Quantitative Macroeconomics w/ AI and ML
Lec. 6: Solving Macroeconomic Models Using Machine Learning

Zhigang Feng

Dec., 2025

Optimal Growth Model

Classical Optimal Growth Model: Basic Setup

- **Objective:** Maximize the discounted sum of utilities:

$$\max_{\{k_{t+1}\}} \sum_{t=0}^{\infty} \beta^t u\Big(f(k_t) - k_{t+1}\Big)$$

- **Production Function:** $f(k_t) = k_t^\alpha$
- **Resource Constraint:** Consumption is given by

$$c_t = f(k_t) - k_{t+1}$$

- **Depreciation:** The model includes a depreciation parameter δ, which in our code is set to $\delta = 1$ (full depreciation) but can be adjusted for more general cases.
- **Steady State:** Capital converges to a steady state k_{ss}, computed as:

$$k_{ss} = \left(\frac{1}{\alpha\beta}\right)^{\frac{1}{\alpha-1}}$$

Note: In this model, the parameter δ is explicitly part of the configuration. Future extensions could incorporate partial depreciation ($\delta < 1$) for a more realistic setting.

Value Function Iteration (VFI) Algorithm

- **Bellman Equation:** The recursive formulation is:

$$V(k) = \max_{k'} \Big\{ u\big(f(k) - k'\big) + \beta V(k') \Big\}$$

- **Iterative Procedure:**
    - Start with an initial guess $V^0(k)$.
    - Update using the Bellman operator T as:

$$V^{n+1}(k) = TV^n(k) = \max_{k'} \Big\{ u\big(f(k) - k'\big) + \beta V^n(k') \Big\}$$

    - Repeat until $\|V^{n+1} - V^n\| <$ tolerance.
- **Contraction Mapping:** Under standard assumptions, the Bellman operator T is a contraction with modulus β on the space of bounded functions, ensuring convergence to a unique fixed point V^*.

Monte Carlo Simulation with a Constrained Policy Network

- **Policy Network:** A neural network whose output is constrained using a sigmoid layer and further scaled to lie within the feasible range.
- **Monte Carlo Approach:**
  - Randomly generate initial capital values.
  - Simulate next-period capital decisions using the policy network.
  - Compute consumption, utility, continnuation values and the Euler equation errors.
- **Loss Function:** Formulated as the value function, or the mean squared error (MSE) between the Euler equation error and zero.

Why Reframe as a Machine Learning Problem?

- The optimal growth model can be recast as a learning problem where the decision rule (policy) is approximated by a neural network.
- The first order conditions provide a natural loss function that guides the training.
- This formulation allows us to use modern optimization methods (e.g., AdamW) and automatic differentiation.
- **Note:** Detailed technical explanations and connections with reinforcement learning concepts will be explored in a future lecture.

DNN Based Value Function Iteration

Why Approximate the Value Function?

- In many dynamic equilibrium models (e.g., the optimal growth model), we need to solve

$$V(k) \;=\; \max_{g(k)} \Big\{ u\big(k, g(k)\big) \;+\; \beta\, V\big(f(k, g(k))\big) \Big\},$$

  where

  - k is the state (e.g., capital),
  - $g(k)$ is the policy (e.g., investment),
  - u is the utility function (e.g., $\log$ utility),
  - β is the discount factor.
- Traditional methods as discussed in following lectures (Chebyshev polynomials, finite elements, grid-based collocation) approximate $V(k)$ by discretizing the state space, but often suffer from the **curse of dimensionality**.
- **Deep Neural Networks (DNNs)** can mitigate this by learning high-dimensional approximations more efficiently, given suitable training data (states and associated values).

Approximating the Value and Policy Functions with DNNs

- We let

$$V(k) \approx \Gamma_{\gamma_v}(k), \quad g(k) \approx \Gamma_{\gamma_g}(k),$$

  where γ_v and γ_g are neural network parameters (weights and biases) as discussed in previous lecture.

- The idea is analogous to value function iteration:
  1. Generate states $k \sim d(k)$ in the state space (where d is some distribution or weighting measure).
  2. Compute approximate target values or gradients for training:

$$\hat{V}(k) \quad \text{or} \quad \nabla_{\gamma_g} \left\{ u(k, \Gamma_{\gamma_g}(k)) + \beta \, \Gamma_{\gamma_v}[f(k, \Gamma_{\gamma_g}(k))] \right\}.$$

  3. Train Γ_{γ_v} or Γ_{γ_g} using standard gradient-based methods to solve the relevant expectation-based objectives.

- This framework naturally handles high-dimensional k.

Embedding Equilibrium Conditions in Neural Networks

- Neural network approximations can (and often should) incorporate **equilibrium conditions** directly into their architecture or training:
  - **Feasibility constraints**: Ensure $g(k) \in \mathcal{G}(k)$, where $\mathcal{G}(k)$ defines the feasible action set (e.g., $0 \leq c \leq f(k)$).
  - **Market clearing**: In general equilibrium models, enforce conditions like $\sum_i c_i = Y$ or asset market clearing $\sum_i a_i = A$.
  - **Budget constraints**: Incorporate household or government budget balance.
- **Implementation strategies**:
  1. *Output transformations*: Use activation functions (e.g., sigmoid, softmax) to bound outputs within feasible ranges.
  2. *Penalty methods*: Add constraint violation terms to the loss function.
  3. *Lagrangian approaches*: Treat constraints via dual variables updated during training.
  4. *Architectural design*: Structure the network so outputs automatically satisfy certain identities.
- Building in economic structure improves convergence and ensures economically meaningful solutions.

