Will Artificial Intelligence Replace Computational Economists Any Time Soon?*

Lilia Maliar† Serguei Maliar‡ Pablo Winant§

October 23, 2019

Abstract

Artificial intelligence (AI) has impressive applications in many fields (speech recognition, computer vision, etc.). This paper demonstrates that AI can be also used to analyze complex and high-dimensional dynamic economic models. We show how to convert three fundamental objects of economic dynamics – lifetime reward, Bellman equation and Euler equation – into objective functions suitable for deep learning (DL). We introduce all-in-one integration technique that makes the stochastic gradient unbiased for the constructed objective functions. We show how to use neural networks to deal with multicollinearity and perform model reduction in Krusell and Smith's (1998) model in which decision functions depend on thousands of state variables (we literally feed distributions into neural networks!) In our examples, the DL method is reliable, accurate and linearly scalable. Our ubiquitous Python code, built with Dolo and Google TensorFlow platforms, is designed to accommodate a variety of models and applications.

Key Words : artificial intelligence, machine learning, deep learning, neural network, stochastic gradient, dynamic models, dynamic programming, Bellman equation, Euler equation, value function

*The earlier version of the paper was titled "Deep Learning for Solving Dynamic Economic Models". The authors are grateful to Marc Maliar for his help with writing the code for Krusell and Smith's (1998) model. We thank the participants of multiple conferences and seminars in which that paper was presented, including 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; Complutense University of Madrid, Stanford University, Deutch Bundesbank.

†The Graduate Center, City University of New York, USA

‡Santa Clara University and Columbia University, USA

§ESCP Europe Business School and CREST, France

1 Introduction

Artificial intelligence (AI) is broadly defined as a device that perceives its environment and takes actions that maximize its chance of successful achieving its goals. The AI definition closely resembles the notion of "agent" in dynamic economic models under the following interpretation: i) the environment is the economy's state; ii) the actions are the decision and value functions of the agent; iii) the goal is an optimum of the objective function. Therefore, economists, like researchers from other fields, can hope to apply general-purpose AI technology for solving their models.

AI has many impressive applications, such as the recognition of handwritten numbers/speech, reconstruction of images, production of non-distinguishable Chopin music, playing Go, facilitation of computer vision, operation of self-driving cars, among others; see Goodfellow et al. (2016) for a review. At the same time, there are many interesting problems that computational economists cannot satisfactorily solve yet, including heterogeneous-agent models, large-scale central banking models, life-cycle models, expensive nonlinear estimation procedures, etc. In the present paper, we show that it is possible to solve challenging dynamic economic models by using the same AI technology and the same combination of software and hardware that led to break-ground applications in data science. Our analysis builds on deep learning (DL) framework and has the following four novel results:

First, we offer a unified approach that makes it possible to cast three fundamental objects of economic dynamics – lifetime reward, Bellman equation and Euler equation – into objective functions of the state-of-the-art deep learning framework. In effect, our approach reduces dynamic economic models to regression equations. Although such regression equations are nonlinear, complex and have many variables, the modern DL technologies can readily handle them. Once the regression coefficients are constructed, we use them to 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 the three constructed objective functions. In each iteration, we train a neural network on just one or few (batch) grid points which are randomly drawn from the state space, instead of using a fixed grid with a large number of grid points (as the conventional projection and value iterative methods do in computational economics). Our grid points are truly random, unlike bootstrap draws from the data sample in the DL literature. We also show that not only neural networks but also other functional approximations such as polynomials and piecewise linear functions can be effectively trained using the stochastic gradient method.

Third, we introduce integration methods that are suitable for the constructed objective functions in the context of DL-based simulation. A distinctive feature of the objective functions, derived from economic models, is that they have two types of expectation operators, one is with respect to current state variables (which arise because grid points that are randomly drawn from the state space), and the other is with respect to future state variables (which arise because next-period shocks are randomly drawn from the given distributions). One integration method we propose is a hybrid method that uses DL-style Monte Carlo simulation for producing random grid points for state variables and that uses deterministic methods (such as quadrature, monomials, sparse grids, low-discrepancy sequences) for constructing expectation functions with respect to future shocks (in the Euler and Bellman equations).

However, our truly novel integration method is the so-called *all-in-one expectation method* that merges the two expectation operators into one. The way we construct such an operator differs for the three objective functions considered: For the lifetime reward maximization, we draw randomly the initial values of state variables, in addition to future shocks. For the Euler-equation method, we use two independent random draws for evaluating two terms of a squared residual – this trick eliminates the correlation between the terms and pulls the expectation operator out of the square. Furthermore, we show how to extend the Euler-equation method to dealing with inequality constraints by representing the Kuhn-Tucker conditions with the Fischer-Burmeister or minimum functions; see Jiang (1996) for a discussion of that function. Finally, for the Bellman-equation method, we offer a novel value-iterative scheme that is particularly suitable for the DL framework, specifically, we combine 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.

Our all-in-one integration method possesses an outstanding *distributive property*: a single Monte Carlo draw of the integrand delivers an unbiased estimator of the stochastic gradient with respect to all random variables. This property makes it possible to use DL-style Monte Carlo simulation for evaluating expectation with respect to both state variables and future shocks simultaneously, merging in one step the approximation of decision functions and the integration with respect to future shocks. To increase the efficiency of integration even further, we employ variance reduction techniques such as antithetic variates; see, e.g., Cheng (1982).

Our last important contribution is to automate the DL solution framework in a way that makes it ubiquitous and portable to a variety of economic models and applications. Our automated implementation includes two steps: First, we input the model into a Dolo platform that provides a user-friendly interface for manipulating the model's equations and for combining such equations into a symbolic Python code.[1] Next, we pass the Dolo output file to our model-free TensorFlow code that simulates the model and constructs the solution. Our software will be made available on both personal web pages and open-source software sites such as *QuantEcon.org*.

We first illustrate our DL solution framework in the context of the benchmark consumption savings problem with borrowing constraints, as well as its multi-shock version. Approximation errors for our most accurate methods do not exceed a fraction of 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 – a prominent feature of DL methods based on stochastic gradient. In our experiments, we find the DL solution framework to be reliable, accurate and scalable.

We then solve a challenging Krusell and Smith (1998) model with up to 1,000 heterogeneous agents by using two remarkable properties of neural networks, namely, their capacity to perform a model reduction and their ability to handle multicollinearity. In that case, we train the machine on stochastic simulation instead of exogenously given domain. We offer a remarkably simple algorithm for solving Krusell and Smith's (1998) model–we simulate forward a panel of heterogenous agents by feeding the entire joint distribution of wealth and productivities into their decision functions in each training step. The resulting decision function of each agent depends on thousands of state variables of all other agents, including variables that are perfectly colinear. However, the neural network can learn compact representations of the state space by extracting and condensing the relevant information from high-dimensional distributions into smaller sets of variables in hidden layers. Furthermore, the neural network can learn to ignore the presence of redundant collinear variables. Krusell and Smith (1998) discovered that a single statistic – the mean of the wealth distribution – can effectively characterize the aggregate state of their model but this result does not hold for all heterogenous-agent models. In our algorithm, the neural network will automatically search for all possible statistics that can effectively characterize the state space of the given heterogeneous agent economy – this is what model reduction means in our analysis.

Our solution framework is connected to both data science and computational economics. In the data science, the related approaches are supervised, unsupervised and reinforcement learning. Concerning economics, our lifetime-reward maximization method is related to an indirect inference procedure of Smith (1987); our Euler-residual minimization method is related to the projection method of Judd (1992) and the parameterized expectations algorithm (PEA) of Den Haan and Marcet (1990); and finally, our Bellman-residual method is related to conventional value and policy function iteration; see, e.g., Rust (1996), Santos (1999), Aruoba et al. (2006), Stachurski (2009). There are other papers that used neural networks in the context of solution methods to dynamic economic models. The early applications date back to Duffy and McNelis (2001) and recent applications include Duarte (2018), Fernández-Villaverde et al. (2018), Villa and Valaitis (2019) and Azinovic et al. (2019). To the best of our knowledge, we were the first to show how to cast the entire economic model into an expectation function of DL framework and how to train such a function on random grids by using stochastic gradient descent methods; see *https://lmaliar.ws.gc.cuny.edu* for our 2018 CEF and ESAM presentations. We explain the connection of our analysis to the literature after we present our framework.

[1] Dolo software is developed by Pablo Winant; see https://github.com/econforge/dolo.

The rest of the paper is organized as follows: Section 2 gives a quick overview of the key ingredients of DL (multilayer neural networks, stochastic gradient training method, etc.). Section 3 shows how to cast three main objects of economic dynamics (lifetime reward, Bellman equation and Euler equations) into expectation functions. Section 4 explains the relation of the proposed methods to the literature. Section 5 and 6 analyze the consumption-saving model. Finally, in Section 7, we conclude.

2 There is AI technology out there

In this section, we describe canonical supervised learning framework, and we review numerical techniques available to data scientists. Later in the paper, we show how this framework can be adapted to solving dynamic economic models.

2.1 Canonical supervised learning and its generalization

In canonical supervised learning framework, a machine attempts to learn a function that maps input 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 $\left\{\varphi(\cdot;\theta) : \theta \in \mathbb{R}^{d_\theta}\right\}$, 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 true output y and predicted output $\varphi(x;\theta)$, and we minimize the expected loss, called *expected risk* $\Xi(\theta)$,

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

where $P : \mathbb{R}^{d_x} \times \mathbb{R}^{d_y} \longrightarrow [0,1]$ is a joint probability distribution function, and $E_x[\cdot]$ is an expectation operator. That is, $\Xi(\theta)$ gives expected loss, for a given $\varphi(\cdot;\theta)$ 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 $\{(x_i,y_i)\}_{i=1}^n \subseteq \mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$ 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*, and is typically performed by using a version of gradient descent method.

An example of supervised learning is a familiar linear regression model $y = \theta x$, in which approximation is given by $\varphi(x;\theta) = \theta x$ and the loss function is defined as the minimum least-squares deviation $\ell(\theta x, y) = (y - \theta x)^2$. However, supervised learning accommodates many other interesting data-science applications, in particular, classification problems.[2]

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

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

[2] An example is a problem of handwritten-digits recognition. In this case, y is an actual value of a handwritten digit, x is a scan of the handwritten digit, and φ is a function that takes the pixels of x and classifies them into one of ten possible outputs $0, ..., 9$.

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

Optimization framework (3) is more suitable for the purpose of solving dynamic economic models than 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, implicit optimization framework (3) can be also 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.

2.2 Approximating family: multilayer neural network

There are many parametric functions $\varphi(\cdot;\theta)$ that can be used for the purpose of approximating a solution to (3), including various polynomial families, splines, radial basis functions, etc. In particular, in our numerical experiments, we use piecewise linear and polynomial functions, among others. However, the class of multilayer neural networks plays so important role in the modern DL literature that it warrants a more detailed exposition.

An artificial neural network is a collection of connected nodes, called *artificial neurons*. Each artificial neuron can receive a signal from another neuron, process it and transmit a processed signal to other neurons connected to it. In Figure 1, a circle represents an artificial neuron, and an 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 linear weighing of x by a coefficients vector $\theta = (\theta_0, ..., \theta_n) \in \mathbb{R}^{n+1}$ (called *weights*) to obtain the non-activated output $\theta x_i = \theta_0 x_{i,0} + ... + \theta_n x_{i,n}$ and by activating θx with an activation function $\tau(\cdot)$, i.e., $\tau(\theta x)$. A coefficient θ_i controls the strength of a signal passing from one node to another. Under some activation functions, there exists a threshold that determines whether the signal is sent in case the aggregate signal overpasses such a threshold.

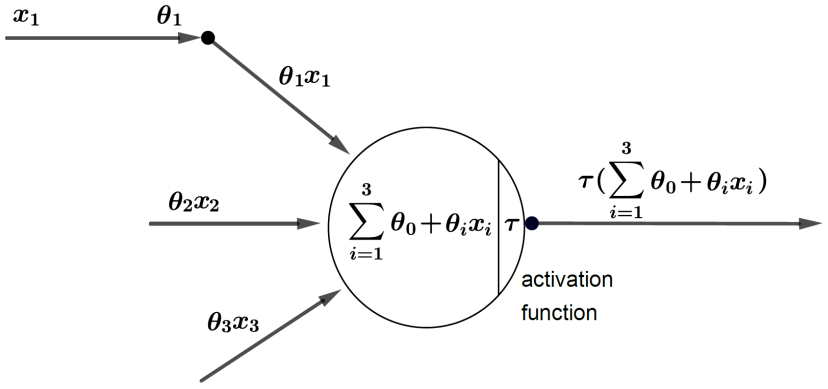


Figure 1. An artificial neuron.

Artificial neurons are aggregated into layers. There are three kinds of layers: the first layer – an *input layer*, the last layer – an *output layer*, and intermediate layers between the input and output layers – *hidden layers*. In a fully connected feed-forward neural network, the neurons of one layer are connected with all neurons of the previous layer. Distinct layers may use different activation functions (i.e., they can perform different kinds of transformations of their inputs).

The predicted output of the output layer is a highly non-linear function of the input layer. The presence of hidden layers allows us to combine information in a more abstract way and makes neural-network approximations more flexible, compared to the case of standard approximations in which inputs and outputs are related directly (e.g., standard polynomial approximation). To use GD-type of methods, we are to compute the gradient of the objective $\Xi_n(\theta)$ with respect to the parameters vector θ. In the context of deep learning, this is done by the chain rule using a technique called *back propagation*; see Appendix A for details.

Neural networks are called *universal approximators* and possess several appealing features that can push the feasibility frontiers in economics, in particular, they are: (i) linearly scalable, i.e., the number of parameters grows linearly with dimensionality; (ii) robust to multicollinearity and can perform model reduction automatically; and (iii) suitable for approximating highly nonlinear environments including kinks, discontinuities, discrete choices, switching, etc. These features will be critical for solving a high-dimensional Krusell and Smith (1998) model in Section 5.

2.3 Training the machine: deep learning

For simple approximation functions, training the machine to solve $\min_{\theta} \Xi_n(\theta)$ can be trivial. For example, for a linear approximation function $\varphi(x;\theta) = \theta x$, the solution is the familiar ordinary least squares estimator that can be derived in a closed form $\theta = (x'x)^{-1} x'y$. But for more complex approximating functions, such as multilayer neural networks, training is performed numerically and can be complicated and costly. Training the multilayer networks is often referred to as *deep learning* because such networks have complex interconnected topologies with the coefficients and weights buried in multiple layers.

