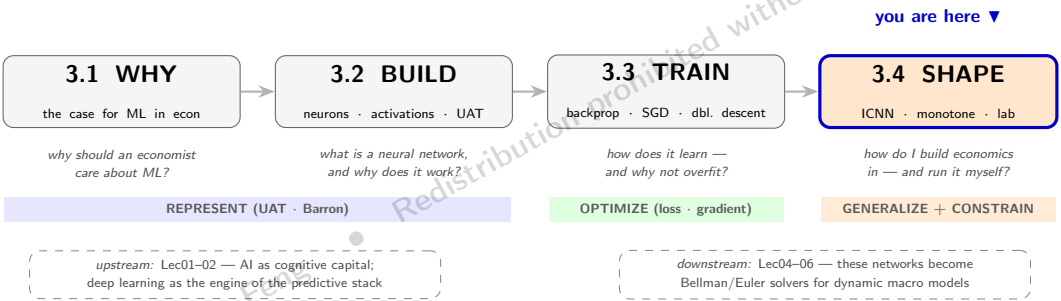
AI for Economic Research: Dynamic Models, Language, and Agents
Lecture 3.4: Specialized Architectures & Curve-Fitting Case

Zhigang Feng

2026-07-07 10:05:29

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 3.4 (here) *closes the loop: ICNN and monotone networks hard-wire economic shape restrictions, a curve-fitting race (polynomial vs. Chebyshev vs. DNN) makes it concrete, and the lab walkthrough (`Lec03_Lab_ML_Basics`) hands you the code.*

Specialized Architectures

ICNN and monotone networks: hard-wiring economic shape restrictions

Why Standard FFNNs Fall Short in Quant Macro

- **Nested "Russian Doll" structure**: Quantitative macro models involve *multiple* nested optimization problems.
- **Typical hierarchy**:
  1. **Outer Loop**: Market clearing / calibration (solve for prices p^*)
  2. **Middle Loop**: Value function iteration (solve Bellman equations)
  3. **Inner Loop**: Agent optimization (choose c, ℓ, k' given state and prices)
- **Problem**: Approximating value functions or costs with generic FFNNs turns the *inner* problem into a non-convex optimization
  - Multiple local optima and unstable outer iterations
  - Economic misspecification (e.g., upward-sloping demand, non-concave value functions)

Idea: Inject Economic Structure into the Network

Key idea: Design architectures that *respect* economic shape restrictions *by construction*.

Property	**Mathematical Gain**	**Economic Payoff**
Convexity / Concavity	Unique global optimum	Stable policy rules / VFI
Monotonicity	Invertible mappings	Well-behaved supply/demand
Homogeneity	$f(\lambda x) = \lambda f(x)$	Consistency with balanced growth

- This turns the neural net from a **black box** into a **glass box**: flexible enough to fit data, but disciplined by theory.

Convolution: Sliding a Small Filter Over the Input

Three objects (first conv layer, classifying a handwritten digit):

- **Input** X — the image: a grid of numbers (e.g. a 28×28 intensity map); in deeper layers, the previous layer's feature map.
- **Filter / kernel** W — a *small* grid of **learned** weights (e.g. 3×3): a "pattern template."
- **Output** Y — the **feature map**: one number per location, scoring how strongly W's pattern appears there.

What convolution does: slide W across X; at each spot overlay the filter, multiply overlapping cells, and add them (a local weighted dot product):

$$\underbrace{Y(i,j)}_{\text{output}} = \sum_m \sum_n \underbrace{X(i-m,\, j-n)}_{\text{input patch}} \underbrace{W(m,n)}_{\text{filter weight}}$$

$Y(i,j)$ is **large** where the local patch of X *looks like* the pattern in W (an edge, a curve, a stroke).

Weight sharing: the same tiny W is reused everywhere — it finds its feature *wherever* it occurs (translation equivariance), with far fewer parameters than a dense layer.

