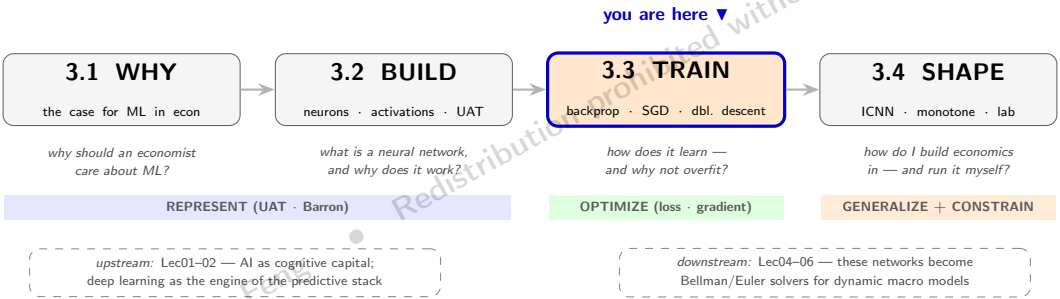
AI for Economic Research: Dynamic Models, Language, and Agents
Lecture 3.3: Forward Propagation, Backpropagation & Training

Zhigang Feng

2026-07-07 10:05:22

Lecture 3 in One Map

One lecture, four decks, one goal: make neural networks a working tool for economists — why they matter, what they are, how they learn, and how to make them respect economics.



Act 3.3 (here) *is the engine room: the forward pass computes predictions, backpropagation organizes the chain rule, SGD/Adam descend the loss, regularization disciplines the fit — and double descent explains why over-parameterization helps rather than hurts.*

Forward Propagation

From input to prediction, one affine-plus-activation layer at a time

Feedforward: Computing Outputs Layer by Layer

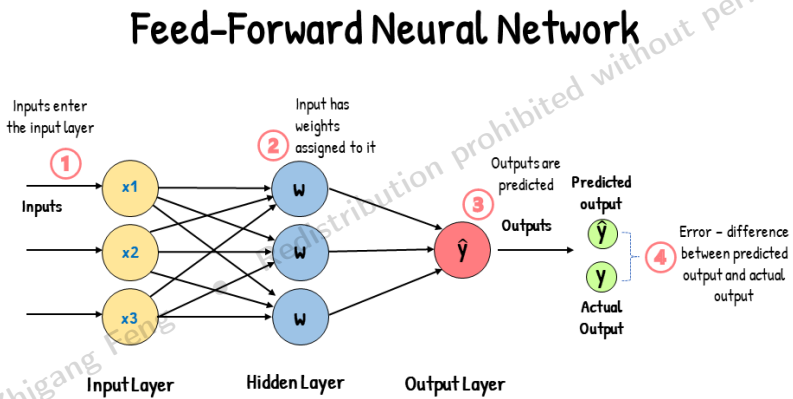
- **Forward propagation** (feedforward): compute the network output given an input by passing through layers sequentially.
- For a network with L hidden layers:

$$\mathbf{h}^{(0)} = \mathbf{x}, \qquad \mathbf{h}^{(\ell)} = \sigma\left(W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}\right), \quad \ell = 1, \ldots, L$$

$$\hat{\mathbf{y}} = W^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}$$

- Each layer: (1) multiply by weight matrix, (2) add bias, (3) apply activation.
- **Analogy:** Forward propagation is like solving a recursive model forward in time — given initial state x, compute successive transformations to arrive at the output.

Forward Pass: Visual



Toy Example: Training a Digit Classifier on One Sample

Recall from Lec. 3 T1: we motivated ML with handwritten digit recognition.

The toy numbers on the next slide come from the simplest version of that classifier:

- *Input* x: one feature from a single digit image (in practice, the flattened 784-dim 28×28 pixel vector).
- *Hidden + output:* sigmoid neurons producing $\hat{y} \in [0, 1]$ — the predicted probability that the image is, say, a "3".
- *Target* t: the true label (1 if it *is* a 3, 0 otherwise).
- *Loss* L: squared error (used here for clarity; in practice cross-entropy).

