Deep Learning for Solving Economic Models

Jesús Fernández-Villaverde[1]

November 17, 2025

[1]University of Pennsylvania

Functional equations

- A large class of problems in economics search for a function d that solves:

$$\mathcal{T}(d) = \mathbf{0}$$

- More formally:

  1. Let J^1 and J^2 be two functional spaces and let $\mathcal{T} : J^1 \to J^2$ be an operator between these two spaces.
  2. Let $\Omega \subseteq \mathbb{R}^l$.
  3. Then, we need to find a function $d : \Omega \to \mathbb{R}^m$ such that $\mathcal{T}(d) = \mathbf{0}$.

- Points to remember:

  1. Regular equations are particular examples of functional equations.
  2. $\mathbf{0}$ denotes the zero element of the space.
  3. This formalism deals both with market equilibrium and social planner problems.

Example I: Decision rules

- Take the basic stochastic neoclassical growth model:

$$\max \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t)$$
$$c_t + k_{t+1} = e^{z_t} k_t^{\alpha} + (1-\delta) k_t, \ \forall \ t > 0$$
$$z_t = \rho z_{t-1} + \sigma \varepsilon_t, \ \varepsilon_t \sim \mathcal{N}(0,1)$$

- The first-order condition:

$$u'(c_t) = \beta \mathbb{E}_t \left\{ u'(c_{t+1}) \left(1 + \alpha e^{z_{t+1}} k_{t+1}^{\alpha-1} - \delta\right) \right\}$$

Example I: Decision rules

- There is a decision rule (a.k.a. policy function) that gives the optimal choice of consumption and capital tomorrow given the states today:

$$d = \begin{cases} d^1(k_t, z_t) = c_t \\ d^2(k_t, z_t) = k_{t+1} \end{cases}$$

- Then:

$$\mathcal{T} = u'\left(d^1(k_t, z_t)\right)$$
$$-\beta \mathbb{E}_t \left\{ u'\left(d^1\left(d^2(k_t, z_t), z_{t+1}\right)\right) \left(1 + \alpha e^{z_{t+1}} \left(d^2(k_t, z_t)\right)^{\alpha-1} - \delta\right) \right\} = 0$$

- If we find d, and a transversality condition is satisfied, we are done!

Example II: Conditional expectations

- Let us go back to our Euler equation:

$$u'(c_t) - \beta \mathbb{E}_t \left\{ u'(c_{t+1}) \left(1 + \alpha e^{z_{t+1}} k_{t+1}^{\alpha-1} - \delta\right) \right\} = 0$$

- Define now:

$$d = \begin{cases} d^1(k_t, z_t) = c_t \\ d^2(k_t, z_t) = \mathbb{E}_t \left\{ u'(c_{t+1}) \left(1 + \alpha e^{z_{t+1}} k_{t+1}^{\alpha-1} - \delta\right) \right\} \end{cases}$$

- Why? Think about a model with a ZLB.

- Then:

$$\mathcal{T}(d) = u'\left(d^1(k_t, z_t)\right) - \beta d^2(k_t, z_t) = 0$$

Example III: Value functions

- There is a recursive problem associated with the previous sequential problem:

$$V(k_t, z_t) = \max_{k_{t+1}} \{u(c_t) + \beta \mathbb{E}_t V(k_{t+1}, z_{t+1})\}$$

$$c_t = e^{z_t} k_t^{\alpha} + (1-\delta) k_t - k_{t+1}, \ \forall \ t > 0$$

$$z_t = \rho z_{t-1} + \sigma \varepsilon_t, \ \varepsilon_t \sim \mathcal{N}(0,1)$$

- Then:

$$d(k_t, z_t) = V(k_t, z_t)$$

and

$$\mathcal{T}(d) = d(k_t, z_t) - \max_{k_{t+1}} \{u(c_t) + \beta \mathbb{E}_t d(k_{t+1}, z_{t+1})\} = 0$$

How do we solve functional equations?

- General idea: substitute $d(x)$ by $d^n(x, \theta)$ where θ is an $n-$dim vector of coefficients to be determined.

- Two main approaches:

  1. Perturbation methods:

  $$d^n(x, \theta) = \sum_{i=0}^{n} \theta_i (x - x_0)^i$$

  We use implicit-function theorems to find θ_i.