Pooling — and Why a CNN Is a "Learned Manifold Map"

Pooling: shrink and summarize. Replace each small window R of the feature map with one summary value; **max-pooling** keeps the strongest response:

$$\text{MaxPool}(i,j) = \max_{(m,n)\in R} X(m,n)$$

- **Downsamples** (less compute) and adds **translation invariance** — "it's a 7" wherever in the frame the 7 sits.

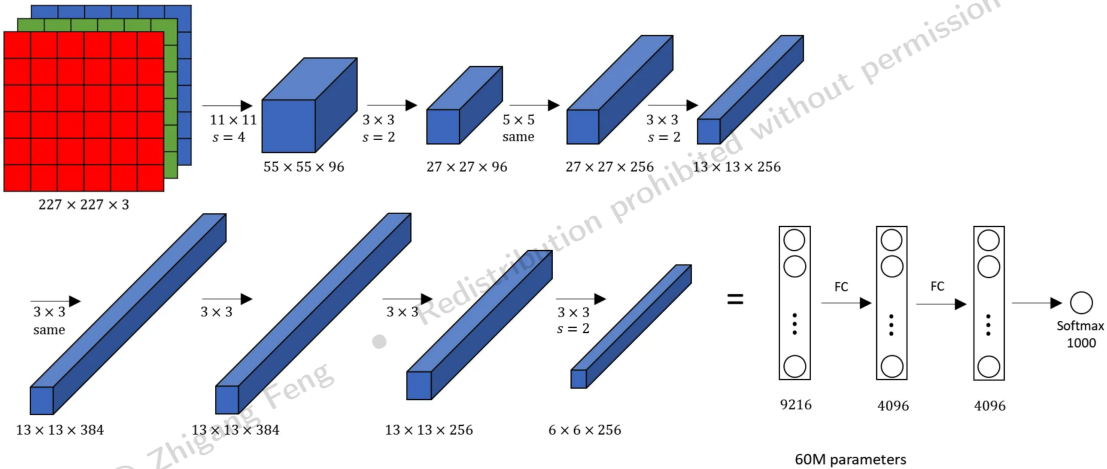
Stacking [convolution → nonlinearity → pooling] builds a hierarchy: *pixels* → *edges* → *motifs* → *objects*.

The two-stage view: the stack is a *learned* map π compressing the raw, high-dimensional input onto a compact, low-dimensional feature vector (a *manifold*); a head g_ϕ reads off the prediction:

$$\mathbf{x} \xrightarrow{\pi \text{ (conv+pool)}} \mathbf{z} \xrightarrow{g_\phi \text{ (head)}} \hat{\mathbf{p}}$$

Economic analogy: π learns a *sufficient statistic* — like reducing a high-dimensional macro state to the few moments that drive decisions (mean wealth in Krusell–Smith).

AlexNet: A Landmark CNN Architecture



Krizhevsky, Sutskever & Hinton (2012, NeurIPS) — 15.3% top-5 error vs. 26.2% for non-DL. Launched the DL agenda

Lec. 4 turns onto dynamic macro.

Input Convex Neural Networks (ICNNs)

- **Motivation:** In economics, many functions are known to be convex or concave: cost functions, duality mappings, value functions (after sign change).
- **ICNN architecture** (Amos, Xu & Kolter 2017, ICML): ensure $f(x;\theta)$ is convex in x by construction:

$$\mathbf{z}^{(0)} = x$$

$$\mathbf{z}^{(\ell+1)} = \sigma\left(\underbrace{W^{(\ell)}}_{\geq 0}\mathbf{z}^{(\ell)} + A^{(\ell)}x + b^{(\ell)}\right), \quad \ell = 1, \ldots, L-1$$

- **Key constraints:**
  - Weights $W^{(\ell)} \geq 0$ (element-wise non-negative) for $\ell \geq 1$
  - Activation σ must be convex and non-decreasing (ReLU, softplus)
  - "Pass-through" connections $A^{(\ell)}x$ can have unconstrained weights
- **Result:** The composition of convex, non-decreasing functions with non-negative weights preserves convexity.
- **For concavity:** Use $g(x) = -f(x;\theta)$ where f is an ICNN. *Lec. 4 T4 revisits with*

ICNN: Proof Sketch (Why It's Convex)

We prove by induction that each $z_l(x)$ is convex in x.