Why one sample? The math is the same as training on millions: one forward pass, one loss, one backward pass, one update — then repeat.

Toy Example: Forward Pass with Numbers

A tiny network makes every step visible: input $x = 1.0$, one sigmoid hidden neuron, one sigmoid output neuron.

The network, in full:

$$z_1 = w_1 x + b_1, \qquad h = \sigma(z_1)$$

$$z_2 = w_2 h + b_2, \qquad \hat{y} = \sigma(z_2)$$

$$L = \tfrac{1}{2}(\hat{y} - t)^2$$

Parameters $w_1{=}0.5,\ b_1{=}0,\ w_2{=}2.0,\ b_2{=}0$; target $t{=}0$.

With the numbers:

$$z_1 = 0.5 \ \Rightarrow\ h = \sigma(0.5) \approx 0.62$$

$$z_2 = 1.24 \ \Rightarrow\ \hat{y} = \sigma(1.24) \approx 0.78$$

$$L = \tfrac{1}{2}(0.78)^2 \approx 0.30$$

Reading the numbers:

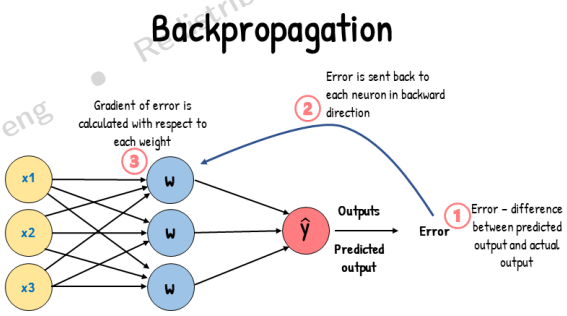
- Sigmoid σ is a *soft on/off switch*: $z_1{=}0.5 \Rightarrow h{=}0.62$, so the hidden neuron is "62% on."
- $\hat{y} = 0.78$ is the network's *predicted probability* that the image is a "3."
- But the truth is $t = 0$ (it is *not* a 3): the model is **confidently wrong** $\Rightarrow$ loss ≈ 0.30.
- **Learning** = nudge w_1, w_2 so $\hat{y}$ moves toward t. *How much* each moves is what backprop computes next.

Backpropagation and Gradient Descent

The chain rule, organized: how a network computes its own gradients

Backpropagation: Learning from Errors

- **Goal:** Compute $\frac{\partial L}{\partial \theta}$ for every parameter θ in the network.
- **Method:** Apply the **chain rule** backwards through the network layers.
- **Steps:**
  1. Forward pass: compute all activations and the loss
  2. Backward pass: propagate error signal from output to input
  3. Compute gradient of loss w.r.t. each weight and bias
  4. Update parameters: $\theta \leftarrow \theta - \eta \cdot \nabla_\theta L$



Toy Example: Backpropagation — Output Layer

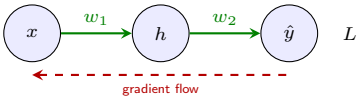
The whole network in five lines — same example ($x = 1.0,\ t = 0,\ w_1 = 0.5,\ w_2 = 2.0,\ b_1 = b_2 = 0$):

$$z_1 = w_1 x + b_1, \quad h = \sigma(z_1), \quad z_2 = w_2 h + b_2, \quad \hat{y} = \sigma(z_2), \quad L = \tfrac{1}{2}(\hat{y} - t)^2.$$

Forward gave $h \approx 0.62,\ \hat{y} \approx 0.78,\ L \approx 0.30$. Backprop builds $\partial L/\partial w_2$ then $\partial L/\partial w_1$ by the **chain rule**, output layer first.

