$ network learns to distinguish phases *without* knowing about m

$$\text{ML: Given } \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}, \quad \text{find } f_\theta \text{ such that } f_\theta(\mathbf{x}_i) \approx y_i$$

ML is powerful when: (1) the rules are too complex to write explicitly, (2) the rules change over time, or (3) we don't know the rules at all.

Four Key Components of ML

ML Component	Description	Physics Parallel
Data	Training examples $\{(\mathbf{x}_i, y_i)\}$	Experimental measurements
Model	Parameterized function f_θ	Theory / ansatz / trial wavefunction Ψ_θ
Objective function	Loss $\mathcal{L}(\theta)$ to minimize	Energy functional $E[\Psi]$ / action $S[\phi]$
Optimization	Algorithm to find θ^*	Variational principle / energy minimization

Recall from Lecture 02: We showed that ML training and the variational principle share **identical mathematical structure** — parameterized ansatz + objective + gradient descent. Today we build the concrete model f_θ.

Supervised Learning: Regression

Task: Predict a continuous output $y \in \mathbb{R}$ from input $\mathbf{x}$.

General ML examples:

- House price from features
- Stock price prediction

Physics examples:

- Predict total energy from atomic positions
- Predict band gap from crystal structure
- Predict scattering cross-section from kinematics

Setup:

- Features: $\mathbf{x}_i \in \mathbb{R}^d$ (inputs)
- Labels: $y_i \in \mathbb{R}$ (targets)
- Training set: $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$
- Test set: held-out data to evaluate generalization
- Goal: learn f_θ that generalizes to **unseen** data

Key distinction: Low training error is not enough. The model must perform well on **new data** it has never seen.

Supervised Learning: Classification

Task: Assign input $\mathbf{x}$ to one of C discrete classes.

Physics examples:

- Ising model: ordered vs disordered phase from spin configuration $\{s_i\}$
- Particle physics: signal vs background events
- Galaxy morphology: spiral vs elliptical

Output: probability distribution over classes

$$\hat{p}_c = P(y = c \mid \mathbf{x}; \theta)$$

Binary classification ($C = 2$):

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b) \in (0, 1)$$

Multi-class ($C > 2$): use softmax

$$\hat{p}_c = \frac{e^{z_c}}{\sum_{c'} e^{z_{c'}}}$$

Predict class: $\hat{c} = \arg\max_c \hat{p}_c$

In physics, classification often means **phase identification** — the ML analog of computing an order parameter.

Loss Functions

Regression: MSE

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(f_\theta(\mathbf{x}_i) - y_i)^2$$

Physics analogy: **harmonic potential**

$$V(x) = \tfrac{1}{2}k(x - x_0)^2$$

- Zero loss at prediction = target
- Quadratic near minimum
- Large errors penalized heavily

Classification: Cross-Entropy

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log \hat{p}_{ic}$$

Physics analogy: **negative log-likelihood** of the Boltzmann distribution

$$P_c = \frac{e^{-\beta E_c}}{Z}$$

- Logit $z_c \leftrightarrow -\beta E_c$
- Softmax = Boltzmann distribution

Softmax IS the Boltzmann distribution. Cross-entropy IS the negative log-likelihood. These are not analogies — they are **identical mathematics**.

Unsupervised Learning and Beyond

Unsupervised Learning

No labels — find structure in data

- **Clustering**: group similar data (e.g., galaxy types)
- **Dimensionality reduction**: PCA, autoencoders
- **Generative models**: learn $P(\mathbf{x})$ to generate new samples (flows, diffusion)

Physics: PCA $\sim$ collective coordinates / normal modes

Self-Supervised

Create labels from the data itself

- Predict masked parts of input
- Contrastive learning
- Foundation models (LLMs)

