Deep learning for solving dynamic economic models.

Lilia Maliar[a], Serguei Maliar[b,*], Pablo Winant[c‡]

[a] The Graduate Center, City University of New York and CEPR;
[b] Santa Clara University;
[c] ESCP Business School and CREST/Ecole Polytechnique

Received Date; Received in Revised Form Date; Accepted Date

Abstract

We introduce a unified deep learning method that solves dynamic economic models by casting them into nonlinear regression equations. We derive such equations for three fundamental objects of economic dynamics – lifetime reward functions, Bellman equations and Euler equations. We estimate the decision functions on simulated data using a stochastic gradient descent method. We introduce an all-in-one integration operator that facilitates approximation of high-dimensional integrals. We use neural networks to perform model reduction and to handle multicollinearity. Our deep learning method is tractable in large-scale problems, e.g., Krusell and Smith (1998). We provide a TensorFlow code that accommodates a variety of applications.

Keywords: Artificial intelligence; Machine learning; Deep learning; Neural network; Stochastic gradient; Dynamic models; Model Reduction; Dynamic programming; Bellman equation; Euler equation; Value function

JEL classification: C61, C63, C65, C68, C88, E32, E37

1. Introduction

Artificial intelligence (AI) has remarkable applications, such as recognition of images and speech, facilitation of computer vision, operation of self-driving cars; see Goodfellow et al. (2016) for a review. At the same time, there are many interesting problems that computational economists cannot solve yet, including high-dimensional heterogeneous-agent models, large-scale central banking models, life-cycle models, and expensive nonlinear estimation procedures, among others. We show that it is possible to solve many challenging economic models by using the same AI technology, software and hardware that led to groundbreaking applications in data science. We specifically introduce an econometric-style deep learning (DL) method that solves dynamic economic models by reformulating them as nonlinear regression equations. Our four novel results are stated below:

First, we offer *a unified approach* which allows us to cast three fundamental objects of economic dynamics – *lifetime reward functions, Bellman equations* and *Euler equations* – into objective functions for Monte Carlo simulation. Such objective functions are given by a weighted sum of all of the model's equations, so we iterate on the entire model and solve for all decision functions at once. To optimize the constructed objective functions, we use deep learning regression

*Corresponding author: maliars@stanford.edu

†We are grateful to Marc Maliar for his help with writing the TensorFlow code for solving Krusell and Smith's (1998) model. We thank the editor and an annonimous referee for many useful comments and suggestions. The paper also circulated under the title "Will Artificial Intelligence Replace Computational Economists Any Time Soon?" We also received useful comments from participants of the 2018 Society for Computational Economics (CEF) Conference in Milan (invited session), 2018 Econometric Society Australasian Meeting (ESAM) in Auckland (invited talk), Oxford University, CEPREMAP, Banque de France, Panorisk Summer School, NYU Abhu Dabi, Paris Dauphine University, CREST (Ecole Polytechnique), Durham University, Santa Clara University, Rutgers University, PASC conference, Model comparison conference in Frankfurt, Complutense University of Madrid, Stanford University, Queens College, Stony Brook University, Columbia University, Deutsche Bundesbank, the Graduate Center, CUNY.

‡Lilia Maliar and Serguei Maliar acknowledge financial support from the NSF grants SES-1949413 and SES-1949430, respectively.

techniques from the fields of econometrics and data science. Once the regression coefficients are constructed, we infer the value and decision functions of the underlying dynamic economic models.

Second, we show how to adapt a *stochastic gradient descent method* to training of the three constructed objective functions. In each iteration, we use just one or a few (batch) grid points, which are randomly drawn from the state space, instead of a fixed grid with a large number of grid points used by conventional projection and value iterative methods. In small problems, we draw grid points from an exogenous solution domain, but in large problems, we produce grid points by stochastic simulation which allows us to focus on the ergodic set in which the solution "lives", avoiding the cost of computing solutions in those areas that are never visited in equilibrium. Thus, our DL framework aims not only on convergence of decision and value functions along iterations but also on convergence of simulated series to the ergodic set.

Third, we introduce the *all-in-one (AiO) expectation operator* for efficient approximation of integrals in Monte Carlo simulation. The objective functions, which we derive from economic models, have two types of expectation operators. One is with respect to next-period shocks (which appears naturally in stochastic models), and the other is with respect to the current state variables (which we created ourselves by drawing grid points randomly from the state space). Approximating these two nested expectation operators is costly, especially, in large-scale applications. The AiO method merges the two expectation operators into one, reducing the cost dramatically. It possesses a remarkable distributive property: a single composite Monte Carlo draw is used both for integration with respect to future shocks and for approximation of decision functions.

The way we construct the AiO operator differs for the three objective functions considered. For the lifetime reward method, we draw randomly the initial condition, in addition to future shocks. For the Euler-equation method, we use two independent random draws (or two independent batches) for evaluating two terms of the squared residual – this method eliminates the correlation between the two terms and helps us pull the expectation operator out of the square. Finally, for the Bellman-equation method, we introduce a value-iterative scheme that combines a minimization of residuals in the Bellman equation with a maximization of the right side of the Bellman equation into a single weighted-sum objective function. We use the Fischer-Burmeister function for a smooth approximation of Kuhn-Tucker conditions.

Our last important contribution is to implement the DL solution framework using the Google TensorFlow data platform–the same software that lead to ground-breaking applications in data science. Our implementation is versatile and portable to a variety of economic models and applications.[1]

The solution framework we introduce is not tied to neural networks but can be used with any approximating family (e.g., polynomials, splines, radial basis functions). However, neural networks possess several features that make them an excellent match for high-dimensional applications; namely, they are linearly scalable, robust to ill-conditioning, capable of model reduction and well suited for approximating highly nonlinear environments including kinks, discontinuities, discrete choices, switching.

We first illustrate our DL solution framework by using a simple one-agent consumption savings problem with a borrowing constraint. We implement three versions of the deep learning method based on *lifetime reward, Bellman equation* and *Euler equation* – they produce very similar solutions. Approximation errors do not exceed a fraction of a percentage point – an impressive accuracy level for a model with a kink in decision rules! Moreover, the computational expense increases practically linearly with the dimensionality of the state space – another outstanding feature of our DL method based on stochastic gradient and the AiO integration operator.

We then solve Krusell and Smith's (1998) model with heterogenous agents. Our solution procedure is conceptually straightforward – we simulate a panel of heterogenous agents, and we feed a distribution of labor productivity and wealth into the constructed objective functions for training. But there are two challenges: First, the decision function of each agent depends on the state variables of all agents, which makes the problem high dimensional. Second, the agent's state variables appear twice in the decision function (as agent's own state variables and as a part of the distribution), which leads to perfect collinearity. Fortunately, a neural network can deal with these challenges: First, it performs model reduction by extracting and condensing information from high-dimensional distributions into a smaller set of features of the hidden layers. Second, it learns to ignore the presence of redundant collinear variables. We again implement deep learning methods based on lifetime reward, Bellman equation and Euler equation and show that they produce very similar solutions. Our solution method is tractable in models with at least 1,000 agents (2,001 state variables) on a serial desktop computer!

We next propose a cheaper deep learning method that replaces the actual state space composed of distributions with a reduced state space composed of some aggregate statistics such the moments of wealth distribution studied in Krusell and Smith (1998). Implementing such a method requires no modifications: as before, we simulate a panel of

[1] Jupyter notebooks illustrating the method are available from open-source QuantEcon.org site https://notes.quantecon.org/submission/5ddb3c926bad3800109084bf.

heterogeneous agents, but we now feed in moments instead of distributions. In contrast, the method proposed by Krusell and Smith (1998) is far more complicated: they alternate between constructing individual and aggregate decision rules. Also, they rely on a regression of current moments on past moments, which is unnecessary in our case. Having relatively few moments, like 10 or 20, implies lower computational expense and allows us to increase the number of agents (at least) to 10,000 agents without a visible accuracy loss, so this cheaper method is a useful alternative to our baseline method.

We finally compare the solutions constructed with actual state space to those produced with a reduced state space. We find that the solution constructed by using the first moment of wealth distribution as in Krusell and Smith (1998) is somewhat shifted up relatively to our baseline solution. We tried to add second and third moments, but it did not help remove the shift. We then constructed a solution with the actual state space but using only 4 neurons which is parallel to 4 state variables in Krusell and Smith's (1998) method with one moment. We find that the 4-neuron solution is also shifted up near the kink area but the shift reduces for larger wealth levels. Furthermore, we find that having more neurons helps one get closer to the reference solution, unlike having more moments. These findings suggest that moments are not the best reduced representation of the actual state space which is not surprising given that the moments are selected by a guess, while neural networks are designed to search for the best possible reduced representation.

Our solution method is related to supervised-learning (because we fit the decision and value functions to the data which are artificial in our analysis), to unsupervised learning (because the decision and value functions are not explicitly labeled) and reinforcement learning (because we attempt to achieve the convergence of simulated series to the ergodic set, in addition to convergence of the neural network coefficients). We are not the only paper that uses machine learning tools for analyzing dynamic economic models. There are numerous methods that solve dynamic economic models on their ergodic sets approximated via stochastic simulation, such as the indirect inference procedure of Smith (1987) for maximizing the lifetime reward, a parameterized expectation algorithm (PEA) by Den Haan and Marcet (1990) for minimizing the Euler equation residuals and a value iterative method of Maliar and Maliar (2005) for minimizing the Bellman equation residuals. There are also methods that use unsupervised learning in order to aim to refine simulated points and determine the irregularly-shaped ergodic sets. In particular, Judd et al. (2011) uses clustering of simulated points, Maliar and Maliar (2015) combine simulated points in epsilon-distinguishable sets, Renner and Scheidegger (2018) and Scheidegger and Bilionis (2019) use Gaussian process machine learning to identify feasible sets. In turn, Jirniy and Lepetyuk (2011) show an early remarkable application of reinforcement learning for solving Krusell and Smith's (1998) model.

Furthermore, machine learning methods for model reduction and dealing with ill-conditioning are analyzed in Judd et al. (2011) including a principle component regression, a truncated SVD method, Tykhonov regularization and regularized least absolute deviation methods. The other methods that use model reduction for solving heterogeneous-agent models are Reiter (2010), Ahn et al (2018), Winberry (2018) and Bayer and Luetticke (2020).

Finally, early applications of neural networks date back to Duffy and McNelis (2001) and more recent applications include Duarte (2018), Fernández-Villaverde et al. (2019), Villa and Valaitis (2019) and Lepetyuk et al. (2020). These papers use neural networks for interpolation instead of polynomial functions. To the best of our knowledge, we are the first to cast an entire economic model into the state-of-the-art DL framework and to construct a solution on simulated points by using stochastic gradient descent method. There is also a paper by Azinovic et al. (2020) that uses a related Euler-equation method to solve a large-scale OLG problem. Like us, that paper uses deep neural network and random grid points but focuses only on the method that minimizes the Euler equation residuals while we offer a unified approach that applies also to the lifetime reward and Bellman operator. Another difference is that Azinovic et al. (2020) assume a finite number of shocks in which case integration is exact, while we show how to integrate stochastic processes with continuous transition density by using the AiO operator – a key contribution of our analysis. Finally, the techniques we developed in the present paper are used in Maliar and Maliar (2020) for constructing a classification deep learning method for modeling non-convex labor choices, and in Gorodnichenko et al. (2020) for solving a version of heterogeneous-agent new Keynesian model with uncertainty shocks.

The rest of the paper is organized as follows: Section 2 shows how to cast three main objects of economic dynamics (lifetime reward, Bellman equation and Euler equations) into expectation functions. Section 3 presents a deep learning solution method and provides a quick overview of its key ingredients (multilayer neural networks, stochastic gradient training method, etc.). Sections 4 and 5 analyze the one-agent consumption-saving model and Krusell and Smith's (1998) heterogeneous-agent model, respectively. Finally, Section 7 concludes.

2. Casting dynamic economic models into DL expectation functions

Deep learning platforms such as TensorFlow or PyTorch provide efficient ways of numerically approximating expectation functions with large numbers of parameters. In this section, we show how to cast dynamic economic models into the

form of expectation functions that can be suitable for deep learning platforms. Specifically, we show how to reformulate as expectation functions three key objects of economic dynamics: lifetime reward, Euler equation and Bellman equation.

2.1. A class of dynamic economic models

We consider a class of dynamic Markov economic models with time-invariant decision functions – the main framework in modern economic dynamics. An agent (consumer, firm, government, central bank, etc.) solves a canonical intertemporal optimization problem.[2]

Definition 3.1 (Optimization problem) *An exogenous state $m_{t+1} \in \mathbb{R}^{n_m}$ follows a Markov process driven by an i.i.d. innovation process $\epsilon_t \in \mathbb{R}^m$ with a transition function M,*

$$m_{t+1} = M(m_t, \epsilon_t). \tag{1}$$

An endogenous state s_{t+1} is driven by the exogenous state m_t and controlled by a choice $x_t \in \mathbb{R}^{n_x}$ according to a transition function S,

$$s_{t+1} = S(m_t, s_t, x_t, m_{t+1}). \tag{2}$$

The choice x_t satisfies the constraint in the form

$$x_t \in X(m_t, s_t). \tag{3}$$

The state (m_t, s_t) and choice x_t determine the period reward $r(m_t, s_t, x_t)$. The agent maximizes discounted lifetime reward

$$\max_{\{x_t, s_{t+1}\}_{t=0}^{\infty}} E_0 \left[\sum_{t=0}^{\infty} \beta^t r(m_t, s_t, x_t) \right], \tag{4}$$

where $\beta \in [0, 1)$ is the discount factor and $E_0[\cdot]$ is an expectation function across future shocks $(\epsilon_1, \epsilon_2, ...)$ conditional on the initial state (m_0, s_0).

Without loss of generality, we assume that the constrained sets are re-mapped into a set of real numbers, so that the transition and reward functions are defined for any succession of choices $x_t \in \mathbb{R}^{n_x}$. We focus on recursive Markov time-invariant solutions.

Definition 3.2 (Decision rules) *i) An optimal decision rule is a function $\varphi : \mathbb{R}^{n_m} \times \mathbb{R}^{n_s} \to \mathbb{R}^{n_x}$ such that $x_t = \varphi(m_t, s_t) \in X(m_t, s_t)$ for all t and the sequence $\{x_t, s_{t+1}\}_{t=0}^{\infty}$ maximizes the lifetime reward (4) for any initial condition (m_0, s_0)*
ii) A parametric decision rule is a member of a family of functions $\varphi(\cdot; \theta)$ parameterized by a real vector $\theta \in \Theta$ such that for each θ, we have $\varphi : \mathbb{R}^{n_m} \times \mathbb{R}^{n_s} \to \mathbb{R}^{n_x}$ and $x_t = \varphi(m_t, s_t) \in X(m_t, s_t)$ for all t.

Our goal is to find a vector of parameters $\theta \in \Theta$ under which the parametric decision rule $\varphi(\cdot; \theta)$ provides an accurate approximation of the optimal decision rule φ on a relevant domain. We do not assume a smoothness of the approximation function $\varphi(\cdot; \theta)$ nor its linearity with respect to coefficients θ and state (m_t, s_t). But we do require the problem to be time consistent, so that its solving amounts to finding time-invariant decision rules.

2.2. Objective 1: Lifetime-reward maximization

We first introduce a method that maximizes the lifetime reward (4) directly.

Definition 3.3 (Value function). *For a given distribution of shocks $(\epsilon_1, ..., \epsilon_T)$, value function $V(m_0, s_0)$ is a maximum expected lifetime reward (4) that is attainable from a given initial condition (m_0, s_0):*

$$V(m_0, s_0) \equiv \max_{\{x_t, s_{t+1}\}_{t=0}^{\infty}} E_{(\epsilon_1, ..., \epsilon_T)} \left[\sum_{t=0}^{\infty} \beta^t r(m_t, s_t, x_t) \right], \tag{5}$$

[2] A general model formulation in this paper matches standard API used by modeling software Dolo available at https://github.com/econforge/dolo. This makes it easily feasible to compare various deep-learning approaches described here with more traditional iterative methods already implemented in Dolo. We leave it for further work.

where transitions are determined by equations (1), (2) and (3).

For numerical approximation of V, we replace the infinite-horizon problem with a finite-horizon problem by truncating it at some finite $T < \infty$. We then simulate time series solution forward under a fixed decision rule $\varphi(\cdot;\theta)$ and evaluate the lifetime reward:

$$V^T(m_0, s_0; \theta) \equiv \quad E_{(\epsilon_1,...,\epsilon_T)}\left[\sum_{t=0}^{T} \beta^t r(m_t, s_t, \varphi(m_t, s_t; \theta))\right]. \tag{6}$$

Our first method constructs approximation $\varphi(\cdot;\theta)$ to the optimal decision rule by searching for a vector of coefficients θ that maximizes the lifetime reward (6).

A potential shortcoming of the objective function (6) depends on a specific initial condition (m_0, s_0). If we always start simulation from the same initial condition, we get an accurate approximation in a neighborhood of this specific initial condition but not for the states further away from this initial condition. Although the simulated series $\{(m_t, s_t)\}_{t=0}^{T}$ may pass many values, the contribution of future utility levels to the lifetime reward decreases with time due to discounting, so the initial condition still dominates accuracy. A possible way to achieve high accuracy on a larger domain would be to construct a solution on a grid of initial conditions $\{(m_0, s_0)\}$. However, here, we propose an alternative approach which is more suitable for Monte Carlo simulation implemented by deep learning tools, namely, we reformulate (6) as an expectation function. Instead of a fixed grid, we assume that initial condition (m_0, s_0) is drawn randomly from the domain on which we want the solution to be accurate, which yields the following objective function:

$$\Xi(\theta) \equiv \quad E_{(m_0,s_0)}\left\{E_{(\epsilon_1,...,\epsilon_T)}\left[\sum_{t=0}^{T} \beta^t r(m_t, s_t, \varphi(m_t, s_t; \theta))\right]\right\}. \tag{7}$$

By solving $\max_{\theta\in\Theta} \Xi(\theta)$, we construct a decision rule $\varphi(\cdot;\theta)$ that maximizes the lifetime reward for a given distribution of initial conditions.

A new feature of the objective function $\Xi(\theta)$ is that it has two types of randomness: one is a random sequence of future shocks $(\epsilon_1, ..., \epsilon_T)$, which appears because the model is stochastic, and the other is a random state (m_0, s_0), which we created ourselves because we converted the initial condition into a random variable. Approximating two nested expectation operators, one after the other, is costly, especially in high dimensional applications. That is, if we make n draws for evaluating expectation with respect to (m_0, s_0) and if we make n' draws for evaluating expectation with respect to $(\epsilon_1, ..., \epsilon_T)$, in total, we must evaluate $n \times n'$ draws.

To reduce the cost of nested integration, we introduce *all-in-one* (AiO) expectation operator that combines the two expectation operators into one.

Definition 3.4 (All-in-one expectation operator for lifetime reward) *Fix time horizon $T > 0$, parametrize a decision rule $\varphi(\cdot;\theta)$ and define the distribution of the random variable $\omega \equiv (m_0, s_0, \epsilon_1, ..., \epsilon_T)$. For given θ, lifetime reward (4) associated with the rule $\varphi(\cdot;\theta)$ is given by*

$$\Xi(\theta) = E_\omega[\xi(\omega;\theta)] \equiv E_{(m_0,s_0,\epsilon_1,...,\epsilon_T)}\left[\sum_{t=0}^{T} \beta^t r(m_t, s_t, \varphi(m_t, s_t; \theta))\right], \tag{8}$$

where transitions are determined by equations (1), (2) and (3), and ξ is an integrand.

The AiO operator can significantly reduce the cost of evaluation expectations. Instead of making $n \times n'$ draws for the two random vectors (m_0, s_0) and $(\epsilon_1, ..., \epsilon_T)$, we make just n draws for a composite random variable $(m_0, s_0, \epsilon_1, ..., \epsilon_T)$. Constructing the AiO operator is easy for the lifetime reward maximization studied in this section but it will be more challenging for the Euler and Bellman methods studied in next sections.

2.3. Objective 2: Euler-residual minimization

We next introduce a DL method that constructs a solution to the Euler equations. We consider a class of economic models in which the objective functions are differentiable, so that the solution is characterized by a set of first-order conditions (Euler equations). Such equations may follow from an optimal control problem of type (4) or from an equilibrium problem and may include first-order conditions, equilibrium conditions, transition equations, constraints, market clearing conditions, etc.

Definition 3.5 (Euler equations) *Euler equations are a set of equations written in the form:*

$$E_\epsilon \left[f_j \left(m, s, x, m', s', x' \right) \right] = 0, \qquad j = 1, ..., J, \tag{9}$$

where the agent's choice satisfies constraints (1), (2) and (3) expressed in a recursive form $m' = M(m, \epsilon)$, $s' = S(m, s, x, m')$ *and* $x \in X(m, s)$, *respectively;* $f_j : \mathbb{R}^{n_m} \times \mathbb{R}^{n_s} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_m} \times \mathbb{R}^{n_s} \times \mathbb{R}^{n_x} \to \mathbb{R}$ *and* $E_\epsilon [\cdot]$ *is an expectation operator with respect to the next-period shock* ϵ.

Equations (9) are again defined just for a given state (m, s). The typical approach in computational economics is to solve the Euler equation on a fixed grid that covers a relevant area of the state space. Like with lifetime reward, we do not follow this approach but assume that states (m, s) are drawn randomly from a given distribution. The corresponding objective function is defined as an expected squared sum of residuals in the Euler equations for a given distribution of states.