Chain the links $w_2 \to z_2 \to \hat{y} \to L$**:**

$$\frac{\partial L}{\partial w_2} = \underbrace{\frac{\partial L}{\partial \hat{y}}}_{\hat{y}-t} \underbrace{\frac{\partial \hat{y}}{\partial z_2}}_{\hat{y}(1-\hat{y})} \underbrace{\frac{\partial z_2}{\partial w_2}}_{h} = 0.78 \times 0.17 \times 0.62 \approx 0.08$$



fwd: compute $\hat{y}$**,** L

bwd: chain rule

The first two factors reusably combine into the **error signal**

$$\delta_2 \equiv \frac{\partial L}{\partial z_2} = (\hat{y} - t)\,\hat{y}(1 - \hat{y}) \approx 0.13,$$

so $\dfrac{\partial L}{\partial w_2} = \delta_2\, h$ and $\dfrac{\partial L}{\partial b_2} = \delta_2 \approx 0.13$. *(We reuse* δ_2 *for the hidden layer next.)*

Read it: $\partial L/\partial w_2 > 0$, so nudging w_2 up would *raise* L — descent *decreases* w_2 (the output overshot t=0).

Toy Example: Backpropagation — Hidden Layer & Update

Propagate the error signal back one layer (from the output slide, $\delta_2 \equiv \partial L/\partial z_2 \approx 0.13$):

$$\frac{\partial L}{\partial h} = \underbrace{\frac{\partial L}{\partial z_2}}_{\delta_2} \cdot w_2 = 0.13 \times 2.0 \approx 0.26$$

$$\frac{\partial h}{\partial z_1} = h(1-h) \approx 0.62 \times 0.38 \approx 0.24 \quad \Rightarrow \quad \delta_1 \equiv \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial z_1} \approx 0.06$$

$$\frac{\partial L}{\partial w_1} = \delta_1 \cdot x = 0.06 \times 1.0 \approx 0.06$$

Parameter update (learning rate $\eta = 0.1$):

$$w_2^{\text{new}} \approx 2.0 - 0.1 \times 0.08 = 1.992, \qquad w_1^{\text{new}} \approx 0.5 - 0.1 \times 0.06 = 0.494$$

Key point: This is exactly what PyTorch's `.backward()` and the optimizer do — just at massive scale with millions of parameters.

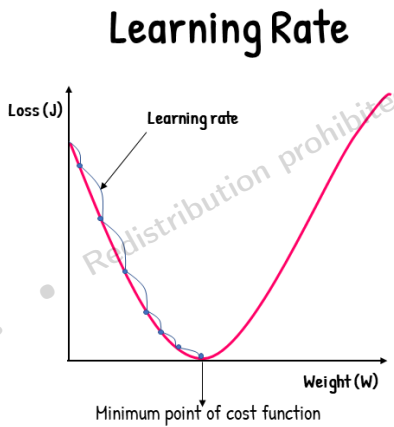
Gradient Descent: The Optimization Engine

- **Core update rule:** $\theta_{n+1} \leftarrow \theta_n - \lambda_n \, g(b_n; \theta_n)$
- **Variants by batch size n:**
  - $n = 1$: **Stochastic Gradient Descent (SGD)** — one random sample per step
  - $n = N$: **Full Gradient Descent (GD)** — all data per step
  - $1 < n < N$: **Mini-batch GD** — the practical sweet spot
- **Stochastic gradient approximation:**

$$\frac{d\mathcal{L}(\Gamma_\gamma)}{d\gamma} \approx \frac{1}{n} \sum_{i=1}^{n} \frac{d\mathcal{L}(\Gamma_\gamma; b_i)}{d\gamma}$$

- **Algorithm:**
  1. Initialize parameters randomly
  2. Draw random mini-batch b_n; compute stochastic gradient $g(b_n; \theta_n)$
  3. Choose learning rate $\lambda_n > 0$; update $\theta_{n+1} \leftarrow \theta_n - \lambda_n \, g(b_n; \theta_n)$
  4. Repeat until convergence

