Exploiting Symmetry in High-Dimensional Dynamic Programming

Mahdi Ebrahimi Kahou[1] Jesús Fernández-Villaverde[2] Jesse Perla[1] Arnav Sood[3]

August 2, 2022

[1]University of British Columbia, Vancouver School of Economics

[2]University of Pennsylvania

[3]Carnegie Mellon University

Motivation

Motivation

- Most models in macro (and many other fields) deal with either:
  1. Representative agent (canonical RBC and New Keynesian models)–or small # of types.
  2. A continuum of agents (canonical Krusell-Smith model, mean-field games).
- When it is feasible, the "continuum" approximations can make things easier. But...
  - Requires jumping through hoops rewriting the model and isn't always possible.
  - Does it limit the economic insights? Steady-states/transitions/shock sizes/etc.
- Existing methods: deal with limitations with aggregate shocks/transitions either
  1. Krusell and Smith: Works in practice, but (seemingly) not in theory.
  2. Perturbative Solutions: Perfectly fine if aggregate shocks are small and distributions make sense as local to a non-stochastic steady state. An economic, not implementation, question.

Limitations of existing methods

- Aggregate uncertainty and transition dynamics are difficult with a continuum approximation
  - Krusell-Smith style approaches: powerful but heuristic, require intimate knowledge of problem structure
  - Perturbative solutions may lose key economics since local to non-stochastic distributional steady state.
- Many interesting models have a finite number of agents:
  - Models with many locations (countries, regions, metropolitan areas, industries).
  - Models of industry dynamics with many firms and industries.
  - Models of networks.
  - Even bread-and-butter heterogeneous agent labor models (e.g., overlapping generations, different types)
- Lots of interesting models we cannot solve (e.g. growth, path-dependence, networks)
  - Avoid selection bias: economics driven by the models we can solve vs. the models we need?

Why did we use a continuum of agents? The curse of dimensionality

- In any models where the distribution impacts decisions (and vice-versa), agents need to keep track and forecast their own states and the states of everyone else—and form ergodic distribution!
  - Sometimes you can reduce the dimension by hand (e.g. continuum + no aggregate shocks)
  - Unavoidable in (most) models with a finite number of agents and aggregate shocks
- Two components to the curse of dimensionality (Bellman, 1958, p. IX):
  1. The cardinality of the state space is enormous: memory requirements, update of coefficients, ...
     - With 266 state variables, a grid of 2 points per state (low and high) or, equivalently, a tensor of 2 polynomials per state (a level and a slope) have more elements than the Eddington number, the estimated number of protons ($\approx 10^{80}$) in the universe.
  2. Even with an approximation, you need to evaluate highly-dimensional conditional expectations over every idiosyncratic shock: continuation value function, Euler equations, LOMs,...
     - Computing integrals naively scales exponentially in the number of dimensions.
     - Monte-carlo integration (e.g. using MCMC) works for high dimensions (sometimes slowly) for a single integral, but for dynamic programming expectations are formed for every possible state!

Symmetry to the rescue!

- We introduce permutation-invariant dynamic programming: formalizes a large class of methods we explicitly (e.g. mean-field games) or implicitly (e.g. Krusell-Smith) already used.
- Common feature of many (most?) models of interest: the policy functions of the agents are the same, they are just evaluated at different points.
- Example: if productivity & capital are the idiosyncratic states, then two firms with the same states make the same decisions (mixed-strategy would be fine)
    - Might happen if prices are invariant to relabeling under all the permutations of other agents' states. In equilibrium, the Walrasian auctioneer removes indexes!

Continuum approximations are an extreme version of imposing symmetry.

Why does permutation invariance help?

Permutation invariance tackles the two components of the "curse of dimensionality":

1. The value and policy functions belong to the family of permutation-invariant functions $\Rightarrow$ they can be represented (exactly!) using a latent dimension (possibly much lower than the number of states).
    - It helps to understand why the Krusell-Smith method works as $N \to \infty$, and why finite N is enough.
2. A (fast) concentration of measure appears $\Rightarrow$ a "fancy" law of large numbers for equilibrium objects such as value and policy functions.
    - We can calculate conditional expectations with even a single Monte Carlo draw from the distribution of idiosyncratic shocks.

Thus, we can handle models with thousands of state variables. Our application today has up to 10,000 states. Perfectly feasible (given enough memory) to handle millions of states.

A deep learning approach

- **Deep Learning:** function approximation with a huge number of parameters ($\gg$ data or grid points), usually fit with specialized stochastic optimization methods.
- Otherwise, standard recursive economics, albeit in high dimensions. No agent-based modeling.
- We show how to train a neural network that implements the permutation-invariant dynamic programming problem as dictated by our representation theorem.
- Strictly speaking, neural networks are not required. You only need a flexible functional form.
- Neural networks have some numerical advantages, though:
  1. Universal nonlinear approximators that scale very well.
  2. Great libraries such as `PyTorch Lightning`, easy to implement invariances.
  3. Massively parallel (our architectures are implemented in GPUs).
  4. An emerging theory of generalizability (i.e. extrapolation and interpolation).