Definition 3.6 (Euler-residual minimization) *Select a decision rule* $\varphi(\cdot; \theta)$, *and define a distribution of random variable* (m, s). *For given* θ, *the expected squared residuals in the Euler equations (9) associated with the rule* $\varphi(\cdot; \theta)$ *are given by*

$$\Xi(\theta) = E_{(m,s)} \left\{ \sum_{j=1}^{J} v_j \left(E_\epsilon \left[f_j \left(m, s, \varphi \left(m, s; \theta \right), m', s', \varphi \left(m', s'; \theta \right) \right) \right] \right)^2 \right\}, \tag{10}$$

where $(v_1, ..., v_J)$ *is a vector of weights on* J *optimality conditions.*

Our goal is to construct a decision rule $\varphi(\cdot; \theta)$ that solves $\max_{\theta \in \Theta} \Xi(\theta)$.

Again, the objective function (10) has also two expectation operators, one with respect to the shocks $E_\epsilon [\cdot]$ and the other is with respect to the state $E_{(m,s)} [\cdot]$. In the case of lifetime reward maximization, combining the two expectation operators was easy because the expectation operators enter the objective linearly, so the AiO operator just merges them together, i.e., $E_{(m_0,s_0)} \left[E_{(\epsilon_1,...,\epsilon_T)} r(\cdot) \right] = E_{(m_0,s_0,\epsilon_1,...,\epsilon_T)} [r(\cdot)]$. However, in the Euler equations, $E_\epsilon [\cdot]$ is squared, so the two expectations cannot be naturally merged since $E_{(m,s)} \left(E_\epsilon [f_j(m, s, \epsilon)] \right)^2 \neq E_{ms} E_\epsilon \left[f_j(m, s, \epsilon)^2 \right]$.

An important contribution of the present paper is to offer a technique that allows us to combine the expectation functions $E_{m,s} [\cdot]$ and $E_\epsilon [\cdot]$ in the AiO expectation operator in the presence of squares. The technique is very simple but effective, namely, instead of using the same random draw ϵ for both terms in the square, we use two independent random draws or two batches ϵ_1 and ϵ_2 which yields

$$E_{\epsilon_1} \left[f(\epsilon_1) \right] E_{\epsilon_2} \left[f(\epsilon_2) \right] = E_{(\epsilon_1, \epsilon_2)} \left[f(\epsilon_1) f(\epsilon_2) \right]. \tag{11}$$

With this approach, we are able to write the Euler-residual function (10) as an expectation function $E_{ms} E_{\epsilon_1 \epsilon_2} [\cdot]$ of a single random vector.

Definition 3.7 (Euler-residual minimization with all-in-one expectation operator) Parametrize a *decision rule* $\varphi(\cdot; \theta)$, *and define a distribution of random variable* $\omega \equiv (m, s, \epsilon_1, \epsilon_2)$. *For a given* θ, *the squared residuals in the Euler equations (9) associated with the rule* $\varphi(\cdot; \theta)$ *are given by*

$$\begin{aligned} \Xi(\theta) &= E_\omega \left[\xi(\omega; \theta) \right] \\ &\equiv E_{(m,s,\epsilon_1,\epsilon_2)} \left\{ \sum_{j=1}^{J} v_j \left[f_j \left(m, s, x, m', s', x' \right)|_{\epsilon=\epsilon_1} \right] \left[f_j \left(m, s, x, m', s', x' \right)|_{\epsilon=\epsilon_2} \right] \right\}, \end{aligned} \tag{12}$$

where $(v_1, ..., v_J)$ *is a vector of weights on* J *optimality conditions.*

The method based on the AiO expectation operator is our main method. However, we also develop and test various hybrid methods that construct two expectations separately. For example, we use a Monte Carlo method for constructing $E_{(m,s)} [\cdot]$, and we use some other methods for constructing $E_\epsilon [\cdot]$ such as quadrature and monomial rules, sparse grids, low-discrepancy sequences – we show these sensitivity results in Section 5. Overall, we find that such hybrid methods are useful for small problems in which the construction of $E_\epsilon [\cdot]$ is inexpensive but the AiO expectation operator is critical for large problems, such as Krusell and Smith (1998) model studied in Section 6.

Objective 3: Bellman-residual minimization

Our last DL method constructs the decision rule to satisfy the Bellman equation.

Definition 3.8 (Bellman equation) *Value function $V : \mathbb{R}^{n_m} \times \mathbb{R}^{n_s} \to \mathbb{R}$ associated with the problem (4) satisfies:*

$$V(m,s) = \max_{x,s'} \left\{ r(m,s,x) + \beta E_\epsilon \left[V(m',s') \right] \right\}, \tag{13}$$

subject to constraints (1), (2) and (3) expressed in a recursive form $m' = M(m,\epsilon)$, $s' = S(m,s,x,m')$ and $x \in X(m,s)$.

Under the standard assumptions about r, M, S and X, the solution to (13) exists and is unique. Similar to the Euler-equation method, we can find an approximate decision rule $\varphi(\cdot;\theta)$ by minimizing the squared residuals in the Bellman equation (13) on a conventional fixed grid but in the spirit of deep learning, we will make (m,s) a random variable which is drawn from a given distribution.

In the case of Bellman operator, we face an additional element – a nontrivial "max" operator appears inside the squared residuals:

$$\Xi(\theta) = E_{(m,s)} \left[V(m,s) - \max_{x,s'} \left\{ r(m,s,x) + \beta E_\epsilon \left[V(m',s') \right] \right\} \right]^2.$$

There are three approaches in the literature for constructing a solution to the maximum operator, namely, the FOCs, the envelope condition and a direct search of maximum:

$$\begin{array}{cc} \text{FOC:} & r_x(m,s,x) + \beta \left\{ E_\epsilon \left[V_{s'}(m',s') \right] \right\} \frac{\partial s'}{\partial x} = 0, \\ \text{Envelope condition:} & r_s(m,s,x) = V_s(m,s), \\ \text{Direct optimization:} & \max\limits_{x,s'} \left\{ r(m,s,x) + \beta E_\epsilon \left[V(m',s') \right] \right\}. \end{array}$$

FOCs and direct optimization are used in conventional value function iteration, e.g., Rust (1996), Santos (1999), Aruoba et al. (2006), Stachurski (2009) while the envelope condition method is introduced in Maliar and Maliar (2013) and developed in Arellano et al. (2016).[3] Provided that any of these three conditions is enforced, we can eliminate the maximum operator from the Bellman equation. Consequently, we can formulate an objective function that solves for both value function and decision rule by combining minimization of the residuals in the Bellman equation with maximization of the right side of the Bellman equation. We focus on the FOC but the other two conditions can be treated in a similar way.[4]

Definition 3.9 (Bellman-residual minimization) *Parametrize a value function $V(\cdot;\theta_1)$ and decision rule $x = \varphi(\cdot;\theta_2)$ and define a distribution of the random variable $\omega \equiv (m,s)$. For given $\theta \equiv (\theta_1,\theta_2)$, the squared residuals in the Bellman equations (13) associated with $V(\cdot;\theta_1)$ and $\varphi(\cdot;\theta_2)$ are given by*

$$\Xi(\theta) \equiv E_{(m,s)} \left\{ V(m,s;\theta_1) - r(m,s,x) - \beta E_\epsilon \left[V(m',s';\theta_1) \right] \right\}^2 + v E_{(m,s)} \left\{ r_x(m,s,x) + \beta \left\{ E_\epsilon \left[V_{s'}(m',s';\theta_1) \right] \right\} \frac{\partial s'}{\partial x} \right\}^2, \tag{14}$$

where $v > 0$ is a vector of exogenous relative weights of equations in the two objectives.

Similar to the previous methods, the objective function for the Bellman equation has two expectation operators under the square. One expectation is taken with respect to the shocks $E_\epsilon[\cdot]$ and the other is with respect to the state $E_{(m,s)}[\cdot]$. Fortunately, we again can use the method of uncorrelated shocks (11) for constructing the AiO expectation operator.

Definition 3.10 (Bellman-residual minimization with all-in-one expectation operator). *Select value function $V(\cdot;\theta_1)$ and decision rule $x = \varphi(\cdot;\theta_2)$ and define the distribution of the random variable $\omega \equiv (m,s)$. For given*

[3] There is also a method of reformulating state space in terms of the future endogenous state variables by Carroll (2006), which is known as endogenous-grid method. It is straightforward to generalize the proposed techniques to include this method as well.

[4] In the earlier version of the paper, we study a version of the method that relies on direct search of a maximum.

$\theta \equiv (\theta_1, \theta_2)$, the squared residuals in the Bellman equations (13) associated with $V(\cdot;\theta_1)$ and $\varphi(\cdot;\theta_2)$ are given by

$$\begin{aligned}\Xi(\theta) = E_\omega\left[\xi(\omega;\theta)\right] \equiv E_{(m,s,\epsilon_1,\epsilon_2)}\Big\{&\left[V(m,s;\theta_1) - r(m,s,x) - \beta\, V(m',s';\theta_1)|_{\epsilon=\epsilon_1}\right] \\ &\times \left[V(m,s;\theta_1) - r(m,s,x) - \beta\, V(m',s';\theta_1)|_{\epsilon=\epsilon_2}\right] \\ &+v\left[r_x(m,s,x) + \beta\, V_{s'}(m',s';\theta_1)|_{\epsilon=\epsilon_1}\frac{\partial s'}{\partial x}\right]\left[r_x(m,s,x) + \beta\, V_{s'}(m',s';\theta_1)|_{\epsilon=\epsilon_2}\frac{\partial s'}{\partial x}\right]\Big\}, \quad (15)\end{aligned}$$

where $v > 0$ is a vector of exogenous relative weights of equations in the two objectives.

Like for the Euler equation, our main Bellman-equation method is the one based on AiO expectation operator but we could also construct hybrid methods based on the objective (14) that would combine Monte Carlo integration for the state space with deterministic integration across future shocks.

3. Deep learning solution method

In each of the considered cases (lifetime reward, Euler and Bellman equations), we represent an economic model as a problem of minimizing an objective function $\Xi(\theta)$ with respect to a vector of parameters θ:

$$\min_{\theta\in\Theta} \Xi(\theta) \quad = \quad \min_{\theta\in\Theta} E_\omega\left[\xi(\omega;\theta)\right], \quad (16)$$

where $\omega \equiv (m,s,\epsilon)$ includes exogenous state variables m, endogenous state variables s and future shocks ϵ. By construction, $\Xi(\theta)$ contains all model's equations (Euler and Bellman equations, constraints, market clearing conditions, transition equations, multipliers, prices), so by minimizing a single objective function, we solve the entire model.

In computational economics, a common approach to solving dynamic economic models is to use a fixed grid of points in the state space (m,s) and to approximate expectation functions over future shock ϵ with quadrature nodes, see, e.g., a projection method of Judd (1992). A distinctive feature of our analysis is that we interpret (16) not as a computational problem but as an estimation / regression model studied in the fields of econometrics and machine learning. In particular, we treat ω as a vector of random variables, and we make no distinction between its components (m,s,ϵ). We simulate the model to produce a set of random draws $\{\omega_i\}_{i=1}^n$, and we replace the expected risk $\Xi(\theta)$ with *empirical risk* $\Xi^n(\theta)$ – the sample average of ξ across n random draws – to obtain the following nonlinear regression model:

$$\min_{\theta\in\Theta} \Xi^n(\theta) \quad = \quad \min_{\theta\in\Theta} \frac{1}{n}\sum_{i=1}^n \xi(\omega_i;\theta). \quad (17)$$

We construct a solution θ by training the machine to minimize the empirical risk $\Xi^n(\theta)$ on the simulated data. Note that our data $\{\omega_i\}_{i=1}^n$ are being constantly re-sampled during the training process, unlike the data in a typical regression model that are assumed to be fixed. As a result, a successful training means two types of convergence: first, the parameter vector θ converges to a value that minimizes an objective function $\Xi^n(\theta)$ on simulated data $\{\omega_i\}_{i=1}^n$; and second, the simulated data themselves, generated via the decision rule $\varphi(\cdot;\theta)$, converge to the ergodic set. While our optimization problem is not equivalent to the typical data science application, we can still solve it by using the same combination of techniques that led to ground breaking applications in data science. Those techniques include deep neural networks, Monte Carlo simulation and stochastic optimization and they are discussed below in the context of a numerical method for solving dynamic economic models. In Supplement D, we discuss how our solution method is related to supervised, unsupervised and reinforcement learning literature.

Neural network: effective approximation on unstructured data. We approximate the decision rules and value function with neural networks instead of conventional polynomial functions. Neural networks posses several properties that make them preferable to polynomial functions in high dimensional applications with unstructured data; namely, they are: (i) linearly scalable, i.e., the number of parameters grows linearly with dimensionality; (ii) robust to multicollinearity and can automatically perform model reduction; and (iii) well suited for fitting highly nonlinear environments including kinks, discontinuities, discrete choices, and switching.

A neural network is a collection of connected nodes – artificial neurons. Each neuron receives a signal (input) from other neurons, processes it and transmits the processed signal to some other neurons connected to it. In Figure 1, we show an example of neural network with three layers – an input layer, hidden layer and output layer.

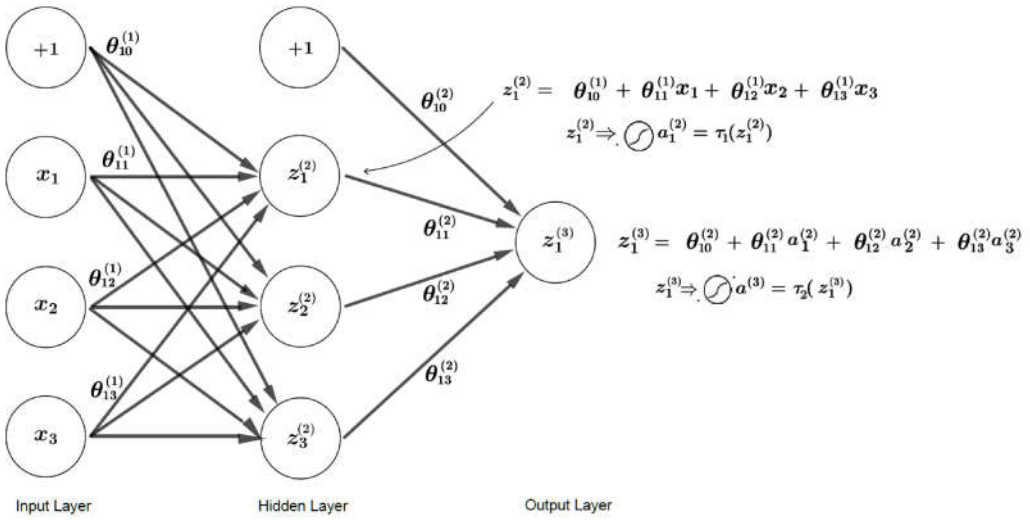

Figure 1. A neural network with one hidden layer.

The input layer consists of a constant term +1 and input features (x_1, x_2, x_3) which correspond to state variables of an economic model. In that layer, we construct linear polynomial functions $z_1^{(2)}$, $z_2^{(2)}$ and $z_3^{(2)}$on given inputs, for example, $z_1^{(2)} = \theta_{10}^{(1)} + \theta_{11}^{(1)}x_1 + \theta_{12}^{(1)}x_2 + \theta_{13}^{(1)}x_3$, where the coefficient on a constant term $\theta_{10}^{(1)}$ is called a bias, and the coefficients on features $\theta_{11}^{(1)}$, $\theta_{12}^{(1)}$ and $\theta_{13}^{(1)}$ are called weights. We next pass $z_1^{(2)}$, $z_2^{(2)}$ and $z_3^{(2)}$ to the hidden layer, where we apply to them a transformation τ_1, such as a sigmoid (logistic) activation function $\tau_1(x) = \frac{1}{1+e^{-x}}$.[5] Finally, in the output layer, we combine the activated signals $a_1^{(2)} = \tau_1(z_1^{(2)})$, $a_2^{(2)} = \tau_1(z_2^{(2)})$ and $a_3^{(2)} = \tau_1(z_3^{(2)})$ with a constant term +1 into a new polynomial function $z_1^{(3)} = \theta_{10}^{(2)} + \theta_{11}^{(2)}a_1^{(2)} + \theta_{12}^{(2)}a_3^{(2)} + \theta_{13}^{(2)}a_3^{(2)}$, and we transform it into the final output using another activation function τ_2 which approximates our decision rule $a_2^{(3)} = \tau_2(z_1^{(2)}) \approx \varphi(x_1, x_2, x_3; \theta)$, where $\theta \equiv \left\{\theta_{10}^{(1)}, \theta_{11}^{(1)}, ..., \theta_{13}^{(2)}\right\}$ contains all biases and weights. The predicted output is a highly non-linear function of inputs. Hidden layers extract information and condense it in a more abstract way which makes neural-network approximations more flexible, compared to polynomial functions that relate inputs and outputs directly; see Supplement B for a general discussion of neural networks.

Generating data: solving the model where the solution lives. Judd et al. (2011) argue that stochastic simulation methods have a remarkable feature that makes them an ideal candidate for analyzing high dimensional applications: they solve the model only in the area of the state space in which the solution "lives" - the ergodic set. Figure 2 illustrates this point by showing the ergodic set of a typical representative-agent neoclassical growth model (with two state variables, capital and productivity level).

[5] This is a convenient choice since it allows for an easy construction of the derivative function $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. See the online supplement for examples of other common activation functions such as hyperbolic tangent and leaky relu.

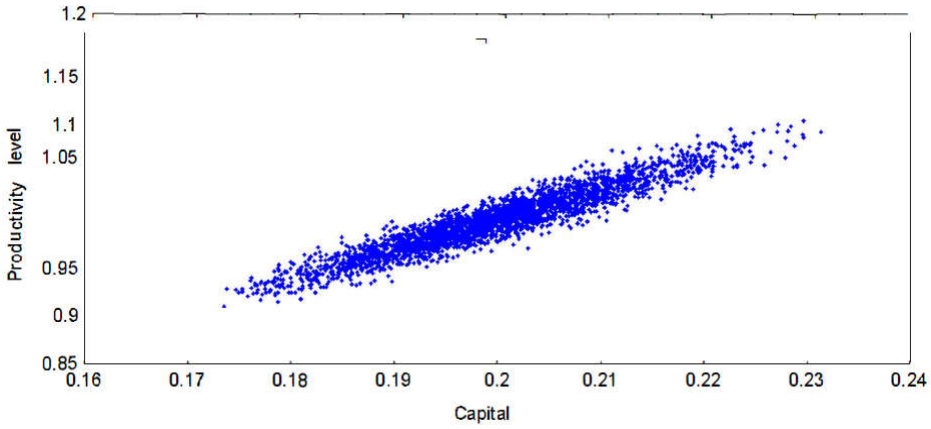

Figure 2. Ergodic set of a neoclassical growth model.

The ergodic set takes the form of an oval and most of the rectangular area that sits outside of the oval's boundaries is never visited. That means that a solution method that operates on a rectangular domain is wasting computational resources on evaluating points that never occur in equilibrium. In the two-dimensional case, a circle inscribed within a square occupies about 79% of the area of the square, so we save 21% of the total cost. The saving is not big in two-dimensional case but Judd et al. (2011) argues that the ratio $\mathcal{V}^d$ of the volume of a $d-$dimensional hypersphere to the volume of a d–dimensional hypercube declines rapidly with the dimensionality of the state space

$$\mathcal{V}^d = \begin{cases} \frac{(\pi/2)^{\frac{d-1}{2}}}{1\cdot 3\cdot ....\cdot d} \text{ for } d = 1,3,5... \\ \\ \frac{(\pi/2)^{\frac{d}{2}}}{2\cdot 4\cdot ....\cdot d} \text{ for } d = 2,4,6... \end{cases}. \tag{18}$$

For example, for dimensions three, four, five, ten and thirty, this ratio is 0.52, 0.31, 0.16, $3\cdot 10^{-3}$ and $2\cdot 10^{-14}$, respectively. Thus, when focusing on the ergodic set, we face just a tiny fraction of the cost that we would have faced on a fixed hypercube grid. The higher is the dimensionality of a problem, the larger is the reduction in cost.

AiO integration operator: we need just two integration nodes. The AiO operator is a critical technique for solving models with a large number of stochastic shocks. As an illustration, consider the Euler-equation method in the model with ℓ heterogeneous agents who face idiosyncratic shocks $\epsilon_1, ..., \epsilon_\ell$. In the absence of the AiO operator, the objective function (10) has the form $\Xi(\theta) = E_{(m,s)}\left(E_{\epsilon_1,...,\epsilon_\ell}\left[f(m,s,\epsilon_1,...,\epsilon_\ell;\theta)\right]\right)^2$, so for each state (m,s), we need to approximate integral of f across $\epsilon_1, ..., \epsilon_\ell$. If we consider p integration nodes for each of ℓ idiosyncratic shock, we obtain a tensor product grid with p^ℓ nodes, running into a severe curse of dimensionality. The AiO method addresses the curse of dimensionality in a remarkably simple manner. By using two independent random draws $(\epsilon'_1, ..., \epsilon'_\ell)$ and $(\epsilon''_1, ..., \epsilon''_\ell)$, we obtain:

$$E_{(m,s)}\left(E_{\epsilon_1,...,\epsilon_\ell}\left[f(m,s,\epsilon_1,...,\epsilon_\ell)\right]\right)^2 = E_{\left(m,s,\epsilon'_1,...,\epsilon'_\ell,\epsilon''_1,...,\epsilon''_\ell\right)}\left[f(m,s,\epsilon'_1,...,\epsilon'_\ell)f(m,s,\epsilon''_1,...,\epsilon''_\ell)\right].$$

Independently of the number of shocks in the economy, we need only two random draws (or two batches) for approximating expectation function for each state (m,s) considered. Of course, the AiO approximation of the double integral is crude in any given iteration, but the approximation is unbiased and thus, converges to the true integral over the iterative process. In sum, the AiO method makes numerical integration very cheap.