Value Function Update (One-Step Look-Ahead)

- Fix a policy $g^{(n-1)}(k) = \Gamma_{\gamma_g}^{(n-1)}(k)$.
- The one-step approximation for the value is:

$$\hat{V}(k) \;=\; u\big(k, g^{(n-1)}(k)\big) \;+\; \beta\,\Gamma_{\gamma_v}^{(n-1)}\big(f(k, g^{(n-1)}(k))\big).$$

  - We are using $T_{sim} = 1$ look-ahead for simplicity: immediate payoff plus approximate next-state value.
- We then solve an *expectation-based* least squares problem:

$$\min_{\gamma_v^{(n)}} \; \mathbb{E}_{k\sim d(k)}\Big[\big(\Gamma_{\gamma_v}^{(n)}(k) - \hat{V}(k)\big)^2\Big],$$

  where $d(k)$ is the chosen state distribution (could be uniform, stationary, etc.).
- In practice, this expectation is approximated by sampling states $\{k_i\}_{i=1}^{N_v}$ from $d(k)$:

$$\min_{\gamma_v^{(n)}} \; \frac{1}{N_v}\sum_{i=1}^{N_v}\Big(\Gamma_{\gamma_v}^{(n)}(k_i) - \hat{V}(k_i)\Big)^2.$$

The Role of Value Function in Policy Evaluation

- Note that $\Gamma_{\gamma_v}^{(n-1)}(k)$ is also **given** (fixed) when computing $\hat{V}(k)$.
- **Key insight**: The value function network serves as an *evaluator* of the policy function:
  - If $\Gamma_{\gamma_v}^{(n-1)}$ is close to the true equilibrium value function V^*, and
  - If $g^{(n-1)}$ is close to the equilibrium policy g^*,
  - Then the computed value $\hat{V}(k)$ will be close to $V^*(k)$.
- **On the expectation operator**: The expectation $\mathbb{E}_{k\sim d(k)}[\cdot]$ is taken over the *state space*, not over stochastic uncertainty.
  - Even in deterministic models (no shocks), we average over sampled states.
  - **Rationale**: If a candidate policy function performs well *on average* across all sampled state values, it must be a good approximation globally.
  - Conversely, poor average performance indicates the policy is inadequate somewhere in the state space.
- This "average fitness" criterion ensures the approximation is robust across the entire domain of interest.

Policy Function Update (One-Step Look-Ahead)

- Given $\Gamma^{(n)}_{\gamma_v}$, we improve the policy by solving:

$$\max_{\gamma_g^{(n)}} \ \mathbb{E}_{k\sim d(k)}\Big[u\big(k,\Gamma^{(n)}_{\gamma_g}(k)\big) + \beta\,\Gamma^{(n)}_{\gamma_v}\big(f(k,\Gamma^{(n)}_{\gamma_g}(k))\big)\Big].$$

  - Again, $T_{sim} = 1$ for simplicity.
  - We typically solve this via gradient ascent methods applied to the neural net parameters $\gamma_g^{(n)}$.
- Approximating by sample average:

$$\max_{\gamma_g^{(n)}} \ \frac{1}{N_g}\sum_{i=1}^{N_g}\Big[u\big(k_i,\Gamma^{(n)}_{\gamma_g}(k_i)\big) + \beta\,\Gamma^{(n)}_{\gamma_v}\big(f(k_i,\Gamma^{(n)}_{\gamma_g}(k_i))\big)\Big],$$

where $\{k_i\}_{i=1}^{N_g} \sim d(k)$.

Gradient-Based Policy Improvement

- To obtain the new policy $\gamma_g^{(n)}$, we use **gradient ascent**:

$$\gamma_g^{(n)} = \gamma_g^{(n-1)} + \alpha \cdot \nabla_{\gamma_g} J(\gamma_g)\Big|_{\gamma_g = \gamma_g^{(n-1)}},$$

  where $J(\gamma_g)$ is the objective function and $\alpha > 0$ is the learning rate (step size).

- **Two roles of the gradient**:
  1. **Direction**: $\nabla_{\gamma_g} J$ tells us *which direction* in parameter space improves the objective.
  2. **Magnitude of improvement**: Combined with the learning rate α, it determines *how far* we move from $\gamma_g^{(n-1)}$ toward $\gamma_g^{(n)}$.

- The update can be written explicitly as:

$$\gamma_g^{(n)} = \gamma_g^{(n-1)} + \alpha \cdot \frac{1}{N_g} \sum_{i=1}^{N_g} \nabla_{\gamma_g} \Big[u\big(k_i, \Gamma_{\gamma_g}(k_i)\big) + \beta\, \Gamma_{\gamma_v}^{(n)}\big(f(k_i, \Gamma_{\gamma_g}(k_i))\big) \Big] \Bigg|_{\gamma_g^{(n-1)}}.$$

- In practice, optimizers like Adam or SGD with momentum adaptively adjust step sizes for faster and more stable convergence.