Step 1: Base Case

- $z_0(x) = x$ is linear; any linear function is both convex and concave.

Step 2: Inductive Step

$$z_{l+1}(x) = \sigma\big(W_z^{(l)} z_l(x) + W_x^{(l)} x + b_l\big)$$

- By induction, $z_l(x)$ is convex.
- With $W_z^{(l)} \geq 0$: $W_z^{(l)} z_l(x)$ is a non-negative sum of convex functions $\Rightarrow$ convex.
- Adding linear term $W_x^{(l)} x + b_l$ preserves convexity.
- Applying convex, non-decreasing $\sigma(\cdot)$ to a convex argument preserves convexity.

Therefore: All layers $z_l(x)$, and hence $f(x)$, are convex in x. *(Lec. 4 T4 uses this directly on the Bellman value function.)*

Do Constraints Limit Expressivity?

The concern: By restricting $W_z \geq 0$, do we lose the universal approximation property?

Theorem (Chen, Shi, & Zhang, 2019):

An Input Convex Neural Network can approximate any convex function over a compact domain to arbitrary accuracy.

Reference: Chen, Y., Shi, Y., & Zhang, B. (2019). *Optimal Control via Neural Networks: A Convex Approach*. International Conference on Learning Representations (ICLR).

Intuition:

- A maximum of affine functions ($\max_i\{a_i^\top x + b_i\}$) is convex and can approximate any convex shape.
- An ICNN with ReLU activations acts as a hierarchical max-affine approximator.

Economist's take: You impose a **shape restriction** (convexity) without imposing a specific parametric form (CES, Cobb-Douglas). Architecture encodes theory; data still choose the function inside the theory-consistent class. *(Lec. 4 T4 deploys this in dynamic equilibrium.)*

Partial ICNN (PICNN): The Macro Reality

The issue: Value functions $V(k, z)$ are typically:

- **Concave** in endogenous states k (capital, assets)
- **Non-concave / arbitrary** in exogenous states z (TFP, income shocks)

A standard ICNN would incorrectly enforce convexity on z as well.

Solution: Partial ICNN (PICNN, Amos et al. 2017) — two paths combined so that convexity holds in k for every fixed z:

1. **Free path for z:** $u_l = \tanh(\widetilde{W}_u^{(l)} u_{l-1} + \widetilde{b}_u^{(l)})$ (unconstrained FFNN of z).
2. **Convex path for k, gated by z:**

$$z_{l+1} = \sigma\Big(\underbrace{W_z^{(l)}\,(z_l \odot u_l)}_{W_z^{(l)} \geq 0} + W_x^{(l)}\, k + b^{(l)}\Big)$$

where $\odot$ is element-wise multiplication; u_l acts as a z-dependent positive gate on z_l.

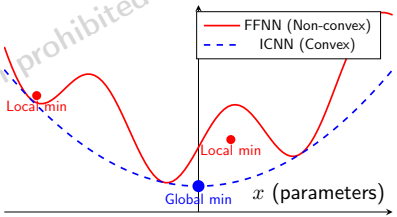
Convexity in k: at fixed z, u_l is constant. Non-negative scaling, adding linear-in-k, and applying convex non-decreasing σ all preserve convexity.

Landscape Geometry: Why FFNNs Get Stuck

Standard FFNN loss surface:

- Rugged, non-convex.
- Many local minima.
- Outcome depends on initialization.

Implication: Re-train the same model with a different seed, get a different fit. Hard to audit; hard to publish.