Stochastic gradient descent method for training: we need just one grid point. Training of multilayer neural network is referred to as *deep learning* because such networks have interconnected topologies with coefficients and weights buried deeply in multiple layers. In data science application, neural networks are typically trained by using variants of a gradient descent method $\theta_{k+1} \leftarrow \theta_k - \lambda_k \nabla \Xi(\theta_k)$, where k is iteration, ∇f is a gradient of f and λ_k is a learning rate. Our definition of (17) means that expectation and gradient operators are commutable, so that we can approximate the gradient of the integral $\nabla \Xi(\theta) = \nabla E_\omega\left[\xi(\omega;\theta)\right]$ with a sample average of the gradient of the integrand

$\frac{1}{n}\sum_{i=1}^{n}\nabla_\theta \xi(\omega_i;\theta_k)$:

$$\theta_{k+1} \leftarrow \theta_k - \lambda_k \left[\frac{1}{n}\sum_{i=1}^{n}\nabla_\theta \xi(\omega_i;\theta_k)\right]. \tag{19}$$

The limiting case of $n = 1$ in (19) corresponds to a stochastic gradient descent (SGD) method that approximates the gradient of the integral with the gradient of the integrand evaluated in just one randomly selected point, i.e., $\nabla\Xi(\theta_k) = \nabla_\theta E_\omega[\xi(\omega;\theta_k)] \approx \nabla_\theta \xi(\omega_i;\theta_k)$. While stochastic gradient is very imprecise in each given step, it is an unbiased estimate of the true gradient and its cumulative average converges to the true gradient over K iterations $\frac{1}{K}\sum_{k=1}^{K}\nabla_\theta \xi(\omega_k;\theta_k) \to \nabla_\theta \Xi(\theta_k)$, provided that the network parameters converge to their true values $\theta_k \to \theta$. Thus, an extreme version of our DL method requires just one random grid point for approximation (and just two random nodes for integration). There are other versions of the SGD method, in particular, we will be using a method called ADAM in our numerical analysis; see Supplement C for a review of SGD methods and their convergence properties.

DL solution algorithm. We now combine the above numerical techniques into a DL solution algorithm.

Algorithm 1. *DL algorithm for solving dynamic economic models.*

Step 1. Initialize the algorithm:

i). construct theoretical risk $\Xi(\theta) = E_\omega[\xi(\omega;\theta)]$ (lifetime reward, Euler/Bellman equations);
ii). define empirical risk $\Xi^n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\xi(\omega_i;\theta)$;
iii). define a topology of neural network $\varphi(\cdot,\theta)$;
iv). fix initial vector of the coefficients θ.

Step 2. Train the machine, i.e., find θ that minimizes the empirical risk $\Xi^n(\theta)$:

i). simulate the model to produce data $\{\omega_i\}_{i=1}^{n}$ by using the decision rule $\varphi(\cdot,\theta)$;
ii). construct the gradient $\nabla\Xi^n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\nabla\xi(\omega_i;\theta)$;
iii). update the coefficients $\widehat{\theta} = \theta - \lambda_k \nabla\Xi^n(\theta)$ and go to step 2.i);
End Step 2 if the convergence criterion $\left\|\widehat{\theta} - \theta\right\| < \varepsilon$ is satisfied.

Step 3. Assess the accuracy of constructed approximation $\varphi(\cdot,\theta)$ on a new sample.

Algorithm 1 has multiple hyperparameters including the topology of neural network, the learning rate, the number of simulation points and integration nodes, and the training method. The algorithm may also include additional hyperparameters such as Tykhonov and Lasso regularization parameters for dealing with overfitting and ill-conditioning; see Judd et al. (2011) for a discussion of numerical stability of stochastic simulation methods. Finally, objective function (17) has its own hyperparameters, namely, the relative weights of different model equations. To select the hyperparameters, we use the usual validation procedure of assessing the algorithm performance under different hyperparameter combinations and by selecting those that dominate others in accuracy and speed.

Finally, to implement the DL solution method, we use the Python programming language and Google TensorFlow data platform. Such a platform represents operations on a computational graph, which is automatically optimized. The elements of the graphs, *tensors*, are multidimensional arrays manipulated by efficient vectorized symbolic engines. In particular, gradient computations are performed using automatic differentiation in a numerically stable way, without any user input. Moreover, the graph operations are massively parallelizable on multiple CPU and GPU cores. This is a particularly useful property for our Monte Carlo simulation, in which we evaluate the same function with many draws of shocks for computing conditional expectation functions.

How our deep learning method differs from the conventional projection method. To appreciate the advantages of our DL algorithm, let us recall a canonical projection method of Judd (1992). That method approximates decision rules $\varphi(\cdot;\theta) \approx \varphi$ with a tensor product of Chebyshev polynomial basis functions and constructs a solution on a fixed tensor-product grid of zeros of Chebychev polynomials. The expectation function $E_\epsilon[\cdot]$ is approximated using Gauss-Hermite quadrature. The algorithm finds θ by minimizing the squared sum of Euler equation residuals $\left(E_\epsilon[f_j(m,s,x,m',s',x')]\right)^2$ using a Newton-style method. The method is remarkably fast for small problems but becomes increasingly costly as the dimensionality of the problem increases.

What makes the canonical projection method expensive in high dimensional applications? i) The volume of a hypercube domain increases exponentially with the number of state variables; ii) the number of points in tensor-product

grid covering that domain grows exponentially with the number of state variables; iii) the volume of a hypercube set for integration across future shocks grows exponentially with the number of shocks; iv) the number of quadrature nodes in the tensor product grid grows exponentially with the number of shocks; v) for each grid point in state space, there are multiple nodes for approximating expectation functions; vi) for more complex models, the cost is higher because we have more equations and more variables to solve for; vii) the number of approximation functions is paired with the number of grid points, so the number of coefficients in θ grows exponentially as well; viii) the least square approximation suffers from ill-conditioning and numerical instability; ix) the code written by individual researchers is not always optimized for the best performance.

Our DL analysis addresses all these shortcomings: i) our simulation-based domain focuses on the ergodic set in which the solution lives; ii) within that reduced domain, we consider just one or few random grid points on each iteration; iii) we approximate integrals with simulated shocks that again come from the ergodic set; iv) we consider only two random integration nodes on each iteration; v) our AiO operator reduces the cost of integration even further by using a composite random draw for both improving the approximation and evaluating the integral; vi) we collect all the model's equations in one objective function, so that we iterate on all decision rules at once; vii) we use a deep learning neural network, in which we control the topology and number of coefficients; viii) neural networks perform the model reduction and automatically deal with ill conditioning and multicollinearity. ix) we use the state-of-the-art combination of software and hardware that allows for effective GPU parallelization and that leads to remarkable applications in data science. Taken together, these methods will allow us to analyze problems with much larger dimensionality (thousands of state variables) than those studied in the related literature.

4. Numerical analysis of the consumption-saving problem

In this section, we solve a one-agent consumption-saving problem with an occasionally binding borrowing constraint. We show the deep learning method for three different objective function: the lifetime reward, and the Euler- and Bellman-equation residuals. Our experiments are designed to illustrate the role of hyperparameters in the solution, as well as to emphasize some useful features of the proposed method, in particular, its ability to accurately approximate kinks, its capacity to handle ill-conditioned problems and its scalability.

4.1. The consumption-saving problem

We consider a simple consumption-saving problem with a borrowing constraint:

$$\max_{\{c_t, w_{t+1}\}_{t=0}^{\infty}} E_0 \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] \tag{20}$$

$$\text{s.t. } w_{t+1} = r(w_t - c_t) + e^{y_t}, \tag{21}$$

$$c_t \leq w_t, \tag{22}$$

where c_t and w_t are consumption and cash-on-hand, respectively; u is a utility function, which is strictly increasing and concave; $\beta \in [0,1)$ is a discount factor; $r \in \left(0, \frac{1}{\beta}\right)$ is an interest rate, and initial condition w_0 is given. Exogenous income shock y_t follows an AR(1) process,

$$y_{t+1} = \rho y_t + \sigma \epsilon_t \text{ and } \epsilon_t \sim \mathcal{N}(0,1), \tag{23}$$

where $|\rho| < 1$ and $\sigma > 0$. The borrowing limit in (22) is set to zero without loss of generality. We parameterize the model by $u(c) = \frac{c^{1-\gamma}-1}{1-\gamma}$ with a risk-aversion coefficient of $\gamma = 2$, and we assume $\beta = 0.9$ and $r = 1.04$.

To facilitate the exposition of the algorithm, we make two simplifying assumptions: First, we assume that the income shock is temporary $y_t = \sigma\epsilon_t$, where $\sigma = 0.1$. Then, we have just one state variable w, which is convenient for illustrating decision functions on two-dimensional plots. Second, we assume that y is drawn from its ergodic (Normal) distribution, but w is drawn from a uniform distribution in an interval $[w_1, w_2]$. That allows us to abstract from the convergence of simulated series for w and to concentrate on the convergence of the coefficients θ. Later in the paper, we consider models with a large number of state variables, and we draw both endogenous and exogenous state variables from their ergodic distributions.

Euler equation The solution to (20)–(23) can be characterized by Kuhn-Tucker conditions

$$A \geq 0,\ H \geq 0 \text{ and } AH = 0, \tag{24}$$

where $A = w - c$ and $H \equiv u'(c) - \beta r E_\epsilon [u'(c')]$ is a Lagrange multiplier. To construct a DL objective function, we rewrite Kuhn Tucker conditions that have inequality constraints (24) as the Fischer-Burmeister (FB) function that holds with equality

$$\Psi^{FB}(a, h) = a + h - \sqrt{a^2 + h^2} = 0, \tag{25}$$

where $a \equiv 1 - \frac{c}{w}$ and $h \equiv 1 - \frac{\beta r E_\epsilon[u'(c')]}{u'(c)}$ are expressed in unit-free form. The FB function is similar to the minimum function $\min\{a, h\} = 0$ and leads to the same solution (24) as the Kuhn Tucker conditions but it is differentiable; see, e.g., Jiang (1996) for a discussion. Even though the two terms a and h are in unit free form, it might be necessary to add a weight v that reflects the relative importance of the two objectives a and h, i.e., in general, we may need to consider $\Psi^{FB}(a, vh)$, where $v \in (0, \infty)$.

Bellman equation The solution to (20)–(23) can be also characterized by Bellman equation:

$$V(y, w) = \max_{c, w'} \{u(c) + \beta E_\epsilon [V(y', w')]\}, \tag{26}$$

subject to constraints (21) and (22) and transition equation (23) written in recursive form. The maximum operator of the Bellman equation is also characterized by the Kuhn-Tucker conditions (24) however the Lagrange multiplier is defined here in terms of the derivative of the value function $H \equiv u'(c) - \beta E_\epsilon \left[\frac{\partial V(y', w')}{\partial w'}\right]$. Consequently, we can get rid of the inequality constraints in the Kuhn-Tucker conditions by using the same FB function (25) as we did for the Euler-equation method. The first term of the FB function is the same, i.e, $a \equiv 1 - \frac{c}{w}$ but the second term will be defined in terms of the derivative of the value function $h \equiv 1 - \frac{\beta E_\epsilon \left[\frac{\partial V(y', w')}{\partial w'}\right]}{u'(c)}$.

4.2. Deep learning solution method

To implement the lifetime reward, Euler and Bellman methods, we used Algorithm 1 formulated in Section 2. The only difference between three methods consists in how we simulate the model in Step 2i). We will describe this step separately for each of the method considered.

We construct the solutions using a neural network with two identical hidden layers, composed of (leaky) relu (rectified linear units) neurons. We compare the results under four neural networks with 8×8, 16×16, 32×32 and 64×64 neurons in two hidden layers, respectively. We parameterize consumption to wealth ratio $\frac{c_t}{w_t}$, unit-free Lagrange multiplier h_t and value function V_t:

$$\begin{aligned} \frac{c_t}{w_t} &= \sigma(\zeta_0 + \eta(y_t, w_t; \vartheta)) \equiv \varphi(y_t, w_t; \theta), \\ h_t &= \exp(\zeta_0 + \eta(y_t, w_t; \vartheta)) \equiv h(y_t, w_t; \theta), \\ V_t &= \zeta_0 + \eta(y_t, w_t; \vartheta) \equiv V(y_t, w_t; \theta), \end{aligned}$$

where $\eta(\cdot; \vartheta)$ is a neural network, $\theta \equiv (\zeta_0, \vartheta)$ and $\sigma(x) = \frac{1}{1+e^{-x}}$ is a sigmoid transformation. Our lifetime reward method will use only $\varphi(y_t, w_t; \theta)$, the Euler method will use both $\varphi(y_t, w_t; \theta)$ and $h(y_t, w_t; \theta)$ and finally, the Bellman method will use all three of them.

Our parametrization is constructed to take into account the properties of economic variables. A sigmoid transformation used for $\frac{c_t}{w_t}$ ensures that the consumption share in wealth $\varphi(\cdot; \theta)$ is bounded to be in an interval $[0, 1]$ and exponentiation used for h_t ensures that it is always nonnegative; and finally, for value function we used a linear activation, i.e., we place no restrictions on $V(\cdot; \theta)$ range. We initialize at $\zeta_0 = 0$ for all parametrized functions. The remaining parameters ϑ are initialized randomly: we used the "*he*" and "*glorot*" uniform distributions for the biases and weights, respectively.

We train the model using a version of the stochastic gradient descent method, called *ADAM*, with an overall learning rate of $\lambda = 0.001$ (in that method, the overall learning rate is optimally adjusted for each coefficient; see Appendix B for details). We perform training over $K = 50,000$ epochs (iterations); and in each epoch, we draw 64 random grid points by using the interval for wealth $[w_1, w_2] = [0.1, 4]$. To evaluate the accuracy, we produce $8,192$ random draws and use the constructed decision rules to produce the lifetime reward and unit-free Euler equation residuals. To approximate integrals in the accuracy test, we use an accurate 10-node Gauss-Hermite quadrature rule. We wrote the code in Python using Google TensorFlow platform version 1.14.0, and we use a laptop with Intel(R) Core(TM) i7-7500U (2.70 GHz), RAM 16GB with 4 physical (and 8 virtual) cores.

4.3. Lifetime reward

The objective function for the lifetime reward for the model (20)–(23) follows directly from our general exposition (8). For a random draw $\omega = (y_0, w_0, \epsilon_1, ..., \epsilon_T)$ and time horizon T, the objective function associated with the decision rule $\frac{c_t}{w_t} = \varphi(y_t, w_t; \theta)$ is given by

$$\Xi(\theta) = E_\omega [\xi(\omega; \theta)] \equiv E_{(y_0, w_0, \epsilon_1, ..., \epsilon_T)} \left[\sum_{t=0}^{T} \beta^t u(c_t) \right]. \tag{27}$$

To construct (27), i.e., to implement Step 2i) of Algorithm 1, we use the assumed decision rule $\frac{c_t}{w_t} = \varphi(y_t, w_t; \theta)$ and simulate the model forward together with transitions (21)–(23).

Figure 3 displays the outcome of training of neural network on the objective function (27) (on the horizontal axis, $K = 50,000$ epochs appears as $4 \cdot \log_{10} 5$).

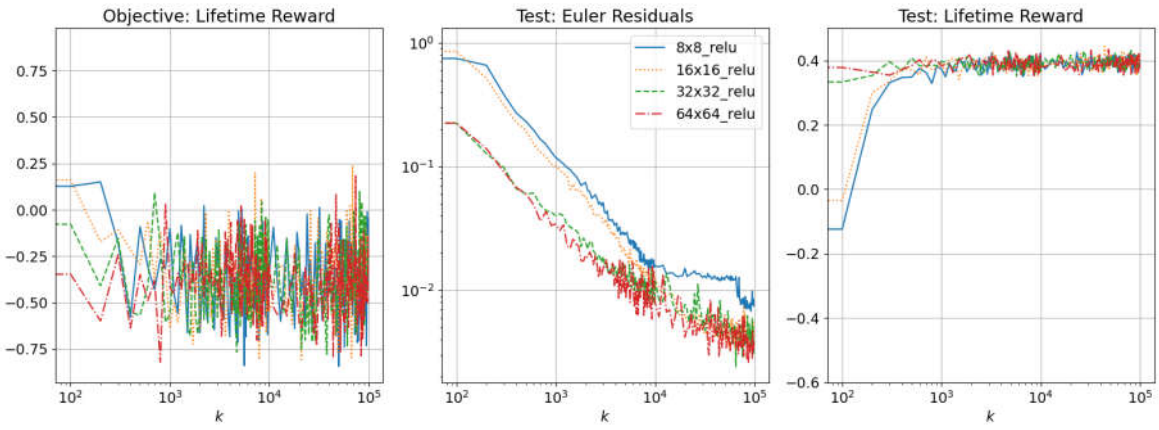

Figure 3. Lifetime reward in the consumption-saving model.

In the first panel of the figure, we illustrate how the objective function – lifetime reward – changes over the epochs; in the second panel, we show the size of the Euler equation residuals constructed on $8,192$ test points using the FB function (25) and accurate quadrature integration, and finally, in the last panel, we plot the evolution of lifetime reward. As is seen from the second panel, the Euler residuals are smaller for a neural network with a larger number of neurons (the smallest residuals are for the one with 64×64 neurons in two hidden layers). They are of order $10^{-2.75}$ (i.e., 0.18%), which is fairly low, given that we solve the model with a kink in the decision rule. In terms of the lifetime reward, all the solutions gradually converge to the same level.

4.4. Euler-equation method with Kuhn-Tucker conditions

For our model with borrowing constraint, the Euler-residual (10) objective function is given by the Fischer-Burmeister function (25):

$$E_{(y,w)} \left[\Psi^{FB} \left(1 - \frac{c}{w}, 1 - \frac{\beta r E_\epsilon [u'(c')]}{u'(c)} \right) \right]^2. \tag{28}$$

In Section 1.3, we constructed an AiO expectation operator that makes it possible to combine the two expectation operators $E_{(y,w)}[\cdot]$ and $E_\epsilon[\cdot]$ in one by approximating $E_\epsilon[\cdot]$ using two uncorrelated shocks (11). However, the Fischer-Burmeister function has expectation term $E_\epsilon[u'(c')]$ inside of the square root, in addition to the squared residual. To extend the AiO operator for the FB function, we introduce a separate approximation for the Lagrange multiplier h and rewrite (28) as a composite objective

$$E_{(z,w)} \left[\Psi^{FB} \left(1 - \frac{c}{w}, 1 - h \right) \right]^2 + v_h \left[\frac{\beta r E_\epsilon [u'(c')]}{u'(c)} - h \right]^2, \tag{29}$$

where v_h is an exogenous weight. Using the technique of two uncorrelated shocks (11), we then arrive to an objective function with the AiO operator that combines integration with respect to z, w and ϵ :

$$\Xi(\theta) = E_{\omega}\left[\xi\left(\omega;\theta\right)\right] = E_{(z,w,\epsilon_1,\epsilon_2)}\left\{\left[\Psi^{FB}\left(1-\frac{c}{w},1-h\right)\right]^2 + v_h\left[\frac{\beta r\ u'\left(c'\right)\big|_{\epsilon=\epsilon_1}}{u'\left(c\right)}-h\right]\left[\frac{\beta r\ u'\left(c'\right)\big|_{\epsilon=\epsilon_2}}{u'\left(c\right)}-h\right]\right\}. \quad (30)$$

The objective function (30) is the one that we minimize by deep learning Algorithm 1. To construct the objective function $\Xi(\theta)$ in Step 2i) of Algorithm 1, we produce a random draw $\omega = (y, w, \epsilon_1, \epsilon_2)$ and use the decision rules for consumption share $\frac{c}{w} = \varphi\left(y, w; \theta\right)$ and Lagrange multiplier $h = h\left(y, w; \theta\right)$ and transition equations (21) and (23).

The results of training under the objective (30) are shown in Figure 4.

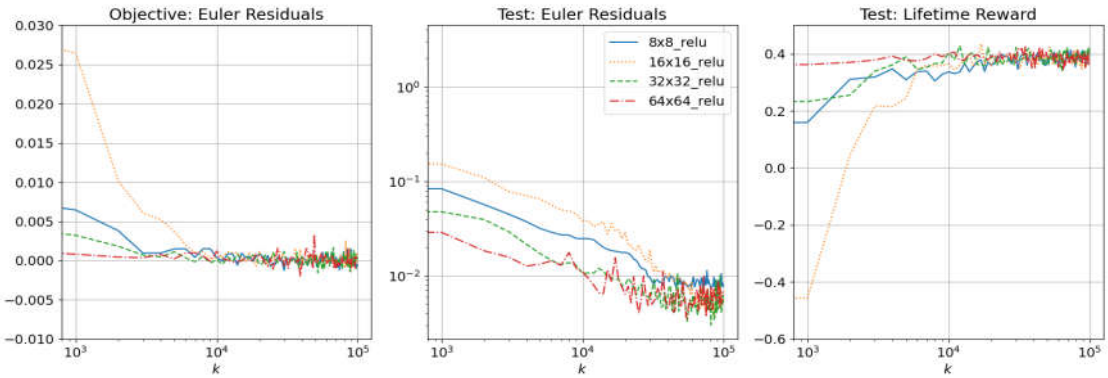

Figure 4. Euler-equation method in the consumption-saving model.

