Continuation Value is All You Need

“Drop-In” Deep Learning for HA Models with Aggregate Shocks

Jeffrey Sun, University of Toronto

Federal Reserve Board, April 30, 2025

Outline

Algorithm Outline

Algorithm Outline

i. Describe a general class of models

ii. Make some observations about it

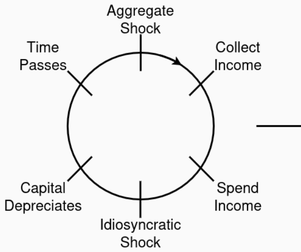
iii. Outline an algorithm

Model

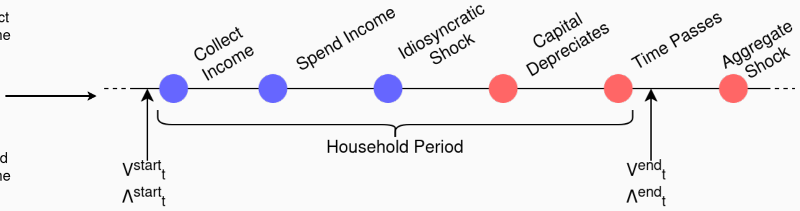
- Time is discrete
- Heterogeneous agents (Households, Firms, etc.)
- Infinite horizon, defined recursively
- Aggregate shock $Z_t \sim \mathcal{D}()$ hits once per period
- Aggregates (including distributions) are deterministic conditional on $\{Z_t\}_t$

Timing

Denote the spell between aggregate shocks the “household period,”



Model As A Circle



One Period of Model

Denote the value function and aggregate state at start and end of household period,

$$V_t^{\text{start}}, V_t^{\text{end}} \quad \text{and} \quad \Lambda_t^{\text{start}}, \Lambda_t^{\text{end}}.$$

Definition: Λ Definition: V

Observation 1: “Within-Period Problem” Solvable Conventionally

Observation 1: There exists a function, typically implementable by conventional methods,

$$\Phi : \left(\Lambda_t^{\text{start}}, V_t^{\text{end}}\right) \mapsto \left(\Lambda_t^{\text{end}}, V_t^{\text{start}}\right),$$

- The “Within-Period Problem (WPP)” is the set of equations defining Φ.
- Φ is the “Solution” to the WPP.
- A specific implementation of Φ is a “Solver” for the WPP.

Argument:

- By assumption, WPP is aggregate-deterministic.
- Solved like one-period model where V_t^{end} is terminal payoff.
- Often solved by iterating V_t^{end} backward, then simulating Λ_t^{start} forward.

Observation 2: "Between-Period Problem"

Observation 2: There exists a function,

$$\Pi : \Lambda_t^{\text{start}} \mapsto V_t^{\text{end}},$$

as long as the model is Markov and has a solution.

- However, Π can be **infeasible** to implement, especially with aggregate uncertainty.
- Brute force method: VFI over every possible $V^{\text{start}}(x_{it}, \Lambda^{\text{start}})$.
- But aggregate state space can be vast, curse of dimensionality binds.

However, this is the *only place* the curse of dimensionality applies...

Observation 3: Π Is Enough For Simulation

Observation 3: If you *somehow* had,

$$\Pi : \Lambda_t^{\text{start}} \mapsto V_t^{\text{end}},$$

then you could simulate forward:

$$\begin{aligned}
\Lambda_t^{\text{end}} &= \Phi_1\big(\Lambda_t^{\text{start}}, \Pi\big(\Lambda_t^{\text{start}}\big)\big) \\
\Lambda_{t+1}^{\text{start}} &= \Omega\big(\Lambda_t^{\text{end}}, Z_t\big) \\
\Lambda_{t+1}^{\text{end}} &= \Phi_1\big(\Lambda_{t+1}^{\text{start}}, \Pi\big(\Lambda_{t+1}^{\text{start}}\big)\big) \\
&\dots
\end{aligned}$$