Why Convexity Matters for Macro Solvers

- **Bellman / Euler equations have a unique fixed point** (under standard contraction conditions). A solver that converges to a *local* minimum of the residual is not converging to that fixed point.
- **Reproducibility:** For a published paper, the policy function should not depend on the random seed. Convex training removes that dependence.
- **HA models compound the problem:** Aiyagari and Krusell–Smith require thousands of inner solves at every outer iteration. A 5% chance of a bad local min per inner solve becomes a near-certain failure mode at scale.
- **Validation cost:** Without convexity guarantees, every solution needs a hand-checked sanity audit (analytical limit, Euler residual heatmap — Lec. 4 T3). With convexity, validation is once at the architecture level.

The ICNN Advantage: Bounding Optimization Risk

- **Restated formally:** An ICNN makes $f(x;\theta)$ convex in its *input* x by construction, so *inference* (optimizing/inverting over x at fixed θ) is a convex problem with a unique global optimum. Training over the parameters θ remains non-convex (cf. T3's non-convex loss landscape); convexity is a property of the learned function, not of the loss surface. For the convex inference subproblem, gradient descent's worst case is $O(1/T)$ convergence (Nesterov 2018, *Lectures on Convex Optimization*).
- **Reference theorem:** Chen, Shi & Zhang (2019), ICLR, "Optimal Control via Neural Networks: A Convex Approach" — shows that the ICNN-parameterized optimal control problem has the same convex structure as the underlying linear-quadratic regulator.
- **Macro corollary:** You can swap a hand-coded convex solver (Sirignano & Spiliopoulos 2018 (DGM), Maliar et al. 2021 deep-learning Euler method) for an ICNN-based solver without losing convergence guarantees. The architecture provides the math; PyTorch provides the speed.
- **Cost:** Restricted expressivity (Lec. 3 T4 "Do Constraints Limit Expressivity?"). Partial ICNN (Amos et al. 2017) recovers most of it.

ICNN: Robust PyTorch Implementation

Strategy: Use reparameterization (Softplus) instead of clamping to enforce $W_z \geq 0$.

```python
class ICNN(nn.Module):
    def __init__(self, d_in, hidden_sizes):
        super().__init__()
        # Passthrough weights (Input -> Hidden) [unconstrained]
        self.W_x = nn.ParameterList([
            nn.Parameter(torch.randn(h, d_in))
            for h in hidden_sizes])
        # Recurrent weights (Hidden -> Hidden) [unconstrained params]
        # Apply Softplus() during forward pass to enforce W_z >= 0
        self.W_z_params = nn.ParameterList([
            nn.Parameter(torch.randn(h, h_prev))
            for h, h_prev in zip(hidden_sizes[1:], hidden_sizes[:-1])])
        self.biases = nn.ParameterList([
            nn.Parameter(torch.zeros(h)) for h in hidden_sizes])

    def forward(self, x):
        z = x
        for i, (Wz_p, Wx, b) in enumerate(
                zip(self.W_z_params, self.W_x, self.biases)):
            Wz = F.softplus(Wz_p)  # Enforce non-negativity
            z_next = F.linear(z, Wz) + F.linear(x, Wx) + b
            z = F.relu(z_next)
        return z
```

Monotone Neural Networks

- **Motivation:** Economic theory often implies monotonicity:
  - Demand is non-increasing in price (law of demand)
  - Consumption is non-decreasing in wealth (normal goods)
  - Value functions are non-decreasing in the state
- **Architecture for monotonicity in input x_j:**
  - Constrain all weights connecting from x_j through the network to be **non-negative**
  - Use **non-decreasing** activation functions (ReLU, sigmoid, softplus)
  - Composition of non-decreasing functions is non-decreasing
- **Partial monotonicity:** Can enforce monotonicity w.r.t. *some* inputs while leaving others unconstrained — matching economic theory that demand is monotone in price but non-monotone in income (Giffen goods aside).
- **Practical enforcement:** After each gradient step, project weights to $\mathbb{R}_+$ via $w \leftarrow \max(w, 0)$ or use $w = \exp(\tilde{w})$ reparameterization.