Again, as in the previous figure, we plot the objective function – Euler residuals, the size of the Euler equation residuals on the test data, and the evolution of the lifetime reward in the first, second and third panels, respectively. As is evident from the second panel, the Euler-equation method is slightly more accurate than the lifetime reward method. We again observe that the deep learning method with a larger number of neurons is more accurate.

4.5. Objective 3: Bellman equation

For the Bellman-equation method, the objective function (14) includes the residuals in the Bellman equation and the residuals in the Fischer-Burmeister function that characterize the maximum operator in the Bellman equation

$$E_{(z,w)}\left[V\left(z,w;\theta_1\right)-u\left(c\right)-\beta E_{\epsilon}\left[V\left(z',w';\theta_1\right)\right]\right]^2 + vE_{(y,w)}\left[\Psi^{FB}\left(1-\frac{c}{w},1-\frac{\beta E_{\epsilon}\left[\frac{\partial}{\partial w'}V\left(z',w';\theta\right)\right]}{u'\left(c\right)}\right)\right]^2. \quad (31)$$

By following the same approach as in the case of the Euler-equation method, we introduce a separate approximation for the Lagrange multiplier h and apply the method of two uncorrelated shocks to obtain the following objective function for the Bellman-equation method:

$$\Xi(\theta) = E_{\omega}\left[\xi\left(\omega;\theta\right)\right] \equiv E_{(y,w,\epsilon_1,\epsilon_2)}\left\{\left[V\left(y,w;\theta\right)-u\left(c\right)-\beta\ V\left(y',w';\theta\right)\big|_{\epsilon=\epsilon_1}\right] \times\left[V\left(y,w;\theta\right)-u\left(c\right)-\beta\ V\left(y',w';\theta\right)\big|_{\epsilon=\epsilon_2}\right]+v\left[\Psi^{FB}\left(1-\frac{c}{w},1-h\right)\right]^2 + v_h\left[\frac{\beta\ \frac{\partial}{\partial w'}V\left(y',w';\theta\right)\big|_{\epsilon=\epsilon_1}}{u'\left(c\right)}-h\right]\left[\frac{\beta\ \frac{\partial}{\partial w'}V\left(y',w';\theta\right)\big|_{\epsilon=\epsilon_2}}{u'\left(c\right)}-h\right]\right\}. \quad (32)$$

The objective function (32) is the one that we use as an input for deep learning Algorithm 1. To implement Step 2i), i.e, to construct the objective function (32), we produce a random draw $\omega = (y, w, \epsilon_1, \epsilon_2)$, use the decision rules for value function $V\left(y, w; \theta\right)$, consumption share $\frac{c}{w} = \varphi\left(y, w; \theta\right)$ and Lagrange multiplier $h = h\left(y, w; \theta\right)$ and transition equations

(21) and (23).

The choice of weights v and v_h is a nontrivial problem. While in the previous methods, we expressed the residuals in unit free form, so that we could use unit weights, it is not a feasible approach here. Value function for our model has the range from negative to positive numbers. If we do it unit free, we will get sometimes a division by numbers close to zero. Additionally, the residuals can switch from positive to negative leading to spurious iterations of the gradient algorithm. In our application, we choose weights so that the size of the residuals was approximately the same in all three equations.

Figure 5 plots the training process for Objective 3 when neural networks have different number of neurons.

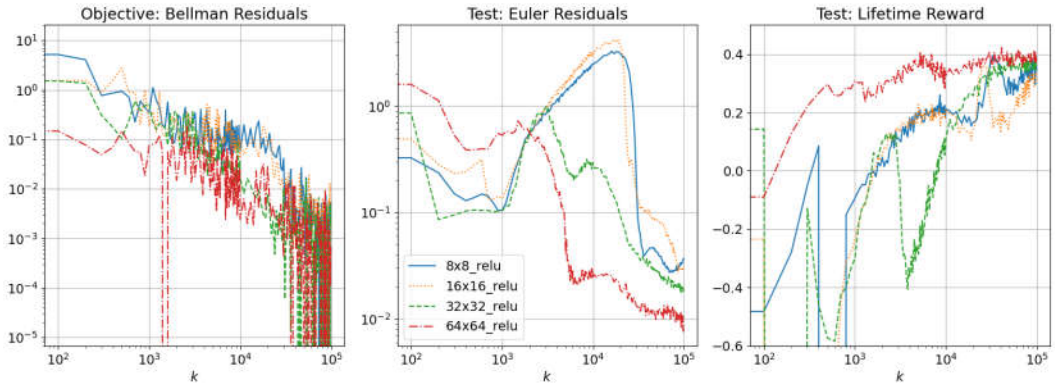

Figure 5. Bellman-equation method in the consumption-saving model.

Figure 5 displays the changes in the objective function over the epochs (the first panel), the size of test Euler-equation residuals (the second panel), and the size of the lifetime reward (the third panel). The value iterative method is less accurate than the two previous methods. This fact is not surprising: Coleman et al. (2018) compare the conventional VFI method which solves the Bellman equation by constructing the decision function via direct maximization, as we do here, versus otherwise identical methods that use derivatives of the value function via first-order and envelope conditions. They find that the methods that approximate the derivatives of V are far more accurate than the methods that approximate V alone. In the second panel, Euler residuals are higher for this objective function than for the other two, although they decrease to 10^{-2} (i.e., 1%) when the number of neurons in two hidden layers increases to 64×64.

4.6. Comparing decision rules

In Figure 6, we plot decision rules for the three methods and simulated series for wealth.

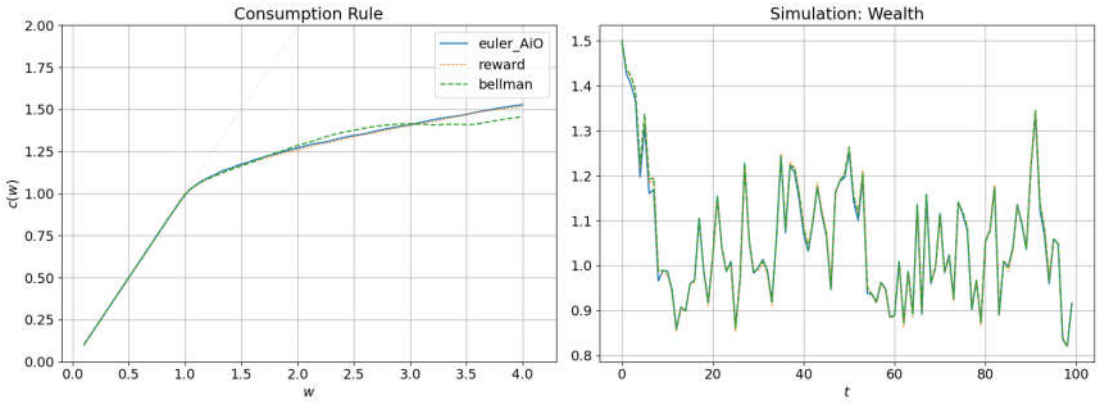

Figure 6. Comparison of methods in the consumption-saving model.

We observe that the decision rule for the Bellman method appears to be less accurate than the other two methods, however, simulated series are remarkably close to one another across the three methods. What is surprising that the three methods visually coincide in the area of kink. The second panel shows that there is a substantial variability in wealth (cash-on-hand) over a simulation of 100 periods.

4.7. Euler-equation method: sensitivity results

We now limit attention to the Euler-equation method and consider how its performance is affected by specific techniques used. In Figure 7, we plot the convergence of the objective function.

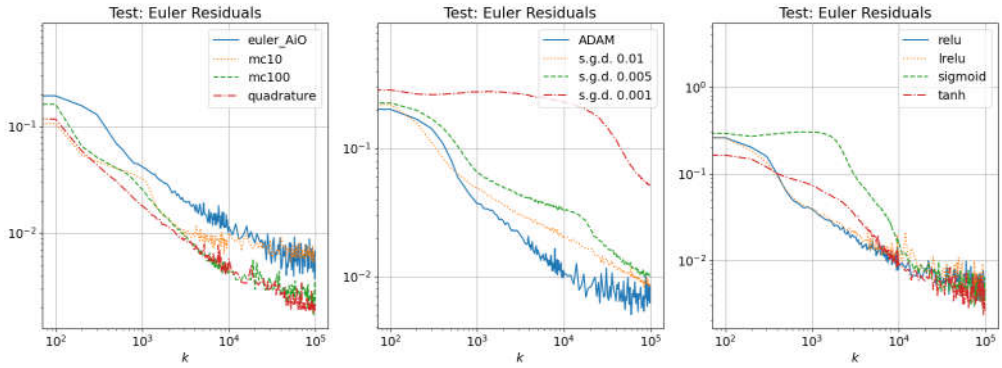

Figure 7. Comparison of integration, SGD and activation methods.

In the first panel, we show the results in which we vary an integration method, namely, we consider the AiO method with 2 nodes, Monte Carlo (MC) method with 10 random draws, MC method with 100 random draws and very accurate Gauss Hermite quadrature with 10 nodes. We observe that the integration method we use plays a visible role in convergence. A very accurate quadrature method leads to the fastest convergence while low accuracy MC leads to slower convergence than the AiO method. In the second panel, we consider four training methods, namely, SGD with an updating parameter $\lambda \in \{0.01, 0.005, 0.001\}$ and ADAM. We see that ADAM leads to the fastest convergence even though the convergence is noisier at the end with ADAM than with the other methods. Finally, in the last panel, we compare the convergence with different activation functions, namely, relu, lrely, sigmoid and tanh, and we document that they lead to comparable results.

In Figure 8, we illustrate the scalability of the Euler-equation method. TensorFlow has a possibility of using batches where the same estimation is ran in parallel on multiple virtual cores. We vary the batch size from $N = 8$ to $N = 8,192$ draws and we document running time and Euler-equation residuals.

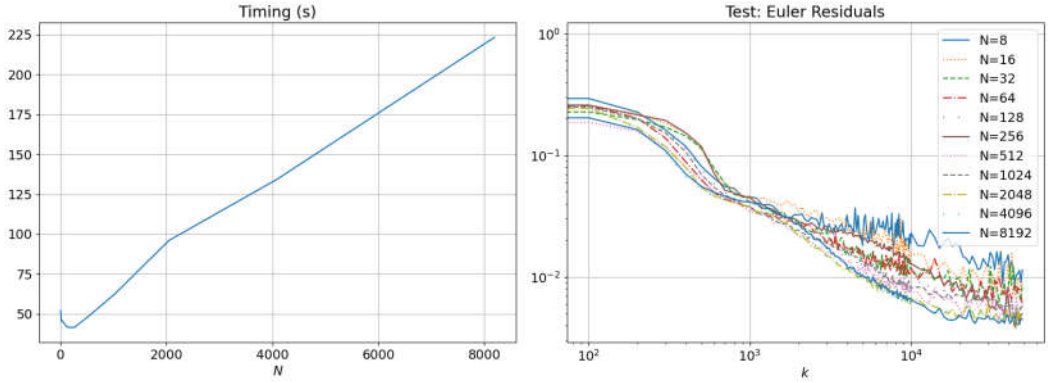

Figure 8. Effects of the batch size on the accuracy and speed.

In the left panel, we see that the training time changes roughly linearly with the batch size. In the right panel, we observe that a larger batch size leads to more accurate solutions although the convergence rate is similar for all batch sizes. The same is true for all the methods built on the AiO operator.

4.8. Multicollinearity, ill-conditioning and model reduction

We consider a version of the consumption-saving problem with multiple shocks,

$$\max_{\{c_t, w_{t+1}\}_{t=0}^{\infty}} E_0 \left[\sum_{t=0}^{\infty} \beta^t e^{\chi_t} u\left(c_t\right) \right] \tag{33}$$

$$\text{s.t. } w_{t+1} = r e^{\varrho_t} \left(w_t - c_t\right) + e^{y_t} e^{p_t}, \tag{34}$$

$$c_t \leq w_t, \tag{35}$$

where $\{y_t, p_t, \varrho_t, \chi_t\} \equiv z_t$ is a vector of exogenous state variables, which includes a temporary income shock y_t, a long-lasting income shock p_t, an interest-rate shock ϱ_t and a preference shock χ_t. We assume that each exogenous state variable $z_j \in \{y, p, \varrho, \chi\}$ follows an AR(1) process,

$$z_{j,t+1} = \rho_j z_{j,t} + \sigma_j \epsilon_{j,t} \text{ and } \epsilon_{j,t} \sim \mathcal{N}\left(0,1\right), \tag{36}$$

where $\left|\rho_j\right| < 1$ and $\sigma_j > 0$. We parameterize the model by $u\left(c\right) = \frac{c^{1-\gamma}-1}{1-\gamma}$ with a risk-aversion coefficient of $\gamma = 2$, and we assume $\beta = 0.9$, $r = 1.04$, $\rho_y = 0.9$ and $\sigma_y = 0.1$; $\rho_p = 0.999$ and $\sigma_p = 0.001$; $\rho_\varrho = 0.2$ and $\sigma_\varrho = 0.001$; and $\rho_\chi = 0.9$ and $\sigma_\chi = 0.01$.

The lifetime-reward objective function (27) is built on the AiO operator and it is directly suitable for high-dimensional applications. In Figure 9, we present the results obtained with reward maximization for the multistate model (20)–(23). We use 64 relu nodes in each of the two hidden layers, and the training method was ADAM.

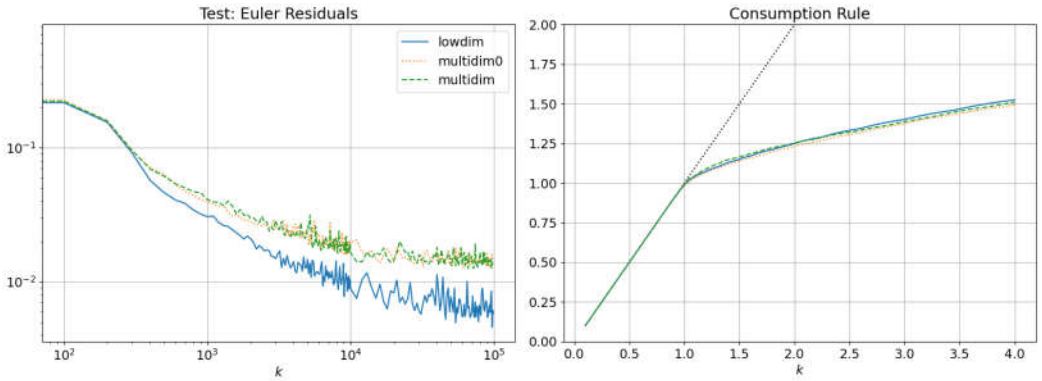

Figure 9. Multicollinearity and model reduction in the multishock model.

Three cases are considered in the figure. The main case is the multi-shock model denoted by "multidim". The other two cases, "lowdim" and "multidim0", correspond to a version of the model with only one income shock. But the two models differ in the inputs that we supply to the neural network: in the "lowdim" model, the irrelevant shocks other than income shock are not supplied at all, while in "multidim0", they are supplied to the neural network by setting all of them equal to zero. Thus, the latter model has perfect multicollinearity, so that the inverse problem is ill-conditioned and cannot be solved with conventional regression or approximation methods, such as ordinary-least squares (OLS).

There are two main results to learn from this experiment: First, neural-network approximations do not suffer from multicollinearity and ill conditioning, unlike the conventional polynomial approximation. Training of the model with zero shocks leads to the same solution and has roughly the same convergence rate as those of the other two models. This experiment illustrates how neural networks can do the model reduction: they learn to ignore the effect of nonexisting shocks although at some additional initial cost (i.e., the residuals of the last model are slightly larger in the beginning of training than those of the other models). Second, training in the multi-shock model has approximately the same convergence rate as that of the other models. The cost of iteration in the multi-shock model is slightly larger than in the unishock model but this difference is relatively small. This finding indicates that the proposed solution method is potentially tractable in problems with high dimensionality.

5. Numerical analysis of Krusell and Smith's (1998) model

We now use our deep learning method to solve Krusell and Smith's (1998) model. We formulate the model in terms of cash-in-hand to make it comparable to the consumption-saving problem studied in Section 4.

5.1. Krusell and Smith's (1998) model

The economy consists of a set of heterogeneous agents $i = 1, ..., \ell$ that are identical in fundamentals but differ in productivity and capital. Each agent i solves

$$\max_{\left\{c_t^i, k_{t+1}^i\right\}_{t=0}^{\infty}} E_0\left[\sum_{t=0}^{\infty}\beta^t u\left(c_t^i\right)\right] \tag{37}$$

$$\text{s.t. } w_{t+1}^i = R_{t+1}\left(w_t^i - c_t^i\right) + W_{t+1}\exp\left(y_{t+1}^i\right), \tag{38}$$

$$c_t^i \leq w_t^i, \tag{39}$$

where c_t^i, w_t^i, y_t^i, R_t, W_t and $k_{t+1}^i = w_t^i - c_t^i$ are consumption, cash-on-hand, labor productivity, interest rate, wage and next-period capital, respectively. Initial condition $\left(y_0^i, w_0^i\right)$ is given. The individual productivity evolves as

$$y_{t+1}^i = \rho_y y_t^i + \sigma_y \epsilon_t^i \text{ with } \epsilon_t^i \sim \mathcal{N}(0,1). \tag{40}$$

The production side of the economy is described by a Cobb-Douglas production function $z_t k_t^{\alpha}$, where $\alpha \in (0,1)$ and z_t is an aggregate productivity shock,

$$z_{t+1} = \rho z_t + \sigma \epsilon_t \text{ with } \epsilon_t \sim \mathcal{N}(0,1). \tag{41}$$

Initial condition z_0 is given. The equilibrium prices are

$$R_t = 1 - d + z_t \alpha k_t^{\alpha-1}\left[\sum_{i=1}^{\ell}\exp\left(y_t^i\right)\right] \text{ and } W_t = z_t(1-\alpha)k_t^{\alpha}\left[\sum_{i=1}^{\ell}\exp\left(y_t^i\right)\right], \tag{42}$$

where $k_t = \sum_{i=1}^{\ell} k_t^i$ is aggregate capital, and $d \in (0,1]$ is the depreciation rate. Note that (38) implies that $w_t^i = R_t k_t^i + W_t \exp\left(y_t^i\right)$. We parametrize the model by $u(c) = \frac{c^{1-\gamma}-1}{1-\gamma}$ with a risk-aversion coefficient of $\gamma = 1$ and assume $\beta = 0.96$, $\rho = 0.95$, $\sigma = 0.01$, $\rho_y = 0.9$, and $\sigma_y = 0.2\left(1-\rho_y^2\right)^{1/2}$.

5.2. Deep learning solution algorithm

Our analysis of Krusell and Smith's (1998) model parallels that of the consumption-saving problem of Section 4. We again construct the solution using the lifetime reward, Euler and Bellman objectives.

State space. The state space consists of the state variables of all agents $\left\{y_t^i, w_t^i\right\}_{i=1}^{\ell}$, as well as the aggregate shock z_t. Since agents are homogeneous in fundamentals, as in Krusell and Smith (1998), we need just one $2\ell+1$–dimensional decision and value functions to characterize the choices of all ℓ agents.[6] If agents were heterogenous in fundamentals, we would need to construct separate decision and value functions for each heterogenous agent; each of such functions has $2\ell+1$ dimensions.

Parameterization. Like in the consumption-saving problem, we parametrize the consumption to wealth ratio $\frac{c_t^i}{w_t^i}$, unit-free Lagrange multiplier h_t^i and value function V_t^i:

$$\begin{aligned}
\frac{c_t^i}{w_t^i} &= \sigma\left(\zeta_0 + \eta\left(y_t^i, w_t^i, D_t, z_t; \vartheta\right)\right) \equiv \varphi(\cdot;\theta), \\
h_t^i &= \exp\left(\zeta_0 + \eta\left(y_t^i, w_t^i, D_t, z_t; \vartheta\right)\right) \equiv h(\cdot;\theta), \\
V_t^i &= \zeta_0 + \eta\left(y_t^i, w_t^i, D_t, z_t; \vartheta\right) \equiv V(\cdot;\theta),
\end{aligned}$$

where $\eta(\cdot;\vartheta)$ is a neural network, $D_t \equiv \left\{y_t^i, w_t^i\right\}_{i=1}^{\ell}$ is the distribution, $\theta \equiv (\zeta_0, \vartheta)$ and $\sigma(x) = \frac{1}{1+e^{-x}}$.[7]

[6] Since the true decision and value functions are invariant to any permutation of $\left\{y_t^i, w_t^i\right\}_{i=1}^{\ell}$, neural networks should eventually learn this symmetry. In general, a faster learning speed could be achieved if the symmetry is imposed in the solution method. For instance, because of unequal initial asset levels, some agents are given higher weight in the objective functions than the others. By reshuffling randomly the positions of agents, we can prevent overfitting during the training.

[7] We do not use a recursive representation for Krusell and Smith's (1998) model but keep the time subscripts which is more appropriate for describing a solution constructed on stochastic simulation.

A sigmoid transformation of $\varphi(\cdot;\theta)$ ensures that $\frac{c_t^i}{w_t^i}$ is in the interval $[0,1]$; the exponentiation of h_t ensures that it is nonnegative; there is no restriction on the range of $V(\cdot;\theta)$. The parameter ζ_0 is calibrated, and the biases and weights are initialized randomly by using "*he*" and "*glorot*" uniform distributions, respectively. In the baseline case, we use a neural network with a sigmoid activation function and two hidden layers of 64×64 neurons.