In particular, computing the gradient of the expectation function $\nabla_\theta \Xi_n(\theta)$ can be expensive when the data size n is large. To reduce the cost, a popular approach is a stochastic batch gradient descent (BGD) method which constructs the gradient on a small random subset of data with $n' \ll n$ data points,

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

where a subset $(\omega_1, ..., \omega_{n'})$ is called *batch*, ∇_θ is 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 $n' = 1$. In the former case, we use all data points for constructing the gradient and thus, the BGD method is equivalent to the conventional gradient descent (GD) method. In the latter case, the BGD method is equivalent to a stochastic gradient (SG) method which approximates the expectation function with the value of such a function in one randomly chosen data point ω_k, i.e., $\nabla_\theta E_\omega \left[\xi\left(\omega;\theta_k\right)\right] \approx \nabla_\theta \xi\left(\omega_k;\theta_k\right)$. While such approximation can be very imprecise on each given step, SG is 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 Appendix B, we provide a detailed discussion of gradient descent methods and their convergence properties.

2.4 Summarizing up: an DL algorithm for optimization

We now summarize the techniques developed in Sections 2.1–2.3 in the form of an DL computational algorithm. We also add a discussion of the hyperparameters (denoted by λ) such as regularization techniques dealing with overfitting and ill conditioning, for example, Tykhonov and LASSO regularization techniques.

Algorithm 1. *DL algorithm for supervised learning.*

Step 1. Initialize the algorithm.

i). Set up an expected risk $\Xi(\theta) = E_\omega\left[\xi\left(\omega;\theta\right)\right]$.
ii). Define approximation $\varphi\left(\cdot,\theta\right)$ for φ, where $\theta \equiv [\vartheta, \lambda]$ and ϑ and λ are the approximation coefficients and hyperparameters of the algorithm, respectively.
iii). Define an empirical risk $\Xi_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\xi\left(\omega_i;\theta\right)$.
iv). Fix convergence criteria c_{inn} and c_{out} for inner and outer loops, respectively.
v). Split the data into 3 samples for constructing a solution (Sample 1), for validation (Sample 2) and for evaluating the accuracy (Sample 3).

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

Outer loop (validation on Sample 2): Fix the hyperparameters λ.
Inner loop (approximation on Sample 1): Fix the approximation coefficients ϑ.
Use data from Sample 1 to evaluate $\nabla_\vartheta \xi\left(\omega_i;\theta\right)$ (SGD or BGD) and update ϑ.
End the inner loop if the convergence criterion c_{inn} is reached.
Use data from Sample 2 for validation and update λ.
End the outer loop if if the convergence criterion c_{out} is reached.

Step 3. Assess the accuracy of constructed approximation $\varphi\left(\cdot,\theta\right)$ on Sample 3.

What is the role of regularization? Fitting an approximating function $\varphi\left(\cdot,\theta\right)$ on the training Sample 1 insures the best fit to the data on that specific sample but not necessarily on other samples. Regularization increases the fit on another (validation) Sample 2 at the cost of reducing the fit on training Sample 1 and ensures that the resulting approximation is not critically determined by a specific set of points used for training.

Let us consider, for example, Tykhonov regularization in a linear regression model $y = \theta x$. Instead of minimizing $\ell\left(\theta x, y\right) = \left(y - \theta x\right)^2$, we augment the loss function to include a penalty on the coefficients size $\widetilde{\ell}\left(\theta x, y, \lambda\right) = \left(y - \theta x\right)^2 + \lambda\theta^2$. If there is an ill conditioning, (e.g. multicollinearity) the penalty parameter λ helps us discriminate among multiple solutions with similar fit in favor of more stable solutions with smaller coefficients; see Judd et al. (2011) for a review of regularization techniques for dealing with ill conditioned inverse problems.

As we will see, solution methods for dynamic economic models will bring additional hyperparameters. Such methods lead to objective functions in the form of weighted averages of multiple objectives with unknown weights; and we will use the same procedure for identifying the weights as the one we use for identifying other hyperparameters such as the unknown regularization parameters.

2.5 The main novelty: computational technology!

An objective function in (3) and Algorithm 1 look fairly unsophisticated. It might be disappointing to see that machine learning is essentially the familiar gradient descent optimization. On the other hand, it is absolutely remarkable that so many real-life problems can be solved in that way.

There is not much methodological novelty in the DL framework. The gradient descent method is known since Newton and its stochastic optimization version was developed in early 1950th. Neural networks are also discovered long ago; see Rosenblatt (1958). There are even remarkable early applications of neural networks to solving dynamic economic models, see Duffy and McNelis (2001).

However, machine learning is so technology intensive that DL methods were put aside until platforms like TensorFlow or Pytorch had been developed to facilitate their implementation.[3] Such platforms provide a graph representation of operations, which can be manipulated and optimized automatically. In particular, gradient computations are performed using automatic differentiation in a numerically stable way, without any user input. The elements of the graphs, the so called *tensors*, are nothing else than multidimensional

[3] TensorFlow is defined as a dataflow programming software, or a ML platform, while Pytorch that has essentially the same functions is called DL software.

arrays manipulated by efficient vectorized symbolic engines. This is particularly useful if one wants to evaluate the same function with many draws of shocks for computing conditional expectation.

In fact, Monte Carlo simulations are essentially single instruction multiple device (SIMD) calculations, and they are well-suited to modern hardware. For instance, high-end GPUs feature thousands of powerful CUDA cores, which can all operate at the same time. Google has developed its own TPU units too. Also, commercial interest is high and the surge of cloud computing has made it possible to rent the vast amounts of computing power, e.g., Amazon.

3 Casting dynamic economic models into expectation functions

In Section 2, we described DL technology that can effectively optimize expectation functions. But to make use of that technology, we must convert dynamic economic models into expectation functions. In this section, we show how this can be done for three key objects of economic dynamics: i) lifetime reward, ii) Bellman equation, and iii) Euler equation. We also outline the challenges that the economists face when adapting the DL tools to their applications.

3.1 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.

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,*

$$m_{t+1} = M(m_t, \epsilon_t). \tag{5}$$

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,

$$s_{t+1} = S(m_t, s_t, x_t, m_{t+1}). \tag{6}$$

The choice x_t satisfies the constraint in the form

$$x_t \in X(m_t, s_t). \tag{7}$$

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

$$\max_{\{x_t, s_{t+1}\}_{t=0}^{\infty}} E_0 \left[\sum_{t=0}^{\infty} \beta^t r(m_t, s_t, x_t) \right], \tag{8}$$

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 the real set, so that the transition and reward functions are defined for any succession of choices $x_t \in \mathbb{R}^{n_x}$.

Definition 3.2 (Decision rule) *A decision rule is a function $\varphi : \mathbb{R}^{n_m} \times \mathbb{R}^{n_s} \to \mathbb{R}^{n_x}$, such that $x_t = \varphi(m_t, s_t) \in X(m_t, s_t)$. An approximate decision rule is a member of a family of functions $\varphi(\cdot; \theta)$ parameterized by a real vector θ, such that $\varphi(m_t, s_t; \theta) \in X(m_t, s_t)$.*

We do not assume the smoothness of the approximation function and 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.

Objective 1: Lifetime-reward maximization

We first focus on the method that maximizes the lifetime reward directly. We approximate the infinite-horizon problem with a finite-horizon solution by truncating the model at some finite $T < \infty$. Luckily, the objective function (8) has the form of expectation function $E_0[\cdot]$, i.e., it appears exactly in the form (3) that is necessary for Algorithm 1. However, the above formulation implies that the lifetime reward is to be maximized for just one fixed initial condition (m_0, s_0), whereas economists typically want the solution that is valid for any initial condition (state) within a given domain. Therefore, we reformulate the problem so that initial condition (m_0, s_0) is drawn randomly from the domain on which we want the solution to be accurate. The resulting expectation operator $E_0[\cdot]$ includes two types of randomness, one is a random state (m_0, s_0) and the other is a random sequence of future shocks $(\epsilon_1, ..., \epsilon_T)$. We call this operator *all-in-one expectation operator* because it summarizes all random variables in one place. The following definition shows such operator for the lifetime-reward maximization problem (8).

Definition 3.3 (All-in-one expectation operator for lifetime reward) *Fix time horizon $T > 0$, select 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 (8) associated with the rule $\varphi(\cdot;\theta)$ is given by*

$$\Xi(\theta) = E_\omega[\xi(\omega;\theta)] \equiv E_{(m_0,s_0,\epsilon_1,...,\epsilon_T)}\left[\sum_{t=0}^{T} \beta^t r(m_t, s_t, \varphi(m_t, s_t;\theta))\right], \tag{9}$$

where transitions are determined by equations (5), (6) and (7).

The expectation operator (9) can be viewed as theoretical risk (3) in DL optimization and can be used as an input to Algorithm 1.

Note that our all-in-one expectation operator has a remarkable distributive property: a single random draw or batch is used to evaluate all the expectation functions and stochastic gradients in the model at once. Whenever our all-in-one expectation operator uses n random draws, an integration method that constructs the two expectation functions separately would require n' draws for evaluating the expectation function with respect to (m_0, s_0) and n'' draws for evaluating the expectation function with respect to $(\epsilon_1, ..., \epsilon_T)$ – in total, $n' \times n''$ random draws. The associated difference in cost can be immense, especially, in high-dimensional applications, and the all-in-one expectation operator can reduce that cost dramatically.

Objective 2: Euler-residual minimization

We now 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 can follow from the lifetime reward maximization problem (8) or from an equilibrium problem and may include first-order conditions, equilibrium conditions, transition equations, constraints, etc.

Definition 3.4 (Euler equations) *Euler equations are a set of equations written in the form:*

$$E_\epsilon\left[f_j\left(m, s, x, m', s', x'\right)\right] = 0, \qquad j = 1, ..., J, \tag{10}$$

where $f_j : \mathbb{R}^{n_m} \times \mathbb{R}^{n_s} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_m} \times \mathbb{R}^{n_s} \times \mathbb{R}^{n_x} \to \mathbb{R}$, $E_\epsilon[\cdot]$ is an expectation operator with respect to next-period shock ϵ, and the optimal agent's choice satisfies constraints (5), (6) and (7) expressed in a recursive form $m' = M(m, \epsilon)$, $s' = S(m, s, x, m')$ and $x \in X(m, s)$, respectively.

The equations (10) are again defined just for a given state (m, s), and we want the solution to be accurate on a larger area of the state space, so we draw the state (m, s) randomly from that area. To identify

the solution, we minimize the squared sum of residuals in the Euler equations. Below, we define the corresponding expected risk to be used as an input to Algorithm 1.

Definition 3.5 (Euler-residual minimization) *Select a decision rule $\varphi(\cdot;\theta)$, and define the distribution of random variables $\omega \equiv (m,s)$. For given θ, the squared residuals in the Euler equations (10) associated with the rule $\varphi(\cdot;\theta)$ are given by*

$$\Xi(\theta) = E_{\omega}\left[\xi(\omega;\theta)\right] \equiv E_{(m,s)}\left\{\sum_{j=1}^{J} v_j \left(E_{\epsilon}\left[f_j\left(m,s,x,m'(\epsilon),s'(\epsilon),x'(\epsilon)\right)\right]\right)^2\right\}, \tag{11}$$

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

In the objective function (11), we again have two expectation operators $E_{(m,s)}[\cdot]$ and $E_{\epsilon}[\cdot]$. One integration approach we develop and test in the paper is a hybrid method that constructs two expectations separately, namely, we use Monte Carlo simulation for constructing the expectation function $E_{(m,s)}[\cdot]$ with respect to the state (m,s), and we use another integration methods for constructing the expectation function $E_{\epsilon}[\cdot]$ with respect to the future shocks ϵ, which can be either Monte Carlo or some deterministic integration method such as quadrature and monomial rules, sparse grids, low-discrepancy sequences, etc. We find that such a hybrid integration method is useful for small problems in which the construction of $E_{\epsilon}[\cdot]$ is cheap. For larger problems, it would be highly beneficial to construct all-in-one expectation operator – we will show this construction in Section 3.2.

Objective 3: Bellman-residual minimization

We now consider Bellman-equation formulation of the problem (8).

Definition 3.6 (Bellman equation) *Value function $V : \mathbb{R}^{n_m} \times \mathbb{R}^{n_s} \to \mathbb{R}$ associated with the problem (8) satisfies:*

$$V(m,s) = \max_{x,s'}\left\{r(m,s,x) + \beta E_{\epsilon}\left[V\left(m'(\epsilon),s'(\epsilon)\right)\right]\right\}, \tag{12}$$

subject to constraints (5), (6) and (7) expressed in a recursive form as $m' = M(m,\epsilon)$, $s' = S(m,s,x,m')$ and $x \in X(m,s)$, respectively.

Under the standard assumptions about r, M, S and X, the solution to (12) exists and is unique. It consists of policy and value functions that make the left and right sides of the Bellman equation coincide. Similar to the Euler-equation method, we can find the solution by minimizing the squared residuals in the Bellman equation (12) over the given area of the state space. However, the Bellman equation has also a nontrivial max operator which requires nested optimization, namely, it requires finding a maximum within a minimization cycle:

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

There are three main approaches in the literature to constructing a solution to the maximum operator in (12), including, i) direct optimization such as a grid search or numerical solver; ii) the first-order conditions (FOC); and iii) the envelope conditions (EC). We can formulate these three procedures, respectively, in terms of the following objective functions to minimize:

$$\zeta^{\max} = -\left\{r(m,s,x) + \beta E_{\epsilon}\left[V\left(m'(\epsilon),s'(\epsilon);\theta_1\right)\right]\right\}, \tag{13}$$

$$\zeta^{FOC} = \left\{r_2(m,s,x) - \beta E_{\epsilon}\left[V_2\left(m'(\epsilon),s'(\epsilon);\theta_1\right)\right]\right\}^2, \tag{14}$$

$$\zeta^{EC} = \left\{V_2(m,s;\theta_1) - r_2(m,s,x)\right\}^2, \tag{15}$$

where f_i denotes a first-order partial derivative with respect to the ith argument of function f. The objective functions (13) and (14) correspond to the conventional value function iteration (VFI) and the last objective function corresponds to the envelope condition method introduced in Maliar and Maliar (2013) and developed in Arellano et al. (2016).[4]

The value iterative methods existing in economic dynamics minimize the objective problems (13)–(15) inside the iterative cycle the Bellman equation; see Coleman et al. (2018) for a review and comparison analysis. Such nested optimization is expensive, which makes the conventional value-iterative methods prohibitive in high-dimensional applications.

