CAHIER D'ÉTUDES
WORKING PAPER

N° 184

DEEP LEARNING SOLUTIONS OF DSGE MODELS: A TECHNICAL REPORT

PIERRE BECK, PABLO GARCIA SANCHEZ, ALBAN MOURA, JULIEN PASCAL,
AND OLIVIER PIERRARD

MAY 2024

BANQUE CENTRALE DU LUXEMBOURG
EUROSYSTÈME

DEEP LEARNING SOLUTIONS OF DSGE MODELS: A TECHNICAL REPORT

PIERRE BECK, PABLO GARCIA SANCHEZ, ALBAN MOURA, JULIEN PASCAL, AND OLIVIER PIERRARD

ABSTRACT. This technical report provides an introduction to solving economic models using deep learning techniques. We offer a simple yet rigorous overview of deep learning methods and their applicability to economic modeling. We illustrate these concepts using the benchmark of modern macroeconomic theory: the stochastic growth model. Our results emphasize how various choices related to the design of the deep learning solution affect the accuracy of the results, providing some guidance for potential users of the method. We also provide fully commented computer codes. Overall, our hope is that this report will serve as an accessible, useful entry point to applying deep learning techniques to solve economic models for graduate students and researchers interested in the field.

JEL Codes: C45, C60, C63, E13.

Keywords: solutions of DSGE models; deep learning; artificial neural networks.

March 2024. Pierre Beck: École Polytechnique Fédérale de Lausanne, Emergent Complexity in Physical Systems Laboratory, CH-1015 Lausanne, Switzerland (contact: pierre.beck@epfl.ch). Pablo Garcia Sanchez, Alban Moura, Julien Pascal, Olivier Pierrard: Banque centrale du Luxembourg, Département Économie et Recherche, 2 boulevard Royal, L-2983 Luxembourg, Luxembourg (contact: pablo.garciasanchez@bcl.lu, alban.moura@bcl.lu, julien.pascal@bcl.lu, olivier.pierrard@bcl.lu). This project was started in 2022, when Pierre Beck was in the BCL Graduate Program. For useful comments and suggestions, we thank Paolo Guarda and BCL colleagues. This paper should not be reported as representing the views of the BCL or the Eurosystem. The views expressed are those of the authors and may not be shared by other research staff or policymakers in the BCL or the Eurosystem.

Résumé Non Technique

Les banques centrales ont rapidement adopté les méthodes basées sur l'intelligence artificielle pour mener aussi bien des analyses statistiques que des tâches de supervision (Banque des règlements internationaux, 2024). Récemment, il a été établi que les techniques du deep learning (DL, apprentissage profond), inspirées par la structure des réseaux neuronaux, permettent de résoudre les modèles économiques à agents hétérogènes avec risque agrégé, les modèles de cycle de vie et les modèles multi-pays souvent utilisés par les banques centrales.

Toutes les techniques de DL reposent sur deux idées simples mais puissantes. Premièrement, il est possible d'approximer des fonctions très complexes par une combinaison de nombreuses fonctions simples. Deuxièmement, la qualité de cette approximation peut être évaluée et améliorée par des méthodes de simulation Monte Carlo. Ces deux idées permettent aux techniques de DL de s'attaquer à des problèmes très difficiles, de la reconnaissance visuelle à la compréhension de la parole.

Dans le domaine de la résolution de modèles économiques, deux obstacles principaux limitent le recours aux techniques de DL. Le premier est la terminologie souvent abstraite et parfois confuse associée aux techniques de DL, qui crée une aura intimidante pour les non-initiés. Le second obstacle est la difficulté liée aux multiples décisions de spécification à prendre avant de pouvoir appliquer les techniques de DL.

Dans ce rapport, notre objectif est de contribuer à surmonter ces deux obstacles. Nous commençons par fournir un aperçu des techniques de DL et de leur applicabilité à la résolution de modèles économiques. Nous soulignons notamment les différentes décisions que l'utilisateur de la méthode doit prendre et donnons quelques indications susceptibles de guider son choix. Ensuite, nous considérons une application de base : la résolution du modèle de croissance stochastique, l'environnement de référence de la théorie macroéconomique moderne. Nous montrons comment chaque choix de l'utilisateur affecte la qualité de la solution DL, ce qui nous permet d'illustrer certains principes généraux susceptibles d'aider à choisir la meilleure configuration.

1. Introduction

Central banks have been early adopters of artificial intelligence (AI) and deep learning (DL) for bank supervision, payment systems oversight, and statistical analysis (Bank for International Settlements, 2024). Recent advances in computational economics show that DL techniques are useful to tackle high-dimensional problems in economics, including heterogeneous-agent models with aggregate risk (Maliar, Maliar, and Winant, 2021), large-scale central-bank models (Lepetyuk, Maliar, and Maliar, 2020), life-cycle models (Azinovic, Gaegauf, and Scheidegger, 2022), and multi-country models with migration linkages (Fernandez-Villaverde, Nuno, Sorg-Langhans, and Vogler, 2020).

All DL techniques rely on two simple but powerful ideas. First, a composition of many simple functions can provide a good approximation to even very complex mappings. Second, Monte Carlo methods can be used to assess the quality of the approximation and to improve it. Together, these two ideas make it possible for DL techniques to tackle very difficult problems, from pattern recognition in images to speech understanding. Within the economic literature, they also make DL techniques a special case of projection methods, similar for instance to the grid-free parametrized-expectation algorithm proposed by Christiano and Fisher (2000).[1] However, in contrast to traditional projection methods, DL algorithms are not subject to the *curse of dimensionality*, which occurs whenever the computational burden of solving a problem increases exponentially as the dimension of the problem grows. This advantage makes DL methods particularly appealing when dealing with large models.

DL overcomes the curse of dimensionality by harnessing the power of *neural networks*. Inspired by the structure of the human brain, neural networks are computational models made of interconnected nodes, known as neurons, organized in successive layers. Each neuron receives input data, processes it, and passes the result to the next layer. Through a process called training, neural networks learn from examples to perform various tasks. Once well trained, neural networks are powerful tools to discover and mimic complex patterns. In particular, they can learn how to solve economic models by approximating the decision rules of the various agents populating the economy.

In our view, there are two main obstacles to the development of DL solution methods in economics. The first is the often abstract and sometimes confusing terminology associated with DL techniques, which creates an intimidating aura for outsiders. The second is the difficulty involved in various decisions that a researcher must make to apply DL techniques to solve an economic model: How should the model be written? How should the neural network be designed? How should it be trained? These choices all affect the performance of the method and the quality of the solution it yields, but there is limited available guidance. In fact, finding a good configuration requires both expertise and some trial-and-error process,

[1]See Fernández-Villaverde, Rubio-Ramírez, and Schorfheide (2016) for a detailed review of projection methods.

but most published studies tend to understate these difficulties by focusing on successfully trained networks.

In this report, our aim is to contribute to relaxing these two obstacles. In Section 2, we provide a relatively self-contained overview of DL techniques and of their applicability to solving economic models. Our objective is to provide an accessible yet rigorous presentation that demystifies some of the aura of the approach and allows the reader to grasp its key features. We emphasize the various choices an user of the method must make and summarize the existing guidance. Then, in Section 3, we consider an application to solving the canonical stochastic growth model, the benchmark environment of modern macroeconomic theory. To highlight the importance of the choices related to the design of the neural network and to the training process, we study a variety of possible setups and characterize how each choice affects the final performance of the approximation method. This allows us to draw some conclusions to guide the fine-tuning process necessarily required by DL methods. Throughout, our main objective is pedagogical. Our main hope is that interested graduate students and researchers find our presentation a beneficial entry point to DL techniques. We also make our code freely available at `https://notes.quantecon.org/submission/65449f6b8f6a1a0016fc4544`.

Two additional observations are in order. First, the literature on DL in general as well as on its applications in economics is enormous. While we point to a number of relevant references, our presentation mostly builds upon Goodfellow, Bengio, and Courville (2016) for DL in general, and on Maliar, Maliar, and Winant (2021) for the economic application. Second, the application of DL techniques to solving the stochastic growth model is deliberately simple. The framework is extremely well known and should be familiar to most economists, and the step-by-step nature of numerical methods, in which understanding elementary topics lays the groundwork for tackling more complex problems, makes it an attractive reference. Thus, even though we do not claim to provide universal principles, our insights likely hold for most DL applications to economics.

2. A Deep Learning Approach to Solve Economic Models

This section provides a non-technical overview of the DL approach proposed by Maliar, Maliar, and Winant (2021) to solve economic models. To make the discussion accessible to economists without background in AI and relatively self-contained, we also review the most important DL concepts. Goodfellow, Bengio, and Courville (2016) provide a thorough textbook treatment of DL theory and applications.

2.1. **The approximation problem.**

A generic AI problem involves approximating an unknown function $F^\star : \mathbb{R}^m \to \mathbb{R}^n$ using the information provided by a set of potentially noisy observations on inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y} = F^\star(\boldsymbol{x})$. With appropriate choices of $F^\star$, this setup encompasses both simple statistical problems (estimating conditional moments or full conditional distributions) and more complex AI tasks (classification, detection, image or speech recognition, translation, ...).

A strategy to solve the approximation problem is as follows. Let $\mathcal{S} = \{F(\cdot;\boldsymbol{\theta}) : \theta \in \mathbb{R}^p\}$ denote a set of known functions indexed by the vector of parameters $\boldsymbol{\theta}$ and $L : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ denote a loss function penalizing approximation errors, so that $L\left[F(\boldsymbol{x}, \boldsymbol{\theta}),\, \boldsymbol{y}\right]$ measures the cost associated with a discrepancy between the prediction $F(\boldsymbol{x}, \boldsymbol{\theta})$ and the actual output $\boldsymbol{y}$. Letting $P : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^+$ denote the joint cumulative distribution function of $(\boldsymbol{x}, \boldsymbol{y})$, solving the approximation problem over $\mathcal{S}$ is equivalent to finding the parameter vector associated with the smallest expected approximation error:

$$\boldsymbol{\theta}^\star = \operatorname{argmin}_{\boldsymbol{\theta}}\ \Xi(\boldsymbol{\theta}), \tag{1}$$

where the expected approximation error

$$\Xi(\boldsymbol{\theta}) := \int L\left[F(\boldsymbol{x}, \boldsymbol{\theta}),\, \boldsymbol{y}\right] dP(\boldsymbol{x}, \boldsymbol{y}) = \ E\left\{L\left[F(\boldsymbol{x}, \boldsymbol{\theta}),\, \boldsymbol{y}\right]\right\} \tag{2}$$

is also called *expected cost*, *expected loss*, or *expected risk*.

Clearly, this strategy will only provide an interesting solution to the approximation problem under specific conditions. First, the candidate set $\mathcal{S}$ must be 'flexible' enough to contain functions close to $F^\star$. If this is not the case, even the loss-minimizing function will be a poor approximation to $F^\star$, a phenomenon called *underfitting*. Second, solving the minimization problem over $\boldsymbol{\theta}$ must be feasible in practice. This requires being able to replace the unknown objective function $\Xi(\boldsymbol{\theta})$ by a valid approximation and being able to perform the actual minimization.

The crux of DL methods is to provide a framework satisfying all these conditions to make the approximation strategy operational. In particular, DL offers: (i) a way to replace the unknown expected risk by an easy-to-measure empirical counterpart based on observed data; (ii) a set of candidate functions with attractive properties, known as neural networks; (iii) efficient numerical methods to find parameter values associated with small approximation errors, namely variations of gradient descent; (iv) validation techniques that ensure the robustness of the results. We present these elements in turn, before discussing their application to solving economic models.

2.2. **The objective function.** Minimizing the expected loss $\Xi(\boldsymbol{\theta})$ is not feasible in practice because the joint distribution of inputs and outputs is unknown. A key motivation of DL is that data availability solves this issue: given enough observations on input-output pairs, it is possible to invoke standard laws of large numbers to build an accurate estimate of the objective function.

In particular, let $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$ denote a sample of n independent observations on inputs and outputs. Then, the *empirical cost*

$$\widehat{\Xi}_n(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L\left[F(\boldsymbol{x}_i, \boldsymbol{\theta}),\, \boldsymbol{y}_i\right] \tag{3}$$

provides an unbiased and consistent estimate of $\Xi(\boldsymbol{\theta})$. Given a loss function L, a candidate function F, and a parameter vector $\boldsymbol{\theta}$, it is easy to compute $\widehat{\Xi}_n(\boldsymbol{\theta})$, which is sometimes referred to as the *empirical loss* or the *empirical risk*. In practice DL algorithms approximate $F^\star$ by minimizing $\widehat{\Xi}_n(\boldsymbol{\theta})$ rather than $\Xi(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$.