Indeed, conditional on any candidate Π_0, we have,

$$(\Lambda^{\text{end}}, V^{\text{start}}) \quad = \quad \widetilde{\Phi}\big(\Lambda^{\text{start}}; \Pi_0\big) \quad \equiv \quad \Phi\big(\Lambda^{\text{start}}, \Pi_0(\Lambda^{\text{start}})\big),$$

and a **computable** law of motion,

$$\Gamma(\Lambda^{\text{start}}; \Pi_0, Z) \quad \equiv \quad \Omega(\widetilde{\Phi}_1\big(\Lambda^{\text{start}}; \Pi_0\big), Z).$$

Observation 4: Π Recursively Defined

Observation 4: $\Pi : \Lambda^{\text{start}} \mapsto V^{\text{end}}$ can be recursively defined as:

$$\begin{aligned}
&\forall\ \Lambda^{\text{start}} \in (\text{Aggregate State Space}) \\
V^{\text{end}} &\equiv \Pi(\Lambda^{\text{start}}) \\
&= \mathbb{E}_Z\left[V^{\text{start}\prime} \;\middle|\; \Lambda^{\text{start}} \right] \\
\text{where} \quad V^{\text{start}\prime} &\equiv \widetilde{\Phi}_2\left(\Lambda^{\text{start}\prime}; \Pi\right) \\
\Lambda^{\text{start}\prime} &\equiv \Gamma(\Lambda^{\text{start}}; \Pi, Z)
\end{aligned}$$

(i.e. "Projecting 1 step forward" = "Projecting 2 steps forward and solving 1 step back".)

Essentially a rearrangement of the Bellman equation to define Π.

Π As A Fixed Point

Π is thus a fixed point of the following "lookahead" operator:

$$\mathrm{L}\Pi_0(\Lambda^{\text{start}}) \equiv \mathbb{E}_Z\Big[\widetilde{\Phi}_2\big(\Gamma(\Lambda^{\text{start}};\Pi_0,Z);\Pi_0\big)\Big].$$

It remains only to train a neural network Π_0 to minimize:

$$\Big\|\Pi_0(\Lambda^{\text{start}}) - \mathrm{L}\Pi_0(\Lambda^{\text{start}})\Big\|.$$

When Π is a neural network, this is the "Neural Bellman Equation."

Neural Bellman Equation

Explicitly, for a neural network $\mathcal{N}_\theta$ with parameters θ,

$$\mathrm{NBE}(\theta;\Lambda^{\text{start}}) = \mathcal{N}_\theta(\Lambda^{\text{start}}) - \mathbb{E}_Z\left[\widetilde{\Phi}_2(\Gamma(\Lambda^{\text{start}};\mathcal{N}_\theta, Z);\mathcal{N}_\theta)\right]$$

Measures the failure of $\mathcal{N}_\theta$ to produce a lookahead-consistent V^{end} projection at Λ^{start}.

Algorithm Sketch

Let $\mathcal{N}_\theta$ be a neural network with parameters θ.

1. Initialize θ somehow.
2. Draw N samples Λ_i^{start} somehow.
3. For each i:
   i. Compute $\widehat{V^{\text{end}}}_i \longleftarrow \Gamma\mathcal{N}_\theta(\Lambda_i^{\text{start}})$
   ii. Compute,
   $$g_i \longleftarrow \frac{\partial}{\partial\theta}\left\|\mathcal{N}_\theta(\Lambda_i^{\text{start}}) - \widehat{V^{\text{end}}}_i\right\|^2$$
4. Update $\theta \longleftarrow \theta - \alpha\frac{1}{N}\sum_i g_i$
5. Repeat from Step 2 until convergence.

Literature

Key Departure

Key departure from literature on ML for HA models with aggregate shocks:

- Use **conventional** methods to solve for **everything** but V^{end}: policy, prices, all intermediate value functions, evolution of state, etc.
- In particular, use conventional methods to solve the **policy** function

Key Choices (Within Econ. Lit on ML for HA Models w/ Agg. Shocks)

- Time:
  - **Discrete**
  - Continuous: Gu et al. (2024), Fernández-Villaverde et al. (2023), etc.
- Solution scope
  - **Global**: Most DL based methods
  - Local: Most projection/perturbation methods: Bhandari et al. (2023), Bilal (2023), Auclert et al. (2021), Winberry (2018), etc.
- Policy function
  - **Conventional**: Krusell and Smith (1998), Hull (2015), etc.
  - Deep Learning: Han et al. (2024), Azinovic et al. (2022), Maliar et al. (2021), etc.