Physics: predict missing lattice sites from neighbors

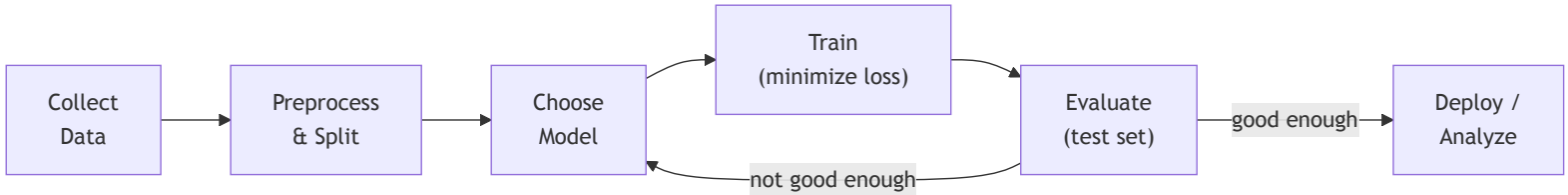
Reinforcement Learning

Agent learns by trial and error

- State $\rightarrow$ action $\rightarrow$ reward
- Maximize cumulative reward
- AlphaGo, robotic control

Physics: optimal experimental design, adaptive measurements

The ML Pipeline



Step	ML	Physics experiment analog
Collect data	Gather $(\mathbf{x}, y)$ pairs	Run simulations or experiments
Preprocess	Normalize, split train/test	Calibration, systematic error control
Choose model	MLP, CNN, Transformer…	Choose ansatz / theoretical framework
Train	Minimize $\mathcal{L}(\theta)$	Fit parameters to data
Evaluate	Test set performance	Cross-validation, blind analysis

Part 1 Summary

Concept	Key Point
Machine learning	Learn rules from data, not hand-code them
Four components	Data, Model, Objective, Optimization
Supervised learning	Regression (continuous) and Classification (discrete)
Loss functions	MSE $\sim$ harmonic potential; Cross-entropy $=$ Boltzmann neg. log-likelihood
Unsupervised & beyond	Clustering, generative models, RL — brief awareness
ML pipeline	Data $\rightarrow$ Model $\rightarrow$ Train $\rightarrow$ Evaluate $\rightarrow$ Deploy

Part 2

From Linear Models to Neural Networks

Linear Regression: The Simplest Model

Model: $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$

Physics example: Hooke's law

$$F = -kx$$

- Input: displacement x
- Output: force F
- Parameter: spring constant k
- Loss: $\mathcal{L}(k) = \frac{1}{N}\sum_i (F_i + kx_i)^2$

Closed-form solution

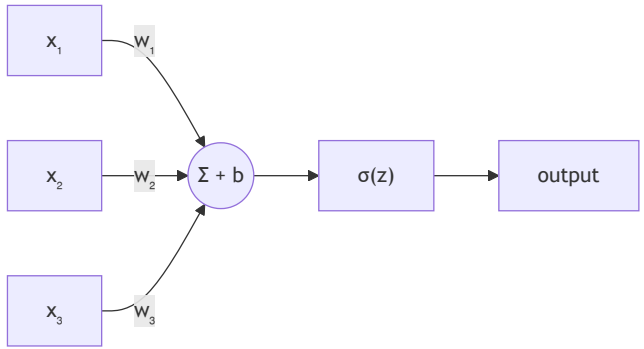
Minimize MSE analytically:

$$\nabla_{\mathbf{w}} \mathcal{L} = 0 \implies \mathbf{w}^* = (X^\top X)^{-1} X^\top \mathbf{y}$$

- Linear model = hyperplane in input space
- $\mathbf{w}$: normal to decision boundary
- b: offset

Linear regression has a **closed-form solution**. Neural networks do not — that's why we need gradient descent.

The Single Neuron



Equations:

$$z = \sum_i w_i x_i + b = \mathbf{w}^\top \mathbf{x} + b$$

$$\hat{y} = \sigma(z)$$

- **Without** activation σ: just linear regression
- **With** activation: nonlinear transformation

A single neuron computes: **linear combination of inputs** $\rightarrow$ **nonlinear activation**. This is the fundamental building block. Everything else is stacking these together.