Because the loss function L quantifies the discrepancy between model-implied outputs and actual observations, its choice is application-dependent. For instance, a continuous loss function might work well when the output is a real number, but would be less adapted to a classification problem with binary output. As discussed below, Maliar, Maliar, and Winant (2021) cast economic models in DL form using a *regression representation* in which F outputs a real number. A natural loss function in this case is the *mean squared error* (MSE):

$$L_{MSE}(y_1,\, y_2) = (y_1 - y_2)^2.$$

2.3. **Artificial neural networks.** Artificial neural networks are computational models designed to transform input data into output data in potentially complex ways. More simply, they are software representations of functions mapping an input space into an output space. Modern DL techniques use neural networks to solve various approximation problems. As explained in Goodfellow, Bengio, and Courville (2016), the increasing ability to develop, optimize, and use neural networks is the key factor behind recent progress in AI.

2.3.1. *Neurons, layers, and networks.* The basic units of neural networks are called *neurons*, *nodes*, *perceptrons*, or *units*. A neuron is a simple function $g : \mathbb{R}^k \to \mathbb{R}$ that takes as input a vector $\boldsymbol{z} \in \mathbb{R}^k$, transforms it using an affine mapping parametrized by a vector of *weights* $\boldsymbol{\gamma} \in \mathbb{R}^{k+1}$, and finally applies a non-linear function f to output a real number. Formally,

$$g(\boldsymbol{z}; \boldsymbol{\gamma}) = f\left(\gamma_0 + \sum_{i=1}^{k} \gamma_i z_i\right).$$

The function $f : \mathbb{R} \to \mathbb{R}$ is called the *activation function* of the neuron and shapes the properties of its output. The intercept γ_0 is sometimes called the *bias* of the affine transform.

Stacking p neurons in a vector forms a vector-valued mapping from $\mathbb{R}^k$ to $\mathbb{R}^p$, defined by

$$G(\boldsymbol{z}; \boldsymbol{\Gamma}) = \begin{bmatrix} g(\boldsymbol{z}; \boldsymbol{\gamma}_1) \\ \dots \\ g(\boldsymbol{z}; \boldsymbol{\gamma}_p) \end{bmatrix},$$

where $\boldsymbol{\Gamma} = (\boldsymbol{\gamma}_1, \dots, \boldsymbol{\gamma}_p)$ contains the parameters of the affine transforms associated with the p nodes. Such vector functions stacking neurons are called *layers*, and the number of neurons in a specific layer is called its *width*. Our presentation assumes that all neurons in a layer share the same input $\boldsymbol{z}$ and the same activation function f for clarity only; more sophisticated structures are possible.

Finally, a *neural network*, also known as a *neural net* or a *multilayer perceptron*, is a composition of several layers. For instance, letting $G_1, \ldots, G_l$ denote l layers of conformable dimensions, and keeping affine parameters implicit, a neural network mapping $\mathbb{R}^m$ into $\mathbb{R}^n$ can be defined as

$$N(\boldsymbol{x}) = G_l \left\{ \ldots G_2[G_1(\boldsymbol{x})] \ldots \right\},$$

where $\boldsymbol{x} \in \mathbb{R}^m$, G_1 admits an m-dimensional vector as input, and G_l outputs an n-dimensional vector. To introduce further vocabulary, the first and last layers in a network are called respectively the *input layer* (G_1 in this example) and the *output layer* (G_l), while the additional layers located between them are called the *hidden layers*. The number of layers l in a neural network is called its *depth*, while its width is that of the widest layer.

This very specific structure explains most of the terminology related to neural network:

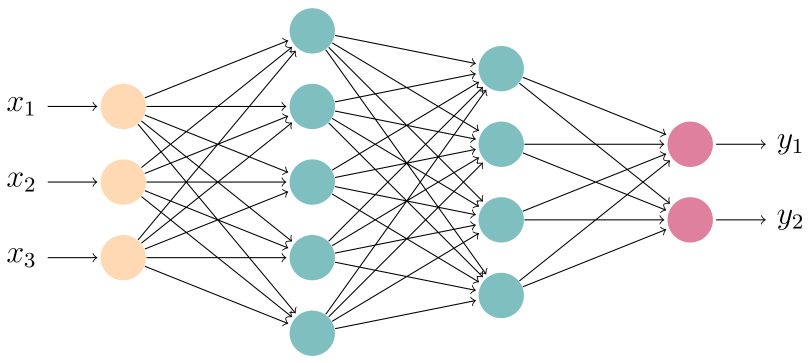
- From a mathematical perspective, the term *network* refers to the composition process, which can be represented visually by a graph describing how the various functions act together to transform inputs into outputs. Figure 1 provides such a visual representation for a network mapping $\boldsymbol{x} \in \mathbb{R}^3$ to $\boldsymbol{y} \in \mathbb{R}^2$ by composing four layers. Note that the number of neurons in the input and output layers are constrained by the dimensions of the variables of interest, while the widths of the two hidden layers are not.
- From a biological perspective, the term *neural* refers to the analogy with the biological neural networks present in animal and human brains. These structures also consist of several layers of connected units called neurons, which all receive a signal as input, transform it, and then process it to other units in different layers.

Importantly, not all neural networks qualify as DL tools. The adjective *deep* refers to the depth of the network, i.e. to the number of layers. Equivalently, the concept refers to the number of successive function compositions defining the network. A network consisting of more than three layers qualifies as a DL algorithm, with less sophisticated networks referred to as shallow. The main idea of DL is that deeper networks are more flexible and have the potential to learn more complex functions.

2.3.2. *Approximation properties.* Neural networks have good approximation properties. A variety of *universal approximation theorems* indicate that a neural network can approximate any continuous function from one finite-dimensional space to another with any desired non-zero amount of error, provided that it is given sufficient depth or sufficient width.[2] In

[2]See, e.g., Cybenko (1989), Hornik, Stinchcombe, and White (1989), and Leshno, Lin, Pinkus, and Schocken (1993) for formal derivations.

FIGURE 1. Architecture of a neural network



Notes. This neural network takes $x \in \mathbb{R}^3$ as input and produces $y \in \mathbb{R}^2$ as output. It features four different layers: the orange nodes correspond to the input layer, the green nodes to two hidden layers, and the pink nodes to the output layer. The arrows represent the flows of information through the network, as the original input is successively transformed by the different layers of neurons.

addition, for a given approximation error, the number of parameters required to approximate a function with a neural network increases linearly with the dimension of the input $\boldsymbol{x}$, while it increases exponentially for most other classes of approximators; hence, neural networks are less subject to the well-known *curse of dimensionality* (Barron, 1994; Fernández-Villaverde, Hurtado, and Nuño, 2019). Finally, neural networks are robust to multi-collinearity and perform well even when approximating functions with various kinks and discontinuities. These attractive features explain the success of deep neural networks in a variety of real-world applications.

Unfortunately, the approximation theorems are not constructive: they imply that a neural network *can* represent a set of interesting functions, but do not specify how to identify the network structure associated with the best fit. In addition, there is no guarantee that the network will have the ability to actually *learn* the function in practice, i.e. that identifying the loss-minimizing value of $\boldsymbol{\theta}$ is feasible given the amount of available data.

2.3.3. *Design issues.* It follows from the previous discussion that the ideal network architecture for a specific task must be found by experimentation and validation. From a practical perspective, this experimentation involves a number of design decisions related to (i) the activation function(s) in each layer, (ii) the width of each layer, and (iii) the depth of the network. Variations in the way neurons in one layer connect to the next layer have been explored, but we will not discuss them; instead, we focus on the *fully connected* case shown in Figure 1, in which each neuron connects to all neurons in the next layer. Together, these decisions define the *architecture* of the network and influence its properties.

Activation functions. As described above, without activation function the output of a neuron would be an affine transform of the input vector. The role of activation functions is to introduce nonlinearity in the algorithm, which is crucial to the good performance of neural networks in complex settings. For instance, all universal approximation theorems require the presence of nonlinear activation functions, although the specific type of nonlinearity varies from one result to the other.

Historically, the earliest ML algorithms were asked to decide whether or not an input, represented by a vector of characteristics, belongs to some specific class. The output of a neuron was a binary function, returning one if the affine transform $\gamma_0 + \sum_i \gamma_i z_i$ was above some threshold and zero otherwise. In today's DL vocabulary, this setting corresponds to using a step function as activation function. With the development of larger models, the poor gradient properties of the step function became problematic: its derivative is zero everywhere, except at the discontinuity where it explodes. As a result, smooth approximations to the step function became the norm for activation functions, for instance the logistic sigmoid and the hyperbolic tangent.[3] These functions are continuous and differentiable, thus allowing gradient-based optimization. However, they *saturate* quickly, which means that their gradient goes to zero quickly away from the origin, which can stall out gradient-based learning. This is also known as the *vanishing gradient problem* (Hochreiter, 1998).

As discussed in Goodfellow, Bengio, and Courville (2016, Chapter 6), recent experience with large DL models has highlighted two important points. First, so-called *rectified linear units* (ReLU) activation functions, $f(z) = \max(0, z)$, are excellent default choices for neurons located in the input and hidden layers of deep neural neural networks. From a numerical perspective, ReLU functions benefit from sparse activation (in a randomly initialized network, only about 50% of hidden nodes return a non-zero result) and efficient computation (they require only multiplication, addition, and comparison). Their gradient properties are also good and can be improved further through simple generalization, like the leaky ReLU function $f(z) = \max(0, z) + 0.01 \min(0, z)$ which has non-zero gradient almost everywhere. Second, in actual applications, many different activation functions yield comparable performance. As a result, a reasonable benchmark is to use ReLU activation functions for neurons located in the input and hidden layers.

Choosing the activation function(s) for the output layer requires special attention because they define the overall output of the network. In particular, it is desirable to adapt the choice of the activation function(s) to the properties of the desired output. For instance, a linear function $f(z) = z$ works well when the output is unconstrained. On the other hand, an exponential function $f(z) = \exp(z)$ is adapted to settings in which output is

[3]The logistic sigmoid function is $\sigma(z) = (1 + e^{-z})^{-1}$; it is symmetric around 0 and outputs a value in the range $[0, 1]$. The hyperbolic tangent function is $\tanh(z) = (e^z - e^{-z})/(e^z + e^{-z}) = 2\sigma(2z) - 1$; it is also symmetric around $z = 0$ and outputs a value in the range $[-1, 1]$.

non-negative. Alternatively, when the network represents a probability, sigmoid or softmax functions implement the correct restrictions.[4]

Depth and width. Choosing the depth (number of layers) and width (number of neurons per layer) of a neural network is the most difficult design decision because the best choice is largely application dependent. Two rules of thumb may help researchers. First, deeper networks typically require far fewer neurons per layer, and therefore far fewer parameters, and they tend to generalize better (Goodfellow, Bengio, and Courville, 2016, Chapter 6). Therefore, in general deeper networks are preferable to wider networks. Second, the complexity of the neural network should be adapted to the complexity of the underlying approximation problem, related for instance to the dimensions of the target function $F^\star$ or to its nonlinearity. As shown below, typical economic models are simple enough not to warrant using overly complex networks.

Finally, we note that DL methods propose strategies to fine-tune the architecture of the network *after* the optimization step. We discuss this *validation* step below.

2.4. **Training the network with gradient descent.** With a candidate architecture in place, the next task is to optimize the performance of the network with respect to the vector of parameters. Using the notation introduced in Section 2.1, $\mathcal{S}$ denotes the set of neural networks with the chosen architecture, $\boldsymbol{\theta} = \boldsymbol{\Gamma}$ contains all neuron parameters, and the goal is to find the value of $\boldsymbol{\theta}$ associated with the smallest empirical loss. In the DL context, this optimization step is called *learning* or *training* because it improves the network performance using the information contained in observed data.

In most DL applications, minimization of $\widehat{\Xi}_n$ with respect to $\boldsymbol{\theta}$ is performed using variations of *gradient descent*, a standard algorithm for finding a local minimum of a differentiable function by taking repeated small steps in the direction opposed to the current gradient of the function.[5] The method rests on a basic result in linear algebra: the gradient of a differentiable function $G : \mathbb{R}^m \to \mathbb{R}$ indicates the local direction of steepest ascent, so that $G(\boldsymbol{x} + \lambda \boldsymbol{v}) - G(\boldsymbol{x})$ is maximized in the neighborhood of $\boldsymbol{x} \in \mathbb{R}^m$, i.e. for $\lambda > 0$ small, when $\boldsymbol{v} = \nabla G(\boldsymbol{x})$ is the gradient of G evaluated at $\boldsymbol{x}$. Conversely, $G(\boldsymbol{x} + \lambda \boldsymbol{v}) - G(\boldsymbol{x})$ is locally minimized by moving in the direction opposite to the gradient (steepest descent).