Bridging ML and Economics Literatures

One perspective:

- Method is Krusell-Smith, replacing their “finite-moments value function” with a “neural net value function”

Another perspective:

- Method is Approximate Dynamic Programming (ADP) applied to heterogeneous-agent equilibrium models

Implementation

Model Setup

- Standard Krusell-Smith Model Details
- 65 wealth gridpoints, 3 income gridpoints = 195 idiosyncratic gridpoints

Algorithm Sketch

Let $\mathcal{N}_\theta$ be a neural network with parameters θ. $\longleftarrow$ We Are Here

1. Initialize θ somehow.
2. Draw N samples Λ_i^{start} somehow.
3. Compute $\widehat{V^{\text{end}}}_i \longleftarrow \Gamma\mathcal{N}_\theta(\Lambda_i^{\text{start}})$.
4. For each i, compute,
$$g_i \longleftarrow \frac{\partial}{\partial\theta}\left\|\mathcal{N}_\theta(\Lambda_i^{\text{start}}) - \widehat{V^{\text{end}}}_i\right\|^2$$
5. Update $\theta \longleftarrow \theta - \alpha\frac{1}{N}\sum_i g_i$
6. Repeat from Step 2 until convergence.

Basic Neural Network

Idea: Compose matrix multiplication with elementwise nonlinear function.

This can create arbitrary nonlinear relationships between inputs and outputs.

Parameters: $\theta = (M^1, b^1, M^2, b^2)$

$$\begin{aligned}
\text{Input: } & x^1 \\
& y^1 \longleftarrow M^1 x^1 + b^1 \\
& x^2 \longleftarrow \mathrm{elu}(y^1) \\
& y^2 \longleftarrow M^2 x^2 + b^2 \\
& y^3 \longleftarrow \mathrm{elu}(y^2) \\
\text{Output: } & y^3
\end{aligned}$$

Where

$$\mathrm{elu}(x)_i \equiv \begin{cases} x_i & x_i > 0 \\ e^{x_i} - 1 & x_i \leq 0 \end{cases}$$

is an **elementwise** nonlinear function (“activation function.”)

Automatic Differentiation

The real magic of neural networks: automatic differentiation.

For a “label” ℓ_x associated with each input x, and a neural network $\mathcal{N}_\theta(x)$, the gradient

$$\frac{\partial}{\partial\theta}\|\mathcal{N}_\theta(x), \ell_x\|^2$$

can be quickly and automatically computed.

- Can often thus find θ such that $\mathcal{N}_\theta(x)$ approximates/predicts ℓ_x well.
- Theorem: a sufficiently large neural net can approximate *any* θ arbitrarily well

Specific Neural Net Implementation: Step 1

Step 1: Use “Generalized Moments” mean field game-inspired method of Han, Yang, and E (2025) to summarize household state distribution μ.

$$\forall x \in X:$$

$$\begin{aligned}
\gamma_x^{GM,1} &\longleftarrow \mathrm{elu}\left(M^1_{10\times 2}\begin{pmatrix} k_x \\ z_x \end{pmatrix} + b^{GM,1}\right) \\
\gamma_x^{GM,2} &\longleftarrow \mathrm{elu}\left(M^2_{10\times 10}\gamma_x^{GM,1} + b^{GM,2}\right) \\
\gamma^{GM,3} &\longleftarrow \sum_{x\in X} \mu(x)\gamma_x^{GM,2} \\
\gamma^{GM} &\longleftarrow \mathrm{elu}\left(M^3_{10\times 10}\gamma^{GM,3} + b^{GM,3}\right)
\end{aligned}$$

Output: γ^{GM}, containing information about aggregate state distribution.