Summarizing our contribution

- **Method** for solving **high-dimensional** dynamic programming problems and competitive equilibria
  - Dynamic models with heterogeneity & idiosyncratic/aggregate shocks
  - Global solution starting from an initial condition
  - No backwards induction or even a calculations of a steady-state
  - Dimensionality is a **bless**; only a **curse** where symmetry doesn't exist
- Reevaluate **why macro tricks work** so well, and how far they can be pushed. In particular
  - Why Krusell-Smith works in theory, and how far it can be pushed (e.g. "deep learning")
  - Continuum approximations, "MIT shocks," and mean-field games
- Some teasers (from a "**deep-learning**" based implementation)
  - 10000+ dimensional state-spaces are not a problem
  - 10000+ dimensional expectations with one Monte-carlo draw
  - Fit linear case with 17.7 K parameters fit with only 3 data points!
- Primary goal: develop the tools to solve **(previously) intractable models**

Application

How do we pick our application to show how all this works?

- In terms of application, there are two routes:
  1. I can introduce a sophisticated application where our method "shines."
  2. Or, I can show you how our ideas work in a well-known example.
- Besides, with a sophisticated application, how do you know our "solution" works?
- So, let us present a well-known example (with a twist)...
- ...and leave for another day the more sophisticated applications.

Our application

A variation of the Lucas and Prescott (1971) model of investment under uncertainty with N firms.

Why?

1. Ljungqvist and Sargent (2018), pp. 226-228, use it to introduce recursive competitive equilibria.
2. Simple model that fits in one slide.
3. Under one parameterization, the model has a known LQ solution, which gives us an exact benchmark:
   - 3.1 We can show that our solution will be extremely accurate.
   - 3.2 The classical control solution is of complexity $\mathcal{O}(N^3)$, whereas our solution is $\mathcal{O}(1)$ for reasonable N.
4. By changing one parameter, the model is nonlinear and, yet, our method handles the nonlinear case as easily as the LQ case and, according to the Euler residuals, with high accuracy.

A 'big X, little x' dynamic programming problem

Consider:

$$
\begin{aligned}
v(x, X) &= \max_{u} \left\{ r(x, u, X) + \beta \mathbb{E}\left[v(x', X') \right] \right\} \\
\text{s.t. } x' &= g(x, u) + \sigma w + \eta \omega \\
X' &= G(X) + \Omega W + \eta \omega 1_N
\end{aligned}
$$

where:

1. x is the individual state of the agent.
2. X is a vector stacking the individual states of all of the N agents in the economy.
3. u is the control.
4. w is random innovation to the individual state, stacked in $W \sim \mathcal{N}(0_N, \mathrm{I}_N)$ and where, w.l.o.g., $w = W_1$.
5. $\omega \sim \mathcal{N}(0, 1)$ is a random aggregate innovation to all the individual states.

Some preliminaries

- A permutation matrix is a square matrix with a single 1 in each row and column and zeros everywhere else.
  - These matrices are called "permutation" because, when they premultiply (postmultiply) a conformable matrix A, they permute the rows (columns) of A.
- Let $\mathcal{S}_N$ be the set of all $n!$ permutation matrices of size $N \times N$. For example:

$$S_2 = \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\}$$

- (If you know about this): $\mathcal{S}_N$ is the *symmetric group* under matrix multiplication.
  - The algebraic properties of the symmetric group will be doing a lot of the heavy lifting in the proof of our theorems, but you do not need to worry about it.

Permutation-invariant dynamic programming

A 'big X, little x' dynamic programming problem is a permutation-invariant dynamic programming problem if, for all $(x, X) \in \mathbb{R}^{N+1}$ and all permutations $\pi \in \mathcal{S}_N$, the reward function r is permutation invariant:

$$r(x, u, \pi X) = r(x, u, X)$$

the deterministic component of the law of motion for X is permutation equivariant:

$$G(\pi X) = \pi G(X)$$

and the covariance matrix of the idiosyncratic shocks satisfies

$$\pi \Omega = \Omega \pi$$

Permutation invariance of the optimal solution

Proposition

The optimal solution of a permutation-invariant dynamic programming problem is permutation invariant. That is, for all $\pi \in \mathcal{S}_N$:

$$u(x, \pi X) = u(x, X)$$

and:

$$v(x, \pi X) = v(x, X)$$

Main result I: Representation of permutation-invariant functions

Proposition (based on Wagstaff et al., 2019)

Let $f : \mathbb{R}^{N+1} \to \mathbb{R}$ be a continuous permutation-invariant function under $\mathcal{S}_N$, i.e., for all $(x, X) \in \mathbb{R}^{N+1}$ and all $\pi \in \mathcal{S}_N$:

$$f(x, \pi X) = f(x, X)$$

Then, there exist a latent dimension $L \leq N$ and continuous functions $\rho : \mathbb{R}^{L+1} \to \mathbb{R}$ and $\phi : \mathbb{R} \to \mathbb{R}^L$ such that:

$$f(x, X) = \rho\left(x, \frac{1}{N}\sum_{i=1}^{N}\phi(X_i)\right)$$

- This proposition should remind you of Krusell-Smith!. Consider representing the mean vs. max of X
  - Mean: use $\phi(X) = X$ and $\rho(x, y) = y$. $L = 1$
  - Max: need to setup a system for ρ to invert, and $L = N$.
  - $\sum_{i=1}^{N}$ not special. Use $f(x, X) = \rho\left(x, \max_i\{\phi(X_1), \ldots \phi(X_N)\}\right)$ given economic insights.
- If $L = N$ then would it really help? **Yes in practice.** Geometry for fitting is completely different.