A single neuron = linear regression + nonlinear activation function.

Why Nonlinearity?

Without activation functions, stacking layers is pointless:

$$\mathbf{h} = W_2(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = \underbrace{(W_2W_1)}_{W'}\mathbf{x} + \underbrace{(W_2\mathbf{b}_1 + \mathbf{b}_2)}_{\mathbf{b}'} = W'\mathbf{x} + \mathbf{b}'$$

Without activation

Any number of linear layers collapses to a **single linear layer**.

100 layers of linear transforms = 1 linear transform.

Can only represent hyperplanes.

With activation

Nonlinearity between layers prevents collapse.

Each layer genuinely increases expressiveness.

Can represent arbitrarily complex functions.

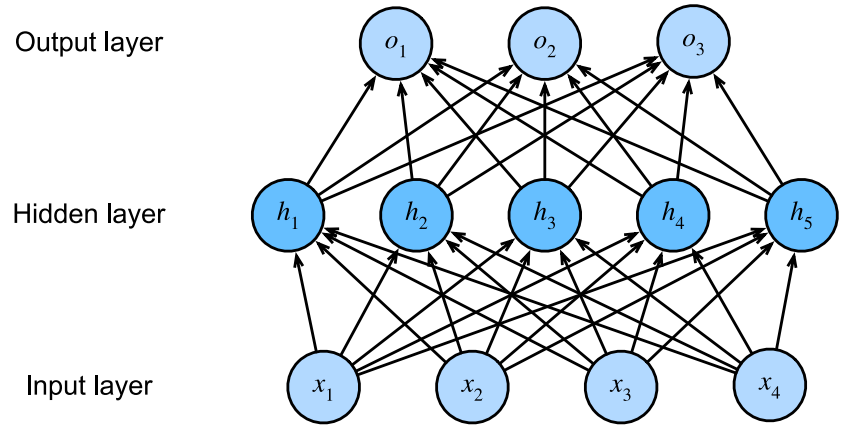
Activation Functions

Name	Formula	Range	Physics	Notes
Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$	$(0, 1)$	**Fermi-Dirac**	Saturates; vanishing gradients
Tanh	$\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$(-1, 1)$	Rescaled sigmoid	Zero-centered output
ReLU	$\max(0, z)$	$[0, \infty)$	Rectifier / half-wave	Default; can have dead neurons
GELU	$z \cdot \Phi(z)$	$\approx (-0.17, \infty)$	Gaussian-weighted gate	Used in Transformers; no dead neurons

Sigmoid and Fermi-Dirac: $\sigma(z) = f_{\mathrm{FD}}(E)$ with $z = (\mu - E)/k_B T$. Note the sign: sigmoid's argument **increases** as energy **decreases**. At $\mu = 0, k_B T = 1$: $f(E) = \sigma(-E)$.

Rule of thumb: ReLU/GELU for hidden layers (no saturation). Sigmoid/softmax only at the output when needed. Leaky ReLU or GELU avoid the "dead neuron" problem of ReLU.

Multi-Layer Perceptron (MLP)



Source: D2L.ai

Layer-by-layer equations:

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

$$\mathbf{h}^{(2)} = \sigma\left(W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right)$$

$$\hat{\mathbf{y}} = W^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}$$

Each layer: **affine transform** $\rightarrow$ **nonlinear activation**

No activation on the output layer (for regression).

Each layer performs a **learned coordinate transformation** — like canonical transforms in Hamiltonian mechanics, mapping to coordinates where the problem becomes trivial.

Parameter Counting

For an MLP with layer sizes $[n_0, n_1, \ldots, n_L]$:

$$\text{Total parameters} = \sum_{\ell=1}^{L} \left(n_{\ell-1} \cdot n_\ell + n_\ell \right)$$

Example: Input=3, Hidden=64, 64, Output=1

Layer	Weight matrix	Bias	Parameters
1	$3 \times 64 = 192$	64	256
2	$64 \times 64 = 4096$	64	4160
3	$64 \times 1 = 64$	1	65
Total			**4,481**

Universal Approximation Theorem

Theorem (Cybenko 1989, Hornik 1991):

A feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary accuracy.