Gradient Descent: Visual



Adaptive Learning Rates: RProp → RMSProp → Adam

- **RProp** (Riedmiller & Braun, 1993): a *full-batch* method that sizes each parameter's step from the *sign* of its gradient alone — robust, but the signs get too noisy under mini-batching.
- **RMSProp** (Hinton, 2012): RProp adapted to mini-batches — give each parameter its own step by dividing by a running average of *squared* gradients:

$$g_k \leftarrow \frac{1}{n}\sum_{i=1}^{n} \nabla \mathcal{L}(\Gamma_\gamma; b_i), \qquad r_{k+1} \leftarrow \rho\, r_k + (1-\rho)\, g_k \odot g_k$$

$$\Delta\theta_{k+1} = -\frac{\lambda}{\sqrt{\delta + r_{k+1}}} \odot g_k \qquad \text{(big-gradient directions damped, small ones amplified).}$$

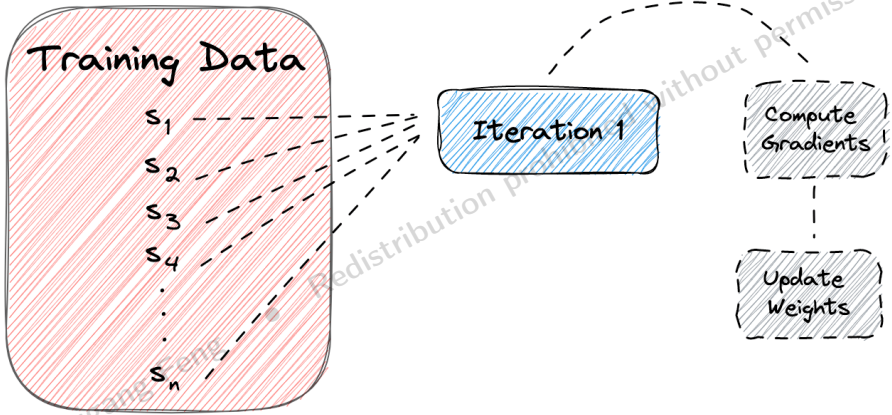
- **Adam** (Kingma & Ba, 2015): momentum + RMSProp — tracks moving averages of the gradient (1st moment) and squared gradient (2nd moment); decay rates ρ_1, ρ_2.

"Dominant default through ∼2022" means: from its 2015 debut until roughly 2022, Adam was the optimizer nearly everyone reached for *by default*, with no tuning — the safe out-of-the-box choice — before AdamW and others (next slide) displaced it for large models.

Beyond Adam: The Modern Optimizer Landscape (2024–2026)

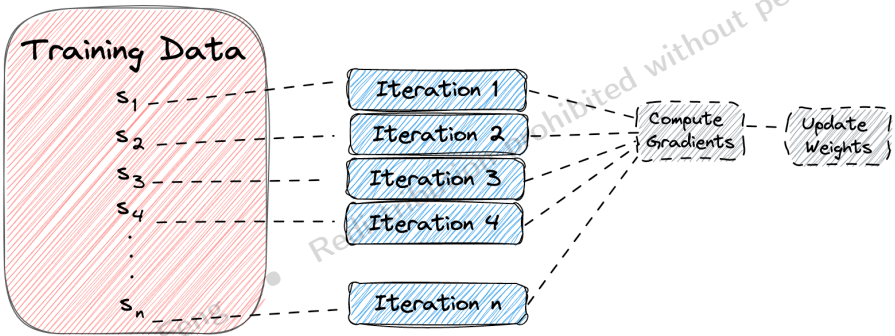
- **AdamW** (Loshchilov & Hutter, 2019): decoupled weight decay; the de-facto default for transformers.
- **Lion** (Chen et al. 2023): sign-based update; lower memory than Adam, competitive on large models.
- **Sophia** (Liu et al. 2023): cheap second-order optimizer using a diagonal Hessian estimate; faster pre-training convergence.
- **Schedule-free optimizers** (Defazio et al. 2024): remove the learning-rate schedule altogether — the optimizer self-regulates.
- **Course default for all labs (5A/5B, 6A/6B):** `torch.optim.AdamW(lr=1e-3, weight_decay=1e-2)`. Vary only when explicitly noted.
- **Economist's take:** optimizer choice is the ML analogue of which numerical solver to use — the right pick depends on problem geometry. For macro-scale networks (10^3–10^6 params), AdamW is the safe default.