[4]The softmax function is an unusual activation function in that it applies to the full vector of inputs, rather than entry-by-entry: if $\boldsymbol{z} \in \mathbb{R}^k$, then $\text{softmax}(\boldsymbol{z}) = [\mu(z_1), \mu(z_2), \ldots, \mu(z_k)]'$, with $\mu(z_i) = e^{z_i}/(\sum_j e^{z_j})$. Softmax$(\boldsymbol{z})$ outputs a vector whose entries sum to one, which can be interpreted as a probability distribution.

[5]Applying convex optimization techniques like gradient descent to non-convex objective functions like neural networks may seem suspicious. However, recent experience shows that DL models work very well when trained with gradient descent (Goodfellow, Bengio, and Courville, 2016, Chapter 6). One explanation is that gradient descent is typically able to find a very low value of the objective function in a reasonable amount of time, which is enough for the trained model to perform well.

It follows from equation (3) that the gradient of the empirical loss is

$$\nabla \widehat{\Xi}_n(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L \left[F(\boldsymbol{x}_i, \boldsymbol{\theta}),\, \boldsymbol{y}_i \right]. \tag{4}$$

Therefore, each evaluation of the gradient requires a full pass through the dataset. Algorithms implementing such complete passes are referred to as *batch learning*, because they consider a full 'batch' of data before updating the parameter vector $\boldsymbol{\theta}$. However, such passes become prohibitively expensive when the number of available observations increases, so that basic gradient descent needs some modifications to perform well in DL applications powered by very large datasets.

A key insight is that the gradient defined by equation (4) is a sample mean. Therefore, under the assumption that the dataset consists of independent and identically distributed (i.i.d.) points, basic statistical theory implies that averages computed from much smaller subsamples provide unbiased estimates of the true gradient. These smaller subsamples are called *mini-batches* and the corresponding algorithm implements *mini-batch learning*. See Algorithm 1 for a formal description.

Algorithm 1: Stochastic gradient descent (Goodfellow, Bengio, and Courville, 2016)

```
input : initial parameter vector $\boldsymbol{\theta_0}$
input : learning rate $\lambda > 0$
input : stopping criterion $\mathcal{C}$
output: loss-minimizing parameter vector $\boldsymbol{\theta}^\star$
while stopping criterion $\mathcal{C}$ not met do
    sample a mini-batch of $n'$ input-output pairs uniformly from the training dataset,
      with $n' << n$ ;
    estimate the gradient of the objective function as
        $$\boldsymbol{g} = \frac{1}{n'} \sum_{j=1}^{n'} \nabla_{\boldsymbol{\theta}} L \left[ F(\boldsymbol{x}_j, \boldsymbol{\theta}),\, \boldsymbol{y}_j \right]$$
      using observations from the mini-batch ;
    update the vector of parameters
        $$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \boldsymbol{g}$$
      using gradient descent ;
    check if $\mathcal{C}$ holds true ;
end
```

Goodfellow, Bengio, and Courville (2016) refer to mini-batch learning as an example of *stochastic gradient descent*, where the adjective 'stochastic' emphasizes that a random estimate of the gradient is used in place of the true sample gradient to update the parameters of the model. For other authors, e.g. LeCun, Bottou, Orr, and Müller (2012), stochastic gradient descent corresponds to the more specific case of a mini-batch of a single observation. In any case, stochastic gradient descent has become the method of choice to train large neural networks: it is much faster than batch learning; it does not discard much information; and it also often results in better solutions.[6]

From a practical perspective, batch and stochastic gradient descent methods need three elements: (i) the ability to compute the gradient of the objective function, (ii) a way to initialize the vector of parameters, and (iii) a value for the *learning rate* λ. Most DL algorithms compute the gradient using a method called *back-propagation*. The name comes from a simple analogy: just as propagating the input $\boldsymbol{x}$ forward through the network allows to compute the output $\boldsymbol{y}$, propagating information from the cost function backward through the network and applying the chain rule of differentiation provides an efficient way to compute and evaluate the relevant gradient function $\nabla_{\boldsymbol{\theta}} L$. Back-propagation is automatically implemented in most DL libraries, so we do not discuss it here.

A standard way to initialize the parameter vector is to choose small independent values for the elements of $\boldsymbol{\theta}$, for instance by drawing from Gaussian or uniform distributions. More sophisticated initialization techniques exist: for instance, our implementation below runs on PyTorch, a Python machine-learning library that initializes weights according to a methodology proposed by He, Zhang, Ren, and Sun (2015).

Finally, the choice of the learning rate may be the most important decision related to gradient descent. Although we denote λ as a constant, in practice it is preferable to *gradually decrease* the value of the learning rate over time. The idea is that the noise induced in successive gradient estimates by random mini-batch sampling does not vanish at the minimum of the objective function and needs to be dampened by a smaller learning rate.[7] In addition, fine-tuning the value of the learning rate improves the properties of gradient descent: too large a rate might create violent oscillations in parameter values because of shifts in the gradient slope, while too low a rate might result in excessively slow convergence to the optimum. Possible strategies are to decrease linearly or exponentially the learning rate over successive iterations, but the best choices for the initial learning rate and the speed of the dampening process are extremely application dependent: as noted by Goodfellow, Bengio, and Courville (2016, Chapter 8), "[t]his is more of an art than a science, and most guidance

[6]Stochastic learning does not discard much information because observations are often redundant in large datasets. The ability to reach better solution arises from the noise induced by mini-batch sampling, which might push the parameter vector away from a local minimum and to the neighborhood of a deeper minimum. In most applications, the mini-batches are sampled without replacement from the training dataset to ensure that successive gradient estimates are independent. Performing a complete pass through the dataset using mini-batches is sometimes called an *epoch* in the DL literature.

[7]Goodfellow, Bengio, and Courville (2016, Chapter 8) provide sufficient conditions on the learning rate that guarantee convergence of stochastic gradient descent.

on this subject should be regarded with some skepticism.” Richer learning schemes are also possible, for instance by using an average of present and past gradient estimates at each iteration to accelerate convergence (a practice known as *momentum*) or by using a different learning rate for each weight coefficient.

2.5. **Validation and regularization.** The overall objective of DL techniques is to provide algorithms that perform well on *new* inputs: indeed, training a model on a given dataset is useless if its performance does not generalize well to the yet unknown inputs that will arise in actual applications.

As often in statistics, the implicit assumption is that the training sample is representative of the data generating process, so that a low empirical loss ensures good performance on new observations. One obvious condition for such a good performance is that the model does not *underfit*, i.e. that it is able to identify and learn the important properties of the data from the training sample. This is easy to check by gauging the empirical performance on the training sample. Another important condition, however, is that the model does not *overfit*, i.e. that it has not learned various idiosyncrasies of the training dataset that will not generalize well to new inputs.

From a broader perspective, these two conditions imply that the model should have the right *capacity* for the task it faces, i.e. that it is complex enough to represent the data, but not overly complex. Since the architecture of a DL model largely determines its capacity, this is equivalent to saying that the model should have the right structure, e.g. in terms of depth and width for neural networks. Checking that this is indeed the case is called *validation*.

A simple way to perform model validation is to split the available observations into three disjoint datasets: a *training* set, a *validation* set, and a *test* set. Typically, the split attributes 70-80% of the data to the training set and 10-15% to both the validation and test sets. As the names indicate, learning the parameter values that minimize the empirical loss takes place only on the training set, the final tuning of the model hyperparameters occurs on the validation set, and the final performance of the model is estimated from the test set. A large generalization error, as measured by the empirical loss on the test set, signals an overfit problem. In this case, the capacity of the model needs to be reduced, for instance by reducing the width of a network or removing a layer. Iterating back and forth between these steps is the best way to identify the best architecture for a given DL problem.

Another possibility to reduce overfitting issues is to automatically penalize model complexity during the training step. This process, called *regularization*, aims at favoring parsimonious representations of the data, while allowing enough generality to avoid underfitting. It has a long history in statistics: standard model selection tools such as the Akaike and Bayesian information criteria and ridge regression are classical examples of regularization for linear models. Goodfellow, Bengio, and Courville (2016, Chapter 7) discuss applications of this idea to DL problems.

2.6. **Solving economic models using DL.** Finally, we present the method proposed by Maliar, Maliar, and Winant (2021) to solve dynamic economic models with DL. Their approach involves two key steps: (i) transforming the problem of solving the economic model into a simpler minimization problem and (ii) applying DL tools to solve this minimization problem. Maliar, Maliar, and Winant discuss three transformations of economic models. For simplicity, we focus on the transformation based on first-order optimality conditions. The other two transformations, based on lifetime utility and Bellman equations, are fairly similar; see Maliar, Maliar, and Winant (2021) for details.

2.6.1. *From the economic model to the minimization program.* The general equilibrium of a wide class of dynamic models used in macroeconomics admits the following generic representation. An exogenous state vector $\boldsymbol{z}_t \in \mathbb{R}^{n_z}$ evolves according to the Markov process

$$\boldsymbol{z}_t = Z(\boldsymbol{z}_{t-1}, \boldsymbol{\epsilon}_t), \tag{5}$$

where $\boldsymbol{\epsilon}_t \in \mathbb{R}^{n_\epsilon}$ is an i.i.d. innovation and $Z : \mathbb{R}^{n_z} \times \mathbb{R}^{n_\epsilon} \to \mathbb{R}^{n_z}$ is a known function. An endogenous state vector $\boldsymbol{s}_t \in \mathbb{R}^{n_s}$ is driven by the exogenous process and by an endogenous control vector $\boldsymbol{c}_t \in \mathbb{R}^{n_c}$ according to a known function $S : \mathbb{R}^{n_z} \times \mathbb{R}^{n_s} \times \mathbb{R}^{n_c} \to \mathbb{R}^{n_s}$:

$$\boldsymbol{s}_{t+1} = S(\boldsymbol{z}_t, \boldsymbol{s}_t, \boldsymbol{c}_t). \tag{6}$$

(Notice that the value of s_{t+1} is realized at date t. This notation, which follows Maliar, Maliar, and Winant, is different from the usual timing convention in macroeconomics, according to which the date of a variable indicates the period at which it is realized.) At each period, the various elements of the control vector solve some constrained optimization problem(s). Consequently, they verify n_c first-order/equilibrium conditions of the form

$$E_t f_j(\boldsymbol{z}_t, \boldsymbol{s}_t, \boldsymbol{c}_t, \boldsymbol{z}_{t+1}, \boldsymbol{s}_{t+1}, \boldsymbol{c}_{t+1}) = 0, \qquad j = 1, \dots, n_c, \tag{7}$$

where $(\boldsymbol{z}_t, \boldsymbol{s}_t)$ is fixed, E_t denotes the expectation operator conditional on date-t information, and the $f_j : \mathbb{R}^{n_z} \times \mathbb{R}^{n_s} \times \mathbb{R}^{n_c} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_s} \times \mathbb{R}^{n_c} \to \mathbb{R}$, $j = 1, \dots, n_c$, are known functions describing the economic forces at play in the model.[8] Since the only source of uncertainty in equation (7) is $\boldsymbol{\epsilon}_{t+1}$, which is implicitly contained in $\boldsymbol{z}_{t+1}$, an equivalent representation of the equilibrium conditions is

$$E_{\boldsymbol{\epsilon}'} f_j(\boldsymbol{z}, \boldsymbol{s}, \boldsymbol{c}, \boldsymbol{z}', \boldsymbol{s}', \boldsymbol{c}') = 0, \qquad j = 1, \dots, n_c, \tag{8}$$

with primes denoting next-period variables.

Solving the model amounts to finding a time-invariant decision rule, i.e. a function $\varphi^\star : \mathbb{R}^{n_z} \times \mathbb{R}^{n_s} \to \mathbb{R}^{n_c}$ such that $\boldsymbol{c} = \varphi^\star(\boldsymbol{z}, \boldsymbol{s})$ verifies all the conditions in equation (8):

$$E_{\boldsymbol{\epsilon}'} f_j \left[\boldsymbol{z}, \boldsymbol{s}, \varphi^\star(\boldsymbol{z}, \boldsymbol{s}), \boldsymbol{z}', \boldsymbol{s}', \varphi^\star(\boldsymbol{z}', \boldsymbol{s}')\right] = 0, \qquad j = 1, \dots, n_c, \tag{9}$$

[8] More precisely, these n_c equations characterize the solution of the model, under standard assumptions on concavity and smoothness. They can be Euler equations, market-clearing conditions, or versions of Kuhn-Tucker conditions.

for all possible $(\boldsymbol{z}, \boldsymbol{s})$ pairs, given the laws of motions for $\boldsymbol{z}$ and $\boldsymbol{s}$.