  2. Projection methods:

  $$d^n(x, \theta) = \sum_{i=0}^{n} \theta_i \Psi_i(x)$$

  We pick a basis $\{\Psi_i(x)\}_{i=0}^{\infty}$ and "project" $\mathcal{T}$ against that basis.

All traditional solution methods are variations of these two ideas

- Linearization (or loglinearization): equivalent to a first-order perturbation.
- Linear-quadratic approximation to the utility function: equivalent (under certain conditions) to a first-order perturbation.
- Parameterized expectations: a particular example of projection.
- Value function iteration: it can be interpreted as an iterative procedure to solve a particular projection method. Nevertheless, I prefer to think about it as a different family of problems.
- Policy function iteration: similar to VFI.

Neural networks

A different approximation: A neural network

- A neural network is an approximation to $d(x)$ of the form:

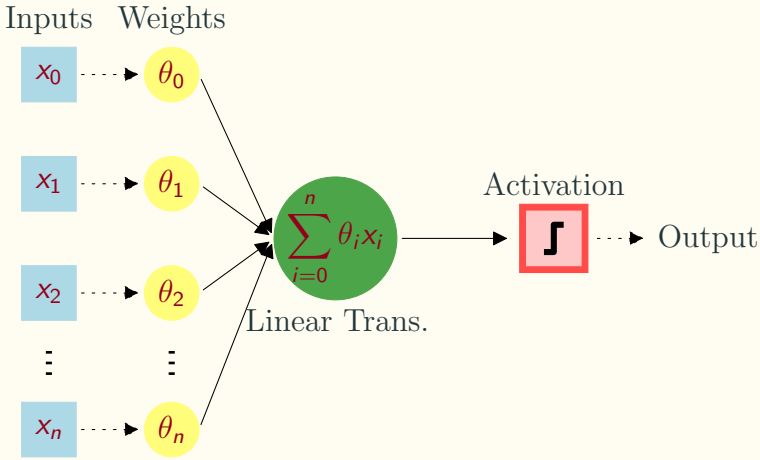
$$y = d(x) \cong d^{NN}(x;\theta) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi(z_m)$$

  where $\phi(\cdot)$ is an arbitrary activation function and:

$$z_m = \sum_{n=0}^{N} \theta_{n,m} x_n$$

- The x_n's are known as the features of the data, which belong to a feature space $\mathcal{X}$.
- The $\phi(z_m)$'s are known as the representations of the data.
- M is known as the width of the model (wide vs. thin networks).
- "Training" the network: We select θ such that $d^{NN}(x;\theta)$ is as close to $d(x)$ as possible given some relevant metric (e.g., the ℓ_2 norm).

Flow representation



Comparison with previous approximations

- Compare:

$$d(x) \cong d^{NN}(x;\theta) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi \left(\sum_{n=0}^{N} \theta_{n,m} x_n \right)$$

  with a standard projection:

$$d(x) \cong d^{CP}(x;\theta) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x)$$

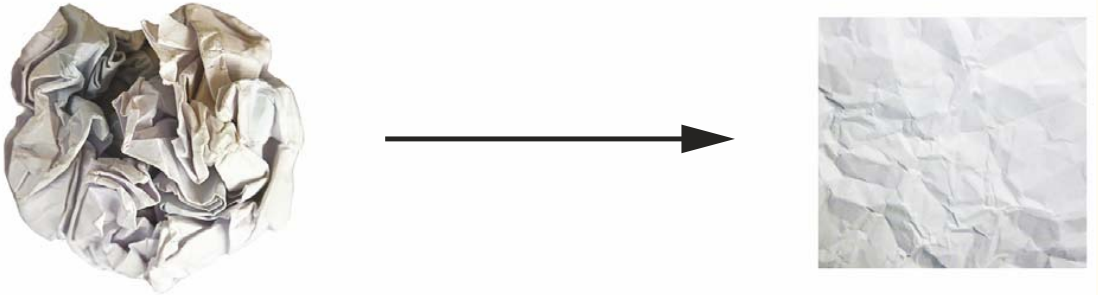
  where ϕ_m is, for example, a Chebyshev polynomial.