Specific Neural Net Implementation: Step 2

Step 2: Combine state distribution summary γ^{GM} with idiosyncratic state x and aggregate productivity A.

$$\begin{aligned}
\gamma_x^{HH,1} &\longleftarrow \mathrm{elu}\left(M_{10\times 2}^{HH,1}\begin{pmatrix}k_x\\ z_x\end{pmatrix}+b^{HH,1}\right)\\
\gamma^{A,1} &\longleftarrow \mathrm{elu}\left(M_{10\times 1}^{A,1}A+b^{A,1}\right)\\
\gamma^{A} &\longleftarrow \mathrm{elu}\left(M_{10\times 10}^{A,2}\gamma^{A,1}+b^{A,2}\right)\\
\gamma_x^{HH,2} &\longleftarrow \gamma_x^{HH,1}+\gamma^{GM}+\gamma^{A}
\end{aligned}$$

Output: $\gamma_x^{HH,2}$, containing information about idiosyncratic x and aggregate $\Lambda^{\text{start}} = (\mu, A)$.

Specific Neural Net Implementation: Step 3

Step 3: Put $\gamma_x^{HH,2}$ through three more layers:

$$\begin{aligned}
\gamma^{HH,3} &\longleftarrow \mathrm{elu}\Big(M_{8\times 10}^{HH,3}\gamma_x^{HH,2} + b^{HH,3}\Big)\\
\gamma^{HH,4} &\longleftarrow \mathrm{elu}\Big(M_{8\times 8}^{HH,4}\gamma^{HH,3} + b^{HH,4}\Big)\\
\gamma^{HH,5} &\longleftarrow \mathrm{elu}\Big(M_{5\times 8}^{HH,5}\gamma^{HH,4} + b^{HH,5}\Big)\\
\widehat{V}(x;\Lambda^{\text{start}},\theta) &\equiv M_{1\times 5}^{HH,6}\gamma^{HH,5} + b^{HH,6}
\end{aligned}$$

Output: $\widehat{V}(x;\Lambda^{\text{start}},\theta)$, an estimate of the state-contingent value $V(x;\Lambda^{\text{start}})$.

Krusell-Smith Comparison

The Krusell-Smith *method* is essentially the same, but with $\widehat{V}$ given by,

$$\begin{aligned}
\overline{k} &\longleftarrow \sum_{x \in X} \mu(x) k_x \\
\gamma^{HH} &\longleftarrow \begin{pmatrix} v(k,1;\overline{k},z_g) & v(k,1;\overline{k},z_b) & v(k,0;\overline{k},z_g) & v(k,0;\overline{k},z_b) \end{pmatrix}' \\
\widehat{V}^{KS}(x;A,\mu) &\equiv \begin{pmatrix} \mathbb{1}_{[A=1,z_x=z_g]} & \mathbb{1}_{[A=1,z_x=z_b]} & \mathbb{1}_{[A=0,z_x=z_b]} & \mathbb{1}_{[A=0,z_x=z_b]} \end{pmatrix} \gamma^{HH}
\end{aligned}$$

No need to interpret as "predicting the law of motion of aggregate capital." It's just value function approximation!

Data Representation

- Represent V_t^{start}, ${\mu^{\text{start}}}_t$ by data arrays $A_t^{V^{\text{start}}}$, $A_t^{\mu^{\text{start}}}$. Similarly, $A_t^{V^{\text{end}}}$, $A_t^{\mu^{\text{end}}}$.
- Φ is then implemented as a function taking arrays to arrays:

$$\Phi : \left(A^{V^{\text{end}}}, A^{\Lambda^{\text{start}}}, S_t^{\text{start}}\right) \mapsto \left(A^{V^{\text{start}}}, A^{\Lambda^{\text{end}}}, S_t^{\text{end}}\right)$$

  where S_t^{start} and S_t^{end} are other state variables.
- Implemented **conventionally** (no neural net)

Algorithm Sketch

Let $\mathcal{N}_\theta$ be a neural network with parameters θ.

1. Initialize θ somehow. $\longleftarrow$ We Are Here
2. Draw N samples Λ_i^{start} somehow.
3. Compute $\widehat{V^{\text{end}}}_i \longleftarrow \Gamma\mathcal{N}_\theta(\Lambda_i^{\text{start}})$.
4. For each i, compute,
$$g_i \longleftarrow \frac{\partial}{\partial\theta}\left\|\mathcal{N}_\theta(\Lambda_i^{\text{start}}) - \widehat{V^{\text{end}}}_i\right\|^2$$
5. Update $\theta \longleftarrow \theta - \alpha\frac{1}{N}\sum_i g_i$
6. Repeat from Step 2 until convergence.

Initializing θ

Starting with a completely random θ may lead to terrible initial simulations.