Main result II: Concentration of measure

Concentration of measure when expected gradients are bounded in N

Suppose $z \sim \mathcal{N}(0_N, \Sigma)$, where the spectral radius of Σ, denoted by $\rho(\Sigma)$, is independent of N and $f : \mathbb{R}^N \to \mathbb{R}$ is a function with expected gradient bounded in N. Then for any z_0 drawn from z:

$$\mathbb{P}\left(\left|f(z_0) - \mathbb{E}\left[f(z)\right]\right| \geq \epsilon\right) \leq \frac{\rho(\Sigma) C}{\epsilon^2} \frac{1}{N}$$

As Ledoux (2001) puts it: "A random variable that depends in a Lipschitz way on many independent variables (but not too much on any of them) is essentially constant."

With concentration of measure, dimensionality is not a curse; it is a blessing!

A permutation-invariant economy

- Industry consisting of $N > 1$ firms, each producing the same good.
- A firm i produces output x with x units of capital.
- Thus, the vector $X \equiv [x_1, \dots x_N]^\top$ is the production (or capital) of the whole industry.
- The inverse demand function for the industry is, for some $\nu \geq 1$ (this is our twist!):

$$p(X) = 1 - \frac{1}{N} \sum_{i=1}^{N} x_i^\nu$$

- The firm does not consider the impact of its individual decisions on $p(X)$.
- Due to adjustment frictions, investing u has a cost $\frac{\gamma}{2} u^2$.
- Law of motion for capital $x' = (1 - \delta)x + u + \sigma w + \eta\omega$ where $w \sim \mathcal{N}(0, 1)$ an i.i.d. idiosyncratic shock, and $\omega \sim \mathcal{N}(0, 1)$ an i.i.d. aggregate shock, common to all firms.
- The firm chooses u to maximize $\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t \left(p(X)x - \frac{\gamma}{2} u^2\right)\right]$.

Recursive problem

- The recursive problem of the firm taking the exogenous policy $\hat{u}(\cdot, X)$ for all other firms as given is:

$$
\begin{aligned}
v(x, X) = \max_{u} \left\{ p(X)x - \frac{\gamma}{2}u^2 + \beta \mathbb{E}\left[v(x', X')\right] \right\} \\
\text{s.t. } x' = (1-\delta)x + u + \sigma w + \eta\omega \\
X_i' = (1-\delta)X_i + \hat{u}(X_i, X) + \sigma W_i + \eta\omega, \quad \text{for } i \in \{1, ..., N\}
\end{aligned}
$$

Equilibrium and Euler equation

Definition

A recursive permutation-invariant competitive equilibrium is a $v(x, X)$, $\hat{u}(x, X)$, and laws of motion for capital such that:

- Given $\hat{u}(x, X)$, $v(x, X)$ is the value function solving the recursive problem from previous slide for each agent and $u(x, X)$ is the optimal policy function.
- The optimal policy is symmetric, i.e., $u(x, X) = \hat{u}(x, X)$.
- The laws of motion for capital satisfy:

$$X_i' = (1 - \delta)X_i + \hat{u}(X_i, X) + \sigma W_i + \eta\omega, \quad \text{for } i \in \{1, ..., N\}$$

The Euler equation for the firm:

$$\gamma u(x, X) = \beta \mathbb{E}\left[p(X') + \gamma(1 - \delta)u(x', X')\right]$$

Solving the Model

Key techniques in theory and practice

The key tools used throughout are

1. Implement **symmetry** and **invariants** in the design of the approximation (e.g. sets $\neq$ vectors with flexible forms like neural networks) using **economic theory/models**
2. Calculate high-dimensional integrals with a **concentration of measure**
3. Draw points local to simulations from initial conditions of interest using the approximation—**don't worry about stationarity**
   - Provides accuracy tradeoff: care more about trajectories local to a small set of initial conditions rather than in all of $\mathbb{R}^N$—most of which you would never reach.
   - "Corners" become increasingly irrelevant in higher dimensions because they are harder to reach, and geometry in high-dimensional probability has most of the "volume" in corners as dimension increases.
4. The solution to **over-fitting** is adding **more parameters**!

Then let the magic of generalization in highly over-parameterized neural networks do the rest (i.e. the **double-decent** phenomena, which we can discuss later)

Solving the model

- We want to find a global solution that is accurate beyond a ball around some particular X^{ss} (usually the steady state of the model).
- Why? Compute transitional dynamics from far away the steady state, study large shocks, ...
- From the representation of permutation-invariant functions, we know that the policy function that satisfies the previous Euler equation has the form:

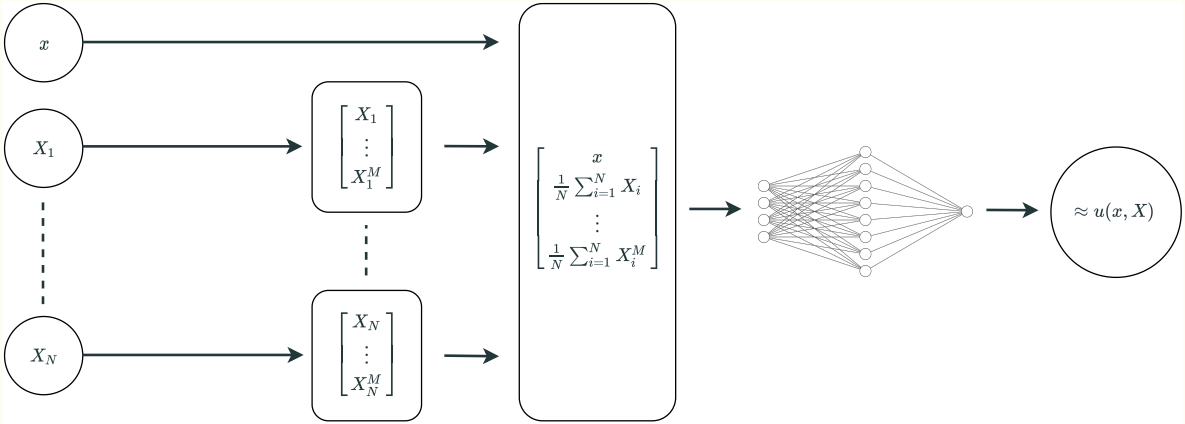
$$u(x, X) = \rho\left(x, \frac{1}{N}\sum_{i=1}^{N}\phi(X_i)\right)$$

- But, in general, we do not know $\rho(\cdot)$ or $\phi(\cdot)$.
- Thus, we will approximate $\rho(\cdot)$ and $\phi(\cdot)$ using deep learning.

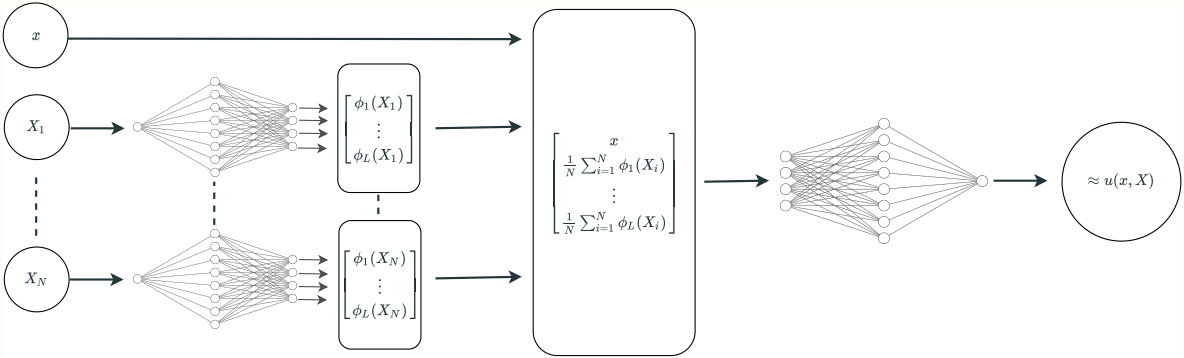
Our deep learning architectures

- We will specify several deep learning architectures $\mathcal{H}(\theta)$:
  1. ϕ is approximated as a function of a finite set of moments à la Krusell-Smith but in a fully nonlinear way as in Fernández-Villaverde et al. (2019). We use 1 and 4 moments.
  2. ϕ is approximated by a flexible ReLU network with two layers in ϕ and 128 coefficients).
- The baseline ϕ(Identity), ϕ(Moments), and ϕ(ReLU) have 49.4K, 49.8K, and 66.8K coefficients respectively regardless of N.
- In all cases, ρ is a highly parameterized neural network with four layers.
- A surprising benefit of a high-dimensional approximation is the "double-descent" phenomenon in machine learning (see Belkin et al., 2019, and Advani et al., 2020): more coefficients makes it easier to find minimum-norm solutions.
- All the code in `PyTorch Lightning` and run on GPUs.

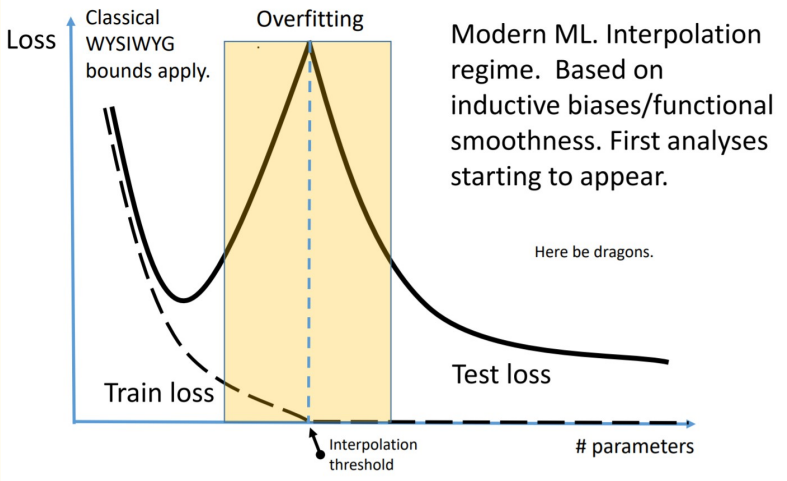
Architecture using summary statistics as moments for ϕ



Architecture using neural networks (e.g. ReLU) for ϕ



The cure to overfitting is to add more parameters



Training and calibration

Algorithm Network training

1: Given by network architecture $\mathcal{H}(\theta)$ for $u(x, X)$, define the Euler residuals:

$$\varepsilon(x, X; \theta) \equiv \gamma u(x, X) - \beta \mathbb{E}\left[P(X') + \gamma(1-\delta)u(x', X')\right]$$

2: Pick $\{X^m(0), ..., X^m(T)\}$ for $m = 1, .., M$ trajectories given some initial point of interest.