Choosing the Distribution $d(k)$ and Weights

- In the above, $\mathbb{E}_{k\sim d(k)}$ indicates an expectation with respect to the distribution or weighting measure $d(k)$.
- Common choices:
  - **Uniform distribution** on a bounded interval (simple, but may be inefficient for large state spaces).
  - **Stationary distribution** induced by an approximate policy (focuses on states the system is likely to visit).
  - **Adaptive or importance sampling** strategies to focus on high-value or high-uncertainty regions.
- In practice, we often approximate the integral by a sample average:

$$\mathbb{E}_{k\sim d(k)}[F(k)] \approx \sum_{i=1}^{N} w_i F(k_i), \quad \text{where} \sum_i w_i = 1.$$

- Weights w_i may be uniform ($w_i = 1/N$) or adjusted to account for different sampling schemes.

Bias-Variance Tradeoff for $T_{sim} = 1$

- $\mathbf{T_{sim} = 1}$ approach:

$$\hat{V}(k_i) \;\approx\; u\big(k_i, g^{(n-1)}(k_i)\big) + \beta\, \Gamma^{(n-1)}_{\gamma_v}\big(f(k_i, g^{(n-1)}(k_i))\big).$$

- **Pros:**
    - Easy to implement and fast to sample/compute.
    - Lower *variance* in target values $\hat{V}(k_i)$.
- **Cons:**
    - Higher *bias*: we rely heavily on the *previous* value function approximation for future returns.
    - May converge more slowly or to a suboptimal policy if the approximation is poor.
- A natural extension is to use $\mathbf{T_{sim}}$-step or multi-period look-ahead to reduce bias. This will, however, increase variance in the training targets.

Improvement 1: Focusing on the Stationary Distribution (1/2)

- **Motivation:** Uniformly sampling $\{k_i\}$ over a large domain can be highly inefficient:
  - Many sampled states may be very unlikely to occur in practice.
  - This wastes model capacity on unimportant regions of the state space.
- **Stationary Distribution Approach:**
  - *Simulate* the economy under the current or near-optimal policy for many periods.
  - Wait until the capital stock (or other state variables) converges to a *stationary distribution*.
  - *Draw training points* $\{k_i\}$ from that distribution.
- **Benefits**:
  - *Sample efficiency:* We focus on the part of the state space that truly matters for long-run outcomes.
  - Often converges faster because the DNN learns to approximate $V(\cdot)$ and $g(\cdot)$ well in the most relevant regions.
  - *Law of Large Numbers:* By simulating long enough, the empirical distribution of states stabilizes, giving reliable samples without needing to handcraft a grid.

Improvement 1: Focusing on the Stationary Distribution (2/2)

- **Why not Chebyshev or other grid-based methods here?**
  - Chebyshev and finite-element methods typically *must* approximate over the entire domain.
  - As dimensionality grows, covering the entire domain with a fine grid becomes infeasible (*curse of dimensionality*).
  - In contrast, stationarity-based sampling zooms in on the *likely* region of interest.
- **Limitations:**
  - *Rare but important events* (e.g., extreme shocks or tail risks) may be under-represented or entirely missed if they have low probability but high impact.
  - If the system has multiple steady states or complex dynamics, a single stationary distribution might not capture important transient states.
- **Possible Remedies:**
  - Combine stationary distribution sampling with targeted sampling of rare states or shocks (sometimes called *importance sampling*).
  - Conduct stress tests or out-of-distribution sampling separately to ensure the DNN's performance does not degrade drastically in rare scenarios.

Improvement 2: Extending to T_{sim}-Step Returns (1/3)

- **Recall One-Step Return:**

$$\hat{V}(k_i) \;=\; u\big(k_i, g^{(n-1)}(k_i)\big) \;+\; \beta\, \Gamma_{\gamma_v}^{(n-1)}\big(f(k_i, g^{(n-1)}(k_i))\big).$$

$\Rightarrow$ *Fast* but *biased* if $V(\cdot)$ is inaccurate.

- T_{sim}**-Step Return Idea:**

$$\hat{V}(k_i) \;=\; \sum_{t=0}^{T_{sim}-1} \beta^t\, u\big(k_{i,t}, g^{(n-1)}(k_{i,t})\big) \;+\; \beta^{T_{sim}}\, \Gamma_{\gamma_v}^{(n-1)}\big(k_{i,n}\big),$$

where $k_{i,t+1} = f\big(k_{i,t}, g^{(n-1)}(k_{i,t})\big)$.

- **Pros & Cons**:
  - $+$ *Lower bias*: uses actual simulated rewards over multiple steps.
  - $-$ *Higher variance*: the simulated path can vary significantly, especially under uncertainty.
- As $n \to \infty$, we approximate the infinite horizon directly. But in practice, large n can be costly and noisy.

Improvement 2: Extending to T_{sim}-Step Returns (2/3)

- **Policy Function Update with T_{sim}-Step:**
  - Similarly, when *improving* the policy, we could maximize the n-step return plus the value function at the end of the n-step path:

$$\max_{\gamma_g^{(n)}} \sum_{i=1}^{N_g} \left[\sum_{t=0}^{T_{sim}-1} \beta^t u\big(k_{i,t}, \Gamma_{\gamma_g}^{(n)}(k_{i,t})\big) \; + \; \beta^{T_{sim}} \Gamma_{\gamma_v}^{(n)}(k_{i,n}) \right].$$

  - This can potentially improve policy more significantly each iteration but also introduces more variability in gradients.