Recall: We stated this result in Lecture 02. Here we unpack what it does and does not guarantee.

What this does NOT tell you:

- How **many** neurons you need (could be astronomically large)
- How much **data** is required to learn the approximation
- Whether gradient descent will **find** the right parameters
- Whether the solution **generalizes** to unseen data

MLP as Learned Coordinate Transforms

Geometric interpretation: each layer performs a coordinate transformation.

$$\mathbf{x} \xrightarrow{W^{(1)},\sigma} \mathbf{h}^{(1)} \xrightarrow{W^{(2)},\sigma} \mathbf{h}^{(2)} \xrightarrow{W^{(3)}} \hat{y}$$

Input space

Data is tangled — classes overlap in original coordinates

Hidden space

Layers progressively untangle the data

Output space

Data is linearly separable — final layer draws a line

Physics analogy: Like canonical transformations in Hamiltonian mechanics — transform to coordinates where the problem becomes trivial. The network learns the "right" coordinate system.

Width vs Depth

Shallow and Wide

$$[1] \to [1000] \to [1]$$

- 1 hidden layer, 1000 neurons
- ~2,000 parameters
- Can approximate any function (UAT)
- But may need exponentially many neurons

Deep and Narrow

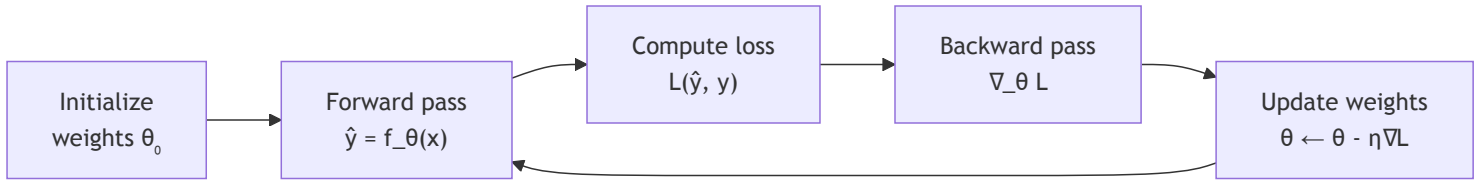
$$[1] \to [32] \to [32] \to [32] \to [1]$$

- 3 hidden layers, 32 neurons each
- ~2,200 parameters
- Learns **hierarchical features**
- More parameter-efficient for structured data

Why depth helps — compositionality: Layer 1 learns local features, Layer 2 combines them, Layer 3 builds abstractions.

Physics analogy: Reminiscent of RG coarse-graining — each layer builds effective descriptions at larger scales. (This analogy is made rigorous for specific architectures; see Mehta & Schwab, 2014.)

The Training Recipe



1. **Initialize** parameters randomly (break symmetry)
2. **Forward pass**: compute prediction $\hat{y} = f_\theta(\mathbf{x})$
3. **Loss**: measure error $\mathcal{L}(\hat{y}, y)$
4. **Backward pass**: compute gradients $\nabla_\theta \mathcal{L}$ via backpropagation
5. **Update**: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$
6. **Repeat** steps 2-5 until convergence

> Same structure as variational optimization in physics — just with more parameters.