3: Evaluate $\varepsilon_{m,t}(x, X; \theta)$ for some or all the points above.

4: Solve using ADAM (a stochastic gradient descent with adaptive moment estimation):

$$\min_{\theta} \frac{1}{M} \sum_{m=1}^{M} \sum_{t=0}^{T} \left(\varepsilon_{m,t}(x, X; \theta)\right)^2$$

- Parameter values: $\beta = 0.95, \gamma = 90, \sigma = 0.005$, and $\eta = 0.001$. Idiosyncratic risk 5 times larger than aggregate risk.
- We will study two cases: linear ($\nu = 1$) and nonlinear ($\nu > 1$) demand functions.

Linear case to verify algorithms and methos

- With $\nu = 1$, we have a linear demand function: $p(X) = 1 - \frac{1}{N}\sum_{i=1}^{N} x_i$.
- It generates an LQ dynamic programming problem (only the mean of x_i matters!).
- We can find the exact $u(x, X)$ using the linear regulator solution.
- The LQ solution gives us a benchmark against which we can compare our deep learning solution.
- The neural network "learns" very quickly that the solution is $u(x, X) = H_0 + \frac{1}{N} H_1 \sum_{i=1}^{N} x_i$ and finds a high-dimensional approximation which matches that for the training grid.
- We also compute a modified linear regulator solution with *one* Monte Carlo draw instead of setting the individual shocks to zero: illustrates how concentration of measure works.
- Bonus point: we show how to implement this modified linear regulator solution. Useful for non-Gaussian LQ problems where certainty equivalence does not hold.

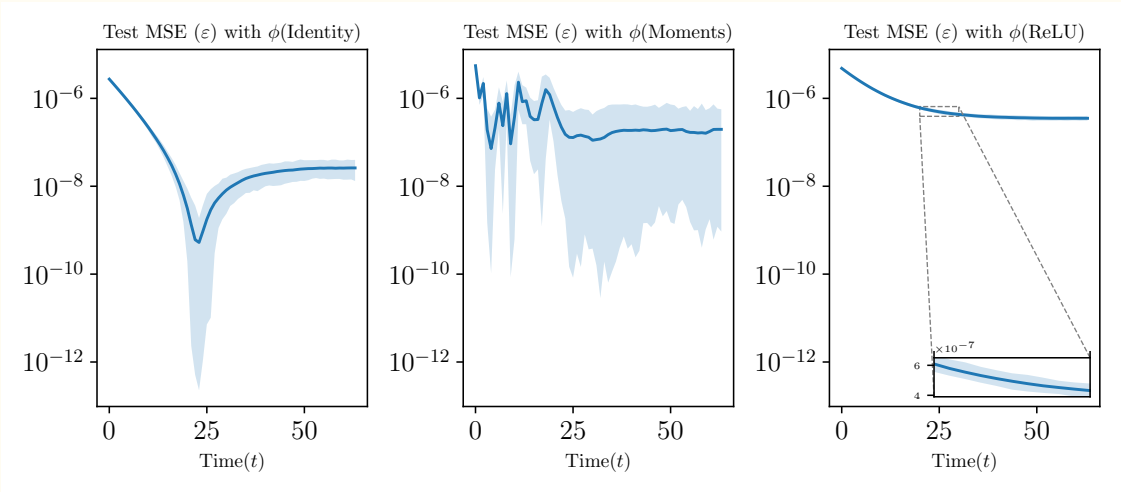


Figure 1: The Euler residuals for $\nu = 1$ and $N = 128$ for ϕ(Identity), ϕ(Moments), and ϕ(ReLU). The dark blue curve shows the average residuals along equilibrium paths for 256 different trajectories. The shaded areas depict the 2.5th and 97.5th percentiles.

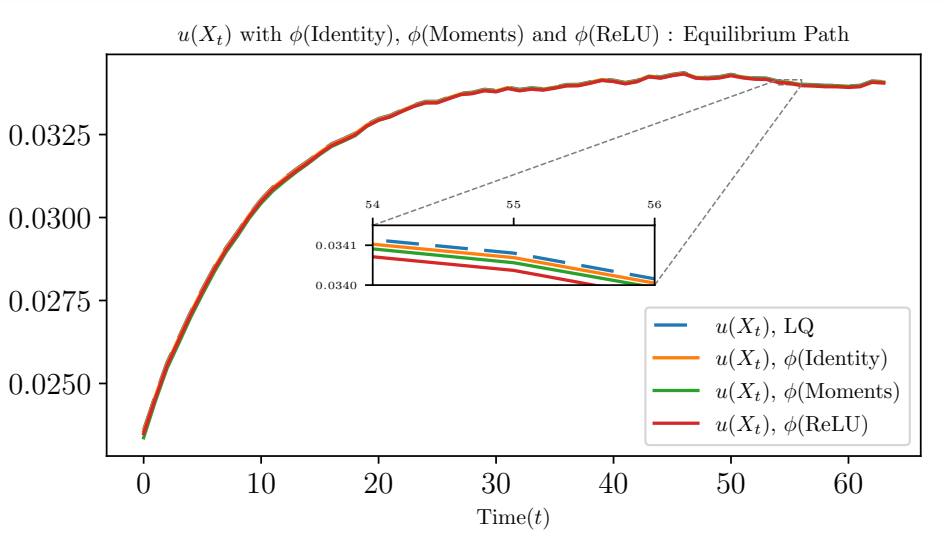


Figure 2: Comparison between baseline approximate solutions and the LQ-regulator solution for the case with $\nu = 1$ and $N = 128$.

