VNU Journal of Science: Mathematics - Physics

Journal homepage: https://js.vnu.edu.vn/MaP

Original Article

Using Deep Learning Neural Networks to Solve Optimization Problems in Economy

Vu Tuan Anh[1,*], Le Thi Thuy Giang[2], Do Thi Loan[2], Pham Van Khanh[2]

[1]*Department of Investment Promotion, Vietnam National University, Hanoi, VNU Town in Hoa Lac, Thach That, Hanoi, Vietnam*

[2]*Institue of Information Technology, Vietnam Academy of Science and Technology; A3 Building, 18 Hoang Quoc Viet, Cau Giay, Hanoi, Vietnam*

Received 18 June 2024

Revised 31 August 2024; Accepted 10 December 2024

Abstract: In this work, we have used multilayer neural networks to solve high-dimensional dynamic programming problems. We propose a deep learning algorithm to efficiently compute the overall solution for this class of problems. Importantly, our method does not rely on integral approximation but instead on derivative approximation. We evaluate the effectiveness of the proposed method through the standard neoclassical growth model.

Keywords: Optimization, HJB equation, machine learning, neural network.

1. Introduction

Artificial intelligence (AI) has notable applications such as image and voice recognition, computer vision support, and autonomous driving as shown in [1]. At the same time, there are many interesting problems that computational economists have not been able to solve, including heterogeneous agent models with large dimensions, large-scale central bank models, circular models' life and complex nonlinear estimation procedures, among many other problems. We specifically introduce an econometric-style deep learning (DL) approach to solve dynamic economic models by reformulating them as nonlinear regression equations via deep learning neural networks.

Authors of [2] showed that solving the multidimensional dynamic programming problems is extremely difficult due to the huge amount of the computation. As the number of states in a dynamic

[*] Corresponding author.
E-mail address: vtanhhsv@gmail.com

https//doi.org/10.25073/2588-1124/vnumap.4949

programming problem increases linearly, the computational burden increases exponentially, both in the number of operations to be performed and in memory requirements. Although much progress has been made with new approaches such as sparse grids over the past decade [3], solving dynamic programming problems with more than 30 state variables and karma is still a challenge.

There are many methods for solving dynamic economic models on their ergodic sets approximated through stochastic simulation, such as indirect inference procedures for lifetime reward maximization, algorithms parameterization expectation (PEA) of Haan and Marcet [4] to minimize the Euler equation residual and the value iteration method of Maliar and Maliar [5] to minimize the Bellman equation residual. There are also methods where unsupervised learning is used, aiming to refine simulation points and identify ergodic sets with irregular shapes. In particular, Judd et al. [6] used clustering of simulated points, Maliar and Maliar [7] combined simulated points in epsilon-distinguishable sets. In contrast, Jirniy and Lepetyuk [8] showed a notable early application of reinforcement learning to solve the Krusell and Smith model [9].

Early applications of neural networks date back to Duffy and McNelis [10] and more recent applications include Duarte [11] Fernández-Villaverde et al., [12]; Lepetyuk et al., [13]; Villa and Valaitis [14]. In these works, neural networks for interpolation instead of polynomial functions were used. There is also a work of Azinovic et al. [15] where the authors used related Euler equation method to solve the problem of large-scale multi-generation dynamic economic modeling. They used deep neural networks and random grid points but focuses only on the Euler equation residual minimization method while we present a unified approach that also is applied to lifetime rewards and Hamilton - Jacobi - Bellman operator.

In this work, a deep learning algorithm is used to solve the difficulty of dimensionality and provide an effective overall solution to high-dimensional dynamic programming problems. Our approach builds four deep neural networks for estimation: the value function of the problem, the policy function, and the associated Karush-Kuhn-Tucker multipliers for equation and inequality constraints. Similarly, one can think of these four deep neural networks as just one big deep neural network with multiple outputs. However, presenting the algorithm in terms of four separate networks simplifies the presentation.

We apply two deep neural networks to the Hamilton-Jacobi-Bellman (HJB) equation to define the dynamic programming problem. Then, construct the loss function by adding the HJB loss function, policy loss function, and constraint loss functions. We train our neural networks by minimizing the loss criterion via mini-batch gradient descent on points drawn from the ergodic distribution of state vectors.

Among all the different neural network architectures, we use deep neural networks because they have been proven to work surprisingly well in many contexts. We demonstrate our algorithm in solving the standard neoclassical growth model using conventional calibration. Computationally, this is a simple problem, but it provides us with a sharp testbed to show how our method works and evaluate its accuracy.

After Introduction, the paper is organized as follows. Section 2 introduces the problems to be solved, Section 3 presents our deep learning algorithm. Section 4 introduces our application to solving the neoclassical growth model. Finally, conclusions are given in Part 5.

2. Math Problem

Our goal is to solve the recursive continuous-time HJB equation:

$$\rho V(x) = \max_{\alpha}\left[\mathrm{r}(\mathrm{x},\alpha) + \nabla_x V(x) f(x,\alpha) + \frac{1}{2} tr(\sigma(x))^T \Delta_x V(x)\sigma(x) \right] \tag{1}$$

satisfying the following inequality constraints and equations:

$$G(x,\alpha) \le 0, H(x,\alpha) = 0 \tag{2}$$

where V: $R^N \to R$ is a value function that depends on the state vector $x \in R^N$, $\nabla V(x) \in R^N$ is the gradient of V, and $\Delta_x V(x) \in R^{NxN}$ is the Hesian matrix of V.

In problem (1), ρ denote the discount rate, $r: R^{NxM} \to R$ is a function that returns an immediate reward depending on the state and control vector $\alpha \in R^M$. The state vector follows the process $f: R^{NxM} \to R^N$ and standard deviation $\sigma: R^N \to R^N$. The problem is constrained by constraint functions $G: R^M \to R^{L_1}$ and $H: R^M \to R^{L_2}$ where $L_1, L_2 \in R$ are the number of inequality and equality constraints, respectively. We denote by $\alpha: R^N \to R^M$ policy function, which is the optimal control vector for each state.

We are interested in solving problem (1) when N is large, that is, we are solving a multidimensional problem. It is well known that, in this case, problem (1) suffers from an acute loss of dimensionality. Grid-based methods quickly become unfeasible because the number of required grid points increases exponentially with N for a given accuracy. Local solution methods such as linearization or higher-order polynomial expansion are not suitable for problems with unusual behavior like kinks or other strong nonlinearities. On the contrary, using algorithm, one can solve both challenges well: globally approximate solutions to high-dimensional problems.