Simulation. Our implementation of the deep learning solution method follows Algorithm 1 we used for the consumption-saving problem. The algorithm is remarkably straightforward: we simulate a panel of heterogeneous agents forward and train their decision functions as we go. For the given economy's state $(\{w^i, y^i\}_{i=1}^{\ell}, z)$ and neural network coefficients $\theta \equiv (\zeta_0, \vartheta)$, we do the following calculations to implement the simulation step 2i):

1. Compute $\frac{c_t^i}{w_t^i} = \varphi\left(y_t^i, w_t^i, D_t, z_t; \theta\right)$ and $k_{t+1}^i = w_t - c_t$ for each agent $i = 1, ..., \ell$.
2. Draw y_{t+1}^i for $i = 1, ..., \ell$ and z_{t+1} using (40) and (41), respectively.
3. Compute prices R_{t+1} and W_{t+1} from (42) given $k_{t+1} = \sum_{i=1}^{\ell} k_{t+1}^i$.
4. Compute next period cash-in-hand $w_{t+1}^i = R_{t+1} k_{t+1}^i + W_{t+1} \exp\left(y_{t+1}^i\right)$.
5. Compute $\frac{c_{t+1}^i}{w_{t+1}^i} = \varphi\left(y_{t+1}^i, w_{t+1}^i, D_{t+1}, z_{t+1}; \theta\right)$ for $i = 1, ..., \ell$.
6. Evaluate the objective function (lifetime reward, Euler and Bellman residuals), train the neural networks and go to next iteration.

We simulate the model over $K = 300,000$ periods, however, we perform the training only each 10th period.[8] In each iteration, we use 100 simulated points in each iteration. We use ADAM with the learning rate of $\lambda = 0.001$. As the machine is trained and the panel is simulated, the decision functions are refined jointly with the ergodic distribution.

Perfect multicollinearity. To parameterize the decision and value functions, we represent the state space as $(y_t^i, w_t^i, \{y_t^i, w_t^i\}_{i=1}^{\ell}, z$ i.e., we list the individual variables twice, as the state variables of a agent i and as a part of the distribution $\{y_t^i, w_t^i\}_{i=1}^{\ell}$. A repetition implies perfect multicollinearity in explanatory variables, so that the inverse problem is not well defined. Such a multicollinearity would break down a conventional numerical method which solves the inverse problem but neural networks can learn to ignore redundant colinear variables, as we saw earlier. Even though it is possible to design a transformation that avoids a repetition of variables, it would require cumbersome permutations. We find it easier to keep the repeated variables.

Model reduction. We solve the models with $\ell = 1,000$ agents which corresponds to $2\ell + 1 = 2,001$ state variables. How can the DL method deal with such a huge state space? In addition to focusing on the ergodic set, cheap AiO integration and stochastic optimization, we invoke the remarkable property of neural networks to perform model reduction. When we supply a large number of state variables to the input layer, the neural network condenses the information into 64 neurons in two hidden layers, making it more abstract and compact. In a sense, this procedure is similar to the photo compression or principal component transformation when a large set of variables is condensed into a smaller set of principal components without losing essential information; see Goodfellow et al. (2016) for a discussion of neural networks.

5.3. Lifetime reward

The objective function for the lifetime reward for the model (37)–(41) follows directly from our general exposition (8):

$$\Xi(\theta) = E_{\omega}\left[\xi(\omega;\theta)\right] \equiv E_{(Y_0, W_0, z_0, \mathbf{\Sigma}, \boldsymbol{\epsilon})}\left[\sum_{t=0}^{T} \beta^t u(c_t)\right], \tag{43}$$

where the transitions are determined by (38)–(41); $Y_0 = \left(y_0^1, ..., y_0^{\ell}\right)$, $W_0 = \left(w_0^1, ..., w_0^{\ell}\right)$ and z are the economy's state produced over stochastic simulation; $\mathbf{\Sigma} \equiv \left(\epsilon_1^1, ..., \epsilon_1^{\ell}, ..., \epsilon_T^1, ..., \epsilon_T^{\ell}\right)$ represents shocks to productivity of all heterogenous agents $i = 1, .., \ell$ over the periods $t = 1, .., T$; and $\boldsymbol{\epsilon} = (\epsilon_1, ..., \epsilon_T)$ is the sequence of innovations to aggregate productivity.

There is an important conceptual question on how to train the objective function (43). We are solving for competitive equilibrium so we must maximize the utility of each agent with respect to her own variables but not with respect to

[8] Since random variables are autocorrelated in our model, the stochastic gradient is correlated over time and hence, it is biased. To reduce the bias, we train the model on cross-sections which are sufficiently separated in time instead of using all consecutive periods.

variables of other agents. In practice, we achieve this by "muting" in TensorFlow the gradient of the objective function of a given agent with respect to variables of the other agents.

Figure 10 displays the outcome of training of neural network on the objective function (43).

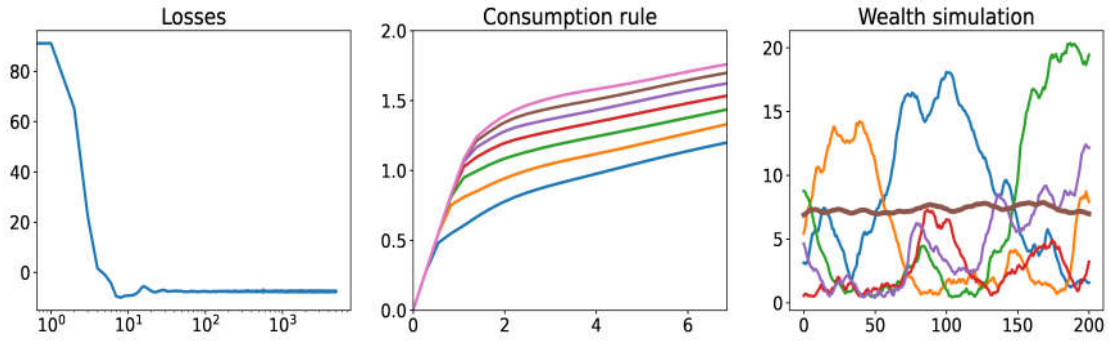

Figure 10. Lifetime reward in Krusell and Smith's (1998) model.

In the first panel, we illustrate how the loss function changes during training; in second panel, we show the consumption functions depending on individual wealth under 7 different productivity levels ranging from -2 to 2 of the standard deviations of productivity (to produce these decision rules, we set the aggregate state and productivity of all other agents to their steady state levels). Finally, in the last panel, we show a simulated wealth series for 5 heterogeneous agents selected randomly from the sample. We see that the consumption decision function is similar to the one shown in Figure 3 for the one-consumer problem. We also observe that the simulated series for wealth are stationary; they fluctuate within a reasonable range, occasionally reaching the borrowing limit.

5.4. Euler-equation method with Kuhn-Tucker conditions

The Euler objective function for Krusell and Smith's (1998) model is parallel to the objective (30) of the consumption-saving problem. Using the technique of two uncorrelated shocks (11) to facilitate the AiO operator, we obtain:

$$\Xi(\theta) = E_{\omega}\left[\xi\left(\omega;\theta\right)\right] = E_{(Y_t,W_t,z_t,\Sigma_1,\Sigma_2,\epsilon_{1,},\epsilon_2)}\left\{\left[\Psi^{FB}\left(1-\frac{c_t^i}{w_t^i},1-h_t^i\right)\right]^2 + v\left[\frac{\beta R_{t+1}\ u'\left(c_{t+1}^i\right)\big|_{\Sigma=\Sigma_1,\epsilon=\epsilon_1}}{u'\left(c_t^i\right)}-h_t^i\right]\left[\frac{\beta R_{t+1}\ u'\left(c_{t+1}^i\right)\big|_{\Sigma=\Sigma_2,\epsilon=\epsilon_2}}{u'\left(c_t^i\right)}-h_t^i\right]\right\}. \quad (44)$$

where the transitions are determined by (38)–(41); $Y_t = \left(y_t^1,...,y_t^\ell\right)$ and $W_t = \left(w_t^1,...,w_t^\ell\right)$ and z_t are the economy's state produced stochastic simulation; $\Sigma_1 = \left(\epsilon_1^1,...,\epsilon_1^\ell\right)$, $\Sigma_2 = \left(\epsilon_2^1,...,\epsilon_2^\ell\right)$ are two uncorrelated random draws of individual productivity shocks; and $\epsilon_{1,}$, ϵ_2 are two uncorrelated random draws for the aggregate productivity innovations.

The results of training under the objective (44) are shown in Figure 11.

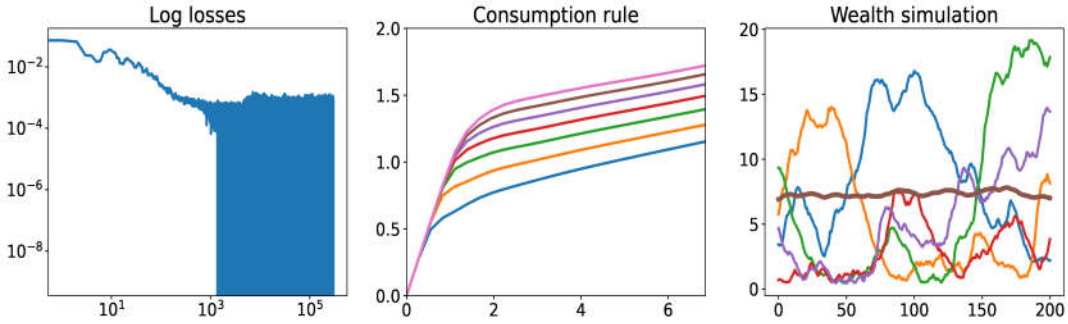

Figure 11. Euler-equation method in Krusell and Smiths (1998) model.

For the Euler-equation method, the residuals are generally below 10^{-3} (i.e., a fraction of a percentage point). The solutions are very similar to those produced by the lifetime reward maximization method and presented in the previous section.

5.5. Objective 3: Bellman equation

The Bellman objective function for Krusell and Smith's (1998) model is also parallel to the objective function (32) derived for the consumption-saving problem. In particular, using the technique of two uncorrelated shocks (11), we then arrive to an objective function with the AiO operator:

$$\Xi(\theta) = E_{\omega}\left[\xi\left(\omega;\theta\right)\right] \equiv E_{\left(Y_t,W_t,z_t,\Sigma_1,\Sigma_2,\epsilon_1,,\epsilon_2\right)} \left\{ \left[V\left(s_t^i;\theta\right) - u\left(c_t^i\right) - \beta\, V\left(s_{t+1}^i;\theta\right)\Big|_{\Sigma=\Sigma_1,\epsilon=\epsilon_1}\right] \right.$$
$$\times \left[V\left(s_t^i;\theta\right) - u\left(c_t^i\right) - \beta\, V\left(s_{t+1}^i;\theta\right)\Big|_{\Sigma=\Sigma_2,\epsilon=\epsilon_2}\right] + v\left[\Psi^{FB}\left(1-\frac{c_t^i}{w_t^i}, 1-h_t^i\right)\right]^2$$
$$\left. + v_h \left[\frac{\beta\,\frac{\partial}{\partial w_{t+1}^i} V\left(s_{t+1}^i;\theta\right)\Big|_{\Sigma=\Sigma_1,\epsilon=\epsilon_1}}{u'\left(c_t^i\right)} - h_t^i\right]\left[\frac{\beta\,\frac{\partial}{\partial w_{t+1}^i} V\left(s_{t+1}^i;\theta\right)\Big|_{\Sigma=\Sigma_2,\epsilon=\epsilon_2}}{u'\left(c_t^i\right)} - h_t^i\right]\right\}, \quad (45)$$

where all the ingredients are defined as in (44).

The results of training under the objective (45) are shown in Figure 12.

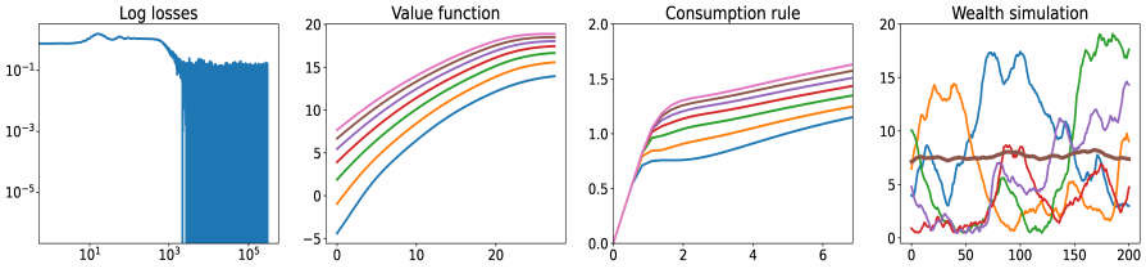

Figure 12. Bellman-equation method in Krusell and Smith's (1998) model.

To initialize the algorithm, we pre-train the value function during the first 100,000 iterations holding initial decision functions for consumption and multiplier fixed – this explains the initial flat area in the loss function. In the figure, we also plot the resulting value function under 7 different individual productivity levels ranging from -2 to 2 of the standard deviations of productivity. The constructed decision functions and simulated series look very similar to those produced by the two previous methods.

5.6. Comparison of the solutions produced by three methods

To make a more conclusive judgement, we compare the decision rules produced by the three methods.

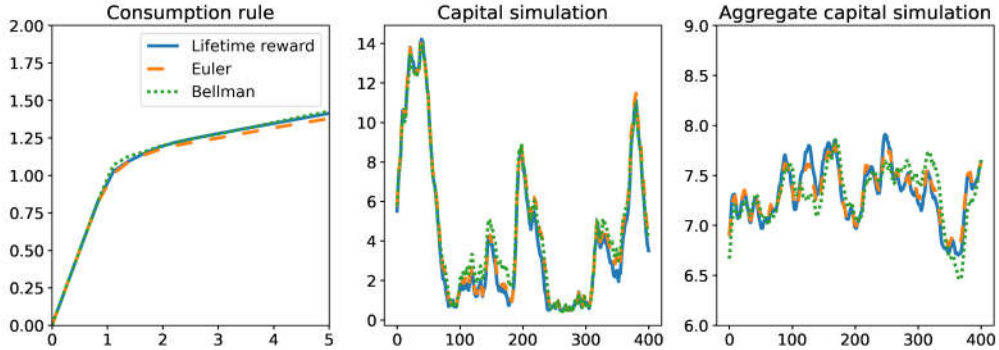

Figure 13. Comparison of methods in Krusell and Smith's (1998) model.

In the first panel, we plot the decision rules of one agent for the thee methods by assuming that all other individual and aggregate state variables are in the steady state. In the second panel, we show simulation of individual wealth for all three methods under an identical sequence of shocks and in the last panel, we show simulation of aggregate wealth. The constructed decision rules and time series solutions are visually close, and they produce very similar statistics such as the first and second moments. We should take into account that some of the differences between these three solutions are explained by randomness that is innate to stochastic optimization, namely, the constructed decision rules may somewhat

fluctuate along iterations depending on the specific sequence of random shocks.

5.7. Euler-equation method: the properties of the solution

In Table 1, we present some statistics for the model with different number of agents produced by the Euler-equation version of our solution method.

ℓ	$std(y)$	$corr(y,c)$	$Gini(k)$	*Bottom* 40%	*Top* 20%	*Top* 1%	*Time, sec.*	R^2
1	1.69	0.862	-	-	-	-	522	0.9837
5	1.69	0.681	0.403	0.143	0.446	0.034	678	0.9910
10	1.64	0.671	0.443	0.115	0.469	0.037	805	0.9934
50	1.64	0.681	0.447	0.113	0.473	0.036	1721	0.9898
100	1.66	0.708	0.430	0.123	0.460	0.036	3297	0.9936
500	1.63	0.699	0.438	0.119	0.467	0.037	21823	0.9965
1000	1.66	0.707	0.430	0.118	0.465	0.037	43241	0.9977

Table 1. Selected statistics for Krusell and Smith's (1998) model.

In the first column, we show that all the studied models have practically the same standard deviation of output equal to $std(y) \approx 1.64$. This is because we normalize the mean of individual shocks to one in every period to eliminate the effect on idiosyncratic shocks on the aggregate economy. In the second column, we provide the correlation between output and aggregate consumption, which visibly decreases with the number of agents. There is literature that tries to understand why real business cycle models overstate the correlation between these two variables and our analysis suggests that the heterogeneity can be a clue.

Columns 3–6 report the Gini coefficient of the wealth distribution and the share of income by quantiles. Here, the numbers are comparable across the models and are similar to those obtained in Aiyagari (1994). This fact is not particularly surprising since our calibration closely follows the one used in that paper. We differ from that paper in that we also introduce the aggregate shocks but this is not a sufficiently strong mechanism to change the distributional implications of the baseline Aiyagari's (1994) model.

Column 7 reports the running time. We see that the time varies from 522 to $43,241$ seconds which is not prohibitively large. We conclude that much larger models can be solved using a more powerful hardware beyond a laptop. In fact, the bottleneck is actually not the running time but memory: manipulating large neural networks becomes increasingly expensive as the number of agents increases.

Finally, column 8 contains the most interesting and controversial statistic which is R^2 of Krusell and Smith's (1998) style regression:

$$\ln(k_{t+1}) = \xi_0 + \xi_1 \ln(k_t) + \xi_2 \ln(z_t), \tag{46}$$

i.e., a regression of aggregate capital on the past aggregate capital and aggregate productivity; see Den Haan (2010) for a discussion. Krusell and Smith (1998) find that R^2 in their model was in excess of 0.99999, which means that the aggregate capital k_{t+1}, and hence, the prices, can be accurately predicted by using just aggregate state variables k_t and z_t. This result is referred to as *approximate aggregation*. In Table 1, we see that R^2 is also relatively large, e.g., it is in excess of 0.98 for all models. However, it is not as large as the one reported by Krusell and Smith (1998) and other papers that implemented related methods, e.g., Maliar et al. (2010).

However, our analysis is not exactly identical to the one studied by Krusell and Smith (1998). They had two aggregate shocks and solve for two state-contingent rules $\ln(k_{t+1}) = \xi_0^g + \xi_1^g \ln(k_t)$ and $\ln(k_{t+1}) = \xi_0^b + \xi_1^b \ln(k_t)$, where "$g$" and "$b$" denote the good and bad aggregate-productivity states. In their state-contingent regressions, the sampling errors are associated only with the aggregate capital. We have a more complicated setup with a continuum of aggregate states. Our sampling errors in (46) are driven by both the aggregate capital and aggregate productivity. Possibly, if we split the data by the level of aggregate productivity to mimic Krusell and Smith's (1998) state-contingent regressions, we would get R^2 which is closer to theirs.

5.8. Deep learning method with a reduced state space

To reduce the computational expense, Krusell and Smith (1998) came up with a simple and effective idea of replacing the distributions of state variables $D_t \equiv \left\{y_t^i, w_t^i\right\}_{i=1}^{\ell}$ by a set of moments m_t. To implement this idea, they designed a fixed-point iterative procedure that alternates between constructing the individual decision rules on a grid (by taking the law of motion of the moments as given) and solving for the law of motion for the moments (by taking the individual

decision rules as given). An essential part of their solution method is a regression of moments on lagged moments shown in Table 1. If such regression has a high explanatory power, the agents can accurately predict future prices without knowing the distributions which reduces the individual state space to just four state variables $\left(y_t^i, w_t^i, m_t, z_t\right)$ if only first moment is used. Krusell and Smith's (1998) approach proved to work remarkably well in a variety of models and applications.[9]

Our deep learning framework provides a simple way to incorporate Krusell and Smith's (1998) idea of using a reduced state space. As before, we simulate a panel of heterogeneous agents but we feed moments in the decision functions instead of distributions. Under our implementation, there is no need to alternate between constructing the individual and aggregate solutions as we solve the entire model at once. Moreover, the regression step of Krusell and Smith's (1998) analysis is also unnecessary, which allows us to use our method in those models in which the explanatory power of such regression is insufficient. Our deep learning method admits any other statistics instead of and in addition to moments and provides a cheap alternative to our baseline deep learning operating on the actual state space. Having relatively few moments, like 10 or 20, makes our method tractable even with larger number of agents, in particular, we are able to increase the number of agents to 10,000. To save on space, we do not show the results since the statistics and figures are practically the same as in the baseline case.

5.9. Deep learning versus moments

Krusell and Smith (1998) discovered that a single statistic – the mean of the wealth distribution – can effectively characterize the state space of their heterogeneous-agent economy. An interesting question is how their solution compares to our analysis. To address this question, we compare a solution produced when using the first moment of wealth distribution (KS1) with our baseline solution produced when using the actual state space (64×64 neurons). We find that the one-moment KS1 solution is systematically shifted up relative to our baseline 64×64 solution (although the difference is not quantitatively important). We then compute a deep learning solution with two hidden layers composed of 64 and 4 neurons, respectively, so that the second hidden layer has same dimensionality as the one-moment solution. We find that the 64×4–neuron solution is also shifted up near the kink but it gradually approaches our reference solution for larger levels of capital; see Figure 14.

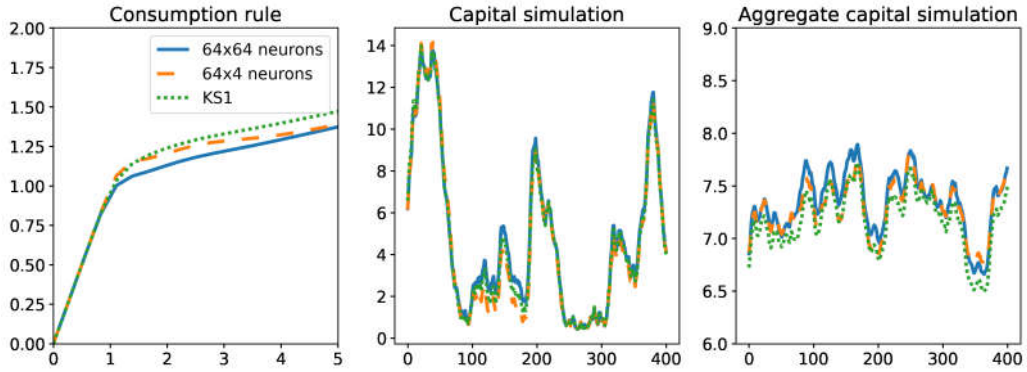

Figure 14. Comparison of one-moment, 64×64 and 64×4 neuron solutions.