- We trade rich coefficient parameterizations for parsimonious basis functions.

Why do neural networks "work"?

- Neural networks consist entirely of chains of tensor operations: we take x, we perform affine transformations, and apply an activation function.
- Thus, these tensor operations are geometric transformations of x. In fact, a better name for neural networks could be *chained geometric transformations*.
- In other words: neural networks look for convenient geometrical representations of high-dimensional manifolds.
- The success of any functional approximation problem is to search for the right geometric space in which to perform it, not to search for a "better" basis function.
- Think about:

$$y = k^{\alpha} l^{1-\alpha} \Rightarrow \log y = \alpha \log k + (1-\alpha) \log l$$



Deep learning, I

- A deep learning network is an acyclic *multilayer* composition of $J > 1$ neural networks:

$$z_m^0 = \theta_{0,m}^0 + \sum_{n=1}^{N} \theta_{n,m}^0 x_n$$

and

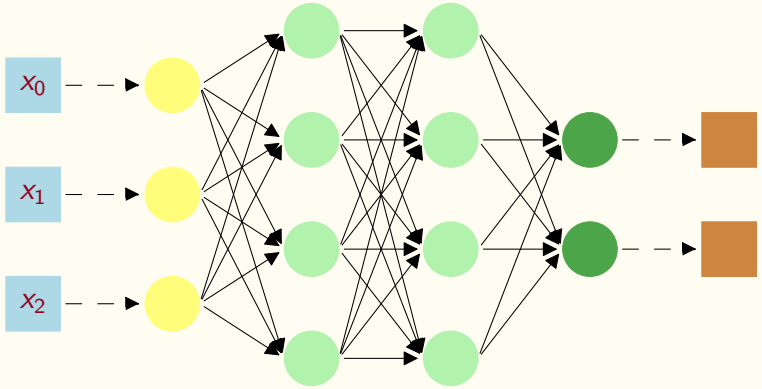
$$z_m^1 = \theta_{0,m}^1 + \sum_{m=1}^{M^{(1)}} \theta_m^1 \phi^1 \left(z_m^0\right)$$

$$\ldots$$

$$y \cong d^{DL}(x;\theta) = \theta_0^J + \sum_{m=1}^{M^{(J)}} \theta_m^J \phi^J \left(z_m^{J-1}\right)$$

where the $M^{(1)}, M^{(2)}, \ldots$ and $\phi^1(\cdot), \phi^2(\cdot), \ldots$ are possibly different across each layer of the network.

- A deep network creates new representations by composing older representations.