Monotone NN: Architecture — Partial Monotonicity

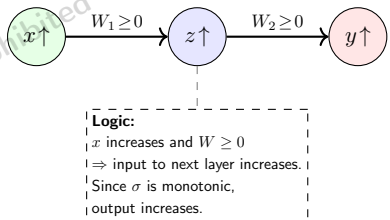
Objective: Ensure $f(x)$ is non-decreasing in selected inputs:

$$x_i \uparrow \Rightarrow \ f(x) \uparrow$$

Two necessary constraints:

1. **Non-negative weights**: $W^{(l)} \geq 0$ for monotone-input paths (biases unconstrained).
2. **Monotonic activations**: $\sigma'(\cdot) \geq 0$ (ReLU, Sigmoid, Tanh, Softplus).

Partial monotonicity: Some inputs monotone (e.g., income $\rightarrow$ consumption); others free (e.g., interest rate $\rightarrow$ savings). Mirror the partial-ICNN trick.



Monotone NN: Enforcement and Economic Examples

- **Enforcement (same two methods as ICNN):**
  - *Clamping:* $W \leftarrow \max(W, 0)$ after each gradient step. Simple but can kill neurons (dead-ReLU analogue).
  - *Reparameterization (Softplus):* train an unconstrained $\widetilde{W}$ and use $W = \mathrm{softplus}(\widetilde{W}) = \log(1 + e^{\widetilde{W}})$. Preferred — gradients flow.

- **Where macro needs monotonicity:**
  - *Wage curves* w(skill) — weakly increasing.
  - *Consumption functions* c(wealth) — non-decreasing in wealth.
  - *Demand curves* $q(p)$ — non-increasing in price (flip sign).
  - *Value functions* $V(k)$ in growth models — monotone in capital.

- **PyTorch idiom:** wrap each linear layer's weight as `F.softplus(W_raw)` in the forward call; train normally with AdamW. No special optimizer needed.

Economic Applications of Constrained Architectures

- **ICNN for optimal transport / Wasserstein distance:**
  - Kantorovich duality: $W_1(\mu, \nu) = \sup_{f \text{ 1-Lipschitz}} \mathbb{E}_\mu[f] - \mathbb{E}_\nu[f]$
  - ICNNs parameterize the convex conjugate — used in generative models and distributional analysis

- **Concave value functions:**
  - In many macro models, $V(k)$ is concave in capital
  - Approximate V with $-\text{ICNN}(k)$ to enforce concavity *by construction*
  - Eliminates spurious non-concavities from unconstrained approximation

- **Monotone policy functions:**
  - Savings policy $a'(a, z)$ is typically increasing in current assets a
  - Monotone networks guarantee this without post-hoc corrections

- **General principle:** Encoding economic structure into the network architecture reduces the hypothesis space, improves sample efficiency, and produces economically interpretable solutions.

Worked Example: Recovering Preferences via ICNN

Economic problem: Expenditure minimization

$$E(p,u) = \min_{c \geq 0} \; p \cdot c$$
$$\text{s.t.} \quad U_\theta(c) \geq u \quad \text{(target utility)}$$

The "structured" solution:

- Learn the expenditure function $E_\theta(p,u)$ using a Concave ICNN.
- **Theory**: $E(p,u)$ must be concave in prices p — a structural property of the expenditure function.

Why this is powerful:

1. **Shephard's Lemma + Auto-Diff**: $c^*(p,u) = \nabla_p E_\theta(p,u)$ — demand functions for free.
2. **Integrability**: Because E_θ is structurally concave, the resulting demands satisfy Slutsky-matrix properties by construction.

Same recipe in Lec. 4 T4 with $V(k,z)$ as the macroeconomist's first ICNN.

Choosing the Right Architecture: Cheat-Sheet

Don't use a black box when you have a blueprint.