Improvement 2: Extending to T_{sim}-Step Returns (3/3)

- **Uncertainty and the Role of Monte Carlo:**
  - When the model includes stochastic elements or shocks, the future state evolves randomly.
  - Enumerating all possible future states or integrating over a high-dimensional shock space quickly becomes infeasible (another form of the curse of dimensionality).
  - *Monte Carlo simulation* circumvents this by generating a large sample of *realized* state-and-shock paths, and computing empirical averages to approximate expectations.
  - By drawing sufficiently many simulated paths $(k_{i,t}, \omega_{i,t}, \ldots)$, we can estimate the expected returns or value function without explicitly dealing with the full high-dimensional distribution.
  - DNNs then approximate the underlying functions (value or policy) from these simulated data points, relying on the law of large numbers for consistency.
- **In Practice**:
  - Choose T_{sim} to balance bias vs. variance.
  - Use parallel or GPU-based Monte Carlo to handle large simulation demands.
  - Combine with importance sampling if rare but influential shocks are of interest.

Algorithm Summary

1. **Initialize:**
$$\Gamma^{(0)}_{\gamma_g},\ \Gamma^{(0)}_{\gamma_v}.$$
2. **Policy Evaluation (Value Function Update):**
   - Generate training data $\{k_i, \hat{V}(k_i)\}$ using either:
     - **One-step** approach: $T_{sim} = 1$,
     - or a $\mathbf{T}_{\text{sim}}$-period Monte Carlo simulation for lower bias.
   - Solve
$$\min_{\gamma_v^{(n)}} \sum_i \left[\Gamma^{(n)}_{\gamma_v}(k_i) - \hat{V}(k_i)\right]^2.$$
3. **Policy Improvement (Policy Function Update):**
   - Generate states $\{k_i\}$ (e.g., from the stationary distribution).
   - Solve
$$\max_{\gamma_g^{(n)}} \sum_i \left[u(k_i, \Gamma^{(n)}_{\gamma_g}(k_i)) + \beta\, \Gamma^{(n)}_{\gamma_v}(f(k_i, \Gamma^{(n)}_{\gamma_g}(k_i)))\right].$$
4. **Check convergence** and repeat until Γ_{γ_g} and Γ_{γ_v} stabilize.

Euler Equation Based + ML

Context: Standard Solution Methods

- Before introducing the neural network approach, recall how **standard methods** solve dynamic models:
- **Perturbation methods**:
  - Linearize (or higher-order expand) around the *steady state*.
  - Fast and accurate locally, but may lose accuracy far from steady state.
  - Will be discussed in detail in upcoming lectures.
- **Projection / value function iteration methods**:
  - Construct a sequence of policy functions $\{g^{(n)}\}$ that converges *monotonically* to the equilibrium policy g^*.
  - Relies on contraction mapping properties of the Bellman operator.
  - Globally accurate but computationally intensive in high dimensions.
- **Neural network methods** combine ideas from both: they approximate policy functions globally (like projection) using flexible function approximators trained via gradient-based optimization.

Euler Equation: The Optimality Condition

- Capital accumulation: $k' = g(k)$, consumption: $c = f(k) + (1-\delta)k - g(k)$.
- For CRRA utility $u(c) = \frac{c^{1-\sigma}}{1-\sigma}$ and production $f(k) = k^{\alpha} + (1-\delta)k$, the equilibrium satisfies

$$u'(c) \;=\; \beta\, u'\big(c'\big)\, \Big[\alpha\, g(k)^{\,\alpha-1} + 1 - \delta\Big]. \tag{EE}$$

- The Euler equation characterizes *intertemporal optimality*: the marginal cost of saving today equals the discounted marginal benefit tomorrow.
- **Goal**: Learn a policy g that makes the Euler residual

$$\mathcal{R}(k\,;\,g) \;=\; u'(c) - \beta\, u'(c')\big[\alpha g(k)^{\alpha-1} + 1 - \delta\big]$$

vanish for all economically relevant k.

Euler Equation: Necessary but Not Sufficient

- **Important caveat**: The Euler equation is a *necessary* condition for optimality, but **not sufficient** by itself.
- For a complete characterization, we also need:
    - **Transversality condition**: $\lim_{t\to\infty} \beta^t u'(c_t)k_{t+1} = 0$.
    - **Boundary / feasibility constraints**: $c \geq 0$, $k' \geq 0$, etc.
- **Handling inequality constraints**:
    - Kuhn-Tucker complementary slackness conditions (e.g., $\mu \geq 0$, $g \geq 0$, $\mu \cdot g = 0$) arise with inequality constraints.
    - These can often be *converted to equalities* using techniques like Fischer-Burmeister functions or penalty methods.
    - We will discuss these transformations in a future lecture.
- For now, we focus on interior solutions where Euler equation holds with equality throughout.

