Taming the Curse of Dimensionality: Old Ideas and New Strategies

Jesús Fernández-Villaverde[1]

June 30, 2023

[1]University of Pennsylvania

Motivation

- Many interesting questions in economics require:

  1. Nonlinear techniques. Examples: How do financial crises arise? Why do countries or firms default? When do firms invest in large, lumpy projects? Why do individuals decide to migrate?

  2. Heterogeneous agents. Examples: What mechanisms account for changes in income and wealth inequality? Is there a trade-off between inequality and economic growth? How does inequality affect monetary and fiscal policy? What are the consequences of entry-exit in models of industry dynamics?

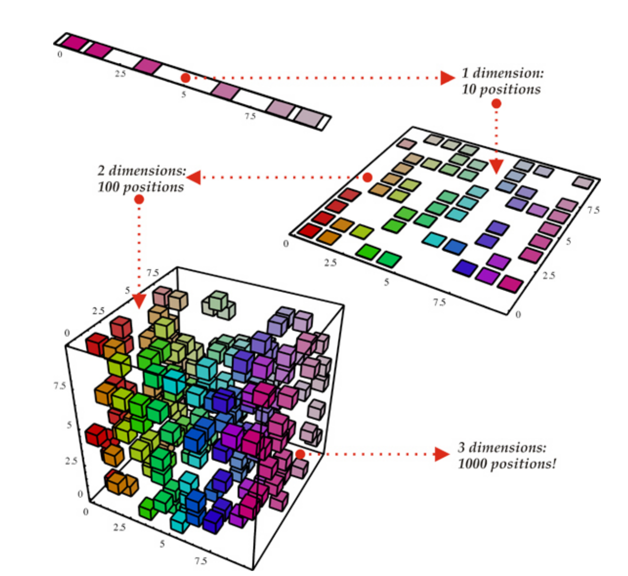
  3. Many state variables. Examples: Discrete node models, corporate finance models, rich life-cycle models, models where parameters are quasi-states.

- Often, all three elements come together. Examples: models of climate change with geographical granularity, heterogeneous agents models with nominal frictions and many assets.

The challenge

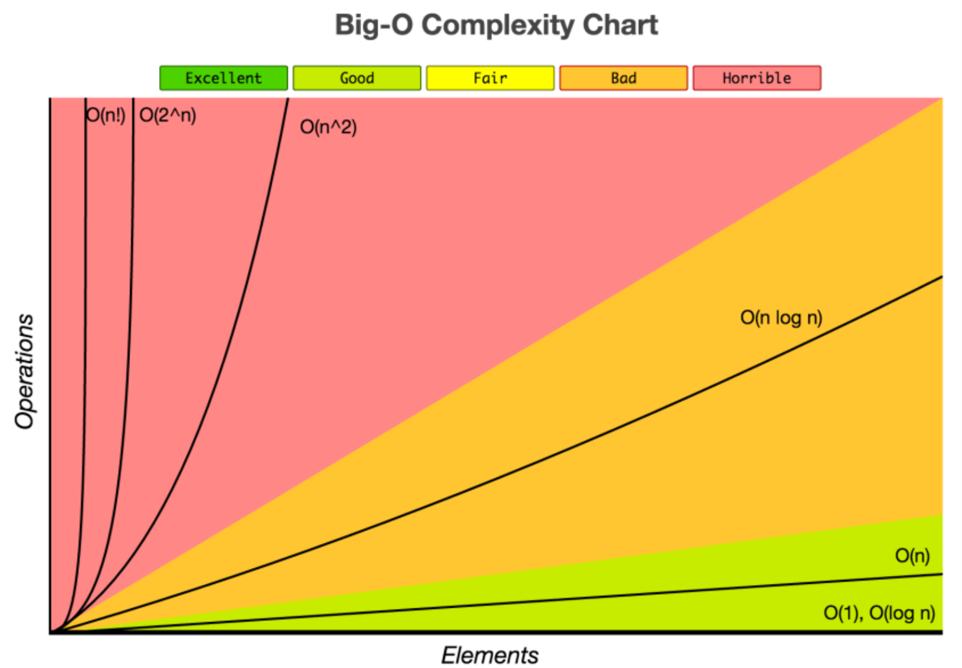
- Modeling this class of problems rarely leads to analytic solutions.
- Thus, we must resort to numerical techniques.
- We want accurate and fast solution methods that can handle these models (solution and estimation).
- Fast includes both coding and running time.
- While classical methods (value function iteration, Metropolis-Hastings) can tackle, in theory, most problems, we would need to struggle with the "curse of dimensionality."