Batch vs. Stochastic vs. Mini-Batch



Batch GD: Smooth convergence path, but each step is expensive (uses all data).

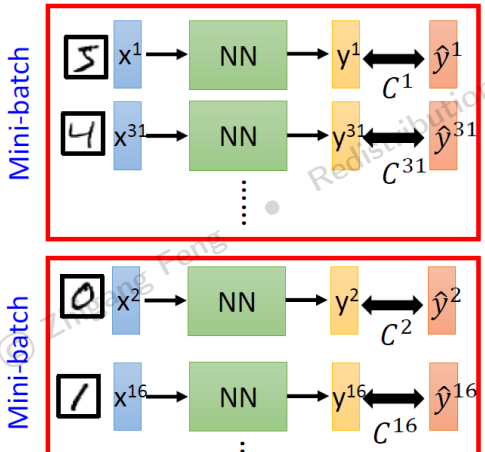
Stochastic Gradient Descent



SGD: Noisy path with high variance, but each step is cheap and can escape local minima.

Mini-Batch Gradient Descent

We do not really minimize total loss!



- Randomly initialize network parameters
- Pick the 1st batch

  $L' = C^1 + C^{31} + \cdots$

  Update parameters once
- Pick the 2nd batch

  $L'' = C^2 + C^{16} + \cdots$

  Update parameters once

  ⋮
- Until all mini-batches have been picked

one epoch

Why Mini-Batch? Four Reasons

1. **Computational efficiency:** More frequent parameter updates than full GD; faster convergence in wall-clock time.
2. **Memory constraints:** Large datasets may not fit in GPU memory. Mini-batches process manageable subsets.
3. **Implicit regularization:** The stochasticity in gradient estimates helps escape local minima and saddle points — acts as a form of regularization.
4. **Parallelization:** Each mini-batch can be processed across multiple GPUs simultaneously.

Economist's note: Mini-batch SGD is the computational analogue of sampling-based estimation in econometrics — trade exactness for speed, and the noise is actually helpful.

Convergence Theory: The Guarantee

Convergence with a Diminishing Learning Rate

Under standard assumptions (Lipschitz gradient, bounded gradient-noise variance), if the learning-rate sequence satisfies

$$\sum_{n=1}^{\infty} \lambda_n = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty} \lambda_n^2 < \infty,$$

then the running-average expected squared gradient vanishes:

$$\mathbb{E}\left[\frac{1}{\Delta_N}\sum_{n=1}^{N} \lambda_n \|\nabla F(\theta_n)\|^2\right] \xrightarrow{N\to\infty} 0, \qquad \Delta_N = \sum_{n=1}^{N} \lambda_n.$$

— Bottou, Curtis & Nocedal (2018). *Optimization Methods for Large-Scale ML*.

- **In words:** with the step size shrinking at the right rate, SGD drives its *average* gradient to zero — it settles at a stationary point. *Why* each of the two conditions is needed: next slide.

Convergence Theory: The Robbins–Monro Conditions, Intuitively

The two conditions are the **Robbins–Monro conditions** (Robbins & Monro, 1951), foundation of *stochastic approximation* — driving a noisy signal to its root. That is exactly SGD: the mini-batch gradient is a *noisy estimate* of the true gradient.