Several options here:

1. Pre-train θ to minimize

$$\left\| \mathcal{N}_\theta(\Lambda^{\text{start}})(x) - \left(k_x + \frac{z_x}{1-\beta}\right) \right\|$$

2. Pre-train θ on $(\Lambda^{\text{start}}, V^{\text{end}})$ data-label pairs from:
   - 2.1 Deterministic version of model
   - 2.2 Krusell-Smith *method* solution of full model

Algorithm Sketch

Let $\mathcal{N}_\theta$ be a neural network with parameters θ.

1. Initialize θ somehow.
2. Draw N samples Λ_i^{start} somehow. $\longleftarrow$ We Are Here
3. Compute $\widehat{V^{\text{end}}}_i \longleftarrow \Gamma\mathcal{N}_\theta(\Lambda_i^{\text{start}})$.
4. For each i, compute,
$$g_i \longleftarrow \frac{\partial}{\partial\theta}\left\|\mathcal{N}\theta(\Lambda_i^{\text{start}}) - \widehat{V^{\text{end}}}_i\right\|^2$$
5. Update $\theta \longleftarrow \theta - \alpha\frac{1}{N}\sum_i g_i$
6. Repeat from Step 2 until convergence.

Sampling Λ^{start}

Options:

1. Draw from simple (e.g. uniform) distribution
2. Draw from ergodic distribution induced by θ
3. Draw from ergodic distribution induced by some other Π_0, in MCMC spirit
    - E.g. Krusell-Smith
4. Take current data state as starting point, sample from possible states within T periods of present

Sampling Λ^{start}

I use MCMC-style sampling:

1. Simulate forward T "burn-in" periods using
$$\Lambda^{\text{start}}_{t+1} \longleftarrow \Gamma(\Lambda^{\text{start}}_t; \mathcal{N}_\theta, Z)$$
2. For $t > T$, add Λ^{start}_t to sample if $t \equiv 0 (\text{mod } n)$ for some n

Basic challenge: Data generation depends on $\mathcal{N}_\theta$.

Algorithm Sketch

Let $\mathcal{N}_\theta$ be a neural network with parameters θ.

1. Initialize θ somehow.
2. Draw N samples Λ_i^{start} somehow.
3. Compute $\widehat{V^{\text{end}}}_i \longleftarrow \Gamma\mathcal{N}_\theta(\Lambda_i^{\text{start}})$. $\longleftarrow$ We Are Here
4. For each i, compute,
$$g_i \longleftarrow \frac{\partial}{\partial\theta}\left\|\mathcal{N}\theta(\Lambda_i^{\text{start}}) - \widehat{V^{\text{end}}}_i\right\|^2$$
5. Update $\theta \longleftarrow \theta - \alpha\frac{1}{N}\sum_i g_i$
6. Repeat from Step 2 until convergence.

Within-Period Problem

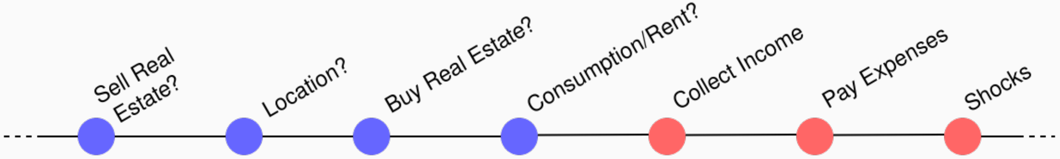
Within-period problem: find,

$$\Phi : \left(\Lambda_t^{\text{start}}, V_t^{\text{end}}\right) \mapsto \left(\Lambda_t^{\text{end}}, V_t^{\text{start}}\right).$$

Performance strategy: break into “stages”: effectively “sub-periods.”

Krusell-Smith model is easy, but we can get much richer.