3. Algorithm

Our approach to solving problem (1) is built on the idea of deep learning. We use a neural network as a global nonlinear approximator for both the value and policy functions. We define a loss criterion, including HJB loss and first-order condition (FOC) loss. We then train the neural networks (i.e. update the weights in the network) by selecting points in the state space from their ergodic distribution and minimizing their loss function, using the method gradually reduces the gradient until convergence.

We first define neural networks

$$\tilde{V}\left(x;\Theta^V\right): \mathbb{R}^N \to \mathbb{R}, \tilde{\alpha}\left(x;\Theta^\alpha\right): \mathbb{R}^N \to R^M, \mu\left(x;\Theta^\mu\right): \mathbb{R}^N \to \mathbb{R}^{L_1}$$

and

$$\tilde{\lambda}\left(x;\Theta^\lambda\right): \mathbb{R}^N \to \mathbb{R}^{L_2}$$

These four neural networks are parameterized according to weight vectors $\Theta^V, \Theta^\alpha, \Theta^\mu$ and Θ^λ to approximate:

i) value function V(x);

ii) policy function, and Karush-Kuhn-Tucker (KKT) multipliers;

iii) μ;

iv) λ.

To simplify notation, we include all the weights of the neural network in a matrix

$$\Theta = \left(\Theta^V, \Theta^\alpha, \Theta^\mu, \Theta^\lambda\right)$$

One can think of neural networks as any parameterized function approximator.

Second, we define the loss function, which consists of three components: HJB loss, policy function loss, and constraint loss. For any point in the state space $x \in \mathbb{R}^N$, the HJB error is defined as the difference between the right and left sides of the HJB when we replace the exact values and policy functions with approximations of them:

$$\mathrm{err}_{HJB}\left(x;\Theta\right) \equiv r\left(x,\tilde{\alpha}\left(s;\Theta^{\alpha}\right)\right) + \nabla_x \tilde{V}\left(x;\Theta^{V}\right) f\left(x,\tilde{\alpha}\left(x;\Theta^{\alpha}\right)\right) + \frac{1}{2} tr\left[\sigma\left(x\right)^T \Delta_x \tilde{V}\left(x;\Theta^{V}\right)\sigma\left(x\right)\right] - \rho \tilde{V}\left(x;\Theta^{V}\right)$$

Similarly, the policy function loss is defined as the difference of FOC from zero when we substitute the approximate values into the policy functions:

$$\mathrm{err}_{\alpha}\left(x;\Theta\right) \equiv \frac{\partial r\left(x,\tilde{\alpha}\left(x;\Theta^{\alpha}\right)\right)}{\partial \alpha} + D_{\alpha} f\left(x,\tilde{\alpha}\left(x;\Theta^{\alpha}\right)\right)^T \nabla_x \tilde{V}\left(x;\Theta^{V}\right) -$$
$$D_{\alpha} G\left(x,\tilde{\alpha}\left(x;\Theta^{\alpha}\right)\right)^T \tilde{\mu}\left(x;\Theta^{\mu}\right) - D_{\alpha} H\left(x,\tilde{\alpha}\left(x;\Theta^{\alpha}\right)\right)\tilde{\lambda}\left(x;\Theta^{\lambda}\right)$$

where $D_{\alpha}G \in \mathbb{R}^{L_1 xM}, D_{\alpha}H \in \mathbb{R}^{L_2 xM}$, and $D_{\alpha}f \in \mathbb{R}^{NxM}$ are the submatrices of the Jacobian matrices of G, H and f containing the derivatives with respect to α, respectively.

Finally, obvious losses include:

$$\mathrm{err}_{PF_1}\left(x;\Theta\right) \equiv max\left\{0, G\left(x,\tilde{\alpha}\left(x;\Theta^{\alpha}\right)\right)\right\}; \quad \mathrm{err}_{PF_2}\left(x;\Theta\right) \equiv H\left(x,\tilde{\alpha}\left(x;\Theta^{\alpha}\right)\right)$$

We combine these losses using the squared loss as the loss function:

$$L\left(x;\Theta\right) \equiv \left\|\mathrm{err}_{HJB}\left(x;\Theta\right)\right\|_2^2 + \left\|\mathrm{err}_{\alpha}\left(x;\Theta\right)\right\|_2^2 + \left\|\mathrm{err}_{PF_1}\left(x;\Theta\right)\right\|_2^2 + \left\|\mathrm{err}_{PF_2}\left(x;\Theta\right)\right\|_2^2 \qquad (3)$$

We train the neural network by minimizing the above loss function through gradient descent on points drawn from the ergodic distribution of state vectors.

Efficient implementation of this final step is the key to the success of our algorithm. We start by initializing the neural network weights. Then we perform K learning steps called episodes, where K can be chosen in many different ways. For each episode, we simulate from the state space its ergodic distribution. Computationally, this is very expensive when a closed-form expression for the ergodic distribution is available and is still relatively cheap if the ergodic distribution has to be simulated. We then randomly divide this sample into B mini-lots of size S. For each mini-batch, we determine the mini-batch loss by averaging the loss function over the batch. Finally, we perform mini-batch gradient descent for all network weights, with η_k as the learning rate in the kth episode.