Input Values Hidden Layer 1 Output Layer

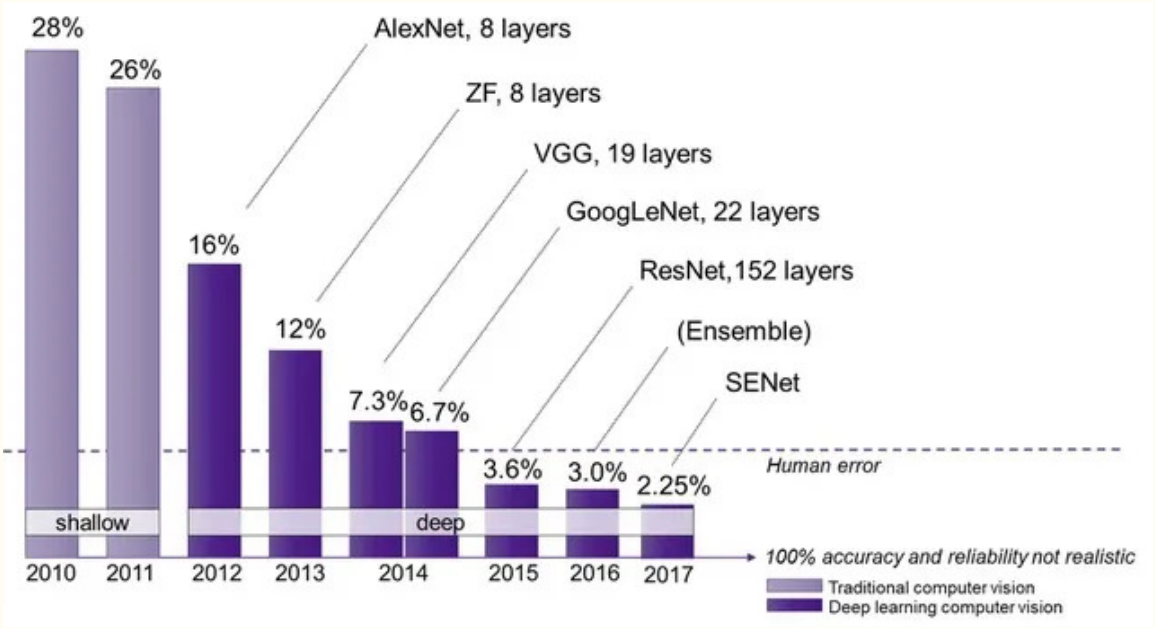
Input Layer Hidden Layer 2

Deep learning, II

- Also known as deep feedforward neural networks or, of fully connected, multilayer perceptrons.
- "Feedforward" comes from the fact that the composition of neural networks can be represented as a directed acyclic graph, which lacks feedback. We can have more general recurrent structures.
- J is known as the depth of the network (deep vs. shallow networks).
- The case $J = 1$ is a standard neural network.
- As before, we can select θ such that $d^{DL}(x; \theta)$ approximates a target function $d(x)$ as closely as possible under some relevant metric.
- All other aspects (selecting $\phi(\cdot)$, J, M, ...) are known as the network architecture.

Why do deep neural networks “work” better?

- Why do we want to introduce hidden layers?
  1. It works! Evolution of ImageNet winners.
  2. The number of representations increases exponentially with the number of hidden layers, while computational cost grows linearly.
  3. Intuition: hidden layers induce highly nonlinear behavior in the joint creation of representations without the need to have domain knowledge (used, in other algorithms, in some form of greedy pre-processing).


28%
26%
16%
12%
7.3%
6.7%
3.6%
3.0%
2.25%
AlexNet, 8 layers
ZF, 8 layers
VGG, 19 layers
GoogLeNet, 22 layers
ResNet,152 layers
(Ensemble)
SENet
Human error
shallow
deep
2010
2011
2012
2013
2014
2015
2016
2017
100% accuracy and reliability not realistic
Traditional computer vision
Deep learning computer vision

Some consequences

- Because of the previous arguments, neural networks can efficiently approximate extremely complex functions.
- In particular, under certain (relatively weak) conditions:
  1. Neural networks are universal approximators.
  2. Neural networks break the "curse of dimensionality."
- Furthermore, neural networks are easy to code, stable, and scalable for multiprocessing (neural networks are built around tensors).

Further advantages

- Neural networks and deep learning often require less "inside knowledge" by experts on the area.
- While results can be highly counter-intuitive, deep neural networks deliver excellent performance.
- Outstanding open source libraries (`Tensorflow`, `Keras`, `Pytorch`, `JAX`) that integrate well with easy scripting languages (`Python`).
- Newer algorithms: batch normalization, residual connections, and depthwise separable convolutions.
- More recently, development of dedicated hardware (TPUs, AI accelerators, FPGAs) is likely to maintain a hedge for the area.
- The richness of an ecosystem is key to its long-run success.

Examples of code

1. An LQ optimal control problem:

   https://github.com/Mekahou/Fun-Stuff/blob/main/codes/linear%20quadratic%20DP%20DNN/3.%20LQ_DP_DNN_Training_Main.ipynb

2. A Neoclassical growth model (discrete time):

   https://colab.research.google.com/drive/1jbSti3LkxASZg04Bkod0EBJFXdf6xu9_?usp=sharing

3. A Neoclassical growth model (continuous time):

   https://colab.research.google.com/drive/1rFfqbJYn26_nm-CS21Fbw_QV7ccM8Xl1T?usp=sharing

4. An OLG model:

   https://github.com/sischei/DeepEquilibriumNets

5. A Krusell-Smith and a financial frictions HA code:

   https://github.com/jesusfv/financial-frictions

Models with a representative agent

The stochastic neoclassical growth model

- A representative household:

$$\max \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \log(c)$$

$$c_t + k_{t+1} = w_t + (1 + r_t - \delta)\, k_t, \; \forall \; t > 0$$

$$\lim_{T \to \infty} \mathbb{E} \beta^T k_{T+1} c_T^{-1} k_0, z_0 = 0$$

- A representative firm:

$$y_t = (e^{z_t})^{1-\alpha} k_t^{\alpha}$$

$$z_t = \rho z_{t-1} + \sigma \nu_t, \quad \nu_t \sim \mathcal{N}(0, \sigma)$$

  Solving the model with $\rho < 1$ is actually harder!