Too many dimensions



Our goal

- Thus, we need to find ways to control the “curse of dimensionality”.
- In particular, we want to move to the “feasible” region of the Big-O complexity chart.
- This is relevant both for time and memory complexity.
- But key, as well, in terms of coding time. In practice, given modern computational resources, this is the real constraint for researchers.


Big-O Complexity Chart
Excellent
Good
Fair
Bad
Horrible
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(1), O(log n)
Operations
Elements

Taming the "curse of dimensionality"

- Three strategies:

  1. Better numerical algorithms (e.g., deep learning, continuous-time methods).
  2. Better software implementations (e.g., differentiable and functional programming, flexible data structures, advances in massive parallelization).
  3. Better hardware designs (e.g., GPUs, AI accelerators, FPGAs, quantum hardware).

- Some of these techniques are relatively new in economics or, at least, less familiar to researchers.
- A complete treatment of the material would require, at the very least, a whole semester.
- Check `www.sas.upenn.edu/~jesusfv`.
- In this talk, I will briefly introduce some of these ideas.

Better numerical algorithms

New methods

- Deep learning:
  - Financial Frictions and the Wealth Distribution, with Galo Nuño and Samuel Hurtado.
  - Ricardian Business Cycles, with Lorenzo Bretscher and Simon Scheidegger.
  - Spooky Boundaries at a Distance: Exploring Transversality and Stability with Deep Learning, with Mahdi Ebrahimi Kahou, Sebastián Gómez-Cardona, Jesse Perla and Jan Rosa.
  - Exploiting Symmetry in High-Dimensional Dynamic Programming, with Mahdi Ebrahimi Kahou, Jesse Perla, and Arnav Sood.
  - Inequality and the Zero Lower Bound, with Joël Marbet, Galo Nuño, and Omar Rachedi.
  - Solving High-Dimensional Dynamic Programming Problems, with Artem Kuriksha and Galo Nuño.
  - Structural Estimation of Dynamic Equilibrium Models with Unstructured Data, with Sara Casella, Stephen Hansen, and Minchul Shin.
- Continuous-time methods:
  - Financial Frictions and the Wealth Distribution, with Galo Nuño and Samuel Hurtado.

The problem

- We want to approximate ("learn") an unknown function:

$$y = f(\mathbf{x})$$

  where y is a scalar and $\mathbf{x} = \{x_0 = 1, x_1, x_2, ..., x_N\}$ a vector (including a constant).

- We care about the case when N is large (possibly in the thousands!).
- Easy to extend to the case where y is a vector (e.g., a probability distribution), but notation becomes cumbersome.
- In economics, $f(\mathbf{x})$ can be a value function, a policy function, a pricing kernel, a conditional expectation, a classifier, ...

A neural network

- An artificial neural network is a approximation to $f(\mathbf{x})$ of the form:

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

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

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

- The z_m's are known as the representations of the data.
- "Training" the network: We select θ such that $g^{NN}(\mathbf{x};\theta)$ is as close to $f(\mathbf{x})$ as possible given some relevant metric (e.g., the ℓ_2 norm).