$\sum \lambda_n = \infty$ — **"go the distance."** Steps must sum to infinity so the iterate can travel *any* distance to reach the optimum, however far the start. If λ_n decays too fast, the total path length is finite and training **stalls short**.

$\sum \lambda_n^2 < \infty$ — **"kill the noise."** Squared steps must be summable so accumulated *gradient noise* stays finite and dies out. This forces $\lambda_n \to 0$ fast enough that late steps stop bouncing and the iterate **settles**.

- **Concretely:** $\lambda_n = 1/n$ satisfies both ($\sum 1/n = \infty$, $\sum 1/n^2 = \pi^2/6 < \infty$) — the textbook schedule. $\lambda_n = 1/\sqrt{n}$ satisfies the first but *fails* the second (too noisy). A *constant* λ also fails the second — SGD then bounces forever in a ball around the optimum (early exploration, not final convergence).
- **Econometrics link:** the same Robbins–Monro recursion underlies stochastic approximation and recursive/online estimation. *ICNN-constrained training (T4; Lec. 4 T4) restores convergence for value/cost-function applications.*

Training Process: Loss, Overfitting, Regularization

Choosing the loss and taming overfitting — the ML analogue of disciplined econometrics

Loss Functions I: Regression (Mean Squared Error)

- **Mean Squared Error (MSE)** — the loss for *continuous* targets:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

  the average squared gap between prediction $\hat{y}_i$ and target y_i. Big misses are penalized quadratically.

- **Why this form?** Minimizing MSE is *exactly* maximum likelihood under Gaussian errors — for a linear model it *is* OLS. The loss is not arbitrary: it encodes an assumption about the noise.

- **When to use it:** GDP growth, yields, prices, and (Lec. 4) Euler-equation residuals — anything real-valued.

- **Economist's bridge:** MSE $\Leftrightarrow$ OLS / Gaussian MLE. Swap the noise model and you get other losses: Poisson $\rightarrow$ count data, quantile (pinball) $\rightarrow$ quantile regression.

Loss Functions II: Classification (Cross-Entropy)

Setup. A classifier chooses among K *classes* — for handwritten digits, $K = 10$ (the labels $0, 1, \ldots, 9$). The network emits a raw score z_k per class; **softmax** turns the scores into probabilities that sum to one:

$$\hat{y}_k = \text{softmax}(z)_k = \frac{e^{z_k}}{\sum_{j=0}^{K-1} e^{z_j}}, \qquad \sum_{k=0}^{K-1} \hat{y}_k = 1.$$

The label is "one-hot." The ground truth $y = (y_0, \ldots, y_{K-1})$ is a vector with a single 1 at the *true* class and 0 everywhere else — "one-hot" = exactly one entry hot. A true "3" is $y = (0, 0, 0, \underset{k=3}{1}, 0, \ldots, 0)$.

Cross-entropy loss:

$$\mathcal{L}_{\text{CE}} = -\sum_{k=0}^{K-1} y_k \log \hat{y}_k \;=\; -\log \hat{y}_{k^\star}$$

Because y is one-hot, every term vanishes except the true class $k^\star$: **CE is just the negative log-probability the model assigned to the correct answer.** Minimizing it = maximizing that probability.

Why this shape is right:

- true-class prob $\hat{y}_{k^\star}$=0.99 $\Rightarrow$ $L = -\log 0.99 \approx 0.01$ (tiny)
- $\hat{y}_{k^\star}$=0.1 $\Rightarrow$ $L = -\log 0.1 \approx 2.3$ (large)
- confident *and* right $\rightarrow$ near 0; confident *but* wrong $\rightarrow$ blows up.

Economist's bridge: this is *exactly* McFadden's (1973) multinomial-logit MLE — softmax is the logit choice probability, one-hot y is the observed choice, minimizing CE is maximizing the log-likelihood. *Classification = discrete choice.* (Our binary toy digit was the K=2 case; softmax generalizes it to all ten digits.)