- Input markets are competitive and aggregate resource constraint holds: $y_t = c_t + k_{t+1} - (1 - \delta)\, k_t$.

The equilibrium Markov process

- With recursive notation, the states of the model are $X \equiv \begin{bmatrix} k & z \end{bmatrix}^\top \in \mathbb{R}^2_+$.
- Euler equation as a functional equation on $k'(k,z)$:

$$1 = \mathbb{E}\left[\beta \frac{c(k,z)}{c(k'(k,z),z')}\left(1-\delta+\alpha(e^{z'})^{1-\alpha}k'(k,z)^{\alpha-1}\right)\right]$$

  where $c(k,z) \equiv (e^z)^{1-\alpha}k^\alpha + (1-\delta)k - k'(k,z)$.
- Thus, conditioning on a policy:

$$X' = \Phi(X,z;k') = \begin{bmatrix} k'(k,z) \\ \rho z + \sigma\nu \end{bmatrix}$$

A deep learning approximation

- We approximate $k'(k,z)$ with a deep neural network $k'_\theta(k,z)$ that belongs to the architecture $\mathcal{H}(\theta)$.
- Euler equation error:

$$\mathscr{L}(k'_\theta, X) = \left[1 - \sum_{i=1}^{M} \omega_i \left[\beta \frac{c(k,z)}{c(k'_\theta, z'(z,\nu_i))} \left(1 - \delta + \alpha(z'(z,\nu_i))^{1-\alpha}(k'_\theta)^{\alpha-1}\right)\right]\right]^2$$

  where $z'(z,\nu_i) = \rho \log z + \sigma\nu_i$.

- Goal:

$$k'^* = \underset{k'_\theta \in \mathcal{H}(\theta)}{\arg\min}\, \mathbb{E}_{X \sim \mu^*(X_0, T_{\text{pop}}; k'^*)} \left[\mathscr{L}(k'_\theta, X)\right]$$

  where is the $\mu^*(X_0, T_{\text{pop}}; k'^*)$ population distribution.

Solving the equilibrium loop

- If we knew $\mu^*(\cdot)$, a trivial algorithm to solve the problem above would be to sample $\mathcal{D}$ points from $\mu^*(\cdot)$ and solve the empirical risk:

$$\theta^* = \arg\min_{\theta \in \Theta} \frac{1}{|\mathcal{D}|} \sum_{X \in \mathcal{D}} \mathscr{L}(k'_\theta, X).$$

- The challenge, of course, is that we do not know $\mu^*(\cdot)$ precisely because of the equilibrium feedback loop.

- This is the primary difference between the applications of deep learning in engineering/natural sciences, and economics.

The idea

- Sample from a misspecified population distribution and periodically regenerate those samples as the policy function is refined.
- That is, given a k'_i, we solve:

$$k'_{i+1} \equiv \arg \min_{k'_\theta \in \mathcal{H}(\theta)} \mathbb{E}_{X \sim \mu^*(X_0, T_{\text{pop}}; k'_i)} \left[\mathscr{L}(k'_\theta, X) \right],$$

  until convergence.
- In our case, we start with a $k'_1(\cdot)$ policy from the Solow model with constant savings rate s_{ss}, $k'_1(k, z) = s_{ss} z^{1-\alpha} k^\alpha + (1 - \delta)k$.