Sub-Periods: "Conventionally" Solved but *Modular*



Similar decomposition to Auclert et al. (2023)

Multiplicative performance benefits:

1. Simple, optimized, prepackaged for CPU/GPU/cluster
2. Search over one dimension at a time
3. Identify and overcome bottlenecks (interpolation, grid search) in non-vectorized Julia

Fixing Bottlenecks: Dynamic Information Sharing

Example: Consumption-saving by grid search

- If $MPC \geq 0$, then my optimal saving is between my wealth-neighbors'
- Don't need to search over entire axis!
- By "sharing" information between wealth-neighbors: $O(N^2) \rightarrow O(N \log N)$
- Incompatible with vectorization (Python, Matlab) but fast in Julia

Pre-Packaged Stages

Another benefit of stages: can be pre-prepared

- Possibly separate project
- Aim to provide highly efficient state implementations for CPU, GPU, and cluster

Household Problem Code Example

```julia
function solve_period!(prealloc, V_next, params)
    V_preshock = get_V_preshock(prealloc, V_next)

    V_consume = get_V_preconsume(V_preshock, prealloc)

    V_income = get_V_preincome(V_consume, prealloc, params)
    enforce_borrowing_constraint!(V_preincome, prealloc)

    V_premove = get_V_premove(V_preincome, prealloc, params)

    V_end = YOUR_CODE_HERE(V_premove, prealloc, params)
end
```

Solving for Prices

Options:

1. Solve analytically (e.g. Krusell-Smith)
2. Solve via rootfinding (accurate but slow)
3. Use additional neural network (e.g. Azinovic et al. (2022), Deep Equilibrium Nets)

Within-Period Problem

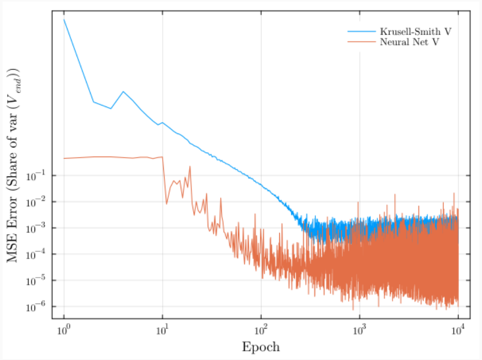
Challenge: Performance of Within-Period Solver is crucial.

Accuracy

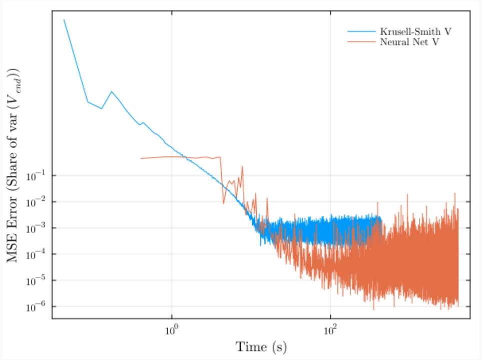
Accuracy

- Strength: $\Phi : (\Lambda_t^{\text{start}}, V_t^{\text{end}}) \mapsto (\Lambda_t^{\text{end}}, V_t^{\text{start}})$ is immediately correct from the beginning
- Only need to check Neural Bellman Equation error
- Weakness: Computing Euler Equation errors nontrivial
- Hardware: One laptop CPU thread (i9-13900H)

Learning Curve Comparison



V error as share of V variance by training epoch



Relative V error by training time (CPU)

- Both stop improving within about 300 epochs (122s for NN, 13s for KS)
- After Epoch 300, mean $\log_{10}$ error is -4.431 for NN, -3.152 for KS

Optimizations

Benchmark

- 1000 locations, 129 wealth states, 5 income types, 6 age groups = 3.87m gridpoints
- Single-thread CPU
- Language: Julia
- Strawman: Jeffrey, May 2023
- One evaluation of household problem
- Initial time: 218s (3m38s)

Low-Hanging Fruit

- **Initial**: 218s
- **Memory Preallocation**: 218s $\rightarrow$ 152s
- **(Almost) Automatic Multithreading**: 152s $\rightarrow$ 49s
- **32 Bit Precision**: 49s $\rightarrow$ 31.9s

Individual Stage Optimizations

Choose Location	25.2s → 9.78s → 0.053s
Receive Income	0.38s → 0.019s
Choose Consumption	0.498s → 0.025s
Income Shock	4.52s → 0.028s

Overall: 31.9s → 0.353s (= 0.126s listed stages + 0.228s other)

Choose Location

$$\text{Let} \qquad V_{ts\iota}^{\text{start}}(\ell) = V_{ts}^{\text{start}}(k_{\iota t-1}, z_{\iota t}, \ell, a_{\iota t}), \quad V_{ts\iota}^{\text{end}}(\ell) \text{ similar}$$

where ι indexes all household types up to location.