We therefore introduce a different value iterative method that is more in line with our DL framework, namely, we combine a minimization of the residuals in the Bellman equation with a minimization of one of the objectives (13)–(15) that characterize the solution to the maximum operator.

Definition 3.7 (Bellman-residual minimization) *Select 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 (12) associated with $V(\cdot;\theta_1)$ and $\varphi(\cdot;\theta_2)$ are given by*

$$\Xi(\theta)=E_{\omega}\left[\xi(\omega;\theta)\right]\equiv \\ E_{(m,s)}\left\{V(m,s;\theta_1)-r(m,s,x)-\beta E_{\epsilon}\left[V\left(m'(\epsilon),s'(\epsilon);\theta_1\right)\right]\right\}^2+vE_{(m,s)}\left\{\zeta(m,s,\epsilon;\theta)\right\}, \quad (16)$$

where $\zeta\in\left\{\zeta^{\max},\zeta^{FOC},\zeta^{EC}\right\}$ and $v>0$ is an exogenous relative weight of the two objectives (ζ^{FOC} and ζ^{EC} can contain multiple conditions, which means that v is a vector that includes multiple weights).

The objective function (16) contains, as a special case, the standard value iterative methods that nest the maximum operator inside the iterative cycle of the Bellman equation. Specifically, for each iteration on value function $V(m,s;\theta_1)$ in the Bellman equation, we compute the decision rule $\varphi(\cdot;\theta_2)$ by minimizing one of the objectives (13)–(15) – in that case, we simply set $v=0$ because the second objective is already minimized and thus, it does not need to be included into the objective function (16) for the DL simulation.

3.2 All-in-one expectation operators for Euler and Bellman equations

In our previous analysis, we constructed all-in-one expectation for the lifetime reward maximization problem (9) but not for the Euler- and Bellman-residual minimization problems (11) and (16), respectively. This is because in the former case, the expectation operators are related linearly, so they are simple to merge together $E_{(m_0,s_0)}\left[E_{(\epsilon_1,...,\epsilon_T)}r(\cdot)\right]=E_{(m_0,s_0,\epsilon_1,...,\epsilon_T)}\left[r(\cdot)\right]$. However, in the latter case, $E_{\epsilon}[\cdot]$ is squared, so the two expectation cannot be naturally combined $E_{(m,s)}\left(E_{\epsilon}\left[f_j(m,s,\epsilon)\right]\right)^2\neq E_{ms}E_{\epsilon}\left[f_j(m,s,\epsilon)^2\right]$. As a result, we cannot use the stochastic gradient with respect to all random variables because it is biased $E_{(m,s)}\left(E_{\epsilon}\left[\nabla f_j(m,s,\epsilon)\right]\right)^2\neq\nabla E_{(m,s)}\left(E_{\epsilon}\left[f_j(m,s,\epsilon)\right]\right)^2$.

An important contribution of the present paper is to offer a technique that allows us to combine the expectation functions $E_{ms}[\cdot]$ and $E_{\epsilon}[\cdot]$ in a single expectation operator in the presence of squares. To be specific, 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 for the two terms which yields

$$E_{\epsilon_1}\left[f(\epsilon_1)\right]E_{\epsilon_2}\left[f(\epsilon_2)\right]=E_{(\epsilon_1,\epsilon_2)}\left[f(\epsilon_1)f(\epsilon_2)\right]. \quad (17)$$

With this approach, we are able to write the Euler-residual function (11) as a single expectation function $E_{ms}E_{\epsilon_1\epsilon_2}[\cdot]$ which is needed for making the stochastic gradient descent method unbiased in our DL solution framework.

[4] 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.

Definition 3.8 (Euler-residual minimization with all-in-one expectation operator) *Select 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 (10) associated with the rule $\varphi(\cdot;\theta)$ are given by*

$$\begin{aligned}\Xi(\theta) &= E_\omega\left[\xi(\omega;\theta)\right] \\ &\equiv E_{(m,s,\epsilon_1,\epsilon_2)}\left\{\sum_{j=1}^{J} v_j f_j\left(m, s, x, m'(\epsilon_1), s'(\epsilon_1), x'(\epsilon_1)\right) f_j\left(m, s, x, m'(\epsilon_2), s'(\epsilon_2), x'(\epsilon_2)\right)\right\}, \quad (18)\end{aligned}$$

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

We can also use the method of uncorrelated shocks in (17) for constructing one-in-all expectation function for the Bellman-residual minimization (16). We first focus on the method that represents the maximum operator with objective function (13) in which case (16) contains both linear and quadratic terms $E_\epsilon[\cdot]$ and $(E_\epsilon[\cdot])^2$. One possibility is to use one draw ϵ_1 for constructing the first term, and to use two draws ϵ_1 and ϵ_2 for constructing the second term. Alternatively, we can apply a simple transformation that eliminates the linear term. Namely, given that $(a-b)^2 - vb = \left(b - a - \frac{v}{2}\right)^2 - v\left(a + \frac{v}{4}\right)$, we can re-write objective (16) in the form shown below.

Definition 3.9 (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 $\theta \equiv (\theta_1, \theta_2)$, the squared residuals in the Bellman equations (12) associated with $V(\cdot;\theta_1)$ and $\varphi(\cdot;\theta_2)$* are given by

$$\begin{aligned}\Xi(\theta) = E_\omega\left[\xi(\omega;\theta)\right] \equiv \\ E_{(m,s,\epsilon_1,\epsilon_2)}\left\{\left[V(m, s;\theta_1) - r(m, s, x) - \beta V\left(m'(\epsilon_1), s'(\epsilon_1);\theta_1\right) - \frac{v}{2}\right] \times \right. \\ \left.\left[V(m, s;\theta_1) - r(m, s, x) - \beta V\left(m'(\epsilon_2), s'(\epsilon_2);\theta_1\right) - \frac{v}{2}\right] - v\left[V(m, s;\theta_1) + \frac{v}{4}\right]\right\}, \quad (19)\end{aligned}$$

$v > 0$ is an exogenous relative weight of the two objectives.

It is straightforward to combine the technique of two uncorrelated shocks in (17) with the other two objectives (14) and (15), namely, the former objective leads to a set of residuals (18) which must be evaluated under uncorrelated shocks ϵ_1, ϵ_2 jointly with the residuals in the Bellman equation and the latter objective does not contain shocks and thus requires no integration. To conclude, we derived all-in-one expectation operators for each method considered in the paper. We can thus evaluate all expectation functions and their gradients with just one draw or batch; this is true for problem with any dimensionality.

3.3 Dolo software: automating the construction of the objective functions

We have shown how to convert a canonical dynamic economic model into three objective functions which can be used in the context of our DL method summarized in Algorithm 1. Our objective functions are formulated in terms of (m, s, x, ϵ), which are vectors of exogenous and endogenous state variables, decision functions and shocks. Therefore, as a first step, we must represent an economic model in this notation. It is not hard to rename the variables in a simple one-agent growth model but it is a burdensome task for larger models, such as, e.g., central banking models that may contain hundreds of equations and unknowns. For some models, even the construction of state space is far from obvious if we do it manually. Finally, a translation of large models into the required (m, s, x, ϵ) format can lead to additional errors and bugs.

We automate the construction of the objective functions by using Dolo software.[5] This is an open-source platform which is similar to Dynare but has a collection of routines for constructing global nonlinear

[5] It is developed by Pablo Winant; see https://dolo.readthedocs.io/en/latest.

solutions, in addition to perturbation methods. We will not use the computational capacities of Dolo but use it as a convenient interface for preprocessing and reformatting the model. Dolo uses a classification scheme of variables and equations that coincides with the one we use in the paper. In particular, Dolo uses the same symbols for exogenous and endogenous state variables (m and s), control variables (x), reward functions (r), and value functions (v). After we introduce the equations into Dolo in the format "yaml", it will produce a generic Python code with the model's equations that can be directly used as an input to our model-free DL optimization routine written in TensorFlow.

3.4 Data science versus economic dynamics

We have shown that economic models can be reformulated as DL problems but there are some important differences between the data science and economic dynamics. We discuss such differences below, and we also explain the relation of our analysis to the literature.

Learning is always deep in economic models. In data science, the learning is called "deep" because the neural networks have interconnected topologies with multiple layers of the 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 under 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; see entry v) in Step 1 of Algorithm 1. In economic dynamics, we just need to fix the distribution for random draws. When solving dynamic models, we have the ability to simulate the model at will and we can generate as much 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. 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 first few draws antithetic can still bring a considerable increase in accuracy.

Validation for identifying the hyperparameters. Objective functions (11), (16), (18) and (19) contain the unknown weight parameters v. How can we deal with such parameters? The common approach to calibrating these and other hyperparameters (i.e., degree of regularization) is validation: we assume some values of such parameters, find the solution, check the accuracy out of sample (i.e., on a new sample) and iterate, until the most accurate solution out of sample is obtained.

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 all-in-one expectation version. For the lifetime reward we only have the version with all-in-one expectation operator. But we can also implement a

method that computes the two expectations separately, for example, one that uses Monte Carlo integration for evaluating $E_{(m_0,s_0)}[\cdot]$, and 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 $\pm\epsilon$, result 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 (9) (11), (16), (18) and (19) 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 an underdeveloped 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.

Is this the best possible AI technology for solving economic models? Our solution framework was designed to take advantage of existing DL technology. "But is this the best possible technology for solving dynamic economic models?" – the answer to this question is not so 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 laid in 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). 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. However, the DL technology that we employ is highly scalable, ubiquitous, optimized and free of errors. It is implemented using the state-of-the-art software and hardware. Taken together, these advantages can compensate for potential inefficiencies.

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 a 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 an 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 but 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; see Section 5 for examples and discussion.

3.5 Connection to the literature

In this section, we discuss the connection between our analysis and three types of learning methods in the data science – supervised, unsupervised and reinforcement learning. Interestingly, there are numerous AI-like approaches in computational economics that were discovered independently or even preceded the AI approaches in the data science.[6]

3.5.1 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 is dated back to Duffy and McNelis (2001) who use neural networks for parameterizing decision functions in a growth model. Recently, multilayer neural networks are used by Duarte (2018) for approximating value function solving continuous-time second-order difference equation; by Fernández-Villaverde et al. (2018) or approximating the 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 parameterize expectations algorithm (PEA) of Den Haan and Marcet (1990); and by Lepetyuk et al. (2019) for solving a large-scale central banking ToTEM model of the Bank of Canada.

However, interpolation does not fully utilize the capacities of the existing AI technology. Here, such 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 is 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 an 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). Like Judd (1992), we formulate a least-squares problem in which we minimize the squared sum of residuals in the Euler equation by using a gradient-descent approach. We differ in that we use deep neural networks instead of polynomial functions; we use a random grid instead of conventional fixed grid; and we use all-in-one expectation operator instead of deterministic integration methods. Taken together, these novel elements make it possible to combine all expectation operators into one to fully benefit from the DL framework.

[6] 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.

In turn, PEA solves the least-squares problem on simulated series instead of a grid, and it uses fixed-point iteration instead of gradient-descent methods. Like us, PEA possesses the property of merging the expectation operators into one but it is achieved via a different mechanism, namely, when iterating on 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 which is not easily parallelizable; in turn, we need only few random grid points and can fully benefit from the modern distributive computing. Also, a recent paper by Azinovic et al. (2019) presents a solution method which is similar to the Euler equation method which we introduced in 2018th conference presentations and which we developed here; see *https://lmaliar.ws.gc.cuny.edu* for our 2018 CEF and ESAM presentations. Azinovic et al.(2019) solve a life-cycle model with 60 state variables but assume a finite set of shocks abstracting thus from the issues of integration and unbiased stochastic gradient that play a key role in our analysis; furthermore, we solve problems with higher dimensinality such as Krusell and Smith's (1998) model with 2000 state variables. 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 maximization and minimization operators into a single optimization step, and second, we implement integration with two uncorrelated shocks in a way that leads to all-in-one expectation operator. However, our most important contribution does not consist in developing some new algorithms but in showing that dynamic economic models can be treated by using the same model-free DL technologies that the scientific community uses in many other fields, leading to truly break-ground applications.

3.5.2 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 or similar, etc. There are examples of applications of unsupervised learning in the economic literature, for instance, Judd et al. (2011) analyze a variety of regularization techniques that can be classified as unsupervised learning; Maliar and Maliar (2015) use clusters and epsilon-distinguishable sets; Judd et al. (2017) and Coleman et al. (2018) use low-discrepancy (quasi-Monte Carlo) sequences, etc. Moreover, there are papers that combine supervised learning with unsupervised learning. In particular, Lepetyuk et al. (2019) 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 used unsupervised learning just as a step of the conventional solution methods, whereas we use DL methods and modern data platforms to produce the entire solution to the model. In fact, Azinovic et al. (2019) interpret their analysis as a version of unsupervised learning which is another possible interpretation given a tight connection between supervised and unsupervised learning discussed in Section 2.1.

3.5.3 Reinforcement learning

Reinforcement learning 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 Sutton and Barto (2018) for a review of RL literature; see Powell (2010) for a related field of approximate dynamic programming; and see Lepetyuk and Jirniy (2012) for early remarkable application of RL for solving Krusell and Smith (1998) model.

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. But in our case, learning is fully offline: after a batch of simulations, decisions are adjusted according to the simulation feedback and new simulations are run again. 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.

4 Numerical analysis of the consumption-saving problem

In this section, we solve a consumption-saving model with occasionally binding borrowing constraint. We elaborate the results for three objective functions: the lifetime reward, and the Euler- and Bellman-equation residuals. Our experiments are designed to illustrate the advantages of the proposed DL method, in particular, its ability to accurately approximate kinks, its capacity to handle ill-conditioned problems and its scalability. We also discuss some problems that the users may run into when using the DL technologies.