Backpropagation: Chain Rule on Graphs

Goal: Compute $\frac{\partial \mathcal{L}}{\partial \theta}$ for every parameter θ. For a composition $\mathcal{L} = \ell \circ g \circ f$: $\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \ell}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial \theta}$

Worked example: $x \xrightarrow{w_1} z \xrightarrow{\text{ReLU}} h \xrightarrow{w_2} \hat{y} \xrightarrow{\text{MSE}} \mathcal{L}$, with $x = 2, w_1 = 0.5, w_2 = 1.5, y_{\text{target}} = 3$, $\mathcal{L} = (\hat{y} - y)^2$:

Direction	Computation	Value
Forward	$z = w_1 x = 1.0,\ h = \text{ReLU}(z) = 1.0,\ \hat{y} = w_2 h = 1.5$	$\mathcal{L} = (1.5 - 3)^2 = 2.25$
$\partial \mathcal{L} / \partial \hat{y}$	$2(\hat{y} - y)$	-3.0
$\partial \mathcal{L} / \partial w_2$	$(\partial \mathcal{L} / \partial \hat{y}) \cdot h$	-3.0
$\partial \mathcal{L} / \partial w_1$	$(\partial \mathcal{L} / \partial \hat{y}) \cdot w_2 \cdot \mathbf{1}_{z>0} \cdot x$	-9.0

Backprop = systematic application of the chain rule, right to left through the computation.

`jax.grad` Does This For You

Manual backpropagation (tedious, error-prone):

```
# Forward
z = w1 * x; h = jnp.maximum(0, z)
y_hat = w2 * h; loss = (y_hat - y)**2
# Backward
dl_dy = 2 * (y_hat - y); dl_dw2 = dl_dy * h
dl_dw1 = dl_dy * w2 * (z > 0) * x
```

With `jax.grad` (one line):

```
def loss_fn(params, x, y):
    w1, w2 = params
    h = jnp.maximum(0, w1 * x)
    return (w2 * h - y)**2

grads = jax.grad(loss_fn)(params, x, y)  # exact gradients, automatic
```

> JAX's automatic differentiation computes **exact** gradients (not numerical approximations) — same result as manual backprop, without the tedium.

Batch, Mini-Batch, and Stochastic GD

Variant	Gradient computed over	Updates per epoch
Batch GD	All N samples	1
Mini-batch GD	Random subset of B samples	N/B
SGD	Single sample ($B = 1$)	N

Mini-batch gradient (the standard in practice):

$$\nabla_\theta \mathcal{L} \approx \frac{1}{B} \sum_{i \in \text{batch}} \nabla_\theta \ell(f_\theta(\mathbf{x}_i), y_i)$$

Physics analogy: Mini-batch SGD $\sim$ **Langevin dynamics**: $\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}(\theta_t) + \underbrace{\text{noise from mini-batch}}_{\sim \sqrt{\eta/B}\, \xi_t}$

Weight Initialization and Gradient Pathologies

Initialization matters: all-zero weights $\rightarrow$ all neurons identical $\rightarrow$ no learning (symmetry problem).

Xavier (sigmoid/tanh): $w \sim \mathcal{N}(0, \frac{2}{n_{\text{in}}+n_{\text{out}}})$

He (ReLU): $w \sim \mathcal{N}(0, \frac{2}{n_{\text{in}}})$

Principle: keep signal magnitude roughly constant across layers.

Deep networks: gradients are products of many factors.

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} \propto \prod_{\ell} \sigma'(z^{(\ell)})$$

Vanishing: $|\sigma'| < 1$

Sigmoid: $\sigma'_{\max} = 0.25$

10 layers: $0.25^{10} \approx 10^{-6}$

Exploding: $\|W\sigma'\| > 1$

Gradients blow up

NaN losses

Solutions: ReLU (gradient = 0 or 1), proper initialization (He/Xavier), residual connections, normalization layers.

MLP in Pure JAX

```python
import jax, jax.numpy as jnp

def init_mlp(key, layer_sizes):
    """Initialize MLP parameters with He initialization."""
    params = []
    for i in range(len(layer_sizes) - 1):
        key, subkey = jax.random.split(key)
        n_in, n_out = layer_sizes[i], layer_sizes[i + 1]
        W = jax.random.normal(subkey, (n_in, n_out)) * jnp.sqrt(2.0 / n_in)
        b = jnp.zeros(n_out)
        params.append((W, b))
    return params

def mlp_forward(params, x):
    """Forward pass through MLP with ReLU activations."""
    for W, b in params[:-1]:
        x = jax.nn.relu(x @ W + b)
    W, b = params[-1]
    return x @ W + b  # no activation on output layer

# Usage
params = init_mlp(jax.random.PRNGKey(42), [3, 64, 64, 1])
print(params[0][0].shape)  # W1: (3, 64)
y_pred = mlp_forward(params, x)  # x shape: (N, 3) → y_pred: (N, 1)
```