We tried to add second and third moments but it did not help remove the shift (we explore both ordinary and orthogonal Hermite polynomials). In fact, a Krusell and Smith's (1998) solution constructed using two moments of the wealth distribution, KS2, was even further away from our reference 64×64 solution than the one-moment solution KS1. We next consider a solution with seven neurons in the hidden layer 64×7 which has the same dimensionality as Krusell and Smith's (1998) solution with first and second moments KS2, and we find that additional neurons do help get closer to the reference solution, see Figure 15.

[9] Other early approaches to solving heterogeneous agent models are offered by Den Haan (1997), Judd (1997) and Reiter (2009); see Den Haan (2010) for a review.

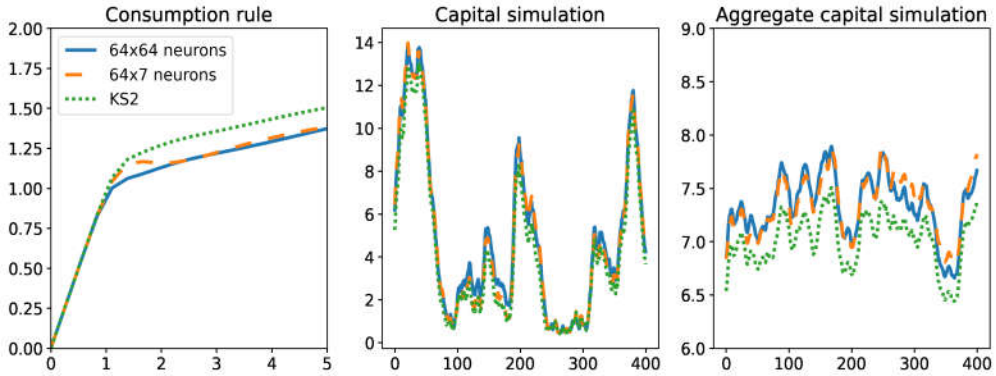

Figure 15. Comparison of two-moment, 64×64 and 64×7 neuron solutions.

Why does adding neurons help but adding more moments does not? This is because the second moments do not have additional explanatory power relative to the first moment, for example, they do not increase R^2 of Krusell and Smith's (1998) regression. Adding such redundant moments does not increase the fit but does increase the variance In contrast, neural network is designed to perform model reduction: it considers many possible ways of extracting the information contained in the distribution and condensing it into a relatively small set of hidden layers. Not surprisingly, it can find representations that are more efficient than a set of moments postulated ad-hoc and it can use additional neurons to improve on the quality of representation.

6. Discussion: Deep learning in economic dynamics

In this section, we discuss a broad set of issues related to applying deep learning in economic dynamics and explain how these issues differ from those arising in data science.

The main novelty: computational technology! There is not much methodological novelty in the canonical AI analysis. The gradient descent method is known since Newton and its stochastic optimization version was developed in early 1950s. Neural networks were also discovered long ago (Rosenblatt, 1958). There are even remarkable early applications of neural networks to solving dynamic economic models (Duffy and McNelis, 2001).

However, machine learning is so technology intensive that DL methods were put aside until platforms like TensorFlow and Pytorch were developed to facilitate their implementation. Google has developed its own TPU units too. High-end GPUs feature thousands of powerful CUDA cores, which can all operate at the same time. Also, commercial interest is high and the surge of cloud computing has made it possible to rent vast amounts of computing power, e.g., Amazon.

Learning is always deep in economic models. In data science, learning is called "deep" because neural networks have interconnected topologies with multiple layers of coefficients. In economic dynamics, there is another reason for calling learning *"deep"*. Specifically, objective functions derived from dynamic economic models contain variables of multiple periods and nested decision functions which lead to multilayer interconnected topologies and approximation coefficients buried in several layers of intertemporal optimization, similar to those observed in multilayer neural network. In that sense, learning is always "deep" in dynamic economic models, even for simple one-layer approximating functions such as polynomial or piecewise linear functions.

Data are truly random in economic dynamics. In data science, a data set is usually fixed and batches are not really random but a pseudo-random bootstrap of the given data. In such applications, we split the available data into 3 samples, namely, for construction of a solution, validation and accuracy assessment. When solving dynamic models, we do not use real-world economic data but generate artificial data by simulating the model. In this case, we just need to fix the distribution for random draws. We then are able to simulate the model at will and we can generate as much artificial data as we want. In that sense, our data are truly random.

Antithetic variates improve efficiency of Monte Carlo integration. Monte Carlo integration has a low square-root rate of convergence. We can increase the efficiency of SGD by using variance reduction techniques; see Cheng (1982). One simple method is antithetic variates: assuming a zero mean, for every realization $(\omega_1, ..., \omega_{n'})$, we also consider its antithetic realization $(-\omega_1, ..., -\omega_{n'})$. Another possibility is a tensor-product antithetic variates, i.e., to

consider all possible combinations $(\pm\omega_1, ..., \pm\omega_{n'})$. In the lifetime reward function, the sequence $(\pm\epsilon_1, ..., \pm\epsilon_T)$ may be expensive to analyze, so we can consider a truncated sequence with antithetic variates just for the first τ periods, namely, $(\pm\epsilon_1, ..., \pm\epsilon_\tau, \epsilon_{\tau+1}, ..., \epsilon_T)$. The distant future terms are discounted so that making the first few draws antithetic can still bring a considerable increase in accuracy.

Lifetime reward maximization with deterministic integration of shocks. For the Euler- and Bellman-equation methods, we had two versions: one in which we compute integrals with respect to state variables and future shocks separately and the other is the AiO expectation version. For the lifetime reward, we only have the version with the AiO expectation operator. But we can also implement a method that computes the two expectation functions separately, for example, one that uses Monte Carlo integration for evaluating $E_{(m_0,s_0)}[\cdot]$, and the other that uses deterministic integration for evaluating $E_{(\epsilon_1,...,\epsilon_T)}$, i.e., we compute sequentially the two expectation operators in $E_{(m_0,s_0)}\left[E_{(\epsilon_1,...,\epsilon_T)}\sum_{t=0}^{T}\beta^t r(\cdot)\right]$. To evaluate $E_{(\epsilon_1,...,\epsilon_T)}$, we can use a deterministic integration method suggested in Adjemian and Juillard (2013). In each period t, we construct n integration nodes (by using quadrature, monomials, etc.) This leads to an exploding tensor-product tree. For example, a tree with just 2 Gaussian nodes $\pm\epsilon$, results in sequence $\pm\epsilon_0, \pm\epsilon_1, \pm\epsilon_2, ...$ that has exponentially growing number of nodes 2, 4, 8,.... But Adjemian and Juillard (2013) propose a clever refinement that makes the problem tractable by eliminating those branches of the tree whose probabilities are low.

Nested local approximations and deterministic models. Each of the three constructed objective functions contains two nested models: One is the model in which the solution is approximated locally around the given state, and the other is the deterministic model. In the former case, state (m, s) is fixed in (8), (10), (12), (14), and (15) and expectation is computed only with respect to exogenous shocks ϵ. Such a solution may be interesting per se, for example, for studying transitions of a developing economy to the steady state because the solution constructed just in the ergodic set may be insufficiently accurate. In turn, the deterministic model is the one in which a realization of shocks ϵ is fixed and the state (m, s) is random.

Accuracy tests in economic dynamics are indirect. In a canonical supervised learning regression, we assess the quality of approximation by looking at the difference between the true and predicted output out-of-sample. In a logistic regression (e.g., the problem of handwritten digits classification), the success can be also measured by the fraction of times the machine classifies the digits correctly, e.g., it recognizes the handwritten digits correctly in 80 percent of the cases.

In the context of economic dynamics, a parallel direct accuracy test would require us to compare approximate and exact solutions. This is generally infeasible since the exact solutions are unknown. One way to deal with this complication is to construct a more accurate reference solution by using more flexible approximation functions, more precise solvers and more accurate numerical integration methods. However, such a reference solution may be infeasible or excessively costly; see Judd et al. (2017) for a discussion and examples of cheaper direct testing methods that rely on numerical construction of the error bounds.

Following the economic literature, we concentrate on indirect approaches to the accuracy evaluation. Specifically, we will check certain properties that an accurate solution is known to satisfy, such as zero residuals in the Euler or Bellman equation. Indirect accuracy tests are simple to design and they can be implemented in an out-of-sample way which is characteristic for AI applications. Moreover, we can define indirect accuracy measures to reflect the economic significance of accuracy. For example, we can express approximation errors in percentage terms of consumption.

7. Concluding comments

In the paper, we propose an AI technology that is tractable in large scale applications – a deep learning method based on Monte Carlo simulation. Our analysis is technology driven: we do not aim to design AI approaches that would work best for a certain class of economic models but rather we adapt the economic models themselves to available AI technologies. The modern data-science tools are ubiquitous, well developed, free of errors and optimized – these may be sufficient to compensate for potential inefficiencies. We have shown the promise of the DL approach by solving the Krusell and Smith (1998) model with thousands of state variables without resorting to a simplifying assumption about the economy's state space – such analysis has been infeasible up to now. Consequently, it seems to be a promising direction to explore.

Our solution framework was designed to take advantage of existing DL technology. "Is this the best possible technology for solving dynamic economic models?" – the answer to this question is not clear.

First, neural networks are powerful universal approximators, but their training is expensive and their convergence to the solutions is not guaranteed. It is actually an open question whether there is much value in using deep neural networks for approximating decision rules in economics which often can be well approximated by simple functions like polynomials and splines.

Second, Monte Carlo simulation lying at the basis of DL framework has a low square-root rate of convergence. It is possible to improve on the Monte Carlo method by engineering sequences that deliver more accurate approximations to integrals (e.g., quadrature, monomials, quasi-random sequence, sparse grids, clusters, epsilon-distinguishable sets), as well as by applying variance-reduction techniques such as antithetic variates; see Maliar and Maliar (2014) for a review.

Third, instead of stochastic optimization, we can use other numerical solvers, e.g., fixed-point iteration, conventional GD methods, Gauss-Jacobi, Gauss-Siedel and linear programming; see Judd (1998). These techniques are commonly used in computational economics, and we expect them to be useful alternatives to our baseline SGD in some applications.

Finally, there are other AI-style methods that can be used for solving economic models, in particular, unsupervised and reinforcement learning methods. These methods offer a possibility of online learning and additional powerful approximation techniques such as alternating of learning, exploration or exploitation – such techniques are absent in our static offline supervised learning framework.

References

Adjemian, S., Juillard, M., 2013. Stochastic extended path approach. Manuscript.

Ahn, S., Kaplan, G., Moll, B., Winberry, T., Wolf, C., 2018. When inequality matters for macro and macro matters for inequality. NBER Macroeconomics Annual, University of Chicago Press, vol. 32(1), 1-75.

Arellano, C., Maliar, L., Maliar, S., and Tsyrennikov, V., 2016. Envelope condition method with an application to default risk models. Journal of Economic Dynamics and Control 69, 436-459.

Aruoba, S. B., Fernández-Villaverde, J., Rubio-Ramírez, J., 2006. Comparing solution methods for dynamic equilibrium economies. Journal of Economic Dynamics and Control 30, 2477–2508.

Azinovic, M., Luca, J., Scheidegger, S., 2020. Deep equilibrium nets. SSRN: https://ssrn.com/abstract=3393482

Bayer, C., Luetticke, R., 2020. Solving discrete time heterogeneous agent models with aggregate risk and many idiosyncratic states by perturbation. Quantitative Economics 11, 1253-1288.

Carroll, C., 2006. The method of endogenous gridpoints for solving dynamic stochastic optimization problems. Economics Letters 91(3), 312-320.

Cheng, R., 1982. The use of antithetic variates in computer simulations. Journal of the Operational Research Society 33 (3), 229-237.

Coleman, C., Lyon, S., Maliar, L., Maliar, S., 2018. Matlab, python, julia: what to choose in economics? CEPR working paper DP 13210. Computational Economics, forthcoming.

Den Haan, W., 1997. Solving dynamic models with aggregate shocks and heterogeneous agents. Macroeconomic Dynamics 1(02), 355-386.

Den Haan, W., 2010. Comparison of solutions to the incomplete markets model with aggregate uncertainty. Journal of Economic Dynamics and Control 34, 4–27.

Den Haan, W., Marcet, A., 1990. Solving the stochastic growth model by parameterized expectations. Journal of Business and Economic Statistics 8, 31–34.

Duarte, V., 2018. Machine learning for continuous-time economics. SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=30

Duffy, J., McNelis, P., 2001. Approximating and simulating the real business cycle model: parameterized expectations, neural networks, and the genetic algorithm, Journal of Economic Dynamics and Control 25(9), 1273-1303.

Fernández-Villaverde, J., Hurtado, S., Nuño, G., 2019. Financial frictions and the wealth distribution. NBER Working paper 26302.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep learning. Massachusetts Institute Technology Press.

Gorodnichenko, Y., Maliar, L., Maliar, S., Naubert, C., 2020. Household savings and monetary policy under individual and aggregate stochastic volatility. CEPR working paper 15614.

Jiang, H., 1996. Smoothed Fischer-Burmeister equation methods for the complementarity problem. Manuscript.

Jirniy, A., Lepetyuk, V., 2011. A reinforcement learning approach to solving incomplete market models with aggregate uncertainty. SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1832745

Judd, K. L., 1992. Projection methods for solving aggregate growth models. Journal of Economic Theory 58, 410–452.

Judd, K. L., 1998. Numerical methods in economics. MIT Press.

Judd, K. L., Maliar, L., Maliar, S., 2011. Numerically stable and accurate stochastic simulation approaches for solving dynamic models. Quantitative Economics 2, 173–210.

Judd, L., Maliar, L., Maliar, S., 2017. Lower bounds on approximation errors to numerical solutions of dynamic economic models. Econometrica 85 (3), 991-1020.

Lepetyuk, V., Maliar, L., Maliar, S., 2020. When the U.S. catches a cold, Canada sneezes: a lower-bound tale told by deep learning. Journal of Economic Dynamics and Control 117, 103926.

Krusell, P., Smith, A., 1998. Income and wealth heterogeneity in the macroeconomy. Journal of Political Economy 106, 868–896.

Maliar, L., Maliar, S., 2005. Parameterized expectations algorithm: how to solve for labor easily. Computational Economics 25, 269–274.

Maliar, L., Maliar, S., 2013. Envelope condition method versus endogenous grid method for solving dynamic programming problems. Economics Letters 120, 262-266.

Maliar, L., Maliar, S., 2014. Numerical methods for large scale dynamic economic models in: Schmedders, K. and K. Judd (Eds.), Handbook of Computational Economics, Volume 3, Chapter 7, Amsterdam: Elsevier Science.

Maliar L., Maliar, S., 2015. Merging simulation and projection approaches to solve high-dimensional problems with an application to a new Keynesian model. Quantitative Economics 6, 1-47.

Maliar, L., Maliar,S., 2020. Deep learning classification: Modeling discrete labor choice. CEPR working paper DP 15346.

Maliar, L., Maliar, S., Valli, F., 2010. Solving the incomplete markets model with aggregate uncertainty using the Krusell-Smith algorithm. Journal of Economic Dynamics and Control 34 (special issue), 42–49.

Reiter, M., 2010. Approximate and almost-exact aggregation in dynamic stochastic heterogeneous-agent models. IHS Working Paper 258.

Rosenblatt, F., 1958. The perceptron: a probabilistic model for information storage and organization in the brain. Psychological Review 65 (6), 386-408.

Rust, J., 1996. Numerical dynamic programming in economics. In: Amman,H., Kendrick,D., Rust, J. (Eds.), Handbook of Computational Economics. Elsevier Science,Amsterdam, pp. 619–722.

Santos, M., 1999. Numerical solution of dynamic economic models. In: Taylor, J.,Woodford, M. (Eds.), Handbook of Macroeconomics. Elsevier Science, Amsterdam, pp. 312–382.

Smith, A., 1993. Estimating nonlinear time-series models using simulated vector autoregressions. Journal of Applied Econometrics 8, S63–S84.

Stachurski, J., 2009. Economic dynamics: theory and computation. MIT Press.

Villa, A., Valaitis, V., 2019. Machine learning projection methods for macro-finance models. SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3209934.

Winberry, T., 2018. A method for solving and estimating heterogeneous agent macro models, Quantitative Economics 9(3), 1123–1151.

Supplement to "Deep learning for solving dynamic economic models".

Lilia Maliar[a], Serguei Maliar[b,*], Pablo Winant[c‡]

[a] The Graduate Center, City University of New York, CEPR and Hoover Institution, Stanford University;
[b] Santa Clara University;
[c] ESCP Business School and CREST/Ecole Polytechnique

Received 11 March 2020; Revised 16 July 2021; Accepted 19 July 2021.

A. Canonical supervised learning and its generalization

In this section, we describe canonical supervised learning framework, and we show how it can be generalized for estimating the objective functions derived from economic models.

A machine attempts to learn a function that maps inputs to output given a collection of input-output pairs. Formally, let $x \in \mathbb{R}^{d_x}$ be input data, called *features*, and let $y \in \mathbb{R}^{d_y}$ be output data. The goal of the machine is to learn an approximation function (also, referred to as a hypothesis or prediction function) $\varphi : \mathbb{R}^{d_x} \to \mathbb{R}^{d_y}$ such that, given x, the value $\varphi(x)$ provides an accurate prediction about the true output y. The function φ is selected within a given family of parametric functions $\{\varphi(\cdot;\theta) : \theta \in \mathbb{R}^{d_\theta}\}$, where θ is a parameters vector.

In order to construct φ, we minimize losses from inaccurate predictions. We define a loss function $\ell : \mathbb{R}^{d_y} \longrightarrow \mathbb{R}$ as the difference between a true output y and predicted output $\varphi(x;\theta)$ minimizing the expected loss, called *expected risk*

*Corresponding author: maliars@stanford.edu

†We are grateful to Marc Maliar for his help with writing the TensorFlow code for solving Krusell and Smith's (1998) model. We thank the editor and an annonimous referee for many useful comments and suggestions. The paper also circulated under the title "Will Artificial Intelligence Replace Computational Economists Any Time Soon?" We also received useful comments from participants of the 2018 Society for Computational Economics (CEF) Conference in Milan (invited session), 2018 Econometric Society Australasian Meeting (ESAM) in Auckland (invited talk), Oxford University, CEPREMAP, Banque de France, Panorisk Summer School, NYU Abhu Dabi, Paris Dauphine University, CREST (Ecole Polytechnique), Durham University, Santa Clara University, Rutgers University, PASC conference, Model comparison conference in Frankfurt, Complutense University of Madrid, Stanford University, Queens College, Stony Brook University, Columbia University, Deutsche Bundesbank, the Graduate Center, CUNY.

‡Lilia Maliar and Serguei Maliar acknowledge financial support from the NSF grants SES-1949413 and SES-1949430, respectively.

$\Xi(\theta)$,

$$\Xi(\theta) \equiv \int_{\mathbb{R}^{d_x} \times \mathbb{R}^{d_y}} \ell\left(\varphi\left(x;\theta\right), y\right) dP\left(x, y\right) = E_x\left[\ell\left(\varphi\left(x;\theta\right), y\right)\right], \tag{1}$$

where $P : \mathbb{R}^{d_x} \times \mathbb{R}^{d_y} \longrightarrow [0,1]$ is a joint probability distribution function, and $E_x\left[\cdot\right]$ is an expectation operator. That is, $\Xi(\theta)$ gives expected loss, for a given $\varphi\left(\cdot;\theta\right)$ with respect to P. The goal of minimizing $\Xi(\theta)$ is generally unattainable because there is incomplete information on P.

In practice, we solve the minimization problem by using a finite set of draws from P. We define an estimate of the expected risk, called *empirical risk* $\Xi^n(\theta)$,

$$\Xi^n(\theta) \equiv \frac{1}{n}\sum_{i=1}^{n} \ell\left(\varphi\left(x_i;\theta\right), y_i\right), \tag{2}$$

where $\left\{\left(x_i, y_i\right)\right\}_{i=1}^n \subseteq \mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$ is a set of n independently drawn input-output pairs. The goal of the machine is to learn θ that minimizes (2) – this step is referred to as *training*.

A well-known example of supervised learning is a linear regression model $y = \theta x$, in which we have $\varphi\left(x;\theta\right) = \theta x$ and the loss function is $\ell\left(\theta x, y\right) = \left(y - \theta x\right)^2$. However, many other approximating functions $\varphi\left(\cdot;\theta\right)$ can be used instead of linear regression model θx, in particular, it works for linear functional spaces (e.g., polynomials, splines) and nonlinear functions alike (e.g., radial basis functions, neural networks).