Deep learning

- 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

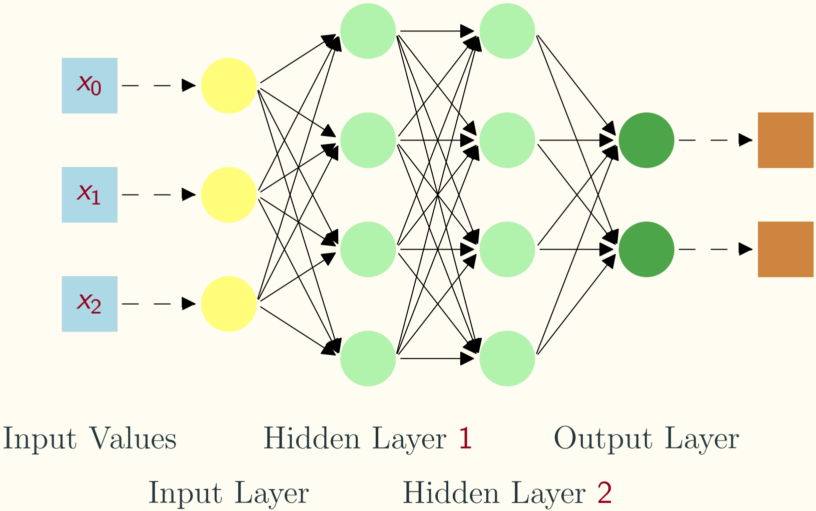
$$z_m^1 = \theta_{0,m}^1 + \sum_{m=1}^{M^{(1)}} \theta_m^1 \phi^1 \left(z_m^0 \right)$$

$$\dots$$

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

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

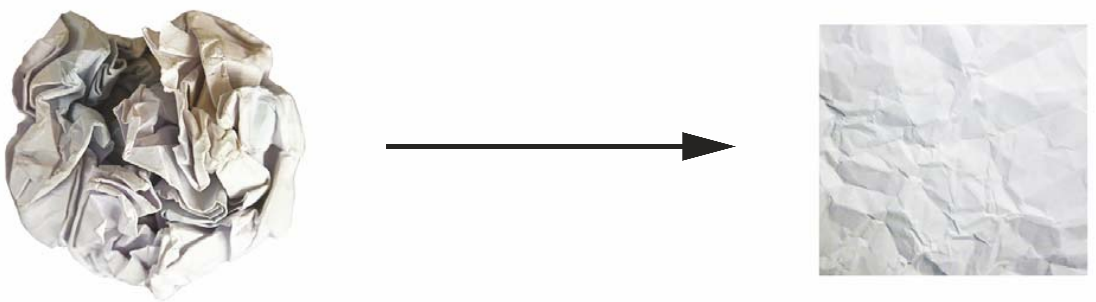
- A deep network creates new representations by composing older representations.


x_0
x_1
x_2
Input Values
Hidden Layer 1
Output Layer
Input Layer
Hidden Layer 2

Why do neural networks "work"?

- Neural networks consist entirely of chains of tensor operations: we take $\mathbf{x}$, perform affine transformations, and apply an activation function.
- Thus, these tensor operations are geometric transformations of $\mathbf{x}$.
- In other words: a neural network is a complex geometric transformation in a high-dimensional space.
- Deep 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:

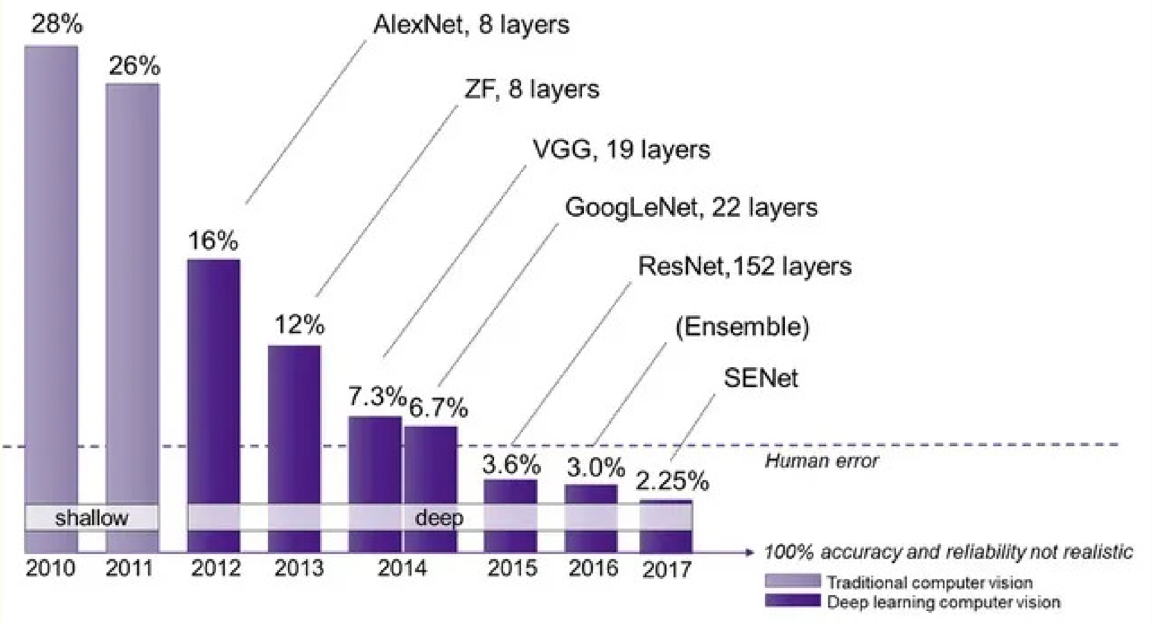
$$y = k^{\alpha} l^{1-\alpha} \Rightarrow \log y = \alpha \log k + (1-\alpha) \log l$$



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).
- The richness of an ecosystem is key to its long-run success.

TensorFlow

PyTorch

flux

Why continuous time? I

- Long and illustrious tradition in finance: classical results by Merton and others.

- However, less used in macroeconomics (except in growth and neoclassical investment theories).

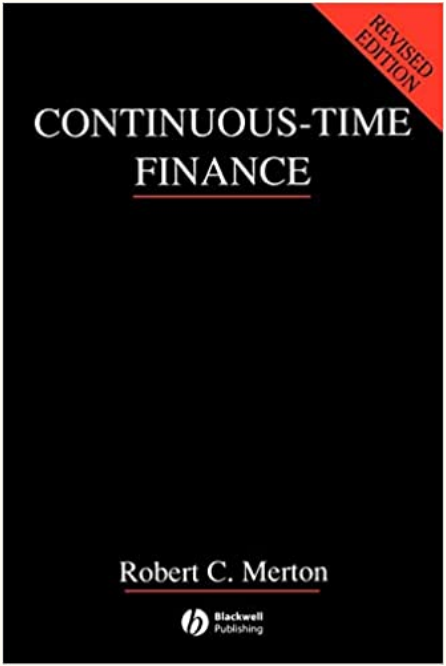
- Why?

  1. Economic data comes in discrete intervals: most time-series are in discrete time.

  2. Arrival of dynamic programming in the early 1970s.

  3. Stochastic calculus has some entry cost (notice: in growth theory, you can often skip stochastic calculus because you deal with deterministic models).

- Recent "boom" of continuous-time methods in business cycle research and related areas: Stokey (2009), Brunnermeier and Sannikov (2014), Ahn et al. (2017), ...


REVISED EDITION
CONTINUOUS-TIME FINANCE
Robert C. Merton
Blackwell Publishing

Why continuous time? II

- Itô's Lemma allows us to substitute the integrals of discrete time for derivatives in continuous time.

- Bellman equation:

$$V(x) = \max_{\alpha} \left\{ u(\alpha, x) + \beta \int V(x') \, p(dx|\alpha, x) \right\}$$

  vs. Hamilton-Jacobi-Bellman equation:

$$\rho V(x) = \max_{\alpha} \left\{ u(\alpha, x) + \sum_{n=1}^{N} \mu_n(x, \alpha) \frac{\partial V}{\partial x_n} + \frac{1}{2} \sum_{n_1, n_2 = 1}^{N} \left(\sigma^2(x, \alpha) \right)_{n_1, n_2} \frac{\partial^2 V}{\partial x_{n_1} \partial x_{n_2}} \right\}$$

- Why is this so important? Integrals depend on typical sets, and typical sets are hard to characterize (I will return to this point momentarily).