Architecture	Shape constraint	Where useful in macro	Reuse
FFNN (baseline)	None	Generic regression, forecasting, classification	Lec 3 lab
CNN / AlexNet	Translation invariance	Satellite imagery, spatial data; not core macro	—
ICNN	Input convexity	Bellman equation surfaces, cost & value functions	Lec. 4 T4, Lab 3A/6B
PICNN	Partial convexity in k	Aiyagari, growth model with shocks z	Lec. 4 T4
Monotone NN	Partial monotonicity	Wage curves, consumption rules, demand	Lec. 4 T4
DEQ	Implicit fixed point	Equilibrium-as-layer; HA models	Lec. 6
Dropout (regularizer)	—	Overfitting control	all labs

Takeaway: Structured architectures (shaded) encode economic theory directly — you trade some flexibility for guarantees that match the problem.

Case Study: Curve Fitting and PyTorch

One function, three approximators — and your first PyTorch training loop

Case Study: DNN vs. Polynomial vs. Chebyshev

- **Objective:** Approximate a "kinked" function using three methods:
  - Deep Neural Network (DNN)
  - Traditional polynomial approximation (`np.polyfit`)
  - Chebyshev polynomial approximation

- **Why this matters for economists:**
  - Many economic models feature kinks: borrowing constraints, tax brackets, occasionally binding constraints
  - Polynomials suffer from **Gibbs phenomenon** near discontinuities
  - DNNs with ReLU activations handle kinks naturally (piecewise linear)

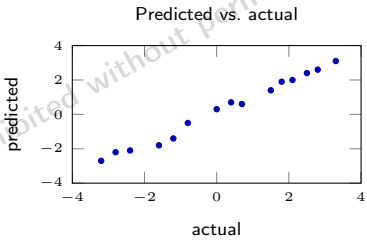
- **Test function:**

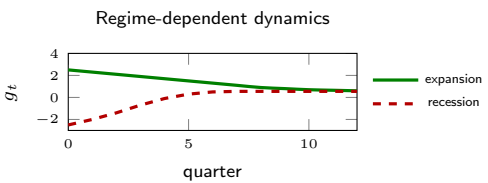
$$y(x) = \begin{cases} \alpha\beta\, x^{\alpha} & x \leq 0.5 \\ 2\alpha\beta\, x^{\alpha} & x > 0.5 \end{cases}$$

  A power function with a discrete jump at $x = 0.5$ — like a policy threshold.

Example: GDP Forecasting with a Tiny MLP

- **Objective:** Predict next-quarter real GDP growth g_t from four lags.
- **Input:** $\mathbf{x}_t = (g_{t-1}, g_{t-2}, g_{t-3}, g_{t-4})$
- **Architecture:** $4 \to 16$ (ReLU) $\to 8$ (ReLU) $\to 1$ (Linear).
- **Why a NN?** GDP dynamics are state-dependent: shocks persist differently in recessions vs. expansions; AR models miss this.
- **Loss:** MSE.





PyTorch: Defining and Training the Model

```python
import torch
import torch.nn as nn
import torch.optim as optim

class GDPNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(4, 16), nn.ReLU(),
            nn.Linear(16, 8), nn.ReLU(),
            nn.Linear(8, 1)
        )
    def forward(self, x):
        return self.net(x)

model = GDPNet()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(1000):
    optimizer.zero_grad()          # 1. Reset gradients
```

PyTorch: Class-Based Network with Batch Normalization

```python
class KinkNet(nn.Module):
    """Network for approximating kinked functions."""
    def __init__(self, input_dim, hidden_dim1,
                 hidden_dim2, output_dim=1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim1),
            nn.BatchNorm1d(hidden_dim1),
            nn.ReLU(),
            nn.Linear(hidden_dim1, hidden_dim2),
            nn.BatchNorm1d(hidden_dim2),
            nn.ReLU(),
            nn.Linear(hidden_dim2, output_dim)
        )

    def forward(self, x):
        return self.net(x)
```


Key additions: `BatchNorm1d` normalizes activations within each mini-batch — stabilizes training and acts as light regularization.

Lab Preview

Hands-on: ML basics and the FOMC-minutes yield-curve application

Lab: Machine Learning Basics and Applications

Notebook: `Lec03_Lab_ML_Basics.ipynb` — one self-contained notebook, three parts.