MLP in Flax NNX

```python
import flax.nnx as nnx
import optax

class PhysicsMLP(nnx.Module):
    def __init__(self, in_size, hidden_sizes, out_size, *, rngs):
        sizes = [in_size] + hidden_sizes + [out_size]
        self.layers = []
        for i in range(len(sizes) - 1):
            self.layers.append(nnx.Linear(sizes[i], sizes[i + 1], rngs=rngs))

    def __call__(self, x):
        for layer in self.layers[:-1]:
            x = nnx.relu(layer(x))
        return self.layers[-1](x)

# Create model: 3 inputs → [64, 64] hidden → 1 output
model = PhysicsMLP(3, [64, 64], 1, rngs=nnx.Rngs(42))
```

- `nnx.Linear(in_features, out_features)` stores W and $\mathbf{b}$ internally
- `nnx.relu` = `jax.nn.relu`
- Flax handles parameter management; JAX handles the math

> Flax is syntactic sugar over JAX. Understand the pure JAX version first.

Training Loop in Flax NNX

```python
# Setup optimizer (NNX wraps optax)
optimizer = nnx.Optimizer(model, optax.sgd(learning_rate=0.01), wrt=nnx.Param)

# Loss function
def loss_fn(model, x, y):
    pred = model(x)
    return jnp.mean((pred - y) ** 2)

# JIT-compiled training step
@nnx.jit
def train_step(model, optimizer, x, y):
    loss, grads = nnx.value_and_grad(loss_fn)(model, x, y)
    optimizer.update(model, grads)  # updates model parameters in-place
    return loss

# Training loop
for epoch in range(1000):
    loss = train_step(model, optimizer, x_train, y_train)
    if epoch % 200 == 0:
        print(f"Epoch {epoch}: loss = {loss:.4f}")
```

The standard pattern: forward → loss → backward → update → repeat. `nnx.Optimizer` handles the update step cleanly.

Part 2 Summary

$$\text{Linear model} \xrightarrow{+\text{ activation}} \text{Neuron} \xrightarrow{+\text{ layers}} \text{MLP}$$

Concept	Key Idea
Linear regression	$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ — closed-form solution exists
Why nonlinearity	Stacking linear layers collapses to one; activation prevents this
Activation functions	ReLU for hidden layers; sigmoid = Fermi-Dirac (with sign flip)
MLP	Stack of affine + nonlinear layers; universal approximator
Backpropagation	Chain rule applied backward through computational graph
`jax.grad`	One-line automatic differentiation — exact gradients
Mini-batch SGD	Noisy gradients $\sim$ Langevin dynamics

Key message: Training = repeated application of the chain rule to update parameters. `jax.grad` automates this entirely.

Training Error vs Generalization Error

Training error — performance on data we trained on:

$$R_{\text{emp}} = \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(\mathbf{x}_i),\, y_i)$$

Generalization error — expected performance on **unseen** data from the same distribution:

$$R = \mathbb{E}_{(\mathbf{x},y)\sim P_{\text{data}}}[\ell(f_\theta(\mathbf{x}),\, y)]$$

We can never compute R exactly — we estimate it with a **test set**.