Training Terminology

- **Epoch:** One complete pass through the entire training dataset.
- **Mini-batch:** A small subset of training data used for one gradient update.
- **Iteration:** One parameter update (= one mini-batch processed).
  - If dataset has 10,000 samples and batch size is 100, one epoch = 100 iterations.
- **Automatic Differentiation:** Computational technique for exact gradient computation — the engine behind `loss.backward()` in PyTorch.
- **Episode:** (Reinforcement learning context) One sequence of steps from initial state to terminal state — analogous to one simulated life-cycle path.

Overfitting and Stopping Criteria

- **The fundamental tension:**
  - Training loss: always decreasing with more epochs
  - Validation loss: decreases, then **increases** $\rightarrow$ overfitting
- **Stopping criteria:**
  1. Fixed number of epochs (simple but suboptimal)
  2. Monitor validation loss — stop when it stops improving
  3. **Early stopping:** halt training when validation loss increases for k consecutive epochs
- **Model selection:** Choose the model (epoch checkpoint) with best validation performance.
- **Economist's analogy:** Overfitting is the ML version of data mining — the model memorizes noise in the training sample rather than learning the true data-generating process. Out-of-sample validation is the discipline.

Local Optima vs. Global Optimum

- **Non-convex loss landscape:**
  - Neural network loss functions are *not* convex — many local optima and saddle points
  - Global optimum exists but is generally unreachable in practice
- **Mitigation strategies:**
  - Random initialization of weights (different starting points)
  - Momentum and adaptive learning rates (escape shallow minima)
  - Mini-batch stochasticity (natural exploration of the loss surface)
- **Practical reality:** Finding the global optimum is neither necessary nor desirable. A *good local optimum* that generalizes well is sufficient.
- **Is this still the research frontier?** No — the "benign landscape" result (over-parameterized nets: most local minima perform similarly; Choromanska et al., 2015) is now *standard*, not frontier. The live frontier: *mode connectivity* (minima joined by near-constant-loss paths — Garipov et al. 2018; Frankle et al. 2020) and *why* over-parameterization helps (*double descent* — Belkin et al. 2019, next section). The question is now *which* minima generalize, not whether good ones exist.

Regularization: Preventing Overfitting

- **L1 / L2 Regularization (weight decay):**

$$\mathcal{L}_{\text{reg}} = \mathcal{L}_{\text{data}} + \lambda \|\theta\|_p, \quad p \in \{1, 2\}$$

  - L1 ($p = 1$): promotes sparsity (like LASSO in econometrics)
  - L2 ($p = 2$): shrinks weights toward zero (like Ridge regression)

- **Dropout:** Randomly set a fraction of neurons to zero during each training step.
  - Forces the network to learn redundant representations
  - Equivalent to training an ensemble of sub-networks

- **Batch Normalization:** Normalize activations within each layer across the mini-batch. Stabilizes and accelerates training.

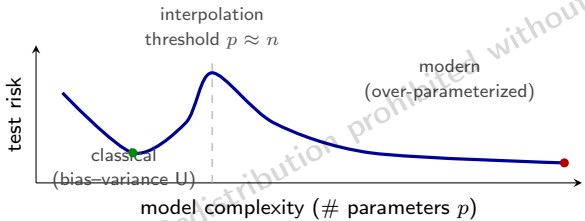
- **Data Augmentation:** Expand the training set with transformed copies (rotation, scaling, noise injection).

Modern Twist: Double Descent

Why over-parameterized networks generalize — and what that buys economic modelling

The Modern Twist: Double Descent

The "validation loss turns up ⇒ overfitting" story (two slides ago) is only the under-parameterized half of the picture.



- **Interpolation threshold.** As parameters grow, test risk follows the classical U — until $p \approx n$, where the model just fits the data exactly and risk *peaks*.
- **Second descent.** Push *past* it ($p \gg n$): (S)GD selects a **minimum-norm** interpolant (*implicit regularization*) and test risk *descends again* (Belkin, Hsu, Ma & Mandal, 2019).