Algorithm 1: Deep learning algorithm for HJB

```
Parameterization of value and policy functions via neural networks $\tilde{V}, \tilde{\alpha}, \tilde{\mu}$ and $\tilde{\lambda}$;
Determine the loss function L as in (3);
Initialize weights for neural networks $\Theta$;
for k = 1, …, K do
    Simulation I gathers sample points $\left\{x_i\right\}_{i=1}^{I}$ from an ergodic distribution $V\left(x\right)$
    Randomly divide these points into B mini-lots of size S: $\left\{\left\{x_{s,b}\right\}_{s=1}^{S}\right\}_{b=1}^{B}$;
    Use the previous weight as the starting weight for the new step: $\Theta_{k,1} = \Theta_k$;
        for b = 1, …, B do
            Determine the average loss function: $\bar{L}\left(\{x_{s,b}\}_{s=1}^{S};\Theta_{k,b}\right) \equiv \frac{1}{S}\sum_{s=1}^{S} L\left(x_{s,b};\Theta_{k,b}\right)$;
            Train $\tilde{V}$: $\Theta_{k,b+1}^{V} = \Theta_{k,b}^{V} - \eta_k \nabla_{\Theta_{k,b}^{V}} \bar{L}\left(\{x_{s,b}\}_{s=1}^{S};\Theta_{k,b}\right)$;
            Train $\tilde{\alpha}$: $\Theta_{k,b+1}^{\alpha} = \Theta_{k,b}^{\alpha} - \eta_k \nabla_{\Theta_{k,b}^{\alpha}} \bar{L}\left(\{x_{s,b}\}_{s=1}^{S};\Theta_{k,b}\right)$;
            Train $\tilde{\lambda}$: $\Theta_{k,b+1}^{\lambda} = \Theta_{k,b}^{\lambda} - \eta_k \nabla_{\Theta_{k,b}^{\lambda}} \bar{L}\left(\{x_{s,b}\}_{s=1}^{S};\Theta_{k,b}\right)$;
            Train $\tilde{\mu}$: $\Theta_{k,b+1}^{\mu} = \Theta_{k,b}^{\mu} - \eta_k \nabla_{\Theta_{k,b}^{\mu}} \bar{L}\left(\{x_{s,b}\}_{s=1}^{S};\Theta_{k,b}\right)$;
        end
    Update weights: $\Theta_{k+1} = \Theta_{k,B+1}$;
end
```

4. Application

4.1. The Neoclassical Growth Model

In this section we consider a standard model, the neoclassical continuous growth model. This problem has a closed-form solution for the policy functions, allowing us to focus on the approximate analysis of the value function. We can then derive the policy function from this method and compare it with the results of the next step in which we directly approximate the policy functions using a neural network.

Let c(t) be the consumption function at time t of a household and k(t) be the capital at time t of a household.

We denote the derivative of capital k with respect to t by $\dot{k} = \frac{dk}{dt} = k_t^{'} \approx k(t+1) - k(t)$ and let f(k) be the production or output function of the household, u(c(t)) be the utility function of the household.

The problem is to find c(t) so that the objective function $\int_0^{\infty} e^{-\rho t} u\left(c\left(t\right)\right) dt$ reaches the maximum value, satisfy the following constraints:

$$\dot{k} = f(k) - \delta k - c$$
$$0 \le c(t) \le f(k(t))$$
$$k(0) = k_0$$

To solve this problem, we create the Hamilton function:

$$H = e^{-\rho t}.u(c) + \lambda.[f(k) - \delta.k - c]$$

We have the first-order conditions of the problem:

$$\begin{cases} \dfrac{\partial H}{\partial c} = 0 \\ \dot{\lambda} = -\dfrac{\partial H}{\partial k} \\ \dot{k} = f(k) - \delta k - c \\ \end{cases}$$

We can transform:

$$\begin{cases} e^{-\rho t} u^{'}(c(t)) - \lambda = 0 \\ \dot{\lambda} = -\lambda f'(k) + \lambda \delta \\ \dot{k} = f(k) - \delta k - c \end{cases}$$

deduce from here:

$$\lambda = e^{-\rho t} u'\left(c(t)\right) \tag{4}$$

$$\frac{\dot{\lambda}}{\lambda} = \delta - f'(k) \tag{5}$$

$$\dot{k} = f(k) - \delta k - c \tag{6}$$

From (4) we deduce:

$$\dot{\lambda} = e^{-\rho t}(-\rho)u'(c) + e^{-\rho t}u''(c)\dot{c} = e^{-\rho t}\left[u''(c)\dot{c} - \rho u'(c)\right]$$

Combining with (5) we obtain:

$$\frac{\dot{\lambda}}{\lambda} = \delta - f'(k) = \frac{u''(c)\dot{c}}{u'(c)} - \rho$$

and infer:

$$\dot{c} = [\delta + \lambda - f'(k)].\frac{u'(c)}{u''(c)} \tag{7}$$

Consider a utility function of the form: $u = \dfrac{c^{1-\gamma}}{1-\gamma}$ and production function: $f(k) = A.k^{\alpha}$

then

$$u' = \frac{(1-\gamma)c^{\gamma}}{1-\gamma} = c^{-\gamma}; u'' = -\gamma.c^{-\gamma-1}, f'(k) = A.\alpha.k^{\alpha-1}$$

Substituting (6) and (7) we get:

$$\begin{cases} \dfrac{\dot{c}}{c} = \dfrac{A.\alpha.k\left(t\right)^{\alpha-1} - \delta - \rho}{\gamma} \\ \dot{k} = A.k(t) - \delta.k - c(t) \end{cases}$$

Thus, we can obtain a closed solution for the neoclassical growth model. We now consider the solution approximated by the neural network.

The model has a basic form, with a single agent deciding whether to save in capital or spend. The HJB equation has the following form:

$$\rho V(k)=\max_{c} U(c)+V'(k)\left[F(k)-\delta * k-c\right] \tag{8}$$

Since the model is only used to demonstrate the algorithm, we use a standard calibration for all parameters in the model. Utility is $u=\frac{c^{1-\gamma}}{1-\gamma}$ with $\gamma=2$. The discount rate ρ is set to 0.06. The production function has the form $f(k)=A.k^{\alpha}$. Here, total factor productivity A is set to 0.5 and α to 0.36. Finally, depreciation is set to 0.05, which gives an annual depreciation rate of 5%. We approximate the value function $V(k)$ with a neural network $\tilde{V}(k;\Theta^V)$ and consumtion function $c(k)$ with a neural network $\tilde{C}(k;\Theta^c)$. HJB loss is defined as:

$$err_{HJB}=\rho\tilde{V}(k;\Theta^V)-U\left(\tilde{C}(k;\Theta^c).\right)-\frac{\partial\tilde{V}(k;\Theta^V)}{\partial k}\left[F(k)-\delta * k-\tilde{C}(k;\Theta^c).\right]$$

and first-order condition error:

$$err_C=(U')^{-1}\left(\frac{\partial\tilde{V}(k;\Theta)}{\partial k}\right)-\tilde{C}(k;\Theta^c).$$

The total error that needs to be minimized is $err_{total}=err_{HJB}+err_C$.

The results are shown in Figs. 1, 2 and 3 below. In particular there is a comparison of results from our neural network approximation with results obtained via a traditional finite element method. Both the value function and the policy function are almost identical to the reference solution, demonstrating the high accuracy of our solution method.