The fundamental question:

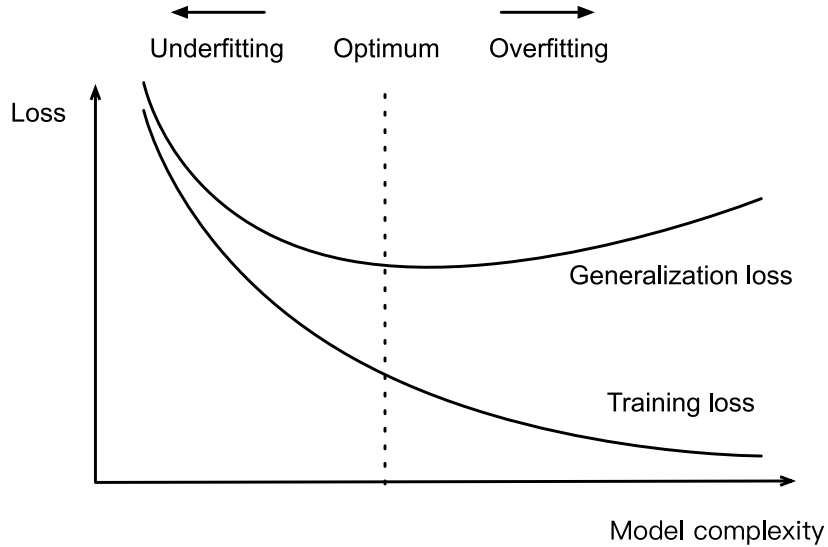
> When does success on **observed** data justify conclusions about **unobserved** data?

This is the same question physicists ask: when does a model fitted to experimental data make valid **predictions**?

- Kepler's laws: fitted to Tycho's data → predicted future orbits ✓
- Ptolemy's epicycles: fitted the data → no predictive power ✗

Low training error alone is meaningless. What matters is the **generalization gap**: $R - R_{\text{emp}}$.

Overfitting and Underfitting



Source: D2L.ai

Underfitting (too simple)

- High training error, high test error
- Model cannot capture the pattern
- Physics: fitting a straight line to $\sin(x)$

Sweet spot (just right)

- Low training error, low test error
- Model captures signal, ignores noise

Overfitting (too complex)

- Low training error, **high test error**
- Model memorizes noise
- Physics: fitting degree-100 polynomial to 20 data points

Model Complexity and Data

What controls overfitting?

Factor	More →
Model parameters	More capacity → easier to overfit
Training data	More data → harder to overfit
Parameter range	Wider weights → more capacity
Feature count	More features → more capacity

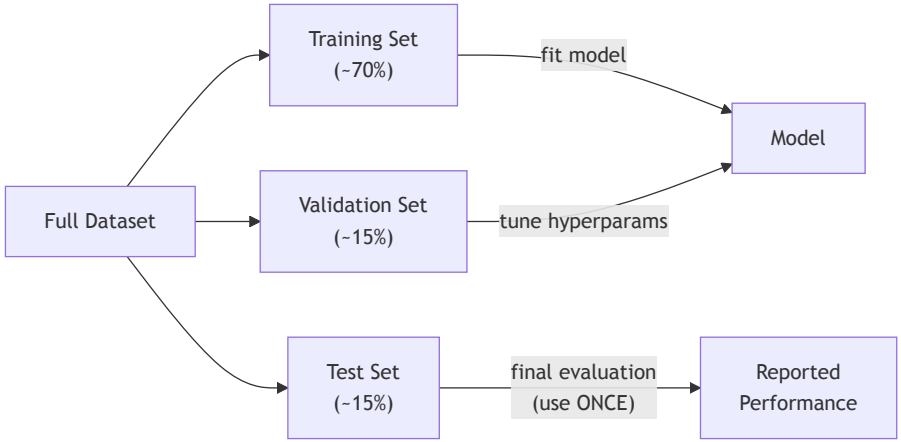
Polynomial example: fit $y = f(x) + \epsilon$ with degree-d polynomial

$$\hat{y} = \sum_{i=0}^{d} w_i x^i$$

- $d = 1$: underfits (can't capture curvature)
- $d = 3$: good fit (matches true structure)
- $d = 20$: overfits (wiggles through noise)

Physics parallel: fitting a Hamiltonian with too many coupling terms — perfect fit to data, but physically meaningless and poor predictions.

Train / Validation / Test Split



Why three splits?