From Euler Residual to Loss Function

- Approximate the policy by a neural net $g(k) \;\approx\; \Gamma_{\gamma_g}(k)$.
- Sample $k_i \sim d(k)$ (e.g. the stationary distribution used in the earlier value-iteration slides).
- Define the **mean-squared Euler residual** loss:

$$\mathcal{L}_g(\gamma_g) \;=\; \frac{1}{N}\sum_{i=1}^{N}\bigl[\mathcal{R}\bigl(k_i\,;\,\Gamma_{\gamma_g}\bigr)\bigr]^2. \qquad \text{(ML objective)}$$

- Optimal policy parameters satisfy $\nabla_{\gamma_g}\,\mathcal{L}_g(\gamma_g^\star) = 0$.

The Mean Value Intuition

- **Key insight**: We minimize the *average* Euler residual across sampled states:

$$\mathcal{L}_g(\gamma_g) = \mathbb{E}_{k\sim d(k)}\left[\mathcal{R}(k;\Gamma_{\gamma_g})^2\right] \approx \frac{1}{N}\sum_{i=1}^{N}\mathcal{R}(k_i;\Gamma_{\gamma_g})^2.$$

- **Intuition**:
  - If a candidate policy function satisfies the Euler equation *on average* across all sampled state values, it should be a good approximation to the true equilibrium policy.
  - Conversely, if the average Euler residual is large, the policy must be violating optimality conditions somewhere in the state space.
- This "average fitness" criterion:
  - Ensures global accuracy across the domain of interest.
  - Is computationally tractable via Monte Carlo sampling.
  - Naturally integrates with gradient-based neural network training.
- The distribution $d(k)$ can be chosen to emphasize economically relevant regions (e.g., near steady state, or the ergodic distribution).

Neural Network for the Policy Function

- Two-hidden-layer MLP (64–64–1) with ReLU non-linearities:

$$\boxed{\Gamma_{\gamma_g}(k) = \sigma_2\Big(W_2 \ \cdot \sigma_1\big(W_1\, k + b_1\big) + b_2\Big),}$$

- Output activation $\sigma_1 = \texttt{ReLU}$, and $\sigma_2 = \texttt{Sigmoid}$; we scale it to $[0, f(k) + (1-\delta)k - \varepsilon]$ so that $0 \leq \Gamma_{\gamma_g}(k) \leq f(k) + (1-\delta)k$.
- All derivatives $\partial \mathcal{L}_g / \partial \gamma_g$ needed in the Euler residual are provided automatically by `autograd`.

Minimising the Euler Residual Loss

- Initialise $\gamma_g^{(0)}$; choose learning rate α_g.
- **Repeat until convergence**
  1. Draw a mini-batch $\{k_i\}_{i=1}^{B}$.
  2. Compute loss $\mathcal{L}_g(\gamma_g)$ and its gradient via back-propagation.
  3. Gradient step
  $$\gamma_g^{(j+1)} = \gamma_g^{(j)} - \alpha_g \nabla_{\gamma_g} \mathcal{L}_g\big(\gamma_g^{(j)}\big).$$
  4. Optionally resample k_i from the *same* distribution $d(k)$ used earlier in value-iteration slides (uniform grid, simulated stationary, or importance-sampling mix).

Pytorch Implementation

VFI + DNN + RL

- **Goal**: Solve optimal-growth model
    - `ValueNet` $\widehat{V}(k \,|\, \gamma_v)$
    - `PolicyNet` $\widehat{g}(k \,|\, \gamma_g)$
- **Key steps**:
    1. Network architecture
    2. Bellman MSE evaluation
    3. Policy gradient descent
    4. Training loop
- Single Python file + JSON config

1–Network Architecture

```python
class ValueNet(nn.Module):
    def __init__(self, nh=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, nh), nn.ReLU(),
            nn.Linear(nh, nh), nn.ReLU(),
            nn.Linear(nh, 1))
    def forward(self, k): return self.net(k)

class PolicyNet(nn.Module):
    def __init__(self, alpha, eps, nh=64):
        super().__init__()
        self.alpha, self.eps = alpha, eps
        self.net = nn.Sequential(
            nn.Linear(1, nh), nn.ReLU(),
            nn.Linear(nh, nh), nn.ReLU(),
            nn.Linear(nh, 1), nn.Sigmoid())
    def forward(self, k):
        max_kp = torch.clamp(k**self.alpha - self.eps, 1e-9)
        return self.net(k) * max_kp
```

- `Sigmoid` ensures $g(k) \in [0, f(k) - \varepsilon]$

2–Policy Evaluation (Bellman MSE)

```python
def value_update_step(self):

        self.value_optimizer.zero_grad()
        # Sample batch of capital states
        k_batch = torch.rand(self.n_batch, 1, device=self.device)*(self.k_max-self.k_min) + self.k_min

        # Current estimate: V(k)
        v_current = self.value_net(k_batch)

        # Next capital according to current policy
        kp_batch = self.policy_net(k_batch)

        # Bellman RHS: u + beta*V(k')
        u_batch = self.utility(k_batch, kp_batch)
        v_next = self.value_net(kp_batch).detach()  # detach so we don't backprop through V(k') here
        bellman_rhs = u_batch + self.beta * v_next

        # MSE loss
        loss = nn.functional.mse_loss(v_current, bellman_rhs)
        loss.backward()
        self.value_optimizer.step()

        return loss.item()
```