The i.i.d. Gumbel location preference shocks imply:

$$\exp\left(\psi V_{ts\iota}^{\text{start}}(\ell)\right) = \sum_{\ell'} \exp\left(\psi\left(V_{ts\iota}^{\text{end}}(\ell') - D_{\ell\ell'}\right)\right)$$

$$P(\ell' = \ell_0 \mid \ell) = \frac{\exp\left(\psi\left(V_{ts\iota}^{\text{end}}(\ell') - D_{\ell\ell'}\right)\right)}{\exp\left(\psi V_{ts\iota}^{\text{start}}(\ell)\right)}$$

$$\lambda_{ts\iota}^{\text{end}}(\ell) = \sum_{\ell'} P(\ell' = \ell \mid \ell) \lambda_{ts\iota}^{\text{start}}(\ell')$$

First optimization: Precompute $\exp\left(\psi V_{ts\iota}^{\text{start}}(\ell)\right)$, then $P(\ell' = \ell_0 \mid \ell)$, then $\lambda_{ts\iota}^{\text{end}}(\ell)$

Time: 25.2s $\rightarrow$ 9.78s

Choose Location

Second optimization: observe that (with $\otimes$ and $\oslash$ elementwise mult. and div.)

$$
\begin{aligned}
\widetilde{V}_{ts}^{\text{start}} &= D\widetilde{V}_{ts}^{\text{end}} \\
\Lambda_{ts}^{\text{end}} &= \widetilde{V}_{ts}^{\text{end}} \otimes \left(D'\Lambda_{ts}^{\text{start}} \oslash \widetilde{V}_{ts}^{\text{start}}\right) \\
\text{where matrices} \quad D_{\ell\ell'} &= \exp\left(-\psi D_{\ell\ell'}\right) \\
\left(\widetilde{V}_{ts}^{\text{start}}\right)_{\ell\iota} &= \exp\left(\psi V_{ts\iota}^{\text{start}}(\ell)\right) \\
\left(\widetilde{V}_{ts}^{\text{end}}\right)_{\ell\iota} &= \exp\left(\psi V_{ts\iota}^{\text{end}}(\ell)\right) \\
\left(\Lambda_{ts}^{\text{end}}\right)_{\ell\iota} &= \lambda_{ts\iota}^{\text{end}}(\ell)
\end{aligned}
$$

No matter the size of the state space, just two matrix multiplications!

Time: 9.78s $\rightarrow$ 0.053s

The Power of Matrix Multiplication

Why is matrix multiplication 200 faster than an explicit loop?

- Surprising algorithms exist to multiply two matrices in as little as $O(n^{2.371552})$ time
- Most CPUs have specialized hardware for matrix multiplication
- Pretty much exactly the same thing works for CES production functions, etc.
- Similar approach to optimizing income shocks, or any finite-state Markov process

Choose Consumption

- Strategy: Gridsearch
- If $MPC \geq 0$, then my optimal saving is between my wealth-neighbors'
- Don't need to search over entire axis!
- By "sharing" information between wealth-neighbors: $O(N^2) \to O(N \log N)$
- Incompatible with vectorization (Python, Matlab) but fast in Julia
- Similar approach for linear interpolation of many gridpoints

Global Solution

- A neural network is trained to predict V^{end}. Everything else is conventional
- In particular, no neural network used to approximate policy function

Conclusion

Conclusion

- Describe a solution method for HA models with agg. shocks that overcomes curse of dimensionality (of aggregate state)
- Method uses neural nets only where needed – solution otherwise conventional
- Much complexity is offloaded to **Within-Period Problem** solver
- In other work, provide tools to implement WPP flexibly, easily, performantly

Discussion

- Key advantages:
  - Complex household problems supported
  - No need to train policy network
- Disadvantages:
  - Individual state must be low-dimensional (≤ 6 or so)
  - In more complex models, prices require inner loop around IPP or price neural net à la Azinovic et al. (2022)
- Future work:
  - Assess other error metrics, e.g. Euler equation error, risk premia
  - Compare economics of solutions

