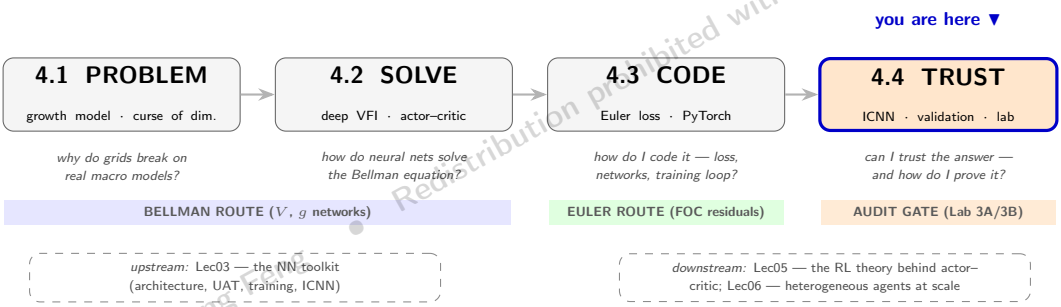
AI for Economic Research: Dynamic Models, Language, and Agents
Lecture 4.4: Structured NNs for Equilibrium & Accuracy Validation

Zhigang Feng

2026-07-07 10:05:55

Lecture 4 in One Map

One lecture, four decks, one running example: solve the optimal growth model with deep learning — pose the problem, solve it two ways, code it, and prove the answer right.



Act 4.4 (here) *earns the trust: inject concavity and monotonicity into the architecture (ICNN, monotone networks), then validate — Euler-residual diagnostics and the Lab 3A/3B audit gate.*

Structured Neural Networks for Equilibrium Models

ICNN and monotone networks: theory-consistent by construction

The Computational Challenge in Quantitative 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 with unstructured nets**: 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

Baseline: Standard Feed-Forward Networks

Generic FFNN architecture:

- Composition of affine maps and nonlinearities:

$$y = f_\theta(x) = W_L\, \sigma\big(\dots \sigma(W_1 x + b_1)\big)$$

- Layer weights $W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ are **unconstrained**

The optimization trap:

- Even if the *true* value function is concave, the FFNN approximation generally is not
- When agents optimize against this approximation:
  - FOCs are necessary but not sufficient
  - Numerical solvers can converge to spurious local optima
  - Comparative statics and equilibrium mappings can be erratic
- **Solution**: Constrain the network architecture to guarantee the desired shape

Input Convex Neural Networks (ICNN)

Amos, Xu & Kolter (2017, ICML), "Input Convex Neural Networks." *Recall:* the ICNN architecture and the convexity proof are introduced in Lec03 T4 (canonical foundations). Here we keep them only as much as needed and focus on the *macro* application — Lab 3A (classical ground truth) and Lab 3B (NN solution validated by Euler residual).

Goal: Construct $f(x)$ that is convex in x *by design*.

Recursive definition:

$$z_{l+1} = \sigma_l\left(\sum_{i=0}^{l} W_{l,i}^{(z)} z_i + W_l^{(x)} x + b_l\right)$$

Sufficient conditions for convexity:

1. **Non-negative recurrent weights**: all $W_{l,i}^{(z)} \geq 0$
2. **Convex, non-decreasing activations**: σ_l is convex and non-decreasing (e.g., ReLU, Softplus)

Result:

- $f(x)$ is convex in x as a composition of convex, non-decreasing functions

Why Impose Convexity in Macro?

Using ICNNs for value functions V or pricing kernels delivers three benefits:

1. **Robust Value Function Iteration**
   - We solve $\max_{x'}\{u(x, x') + \beta \hat{V}(x')\}$
   - If $\hat{V}$ is concave (or $-\hat{V}$ convex via ICNN), the objective is globally concave
   - Fast, gradient-based *convex* optimization with no local-maxima issues
2. **Well-Behaved Equilibrium Prices**
   - Convex preferences $\Rightarrow$ monotone demand
   - Unique, stable market-clearing price vectors
3. **Scalability to High Dimensions**
   - ICNNs scale to high-dimensional state spaces while retaining the structure needed for Bellman contractions

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, hence convex ✓

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

Do Constraints Limit Expressivity?