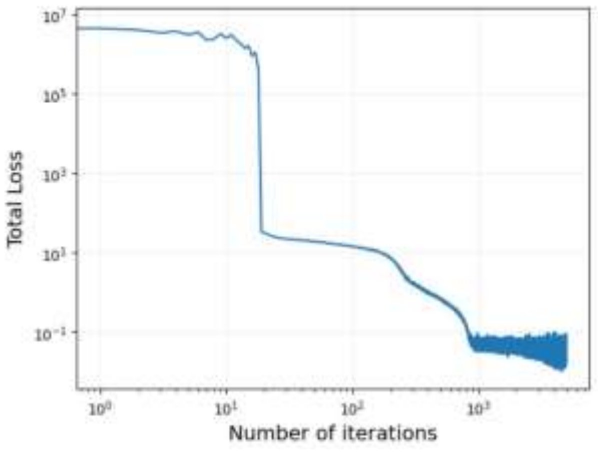

Figure 1. Total loss function.

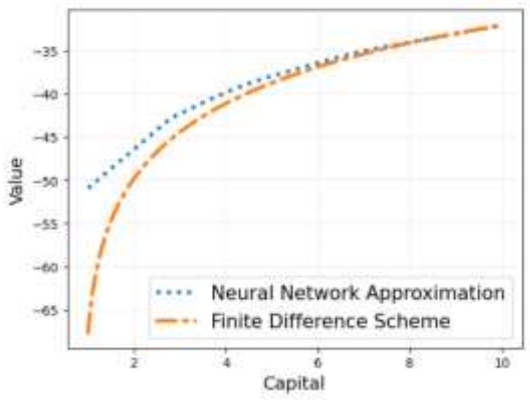

Figure 2. Value function.

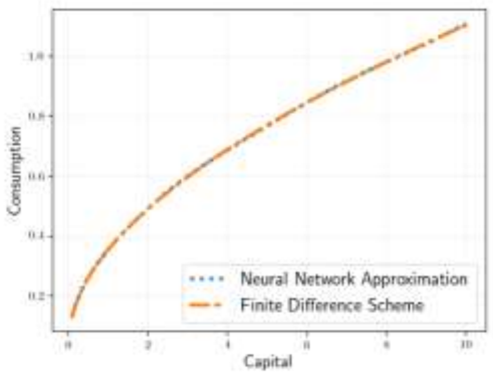

Figure 3. Consumption function (policy function).

4.2. The Overlapping Multigeneration Model

In this section, we consider an overlapping multigeneration model with stochastic output. In this model, the dimensionality of the state space increases linearly with the number of age groups. Therefore, a multidimensional state space arises naturally and makes economic sense. To demonstrate the applicability of our method in a context with multiple assets and sometimes binding constraints we include borrowing constraints and capital adjustment costs, making it illiquid. asset and allow dealers to trade one-period liquid bonds with collateral constraints. Thus, beyond the computational challenges, these extensions allow the model to connect with the literature highlighting the role of different liquid assets in cross-consumption responses to aggregate shock.

The model we are considering has discrete time t = 0,1,... In each period, discrete exogenous shock Z can occur. We assume that the shocks follow a first-order Markov process with a finite support set $\mathcal{Z} = \{1,2,...,Z\}$ with a transition probability matrix $\prod$. We denote the aggregate shock occurring in period t by z_t and the history of aggregate shocks up to period t by $z^t = (z_0,. ..,z_t)$.

We study a model in which agents live for N time periods. There is no certainty about longevity and there is a representative household for each cohort. At each node z^t, a representative household is born. Households distinguish themselves only by birth node z^{t^0}. At time $t \in \{t^0,...,t^0 + N - 1\}$, we define the birth household at the time of birth by its current age $s = t - t^0 + 1$. For example, the consumption level of a household with age s at time t is denoted by $c^s(z^t)$. We omit the explicit dependency on z^t when there is no risk of confusion and write c_s^t . In each period, living agents receive a strictly active labor supply, which depends only on the agent's age. The agent's labor source at age s is denoted by l^s. The prices of consumer goods are normalized to one. Furthermore, we assume that households supply their labor inelastically to the market wage $w\left(z^t\right)=w_t$. At each node z^t, the living agents maximize their remaining time-decomposable discounted expected utility given by:

$$\sum_{i=0}^{N-s} E_t\left[\beta^i u\left(c_{t+i}^{s+i}\right)\right] \tag{9}$$

where $\beta<1$ is the discount factor. Furthermore, we assume that the utility function $u:\mathbb{R}_{++}\rightarrow\ \mathbb{R}$ is smooth, strictly monotonically increasing, strictly concave and satisfies the Inada condition: $\lim_{c\to 0} u'\left(c\right)=\infty$.

There are two ways for households to shift consumption over time. First, households can save capital at risk, which is subject to adjustment costs and is therefore illiquid. The illiquid savings of household s in period t is denoted by $a^s\left(z^t\right)=a_t^s$. The savings will become capital in the next period:

$$a_t^s=k_{t+1}^{s+1},\qquad \forall t,\forall s\in\left\{1,2,\ldots,N-1\right\} \tag{10}$$

where k_t^s denotes the amount of illiquid capital of age group s at the beginning of holding period t. Households sell their capital to the firm at the market price r_t. Capital gains will accumulate in the household's illiquid account. The amount deposited into a household's illiquid account in period t is calculated by:

$$\Delta_t^s=a_t^s-k_t^s r_t \tag{11}$$

Illiquid capital due to convex adjustment costs is given by

$$\frac{\nu}{2}\left(\Delta_t^s\right)^2 \tag{12}$$

where ν is the level of capital adjustment costs. Furthermore, we impose an exogenous borrowing limit $\underline{a}$, which can be set to zero when borrowing is prohibited:

$$a_t^s\geq\underline{a} \tag{13}$$

Besides saving illiquid, risky capital, households can buy term bonds at market price p_t. Bonds have a net supply of zero. A bond that promises a payoff of 1 in the next period. Let b_t^s be the amount of bonds that household s holds in period t and d_t^s be the number of bonds that household s buys in period t, such that

$$b_{t+1}^{s+1}=d_t^s \tag{14}$$

Bonds are liquid in the sense that there are no costs associated with adjusting bond holdings. However, the sale of bonds is subject to collateral constraints. To model collateral constraints, we use

$$\psi d_t^s+a_t^s\geq 0 \tag{15}$$

where κ is an exogenous constant and only capital act as collateral. In case the household selling the bond is unable to pay in the next period, the corresponding capital value will be transferred to the household buying the bond; otherwise there will be no punishment. Therefore, households default only if the value of the collateral is less than the promised bond amount. The actual payment of the bond is then given by

$$\min\{\psi r_t, 1\} \tag{16}$$

In the numerical tests below, we have chosen κ high enough that the bond is truly a risk-free asset and default does not occur in equilibrium. Therefore, $\min\{\psi r_t, 1\} = 1$, which we use in the following for clearer notation. Therefore, the budget constraint of household s in period t is given by

$$c_t^s + p_t d_t^s + a_t^s + \frac{v}{2}\left(\Delta_t^s\right)^2 = l^s \omega_t + b_t^s + r_t k_t^s \tag{17}$$