The *statistical* companion to two facts we met: over-parameterized nets have a *"benign"* loss landscape (Choromanska, Henaff, Mathieu, Ben Arous & LeCun, 2015) and SGD carries *implicit regularization*. **More parameters than data can help, not hurt.**

Where Does "Implicit Regularization" Come From?

Over-parameterized ⇒ the zero-loss set is huge. With $p \gg n$, infinitely many weight vectors fit the data *exactly*; the loss cannot choose among them. **So which interpolant does training return?**

The optimizer's hidden preference — it is the ℓ_2 norm

GD from small initialization is *not* neutral — it is driven to the **minimum-norm** interpolant.

- *Least squares:* GD from the origin converges *exactly* to the minimum-$\|w\|_2$ solution — the pseudoinverse $w = \Phi^+ y$ (classical).
- *Separable classification:* GD converges in direction to the max-margin — again minimum-norm — predictor (Soudry, Hoffer, Nacson, Gunasekar & Srebro, 2018).

No penalty is ever written down — the dynamics of descent supply the bias.

- **Mechanism.** From near zero, GD only moves along directions the data excites; the "wiggly" null-space directions stay untouched, so the solution stays small, hence *smooth*. Smaller norm ⇒ smoother ⇒ lower test risk — what powers the *second* descent. *Economist's analogy:* Ridge/Tikhonov with $\lambda \to 0^+$, selected *implicitly*.

See it in Lab 5A: GD on an under-determined fit lands on the min-$\|w\|_2$ point; a large-norm interpolant hits the *same* points yet wiggles between them. *Small norm = smooth.*

Why Economists Care: The Blessings of Dimensionality

The same phenomenon, cashed out in two economic settings:

- **Econometrics / forecasting** — *Liao, Ma, Neuhierl & Shi (2023), "Does Noise Hurt Economic Forecasts?"* When the target is driven by *latent factors*, forecast models are *not* sparse: adding **many predictors — even pure noise —** *past* the sample size *lowers* out-of-sample risk (a second descent), beating PCA / LASSO-style dimension reduction (e.g. U.S. inflation). *Overfitting need not inflate out-of-sample variance.*
- **Solving dynamic equilibrium models** — *Ebrahimi Kahou & Fernández-Villaverde (2026), "The Blessings of Overparameterization: Applications in Solving Economic Models."* Deep nets that *minimize Euler / Bellman residuals* show the same double descent: **(i) no overfitting** — as width grows, *out-of-grid* residuals keep falling toward the benchmark solution; **(ii) algorithmic stability** — wide networks concentrate across random seeds where small ones do not (LQ, McCall, RBC / neoclassical growth).

Economist's take: over-parameterization is a *feature*, not a bug — it reframes the bias–variance tradeoff and turns the *curse* of dimensionality (Lec. 3 T1; Barron 1993) into a *blessing*. We exploit it next: specialized architectures (**T4**) and deep-learning solutions of macro models (**Lec. 4**, **Lec. 6**).

Bridge: A Trained Network Is Not Yet an Economic Network

Act 3.3 delivered the learning loop: forward pass → loss → backpropagated gradient → update, stabilized by regularization — plus the modern surprise that over-parameterized networks generalize (double descent).

- You can now train a generic f_θ end to end — and explain *why* it fits out of sample instead of memorizing the data.
- **Next (3.4):** generic is not enough for economics. ICNN and monotone networks hard-wire the shape restrictions theory dictates; a curve-fitting case race and the lab preview turn all of it into code you run.

*Arc: 3.1 WHY (the case for ML) → 3.2 BUILD (architecture + UAT) → **3.3 TRAIN** (backprop + SGD) → 3.4 SHAPE (economics-aware nets + lab).*