Maliar, Maliar, and Winant (2021) replace the problem of finding the true decision rule $\varphi^\star$ by a simpler one. Consider a family of parametric candidate decision rules $\mathcal{S}_\varphi = \{\varphi(\cdot;\boldsymbol{\theta}) : \theta \in \mathbb{R}^p\}$ indexed by the vector of parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. Intuitively, the most accurate approximation to the true decision rule contained in $\mathcal{S}_\varphi$ is the $\varphi(\cdot;\boldsymbol{\theta})$ function that best fits the zero restrictions listed in equation (9). This can be formalized as a residual minimization program: given the distribution of $(\boldsymbol{z}, \boldsymbol{s}, \boldsymbol{\epsilon}')$, the best approximation to $\varphi^\star$ in $\mathcal{S}_\varphi$ is $\varphi(\cdot;\boldsymbol{\theta}^\star)$, where

$$\boldsymbol{\theta}^\star = \operatorname{argmin}_{\boldsymbol{\theta}} \; \Xi(\boldsymbol{\theta}), \tag{10}$$

with

$$\Xi(\boldsymbol{\theta}) := E_{(\boldsymbol{z},\boldsymbol{s})} \left\{ \sum_{j=1}^{n_c} \nu_j \left[E_{\boldsymbol{\epsilon}'} f_j \left(\boldsymbol{z}, \boldsymbol{s}, \varphi[\boldsymbol{z}, \boldsymbol{s}; \boldsymbol{\theta}], \boldsymbol{z}', \boldsymbol{s}', \varphi[\boldsymbol{z}', \boldsymbol{s}'; \boldsymbol{\theta}] \right) \right]^2 \right\} \tag{11}$$

for some weights $\nu_1, \ldots, \nu_{n_c} > 0$ that may affect the relative scaling of the equilibrium conditions. Due to the specific form of the objective function, Maliar, Maliar, and Winant refer to their method as *Euler residual minimization.*

One notable feature of the objective function (11) is the double level of randomness, coming from the innovation $\boldsymbol{\epsilon}'$ on the one hand and the current state variable $(\boldsymbol{z}, \boldsymbol{s})$ on the other. This is an important difference with respect to the equilibrium conditions (9), whose randomness originates from $\boldsymbol{\epsilon}'$ only since $(\boldsymbol{z}, \boldsymbol{s})$ is taken as given. Averaging across possible state variables in equation (11) ensures that the decision rule will be optimized over the full state space, instead of being specifically trained around an arbitrary point. This strategy helps the learning step find an accurate decision rule over the relevant domain.

Because approximating nested expectations is costly from a computational perspective, Maliar, Maliar, and Winant (2021) devise an *all-in-one* (AiO) expectation operator that combines the separate expectation operators present in equation (11) into a single one. Mathematically, AiO expectation is based on the property that

$$\left[E_{\boldsymbol{\epsilon}} g(\boldsymbol{z}, \boldsymbol{\epsilon})\right]^2 = \left[E_{\boldsymbol{\epsilon}_1} g(\boldsymbol{z}, \boldsymbol{\epsilon}_1)\right] \left[E_{\boldsymbol{\epsilon}_2} g(\boldsymbol{z}, \boldsymbol{\epsilon}_2)\right] = E_{(\boldsymbol{\epsilon}_1, \boldsymbol{\epsilon}_2)} g(\boldsymbol{z}, \boldsymbol{\epsilon}_1) g(\boldsymbol{z}, \boldsymbol{\epsilon}_2),$$

where $\boldsymbol{z}$ and $\boldsymbol{\epsilon}$ are random vectors, $\boldsymbol{\epsilon}_1$ and $\boldsymbol{\epsilon}_2$ are *independent* random vectors with the *same* distribution as $\boldsymbol{\epsilon}$, and g is a measurable function of appropriate dimensions. Using this result, it is possible to write the objective function as

$$\begin{aligned} \Xi(\boldsymbol{\theta}) = E_{(\boldsymbol{z},\boldsymbol{s},\boldsymbol{\epsilon}_1,\boldsymbol{\epsilon}_2)} \Bigg\{ & \sum_{j=1}^{n_c} \nu_j \left[f_j \left(\boldsymbol{z}, \boldsymbol{s}, \varphi[\boldsymbol{z}, \boldsymbol{s}; \boldsymbol{\theta}], \boldsymbol{z}', \boldsymbol{s}', \varphi[\boldsymbol{z}', \boldsymbol{s}'; \boldsymbol{\theta}] \right) \right] \Big|_{\boldsymbol{\epsilon}=\boldsymbol{\epsilon}_1} \\ & \times \left[f_j \left(\boldsymbol{z}, \boldsymbol{s}, \varphi[\boldsymbol{z}, \boldsymbol{s}; \boldsymbol{\theta}], \boldsymbol{z}', \boldsymbol{s}', \varphi[\boldsymbol{z}', \boldsymbol{s}'; \boldsymbol{\theta}] \right) \right] \Big|_{\boldsymbol{\epsilon}=\boldsymbol{\epsilon}_2} \Bigg\}, \end{aligned}$$

which can be evaluated using draws from a single composite random vector $\boldsymbol{\omega} := (\boldsymbol{z}, \boldsymbol{s}, \boldsymbol{\epsilon}_1, \boldsymbol{\epsilon}_2)$. Thus, the objective can be written more compactly as

$$\Xi(\boldsymbol{\theta}) = E_{\boldsymbol{\omega}} \left[\sum_{j=1}^{n_c} \nu_j f_{j,1}(\boldsymbol{\omega}) f_{j,2}(\boldsymbol{\omega}) \right] := E_{\boldsymbol{\omega}} g(\boldsymbol{\omega}; \boldsymbol{\theta}), \tag{12}$$

where $f_{j,i}(\boldsymbol{\omega}) = f_j\left(\boldsymbol{z}, \boldsymbol{s}, \varphi[\boldsymbol{z}, \boldsymbol{s}; \boldsymbol{\theta}], \boldsymbol{z}', \boldsymbol{s}', \varphi[\boldsymbol{z}', \boldsymbol{s}'; \boldsymbol{\theta}]\right)|_{\epsilon=\epsilon_i}$.

2.6.2. *DL implementation.* As noted by Maliar, Maliar, and Winant (2021), the obvious similarity between the generic AI problem (1) and the minimization of the expected residual function (12) makes it possible to solve economic models using the DL machinery. In practice, the implementation follows the steps described above: (i) define an empirical counterpart to the expectation function $\Xi(\boldsymbol{\theta})$ in equation (12) and the set of candidate decision rules $\mathcal{S}_\varphi$, (ii) optimize the model, and (iii) evaluate and validate the solution. See Algorithm 2.

As before, steps (i) and (ii) involve replacing expectation functions by empirical averages. An important difference between standard DL applications and solving economic models is that, in the former case, the DGP is unknown but real-world data make it possible to approximate expectations using sample averages, while in the later case, the DGP is known conditional on the candidate decision rule but no real-world data exist. Because of this difference, Maliar, Maliar, and Winant (2021) introduce a slight twist in their DL method: at each step of the algorithm, they use the current decision rule to generate artificial data from the economic model, and build the empirical loss function and gradient from these simulated data. This procedure implies that the training data are resampled from *different* distributions as the learning process goes on, unlike in standard DL setups. As a result, successful learning in economic applications implies the simultaneous convergence of the parameter vector $\boldsymbol{\theta}$ to the loss-minimizing value and of the implied DGP to the ergodic distribution. On the other hand, step (iii) needs no special change.

3. Application: The Stochastic Growth Model

In this section, we apply the DL approach to solve a basic stochastic growth model. We use this example to illustrate in detail all the steps of the algorithm and to show how various design choices affect the performance of the method. Of course, faster solution methods such as local perturbation and projection are likely to outperform DL in this simple setting.[9] The value of our example lies in the central place of the stochastic growth model in modern macroeconomic theory, which makes it an attractive exposition tool. Our codes are available from `https://notes.quantecon.org/submission/65449f6b8f6a1a0016fc4544`.

[9]See Judd (1998) and Fernández-Villaverde, Rubio-Ramírez, and Schorfheide (2016) for reviews of solution methods for macroeconomic models, including perturbation and projection.

Algorithm 2: DL algorithm to solve economic models (Maliar, Maliar, and Winant, 2021)

```
input  : an economic model
output: a decision rule that solves the model
step 1. initialization ;
    find the equilibrium conditions of the model, i.e. the functions $Z$, $S$, and $f$ in
    equations (5), (6), and (8) ;
    define the theoretical loss function $\Xi(\boldsymbol{\theta})$ in equation (12), given the equilibrium
    conditions ;
    define an empirical loss function
        $$\widehat{\Xi}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{j=1}^{n} g(\boldsymbol{\omega}_i;\boldsymbol{\theta}),$$
    given a sample of artificial observations $\{\boldsymbol{\omega}_i\}_{i=1}^{n}$ ;
    define the set of candidate decision rules, i.e. choose the architecture of the neural
    networks in the set $\mathcal{S}_\varphi = \{\varphi(\cdot;\boldsymbol{\theta}) : \theta \in \mathbb{R}^p\}$ ;
    choose an initial parameter vector $\boldsymbol{\theta}$ ;
    choose a stopping criterion $\mathcal{C}$ ;
step 2. learning process ;
    given the current decision rule $\varphi(\cdot;\boldsymbol{\theta})$, simulate the model to produce artificial
    data $\{\boldsymbol{\omega}_i\}_{i=1}^{n}$ ;
    update the parameter vector using stochastic gradient descent, as described in
    Algorithm 1 ;
    go to step 3 if the stopping criterion $\mathcal{C}$ is satisfied; otherwise return to step 2 ;
step 3. evaluation and validation ;
    evaluate the accuracy of the candidate solution $\varphi(\cdot;\boldsymbol{\theta}^\star)$ on a new sample ;
    stop if the solution is accurate; otherwise return to step 1 and update the network
    architecture ;
```

3.1. **The model.** We work with the central-planner version of the model. Given an initial capital stock k_0 and an exogenous process for total factor productivity a_t, a benevolent planner chooses sequences of consumption c_t, capital k_{t+1}, labor h_t, and investment x_t to maximize the expected lifetime utility of a representative consumer. Formally, the planner solves

$$\max_{\{c_t,k_{t+1},h_t,x_t\}_{t=0}^{\infty}} \quad E_0 \sum_{t=0}^{\infty} \beta^t \left[\eta \ln c_t + (1-\eta)\ln(1-h_t)\right]$$

subject to

$$
\begin{aligned}
y_t &= c_t + x_t, && (13)\\
y_t &= a_t k_t^{\alpha} h_t^{1-\alpha}, && (14)\\
k_{t+1} &= (1-\delta)k_t + x_t, && (15)\\
\ln a_t &= \rho \ln a_{t-1} + \epsilon_t. && (16)
\end{aligned}
$$

As usual, y_t denotes total output, $\beta \in (0,1)$ is the subjective discount factor, $\eta \in (0,1)$ is a preference weight, $\alpha \in (0,1)$ is the capital share in production, $\delta \in (0,1)$ is the depreciation rate, $\rho \in (0,1)$ is the persistence of total factor productivity (TFP), and ϵ_t is an independently and identically distributed Gaussian innovation with mean zero and variance $\sigma^2 > 0$.

The optimal choices for consumption, capital, labor, and investment must verify two necessary conditions

$$
\begin{aligned}
(1-\eta)c_t h_t &= \eta(1-\alpha)(1-h_t)y_t, && (17)\\
1 &= \beta E_t \frac{c_t}{c_{t+1}}\left[\alpha \frac{y_{t+1}}{k_{t+1}} + 1 - \delta\right]. && (18)
\end{aligned}
$$

The first condition characterizes the optimal consumption-leisure trade-off, while the second condition is the Euler equation defining the optimal consumption-investment strategy. Equations (13)-(18) form a nonlinear dynamic system of 6 equations in 6 unknowns: a_t, c_t, k_{t+1}, h_t, x_t, y_t. This generic system has no known closed-form solution, calling for the use of numerical techniques.[10]

We calibrate the model at the yearly frequency, taking standard values from the literature (Prescott, 1986; King and Rebelo, 1999). We set the capital share to $\alpha = 0.36$, the discount factor to $\beta = 0.96$, the depreciation rate to $\delta = 0.10$, the preference weight to $\eta = 0.33$, the persistence of TFP to $\rho = 0.92$, and the standard innovation of TFP shocks to $\sigma = 0.014$.