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

Theorem (Chen, Shi & Zhang, 2019, ICLR, "Optimal Control Via Neural Networks: A Convex Approach"):

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

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

Implication for macro:

- You are not imposing a specific parametric form (like CES or Cobb-Douglas)
- You are imposing a **shape restriction** (convexity) while retaining non-parametric flexibility

Expressivity: The Economic Interpretation

Implication for macro:

- You are not imposing a specific parametric form (like CES or Cobb-Douglas)
- You are imposing a **shape restriction** (convexity) while retaining non-parametric flexibility
- This is useful when theory tells us the global shape, but not the exact curvature

Architecture encodes theory; the data still choose the function inside that theory-consistent class.

Partial Convexity: 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.

The solution: Partial ICNN (PICNN)

Modify the architecture to distinguish between convex inputs (u) and non-convex inputs (v):

$$z_{l+1} = \sigma\left(\underbrace{W_z^{(l)} z_l}_{\geq 0} + \underbrace{W_u^{(l)}(u \circ v)}_{\text{interaction}} + \underbrace{W_v^{(l)} v}_{\text{free}} + b_l\right)$$

Implementation: Pass k through the "convex path" (constrained weights); pass z through a standard FFNN path; combine such that convexity w.r.t. k is preserved

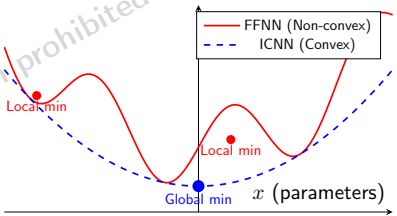
Geometry of Convex vs. Non-Convex Loss

Standard FFNN

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

ICNN

- Smooth, convex basin
- Unique global minimum
- Optimization is predictable



What ICNN Buys for Macro Solvers

Bellman equation stability and convergence.

- **Inner Bellman maximization is convex**: with a concave V (or $-$ICNN), each $\max_{k'}\{u(k,k') + \beta \hat{V}(k')\}$ is a single-basin convex program
  - Gradient / Newton methods land on the global maximum, every time
  - No bracketing, no multi-start, no policy-iteration "flips"

- **Bellman operator remains a contraction**: ICNN preserves the shape required for T to be a β-contraction on the value-function space
  - VFI converges geometrically (same theoretical rate as classical VFI)
  - Outer iterations are stable under repeated inner solves

- **Equilibrium prices are well-behaved**: convex preferences $\Rightarrow$ monotone demand $\Rightarrow$ unique market-clearing prices in calibration loops

- **Cost paid**: a per-layer non-negativity reparameterization (Softplus) — negligible at training time, large payoff in solver reliability