Part 1: Neural nets as flexible regression

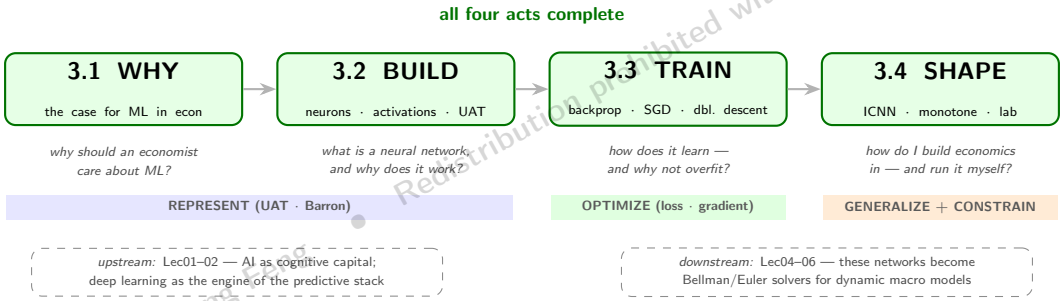
- Train a small PyTorch MLP to learn a consumption function $c = f(w)$ from noisy data — "a neural network is just non-parametric regression."
- **Universal approximation, hands-on:** build $\hat{f}(x) = a_0 + \sum_i a_i \,\mathrm{ReLU}(w_i x + b_i)$ and watch ReLU units sum to $\sin x$; then see **implicit regularization** — gradient descent finds the minimum-norm, smooth fit.

Parts 2–3: Two economic applications

- **Predict the 10-year Treasury yield** from the policy rate + unemployment (live FRED data, synthetic fallback), using a proper *time-series* split.
- **Read the Fed's words:** turn hawkish/dovish sentences into bag-of-words features, classify with logistic regression, then a PyTorch MLP.

Lecture 3, Complete: One Map, Four Acts

The toolkit is assembled: you know why ML matters for economists, what a network is, how it trains, and how to bend it to economic theory.



Every later lecture draws on this map: Lec. 4 turns these networks into Bellman/Euler solvers, Lec. 6 scales them to heterogeneous agents, and Lec. 7's transformers are the same approximation theory wearing attention.

Preview: What Lec. 4 Does With These Tools

- **Lec. 4 T1 — Motivation + optimal growth setup:** why grids fail in $d \geq 5$; AlphaGo-style intuition for macro state spaces.
- **Lec. 4 T2 — Deep VFI + actor-critic preview:** the NN architectures of T2 + T4 deployed as Bellman solvers; actor-critic algorithm previewed for Lec. 5.
- **Lec. 4 T3 — Euler equation as supervised learning:** the backprop machinery of T3 turned on first-order conditions; PyTorch implementation.
- **Lec. 4 T4 — Structured NNs + validation:** ICNN + Monotone re-applied to dynamic macro with Lab 3A (ground-truth VFI) + Lab 3B (NN solution) + Euler-residual heatmap as the validation gate.

Lec. 5 then formalizes the RL machinery; Lec. 6 scales to heterogeneous-agent models.

Summary

1. **Why ML:** Unstructured data, high-dimensional models, and non-linear dynamics make ML essential for modern macro research.

2. **Neural Networks:** Universal approximators that compose linear transformations with nonlinear activations. Barron (1993): break the curse of dimensionality.

3. **Training:** Forward pass $\rightarrow$ loss $\rightarrow$ backpropagation $\rightarrow$ gradient descent. Mini-batch SGD with Adam is the practical standard.

4. **Regularization:** Weight decay, dropout, early stopping, batch normalization prevent overfitting — the ML analogue of avoiding data mining.

5. **Constrained Architectures:** ICNNs enforce convexity; monotone networks enforce monotonicity. Encoding economic structure into network design improves both accuracy and interpretability.

6. **Next:** Lec. 4 applies these tools to *solve* macro models — deep learning meets the Bellman equation.