- $\min_{\gamma_v} \mathbb{E}\big[(V - (u + \beta V'))^2\big]$

3–Policy Improvement (Gradient Descent)

```python
def policy_step(self):
        self.policy_optimizer.zero_grad()
         # Sample batch of capital states
         k_batch = torch.rand(self.n_batch , 1, device=self.device)*(self.k_max-self.k_min) + self.k_min

         # Proposed next capital
         kp_batch = self.policy_net(k_batch)
         # Evaluate the objective
         u_batch = self.utility(k_batch , kp_batch)
         v_next = self.value_net(kp_batch)

         # Mean objective
         objective = torch.mean(u_batch + self.beta * v_next)

         # We want to maximize -> so we minimize -objective
         loss = -objective
         loss.backward()
         self.policy_optimizer.step()

         # Return the positive objective for logging
         return objective.item()
```

- $\max_{\gamma_g} \mathbb{E}\left[u(k,\widehat{g}(k)) + \beta\widehat{V}(\widehat{g}(k))\right]$

4–Training Loop

```python
def train(self):
        print(f"Training started on device={self.device}")
        pbar = tqdm(range(self.n_epoch), desc="Outer Loop")

        for _ in pbar:
            # Value net update
            for _ in range(self.n_inner_value):
                v_loss = self.value_update_step()
            self.value_losses.append(v_loss)

            # Policy net update
            for _ in range(self.n_inner_policy):
                policy_obj = self.policy_update_step()
            self.policy_objectives.append(policy_obj)

            pbar.set_postfix({
                "ValueLoss": f"{v_loss:.4e}",
                "PolicyObj": f"{policy_obj:.4e}"
            })
```

5–Visualization

```python
def plot_results(self):
        with torch.no_grad():
            # Evaluate the value function
            V_eval = self.value_net(self.k_grid_eval).cpu().numpy()
            # Evaluate the policy function
            g_eval = self.policy_net(self.k_grid_eval).cpu().numpy()

        # Plot Value Function
        plt.figure(figsize=(12,5))
        plt.subplot(1,2,1)
        plt.plot(self.k_grid_eval.cpu().numpy(), V_eval, label="V(k)")
        plt.xlabel("Capital, k")
        plt.ylabel("Value Function")
        plt.grid(True, alpha=0.3)
        plt.legend()

        # Plot Policy Function
        plt.subplot(1,2,2)
        plt.plot(self.k_grid_eval.cpu().numpy(), g_eval, label="g(k)", color="green")
        plt.xlabel("Capital, k")
        plt.ylabel("Policy Function k'")
        plt.grid(True, alpha=0.3)
        plt.legend()

        plt.tight_layout()
        plt.show()
```

The Role & Advantage of Reinforcement Learning

From Standard DP to Deep RL: Why the Shift?

- **Standard dynamic programming** relies on:
  - Contraction mapping theorems guaranteeing convergence.
  - Structured grids (uniform, Chebyshev nodes, Smolyak sparse grids).
- **These methods work well** for low-dimensional, smooth problems.
- **But they struggle when**:
  - **Curse of Dimensionality:** Cost scales exponentially with state dimension (N^d).
  - **Kinks & Non-convexities:** FOCs often fail or require complex root-finding.
  - **Unknown Transitions:** When the law of motion (e.g., distribution Γ') is hard to compute explicitly.
- **Deep RL offers a different paradigm**: Learn from sampled experience (simulation) rather than exhaustive grid evaluation.

The Inspiration: AlphaGo

- **AlphaGo** (DeepMind, 2016) solved a problem with $\sim 10^{170}$ states.
- **How?** It did not solve the full Bellman equation on a grid.
- **It used two networks:**
    - **Policy Net (Actor):** ”Intuition” regarding which move to play.
    - **Value Net (Critic):** ”Judgment” regarding who is winning.
- **Lesson for Economics:** When the state space is too vast to enumerate (like the distribution of wealth in a HA model), we can use the same two-network approach to find the solution.

The Architecture: Actor-Critic

- **Why not just classical Value Function Iteration?**
  - In Deep RL, the value function is a neural network (non-convex).
  - Finding $a^* = \arg\max_a Q(s, a)$ requires a slow numerical optimization *inside* every simulation step.
- **The Solution: Two Networks**
  1. **The Actor (π):** "The Doer."
     - Inputs state $s \rightarrow$ Outputs action a directly.
     - Avoids the expensive max-step.
  2. **The Critic (V):** "The Judge."
     - Inputs state $s \rightarrow$ Estimates value $V(s)$.
     - Minimizes the **Bellman Residual** (The prediction error).

Mapping Macroeconomics to Reinforcement Learning

To apply RL, we translate economic models into the standard tuple (S, A, P, R, γ).

RL Concept	**Economic Equivalent**	**Note**
Agent	Household / Firm	The optimizer.
Environment	Constraints & Prices	The rules of the game.
State (S)	(k, z, Δ)	Wealth (k), Shocks (z), **Distribution (Δ)**.
Action (A)	c, i, l	Consumption, Investment.
Transition (P)	$k' = (1-\delta)k + i$	**Micro:** Known perfectly.
	$\Delta' = H(\Delta)$	**Macro:** Often unknown/intractable.

The Synergy: RL allows us to handle the complexity of the macro transition Δ' by simply simulating it, without needing to derive the law of motion explicitly.