Appendix

Definition: Aggregate State

Λ_t^{start} (Λ_t^{end}) is the aggregate state at the start (end) of household period t.

$$\Lambda_t^{\text{start}} \equiv \left(\mu_t^{\text{start}}, S_t^{\text{start}}\right) \qquad\qquad (\Lambda_t^{\text{end}} \equiv \left(\mu_t^{\text{end}}, S_t^{\text{end}}\right))$$

where at the start (end) of household period t,

- μ_t^{start} (μ_t^{end}) is the distribution of idiosyncratic states $x^{\text{start}} \in X^{\text{start}}$ ($x^{\text{end}} \in X^{\text{end}}$).
- S_t^{start} (S_t^{end}) is all other aggregate state.

Without loss of generality, parametrize the aggregate shock so that,

$$\Lambda_{t+1}^{\text{start}} = \Omega(\Lambda_t^{\text{end}}, Z_t) \quad \text{where} \quad Z_t \sim \mathcal{D}().$$

Definition: Value Function

For household i, $V_t^{\text{start}}(x_i)$ is NPV of future utility given state x_i^{start} at the beginning of t,

$$\begin{aligned} V_t^{\text{start}}(x_i) &\equiv V^{\text{start}}(x_{it}, \Lambda^{\text{start}}) \equiv \mathbb{E}\left[\sum_{s=0}^{\infty} \beta^s u_{i,t+s} \;\middle|\; x, \Lambda^{\text{start}} \right]. \\ V_t^{\text{end}}(x_i) &\equiv V^{\text{end}}(x_{it}, \Lambda^{\text{start}}) \equiv \mathbb{E}\left[\sum_{s=1}^{\infty} \beta^s u_{i,t+s} \;\middle|\; x, \Lambda^{\text{start}} \right]. \end{aligned}$$

- V_t^{start} refers to the whole *function*, e.g. an array on a grid over X^{start}.
- Conditioning on Λ_t^{start} and Λ_t^{end} are equivalent by aggregate-determinism within t.

Algorithm Details

- I use a gridded CDF representation of Λ, but using a finite number of agents is also possible. However, they have to interpolate over continuation V
- Training data for each simulated period is $A_{it}^{V\text{end}}$ together with

$$\mathbb{E}\left[A_{i,t+1}^{V,\text{start}} \mid \Gamma_{it}\right] = \frac{1}{K}\sum_{k=1}^{K} \text{IPP}_V\left(\mathcal{N}(\Omega(\Gamma_{it},k)\ ;\ \theta_i), A_{it}^{\Lambda\text{end}}, \Omega(\Gamma_{it},k)\right)$$

- For large models, if memory is constrained, you can update θ_i as you go, accumulating gradients but not storing the entire simulation

Back

Neural Network Implementation

Neural net $\mathcal{N}(A_{\Gamma}^{\Lambda\text{start}} ;\theta)$ has following components:

1. Generalized-moment of Han et al. (2023):

$$\text{GM}_{\Gamma} = \sum_{j\in J} \left(A_{\Gamma}^{\Lambda\text{start}}\right)_j \mathcal{N}_{\text{GM}}(X_j)$$

2. One layer $(1 \Rightarrow 10)$ neural net on aggregate productivity A
3. Dense feedforward neural net on input: $(X_j, \text{GM}_{\Gamma}, \mathcal{N}_A(A))$
   - Three hidden layers with 8, 8, and 5 neurons
   - Elu activation

Back

Krusell-Smith Model Details

- 65 wealth gridpoints
- 3 income gridpoints
- $\beta = 0.98$
- Income process by Tauchen discretization with persistence 0.95 and std 0.1
- Log-linear wealth grid from 1k to 10m
- Income states: 15.4k, 40.3k, 105.4k
- Risk aversion: 0.9
- Capital share: 0.36
- Depreciation rate: 0.025

Back

Krusell-Smith Method Details

- 5 aggregate capital gridpoints: 100k, 150k, 200k, 250k, 300k
- Linear extrapolation outside aggregate capital grid
- 2 aggregate productivity states: 0.5 and 1.0
- Unlike Krusell and Smith (1998), use gridded CDF population distribution representation for cleaner comparison

Back