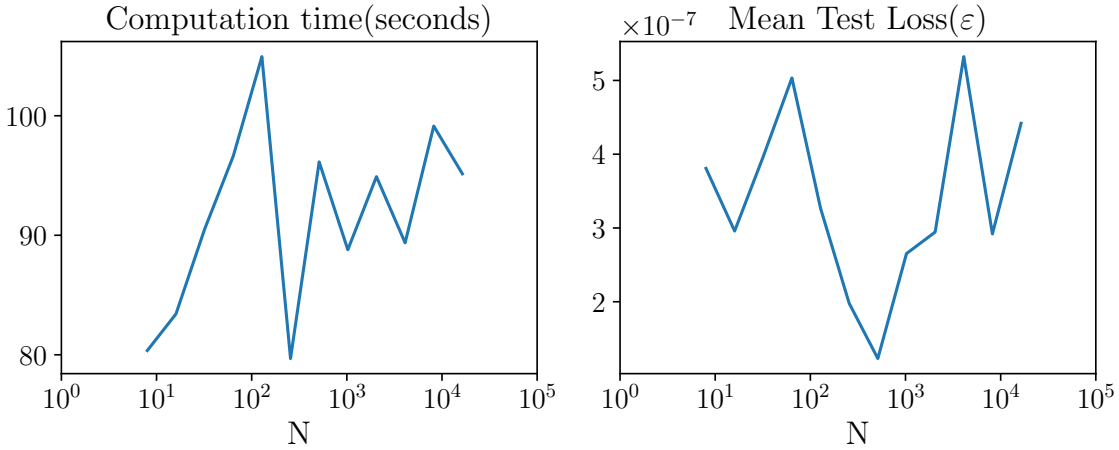


Figure 3: Performance of the ϕ(ReLU) for different N (median value of 21 trials).

Generalizability and Approximation Error

Representation with linear prices

Recall the representation,

$$u(x, X) = \rho\left(x, \frac{1}{N}\sum_{i=1}^{N}\phi(X_i)\right)$$

Can show that the following **exact solution** holds with our representation

- $\phi(X_n) = X_n$ identity, and $L = 1$
- $\rho(x, y) = \theta_1 + \theta_2 y$
- Doesn't matter how to generate X since only need 2 points!

They may let you reflect on symmetry and summary statistics, but is not really a surprise. But...

Extreme example of generalizability of neural networks

- Forgot we know any closed form, and see if overfitting hurts us.
- Fit **three** data points in $\mathbb{R}^{512}$
- Flexible functional form with 17.7 K parameters
- Now, evaluate for a whole bunch of reasonable trajectories from the initial condition and check the policy error
  - 5×10^{-5} MSE of euler, approximately **0.06%** relative error of $u(X)$

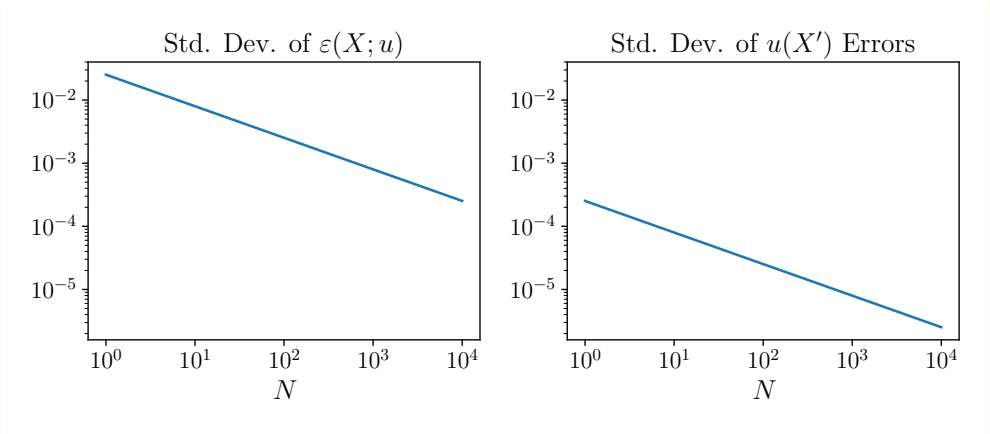
The deep-learning generalization theory to explain this is emerging.