Finally, agents are born and die without any properties - ie

$$k_t^1 = b_t^1 = 0; a_t^N = d_t^N = 0$$

There is a single representative firm with a Cobb–Douglas production function in which total factor productivity (TFP) and depreciation depend only on exogenous shocks. Each period, after a shock occurs, the firm purchases capital and hires labor to maximize profits, accepting the given price. The production function is given by

$$f\left(K_t, L_t, z_t\right) = \eta_t K_t^{\alpha} L_t^{1-\alpha} + K_t\left(1 - \delta_t\right) \tag{18}$$

1where K_t is the total amount of capital purchased, L_t is the total number of workers hired, α is the capital share in production, η_t represents stochastic TFP, and δ_t represents the stochastic depreciation rate.

Capital prices (r_t) and wages (w_t), as well as bond prices (p_t), are determined by market clearing in competitive spot markets for consumption, capital, labor, and bonds. promissory note.

Approximating the equilibrium point with deep neural networks (DNN)

To describe how to solve the model presented in the above section using DNN, we proceed in two steps. Firstly, we define the functional rational expectations equilibrium for the economy presented. Secondly, describes how to use a deep neural network to search for an approximate recursive equilibrium using the projection method.

Our proposed algorithm aims to approximate a recursive equilibrium: a function that maps the state of the economy to options and prices, consistent with equilibrium conditions. We define the function-dependent equilibrium is a system of equilibrium functions

$$\theta = \left[\theta_a^T, \theta_\lambda^T, \theta_d^T, \theta_\mu^T, \theta_p^T\right] : \mathcal{Z} \times \mathbb{R}^{2N} \to \mathbb{R}^{4(N-1)+1}$$

where:

- $\theta_a : \mathcal{Z} \times \mathbb{R}^{2N} \to \mathbb{R}^{N-1}$: is the capital investment function
- $\theta_\lambda : \mathcal{Z} \times \mathbb{R}^{2N} \to \mathbb{R}^{N-1}$: are the KKT factor functions for ceiling constraints
- $\theta_d : \mathcal{Z} \times \mathbb{R}^{2N} \to \mathbb{R}^{N-1}$: is the bond investment function
- $\theta_\mu : \mathcal{Z} \times \mathbb{R}^{2N} \to \mathbb{R}^{N-1}$: are the KKT factorization functions for parallel constraints
- $\theta_p : \mathcal{Z} \times \mathbb{R}^{2N} \to \mathbb{R}^{N-1}$: is the bond price function

for all states $x := \left[z, k^T, b^T\right]^T \in \mathcal{Z} \times \mathbb{R}^{2N}$, in which $z \in \mathcal{Z}$ is the exogenous shock $k = \left[k_1, k_2, \ldots, k_N\right]^T$, is the capital holding portfolio $b = \left[b_1, b_2, \ldots, b_N\right]^T$ along with the bond holding portfolio are endogenous states with $k_1 = 0$ and $b_1 = 0$ for all i=1,…,N-1:

$$\left(1+\nu\Delta(x)_i\right)u'\left(c(x)_i\right)=\beta E_z\left[r(x_+)\left(1+\nu\Delta(x_+)_{i+1}\right)u'\left(c(x_+)_{i+1}\right)\right]+\theta_\lambda(x)_i+\theta_\mu(x)_i \quad (19)$$

$$\theta_\lambda(x)_i\theta_a(x)_i=0 \quad (20)$$

$$\theta_a(x)_i\geq 0 \quad (21)$$

$$\theta_\lambda(x)_i\geq 0 \quad (22)$$

simultaneously:

$$\theta_p(x)u'\left(c(x)_i\right)=\beta E_z\left[u'\left(c(x_+)_{i+1}\right)\right]+\psi\theta_\mu(x)_i \quad (23)$$

$$\theta_\mu(x)_i\left(\theta_a(x)_i+\psi\theta_d(x)_i\right)=0 \quad (24)$$

$$\theta_a(x)_i+\psi\theta_d(x)_i\geq 0 \quad (25)$$

$$\theta_\mu(x)_i\geq 0 \quad (26)$$

$$\sum_{i=1}^{N-1}\theta_d(x)_i=0 \quad (27)$$

where

$$x_+=\left[z_+,0,\theta_a(x)^T,0,\theta_d(x)^T\right]^T \quad (28)$$

where z$^+$ is the random shock in the next period and:

$$\Delta(x)_i:=\theta_a(x)_i-r(x)x_{1+i} \quad (29)$$

$$r(x)=f_K\left(\sum_{i=1}^{N}x_{1+i},\sum_{i=1}^{N}l^i(x_1),x_1\right), \quad (30)$$

$$w(x)=f_L\left(\sum_{i=1}^{N}x_{1+i},\sum_{i=1}^{N}l^i(x_1),x_1\right) \quad (31)$$

$$c(x)_i=l^iw(x)+x_{1+i+N}-\theta_p(x)\theta_d(x)_i-\Delta(x)_i-\frac{\nu}{2}\left(\Delta(x)_i\right)^2 \quad (32)$$

Calculating an approximate value in this model means finding an approximate value for the $4\cdot(N-1)+1$ equilibrium functions such that the above equations are nearly satisfied for all exogenous shocks and all the values received by the 2N-dimensional endogenous state.

Training algorithm. The goal of our solution framework is to estimate the equilibrium functions θ with deep neural networks. To do so, we combine four ingredients. In general, the four components are given by:

i) a suitable class of approximate functions;

ii) the loss function measures the quality of a given approximation at a given state;

iii) update mechanism to improve the approximation;

and

iv) sampling method to select the update state and evaluate the approximate quality.