3.1.1. *Writing the model in DL form.* We solve the model using the DL method reviewed in Section 2. As explained, the first step is to cast the model in suitable form. This involves characterizing the vector of shocks $\boldsymbol{\epsilon}_t$, the vectors of exogenous and endogenous states $\boldsymbol{z}_t$ and $\boldsymbol{s}_t$, the vector of controls $\boldsymbol{c}_t$, and the functions $Z(\cdot)$, $S(\cdot)$, and $f(\cdot)$.

With some experience in DSGE modeling, this characterization is straightforward. The only shock is the TFP innovation, so that $n_\epsilon = 1$ and $\boldsymbol{\epsilon}_t = \epsilon_t$. This shock is associated with a single exogenous state, the level of productivity; hence, $n_z = 1$ and $\boldsymbol{z}_t = a_t$. In absence of consumption habits or investment adjustment costs, the only endogenous state variable is the capital stock, so that $n_s = 1$ and $\boldsymbol{s}_t = k_t$. The remaining variables are controls: $n_c = 4$

[10]One special case in which the model admits an exact solution is full capital depreciation, i.e. $\delta = 1$. Under this restriction, the solution features constant equilibrium labor and saving rate: $h_t = \gamma/(1+\gamma) := \overline{h}$ with $\gamma := (1-\alpha)\eta/[(1-\alpha\beta)(1-\eta)]$, $y_t = a_t k_t^{\alpha} \overline{h}^{1-\alpha}$, $c_t = (1-\alpha\beta)y_t$, $k_{t+1} = x_t = \alpha\beta y_t$.

and $\boldsymbol{c}_t = (c_t,\, h_t,\, x_t,\, y_t)'$. Finally, the function $Z(\cdot)$ corresponds to the law of motion of productivity (16), function $S(\cdot)$ corresponds to the capital accumulation equation (15), and the $n_c = 4$ entries of function $f(\cdot)$ corresponds to the four equilibrium conditions (13), (14), (17), and (18).

Putting everything together, we can write the function $f(\cdot)$ in equation (7) as

$$f(\boldsymbol{z}_t, \boldsymbol{s}_t, \boldsymbol{c}_t, \boldsymbol{z}_{t+1}, \boldsymbol{s}_{t+1}, \boldsymbol{c}_{t+1}) = \begin{bmatrix} y_t - c_t - x_t \\ y_t - a_t k_t^{\alpha} h_t^{1-\alpha} \\ (1-\eta)c_t h_t - \eta(1-\alpha)(1-h_t)y_t \\ \beta \frac{c_t}{c_{t+1}} \left(\alpha \frac{y_{t+1}}{k_{t+1}} + 1 - \delta\right) - 1 \end{bmatrix}.$$

From a numerical perspective, it is useful to simplify the model further by leveraging the various conditions linking the variables. Consider the reduced control vector $\boldsymbol{c}_t = \phi_t$, where $\phi_t := c_t/y_t$ is the consumption share of output. With the current stock of capital k_t and the current level of productivity a_t given, knowledge of $\boldsymbol{c}_t$ allows one to solve for all relevant variables: h_t follows from equation (17) given ϕ_t; output y_t follows from the production function (14) given a_t, k_t, and h_t; consumption follows from knowing ϕ_t and y_t; investment x_t follows from the resource constraint (13); next period capital k_{t+1} follows from the accumulation equation (15). In addition, as discussed below, the fact that the unique element of $\boldsymbol{c}_t$ belongs to the $(0, 1)$ interval conveniently guides the choice of the activation function for the final layer of the network.

The remaining equilibrium condition that needs to be verified is the Euler equation (18), resulting in the alternative function $f(\cdot)$,

$$f(\boldsymbol{z}_t, \boldsymbol{s}_t, \boldsymbol{c}_t, \boldsymbol{z}_{t+1}, \boldsymbol{s}_{t+1}, \boldsymbol{c}_{t+1}) = \beta \frac{c_t}{c_{t+1}} \left(\alpha \frac{y_{t+1}}{k_{t+1}} + 1 - \delta\right) - 1, \tag{19}$$

where we keep implicit that c_t and k_{t+1} are functions of $(\boldsymbol{z}_t, \boldsymbol{s}_t, \boldsymbol{c}_t)$, while c_{t+1} and y_{t+1} are functions of $(\boldsymbol{z}_{t+1}, \boldsymbol{s}_{t+1}, \boldsymbol{c}_{t+1})$.

Given this expression for $f(\cdot)$ and a candidate policy function $\boldsymbol{c}_t = \varphi(\boldsymbol{z}_t, \boldsymbol{s}_t; \boldsymbol{\theta})$, it is straightforward to form the DL objective function (12), which takes the form

$$\Xi(\boldsymbol{\theta}) = E_{\boldsymbol{\omega}} f_1(\boldsymbol{\omega}) f_2(\boldsymbol{\omega}), \tag{20}$$

where $\boldsymbol{\theta}$ is the vector of parameters of the neural network, $\boldsymbol{\omega} := (\boldsymbol{z}, \boldsymbol{s}, \boldsymbol{\epsilon}_1, \boldsymbol{\epsilon}_2)$ is the random vector bundling the states and the two sets of independent shocks, and $f_i(\cdot)$ refers to function $f(\cdot)$ evaluated at shock $\boldsymbol{\epsilon}_i$.

3.1.2. *Network architecture and evaluation.* The second step is to choose the architecture of the neural network. In line with our pedagogical purpose, we consider many possible designs and explore how the various architecture choices affect the performance of the DL approach. To be specific:

- We vary the *depth* by considering networks with one and two hidden layers. Given the simplicity of the model, it would be excessive to try deeper architectures.[11]
- We vary the *width* by considering networks with 4 nodes, 16 nodes, and 32 nodes.
- We vary the *activation function* for the input and hidden layers, considering hyperbolic tangent, ReLU, and sigmoid functions. In all cases, we use a sigmoid function for the output layer to ensure that the value of ϕ_t lies in $(0, 1)$, as implied by the model.
- We vary the *number of observations* n' included in the mini-batches, considering $n' = 1$, 10, 100, and $1,000$.
- We vary the *learning algorithm*, considering basic stochastic gradient descent (SGD) and the Adam algorithm of Kingma and Ba (2017).[12]
- We vary the *learning rate* λ, considering four values between 0.1 and 0.0001 for Adam and making the appropriate adjustment for SGD.[13]

Together, these variations represent 576 different pairs of architecture and learning algorithm[14]. In all cases, we train the network using $5,000$ gradient iterations and measure performance at every step, which allows us to assess how much training is required to solve the model.

We evaluate the accuracy of the neural-network solutions using the unit-free Euler equation error of Judd and Guu (1997) and Barillas and Fernández-Villaverde (2007), defined as

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

with $\phi = \varphi(a, k; \boldsymbol{\theta})$, $h = \eta(1-\alpha)/[\eta(1-\alpha) + (1-\eta)\phi]$, $c = \phi a k^{\alpha} h^{1-\alpha}$, $k' = (1-\delta)k + (1-\phi)ak^{\alpha}h^{1-\alpha}$, $a' = \exp[\rho \ln(a) + \epsilon']$, $\phi' = \varphi(a', k'; \boldsymbol{\theta})$, $h' = \eta(1-\alpha)/[\eta(1-\alpha) + (1-\eta)\phi']$, $y' = a'(k')^{\alpha}(h')^{1-\alpha}$, and $c' = \phi' y'$. As noted above, all model equations are exactly verified conditional on ϕ, except for the Euler equation. This is why the EEE statistic provides a summary accuracy measure for the DL solution. In addition, it has a simple economic

[11]In general, economic models are simple enough that deep architectures would be excessively complex. For instance, both Maliar, Maliar, and Winant (2021) and Azinovic, Gaegauf, and Scheidegger (2022) use networks with only two hidden layers to solve complex models with heterogeneous agents.

[12]As explained in Algorithm 1, SGD updates the value of the parameter vector $\boldsymbol{\theta}$ based on the current value of the gradient and a fixed learning rate λ. On the other hand, Adam averages the gradient over several iterations to create momentum and automatically adjusts the learning rate along the process. Both elements tend to speed up convergence and avoid local minima compared to SGD.

[13]The endogenous adjustment of the learning rate in Adam makes it difficult to compare the values of λ in the two algorithms. We make the comparison formal in Appendix A, where we explain how we fix the learning rate for SGD based on the Adam value.

[14]These represent 2 choices for the number of layers x 3 choices for the number of nodes x 3 choices for the activation function x 4 choices for the number of observations n' x 4 choices for the learning rate x 2 choices for the training algorithm.

interpretation: a value of 0.01 implies that agents make a mistake of \$1 for every \$100 spent, a value of 0.001 implies a mistake of \$1 for every \$1,000 spent, and so on.

When computing the loss function $\Xi(\boldsymbol{\theta})$ and the Euler equation error $EEE(a,k)$, we sample productivity $\ln(a)$ from its ergodic distribution $\mathcal{N}[0,\sigma^2/(1-\rho^2)]$ and capital k from a normal distribution $\mathcal{N}(\mu_k,\sigma_k^2)$, where μ_k and σ_k^2 denote the average and the standard deviation of k implied by the first-order approximation of the model.[15]

To initialize the vector of parameters of the neural network $\boldsymbol{\theta}$, we use the closed-form solution that the model admits when $\delta = 1$, i.e. with full capital depreciation (see Footnote 10). In practice, this solution is $\phi = 1 - \alpha\beta$, and we run an initial training step to have the network learn this value for various productivity-capital draws (a,k). An alternative possibility would be to use the standard first-order approximation of the policy rule to initialize $\boldsymbol{\theta}$.

3.1.3. *Results.* This section presents our main results.[16]

We start by evaluating how the accuracy of the DL solution changes with the network architecture and the training procedure. To do so, we characterize the distributions of the Euler equation errors conditional on the various design choices. We report the results in Figures 2 and 3, which respectively focus on networks trained by SGD and by Adam.

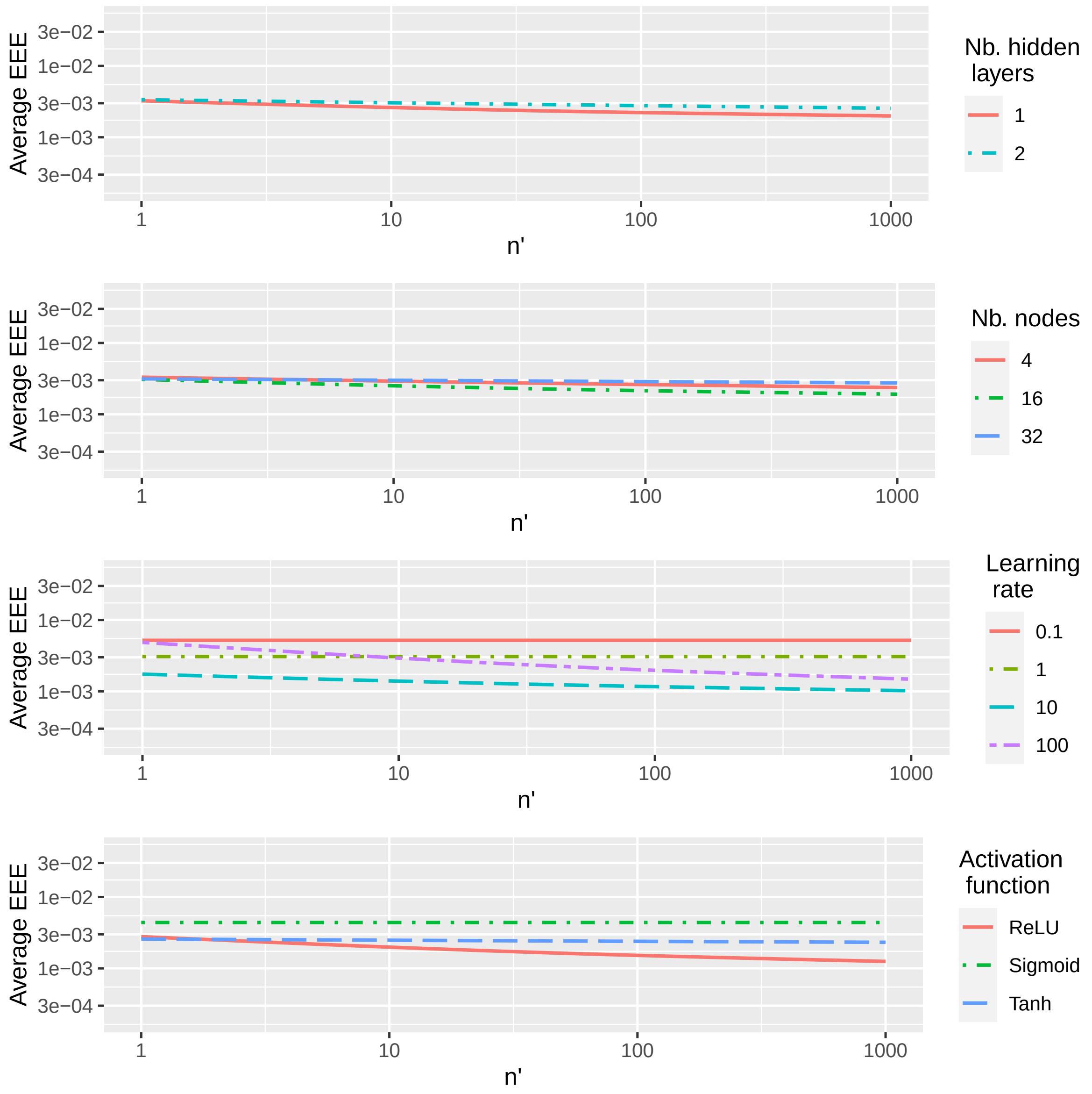
In each figure, the first row considers the role of the network depth, the second row deals with the network width, the third row focuses on the learning rate, and the fourth row looks at the activation function. Each row plots the average Euler equation error $E[|EEE(a,k)|]$ as a function of the number of observations n' used to train the network, conditional on the relevant design characteristic. As commonly done, we use a logarithmic scale to facilitate the interpretation of the Euler equation errors.