Concentration of measure is the bless of dimensionality

$$\mathbb{E}\left[P(X') + \gamma u(X') \mid \omega\right] \approx P(\hat{X}') + \gamma u(\hat{X}'), \text{ for draw of } \hat{X}' \mid \omega$$

- Conditional expectation becomes constant as N gets large!
- Can calculate the expectation with a **single Monte-carlo draw**
  - Draw $\hat{X}'$ conditioned on ω since u, P depend "a lot" on it
- Check $P(X)$ doesn't depend too much on any $X_i \in X$
  - e.g. is expected gradient bounded in N
  - $u(\cdot)$ properties can follow from $P(\cdot)$
- Back to "continuum trick"
  - It worked because the $P(\cdot)$ and $u(\cdot)$ don't depend on any one agent outside of x (i.e. not sensitive to measure zero changes)
  - Very robust result, especially with symmetric functions

Analytic euler error due to the concentration of measure



Euler Error with with one draw $\hat{X}$ using LOM. Recall $\varepsilon \equiv -\gamma u(X) + \beta P(\hat{X}') + \gamma(1-\delta)u(\hat{X}')$

Nonlinear and Beyond

Nonlinear case with no "closed-form" solution

- With $\nu > 1$, we have a nonlinear demand function: $p(X) = 1 - \frac{1}{N}\sum_{i=1}^{N} x_i^{\nu}$.
- Notice how, now, the whole distribution of of x_i matters!
- But we can still find the solution to this nonlinear case using exactly the same functional approximation and algorithm as before.
- We do not need change anything in the code except the value of ν.
- Since the LQ solution no longer holds, we do not have an exact solution to use as a benchmark, but can check residuals.
- Same model and method. Computation time by N nearly the same to linear case

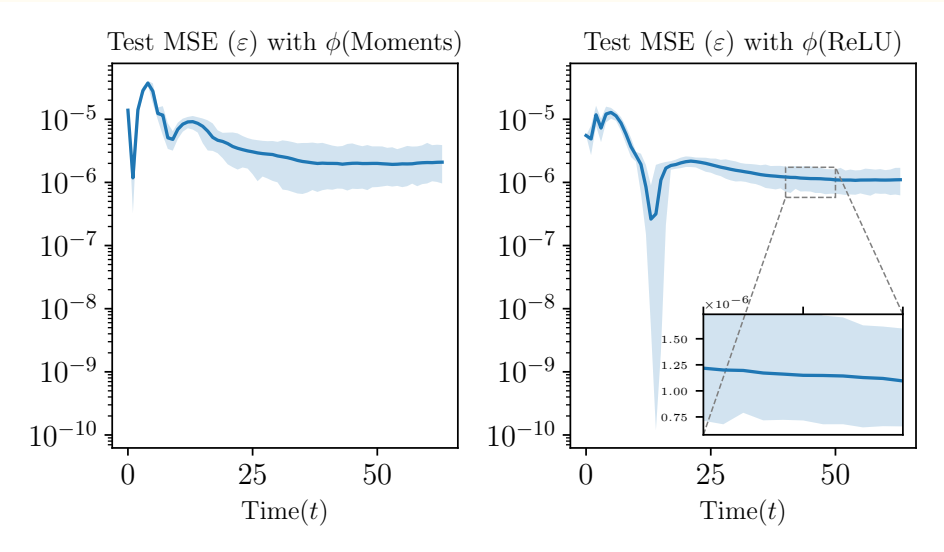


Figure 4: The Euler residuals for $\nu = 1.5$ and $N = 128$ for ϕ(Moments) and ϕ(ReLU). The dark blue curve shows the average residuals along equilibrium paths for 256 different trajectories. The shaded areas depict the 2.5th and 97.5th percentiles.

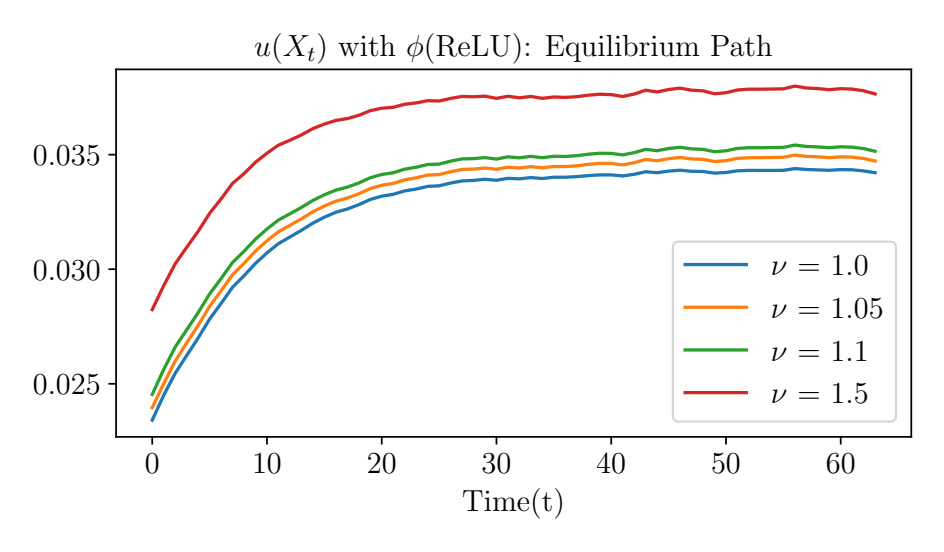


Figure 5: The optimal policy u along the equilibrium paths for $\nu = [1.0, 1.05, 1.1, 1.5]$ and $N = 128$. Each path shows the optimal policy for a single trajectory.

Extensions

1. Decreasing returns to scale: the policy becomes a function of x.
2. Multiple productivity types.
3. Complex idiosyncratic states.
4. Global solutions with transitions and aggregate shocks.
5. Many different network architectures.

Examples of models one can compute now

1. Models with rich ex-ante heterogeneity and aggregate shocks (OLG, many different household types).
2. Models of firm dynamics.
3. Open economy models with many locations.
4. Closed economy business cycle models with multisectors.
5. Network models.
6. Search and matching models.