4.1 The consumption-saving problem

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

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

$$\text{s.t. } w_{t+1} = re^{\varrho_t}\left(w_t - c_t\right) + e^{y_t}e^{p_t(1-\mu)}, \tag{21}$$

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

where c_t and w_t are consumption and the beginning-of-period cash-on-hand, respectively; $\{y_t, p_t, \varrho_t, \chi_t\} \equiv z_t$ is a vector of exogenous state variables, which includes a temporary income shock y_t, a permanent income shock p_t, an interest-rate shock ϱ_t and a preference shock χ_t; $\mu \in \{0,1\}$ is an indicator function; u is a utility function, which is assumed to be strictly increasing and concave; $\beta \in [0,1)$ is a subjective discount factor; $r \in \left(0, \frac{1}{\beta}\right)$ is a (gross) constant interest rate, and initial condition (z,w) is given. Note that the borrowing limit in (22) is set to zero without loss of generality. For each exogenous shock $z \in \{y, p, \varrho, \chi\}$, we assume an AR(1) process,

$$z_{j,t+1} = \rho_j z_{j,t} + \sigma_j \epsilon_{j,t} \text{ and } \epsilon_{j,t} \sim \mathcal{N}\left(0,1\right), \tag{23}$$

where $\left|\rho_j\right| < 1$ and $\sigma_j > 0$. A solution to the model can be characterized by the Bellman equation

$$V\left(z,w\right) = \max_{c,w'}\left\{u(c) + \beta E_\epsilon\left[V\left(z',w'\right)\right]\right\} \qquad \text{s.t.} \qquad (21)\text{–}(23), \tag{24}$$

Also, the solution can be characterized by the Kuhn-Tucker conditions

$$c - w \leq 0,\ h \geq 0 \text{ and } (c-w)\,h = 0, \tag{25}$$

where $h \equiv u'(c)e^{\chi-\varrho} - \beta r E\left[u'\left(c'\right)e^{\chi'}\right]$ is a Lagrange multiplier.

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 $\beta = 0.9$, $r = 1.04$, and $\sigma_y = 0.1$. Our baseline uni-shock model has just a temporary i.i.d. income shock y (i.e., $\rho_j = 0$ for $j = 1, ..., 4$; $\sigma_j = 0$ for $j = 1, 2, 3$ and $\sigma_4 > 0$). Having just one (endogenous) state variable allows us to illustrate the decision function by two-dimensional plots. For the multi-shock model, we use: $\rho_y = 0.9$ 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 $\sigma_\chi = 0.01$.

4.2 Computational details

We wrote the code in Python using a Google TensorFlow library 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. Training is

performed over L_2^2 using a mini-batch gradient descent method and a version of the stochastic gradient descent algorithm, called *ADAM*, in which a learning rate is different for each coefficient; in both cases, we consider 16 random draws; see Appendix B for a discussion of training methods.

Initial points for state variables are drawn from: (i) an ergodic distribution of exogenous processes for shocks z; (ii) a uniform distribution for cash-on-hand in an interval $[w_1, w_2] = [0.1, 4]$.

We parameterize a decision function of a consumption ratio $\frac{c_t}{w_t} \equiv \zeta_t = \sigma(\zeta_0 + \varphi(z_t, w_t; \theta))$, where $\varphi(\cdot)$ is a neural network $\sigma(x) = \frac{1}{1+e^{-x}}$ is a sigmoid function, so that ζ_t is bounded to be in an interval $[0, 1]$. To initialize the coefficients (ζ_0, θ), we assume $\zeta_0 = 0.95$, and we draw θ randomly; in particular, we use a "*he*" uniform distribution for biases and a "*glorot*" uniform distribution for the other (non-bias) coefficients of the hidden layers. We parameterize the value function with $V_t = V_0 + \varphi(z_t, w_t; \theta)$, where $\varphi(\cdot)$ is again a neural network. To initialize the coefficients (V_0, θ), we assume $V_0 = \frac{1}{1-\beta}\frac{c_{ss}^{1-\gamma}-1}{1-\gamma}$, and we perform a random initialization for θ, similarly to the decision-function case. For both the decision and value functions, a neural network φ contains two hidden layers, with each of the hidden layers including 32 (leaky) relu neurons; see our description in Appendix A.

In the training step, we fix the number of iterations to be $K = 50,000$. To evaluate the accuracy, we produce $1,024$ random draws and use the constructed approximate 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 for the uni-shock baseline case and a 100-node Monte Carlo integration method for the multi-shock case.

4.3 Objective 1: Lifetime reward

This lifetime reward objective is constructed as follows: Fix time horizon $T > 0$, select a decision rule $\varphi(\cdot; \theta)$, and define a random vector $\omega \equiv (z_0, w_0, \epsilon_1, ..., \epsilon_T)$ composed of an initial state and a realization of exogenous shocks drawn from the corresponding distribution. For a given random draw ω, we define the lifetime reward (20) associated with the rule $\varphi(\cdot; \theta)$ by

$$\Xi(\theta) = E_\omega[\xi(\omega; \theta)] \equiv E_\omega\left[\sum_{t=0}^{T} \beta^t u(c(z_t, w_t; \theta))\right], \tag{26}$$

where transitions are determined by the constraints (21)–(23).

Figure 2 displays the outcomes of training (on the horizontal axis, $K = 50,000$ appears as $4 \cdot \log_{10} 5$). We show the average Euler-equation residuals (denoted L_2) and lifetime reward in the left and right panels, respectively. We consider four training methods, namely, SGD with an updating parameter $\lambda \in \{0.05, 0.1, 0.5\}$ and ADAM.

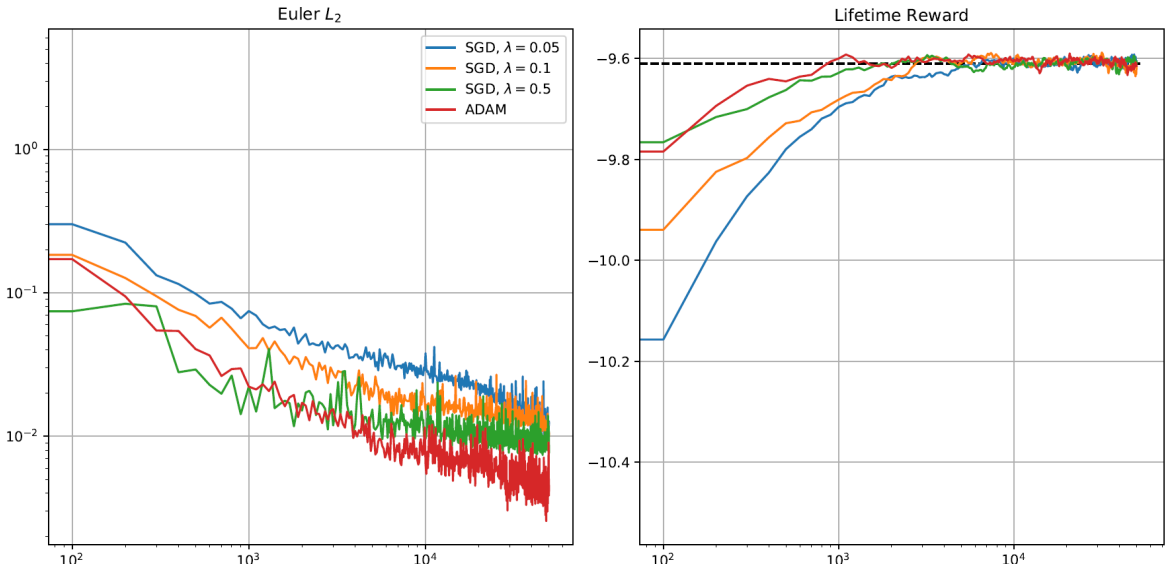


Figure 2. Training with a maximization of the lifetime reward in the baseline model.

As is seen from the first panel, the Euler-equation residuals are the largest for SGD with the slowest learning rate ($\lambda = 0.05$) and are the smallest for ADAM. In the latter case, they are of order $10^{-2.5}$ (i.e., 0.3%), which is quite low, given that we solve the model with a kink in the decision rule. In terms of the lifetime reward, all the training methods gradually converge to the level of -9.61 shown with a dashed line, which is the level of the lifetime reward in the optimum computed with a very accurate projection method. ADAM reaches slightly higher value of the lifetime reward than the other training methods.

4.4 Objective 2: Euler-equation method with Kuhn-Tucker conditions

As a first step, we rewrite the conditions with inequality constraints in (25) as equations that hold with equality with the help of the Fischer-Burmeister (FB) function

$$\Psi^{FB}(a,b) = a + b - \sqrt{a^2 + b^2} = 0; \tag{27}$$

where $a = c - w$ and $b = c - u'^{-1}\left(\beta r e^{\varrho_t} E_\epsilon\left[u'\left(c'\right)\right]\right)$. The FB function is similar to the minimum function $\Psi^{\min}(a,b) = \min\{a,b\} = 0$ and leads to the solution $a \geq 0$, $b \geq 0$ and $ab = 0$ but it is differentiable; see, e.g., Jiang (1996) for a discussion of that function. Here, we expressed the terms a and b in comparable consumption units; in general, it might be necessary to add relative weights that reflect the importance of the two objectives a and b, i.e., to consider $\Psi^{FB}(a, vb)$.

The objective function for the Euler-equation-residual minimization is constructed as follows: Select a decision rule $\varphi(\cdot;\theta)$, draw random state $\omega \equiv (z,w)$ and define residuals in (27) by

$$\Xi(\theta) = E_\omega\left[\xi(\omega;\theta)\right] \equiv E_\omega\left[\Psi^{FB}\left(c - w, c - u'^{-1}\left(\beta r e^{\varrho_t} E_\epsilon\left[u'\left(c'\right)\right]\right)\right)\right]^2. \tag{28}$$

The results of training under the objective (28) are shown in Figure 3.

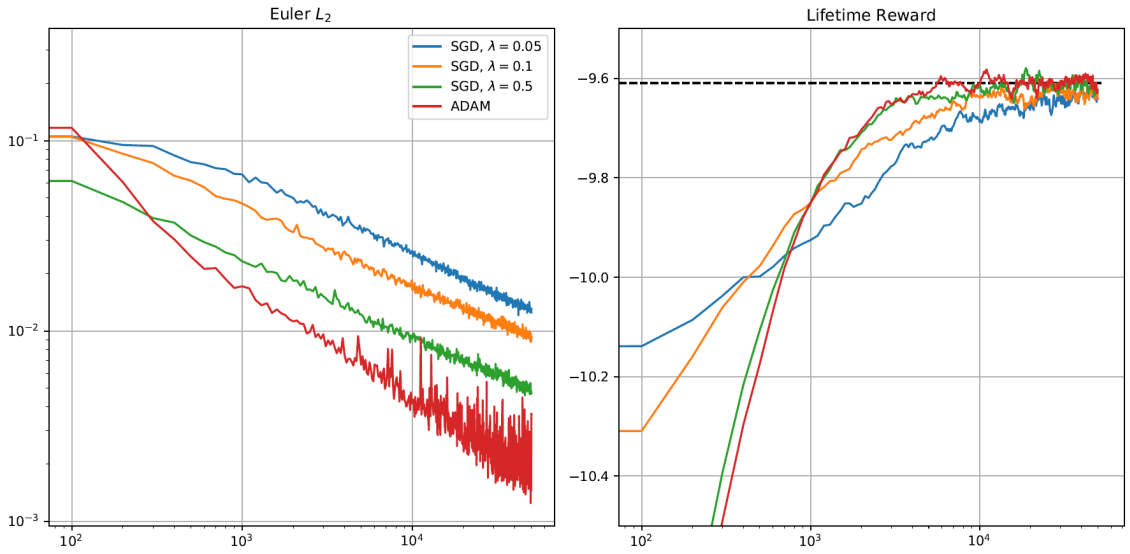


Figure 3. Training with a minimization of Kuhn-Tucker-conditions residuals in the baseline model.

As is evident from the first panel, ADAM is again most accurate, reaching 10^{-3} (i.e., 0.1% of the residuals), although there it has more variability over the training steps. In terms of the value of lifetime reward, ADAM appears to deliver the highest lifetime reward in the final stage of training but starts farther away the true value, relative to the other training methods; in particular, SGD with the lowest learning rate $\lambda = 0.05$ starts with some advantage in the initial stage of training. The comparison of Objectives 1 and 2 in Figures 2 and 3 shows that the ranking of the training methods depends on a specific objective used and can change over different training stages.

4.5 Objective 3: Bellman equation

For the Bellman equation residuals, we focus on the objective function (13) which does not assume differentiability. We select a value function $V(\cdot;\theta_1)$ and a consumption rule $\varphi(\cdot;\theta_2)$, draw random state (z,w) and define residuals in (24) by

$$\begin{aligned}\Xi(\theta) &= E_\omega\left[\xi(\omega;\theta)\right] \\ &= E_{(z,w)}\left[V(z,w;\theta_1) - u(c) - \beta E_\varepsilon V\left(z',w';\theta_1\right)\right]^2 - vE_{(z,w)}\left\{u(c) + \beta E_\varepsilon V\left(z',w';\theta_1\right)\right\}, \quad (29)\end{aligned}$$

Figure 4 plots the training process for Objective 3 with three values of the weight $v \in \{0.0005, 0.001, 0.005\}$ using ADAM.