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
```

Monotonic NNs: Architecture — Partial Monotonicity

Objective: Ensure $f(x)$ is non-decreasing in (a designated subset of) inputs:

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

Two necessary constraints:

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

Enforcement (same two methods as ICNN):

- **Clamping**: simple, but pinned weights can kill neurons during training
- **Reparameterization (Softplus)**: preferred — smooth gradients, well-conditioned

Partial monotonicity: split inputs into a "monotone path" (constrained weights) and a "free path" (unconstrained), then combine such that monotonicity in the designated inputs is preserved. The companion frame shows the propagation logic.

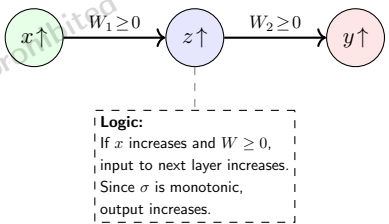
Monotonic NNs: Propagation Logic (visual)

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

- Input layer: x enters with non-negative weights $W_1 \geq 0$
- Hidden activations z inherit the increase
- Next layer: $W_2 \geq 0$ propagates the monotonicity again
- Monotonic σ never flips signs

Failure modes if any condition slips:

- Even one negative weight in the monotone path breaks the guarantee
- Non-monotone activation (e.g., GELU near zero) inverts gradients locally



Macro Application: Shape Restrictions on Policy Functions

- **Economic theory pins down the sign of many partials**:
  - **Demand** non-increasing in price (law of demand)
  - **Consumption** non-decreasing in wealth (normal goods)
  - **Value functions** non-decreasing in the resource state
  - **Savings policy** $a'(a, z)$ increasing in assets a
  - **Wage curves** non-decreasing in productivity (or non-increasing in unemployment, depending on framing)

- **Use monotone networks to bake the theory in**:
  - Savings rule monotone in a but free in $z \Rightarrow$ partial monotonicity (Aiyagari/Krusell–Smith setting, Lec06)
  - Consumption rule monotone in cash-on-hand $\Rightarrow$ avoids spurious wiggles common to FFNNs
  - Wage curve monotone in productivity $\Rightarrow$ stable labor-market mechanism

- **Bottom line**: monotone NNs guarantee economically sensible policies *without post-hoc corrections*, and the validation gate becomes much sharper (Lab 3B's residual plot stops hiding shape violations)

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**: the expenditure function $E(p, u)$ is concave in prices p (a structural property of E)

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 the Slutsky matrix properties by construction

The Shephard's Lemma example is the macroeconomist's first ICNN — same architecture as Lec03 T4, named here for the economic content (cost minimization → conditional factor demands by partial derivatives).

Other Economic Applications

- **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

- **Structured value functions for HA models**:
  - In Krusell–Smith (Lecture 6), the policy function $a'(a, z; \Gamma)$ is monotone in a and concave in cross-section
  - Enforce both via PICNN with monotonicity constraint on a

Summary: Choosing the Right Architecture

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

Economic Property	Architecture	Key Mechanism
Convexity (Utility, costs)	ICNN	Non-neg weights + skip conn.
Partial Convexity (Value functions)	Partial ICNN (PICNN)	Split paths (convex/free)
Monotonicity (Supply/demand)	Monotonic NN Deep Lattice Nets	Non-neg weights Interpolation
Equilibrium (Market clearing)	Deep Equilibrium (DEQ)	Fixed point iteration

Takeaway: Imposing structure improves interpretability, sample efficiency, and ensures your model respects economic theory

Accuracy, Validation, and Lab Preview

Euler-residual audits and the Lab 3A/3B gate: never trust an unvalidated solution

Validation Philosophy: The Research Architect Approach

- In computational economics, getting code to *run* is not enough
- The critical question: **Is the answer correct?**
- **Validation strategy** (used in Labs 6A and 6B):
  1. Solve a model with a **known analytical solution** (log utility, Cobb-Douglas)
  2. Implement the numerical method
  3. Compare numerical output against the closed-form truth
  4. Report maximum absolute error across the state space
- **Key metrics**:
  - Policy error: $\max_k |g_{\text{approx}}(k) - g^*(k)|$
  - Value error: $\max_k |V_{\text{approx}}(k) - V^*(k)|$
  - Euler equation error: $\max_k |\mathcal{R}(k; g_{\text{approx}})|$

Euler Equation Error as a Diagnostic

- Even when no analytical solution exists, we can measure accuracy via the **Euler equation error**:

$$EE(k) = \frac{u'(c)}{\beta \, \mathbb{E}[u'(c') \, f'(k')]} - 1$$

- **Interpretation**: $EE(k) = 0.01$ means the agent is making a 1% optimization error at state k
- **Standard benchmarks** (Judd 1998):
  - $\log_{10} |EE| < -3$: acceptable (error $< 0.1\%$)