Appendix

Deep Learning in a Nutshell

To cut through the colorful language of computer science, take a

1. neural network (i.e. a parametric **functional form** for some θ)
2. ... typically linear functions with something nonlinear as "activator" in the middle (e.g. ReLU is $\rho(x) = \max(0, x)$) and a layer might be $A_1\rho(A_2x)$ for matrices of parameters A_1 and A_2
3. ... which may be "deep" (i.e. **nested** like $f(g(X;\theta_1), h(X;\theta_2);\theta_3)$
4. ... and heavily overparameterized (i.e. often $\text{dimensions}(\theta) \gg \text{dimensions}(\{X_t\})$)
5. ... and a criteria/loss function (e.g. minimize **residuals** given θ)
6. ... and then train it (i.e. **fit the function** to simulated or actual X)
7. ... using an **optimizer** (e.g. stochastic gradient descent variants)
8. ... given some learning rate (i.e. **relaxation parameter**, often using a random subset of X each optimizer step, or "batch")
9. ... and **regularize** implicitly (e.g. double-descent) or explicitly (e.g. a L2 norm penalty)

Comparing Performance: More Different Network Designs (Linear)

group	description	Time (s)	Params (K)	Train MSE (ε)	Test MSE (ε)	Val MSE (ε)	Policy Error ($\lvert u - u_{\text{ref}} \rvert$)	Policy Error $\left(\frac{\lvert u - u_{\text{ref}} \rvert}{u_{\text{ref}}}\right)$
ϕ(Identity)	Baseline	42	49.4	4.1e-06	3.3e-07	3.3e-07	2.9e-05	0.10%
	Thin (64 nodes)	33	12.4	3.7e-06	2.7e-07	2.7e-07	3.4e-05	0.10%
	Shallow (2 layers)	159	16.6	3.7e-06	7.8e-07	7.6e-07	9.4e-03	33.53%
ϕ(Moments)	Baseline	55	49.8	1.4e-06	7.6e-07	7.6e-07	2.8e-05	0.09%
	Moments (1,2)	211	49.5	2.4e-06	1.1e-06	2.3e-06	4.4e-05	0.14%
	Very Shallow(1 layer)	241	0.6	1.1e-05	8.4e-06	7.9e-06	1.1e-02	34.00%
	Shallow (2 layers)	137	17.0	1.6e-06	9.9e-07	9.5e-07	1.8e-02	59.41%
	Deep(8 layers)	241	115.3	2.8e-06	1.2e-06	1.0e-06	5.2e-05	0.16%
	Thin (64 nodes)	82	12.6	1.6e-06	9.1e-07	9.2e-07	3.8e-05	0.12%
	Wide (256 nodes)	61	197.9	1.8e-06	8.7e-07	8.0e-07	4.3e-05	0.13%
ϕ(ReLU)	Baseline	107	66.8	3.7e-06	3.3e-07	3.3e-07	2.7e-05	0.09%
	L = 1	88	66.0	1.8e-05	2.3e-07	2.4e-07	2.8e-05	0.10%
	L = 2	86	66.3	1.3e-05	2.1e-07	2.2e-07	2.6e-05	0.08%
	L = 8	70	67.8	3.0e-05	5.9e-07	5.9e-07	3.3e-05	0.11%
	L = 16	91	69.9	5.5e-06	1.5e-07	1.5e-07	2.1e-05	0.07%
	Shallow(ϕ : 1 layer, ρ : 2 layers)	79	17.7	2.0e-06	5.5e-07	5.5e-07	3.2e-05	0.11%
	Shallow(ϕ : 1 layer)	58	50.4	8.7e-06	1.5e-07	1.5e-07	2.5e-05	0.08%
	Shallow(ρ : 2 layers)	89	34.0	3.1e-06	4.2e-07	4.2e-07	2.7e-05	0.09%

Comparing Performance: Different Networks Designs (nonlinear)

group	description	Time (s)	Params (K)	Train MSE (ε)	Test MSE (ε)	Val MSE (ε)
ϕ(Moments)	Baseline	26	49.8	6.0e-06	5.0e-06	3.8e-06
	Moments (1)	24	49.4	2.7e-05	6.5e-06	3.4e-06
	Moments (1,2)	27	49.5	8.0e-06	5.1e-06	3.6e-06
	Very Shallow (1 layer)	252	0.6	8.3e-06	1.4e+00	5.0e-06
	Shallow (2 layers)	35	17.0	5.8e-06	1.2e+00	4.4e-06
	Thin (32 nodes)	66	3.2	1.1e-05	9.7e-06	4.4e-06
ϕ(ReLU)	Baseline	60	67.1	1.4e-05	4.7e-06	3.3e-06
	L $= 1$	109	66.3	9.4e-06	1.3e-05	4.5e-06
	L $= 2$	73	66.6	1.0e-05	3.3e-06	2.3e-06
	L $= 8$	73	68.1	1.1e-05	4.9e-06	2.0e-06
	L $= 16$	72	70.2	1.5e-05	5.4e-06	1.7e-06
	Very Shallow(ϕ, ρ : 1 layer)	136	1.4	8.9e-06	4.8e+06	4.9e-06
	Shallow(ϕ, ρ : 2 layers)	47	34.3	1.0e-05	9.2e-06	2.8e-06