Looking at the two figures, a first result is that Adam training tends to outperform SGD training, leading to more accurate solutions on average. The superiority of Adam over SGD, which is well documented in the DL literature (Kingma and Ba, 2017), originates from the various adjustments to gradient descent implemented in the algorithm. A second advantage of Adam over SGD is also clear in the charts, namely the fact that increasing the number of draws n' used to compute the gradient during each training step leads to a much larger accuracy improvement with Adam training than with SGD training. Intuitively, using more information at each step improves the accuracy of each gradient estimate, accelerating descent and helping the training process converge to a better solution.

[15]We checked *ex post* that these bounds provide good coverage of the ergodic set, based on simulations from the DL solution. For more complex models, an alternative would be to add a loop to the algorithm in order to draw the endogenous state variable from its simulated ergodic distribution, while checking for the convergence of the distribution. This would obviously slow down the execution time.

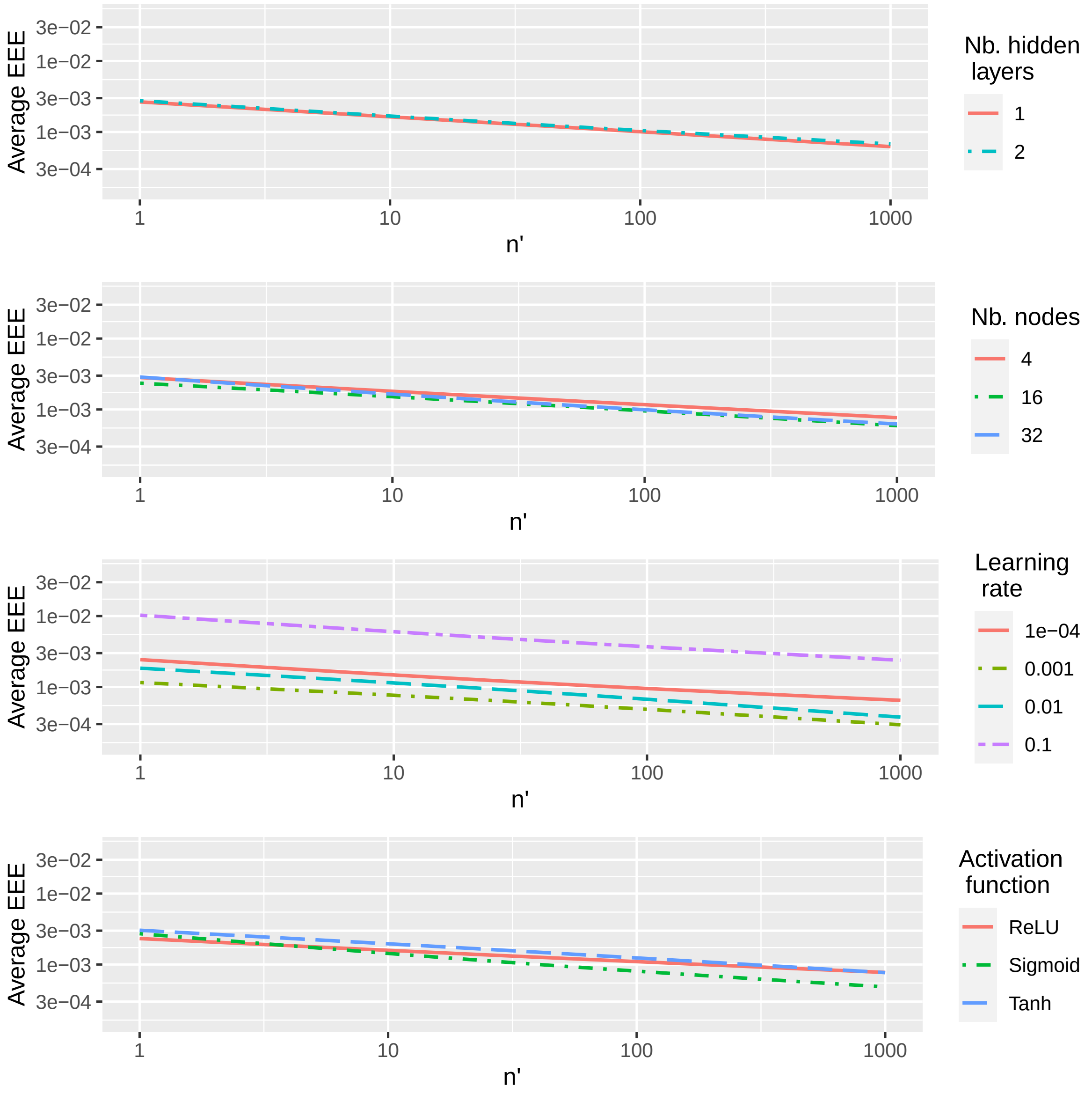
[16]We run our simulations using Pytorch and Python version 3.10.12. Pytorch is the machine learning framework used by Meta. The computer has a 8 cores, an Intel Xeon CPU 2.20GHz, and a 32GB RAM.

FIGURE 2. Design choices and accuracy of DL solution — SGD training



Notes. Each row presents the average of the Euler equation error $E[|EEE(a,k)|]$ across designs, conditional on a given architecture: rows vary the number of hidden layers (first row), the number of nodes in the hidden layer (second row), the learning rate (third row), and the activation function in the input and hidden layers (fourth row). All rows plot the error as a function of the number of observations n' per mini-batch.

FIGURE 3. Design choices and accuracy of DL solution — Adam training



Notes. Each row presents the average of the Euler equation error $E[|EEE(a,k)|]$ across designs, conditional on a given architecture: rows vary the number of hidden layers (first row), the number of nodes in the hidden layer (second row), the learning rate (third row), and the activation function in the input and hidden layers (fourth row). All rows plot the error as a function of the number of observations n' per mini-batch.

The top row in each figure shows that neural networks with one or two hidden layers have similar performance, even though networks with a single layer have a slight edge, especially when trained by SGD. The second row shows that performance is also quite similar across networks of different width. Depending on the training algorithm, networks with 16 or 32 nodes have the highest accuracy, but the improvement with respect to the simpler networks with only 4 nodes is small. Overall, these findings indicate that, given the simplicity of the economic model we consider, a simple architecture performs well and is thus preferable.

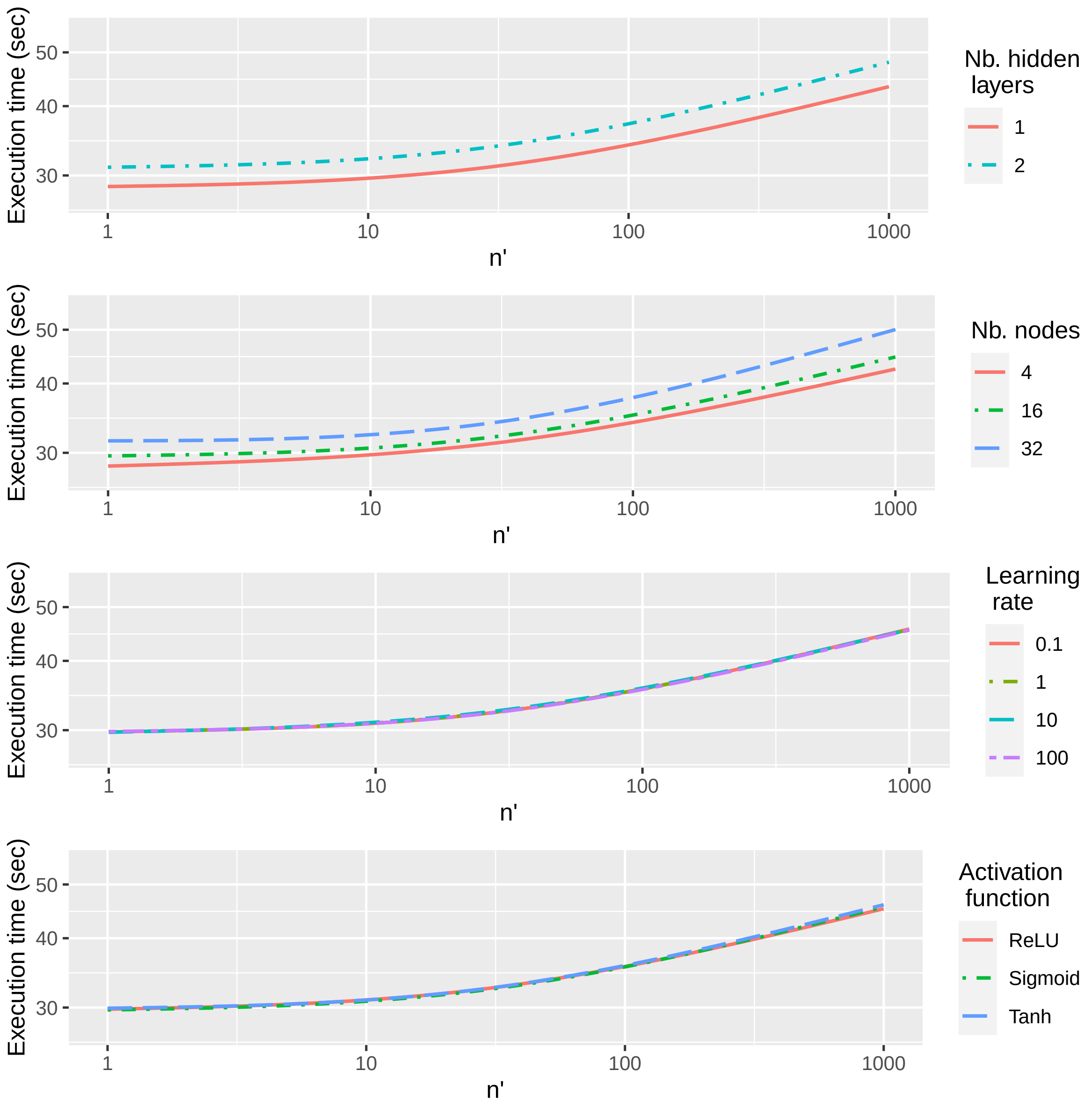
The third row in each figure highlights the importance of the learning rate for accurate DL solutions. In general, too high a learning rate might lead to convergence failures, for instance repeated coefficient oscillations, while too low a learning rate prevents efficient coefficient updates. We can see such effects in Figure 2, which shows that SGD training performs poorly when the learning rate is low, but improves with intermediate rates. We confirmed this insight by trying lower learning rates for SGD, ranging from 0.1 to 0.0001. Such small values lead to essentially impossible learning and very inaccurate solutions. Similarly, we see in Figure 3 that Adam training works worst for the high and low learning rates and best for intermediate values. Thus, even in this simple example, a careful selection of the learning rate is crucial for obtaining accurate DL solutions.

The fourth row in each figure considers the role of the activation functions used in the input and hidden layers of the networks. With SGD, the ReLU function clearly outperforms the sigmoid and hyperbolic tangent alternatives, generating smaller average approximation errors and gaining more from the use of extra training information. With Adam, the performance of the three functions we consider is broadly similar, with a small edge for the sigmoid function when a lot of draws are used to compute the gradient at each step. Thus, the choice of the activation function matters for accuracy, with the best function depending on other hyperparameter choices such as the optimizer and the number of draws used in training.