Specifically, for our algorithm we chose a deep neural network as the function approximator. The loss function is implemented using errors under equilibrium conditions and the neural network parameters are updated using variations of mini-batch gradient descent. To update the parameters of the neural network, as well as to evaluate the quality of the approximation, we sample states from a simulated path of the economy.

The combination of these four ingredients allows us

a) use a large number of states to evaluate the quality of our approximation,

b) sample states from an approximate ergodic distribution of states in the economy,

c) handle irregular geometries of the ergodic set of states, and

d) approximate equilibrium functions have kinks and strong nonlinearity.

Next, we outline each component separately and then combine them into an algorithm.

Function approximation. We use densely connected convolutional neural networks as function approximators because they incorporate a set of desired properties. Neural networks are universal function approximators (Hornik et al., 1989), can resolve discrete localities, have high accuracy, nonlinear characteristics, and can handle large amounts of data. multidimensional input data.

Given hyperparameters $\left\{K,\{m_i\}_{i=1}^{K},\{\sigma_i(.)\}_{i=1}^{K}\right\}$ and training parameters ρ, neural network $\mathcal{N}_\rho$ mapping encoding:

$$x \to \mathcal{N}_\rho(x) = \sigma_K\left(W_K ... \sigma_2\left(W_2\sigma_1\left(W_1x + b_1\right) + b_2\right) ... + b_K\right)$$

where $W_i \in \mathbb{R}^{m_{i+1}\times m_i}$ are the weight matrices and $b_i \in \mathbb{R}^{m_{i+1}}$ are vectors commonly called bias vectors. The vector ρ represents the set of all elements of the weight matrix and bias vector. K is called the number of layers of the neural network and mi is the number of nodes in layer i. The nonlinear functions σi are called activation functions and are applied element by element to each element of the vector: $\sigma_i(x) = \left[\sigma_i(x_1),\sigma_i(x_2),....,\sigma_i\left(x_{m_{i+1}}\right)\right]^T$.

Therefore, a densely connected feedforward neural network is given by a sequence of matrix vector multiplications followed by an activation function.

Loss function. The goal of our algorithm is to approximate the equilibrium functions θ using a neural network. In this section, we will introduce the loss function: an approximate measure of quality at a given state of the economy.

Let ρ denote the set of trainable parameters of the neural network, and let the neural network, with the set of parameters ρ, be denoted as Nρ. The neural network maps the state x into approximate equilibrium functions:

$$\mathcal{N}_\rho : \mathcal{Z}\times\mathbb{R}^{2N} \to \mathbb{R}^{4(N-1)+1} : x \to \mathcal{N}_\rho(x)$$

where

$$x := \left[z, k^T, b^T\right]^T \in \mathcal{Z}\times\mathbb{R}^{2N}$$

and

$$\mathcal{N}_\rho(x) = \hat{\theta}(x) = \left[\hat{\theta}_a^T(x), \hat{\theta}_\lambda^T(x), \hat{\theta}_d^T(x), \hat{\theta}_\mu^T(x), \hat{\theta}_p^T(x)\right]^T$$

$$=: \left[\hat{a}(x)_1,...,\hat{a}(x)_{N-1},\hat{\lambda}(x)_1,...,\hat{\lambda}(x)_{N-1},\hat{d}(x)_1,...,\hat{d}(x)_{N-1},\hat{\mu}(x)_1,...,\hat{\mu}(x)_{N-1},\hat{p}(x)\right]^T$$

The latter is called the training set, which we denote as D_{train}. Given parameters ρ and a set of D_{train} states, we define the loss function as the average squared error of all equilibrium conditions is $l_{D_{train}}(\rho)$.

Update mechanism. This subsection describes how to use the loss function to optimize the trainable parameters ρ. The loss function is determined such that a smaller value corresponds to a lower mean squared error under equilibrium conditions. Therefore, parameters are considered "good" if they minimize the loss function. Due to the functional structure of deep neural networks, variations of the gradient descent method are often used to optimize the parameters ρ. Gradient descent updates the parameters stepwise in the direction in which the loss function decreases—that is:

$$\rho_k^{new} = \rho_k^{old} - \alpha^{learn} \frac{\partial l_{D_{train}}\left(\rho^{old}\right)}{\partial \rho_k^{old}}, \qquad \forall k \in \left\{1,...,length\left(\rho\right)\right\}$$

The parameter $\alpha^{learn} > 0$ that governs how much the parameters are adjusted with each gradient descent step is called the learning rate.

Sample. As described above, the neural network parameters ρ are chosen to minimize the loss function. The training set generation (D_{train}) procedure used to achieve a good approximation of the equilibrium functions for the set of ergodic states of the economy.

Since we want to use approximate equilibrium functions to simulate the modeled economy, they must provide a good approximation of the states that will be visited during the simulation. Therefore, we chose to train the neural network in the states visited on the simulated path of the economy. To do so, we start with an arbitrary, economically feasible starting state and randomly initialized neural network parameters ρ. We then simulate the subsequent T − 1 periods based on the approximate equilibrium functions provided by the neural network. Since our method directly approximates the equilibrium functions, simulating the evolution of the economy is computationally cheap. The resulting T simulated states of the economy make up our data set $D_{train}^0 = \left\{x_1^0,...,x_T^0\right\}$. We call the set T of D_{train} simulation periods an episode. We split this input data into mini-batches—smaller subsets of size m with random membership—and perform gradient descent steps on each subset. Completion of an epoch is defined as when the entire D_{train} data set is passed through the algorithm. Each epoch, the neural network parameters are updated T/m times. Next, we set $x_1^1 = x_T^0$ and use the update parameters of the neural network to simulate the next T − 1 periods, create a new training set $D_{train}^1 = \left\{x_1^1,...,x_T^1\right\}$, and repeat the process. Since we can generate large amounts of training data in this setting, overfitting is not a major concern. As the neural network learns better parameter values, the simulated states become better representations of the set of ergodic states of the economy.

Since the loss function only requires evaluating a set of possible states of the economy, our algorithm does not require us to obtain D_{train} training data from the simulation. If the ergodic set of states of the economy is known, if one wants to approximate equilibrium functions over a larger set, or if one wants to improve the quality of the approximation in specific regions of the state space, then you can choose the D_{train} training set accordingly.

Algorithm. Now, we summarize how to combine the components described in the previous sections to calculate approximate recursive equilibria with DNN.