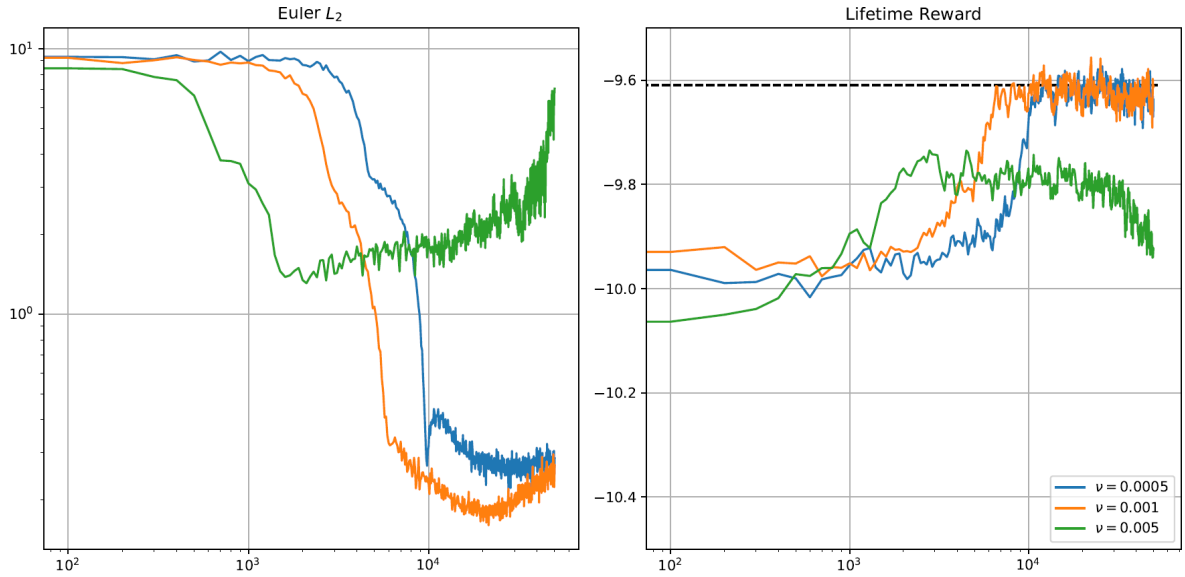


Figure 4. Training with a minimization of Bellman-equation residuals in the baseline model.

Recall that we treat the weight hyperparameter similar to regularization parameters: we try out several values to find the values that yields the most accurate solution. We see that smaller weights $v \in \{0.0005, 0.001\}$ produce the correct solution, while an excessively high value of v leads to nonconvergence: in that case, too much weight is put on approximating the maximum operator, at the expense of the poor fit in the Bellman equation leading to non-convergence.

Another finding in the figure is that 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 (13), as we do here, versus otherwise identical methods that use derivatives of the value function via first-order and envelope conditions given by (14), (15), respectively. They found that the methods that approximate the derivatives of V are far more accurate than the methods that approximate V alone. It is straightforward to extend our Bellman-equation method to include the FOCs: we just need to replace the second objective in (29) with squared Fischer-Burmeister (FB) function by setting $a = c - w$ and by computing b either from FOC of Bellman equation $b = c - u'^{-1}\left(\beta r e^{\varrho_t} E_\varepsilon V_2\left(z', k'; \theta_1\right)\right)$ or from the envelope condition $b = c - u'^{-1}\left\{V_2\left(z, k; \theta_1\right)\right\}$. We do not consider these more accurate versions of the Bellman-equation method because they are more related to the Euler-equation method; see Arellano et al. (2016) for a discussion. Instead, we turn to some interesting finding in our numerical experiments.

4.6 Decision rules: the role of the training method in accuracy

Let us look at the resulting decision rules. We focus on the objective (28) that corresponds to the Kuhn-Tucker conditions. We consider four training methods, namely, ADAM and SGD with $\lambda \in \{0.05, 0.1, 0.5\}$.

Figures 5 and 6 focus on the decision rules of consumption and a consumption share $\frac{c_t}{w_t} \equiv \zeta_t$, respectively. In both figures, we plot a normal-size decision rule, a zoomed-in decision rule, and an error in the corresponding variable (relative to the optimal solution, obtained with an accurate projection method). The consumption function is defined in the interval $\left[\frac{1}{10}, 4\right]$, which corresponds to cash-on-hand w.

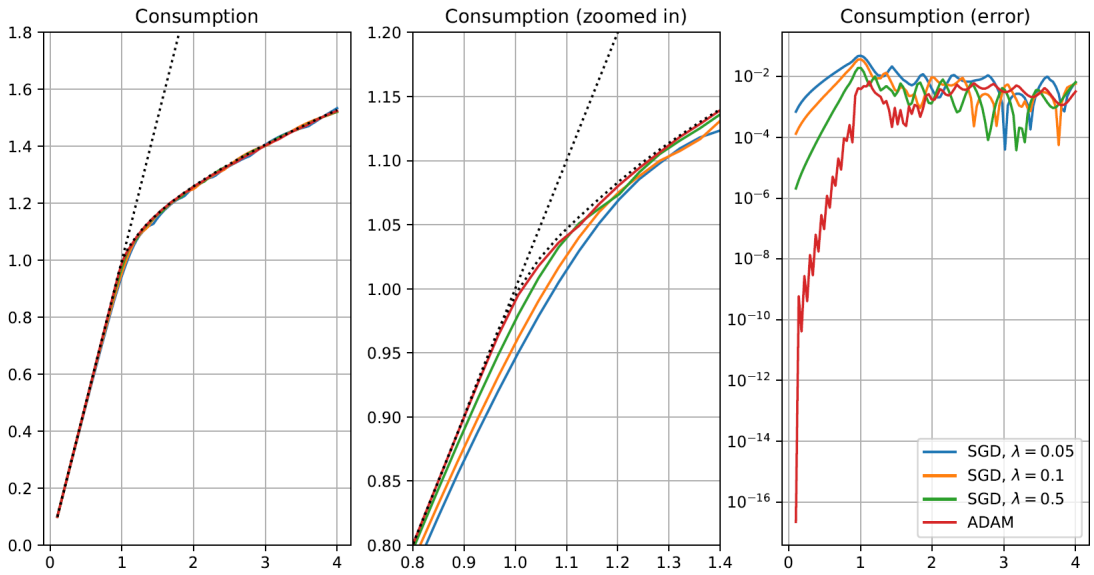


Figure 5. The consumption decision rule in the baseline model.

In the first panel of Figure 5, there is virtually no difference in the consumption decision rules across the training methods considered. A closer look in the second panel shows that there are larger (but still small) differences near the kink, with ADAM being the most accurate and SGD with the lowest λ being the least accurate. Nevertheless, it is remarkable how well neural networks were able to approximate the kink without any information on where the kink was!

In Figure 6, we plot the corresponding panels for the consumption shares.

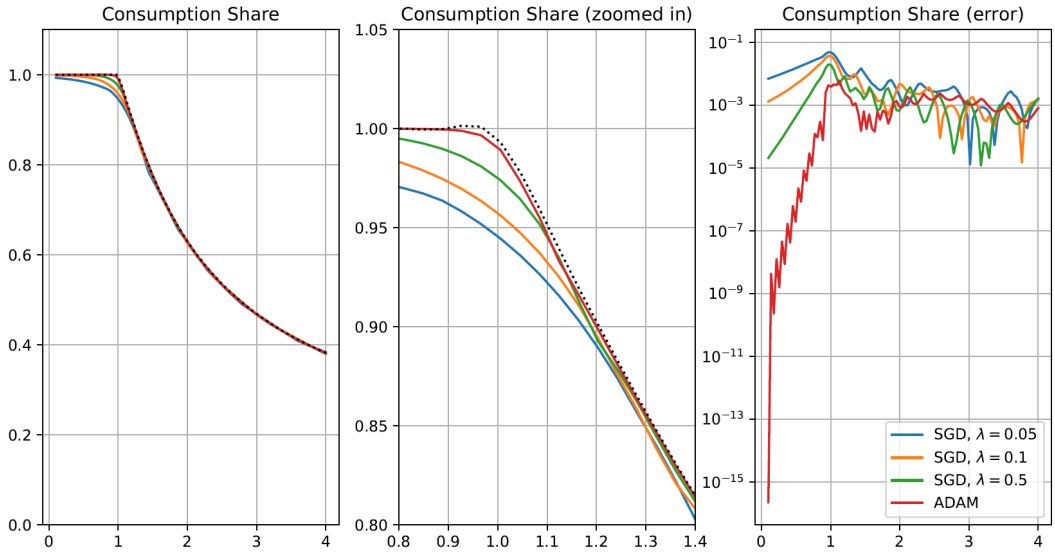


Figure 6. The consumption-share decision rule in the baseline model.

Overall, the tendencies for consumption shares in Figure 6 are similar to those we observed for consumption units in Figure 5 although the differences between the rules are somewhat larger. One visible finding in Figure 6 is that the ADAM remains to be extremely accurate in approximating the kink, while the SGD leads to less accurate solutions, in particular, when the updating rate λ is low.

4.7 Role of activation functions in a convergence to the right solution

The choice that also proved to be important for the results is the activation function. Recall that our benchmark activation function is relu (namely, each of the two hidden layers has 32 leaky-relu neurons). Now, we use a sigmoid function – another common choice in the DL literature. Surprisingly, such a small modification flipped over our results completely: now, the algorithm converged to a wrong solution! The corresponding results are shown in Figure 7.

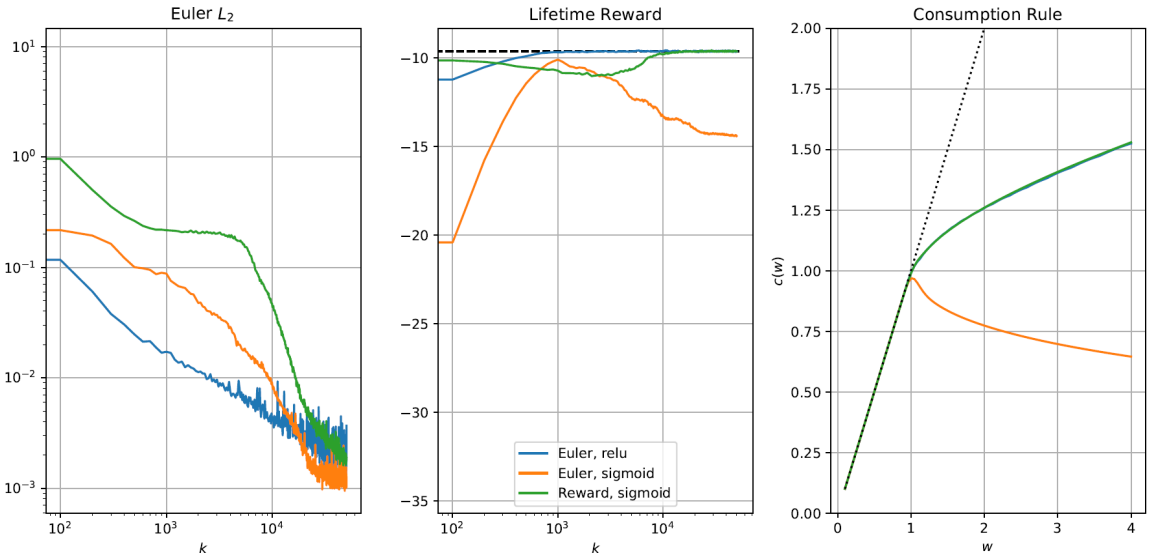


Figure 7. Sigmoid activation function.

In fact, the "wrong" solution is still a valid solution to the Kuhn-Tucker conditions but it is an unstable solution leading to an explosive path that violates the transversality condition. However, the wrong solution leads to residuals that are even smaller in size than those produced by the correct solution. Thus, we cannot detect this problem by looking at the residuals alone but we can do so by looking at the lifetime reward: in that way, we can discard the solution obtained under the sigmoid activation function.

Can sigmoid activation functions be just generically bad for approximating solutions in economic models? To answer this question, we also consider the lifetime reward maximization with sigmoid activators. Here, we got the correct solution in which the decision rule is accurate (and coincides with the one produced with relu). Our conclusion is that we need to have techniques for checking properties of the constructed solutions that go beyond conventional Euler equation residuals, such as the second-order conditions or the size of the objective functions. But such techniques are needed not just for our method but for all methods that rely on first-order conditions.

4.8 Multicollinearity, ill-conditioning and model reduction

The lifetime-reward objective function (26) is built on all-in-one expectation operator and it is directly suitable for high-dimensional applications. In Figure 8, we present the results obtained with reward maximization for the multistate model (20)–(23). We used 64 relu nodes in each of the two hidden layers, and the training method was ADAM.

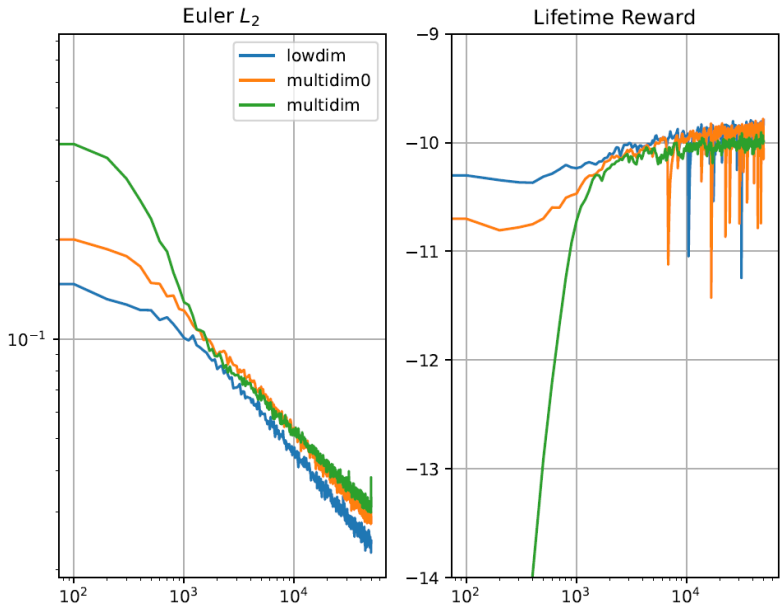


Figure 8. Training with a maximization of the lifetime reward in the multishock model.

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 one shock which is an income shock parameterized by $\rho_y = 0.9$. 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 that 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 uni-shock model but this difference is relatively small. This finding indicates that the proposed solution method is potentially tractable in problems with high dimensionality.

4.9 The Euler- and Bellman-equation methods with all-in-one expectation operator

In our previous implementation of the Euler- and Bellman-equation methods for the uni-shock model (28), we approximated the two expectation operators separately. We now show how to construct the all-in-one expectation operator for these two methods in the presence of Kuhn-Tucker conditions. To this purpose, we introduce a supplementary variable μ that represents the expectation function and rewrite (28) as a composite objective

$$E_{(z,w)}\left[\Psi\left(c-w,u'(c)-\mu\right)\right]^2+v\left[\beta re^{\varrho_t}E_\epsilon\left[u'\left(c'\left(\epsilon\right)\right)\right]-\mu\right]^2, \tag{30}$$

where v is an exogenous weight. Using the technique of two uncorrelated shocks (17), we then arrive to the objective function that combines integration with respect to z, w and ϵ

$$E_{\omega}\left[\xi\left(\omega;\theta\right)\right]=E_{(z,w,\epsilon_1,\epsilon_2)}\left\{\left[\Psi\left(c-w,u'(c)-\mu\right)\right]^2\right.$$
$$\left.+v\left[\beta re^{\varrho_t}\left[u'\left(c'\left(\epsilon_1\right)\right)\right]-\mu\right]\left[\beta re^{\varrho_t}\left[u'\left(c'\left(\epsilon_2\right)\right)\right]-\mu\right]\right\}. \quad (31)$$

(For Bellman-residual minimization, the all-in-one expectation operator (19) is directly applicable to the given problem with just a straightforward change of notation).