An iterative algorithm

1. Solve the empirical risk minimization problem given the current $\mathcal{D}_i$,

$$\theta_{i+1} = \arg\min_{\theta \in \theta} \frac{1}{|\mathcal{D}_i|} \sum_{X \in \mathcal{D}_i} \mathscr{L}(k'_\theta, X).$$

2. Generate a new $\mathcal{D}_{i+1}$ using samples from $\mu^*(X_0, T_{\text{pop}}, k'_{\theta_{i+1}})$.

3. Iterate until the empirical risk is below a threshold.

Parameters, hyperparameters, and optimizer

- Parameter values: $\beta = 0.9$, $\alpha = 1/3$, $\delta = 0.2$, $\rho = 0.9$, and $\sigma = 0.025$.
- $M = 7$ nodes for quadrature.
- A deep neural network with four layers, each with 128 nodes, a ReLU activation function $\max\{0, x\}$, and a softplus final layer, $\log(1 + e^x)$. This leads to approximately $50K$ parameters in θ.
- $T_{\text{pop}} = 20$ and generate trajectories (k_t, z_t) for $t = 0, \ldots, T_{\text{pop}}$.
- $X_0 \sim \mathcal{N}(\begin{bmatrix} k_0 & z_0 \end{bmatrix}^\top, \Sigma)$ with $z_0 = 1, k_0 = 0.8 \times k_{ss}$, and $\Sigma \equiv \begin{bmatrix} 0.008^2 & 0 \\ 0 & 0.02^2 \end{bmatrix}$.
- 100 iterations of the L-BFGS optimizer and regenerate the $\mathcal{D}_i$ every five iterations.
- The algorithm, coded in `Jax`, runs in approximately 1.2 seconds using only the CPU of a laptop.

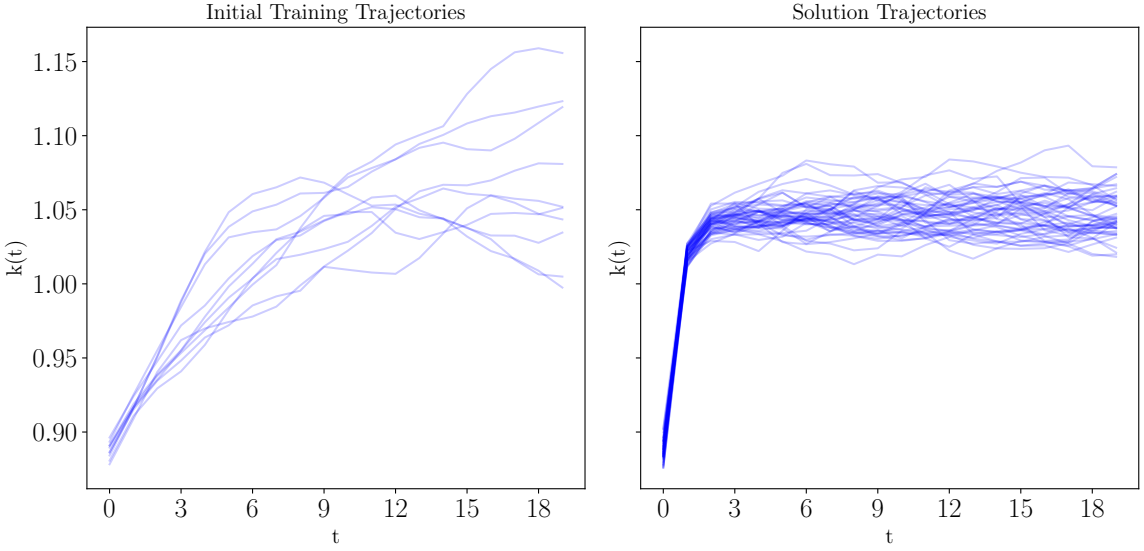


Figure 1: Initial vs. final trajectories of capital.