As alluded to above, design choices also affect computing time. To get an idea of the implied trade-off between accuracy and speed, Figures 4 and 5 report the computing time required to perform $5,000$ parameter updates by gradient descent under the various design choices discussed before. Unsurprisingly, all panels show that increasing the number of draws n' used for each gradient evaluation leads to an increase in computing time. Average computing times are also broadly similar between SGD and Adam training, which favors the latter algorithm since it generally leads to more accurate solutions.

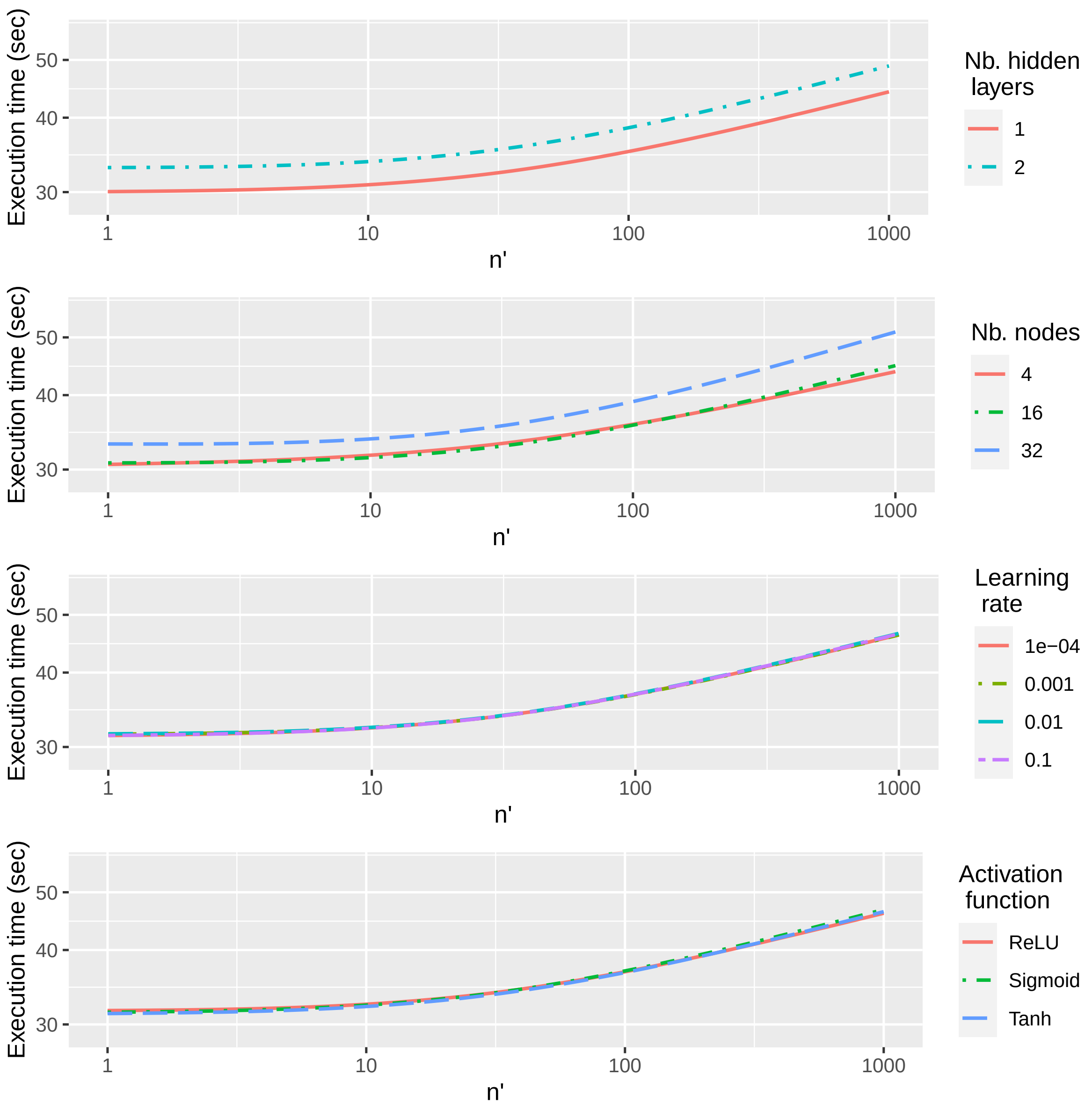
Also expected is the fact that more complex architectures, such as those featuring one additional hidden layer or more nodes per layer, require more time than simpler designs. Interestingly, the choice of the activation function does not seem to affect computing time in this simple setup, although the derivative of the ReLU function can be computed more efficiently than the derivative of the alternative functions we consider. Of course, the value

FIGURE 4. Design choices and computing time — SGD training



Notes. Each row presents the average of the computing time required to perform 5,000 parameter updates across designs, conditional on a given architecture: rows vary the number of hidden layers (first row), the number of nodes in the hidden layer (second row), the learning rate (third row), and the activation function in the input and hidden layers (fourth row). All rows plot the computing time as a function of the number of observations n' per mini-batch.

FIGURE 5. Design choices and computing time — Adam training



Notes. Each row presents the average of the computing time required to perform 5,000 parameter updates across designs, conditional on a given architecture: rows vary the number of hidden layers (first row), the number of nodes in the hidden layer (second row), the learning rate (third row), and the activation function in the input and hidden layers (fourth row). All rows plot the computing time as a function of the number of observations n' per mini-batch.

of the learning rate has no impact on computation time, although it obviously affects the accuracy of the solution.

We can summarize our observations from Figures 2 to 5 as follows:

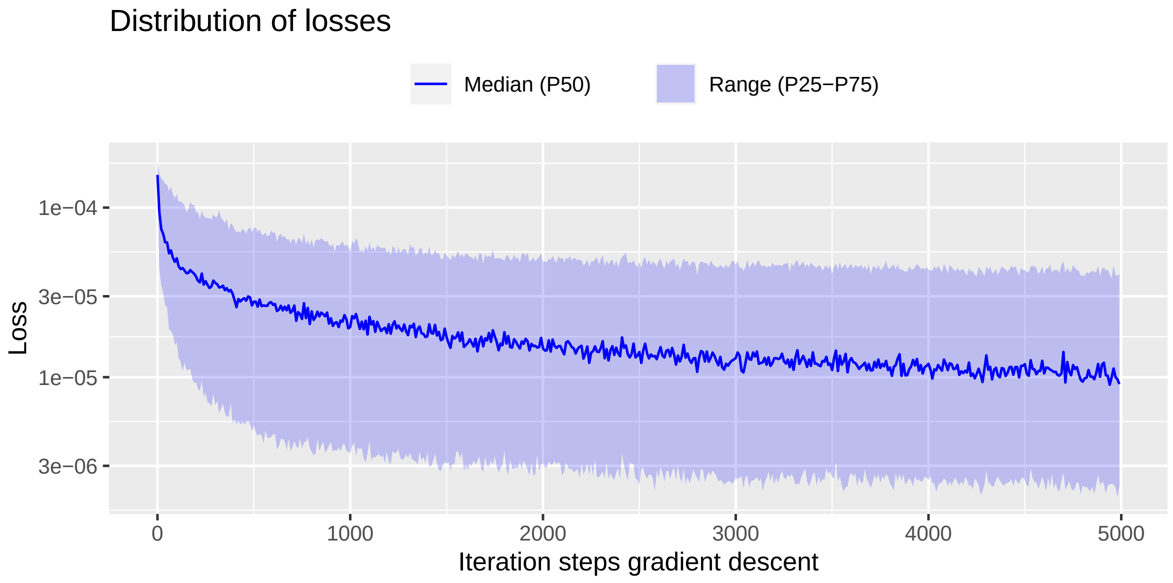
- The Adam algorithm leads to more accurate solutions than SGD without notable difference in computing time.
- Most economic models are simple enough that an architecture with two hidden layers is sufficient.
- The number of nodes should depend on the structure of model. Four nodes are enough for the simple stochastic growth model, but more complex environments may require wider networks. For instance, Maliar, Maliar, and Winant (2021) use 2^6 nodes to solve a Krusell-Smith model and Azinovic, Gaegauf, and Scheidegger (2022) use 2^{10} nodes to solve a large OLG model.
- The choice of the learning rate is crucial: too low a rate prevents an efficient update of the parameter vector while too high a rate may lead to convergence issues.
- In our application, the choice of the activation function has little impact on the accuracy of the approximation. In larger DL applications, the ReLU activation function may be preferable due to its good gradient properties.
- The number of observations n' should not be too low but, above a certain level, there is a trade-off between accuracy and computation time.
- In the context of our simple stochastic growth model, the architecture leading to the most accurate solution features a single hidden layers with 16 nodes and a sigmoid activation function, trained using the Adam algorithm with a learning rate of 10^{-3} and 1,000 observations for each evaluation of the gradient.

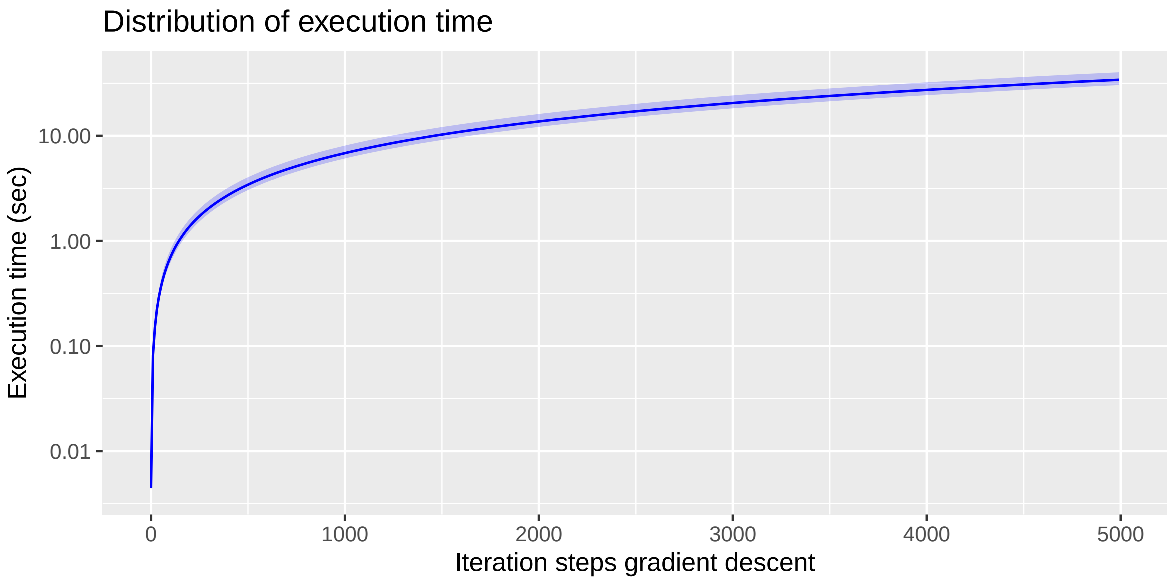
Of course, accuracy improvements are not uniform over time because gradient-based methods tend to yield strong improvement during the first iterations, when the slope of the objective function is most pronounced, and smaller improvement after a while, when the slope flattens near the optimum. This pattern implies that the speed-accuracy trade-off changes with the number of gradient iterations, as shown in Figure 6 in the context of our example. At the start of the training process, updates of the parameter vector lead to a sharp decrease in losses, which signals strong improvement in the accuracy of the DL solution.[17] As learning progresses, further gradient iterations bring less useful information and the loss function slowly stabilizes. In our example, most of the learning occurs during the first 1,000 or 2,000 iterations, with the additional steps providing only modest accuracy gains.

Finally, we compare the DL solution to the basic first-order linear approximation of the model. To do so, we focus on the architecture leading to the most accurate solution, described above. We present the results in Figure 7. The top panel reports the policy functions for

[17]The loss function is $\Xi(\boldsymbol{\theta})$, defined by equation (20) and computed at every gradient iteration. In contrast, the Euler equation error reported in Figures 2 and 3 is only computed after the final iteration.