Why continuous time? III

- A few other mathematical advantages:
    1. Elegant and powerful math.
    2. Sparsity of transitions matrices.
    3. Easier to write complex FOCs, ...
- Related: much more work on PDEs than on stochastic difference equations.
- However: there are many occasions where discrete-time methods are still quite useful.

Better coding

Better coding

- Differentiable State-Space Models and Hamiltonian Monte Carlo Estimation, with David Childers, Jesse Perla, Christopher Rackauckas, and Peifan Wu.
- Functional Programming in Economics, with Jan Žemlička.

Differentiable programming

- Differentiable programming is one of the top research areas in computer science right now.
- This is the programming approach used by `ChatGPT`.
- Idea: write code that can be easily differentiated. How?
- Think about any program as a compositional function that maps inputs to outputs by composing functions along directed acyclical graph (DAG).
- Derivative computed by accumulating derivatives of node functions along a DAG using AD.

High-dimensional geometry

- Expectation values are given by accumulating the integrand over a volume.
- In regular models, posterior density decays exponentially with distance from mode: there is not much volume at the mode!
- Simple example: think about tossing a coin 1000 times, with $p(H) = 0.500000001$.
  1. $\{H, H, ..., H\}$ is the most likely event.
  2. And yet, most events will have around 500 heads!
- In high D, volumes concentrates in thin shell $O(\sqrt{D})$ away from mode: typical set (this a manifestation of concentration of measure).
- That means that:
  1. If you use quadrature, you waste most of your quadrature points.
  2. If you use Metropolis-Hastings, you must take small steps to stay on the typical set.

Hamiltonian Monte Carlo

- Gradient information enables improved samplers $\Rightarrow$ Hamiltonian Monte Carlo (HMC):
  - We add a momentum vector that induces a kinetic energy term (i.e., Hamiltonian dynamics).
  - We direct sampling towards the typical set, and we can explore high-dimensional space efficiently.
- But, how do we (efficiently) find the required gradients of the likelihood of the model?
- Numerical or symbolic derivatives cannot handle this task.
- Automatic differentiation (AD) gets you part of the way there.
- But default implementations of AD (e.g., `Stan`) are unusable. Think about the QZ algorithm complex-valued, eigenvalue sort only almost surely pointwise differentiable.

Automatic differentiation and the cheap gradient principle

- We apply reverse mode AD within and between blocks by relying on a large library of primitives.
- Recall:
  - *Forward mode AD*: accumulate from inputs to outputs.
  - *Reverse mode AD*: pass along sensitivities ("adjoints") from outputs to inputs.
- *Cheap gradient principle*: Reverse mode AD computes gradients in $O(1)$ time:
  - Gradients same order of cost as function evaluation.
  - Gradient-based algorithms (e.g., HMC) as cheap per iterate as 0th order (e.g., RWMH).

Functional programming

- Nearly as old as imperative programming.
- Created by John McCarthy with `LISP` (list processing) in the late 1950s.
- Inspired by Alonzo Church's λ-calculus from the 1930s.
- Minimal construction of "abstractions" (functions) and substitutions (applications).
- Lambda Calculus is Turing Complete: we can write a solution to any problem a computer can solve.

Why functional programming?

- Recent revival of interest.
- Often functional programs are:
  1. Easier to read.
  2. Easier to debug and maintain.
  3. Easier to parallelize.
- Useful features:
  1. Hindley–Milner type system.
  2. Lazy evaluation.
  3. Closures.

Main idea

- All computations are implemented through functions: functions are first-class citizens.
- Main building blocks:
    1. Immutability: no variables get changed (no side effects). In some sense, there are no variables.
    2. Recursions.
    3. Curried functions.
    4. Higher-order functions: compositions ($\simeq$operators in functional analysis).

Interactions

- How do we interact then?

  1. Pure functional languages (like `Haskell`): only limited side changes allowed (for example, I/O) and tightly enforced to prevent leakage.
  2. Impure functional languages (like `OCalm` or `F#`): side changes allowed at the discretion of the programmer.

- Loops get substituted by recursion.
- We can implement many insights from functional programming even in standard languages such as `C++` or`Python`.