- **Training**: fit θ (weights)
- **Validation**: choose hyperparameters (learning rate, architecture, ...)
- **Test**: final unbiased estimate of generalization

K-fold cross-validation (when data is scarce):

1. Split data into K folds
2. Train K times, each time holding out 1 fold
3. Average validation scores

Part 3

Wrap-up

Physics ↔ Neural Networks: Analogy Table

Neural Networks	Physics
Loss function $\mathcal{L}(\theta)$	Energy / Action $S[\phi]$
Gradient descent	Energy minimization
Mini-batch SGD noise	Thermal fluctuations / Langevin dynamics
Softmax / Sigmoid	Boltzmann / Fermi-Dirac distribution
Parameters (weights)	Variational parameters / couplings
Hidden representations	Collective coordinates / order parameters
Depth (many layers)	RG coarse-graining (in specific architectures)
Weight initialization / Overfitting	Initial conditions / finite-size effects

Many of these share **identical mathematical structures**; others are productive analogies that guide intuition.

Hands-on Preview

Today's notebook: Fitting QHO Wavefunctions

1. **Linear regression**: Fit Hooke's law $F = -kx$ with `jax.grad`
2. **Build MLP from scratch**: Pure JAX, He initialization
3. **Quantum Harmonic Oscillator**: Train MLP to approximate $\psi_n(x)$
   - Ground state $\psi_0(x)$: Gaussian — easy for any activation
   - Excited states ψ_1, ψ_2, ψ_3: nodes require expressive networks
4. **Activation function comparison**: How sigmoid/ReLU/tanh handle oscillatory functions
5. **Backprop verification**: Compare `jax.grad` vs finite differences
6. **Flax version**: Same problem with `flax.nnx` and `optax`

> The QHO is an ideal test case: analytic solutions for verification, rich structure (nodes, decay), and direct connection to the variational principle.

Key Takeaways

1. **ML = learning rules from data**. The four components (data, model, loss, optimizer) map directly to physics concepts (measurements, ansatz, energy, variational principle).

2. **Neural network = composition of affine transforms + nonlinear activations**. Each piece is simple; the composition is powerful enough to approximate any continuous function.

3. **Backpropagation = chain rule on computational graphs**. `jax.grad` computes this automatically and exactly.

4. **Training is optimization on a loss landscape** — identical to energy minimization in physics. The goal is **generalization**: low error on unseen data, not just memorizing the training set.

Next: How to train *well* — optimization algorithms, regularization, and inductive bias (Lecture 04).

Coming Next

Lecture 04 (Week 4): Optimization, Regularization, and Inductive Bias

- Momentum, Adam, and learning rate scheduling
- Regularization (L2, dropout, early stopping)
- Bias-variance tradeoff
- Double descent phenomenon
- Inductive bias in architecture choice

Lecture 05 (Week 5): Convolutional Neural Networks

- Translation equivariance by design
- Convolution, pooling, feature maps
- Physics application: Ising phase classification

Today we built the foundation. Next week: how to train *better* — advanced optimizers, regularization, and inductive bias.

References

- Zhang, A., Lipton, Z. C., Li, M. & Smola, A. J. *Dive into Deep Learning*. Cambridge University Press (2023) — Chapters 1, 3.6. d2l.ai
- Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* **65**, 386 (1958)
- Rumelhart, D. E., Hinton, G. E. & Williams, R. J. Learning representations by back-propagating errors. *Nature* **323**, 533 (1986)
- Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems* **2**, 303 (1989)
- Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural Networks* **4**, 251 (1991)
- Glorot, X. & Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. *AISTATS* (2010)
- He, K. et al. Delving deep into rectifiers. *ICCV* (2015)
- Mehta, P. & Schwab, D. J. An exact mapping between the variational renormalization group and deep learning. *arXiv:1410.3831* (2014)
- Goodfellow, I., Bengio, Y. & Courville, A. *Deep Learning*. MIT Press (2016) — Chapters 6-8