Figure 9 displays the outcome of training of (31) for the multi-shock model under the weight $v=0.5$. As before, we parameterize a decision function of the consumption ratio $\frac{c_t}{w_t}\equiv\zeta_t$ by a sigmoid function $\zeta_t=\sigma\left(\zeta_0+\varphi\left(z_t,w_t;\theta\right)\right)$, where $\varphi\left(\cdot\right)$ is a neural network. In addition, we parameterize a decision function of μ_t by an exponential function $\mu_t=\exp\left(\mu_0+\varphi\left(z_t,w_t;\theta\right)\right)$.

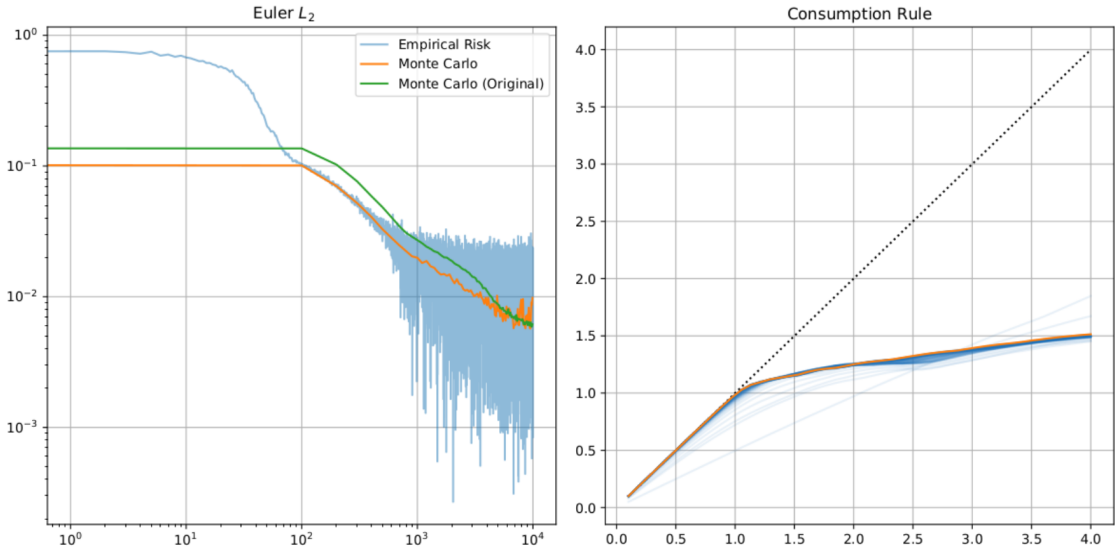


Figure 9. Expectations with all-in-one expectation: training with a minimization of Euler-equation residuals

In the figure, we show the first 10,000 iterations with the ADAM training method. On each iteration, we now use 512 random grid points (instead of 16). In the left panel, we show the Euler equation residuals which we evaluated by using a 10,000-point Monte Carlo integration method. "Monte Carlo (Original)" and "Monte Carlo" correspond to two objectives (28) to (31) that we evaluate in the test. We see that both objectives show very similar convergence patterns. In the right panel, we show how the consumption rules evolve along the iterations.

Finally, in Figure 10, we illustrate the scalability of the Euler-equation method. We vary the batch size from $N=8$ to $N=8192$ draws and we document accuracy and running time.

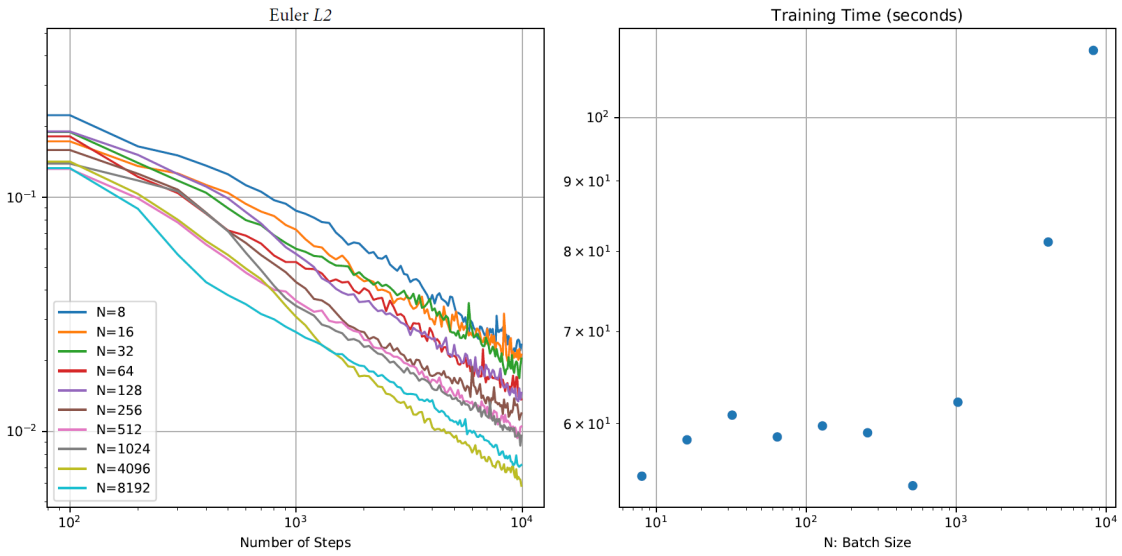


Figure 10. Effects of the batch size on the accuracy and speed.

In the left panel, we observe that a larger batch size leads to more accurate solutions although the convergence rate is similar for all batch sizes. In the right panel, we see that the training time changes roughly linearly with the batch size. Thus, the Euler-equation methods with uncorrelated shocks scale linearly with dimensionality; and the same is true for all the methods built on one-in-all expectation operator.

5 Model reduction in Krusell and Smith's (1998) analysis

We now use the DL method to solve a version of the Krusell and Smith (1998) model. We show that the DL simulation combined with the model reduction can be used to construct decision functions with thousands of state variables.

5.1 Krusell and Smith's (1998) model

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

$$\max_{\left\{c_t^i, k_t^i\right\}_{t=0}^{\infty}} E_0 \left[\sum_{t=0}^{\infty} \beta^t u\left(c_t^i\right)\right] \tag{32}$$

$$\text{s.t. } c_t^i + k_{t+1}^i = R_t k_t^i + W_t z_t^i, \tag{33}$$

$$z_{t+1}^i = \rho_z z_t^i + \sigma_z \epsilon_t^i \text{ with } \epsilon_t^i \sim \mathcal{N}(0,1), \tag{34}$$

$$k_{t+1}^i \geq 0, \tag{35}$$

where c_t^i, k_t^i are z_t^i consumption, capital and labor productivity, and initial condition $\left(k_0^i, z_0^i\right)$ is given. The production side of the economy is described by a Cobb-Douglas production function $z_t k_t^{\alpha}$, so that

$$R_t = 1 - d + z_t \alpha k_t^{\alpha - 1} \text{ and } W_t = z_t (1 - \alpha) k_t^{\alpha - 1}, \tag{36}$$

$$z_{t+1} = \rho z_t + \sigma \epsilon_t \text{ with } \epsilon_t \sim \mathcal{N}(0,1), \tag{37}$$

where $k_t = \sum k_t^i$ is aggregate capital, and z_t is an aggregate productivity shock. In general, the state space of the model includes the state variables of all agents $\left\{k_t^i, z_t^i\right\}_{i=1}^{\ell}$, as well as the aggregate shock z_t, i.e., so we must solve for the decision and value functions in terms of $2\ell+1$ state variables of $(\left\{k_t^i, z_t^i\right\}_{i=1}^{\ell}, z_t)$.

To make the model tractable, Krusell and Smith (1998) introduce a numerical approach that replaces the distributions of state variables with a finite set of their moments m_t, i.e., they approximate the state space by (k_t^i, z_t^i, z_t, m_t)–this approach proved to work remarkably well in a variety of models and applications. 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. Lepetyuk and Jirniy (2012) proposed a non-parametric reinforcement learning algorithm for solving that model. Recent numerical methods for solving Krusell and Smith's (1998) model include Ahn et al. (2017), Bayer and Luetticke (2018), Boppart et al. (2018), Fernández-Villaverde et al. (2018). These new developments in solution techniques are primarily motivated by the recent interest in modeling the effects of fiscal and monetary policies on distributions.

5.2 Deep learning solution algorithm

A distinctive feature of our DL framework is that it makes it possible to solve the Krusell and Smith (1998) model by working directly with the actual state space. We specifically parameterize the consumption share of agent i by $\frac{c_t^i}{w_t^i} \equiv \zeta_t^i = \sigma\left(\zeta_0 + \varphi\left(k_t^i, z_t^i, \left\{k_t^i, z_t^i\right\}_{i=1}^{\ell}, z_t; \theta\right)\right)$, where $\varphi(\cdot)$ is a neural network and $\sigma(x) = \frac{1}{1+e^{-x}}$ is a sigmoid function, which ensures that ζ_t^i is bounded to be in interval $[0,1]$. If agents are heterogenous in fundamentals, we need to approximate ℓ different individual decision functions, each of which has $2\ell+1$ dimensions. With symmetric agents, as in Krusell and Smith (1998), we need just one $2\ell+1$–dimensional decision function to characterize the choices of all ℓ agents.[7]

Objective function. We form the objective function that minimizes the Euler-residual using our technique of two uncorrelated shocks (17):

$$E_\omega\left[\xi\left(\omega;\theta\right)\right] = E_{(K,Z,z,\Sigma_1,\Sigma_2,\epsilon_{1,},\epsilon_2)}\left\{\left[\Psi\left(c^i - w^i, u'(c^i) - \mu^i\right)\right]^2 + v\left[\beta\left[u'\left(c^{i'}\left(\Sigma_1, \epsilon_{1,}\right)\right)\right] - \mu^i\right]\left[\beta\left[u'\left(c^{i'}\left(\Sigma_2, \epsilon_{2,}\right)\right)\right] - \mu^i\right]\right\}, \quad (38)$$

where $K = \left(k^1, ..., k^\ell\right)$, $Z = \left(z^1, ..., z^\ell\right)$ and z are random draws of the economy's state; $\Sigma_1 = \left(\epsilon_1^1, ..., \epsilon_1^\ell\right)$, $\Sigma_2 = \left(\epsilon_2^1, ..., \epsilon_2^\ell\right)$ are two uncorrelated random draws of individual productivity shocks; and $\epsilon_{1,}$, ϵ_2 are two uncorrelated random draws for the aggregate productivity shocks.

Solution algorithm: simulating a panel of heterogeneous agents. Our DL algorithm alternates between the solution and simulation steps. It proceeds as follows: i) draw initial productivity z_0 and initial distributions $\{K_0, Z_0\} = \left\{k_0^i, z_0^i\right\}_{i=1}^{\ell}$ and ; ii) compute aggregate capital k_0 and prices R_0, W_0; iii) train neural network to satisfy (38) for ℓ agents; iv) perform forward simulation to produce next-period productivity z_1 and distribution $\{K_1, Z_1\} = \left\{k_1^i, z_1^i\right\}_{i=1}^{\ell}$, and proceed iteratively until the convergence is achieved. As the machine is trained and the panel is simulated, the decision functions are refined jointly with the ergodic distribution.[8] Our method is similar in spirit to Krusell and Smith's (1998) algorithm but is simpler conceptually as it does not involve a separate approximation of the law of motion for aggregate variables. We just simulate the panel of heterogeneous agents, and we use the resulting distributions to infer the aggregate quantities and prices as the economy evolves in time.

[7] Since the true decision rule $\varphi(\cdot;\theta)$ is invariant to any permutation of $\left\{k_t^i, z_t^i\right\}_{i=1}^{\ell}$, 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.

[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.

Ergodic-set domain. When we solve the consumption-savings problem, we drew the state variables from a prespecified exogenous rectangular domain but for Krusell and Smith's (1998) model we produce state variables by simulating the economy forward. Why don't we use a fixed rectangular domain now? This is because the volume of rectangular domain is huge in high-dimensional problems, and it is prohibitively expensive to attain an accurate approximation everywhere on such a huge domain. In fact, only an infinitesimally small fraction of rectangular domain is generally visited in equilibrium in high-dimensional applications; see Judd et al. (2011) for a discussion. We take advantage of that fact by solving the model on simulated series–we restrict attention to much smaller ergodic-set domain in which the solution "lives". This strategy helps us deal with the curse of dimensionality.

Perfect multicollinearity. In the approximating function of the consumption share, we include the state variables of agent i twice,

$$\varphi^i = \varphi\left(k_t^i, z_t^i, \left\{k_t^i, z_t^i\right\}_{i=1}^{\ell}, z_t; \theta\right),$$

namely, they enter both as variables of agent i and as an element of the distribution. This repetition implies perfect collinearity 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 have shown earlier. Thus, even though it is possible to design a transformation that avoids a repetition of variables, it would require cumbersome and costly permutations, so we find it easier to keep the repeated variables.

Model reduction. We solve the models with up to $\ell = 1,000$ of 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 the ability to handle multicollinearity, neural networks possess another remarkable property: they automatically perform the model reduction. When we supply a large number of state variables to the input layer, the neural network condenses the information into 64 neurons of 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.