  - $\log_{10} |EE| < -5$: good (error $< 0.001\%$)
  - $\log_{10} |EE| < -7$: excellent
- This diagnostic works for *any* model with Euler equations, not just those with analytical solutions

What Lab 3A Establishes: The Ground Truth

Lab 3A: Classical Benchmark (VFI)

- Solves the stochastic optimal growth model using VFI with interpolation
- Validates against the analytical solution ($c^* = (1 - \alpha\beta)y$)
- Establishes the "answer key" for Lab 3B

Key takeaways from Lab 3A:

1. The benchmark exists: we know *exactly* what the optimal policy looks like
2. VFI works, but requires a grid — breaks in high dimensions
3. The 5-step Research Architect workflow:
   - 3.1 Mathematical formulation (Bellman equation)
   - 3.2 Algorithmic specification (VFI with interpolation)
   - 3.3 Define deliverables (class structure, plots, errors)
   - 3.4 AI-augmented implementation (structured prompting)
   - 3.5 Critical validation (numerical vs. analytical)

What Lab 3B Demonstrates: Neural Network Solutions

Lab 3B: Two Deep Learning Approaches

Part 1: Deep Value Function Iteration

- Two networks (`PolicyNet` + `ValueNet`) with T-step lookahead and soft constraints
- **Pros**: provides both policy and value; handles inequality constraints naturally
- **Cons**: slower convergence; more hyperparameters
- Validated: savings rate k'/y converges to $\alpha\beta$

Part 2: Euler Equation Minimization

- Train a `PolicyNetwork` with constrained output to minimize Euler residual
- **Pros**: very precise policy; theoretically grounded
- **Cons**: no value function; requires deriving FOCs
- Validated against analytical policy $k'^{*} = \alpha\beta\, y$

Lab 3A → 3B handshake

Lab 3A produces ground truth via classical VFI on a fine grid. Lab 3B trains the NN. The Euler-residual heatmap (T3) is the validation gate: if the NN's residual is within 10^{-3} of the Lab 3A solution on a held-out test grid, it passes.

Lab Preview: What You Will Do

Lab 3A: The Classical Benchmark (VFI)

- Build an `OptimalGrowthModel` class in Python
- Implement VFI with `scipy.interpolate` and `scipy.optimize`
- Validate against the analytical solution
- **Notebook**: `Lab3A_Dynamic_Models.ipynb`

Lab 3B: Neural Network Solutions (PyTorch)

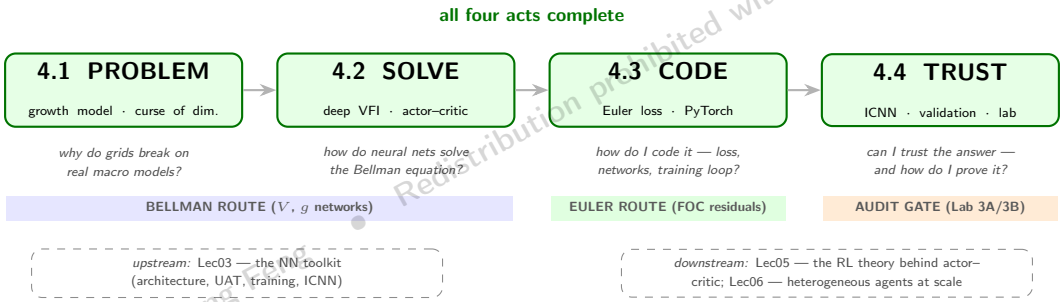
- Part 1: Train a DVFI solver with T-step lookahead and soft constraints
- Part 2: Train an Euler equation solver — compare against Lab 3A's ground truth
- Compare convergence speed, accuracy, and which method yields richer output
- **Notebook**: `Lab3B_Dynamic_Models.ipynb`

Goal: Experience the full pipeline from classical methods to deep learning, using the same model throughout for direct comparison

For a full treatment of classical VFI, interpolation, and perturbation methods, see the video series at

Lecture 4, Complete: One Map, Four Acts

You solved one model four ways — exactly, on a grid, by actor–critic, and by Euler residuals — and audited every neural answer against ground truth.



Keep the pattern: pose the model, choose the loss (Bellman or Euler), train, then validate before you trust. Lec. 5 supplies the RL theory we previewed; Lec. 6 runs this pipeline on heterogeneous-agent models.

Preview: What Lec. 5 & Lec. 6 Deliver

- **Next — Lec05 (Reinforcement Learning in a Nutshell):** the general framework behind the actor-critic we *previewed* in T2–T3.
  - **T1** — RL framework & the dynamic-programming connection (Bellman, back to Lec01).
  - **T2** — tabular methods: Monte Carlo, TD(0), Q-learning, SARSA.
  - **T3** — actor-critic as an *algorithm family* (policy-gradient theorem; A2C / PPO / DDPG / SAC) + applications.

- **Then — Lec06 (Heterogeneous-Agent Models via RL):** the *same* audit gates (Lab 3A ground truth, Lab 3B NN solution, Euler-residual heatmap) reused on much harder state spaces — Aiyagari / Krusell–Smith / DeepHAM, Deep Equilibrium Nets (DEQN), OLG, and mean-field games.

The validation discipline you learned here is the bridge: Lec05 gives you the RL theory, Lec06 puts it to work on the cross-sectional distribution.

Summary

1. **The problem**: Grid-based methods break due to the curse of dimensionality
2. **The opportunity**: Neural networks are universal approximators that scale polynomially, not exponentially, with dimension
3. **Two approaches**:
   - **Actor-Critic / DVFI**: Two networks approximate $V(k)$ and $g(k)$; RL flavor
   - **Euler Equation**: One network minimizes the Euler residual; supervised learning flavor
4. **Structured networks** (ICNN, MNN) embed economic theory into the architecture, ensuring well-behaved solutions
5. **Validation**: Always compare against known analytical solutions or Euler equation errors
6. **Next:** Lec. 5 develops the RL framework behind actor-critic in depth; Lec. 6 scales the validation discipline to heterogeneous-agent models