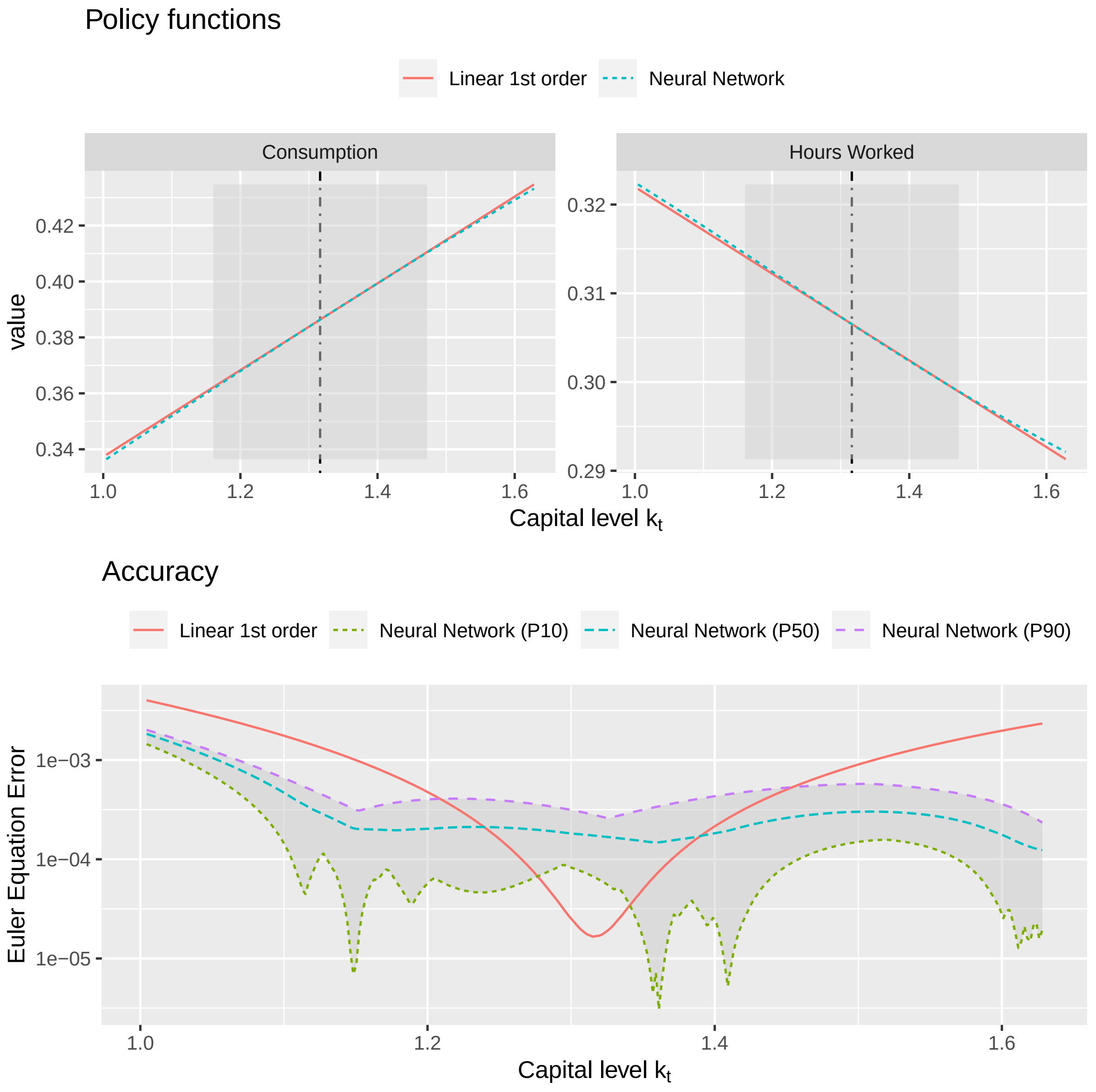
FIGURE 6. Trade-off between accuracy and speed during training





Notes. The distributions of loss function (top panel) and of the average computing time (bottom panel) are computed across design choices, conditional on the number of gradient descent steps. The loss function is $\Xi(\boldsymbol{\theta})$, defined in equation (20). The charts report the median value of each statistic, together with the 25th-75th quantile band across draws.

FIGURE 7. Comparison of the DL and first-order perturbation solutions



Notes. The top panel reports the policy functions for consumption c_t and hours worked h_t in terms of current capital k_t, when productivity is at its deterministic steady-state level $a_t = 1$. In the top panel, the vertical line represents the non-stochastic steady state value for k_t, while the shaded area represents the 95% confidence interval for k_t. The solid red line presents the first-order linear approximation of the model, while the dotted blue line presents the DL solution. The bottom panel reports the distribution of the average Euler equation errors, calculated by integrating aggregate productivity using Gaussian quadrature. The dispersion of Euler equation errors for the DL solution comes from the randomness introduced by the learning process, as neural networks with identical architecture still present different weights after training on a finite sample of draws. To take this uncertainty into account, we train 10 different networks with the same architecture and we report the distribution of Euler equation errors for the DL solution (10th, 50th, and 90th percentiles).

consumption c_t and hours worked h_t in terms of current capital k_t, when productivity is at its deterministic steady-state level $a_t = 1$. We consider a wide range of capital values, which encompasses the ergodic set of the model. The bottom panel reports the distribution of the average Euler equation errors associated with the DL solution.

By definition, the first-order approximation yields linear policy rules. There is a little more curvature in the DL policy rules, which are slightly concave for consumption and slightly convex for hours worked. This curvature reflects the economic forces at play in the model: due to decreasing returns to capital, the incentives to save and work are relatively large when capital is low and relatively small when capital is large. Nevertheless, it is clear that the DL solution almost coincides with the linear approximation over most of the state space, which signals the lack of non-linearity in the frictionless stochastic growth model.

Another way to compare the solutions is to look at their accuracy across the state space. The linear approximation yields small errors close to the deterministic steady-state of the model and larger errors away from it. This property reflects the local nature of the linear solution, which has high accuracy near the approximation point but whose performance deteriorate away from it. Instead, the approximation error associated with the DL solution is flatter across the state space: it is larger than that of the linear approximation by an order of magnitude near the steady state, but smaller for values of capital more than one-standard-deviation away from k_{ss}. This is in line with the global nature of DL approach, which aims at high accuracy across the whole relevant state space.

4. Conclusion

This report provided an introduction to solving economic models using deep learning techniques. We offered a simple and rigorous overview of deep learning methods and illustrated their applicability to the study of economic models by focusing on the simple stochastic growth model. Finally, we emphasized how various choices related to the design of the deep learning solution affect the accuracy of the results, providing some guidance for potential users of the method.

References

Azinovic, M., L. Gaegauf, and S. Scheidegger (2022): "Deep Equilibrium Nets," *International Economic Review*, 63(4), 1471–1525.

Barillas, F., and J. Fernández-Villaverde (2007): "A Generalization of the Endogenous Grid Method," *Journal of Economic Dynamics and Control*, 31(8), 2698–2712.

Barron, A. R. (1994): "Approximation and Estimation Bounds for Artificial Neural Networks," *Machine Learning*, 14, 115–133.

Christiano, L. J., and J. D. M. Fisher (2000): "Algorithms for solving dynamic models with occasionally binding constraints," *Journal of Economic Dynamics and Control*, 24(8), 1179–1232.

Cybenko, G. (1989): "Approximation by Superpositions of a Sigmoidal Function," *Mathematics of Control, Signals, and Systems*, 2(4), 303–314.

de Araujo, D. G., S. Doerr, L. Gambacorta, and B. Tissot (2024): "Artificial Intelligence in Central Banking," BIS Bulletins 84, Bank for International Settlements.

Fernandez-Villaverde, J., G. Nuno, G. Sorg-Langhans, and M. Vogler (2020): "Solving high-dimensional dynamic programming problems using deep learning," *Unpublished working paper*.

Fernández-Villaverde, J., J. Rubio-Ramírez, and F. Schorfheide (2016): "Solution and Estimation Methods for DSGE Models," in *Handbook of Macroeconomics*, ed. by J. B. Taylor, and H. Uhlig, vol. 2 of *Handbook of Macroeconomics*, chap. 9, pp. 527–724. Elsevier.

Fernández-Villaverde, J., S. Hurtado, and G. Nuño (2019): "Financial Frictions and the Wealth Distribution," NBER Working Papers 26302, National Bureau of Economic Research, Inc.

Goodfellow, I., Y. Bengio, and A. Courville (2016): *Deep Learning*. MIT Press, `http://www.deeplearningbook.org`.

He, K., X. Zhang, S. Ren, and J. Sun (2015): "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034. IEEE.

Hochreiter, S. (1998): "The Vanishing Gradient Problem during Learning Recurrent Neural Nets and Problem Solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(2), 107–116.

Hornik, K., M. Stinchcombe, and H. White (1989): "Multilayer Feedforward Networks are Universal Approximators," *Neural Networks*, 2(5), 359–366.

Judd, K. L. (1998): *Numerical Methods in Economics*, no. 0262100711 in MIT Press Books. The MIT Press.

Judd, K. L., and S.-M. Guu (1997): "Asymptotic Methods for Aggregate Growth Models," *Journal of Economic Dynamics and Control*, 21(6), 1025–1042.

King, R. G., and S. T. Rebelo (1999): "Resuscitating Real Business Cycles," *Handbook of Macroeconomics*, 1, 927–1007.

Kingma, D. P., and J. Ba (2017): "Adam: A Method for Stochastic Optimization," arXiv, 1412.6980.

LeCun, Y. A., L. Bottou, G. B. Orr, and K.-R. Müller (2012): "Efficient Backprop," in *Neural networks: Tricks of the trade*, pp. 9–48. Springer.

LEPETYUK, V., L. MALIAR, AND S. MALIAR (2020): "When the U.S. catches a cold, Canada sneezes: A lower-bound tale told by deep learning," *Journal of Economic Dynamics and Control*, 117, 103926.

LESHNO, M., V. Y. LIN, A. PINKUS, AND S. SCHOCKEN (1993): "Multilayer Feedforward Networks with a Nonpolynomial Activation Function Can Approximate Any Function," *Neural Networks*, 6(6), 861–867.

MALIAR, L., S. MALIAR, AND P. WINANT (2021): "Deep Learning for Solving Dynamic Economic Models," *Journal of Monetary Economics*, 122, 76–101.

PRESCOTT, E. C. (1986): "Theory Ahead of Business-Cycle Measurement," in *Carnegie-Rochester Conference Series on Public Policy*, vol. 25, pp. 11–44. Elsevier.

APPENDIX A. LEARNING ALGORITHMS

This appendix provides a more formal comparison between SGD and the Adam learning algorithm. It also explains how we set the learning rate for SGD based on the Adam value.

A.1. **SGD algorithm.** As noted in Algorithm 1, the updating rule for SGD is

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \lambda_{SGD} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i), \tag{21}$$

where λ_{SGD} is the learning rate and $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i)$ denotes the gradient of loss function evaluated at the current parameter guess $\boldsymbol{\theta}_i$.

A.2. **Adam algorithm.** As described in Kingma and Ba (2017), the updating rule for Adam is more complex:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \lambda_{Adam} \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \frac{\mathbf{m}_t}{\sqrt{\mathbf{v}_t} + \varepsilon},$$

where λ_{Adam} is the learning rate, $\mathbf{m}_t$ is a moving average of the gradient, $\mathbf{v}_t$ is a moving average of squared gradient elements, and $\varepsilon = 10^{-8}$ is a small number. The operations on vectors, i.e. the square rooting of $\mathbf{v}_t$ and the ratio between $\mathbf{m}_t$ and $\sqrt{\mathbf{v}_t} + \varepsilon$, are performed element-wise, resulting in parameter-specific effective learning rates. The laws of motion of m_t and $\mathbf{v}_t$ verify $\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1) \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i)$ and $\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2) \left[\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i)\right]^2$, where the square of the gradient is applied element-wise and where β_1 and β_2 are coefficients in $(0,1)$.

Expanding the updating rule leads to

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \underbrace{\lambda_{Adam} \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \frac{\beta_1 \mathbf{m}_{t-1}}{\sqrt{\mathbf{v}_t} + \varepsilon}}_{\text{momentum}} - \underbrace{\lambda_{Adam} \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \frac{(1-\beta_1)}{\sqrt{\mathbf{v}_t} + \varepsilon} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i)}_{\text{update using current gradient}}.$$

Omitting the slow-moving momentum part, the expression simplifies to

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \boldsymbol{\lambda} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i), \tag{22}$$

with $\boldsymbol{\lambda} = \lambda_{Adam} \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \frac{(1-\beta_1)}{\sqrt{\mathbf{v}_t}+\varepsilon}$. As noted above, $\boldsymbol{\lambda}$ is a vector of parameter-specific effective learning rates.

Comparing equations (21) and (22) reveals that setting $\lambda_{SGD} = \overline{\boldsymbol{\lambda}}$, where the bar denotes the average across vector entries, yields a SGD update of similar order as the Adam update. In our application, we consider learning rates between 10^{-4} and 10^{-1} for Adam and we find that the average across entries of the weight vector $\overline{\boldsymbol{\lambda}}$ is close to 10^3 during the training of the model. These elements imply that learning rates between 10^{-1} and 10^2 for SGD approximate well the effective Adam rates.