The above framework is called *supervised* because for each data point x_i, the machine is given correct output y_i to check its prediction $\varphi(x_i;\theta)$. We now generalize this framework to the case when correct output y_i is not given but defined implicitly in the objective function (1). Denoting an input-output pair (x, y) by ω, we rewrite the theoretical risk (1) and empirical risk (2), respectively, as

$$\Xi(\theta) = E_\omega\left[\xi\left(\omega;\theta\right)\right], \qquad \text{and} \qquad \Xi^n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \xi\left(\omega_i;\theta\right), \tag{3}$$

where $\left\{\omega_i\right\}_{i=1}^n$ denotes the given set of input-output pairs $\left\{\left(x_i, y_i\right)\right\}_{i=1}^n$ and $\xi\left(\omega,\theta\right) \equiv \ell\left(\varphi\left(x;\theta\right), y\right)$. This change of variables converts supervised learning into generic optimization of expectation functions.

The optimization framework (3) is more suitable for the purpose of solving dynamic economic models than the original supervised-learning framework because correct output values (i.e., the true values y_i of decision function $\varphi(x_i;\theta)$) are generally unknown to computational economists).

Finally, the implicit optimization framework (3) can also be viewed as a version of unsupervised learning. Generally, unsupervised learning analysis focuses on how to extract and effectively represent information available in the data, for example, by assigning data into clusters, by constructing principal components for dimensionality reduction. In terms of framework (3), an unsupervised machine attempts to learn the value of θ that captures regularities in the data ω, as specified in the objective function $\Xi(\theta)$. In that sense, our representation (3) provides a connection between supervised and unsupervised learning.

B. Neural networks

Neural networks are designed to mimic neurons in the human brain. A human neuron consists of a cell body connected to other neurons through input and output wires; an input wire, called *dendrite*, carries information, impulses of electricity, from another neuron, while an output wire, called *axon*, sends impulses to another neuron.

Perceptron. The first simplified model of a biological neuron – a perceptron – was invented by a psychologist Frank Rosenblatt (1957). An example of artificial neuron is represented in Figure S.1.

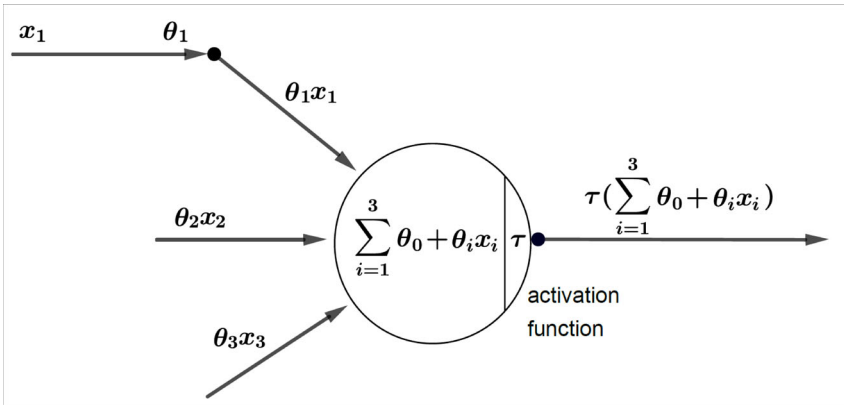

Figure S.1. An artificial neuron.

The circle represents an artificial neuron, and the arrow represents a connection from the output of one neuron to the input of another. An ith input (i.e., a received signal) is denoted by $x_i = (x_{i,0}, ..., x_{i,n})$, with $x_{i,0} = 1$ (by convention) and $n = 3$ in the figure. Signal processing consists of linearly weighing of x by a coefficients vector $\theta = (\theta_0, ..., \theta_n) \in \mathbb{R}^{n+1}$ to obtain the non-activated output $\theta x_i = \theta_0 x_{i,0} + ... + \theta_n x_{i,n}$ and activating θx with an activation function $\tau(\cdot)$, i.e., $\tau(\theta x)$. The coefficient θ_i controls the strength of a signal passing from one node to another. Under some activation functions, only signals that exceed a certain threshold are passed on to subsequent layers.

Activation functions. There is a variety of possible activation functions: (i) heaviside step function: $\tau(x) = 1_{x \geq 0}$; (ii) sigmoid (logistic): $\sigma(x) = \frac{1}{1+e^{-x}}$; (iii) tanh (hyperbolic tangent); (iv) $\tau(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$; relu (rectified linear units): $\tau(x) = \max(0, x)$; (v) leaky relu: $r(x) = \max(\kappa x, x)$, $\kappa \leq 0$; (vi) maxout: $\tau(x) = \max(a_1 x + b_1, a_2 x + b_2, a_3 x + b_3)$.[1] Figure S.2 plots three of such functions, namely, a sigmoid, hyperbolic tangent (tanh), and leaky relu.

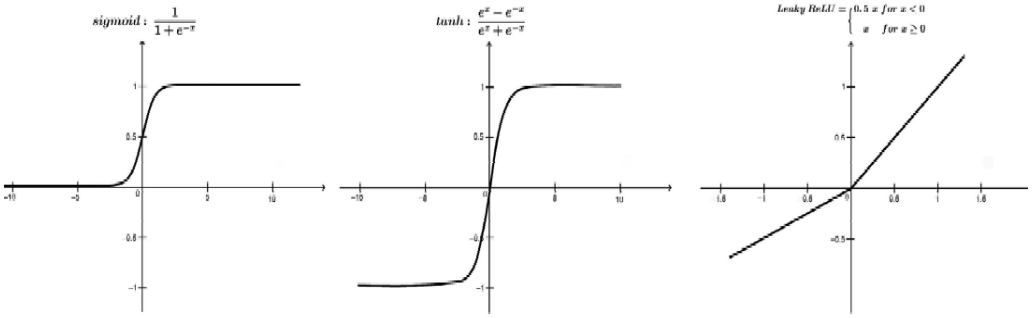

Figure S.2. Common activation functions.

Neural network: general notation. Let us consider a fully connected feed-forward neural network. Such a network consists of K layers of nodes – an input layer L_1 and an output layer L_K, as well as intermediate layers – hidden layers, between the input and output layers. (In Figure 1 of the main text, we show an example of a neural network with three layers). An lth layer $l \in \{1, 2, ..., K\}$ consists of w_l nodes, with w_l being the layer's width. The nodes of the input layer L_1, denoted $x_i = (x_{i,0}, ..., x_{i,w_1})$, are referred to as *input features* (by convention, $x_{i,0} = 1$). The nodes of an lth hidden layer, labeled $a^{(l)} = \left(a_0^l, ..., a_{w_l}^l\right)$, are called *activation units*, where a_j^l is an activation of a unit j in a layer l.

Omitting an ith observation subscript, let us denote by $z^{(l)}$ an input of a layer $l > 1$. It is a linear combination of the activation units of the previous layer , i.e., $z^{(l)} = \theta^{(l-1)} a^{(l-1)}$, where $\theta^{(l)} \in R^{w_{l+1}} \times R^{w_l + 1}$. Note that $z^{(1)} = x$ and $z^{(L)} = \theta^{(L-1)} a^{(L-1)}$. A non-activated input $z^{(l)}$ is transformed into an activated one by applying an activation function $\tau_l \in \mathcal{T}$, such that

$$a^{(l)} = \tau_l \left(\theta^{(l)} z^{(l)}\right),$$

where τ_l is applied element-wise, i.e., in such a way that the dimension of its argument is the same as the dimension of its output. Distinct layers may use different activation functions (i.e., they can perform different kinds of transformations of their inputs). The predicted output is the result of a hypothesis function $\varphi(x; \theta) = \tau_L \left(z^{(L)}\right) = \tau_L \left(\theta^{(L-1)} a^{(L-1)}\right)$.

[1] In applications, it is convenient that $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

Denoting by $\mathcal{T}$ the set of all activation functions, we define a feedforward neural network of depth L by its topology $((\tau_1, w_1), \ldots, (\tau_L, w_L)) \in (\mathcal{T} \times \mathbb{R}^\star)^L$.

The parameters vector θ is obtained by minimizing $\Xi(\theta) = \sum_{i=1}^{w_L} (y_i - \varphi(x_0, ..., x_{w_1}; \theta))^2$, i.e., mean squared error between the data – the given output features – denoted by $y = (y_0, ..., y_{w_L})$ and the predicted output nodes given by $\varphi(x; \theta)$.

Backpropagation algorithm. Given the assumption on $\Xi(\theta)$, we are to compute $\frac{\partial}{\partial \theta_{ij}^{(l)}} \Xi(\theta)$. For this purpose, one can use a backpropagation algorithm. Given the data (called a training set in machine learning) $\{(x_t, y_t)\}_{t=1}^{n'}$, we perform the following steps for a neural network with L layers:

- Initially, set an error $\triangle_{ij}^{(l)}$ to 0 for all (l, i, j). This variable will be used for accumulating the gradient.
- For each observation t, set activated output of the first layer to x, i.e., $a^{(1)} = x_t$. Apply forward activation to compute the activated output $a^{(l)}$ for $l = 2, 3, ..., L$, as $a^{(l)} = \tau_l \left(\theta^{(l)} z^{(l)}\right)$ where the non-activated output is $z^{(l)} = \theta^{(l-1)} a^{(l-1)}$. This computation is performed for the assumed values of the parameters vector θ.
- For each $\{y_t\}_{t=1}^{n'}$,
  - compute the error value for the last layer as the difference between the actual result in the last layer and the true output,
  $$\delta^{(L)} = a^{(L)} - y_t;$$
  - Compute the error of cost for $a^{(l)}$ in all the layers before the last one, i.e., $\delta^{(L-1)}, ..., \delta^{(2)}$ as follows:
  $$\delta^{(l)} = \theta^{(l)\prime} \delta^{(l+1)} .\cdot \tau' \left(z^{(l)}\right),$$
  where $.\cdot$ denotes the element-wise multiplication;
  - update the new error matrix as
  $$\triangle^{(l)} = \triangle^{(l)} + \delta^{(l+1)} a^{(l)\prime}.$$
- The unknown gradient is given by the partial derivative of the lth layer for the parameter $\theta_{i,j}^{(l)}$,
$$\frac{\partial}{\partial \theta_{i,j}^{(l)}} \Xi = \frac{1}{n'} \triangle_{ij}^{(l)}.$$

Neural networks as universal approximators. Multilayer feedforward networks with hidden layers provide a universal approximation framework.

(*Hornik's et al. (1989) universal approximation theorem).* Whenever the activation function is bounded and non-constant, a standard multilayer feedforward network can approximate any (Borel) measurable function arbitrarily well, given that sufficiently many hidden units are considered.[2] Given that any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable, the above theorem establishes that any continuous function on a compact set can be approximated by a neural network. In addition, a neural network can approximate arbitrarily well any function that maps from any finite dimensional discrete space to another.

Leshno et al. (1993) generalize Hornik's et al. (1989) theorem to provide necessary and sufficient conditions for universal approximation. In particular, they show that a standard multilayer feedforward network can approximate any continuous function arbitrarily well if and only if the activation function is nonpolynomial (including rectified linear units). They emphasize the role of the threshold values without which the result does not hold. A threshold is an element that should be added to an activation function if it is not dense in $C(\mathbb{R}^n)$. For example, $\tau(x) = \sin(x)$ cannot be used to approximate $\cos(x)$ in $[-1, 1]$ because $\sin(w \cdot x)$, $w \in \mathbb{R}$ is not dense in $C([-1, 1])$. This can be changed by adding a threshold $\frac{\pi}{2}$, i.e., $\sin\left(x + \frac{\pi}{2}\right) = \cos(x)$. Activation functions need not be smooth or continuous – nonpolynomiality is the only restriction on he activation functions required. Leshno's et al. (1993) results just specify that a sufficiently

[2] Hornik et al. (1990) also demonstrate that the derivatives of a function can be approximated to any desired degree of accuracy by the derivatives of the feedforward network.

wide network could represent any function without addressing the questions of the network's depth or efficiency. In other words, the theory tells us that a large neural network will be able to represent any function. However, there is no guarantee that the training algorithm will be actually able to learn that function.

In particular, a failure can occur due to the following two reasons. First, the optimization algorithm we use to train the neural network may fail to find parameters values corresponding to the desired function. Second, the training algorithm may overfit which leads to finding a wrong function. A no-free-lunch result applies therefore: there exists no universally superior machine learning algorithm. Barron (1993) shows that although a single-layer network is sufficient to approximate a broad class of functions, an exponential number of hidden units may be required in the worse-case scenario.

C. Training methods in machine-learning literature

In this section, we discuss some training methods used in machine-learning literature. Let us remind that in data science, data are normally fixed and batches are pseudo-random (they are obtained by bootstrapping of the given data). In contrast, in economic dynamics, data are truly random.

Stochastic and batch gradient descent methods. Recall that the basic gradient descent method is given by $\theta_{k+1} \leftarrow \theta_k - \lambda_k \nabla \Xi(\theta)$. Computing the gradient of the expectation function $\nabla_\theta \Xi^n(\theta)$ can be expensive when the data size n is large. A stochastic batch gradient descent (BGD) method is a popular approach to reduce the cost of computing $\nabla_\theta \Xi^n(\theta)$. It constructs the gradient on a small random subset of data with $n' \ll n$ data points,

$$\theta_{k+1} \leftarrow \theta_k - \lambda_k \nabla_\theta \Xi(\theta_k), \qquad \text{with} \qquad \nabla_\theta \Xi^{n'}(\theta_k) \approx \nabla_\theta \left[\frac{1}{n'} \sum_{i=1}^{n'} \xi\left(\omega_i; \theta_k\right) \right], \tag{4}$$

where a subset $(\omega_1, ..., \omega_{n'})$ is called *batch*, ∇_θ is a gradient operator, θ_k and λ_k are a parameter vector and a parameter step on iteration k, respectively.

Two limiting cases of the BGD method are $n' = n$ and $n' = 1$. The former corresponds to the conventional gradient descent (GD) method in which all data points are used for constructing the gradient. The latter corresponds to a stochastic gradient (SGD) method which approximates the expectation function with the value of such a function in one randomly chosen data point ω_k, i.e., $\nabla_\theta E_\omega \left[\xi\left(\omega; \theta_k\right)\right] \approx \nabla_\theta \xi\left(\omega_k; \theta_k\right)$.

Under SGD and BGD methods, $\{\theta_k\}$ is not a uniquely determined sequence but a stochastic process that depends on the realized sequence $\{\omega_k\}$. While such approximation can be very imprecise on each given step, SGD and BGD are unbiased and the cumulative average converges to the true gradient $\frac{1}{K} \sum_{k=1}^{K} \nabla_\theta \xi\left(\omega_k; \theta\right) \to \nabla_\theta \Xi(\theta)$ over K updates, provided that the coefficients are stabilized, $\theta_k \approx \theta$. In other words, each direction should not be necessarily descent, however, if it is descent in expectation, we can find a minimum of $\Xi(\theta)$ over a large number of iterations.

SGD and BGD methods have different trade-offs in terms of per iteration costs and expected per-iteration improvement. Because of the sum structure in the coefficients updating, a full BGD algorithm greatly benefits from parallelization. However, the SGD algorithm uses information about the gradient more efficiently than BGD. To understand, create a new sample by copying the original sample multiple times. By construction, the optimum of the larger set coincides with that of the original smaller set. The full BGD algorithm that uses the larger set will be more expensive than its version that uses a smaller set. The SGD performs the same computation and has the same costs in both scenarios. Although in practice samples are not obtained by creating multiple copies of the original sample, there is plenty of redundancy in the data. This observation suggests that it is inefficient to use the whole sample on every iteration, as is done under the full batch approach, and that working with small samples, even one observation, as in the case of the SGD method, might be more beneficial. In practice, however, the SGD is characterized by fast improvements on initial iterations but dramatic posterior slowdowns (see Bertsekas, 2015) for an intuitive explanation of such behavior. This issue is addressed by a steady reduction in the stepsize as iterations progress. Although in theory, the basic SGD has a slower rate of convergence than the full BGD does, SGD's per iteration costs is independent of the sample size n. A mini BGD method still has gains from parallelization and is also more efficient than a full-batch version.

There are other more sophisticated versions of the SGD method that ensure faster convergence and have stronger convergence properties such as Nesterov and ADAM methods; see Goodfellow et al. (2016). There is a trade-off between using lots of parallel simulations with a large N so that the last term is a close approximation of the expected gradient and going for faster updates. These training algorithms feature time-varying learning rates and/or parameter specific

updates rules (so that higher variance parameters are updated slower). In the paper, we use some of these training methods, in particular, ADAM. Below we provide some details on this training method.

ADAM was proposed by Kingma and Ba (2014); its name is an abbreviation from "adaptive moments". It is a combination of two other extensions of stochastic gradient descent, namely, Root Mean Square Propagation, RMSProp, and momentum. RMSProp includes

$$
\begin{aligned}
g_k &\leftarrow \frac{1}{n'}\sum_{i=1}^{n'} \nabla \ell\left(\varphi\left(\omega_i, \theta_k\right)\right), \\
r_{k+1} &\leftarrow \rho r_k + (1-\rho)\, g_k \odot g_k, \\
\triangle\theta_{k+1} &= -\frac{\lambda}{\sqrt{\delta + r_{k+1}}} \odot g_k, \\
\theta_{k+1} &\leftarrow \theta_k + \triangle\theta_{k+1}.
\end{aligned}
$$

where δ is a small constant; $\rho \in (0,1)$. Here, the learning rates of all parameters are adjusted individually by making a step that is inversely proportional to the square root of the exponentially moving average of the previous squared values $g_k \odot g_k$ of the gradient g_k. RMSProp also includes an estimate of the second-order moment $g_k \odot g_k$. In a momentum algorithm, there are two additional parameters, a velocity vector v and a hyperparameter $\alpha \in [0,1)$; the latter determines how quickly the effect of the previous gradients g_k decreases,

$$
\begin{aligned}
g_k &\leftarrow \frac{1}{n'}\sum_{i=1}^{n'} \nabla \ell\left(\varphi\left(\omega_i, \theta_k\right)\right), \\
v_{k+1} &\leftarrow \alpha v_k - \lambda g_k, \\
\theta_{k+1} &\leftarrow \theta_k + v_{k+1}.
\end{aligned}
$$

ADAM applies momentum to the rescaled gradients. The algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters ρ_1 and ρ_2 control the decay rates of these moving averages,

$$
\begin{aligned}
g_k &\leftarrow \frac{1}{n'}\sum_{i=1}^{n'} \nabla \ell\left(\varphi\left(\omega_i, \theta_k\right)\right), \\
s_{k+1} &\leftarrow \rho_1 s_k + (1-\rho_1)\, g_k, \\
r_{k+1} &\leftarrow \rho_2 r_k + (1-\rho_2)\, g_k \odot g_k, \\
\widehat{s}_{k+1} &\leftarrow \frac{s_{k+1}}{1-\rho_1^t}, \\
\widehat{r}_{k+1} &\leftarrow \frac{r_{k+1}}{1-\rho_2^t}, \\
\triangle\theta_{k+1} &= -\frac{\widehat{s}_{k+1}}{\delta + \sqrt{r_{k+1}}}, \\
\theta_{k+1} &\leftarrow \theta_k + \triangle\theta_{k+1}.
\end{aligned}
$$

Thus, ADAM incorporates bias corrections for the first–order-moment estimate g_k and for the second–order-moment estimate $g_k \odot g_k$. In contrast, RMSProp only includes a correction for the second–order-moment estimate (and not the first-order moment estimate) and does not have a correction factor $1-\rho_2^t$.

Convergence of the SGD method. In our minimization (maximization) problems, objective functions should not be necessarily convex (concave) in parameters θ. Such functions may have multiple local minima (maxima) and other stationary points. It turns out that one can still provide some guarantees that the basic SGD method converges in nonconvex (nonconcave) settings.

Below, we provide the proof of this result for the case of a constant learning rate. In this proof, we follow Bottou, Curtis and Nocedal (2018). The proof works interchangeably for both the expected risk $\Xi(\theta)$ and empirical risk $\Xi^n(\theta)$.

Therefore, to represent both of them, we denote the objective by $F:\mathbb{R}^d\to\mathbb{R}$,

$$F(\theta)=\begin{cases}\Xi(\theta)=E\left[\ell\left(\varphi\left(\omega;\theta\right)\right)\right],\\ \text{or}\\ \Xi^n(\theta)=\frac{1}{n}\sum_{i=1}^{n}\ell\left(\varphi\left(\omega_i;\theta\right)\right),\end{cases}\tag{5}$$

where ℓ denotes a loss function equal to the difference between a true output y and predicted output $\varphi(\omega;\theta)$ when minimizing either expected or empirical risks. The difference between these two cases depends on how the SGD chooses the samples in each iteration. If $F(\theta)=\Xi(\theta)$, then it is done uniformly from a finite set of observations. If $F(\theta)=\Xi^n(\theta)$, it is done using the probability distribution $P:\mathbb{R}^d\longrightarrow[0,1]$.

Denote by $g(\omega_k;\theta_k)$ a stochastic vector of gradients that covers, respectively, both the basic SGD and a mini BGD methods,

$$g(\omega_k;\theta_k)=\begin{cases}\nabla\ell\left(\varphi\left(\omega_k;\theta_k\right)\right)\\ \text{or}\\ \frac{1}{n'}\sum_{k=1}^{n'}\nabla\ell\left(\varphi\left(\omega_k;\theta_k\right)\right).\end{cases}$$

Below, we provide a general discussion of the steps for the gradient descent (GD) algorithms studied.

Algorithm 1. (a GD algorithm):

- *Make an initial guess on the parameters vector, i.e.,* θ_1.
- *For* $k=1,2,...,$ *do the following:*
  - *draw a random realization for* ω_k.
  - *compute a stochastic vector of gradients* $g(\omega_k;\theta_k)$.
  - *choose a learning rate* $\lambda_k>0$.
  - *compute the new parameters vector as* $\theta_{k+1}\leftarrow\theta_k-\lambda_k g(\omega_k;\theta_k)$.
- *End iterations when convergence is achieved.*

The proof of convergence relies on two assumptions, one is about a Lipschitz-continuous objective gradient, and the other is about the first and second moments of the gradients.