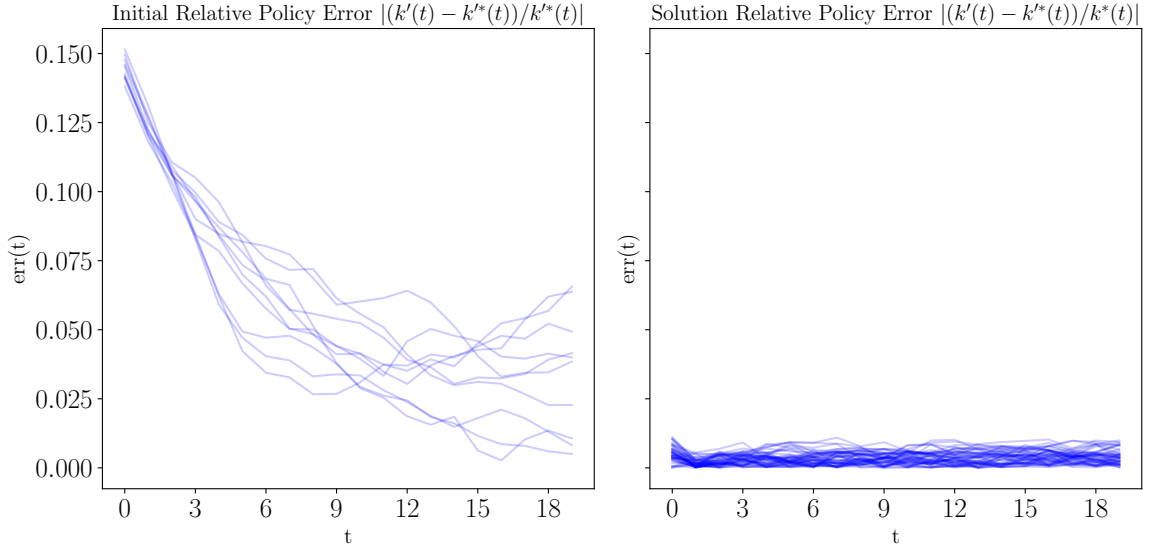


Figure 2: Initial vs. final policy errors

Models with heterogeneous agents

The challenge

- To compute and take to the data models with heterogeneous agents, we need to deal with:

  1. The distribution of agents G_t.
  2. The operator $H(\cdot)$ that characterizes how G_t evolves:

     $$G_{t+1} = H(G_t, S_t)$$

     or

     $$\frac{\partial G_t}{\partial t} = H(G_t, S_t)$$

     given the other aggregate states of the economy S_t.

- How do we track G_t and compute $H(G_t, S_t)$?

A common approach

- If we are dealing with N discrete types, we keep track of $N-1$ weights.
- If we are dealing with continuous types, we extract a finite number of features from G_t:
  1. Moments.
  2. Q-quantiles.
  3. Weights in a mixture of normals...
- We stack either the weights or features of the distribution in a vector μ_t.
- We assume μ_t follows the operator $h(\mu_t, S_t)$ instead of $H(G_t, S_t)$.
- We parametrize $h(\mu_t, S_t)$ as $h^j(\mu_t, S_t; \theta)$.
- We determine the unknown coefficients θ such that an economy where μ_t follows $h^j(\mu_t, S_t; \theta)$ replicates as well as possible the behavior an economy where G_t follows $H(\cdot)$.

Example: Basic Krusell-Smith model

- Two aggregate variables: aggregate productivity shock and household distribution $G_t(a, z)$ where:

$$\int G_t(a, z)da = K_t$$

- We summarize $G_t(a, \cdot)$ with the log of its mean: $\mu_t = \log K_t$ (extending to higher moments is simple, but tedious).

- We parametrize $\underbrace{\log K_{t+1}}_{\mu_{t+1}} = \underbrace{\theta_0(s_t) + \theta_1(s_t) \log K_t}_{h^j(\mu_t, s_t; \theta)}$.

- We determine $\{\theta_0(s_t), \theta_1(s_t)\}$ by OLS run on a simulation.

Problems

- There is limited guidance regarding feature and parameter selection in general cases.
  - Yes, keeping track of the log of the mean and a linear functional form works well for the basic model. But what about an arbitrary model?
  - Method suffers from "curse of dimensionality": difficult to implement with many state variables or high N/higher moments.
  - Lack of theoretical foundations (Does it converge? Under which metric?).

How can deep learning help?

- Deep learning addresses challenges:
  1. How to extract features from an infinite-dimensional object efficiently.
  2. How to parametrize the non-linear operator mapping how distributions evolve.
  3. How to tackle the "curse of dimensionality."
- Given time limitations, I will discuss the last two points today.
- In our notation of $y = f(x)$:
  1. $y = \mu_{t+1}$.
  2. $x = (\mu_t, S_t)$.