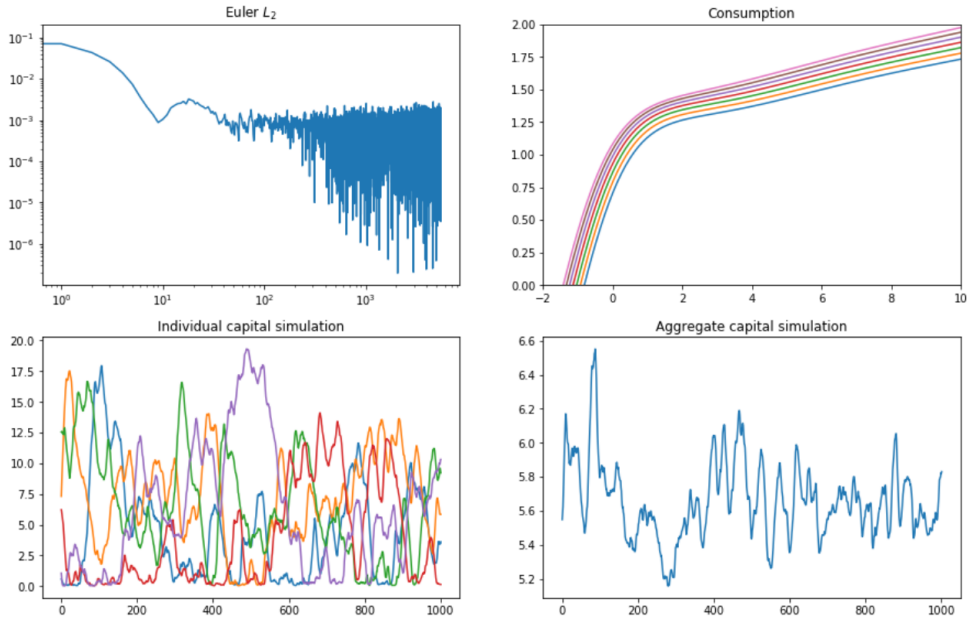
Krusell and Smith (1998) found one specific model reduction that works extremely well for their model, namely, they approximate the distribution of state variables with a finite set of moments. They found that in their model, just one moment – the mean of wealth distribution m_t – is a sufficient statistic for capturing all relevant information, which reduces their state space to just four state variables (k_t^i, z_t^i, z_t, m_t).

If Krusell and Smith's (1998) construction is the most efficient representation of the state space, the neural network will possibly find this representation as an outcome of training. However, the neural network will automatically consider many other possible ways of extracting the information that is contained in the distribution $\left\{k_1^i, z_1^i\right\}_{i=1}^{\ell}$ and condensing it in a relatively small set of hidden layers. The output of the machine can look like moments or some other statistics – we will not always be able to understand how the machine handles the information in the hidden layers but this fact does not prevent us from using this miraculous technology in applications.

Calibration and computational details. In the benchmark case, we parameterize the model by $u(c) = \frac{c^{1-\gamma}-1}{1-\gamma}$ with a risk-aversion coefficient of $\gamma = 1$, and we assume $\beta = 0.96$; $\rho = 0.9$; $\sigma = 0.07$; $\rho_z = 0.9$; and $\sigma_z = 0.4\left(1-\rho_z^2\right)^{1/2}$. We perform training using the *ADAM* method with the batch size of 10 and the learning rate of 0.001. We fix the number of iterations (which is also a simulation length) to be $K = 100,000$.

5.3 Numerical results

We solve the Krusell and Smith (1998) model with $\ell \in \{1, 5, 10, 50, 100, 300, 1000\}$ agents by minimizing the residuals in the Euler equation. As an illustration, in Figure 11 we show a solution with $\ell = 5$ agents.



The Krusell and Smith (1998) model with five agents.

The upper left panel illustrates the training process. We see that the algorithm reaches the accuracy level of order 10^{-4} in unit-free terms. The upper right panel shows the consumption decision rules for 7 productivity states implied by the Gauss-Hermite quadrature points. The kink is smoother than in the simple consumption savings problem but it is still quite visible. Note that here the axis x shows capital and not wealth, so the kink happens in the negative range (that is, zero wealth means that $R_t k_t^i + W_t z_t^i = 0$ which allows for $k_t^i < 0$). In turn, the lower panels show a simulation of capital series of 5 agents and the corresponding aggregate capital. We see that the individual capital occasionally approaches a zero bound and that the aggregate capital series appear to be stationary as expected.

ℓ	$std(y)$	$corr(y, c)$	$Gini(k)$	*Bottom* 40%	*Top* 20%	*Top* 1%	*Time, sec.*	R^2
1	1.51	0.858	-	-	-	-	235	0.999999
5	1.51	0.772	0.335	0.176	0.385	0.031	234	0.993473
10	1.51	0.595	0.391	0.144	0.428	0.036	254	0.995091
50	1.51	0.635	0.497	0.099	0.530	0.050	467	0.995284
100	1.51	0.658	0.450	0.121	0.486	0.047	1020	0.997600
500	1.51	0.462	0.484	0.096	0.506	0.052	7552	0.996554
1000	1.51	0.268	0.501	0.092	0.528	0.047	16435	0.995415

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

In Table 1, we present some statistics for the model with different number of agents. In the first column, we show that all the studied models have the same standard deviation of output equal to $std(y) = 1.51$. 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 reduces visibly with the number of agents. There is a literature

that tries to understand why the real business cycle models overstate the correlation between these two variables and our analysis suggests that heterogeneity can be a clue.

In columns 3–6, we report the Gini coefficient of the wealth distribution and the share of income that belongs to 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 introduced the aggregate shocks but this is not a sufficiently strong mechanism to change the distributional implications of the baseline Aiyagari's (1994) model.

In column 7, we report the running time. We see that the time varies from 228 to 16,435 seconds which is not prohibitively large. We conclude that much larger models can be solved if we use 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:

$$\ln(k_{t+1}) = \xi_0 + \xi_1 \ln(k_t) + \xi_2 \ln(z_t), \tag{39}$$

i.e., a regression of the current capital on the past 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 0.993 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 they 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. In turn, we have a more complicated setup with a continuum of aggregate states. Our sampling errors in (39) 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 close to theirs. But it is not the main goal of our paper to verify the validity of Krusell and Smith's (1998) approximation. Rather, our goal is to demonstrate that the proposed DL solution method makes it possible to solve large-scale heterogeneous-agent models by using the actual state space without making any simplifying assumptions.

6 Concluding comments: the grain of salt, black magic and no-free-lunch theorem

The economists spent much efforts on designing solution methods for their models – some of these methods are surprisingly smart. But maybe it is the time to move from their model-specific methods to general-purpose AI technologies? Imagine the future: All that the economists need to do is to explain their models to machines. The machines are capable of solving such complex games as chess and Go, so they should be able to solve apparently far less sophisticated economic models. But will the artificial intelligence lead to the same breakthrough in economics as in many other fields?

Here is the grain of salt: it is actually unknown at the moment whether economic models are easy or difficult for AI. In fact, the answer to this question may critically depend on the way in which the problem is framed for numerical treatment. For example, many applications, in which DL recently triumphed, were miscategorized as intractable under earlier algorithmic approaches but proved to be relatively straightforward when the deep learning approach was discovered that mimics the behavior of trained humans. Thus, the economists need to find ways to make their models "easy" for machines.

In the paper, we propose one AI technology that makes the economic models tractable – 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 the 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 an analysis was infeasible up to now. Furthermore, there is a variety of methods from computational economics that can be incorporated in the proposed DL framework leading to further efficiency gains so it seems to be a promising direction to explore.

"Will AI replace computational economists completely?" – this question belongs to a science-fiction domain but our guess is that it is unlikely to happen, at least in the near future. The experience from other fields suggests that there is a little hope that the same combinations of techniques will work equally well for all kinds of models and applications. For example, a given class of approximation functions may work well for some problems but not others; increasing accuracy in certain regions can be obtained by neglecting accuracy in some other regions, etc. Similarly, when it comes to convergence of the training algorithm, a popular theorem among mathematicians states that for any optimization algorithm which converges efficiently for one class of problems, there is another class of problems where it is extremely inefficient. These results must look familiar to economists because they are versions of the famous no-free-lunch theorem.

The same logic applies to AI methods. Despite impressive achievements of the Alphazero player in the Go game, it is actually unknown how close it is to solve the chess game, whereas the Deep Blue chess player does not know how to play Go. So, the idea that there is an all-purpose procedure, which will remove the curse of dimensionality without any trade-offs and will produce a highly precise and uniformly accurate solution for any kind of models belongs to magical thinking. The human input is still critical for designing effective machine-learning methods. Computational economists have a unique chance of applying new technologies in order to vastly extend what has been feasible in economics up to now.

References

[1] Adjemian, S. and M. Juillard, (2013). Stochastic extended path approach. Manuscript.

[2] Ahn, S., G. Kaplan, B. Moll, T. Winberry, C. Wolf (2017). When inequality matters for macro and macro matters for inequality. NBER working paper 23494.

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

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

[5] Azinovic, M., G. Luca and S. Scheidegger (2019). Deep equilibrium nets. SSRN: https://ssrn.com/abstract=3393482

[6] Bayer, C. and R. Luetticke, (2018). Solving heterogeneous agent models in discrete time with many idiosyncratic states by perturbation methods. Manuscript.

[7] Bertsekas, D. and J. Tsitsiklis, (1996). Neuro-dynamic programming. Optimization and Neural computation series. Athena Scientific, Belmont, Massachusetts.

[8] Boppart, T., P. Krusell and K. Mitman, (2018). The performance of policy rules in heterogeneous-agent models with aggregate shocks. Journal of Economic Dynamics and Control 89(C), 68-92.

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

[10] Coleman, C., S. Lyon, L. Maliar and S. Maliar, (2018). Matlab, python, julia: what to choose in economics? CEPR working paper DP 13210.

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

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

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

[14] Duffy, J. and P. McNelis, (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.

[15] Fernández-Villaverde, J., S. Hurtado, and G. Nuño, (2018). Financial frictions and the wealth distribution. Manuscript.

[16] Gaspar, J. and K. L. Judd, (1997). Solving large-scale rational-expectations models. Macroeconomic Dynamics 1, 45-75.

[17] Goodfellow, I., Y. Bengio, and A. Courville, (2016). Deep learning. Massachusetts Institute Technology Press.

[18] Jiang, H., (1996). Smoothed Fischer-Burmeister equation methods for the complementarity problem. Manuscript.

[19] Jirniy, A. and V. Lepetyuk, (2012). A reinforcement learning approach to solving incomplete market models with aggregate uncertainty. Manuscript.

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

[21] Judd, K., (1998). Numerical Methods in Economics. MIT Press, Cambridge, MA.

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

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

[24] Lepetyuk, V., L. Maliar and S. Maliar, (2019). When the U.S. catches a cold, Canada sneezes: a lower-bound tale told by deep learning. CEPR discussion paper DP14025.

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

[26] Maliar, L., and S. Maliar, (2005). Parameterized expectations algorithm: how to solve for labor easily. Computational Economics 25, 269–274.

[27] Maliar, L. and S. Maliar, (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.

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

[29] Maliar, L., Maliar, S., and P. Winant, (2018). Deep learning for solving dynamic economic models. Manuscript. https://lmaliar.ws.gc.cuny.edu/

[30] Maliar, L., Maliar, S., and F. Valli, (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.

[31] McNelis, P. (2005). Neural networks in finance: gaining predictive edge in the market. Elsevier Academic Press.

[32] Powell, W. (2010). Approximate dynamic programming. Wiley, A. John Wiley & Sons.

[33] Boppart, T., P. Krusell and K. Mitman, (2018). The performance of policy rules in heterogeneous-agent models with aggregate shocks. Journal of Economic Dynamics and Control 89(C), 68-92.

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

[35] 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.

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

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

[38] Stokey, N. L. and R. E. Lucas Jr. with E. Prescott, (1989). Recursive methods in economic dynamics. Cambridge, MA: Harvard University Press.

[39] Sutton R. and A. Barto (2018). Reinforcement learning: an introduction. The MIT Press, Cambridge, Massachusetts, London, England.

[40] Villa, A. and V. Valaitis (2019). Machine learning projection methods for macro-finance models, Manuscript.

A Neural networks

In this section, we describe neural networks in details. Neural networks were 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. The first simplified model of a biological neuron – a perceptron – was invented by a psychologist Frank Rosenblatt (1957). In his model, an input $x = (x_0, ..., x_n)$ is linearly weighted by a coefficients vector $\theta = (\theta_0, ..., \theta_n)$ (called weights) to deliver a non-activated output, i.e., $\theta x = \theta_0 x_0 + ... + \theta_n x_n$; $x_0 = 1$ by convention The output is activated by an activation function $\tau(\theta x)$ given by a heaviside step function $\tau(\theta x) = 1_{\theta x \geq 0}$; this is precisely what a neuron's computation is. A modern artificial neuron assumes an arbitrary activation function τ. Such a neuron is represented in Figure 1 of the main text. In the figure, a circle represents a neuron's cell body; a dendrite is the input connection with another neuron, and an axon is the output connection with another neuron.

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: $\tau(x) = \max(a_1 x + b_1, a_2 x + b_2, a_3 x + b_3)$.[9] In Figure A.1, we plot three of such functions, namely, a sigmoid, hyperbolic tangent, and leaky relu.

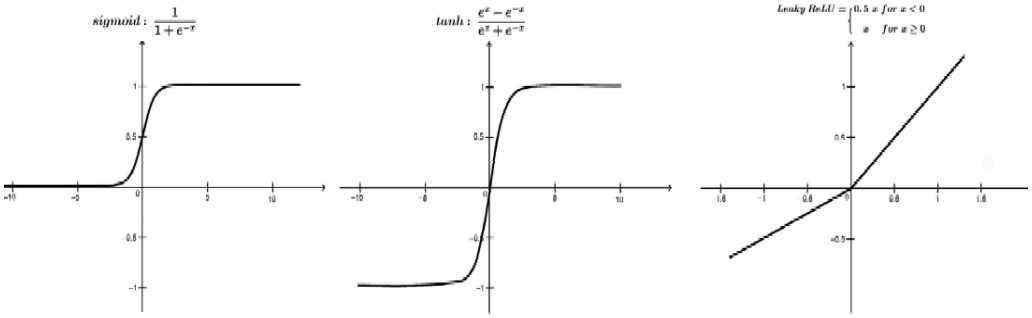


Figure A.1. Common activation functions.

Neural network: general notation. In $\Xi(\theta) = \sum_{i=1}^{n'} (y_i - \varphi_\theta(x_i; \theta))^2$, we are to specify an approximation function $\varphi(\cdot)$ that relates input $\{x_i\}$ and output $\{y_i\}$. Below, we describe how to obtain predicted output nodes $\{\varphi(x_i; \theta)\}$ when one uses neural networks for approximation.