Advantage I: Flexible Function Approximation

- **Old Way: Structured Grids**
    - Requires tensor product grids or sparse grids (Smolyak).
    - *Curse of Dimensionality:* Adding one state variable multiplies the grid size.
- **New Way: Neural Networks**
    - Neural networks are **Universal Approximators**.
    - Parameter count grows linearly (or polynomially) with dimension, not exponentially.
    - **Result:** We can handle high-dimensional states (e.g., the entire history of shocks, or a fine-grained distribution) that breaks grid-based methods.

Advantage II: The ”Mesh-Free” Synergy

Why is the combination of **Monte Carlo (MC)** and **Deep Learning (DL)** so powerful?

- **1. The Data Structure (Monte Carlo):**
    - MC simulation produces a ”cloud” of points concentrated in the **Ergodic Set** (the economically relevant states).
    - Traditional interpolators (Chebyshev, Splines) fail on such irregular, clustered data (Runge's phenomenon).
- **2. The Approximator (Deep Learning):**
    - Neural Networks are **”Mesh-Free”**.
    - They do not require a grid. They simply minimize error on *whatever* points you give them.
- **Conclusion:** MC generates the *right data* (relevant states), and DL is the *right tool* to fit that irregular data.

Addressing the Infinite Horizon Paradox

The Puzzle:

- Macro models maximize over an **infinite horizon**: $\sum_{t=0}^{\infty} \beta^t u_t$.
- Monte Carlo simulations are always **finite**.

The Solution: Bootstrapping

- We do not need to simulate to infinity. We rely on the recursive Bellman structure.
- We use the **Estimated Value Function** (Critic) as a proxy for the infinite future.

$$\text{Total Value} \approx \underbrace{u_t}_{\text{Current Data}} + \beta \underbrace{\hat{V}(s_{t+1})}_{\text{Learned Proxy for } \infty}$$

- This allows us to learn infinite-horizon policies using only short-term simulation steps (transitions).

Advantage III: Hardware Acceleration

- **Parallelism:**
  - We can simulate thousands of economic agents in parallel on a GPU.
  - Neural network training (matrix multiplication) is highly optimized for GPUs/TPUs.
- **Comparison:**
  - Traditional VFI is often serial or hard to parallelize efficiently.
  - Deep RL allows us to scale up the model size (more agents, more shocks) simply by adding more compute power.

Structured Neural Networks for Equilibrium Models

The Computational Challenge in Quantitative Macro

- **Nested "Russian Doll" Structure:** Quantitative macro models typically involve *multiple* nested optimization problems.
- **Typical Hierarchy:**
  1. **Outer Loop:** Market clearing / calibration (solve for prices p^* or parameters θ).
  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 key objects (value functions, costs, demands) with generic FFNNs turns the *inner* problem into a non-convex optimization.
  - **Consequences:**
    - 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 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* economic object (e.g., a cost or value function) is convex/concave, the FFNN approximation generally is not.
- When agents optimize against this approximation:
    - First-order conditions are *necessary* but not *sufficient* for optimality.
    - Numerical solvers can converge to spurious local optima.
    - Comparative statics and equilibrium mappings can be erratic.

Input Convex Neural Networks (ICNN)

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, leaky ReLU, Softplus).

Result:

- $f(x)$ is convex in x as a composition of convex, non-decreasing functions.
- **Crucial:** Direct input weights $W_l^{(x)}$ can be unconstrained (including negative entries). This allows the network to learn the *linear* placement and slope of the convex function freely.

Why Impose Convexity in Macro?

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

1. **Robust Value Function Iteration (VFI)**
   - 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.
   - **Benefit:** Fast, gradient-based *convex* optimization with no local-maxima issues.
2. **Well-Behaved Equilibrium Prices**
   - General equilibrium requires inverting demand to back out prices.
   - Convex preferences $\Rightarrow$ monotone demand; ICNNs can preserve such monotonicity.
   - **Benefit:** Unique, stable market-clearing price vectors.
3. **Scalability to High Dimensions**
   - Grids suffer from the curse of dimensionality.
   - ICNNs scale to high-dimensional state spaces while retaining the structure needed for Bellman contractions.

Do Constraints Limit Expressivity?

The Concern: By restricting $W_z \geq 0$, do we lose the "universal approximation" property of standard neural networks?

Theorem (Chen, Man & Amos, 2019):

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

Intuition:

- A maximum of affine functions $(\max_i\{a_i^T x + b_i\})$ 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.

ICNN Example & Implementation

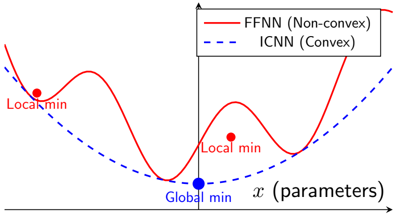
Optimization Landscapes: FFNN vs. ICNN

Standard FFNN

- **Shape:** Rugged, highly non-convex landscape.
- **Issue:** Many local minima; outcome depends on initialization.

ICNN

- **Shape:** Smooth, convex basin.
- **Implication:** A unique global minimum; optimization is predictable.