Starting from a randomly initialized neural network, our algorithm iterates between generating a new Dtrain training set by simulating the desired amount of states and improving the neural network parameters ρ by how to perform a variation of gradient descent steps on the training set. The error on a new set of simulated states is the out-of-sample error and can be used to evaluate the out-of-sample quality of the approximation. Because we use neural networks to directly estimate the equilibrium functions, simulating a newly computed set of points is cheap. Therefore, we can set the number of epochs to train on each set of simulation points to one. Therefore, each simulation state is used for only a single gradient descent step, to protect our algorithm from overfitting. Algorithm 1 provides the pseudocode:

```
Algorithm 2
Data:
T (length of 1 episode), $N^{epochs}$ (Number of epochs per episode), $N^{iter}$ (Maximum number of iterations),
$\tau^{mean}, \tau^{max}$ (necessary threshold for average error and maximum error)
$\varepsilon^{mean}, \varepsilon^{max}$ (starting value for average current error and maximum current error)
$\rho^0$ (initialization parameters for neural network), (initial value of state), i=0 (counter variable),
$\alpha^{learn}$ (learning speed)
Result:
success (boolean variable if thresholds are reached)
$\rho^{final}$ (final parameters of the neural network)
While $\left(\left(i < N^{iter}\right) \wedge \left(\left(\varepsilon^{mean} \geq \tau^{mean}\right) \vee \left(\varepsilon^{\max} \geq \tau^{\max}\right)\right)\right)$ do
    $D^i_{train} \leftarrow \left\{x^i_1, x^i_2, ..., x^i_T\right\}$ (create a new training episode)
    $x^{i+1}_1 \leftarrow x^i_T$ (create a new starting state of an episode)
    $\varepsilon^{\max} \leftarrow \max\left\{\max_{x \in D^i_{train}} \left|e^{\cdots}_x(\rho)\right|\right\}, \varepsilon^{mean} \leftarrow \max\left\{\frac{1}{T}\sum_{x \in D^i_{train}} \left|e^{\cdots}_x(\rho)\right|\right\}$ (error update)
    for $j \in \left[1,..,N^{epochs}\right]$ do
    (learn $N^{epochs}$ on data)
        for $k \in \left[1,..,length(\rho)\right]$
            $\rho^{i+1}_k = \rho^i_k - \alpha^{learn} \frac{\partial l_{D^i_{train}}\left(\rho^i\right)}{\partial \rho^i_k}$
            (perform gradient reduction through updating neural network parameters)
        end
    end
    $i \leftarrow i + 1$ (updated episode number)
end
If $i = N^{iter}$ then return ($success \leftarrow False, \rho^{final} \leftarrow \rho^i$ )
otherwise return ($success \leftarrow True, \rho^{final} \leftarrow \rho^i$ )
```

Neural network hyperparameters. We use a deep neural network with two hidden layers to solve our benchmark model. Heuristically, we found it useful to augment the state of the economy with redundant information before passing it to the neural network. The input layer consists of 8 + 4 • A input nodes, with A = 6 creating 32 input nodes.

After the input layer, the neural network has two hidden layers with 100 hidden nodes activated at the first layer, 50 hidden nodes at the second layer. The output layer includes (6 − 1) = 5 nodes, activated using softplus functions to ensure that non-negative constraints are met.

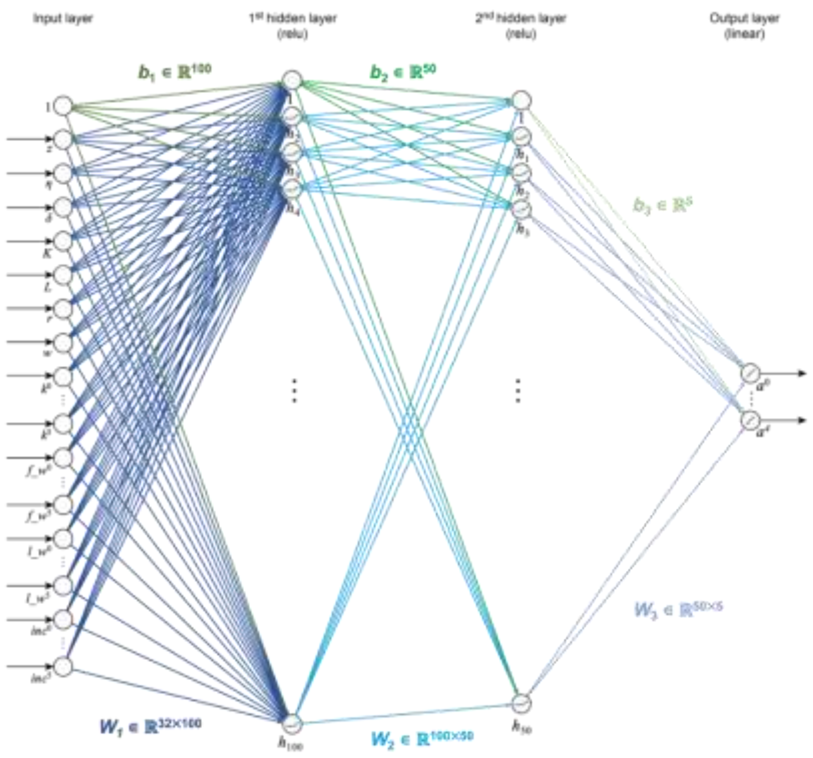
Input layer
1st hidden layer (relu)
2nd hidden layer (relu)
Output layer (linear)
$b_1 \in \mathbb{R}^{100}$
$b_2 \in \mathbb{R}^{50}$
$b_3 \in \mathbb{R}^{5}$
$W_1 \in \mathbb{R}^{32\times100}$
$W_2 \in \mathbb{R}^{100\times50}$
$W_3 \in \mathbb{R}^{50\times5}$
h_{100}
h_{50}

Figure 4. Neural network diagram to approximate the solution of the optimization problem.

A schematic illustration of the neural network architecture is provided in Fig. 4 for clarity, without including redundant information. To avoid excessive complexity, we keep the number of epochs per episode, N^{epoch}, small. The learning rate must be chosen sufficiently small, and the mini-batch size large enough, to ensure that the mini-batch gradient descent steps are not overly noisy. On a heuristic note, we found it beneficial to reduce the learning rate and increase the mini-batch size toward the end of the training process to fine-tune the neural network parameters.