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, called *hidden layers*, between the input and output layers. A 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 linearly combined with a matrix of coefficients $\theta^{(l)} \in \mathbb{R}^{w_{l+1}} \times \mathbb{R}^{w_l + 1}$, i.e., $z^{(l)} = \theta^{(l-1)} a^{(l-1)}$. 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 $\tau_l \in \mathcal{T}$ so that

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

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

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. The predicted output is the result of a hypothesis function $\varphi(x;\theta) = \tau_L\left(z^{(L)}\right) = \tau_L\left(\theta^{(L-1)}a^{(L-1)}\right)$. 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 $\varphi(x;\theta)$.

Neural network: example. As an example, let us consider a simple neural network with three layers ($L = 3$) – an input layer L_1, hidden layer L_2 and output layer L_3. Input features are given by a vector $x = (x_0, x_1, x_2, x_3)$, where $x_0 = 1$ is a bias unit. The parameter vector is $\theta \equiv (\theta^{(1)}, \theta^{(2)})$, where $\theta^{(l)}$ is a matrix made by the weights between the neurons of layer l and the next layer $l+1$; its element $\theta_{ij}^{(l)}$ is a parameter associated with the connection between the neuron j in a layer l and the neuron i in a layer $l+1$, for i, with $j = 0, 1, 2, 3$, and $l = 1, 2$. For our example, $\theta^{(1)} \in \mathbb{R}^4 \times \mathbb{R}^3$ and $\theta^{(2)} \in \mathbb{R}^4 \times \mathbb{R}^1$. Let $z_i^{(l)}$ denote the nonactivated output of a neuron i in the layer l, τ be an activation function (corresponding to computations performed by a neuron), $a_i^{(l)}$ be activated output of neuron i in layer l. Then, for the input layer L_1, we have $a_1^{(1)} = x$. For the second layer L_2, we have $z_1^{(2)} = \theta_{10}^{(1)}x_0 + \theta_{11}^{(1)}x_1 + \theta_{12}^{(1)}x_2 + \theta_{13}^{(1)}x_3$, which is nonactivated output of neuron 1 in layer 2, and $a_1^{(2)} = \tau_1(z_1^{(2)})$, which is activated output of neuron 1 in layer 2; in a matrix form, we have $a^{(2)} = \tau_2(\theta^{(1)}z^{(2)})$. For the output layer L_3, activated output is computed by $a^{(3)} = \tau(\theta^{(2)}z^{(2)}) = \tau(\theta^{(2)}(\tau(\theta^{(1)}x)) = \varphi(x;\theta)$; this is our predicted output; see Figure A.2.

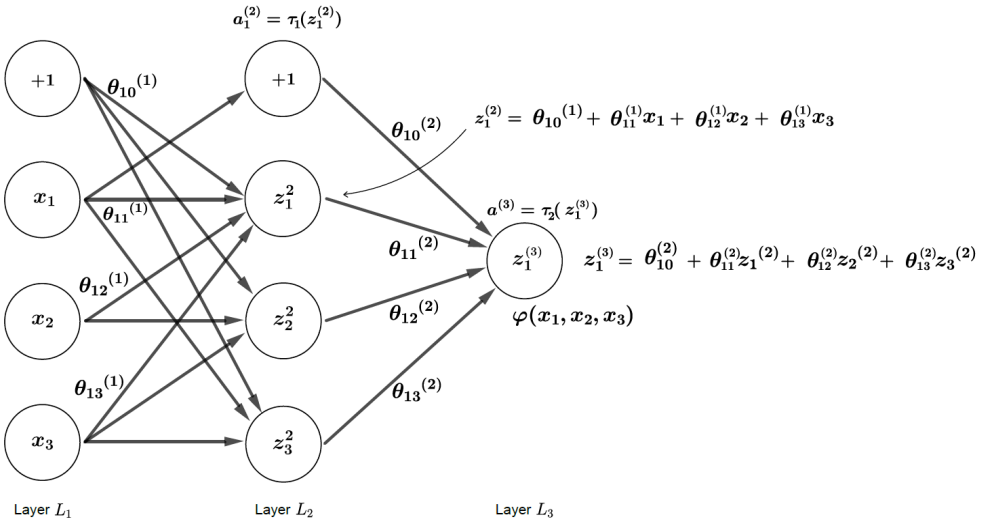


Figure A.2. A neural network with one hidden layer.

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 ML) $\{(x_t, y_t)\}_{t=1}^{n'}$, 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 a 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 a 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 nonconstant, a standard multilayer feedforward network can approximate any (Borel) measurable function arbitrarily well, given that sufficiently many hidden units are considered.[10] 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 activation functions required. Leshno's et al. (1993) results just specify that a sufficiently 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, the failure can occur by 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 used 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) showed 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.

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

B Training methods

In this section,we discuss some training methods that are used in ML literature.

Stochastic and batch gradient descent methods. For very simple approximating functions, the training is easy as a solution to the optimization problem can be constructed in a closed form. The familiar OLS regression is an example in which, $y = \theta x$ and $\theta = (x'x)^{-1} x'y$. For more sophisticated approximating functions, the training is implemented using a numerical iterative procedure. Possible alternatives include Gauss-Jacobi, Gauss-Siedel and gradient descent methods, among others. The basic gradient descent (GD) method is given by

$$\theta_{k+1} \leftarrow \theta_k - \lambda_k \nabla \Xi(\theta). \tag{B.1}$$

Computing the true gradient $\nabla \Xi$ can be costly or infeasible. In ML, optimization methods for finding θ can be classified into two broad categories, namely, stochastic and batch. First, a SGD method approximates the gradient of the expectation function with a single realization of the integrand $\ell(\varphi(\omega))$, i.e., $\nabla \Xi(\theta) \approx \nabla \ell(\varphi(\omega_k, \theta))$. Thus, $\{\theta_k\}$ is not a uniquely determined sequence but a stochastic process that depends on the realized sequence $\{\omega_k\}$. Also, such approximation can be very imprecise on each given step, but the cumulative average converges to the true gradient $\frac{1}{K}\sum_{k=1}^{K} \nabla \ell(\varphi(\omega_k, \theta)) \to \nabla_\theta \Xi(\theta)$ over K updates, provided that the coefficients are stabilized, $\theta_i \approx \theta$. That is, each direction $-\nabla \ell(\varphi(\omega_k, \theta))$ 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. Second, a BGD method in (4) uses multiple random draws on each iteration

$$\theta_{k+1} \leftarrow \theta_k - \frac{\lambda_k}{n'} \sum_{i=1}^{n'} \nabla \ell(\varphi(\omega_i, \theta_k)),$$

where the multiple random draws $(\omega_1, ..., \omega_{n'})$ are called *batches*, $n' \in \{1, ..., n\}$; if $n' = n$, the method is called *a full BGD*, and if $n' < n$, it is called *a mini BGD*.

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. On the other hand, the SGD algorithm does a more efficient use of information about the gradient than a batch one. To understand, create a new sample by copying of 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, its costs per iteration is independent of the sample size n. A mini BGD method still has gains from parallelization but 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 a Nesterov and ADAM methods. There is a trade-off between using lots of parallel simulations with N big 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 a GD 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 and empirical risk. Therefore, to represent both of them, we denote the objective by $F : \mathbb{R}^d \to \mathbb{R}$,

$$F(\theta) = \begin{cases} \Xi(\theta) = E\left[\ell\left(\varphi\left(\omega;\theta\right)\right)\right], \\ \text{or} \\ \Xi_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell\left(\varphi\left(\omega_i;\theta\right)\right). \end{cases} \tag{B.2}$$

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, and if $F(\theta) = \Xi_n(\theta)$, it is done using the probability distribution $P : \mathbb{R}^d \longrightarrow [0, 1]$.

Let us denote by $g(\omega_k;\theta_k)$ a stochastic vector of gradients that covers, respectively, both the basic SGD and a mini-batch GD 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 of the GD algorithms studied.

Algorithm 1. (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* $L > 0$:

$$\left\|\nabla F(\theta) - \nabla F\left(\overline{\theta}\right)\right\|_2 \leq L\left\|\theta - \overline{\theta}\right\|_2 \text{ for all } \left\{\theta, \overline{\theta}\right\} \subset \mathbb{R}^d. \tag{B.3}$$

Note that Assumption 1 is equivalent to the following property:

$$F(\theta) \leq F\left(\overline{\theta}\right) + \nabla F\left(\overline{\theta}\right)'\left(\theta - \overline{\theta}\right) + \frac{1}{2}L\left\|\theta - \overline{\theta}\right\|_2^2 \text{ for all } \left\{\theta, \overline{\theta}\right\} \subset \mathbb{R}^d. \tag{B.4}$$

The equivalence of (B.3) and (B.4) is verified in Bottou et al. (2018). Moreover, this assumption leads to the following useful result for all iterations $k \in \{1, ..., K\}$:

$$E_{\omega_k}\left[F(\theta_{k+1}) - F(\theta_k)\right] \leq -\lambda_k \nabla F(\theta_k)' E_{\omega_k}\left[g(\omega_k;\theta_k)\right] + \frac{1}{2}\lambda_k^2 L E_{\omega_k}\left[\left\|g(\omega_k;\theta_k)\right\|_2^2\right], \tag{B.5}$$

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 to bound the right-hand side of the inequality (B.5). We define the variance of $g(\omega_k;\theta_k)$ as follows:

$$V_{\omega_k}\left[g(\omega_k;\theta_k)\right] \equiv E_{\omega_k}\left[\left\|g(\omega_k;\theta_k)\right\|_2^2\right] - \left\|E_{\omega_k} g(\omega_k;\theta_k)\right\|_2^2. \tag{B.6}$$

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

Assumption 2. (Limits on the first and second moments). *The objective function F in (B.2) 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., *$F(\theta_k) \geq F_{\text{inf}}$ 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

$$\nabla F(\theta_k)' E_{\omega_k}[g(\omega_k; \theta_k)] \geq \mu \|\nabla F(\theta_k)\|_2^2, \tag{B.7}$$

$$\|E_{\omega_k} g(\omega_k; \theta_k)\|_2 \leq \mu_G \|\nabla F(\theta_k)\|_2. \tag{B.8}$$

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

$$V_{\omega_k}[g(\omega_k; \theta_k)] \leq H + H_V \|\nabla F(\theta_k)\|_2^2. \tag{B.9}$$

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

$$\lambda \leq \frac{\mu}{LH_G}. \tag{B.10}$$

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

$$E\left[\sum_{k=1}^{K} \|\nabla F(\theta_k)\|_2^2\right] \leq \frac{K\lambda LH}{\mu} + \frac{2F(\theta_1) - F_{\text{inf}}}{\mu\lambda}, \tag{B.11}$$

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

$$E\left[\frac{1}{K}\sum_{k=1}^{K} \|\nabla F(\theta_k)\|_2^2\right] \leq \frac{\lambda LH}{\mu} + \frac{2[F(\theta_1) - F_{\text{inf}}]}{K\mu\lambda} \tag{B.12}$$

$$\underset{K \longrightarrow \infty}{\longrightarrow} \frac{\lambda LH}{\mu}. \tag{B.13}$$

Proof.

- From (B.5) and (B.7), we have

$$\begin{aligned} &E_{\omega_k}[F(\theta_{k+1}) - F(\theta_k)] \\ &\quad \leq -\lambda_k \nabla F(\theta_k)' E_{\omega_k}[g(\omega_k; \theta_k)] + \frac{1}{2}\lambda_k^2 L E_{\omega_k}\left[\|g(\omega_k; \theta_k)\|_2^2\right] \\ &\quad \leq -\mu\lambda_k \|\nabla F(\theta_k)\|_2^2 + \frac{1}{2}\lambda_k^2 L E_{\omega_k}\left[\|g(\omega_k; \theta_k)\|_2^2\right]. \end{aligned} \tag{B.14}$$

- The definition of the second moment in (B.6), together with Assumption 2 in (B.9), yields

$$E_{\omega_k}\left[\|g(\omega_k; \theta_k)\|_2^2\right] \leq H + \underbrace{[H_V + \mu_G^2]}_{\equiv H_G \geq \mu^2} \cdot \|\nabla F(\theta_k)\|_2^2. \tag{B.15}$$

- Combining the last two equations (B.14) and (B.15) yields

$$E_{\omega_k}[F(\theta_{k+1}) - F(\theta_k)] \leq -\left(\mu - \frac{1}{2}\lambda_k LH_G\right)\lambda_k \|\nabla F(\theta_k)\|_2^2 + \frac{1}{2}\lambda_k^2 LH. \tag{B.16}$$

- Taking the total expectation of (B.16), 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 $F\left(\theta_k\right) \geq F_{\text{inf}}$ for all k, we obtain

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

- Re-grouping the terms in the last expression leads to (B.11). The division of the resulting equation by K implies (B.12). □

According to Assumption 2 about the second moment in (B.9), when $H=0$, there is no noise or noise goes down proportionally to $\left\|\nabla F\left(\theta_k\right)\right\|_2^2$, so that equation (B.13) implies that the sum of squared gradients is finite and that the sequence $\left\{\left\|\nabla F\left(\theta_k\right)\right\|_2\right\} \longrightarrow 0$ as $K \longrightarrow \infty$. When $H>0$, there is an interaction between the learning rate λ and the variance of the stochastic directions. Result (B.12) 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 (B.13), 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

$$\Delta_k \equiv \sum_{k=1}^K \lambda_k=\infty \text{ and } \sum_{k=1}^K \lambda_k^2<\infty. \tag{B.17}$$

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

$$E\left[\sum_{k=1}^K \lambda_k\left\|\nabla F\left(\theta_k\right)\right\|_2^2\right]<\infty, \tag{B.18}$$

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

$$E\left[\frac{1}{\Delta_k} \sum_{k=1}^K \lambda_k\left\|\nabla F\left(\theta_k\right)\right\|_2^2\right] \underset{K \longrightarrow \infty}{\longrightarrow} 0. \tag{B.19}$$

Proof. See Bottou, Curtis and Nocedal (2018).

References

[1] Bertsekas, D., (2015). Convex Optimization Algorithms. Athena Scientific, Nashua, NH, USA.

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

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

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

[5] 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.