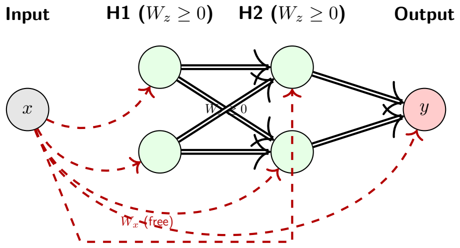


ICNN: Architectural Changes

Key differences vs. FFNN:

1. **Skip connections from x:** The input x is fed into *every* hidden layer.
2. **Constrained recurrent weights:** Hidden-to-hidden weights W_z are restricted to $W_z \geq 0$.

$$z_{l+1} = \sigma\Big(\underbrace{W_z z_l}_{\geq 0} + \underbrace{W_x x}_{\text{free}} + b\Big).$$



Red dashed

arrows: skip connections from input x.

Why the ICNN is Convex: Proof Sketch

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 the linear term $W_x^{(l)} x + b_l$ preserves convexity.
- Applying a convex, non-decreasing activation $\sigma(\cdot)$ to a convex argument preserves convexity.

Therefore: All layers $z_l(x)$, and hence $f(x)$, are convex in x.

The Macro Reality: Partial Convexity

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).

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

Key Implementation Details:

1. Pass k through the "convex path" (constrained weights).
2. Pass z through a separate standard FFNN path.
3. Combine them such that convexity w.r.t k is preserved (e.g., via non-negative mixing weights).

Robust PyTorch Implementation

Strategy: Use reparameterization (Softplus) instead of Clamping to enforce $W_z \geq 0$. This improves gradient stability.

```python
class ICNN(nn.Module):
    def __init__(self, d_in, hidden_sizes):
        super().__init__()
        # 1. Passthrough weights (Input -> Hidden) [unconstrained]
        self.W_x = nn.ParameterList([
            nn.Parameter(torch.randn(h, d_in))
            for h in hidden_sizes
        ])

        # 2. Recurrent weights (Hidden -> Hidden) [unconstrained params]
        # We will apply Softplus() to these during forward pass
        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
        ])

```

Monotonic Neural Networks

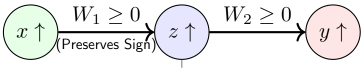
Monotonic Neural Networks (MNNs)

Objective: Ensure output $f(x)$ is non-decreasing w.r.t input x.

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

Two Necessary Constraints:

1. **Non-Negative Weights:**
    - $W^{(l)} \geq 0$ for all layers.
    - Biases b are **unconstrained**.
2. **Monotonic Activations:**
    - $\sigma(\cdot)$ must be non-decreasing ($\sigma' \geq 0$).
    - Valid: ReLU, Sigmoid, Tanh, Softplus.



Logic:
If x increases ($+$), and $W \geq 0$,
then input to next layer increases.
Since σ is monotonic, output increases.

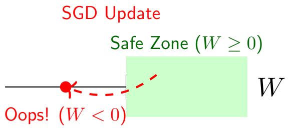
The Optimization Conflict

The Conflict:

- **Theory requires:** $W \geq 0$ (to ensure Monotonicity).
- **Optimizer wants:** To minimize Loss L, regardless of constraints.

What happens without intervention?

- Gradient descent might push a weight from 0.1 down to -0.5.
- **Result:** Monotonicity is broken.



Enforcing Constraints: Two Methods

How do we ensure $W \geq 0$ during training?

1. Clamping (Projected Gradient)

- **Step:** After every optimizer update, manually set negative weights to zero.
- `W.data.clamp_(min=0)`
- **Pros:** Simple.
- **Cons:** Can "kill" neurons permanently if gradients are consistently negative.

2. Reparameterization (Preferred)

- **Step:** Learn an unconstrained parameter v, use $W = \text{Softplus}(v)$ in the model.
- $W = \ln(1 + e^v)$
- **Pros:** W is always strictly positive; gradients flow smoothly even if v is negative.
- **Cons:** Slightly more compute.

Applications & Summary

Worked Example: Recovering Preferences

Economic Problem: Expenditure Minimization

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

Challenge: We don't know $U_\theta(c)$. We observe prices p and demands c.

Worked Example: Recovering Preferences

The "Structured" Solution (ICNN): Instead of learning U_θ directly, we learn the **Expenditure Function** $E_\theta(p, u)$ using an ICNN.

- **Theory:** $E(p, u)$ must be **concave** in prices p (Shephard's Lemma).
- **Implementation:** Train a "Concave ICNN" (negative weights or $-ICNN$) to approximate $E(p, u)$.

Why this is powerful:

1. **Shephard's Lemma (Auto-Diff):**

$$c^*(p, u) = \nabla_p E_\theta(p, u)$$

   We get demand functions c^* for free by differentiating the network!

2. **Integrability:** Because E_θ is structurally concave, the resulting demands c^* are guaranteed to satisfy the Slutsky matrix properties.

Summary: Choosing the Right Tool

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

Economic Property	**Recommended Architecture**	**Key Mechanism**
Convexity	ICNN	Non-neg weights + Skip conn.
(Utility, Costs)		
Partial Convexity	**Partial ICNN (PICNN)**	Split paths (Convex/Free)
(Value Functions)		
Monotonicity	Monotonic NN (MNN)	Non-neg weights
(Supply/Demand)	Deep Lattice Nets (DLN)	Interpolation
Equilibrium	Deep Equilibrium (DEQ)	Fixed Point Iteration
(Market Clearing)		

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