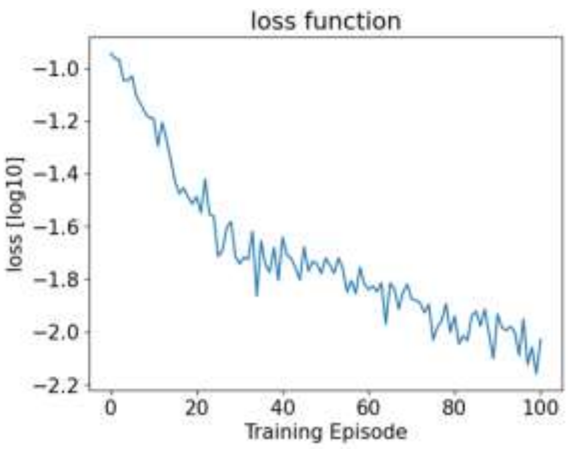

Figure 5. Loss function during training.

The Figure 5 below shows the loss function behavior over 100 training episodes. The vertical axis represents the logarithm (base 10) of the loss, indicating a scale for very small values. The horizontal axis represents the training episodes. The loss decreases steadily as training progresses, indicating that the model is learning effectively. While there are minor fluctuations in the loss values, particularly in the middle and later stages, the general trend remains downward. By the end of 100 episodes, the loss appears to stabilize around −2.2 (log scale), suggesting that the model is nearing convergence.

We have implemented the algorithm and obtained the results of the solution by deep learning method compared with the analytical method, showing that the results are very similar as shown in the figure below:

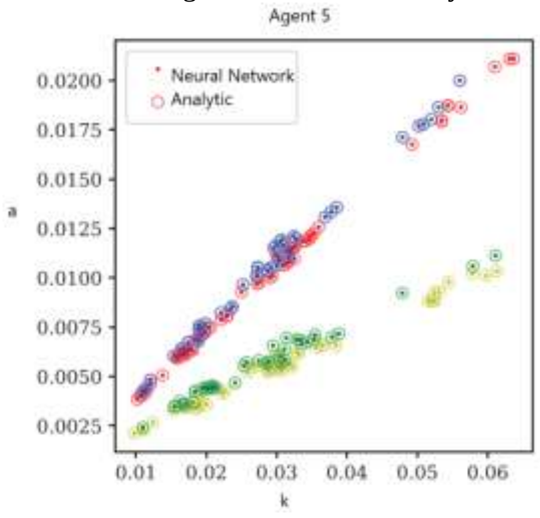

Figure 6. Relationship between capital k and assets a in the 5th generation according to two solution methods: neural network approximation and analysis.

Figure 6 compares the performance of a neural network model (red dots) with an analytic solution (hollow circles) for Agent 5. The k-axis represents the independent variable k, and the a-axis represents the dependent variable a. The neural network predictions (red dots) closely align with the analytic solutions (hollow circles) across different clusters, indicating that the neural network is effectively approximating the analytic solution.

5. Conclusion

In summary, we have used multilayer neural networks to solve optimization problems and specifically applied it to an optimization problem in neoclassical economic theory, including multigenerational optimization models.

Our algorithm is based on HJB to process the value function and uses boundary conditions and additional constraints to make the loss function converge. This method is feasible because the computation of derivatives in these loss functions is efficiently performed by the implementation of the neural network.

We also apply our algorithm to solve the standard neoclassical growth model and compare our results with those of a reference method using a finite difference scheme. This allows us to test the accuracy of our method and show that our policy function approximation performs well.

In addition, we extend this model to include multi-generational optimization problems, demonstrating the applicability of the method in contexts with multiple assets and sometimes binding constraints. The obtained results show that the neural network can approximate the policy and value functions in these models well.

Acknowledgements

The Institue of Information Technology (IoIT), Vietnam Academy of Science and Technology is acknowledged for the financial support under grant number CS24.05.

References

[1] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, Massachusetts Institute Technology Press, 2016.
[2] R. Bellman, Dynamic Programming, Princeton University Press, 1958.
[3] J. Brumm, S. Scheidegger, Using Adaptive Sparse Grids to Solve HighDimensional Dynamic Models, Econometrica, Vol. 85, 2017, pp. 1575-1612.
[4] W. D. Haan, A. Marcet, Solving the Stochastic Growth Model by Parameterized Expectations, Journal of Business and Economic Statistics, Vol. 8, pp. 31-34.
[5] L. Maliar, S. Maliar, Parameterized Expectations Algorithm: How to Solve for Labor Easily, Computational Economics, Vol. 25, 2005, pp. 269-274.
[6] K. L. Judd, L. Maliar, S. Maliar, Numerically Stable and Accurate Stochastic Simulation Approaches for Solving Dynamic Models, Quant Econom, Vol. 2, 2011, pp. 173-210.
[7] L. Maliar, S. Maliar, Merging Simulation and Projection Approaches to Solve High-dimensional Problems with an Application to A New Keynesian Model, Quant Econom, Vol. 6, 2015, pp. 1-47.
[8] A. Jirniy, V. Lepetyuk, A Reinforcement Learning Approach to Solving Incomplete Market Models with Aggregate Uncertainty, SSRN: https://papers. ssrn.com/sol3/papers.cfm?abstract_id=1832745, 2011 (accessed on: April 1st, 2024).
[9] P. Krusell, A. Smith, Income and Wealth Heterogeneity in the Macroeconomy, Journal of Political Economy, Vol. 106, 1998, pp. 868-896.
[10] J. Duffy, P. McNelis, Approximating and Simulating the Real Business Cycle Model: Parameterized Expectations, Neural Networks, and the Geneticalgorithm, Journal of Economic Dynamics and Control, Vol. 25, No. 9, 2001, pp. 1273-1303.
[11] V. Duarte, Machine Learning for Continuous-Time Economics, SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3012602, 2018 (accessed on: April 1st, 2024).
[12] J. F. Villaverde, S. Hurtado, G. Nuño, Financial Frictions and the Wealth Distribution, NBER Working Paper 26302, 2019.
[13] V. Lepetyuk, L. Maliar, S. Maliar, When the U.S. Catches A Cold, Canada Sneezes: A Lower-Bound Tale Told by Deep Learning, Journal of Economic Dynamics and Control, Vol. 117, 2020, pp. 103926.
[14] A. Villa, V. Valaitis, Machine Learning Projection Methods for Macro-Finance Models. SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id= 3209934, 2019 (accessed on: April 1st, 2024).
[15] M. Azinovic, J. Luca, S. Scheidegger, Deep Equilibrium Nets, SSRN: https://ssrn.com/abstract=3393482, 2020 (accessed on: April 1st, 2024).