Better hardware

Better hardware

- Practical Guide to Parallelization in Economics with David Zarruk Valencia.
- Tapping the Supercomputer under your Desk: Solving Dynamic Equilibrium Models with Graphics Processors with Eric M.Aldrich, A.Ronald Gallant, and Juan F. Rubio-Ramírez.
- Programming Field-Programmable Gate Arrays for Economics, with Bhagath Cheela, André DeHon, and Alessandro Peri.
- Using a Quantum Annealer to Solve a Real Business Cycle Models, with Isaiah J. Hull.

Frontier: 9,472 64-core CPUs and 37,888 GPUs

GPUs

FPGAs

What do we do?

- We show how to use field-programmable gate arrays (FPGAs) and their high-level synthesis (HLS) compilers to solve models in economics.
- FPGAs are easily available at Amazon Web Services or similar.
- An application: solving a version of the Krusell-Smith (1998) model.
- Efficiency gains of FPGA acceleration on:
  - *Speedup*: Acceleration of one single FPGA is comparable to 78 CPU cores.
  - *Costs savings*: $<$18% of multi-core CPU acceleration.
  - *Energy savings*: $<$5% of multi-core CPU acceleration.

What is an FPGA?

- Integrated circuit that can be reconfigured by the user with a hardware description language (HDL).
- An FPGA is (basically) an array of programmable logic blocks (which can implement logic gates and combinatorial functions).
- Slower than CPUs/GPUs (3GHz/1GHz vs. 250 MHz), but much more flexible.
- In particular, we can allocate the logic blocks according to the algorithm's requirements.
- How do you do this in practice?
    1. In the past, one had to use lower-level programming in the register-transfer level (RTL) language (Verilog), as in Peri (2020). This was too cumbersome.
    2. Nowadays, we have HLS compilers that simplify programming by orders of magnitude.

Steps, I

- We pick a clean and representative testbed:
  1. We pick the model in Den Haan and Rendahl (2010). Why? → Canonical heterogeneous agent model.
  2. We pick the solution method in Maliar, Maliar, and Valli (2010). Why? → Fast and transparent solution algorithm.
  3. We code it in `C/C++` as in Aruoba and Fernández-Villaverde (2015). Why? → Most powerful programming language.
  4. We compile the code with the `GNU G++ 9.4.0` compiler with `-O3` flag. Why? → State-of-the-art, open-source compiler.
  5. We run the code on `AWS`. Why? → Easily available state-of-the-art processors/clusters at cheap (and measurable) costs.
  6. We check we get the same results as the original `Matlab` code by Maliar, Maliar, and Valli (2010). Our code is four times faster.

Steps, II

- Next, we take the `C/C++` code and add the `#PRAGMA`s available to the `AMD Xilinx HLS Vitis compiler`. Why? $\rightarrow$ industry standard.
- A `#PRAGMA` is a compiler directive that instructs the compiler on how to design the FPGA hardware.
- Using `#PRAGMA`s is not harder than using `MPI` messages.
- HLS compilers are bound to get easier and easier to use.
- We run the FPGA code and compare the results with the CPU code.
- We document that the acceleration delivered by one single FPGA is comparable to that provided by using 78 CPU cores in a conventional cluster.

Quantum computing

- The promise of quantum hardware.
- Solve real business cycle model (RBC) with dynamic programming on a quantum annealer.
- Construct novel algorithms that achieve near-minimum execution time, given the physical limits of the device.
- Demonstrate order-of-magnitude speed-up on quantum annealer over best classical solutions (value function iteration, VFI) taken from Aruoba and Fernández-Villaverde (2015).

Conclusion

- The "curse of dimensionality" is the key challenge dealing in economics.
- Fortunately, the last decade has seen important advances:
    1. Better numerical algorithms (e.g., deep learning, continuous-time methods).
    2. Better software implementations (e.g., differentiable and functional programming, flexible data structures, advances in massive parallelization).
    3. Better hardware designs (e.g., GPUs, AI accelerators, FPGAs, quantum hardware).
- We should expand the imagination of the class of models we can consider.