Assumption 1. (Lipschitz-continuous objective gradients). *(1) The objective function* $F:\mathbb{R}^d\to\mathbb{R}$ *is continuously differentiable.*
(2) The gradient of F *denoted* $\nabla F:\mathbb{R}^d\to\mathbb{R}$ *is Lipschitz continuous with a constant* $L>0$ *:*

$$\left\|\nabla F(\theta)-\nabla F\left(\overline{\theta}\right)\right\|_2\leq L\left\|\theta-\overline{\theta}\right\|_2\ \text{for all}\ \left\{\theta,\overline{\theta}\right\}\subset\mathbb{R}^d.\tag{6}$$

Note that Assumption 1 is equivalent to the following property:

$$F(\theta)\leq F\left(\overline{\theta}\right)+\nabla F\left(\overline{\theta}\right)'\left(\theta-\overline{\theta}\right)+\frac{1}{2}L\left\|\theta-\overline{\theta}\right\|_2^2\ \text{for all}\ \left\{\theta,\overline{\theta}\right\}\subset\mathbb{R}^d.\tag{7}$$

The equivalence of (6) and (7) is verified in Bottou et al. (2018). Moreover, this assumption leads to the following useful result for all iterations $k\in\{1,...,K\}$:

$$E_{\omega_k}\left[F(\theta_{k+1})-F(\theta_k)\right]\leq-\lambda_k\nabla F(\theta_k)'E_{\omega_k}\left[g(\omega_k;\theta_k)\right]+\frac{1}{2}\lambda_k^2LE_{\omega_k}\left[\left\|g(\omega_k;\theta_k)\right\|_2^2\right],\tag{8}$$

which follows directly by Assumption 1.

One can achieve convergence of the studied GD methods by including additional requirements on the first and second moments of $g(\omega_k;\theta_k)$. These requirements help us bound the right-hand side of the inequality (8). We define the variance of $g(\omega_k;\theta_k)$ as follows:

$$V_{\omega_k}\left[g(\omega_k;\theta_k)\right]\equiv E_{\omega_k}\left[\left\|g(\omega_k;\theta_k)\right\|_2^2\right]-\left\|E_{\omega_k}g(\omega_k;\theta_k)\right\|_2^2.\tag{9}$$

To be specific, assume that our optimization problem is a minimization problem.

Assumption 2. (Limits on the first and second moments). *The objective function F in (5) and Algorithm 1 satisfy the following three properties:*
(a) (Objective function). The sequence $\{\theta_k\}$ is in an open set where the objective function F is bounded from below by a scalar $F_{\inf}$, i.e., $F(\theta_k) \geq F_{\inf}$ for all $k \in \{1, ..., K\}$.
(b) (First moment). There exist scalars μ_G and μ such that $\mu_G \geq \mu > 0$ and for all k, we have

$$\nabla F(\theta_k)' E_{\omega_k}[g(\omega_k; \theta_k)] \geq \mu \|\nabla F(\theta_k)\|_2^2, \tag{10}$$

$$\|E_{\omega_k} g(\omega_k; \theta_k)\|_2 \leq \mu_G \|\nabla F(\theta_k)\|_2. \tag{11}$$

(c) (Second moment). There exist scalars $H \geq 0$ and $H_V \geq 0$ such that for all k, we have

$$V_{\omega_k}[g(\omega_k; \theta_k)] \leq H + H_V \|\nabla F(\theta_k)\|_2^2. \tag{12}$$

Given Assumptions 1 and 2, we prove the following convergence result for Algorithm 1.

Theorem (Nonconvex objective and a fixed learning rate). Suppose Assumptions 1 and 2 hold. Assume $\lambda_k = \lambda > 0$ (constant) for all k and it satisfies

$$\lambda \leq \frac{\mu}{LH_G}. \tag{13}$$

Then, the expected sum of squares of the gradients of $F(\theta_k)$ satisfies

$$E\left[\sum_{k=1}^{K} \|\nabla F(\theta_k)\|_2^2\right] \leq \frac{K\lambda LH}{\mu} + \frac{2F(\theta_1) - F_{\inf}}{\mu\lambda}, \tag{14}$$

and the expected average squared gradients of $F(\theta_k)$ satisfies

$$E\left[\frac{1}{K}\sum_{k=1}^{K} \|\nabla F(\theta_k)\|_2^2\right] \leq \frac{\lambda LH}{\mu} + \frac{2[F(\theta_1) - F_{\inf}]}{K\mu\lambda} \tag{15}$$

$$\underset{K \longrightarrow \infty}{\longrightarrow} \frac{\lambda LH}{\mu}. \tag{16}$$

Proof.

- From (8) and (10), we have

$$\begin{aligned} &E_{\omega_k}[F(\theta_{k+1}) - F(\theta_k)] \\ &\quad \leq -\lambda_k \nabla F(\theta_k)' E_{\omega_k}[g(\omega_k; \theta_k)] + \frac{1}{2}\lambda_k^2 L E_{\omega_k}\left[\|g(\omega_k; \theta_k)\|_2^2\right] \\ &\quad \leq -\mu\lambda_k \|\nabla F(\theta_k)\|_2^2 + \frac{1}{2}\lambda_k^2 L E_{\omega_k}\left[\|g(\omega_k; \theta_k)\|_2^2\right]. \end{aligned} \tag{17}$$

- The definition of the second moment in (9), together with Assumption 2 in (12), yields

$$E_{\omega_k}\left[\|g(\omega_k; \theta_k)\|_2^2\right] \leq H + \underbrace{[H_V + \mu_G^2]}_{\equiv H_G \geq \mu^2} \cdot \|\nabla F(\theta_k)\|_2^2. \tag{18}$$

- Combining the last two equations (17) and (18) yields

$$E_{\omega_k}[F(\theta_{k+1}) - F(\theta_k)] \leq -\left(\mu - \frac{1}{2}\lambda_k LH_G\right)\lambda_k \|\nabla F(\theta_k)\|_2^2 + \frac{1}{2}\lambda_k^2 LH. \tag{19}$$

- Taking the total expectation of (19), we obtain

$$E\left[F\left(\theta_{k+1}\right)\right]-E\left[F\left(\theta_{k}\right)\right] \leq-\left(\mu-\frac{1}{2} \lambda_{k} L H_{G}\right) \lambda_{k} E\left\|\nabla F\left(\theta_{k}\right)\right\|_{2}^{2}+\frac{1}{2} \lambda_{k}^{2} L H.$$

- Imposing now that $\lambda_k = \lambda$ and that $\lambda \leq \frac{\mu}{L H_G}$, we get

$$E\left[F\left(\theta_{k+1}\right)\right]-E\left[F\left(\theta_{k}\right)\right] \leq-\frac{1}{2} \mu \lambda E\left\|\nabla F\left(\theta_{k}\right)\right\|_{2}^{2}+\frac{1}{2} \lambda^{2} L H.$$

- Summing up the latter expression over all iterations $k \in \{1, ..., K\}$ and recalling that the sequence $\{\theta_k\}$ is such that $F(\theta_k) \geq F_{\text{inf}}$ for all k, we obtain

$$F_{\inf }-F\left(\theta_{1}\right) \leq E\left[F\left(\theta_{K+1}\right)\right]-F\left(\theta_{1}\right) \leq-\frac{1}{2} \mu \lambda \sum_{k=1}^{K} E\left\|\nabla F\left(\theta_{k}\right)\right\|_{2}^{2}+\frac{1}{2} K \lambda^{2} L H.$$

- Re-grouping the terms in the last expression leads to (14). The division of the resulting equation by K implies (15). □

According to Assumption 2 about the second moment in (12), when $H = 0$, there is no noise or the noise goes down proportionally to $\|\nabla F(\theta_k)\|_2^2$, so that equation (16) implies that the sum of squared gradients is finite and that the sequence $\{\|\nabla F(\theta_k)\|_2\} \longrightarrow 0$ as $K \longrightarrow \infty$. When $H > 0$, there is an interaction between the learning rate λ and the variance of the stochastic directions. Result (15) provides a bound on the average squared gradient of the objective function observed over K iterations. As K increases, this average squared gradient becomes smaller, implying that a GD method spends increasingly more time in regions where the objective function has a (relatively) small gradient. According to (16), when $H \neq 0$, noise in the gradients prevents further progress in convergence (as the presence of nonzero term $\frac{\lambda L H}{\mu}$ indicates). However, the average squared gradient can be reduced by choosing a small learning rate; the drawback of a smaller λ would be a lower speed of convergence.

One can also prove the convergence result for the case of non-constant learning rate λ_k. We state this result without proving it in the theorem below.

Theorem (Nonconvex objective and a diminishing learning rate). Suppose Assumptions 1 and 2 hold. Assume a sequence $\{\lambda_k\}$ satisfies

$$\Delta_{k} \equiv \sum_{k=1}^{K} \lambda_{k}=\infty \text{ and } \sum_{k=1}^{K} \lambda_{k}^{2}<\infty. \tag{20}$$

Then, the expected sum of squares of the gradients of $F(\theta_k)$, weighted by λ_k, satisfies

$$E\left[\sum_{k=1}^{K} \lambda_{k}\left\|\nabla F\left(\theta_{k}\right)\right\|_{2}^{2}\right]<\infty, \tag{21}$$

and the expected average squared gradients of $F(\theta_k)$, weighted by λ_k, satisfies

$$E\left[\frac{1}{\Delta_{k}} \sum_{k=1}^{K} \lambda_{k}\left\|\nabla F\left(\theta_{k}\right)\right\|_{2}^{2}\right] \underset{K \longrightarrow \infty}{\longrightarrow} 0. \tag{22}$$

Proof. See Bottou et al (2018).

D. Connection to supervised, unsupervised and reinforcement learning literature

Our solution method is related to the fields of supervised, unsupervised and reinforcement learning. First, nonlinear regression equations, which we estimate using artificial data, can be viewed as a generalization of canonical supervised learning; see Supplement A for the formal results. Second, since the decision and value functions are not known in economic models, we can also interpret our solution method as a version of unsupervised learning. Finally, given that

our method approximates not only the decision and value functions but also the ergodic set, it has a connection to reinforcement learning. Interestingly, there are numerous AI-like approaches in computational economics that were discovered independently or even preceded similar AI approaches in the data science. We discuss some of related approaches in our literature review.[3]

Supervised learning

Essentially, all solution methods in economics use regression or interpolation techniques for approximating policy and value functions off the grid – such techniques can be classified as supervised learning. The typical approximation family is polynomials (ordinary, Chebyshev, Hermite, etc.) but other families were also considered, including neural networks. The first application of neural networks to economic dynamics dates back to Duffy and McNelis (2001) who use neural networks for parametrizing decision functions in a growth model. Recently, multilayer neural networks are used by Duarte (2018) for approximating value functions which solve Hamilton-Jacobi-Bellman equations; by Fernández-Villaverde et al. (2019) for approximating an aggregate law of motion in a continuous-time version of the Krusell and Smith (1998) model; by Villa and Valaitis (2019) for dealing with ill-conditioning in a parameterized expectations algorithm (PEA) of Den Haan and Marcet (1990); by Lepetyuk et al. (2020) for solving a large-scale central banking model.

However, interpolation does not fully utilize the capacities of the existing AI technology. In that case, AI technology does not solve the entire economic model but serves as one of the ingredients of the conventional solution method. We differ from that literature in that we generalize supervised learning to cast the entire economic model into a single objective function, so that AI produces the entire solution – we do so for three key objects of economic dynamics: lifetime reward, Bellman equation and Euler equation.

There are papers on computational economics that propose numerical approaches related to ours. The lifetime-reward maximization method is related to the indirect inference procedure of Smith (1987). The Euler-equation method is related to seminal contributions to numerical solution methods in economics, namely, a projection method of Judd (1992) and PEA of Den Haan and Marcet (1990). In particular, PEA also approximates expectations from simulated data but it does so via a different mechanism, namely, when iterating on the current decision functions, PEA takes the expectation functions from the previous iteration as given. A shortcoming of PEA is that it requires long time-series simulation for accurate solutions; by contrast, we need only few random grid points. A recent paper by Azinovic et al. (2020) uses a solution method, similar to our Euler-equation method, to solve a life-cycle model. Maliar and Maliar (2020) introduce a deep learning classification method for modeling non-convex choices (e.g, indivisible labor choice, rent versus buy), and Gorodnichenko et al. (2020) use deep learning to solve a heterogeneous-agent new Keynesian model.

Finally, our Bellman-equation method is related to conventional value and policy function iteration (see e.g., Rust, 1996, Judd, 1998, Santos, 1999, Aruoba et al., 2006, Stachurski, 2009) but we differ in two respects: first, we combine the minimization of Bellman equation residuals with maximization of the right side of the Bellman equation into a single optimization step, and second, we implement integration with two uncorrelated shocks in a way that facilitates a construction of expectation functions. However, our most important contribution consists in showing that large-scale dynamic economic models can be treated by using the same model-free DL technologies that the scientific community uses in other fields, leading to truly break-ground applications.

Unsupervised learning

Unsupervised learning literature focuses on how to effectively represent information that is available in a given set of features. For example, it clusters closely situated data, it reduces dimensionality of collinear features by principal component analysis, etc. There are numerous applications of unsupervised learning in the economic literature. In particular, Judd et al. (2011) analyze a variety of model reduction techniques for dealing with ill conditioning, including a principle component regression, a truncated SVD method, Tykhonov regularization and regularized least absolute deviation methods. Furthermore, several papers use model reduction for simplifying the analysis of equilibrium in heterogeneous-agent models, including Reiter (2010), Ahn et al (2018), Winberry (2018) and Bayer and Luetticke (2020).

There is also a literature that uses unsupervised learning to determine ergodic sets of economic models, in particular, Judd et al. (2011) and Maliar and Maliar (2015) use clusters techniques; Judd et al. (2017) and Coleman et al. (2018)

[3] See Goodfellow et al. (2016) for a review of supervised and unsupervised learning in the computer science literature, in particular, deep learning; see Sutton and Barto (2018) for a review of reinforcement learning literature, and see Powell (2008) for a review of the related field of approximate dynamic programming.

use low-discrepancy (quasi-Monte Carlo) sequences, and Renner and Scheidegger (2018) and Scheidegger and Bilionis (2019) use Gaussian-process machine learning.

Finally, there are papers that combine supervised learning with unsupervised learning. In particular, Lepetyuk et al. (2020) use clusters to refine the solution domain and use neural network for parameterizing decision functions. The neural network itself can actually deal with ill conditioning and reduce dimensionality instead of unsupervised learning methods; see Villa and Valaitis (2019). However, again the existing literature uses unsupervised learning just as an ingredient of the conventional solution methods, whereas we use DL methods to produce the entire solution to the model. In fact, Azinovic et al. (2020) interpret a related Euler-equation method as a version of unsupervised learning which is another possible interpretation given a tight connection between supervised and unsupervised learning discussed in Supplement A.

Reinforcement learning

Reinforcement learning (RL) is a field that focuses on solving dynamic problems with a delayed, often discounted, reward. For instance, the game-playing engine Alphazero, gets a positive reward when a game is won, and zero otherwise; see Jirniy and Lepetyuk (2012) for early remarkable application of RL for solving Krusell and Smith (1998) model.

A characteristic feature of RL methods is that they explore the state space. In that respect, our simulation-based method is similar to RL. However, much of RL research focuses on aspects that are absent in our analysis. For instance, an interesting element of RL is the ability to learn online and the trade-off during the learning, exploration or exploitation phases. In contrast, in our case, learning is fully offline in the sense that there is no economic cost associated with suboptimal decisions taken during the training phase. For the same reason there is no tradeoff between exploitation and exploration, a usual feature of RL approaches. Furthermore, RL approaches allow for model-free learning. In contrast, we assume full knowledge of the model and the ability to simulate trajectories. The Euler-equation method sets us further apart from the RL agenda, limited to the optimal control problems. But our Euler-equation method can be adapted to online optimization learning so the frontiers are not fully watertight.

References

[1] Ahn, S., Kaplan, G., Moll, B., Winberry, T., Wolf, C., 2018. When inequality matters for macro and macro matters for inequality. NBER Macroeconomics Annual, University of Chicago Press, vol. 32(1), 1-75.

[2] Aruoba, S. B., Fernández-Villaverde, J., Rubio-Ramírez, J., 2006. Comparing solution methods for dynamic equilibrium economies. Journal of Economic Dynamics and Control 30, 2477–2508.

[3] Azinovic, M., Luca, J., Scheidegger, S., 2020. Deep equilibrium nets. SSRN: https://ssrn.com/abstract=3393482

[4] Barron, A., 1993. Universal approximation bounds for superpositions of a sigmoid function. IEEE Transactions on Information Theory 39, 930-945.

[5] Bayer, C., Luetticke, R., 2020. Solving discrete time heterogeneous agent models with aggregate risk and many idiosyncratic states by perturbation. Quantitative Economics 11, 1253-1288.

[6] Bertsekas, D., 2015. Convex optimization algorithms. Athena Scientific, Nashua, NH, USA.

[7] Bottou, L., Curtis, F., Nocedal, J., 2018. Optimization methods for large-scale machine learning. Manuscript.

[8] Coleman, C., Lyon, S., Maliar, L., Maliar, S., 2018. Matlab, python, julia: what to choose in economics? CEPR working paper DP 13210. Computational Economics, forthcoming.

[9] Den Haan, W., Marcet, A., 1990. Solving the stochastic growth model by parameterized expectations. Journal of Business and Economic Statistics 8, 31–34.

[10] Duarte, V., 2018. Machine learning for continuous-time economics. SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3012602

[11] Duffy, J., McNelis, P., 2001. Approximating and simulating the real business cycle model: parameterized expectations, neural networks, and the genetic algorithm, Journal of Economic Dynamics and Control 25(9), 1273-1303.

[12] Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep learning. Massachusetts Institute Technology Press.

[13] Gorodnichenko, Y., Maliar, L., Maliar, S., Naubert, C., 2020. Household savings and monetary policy under individual and aggregate stochastic volatility. CEPR working paper 15614.

[14] Fernández-Villaverde, J., Hurtado, S., Nuño, G., 2019. Financial frictions and the wealth distribution. NBER Working paper 26302.

[15] Hornik, K. (1989). Multilayer feedforward networks are universal approximators. Neural networks 2, 359-366.

[16] Kingma, D. and J. Ba, (2014). Adam: a method for stochastic optimization. https://arxiv.org/pdf/1412.6980.pdf

[17] Krusell, P., Smith, A., 1998. Income and wealth heterogeneity in the macroeconomy. Journal of Political Economy 106, 868–896.

[18] Jirniy, A., Lepetyuk, V., 2011. A reinforcement learning approach to solving incomplete market models with aggregate uncertainty. SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1832745

[19] Judd, K. L., 1992. Projection methods for solving aggregate growth models. Journal of Economic Theory 58, 410–452.

[20] Judd, K. L., Maliar, L., Maliar, S., 2011. Numerically stable and accurate stochastic simulation approaches for solving dynamic models. Quantitative Economics 2, 173–210.

[21] Lepetyuk, V., Maliar, L., Maliar, S., 2020. When the U.S. catches a cold, Canada sneezes: a lower-bound tale told by deep learning. Journal of Economic Dynamics and Control 117, 103926.

[22] Leshno, M., V. Lin, A. Pinkus, S. Schocken, (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Networks 6, 861-867.

[23] Maliar L., Maliar, S., 2015. Merging simulation and projection approaches to solve high-dimensional problems with an application to a new Keynesian model. Quantitative Economics 6, 1-47.

[24] Maliar, L., Maliar,S., 2020. Deep learning classification: Modeling discrete labor choice. CEPR working paper DP 15346.

[25] Powell, W., 2010. Approximate dynamic programming. Wiley, A. John Wiley & Sons.

[26] Reiter, M., 2010. Approximate and almost-exact aggregation in dynamic stochastic heterogeneous-agent models. IHS Working Paper 258.

[27] Renner, P., Scheidegger, S., 2018. Machine learning for dynamic incentive problems. SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3282487

[28] Rosenblatt, F., 1958. The perceptron: a probabilistic model for information storage and organization in the brain. Psychological Review 65 (6), 386-408.

[29] Rust, J., 1996. Numerical dynamic programming in economics. In: Amman,H., Kendrick,D., Rust, J. (Eds.), Handbook of Computational Economics. Elsevier Science,Amsterdam, pp. 619–722.

[30] Santos, M., 1999. Numerical solution of dynamic economic models. In: Taylor, J.,Woodford, M. (Eds.), Handbook of Macroeconomics. Elsevier Science, Amsterdam, pp. 312–382.

[31] Scheidegger, S., Bilionis, I., 2019. Machine learning for high-dimensional dynamic stochastic economies. Journal of Computational Science 33, 68–82.

[32] Smith, A., 1993. Estimating nonlinear time-series models using simulated vector autoregressions. Journal of Applied Econometrics 8, S63–S84.

[33] Stachurski, J., 2009. Economic dynamics: theory and computation. MIT Press.

[34] Sutton R., Barto, A., 2018. Reinforcement Learning: An Introduction. The MIT Press, Cambridge, Massachusetts, London, England.

[35] Villa, A., Valaitis, V., 2019. Machine learning projection methods for macro-finance models. SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3209934.

[36] Winberry, T., 2018. A method for solving and estimating heterogeneous agent macro models, Quantitative Economics 9 (3), 1123–1151.