Deep Learning Algorithms for Solving Finite-Horizon Models[*]

work-in-progress

Jeppe Druedahl[†]
Jacob Røpke[‡]

This version: January 31, 2025
First version: January 31, 2025

Abstract

We develop and test deep learning algorithms for solving finite-horizon horizon models. We show how the use of neural networks and simulation help tame the curse of dimensionality. We succeed in accurately solving a consumption-saving model with 8 durable goods, convex adjustment costs and irreversible investment in a few hours on a single GPU. We verify the stability and robustness of the solution methods across multiple models with limited hyperparameter tuning. We also show how one of the algorithms can be used to solve a non-convex model with both discrete and continuous choices. Our algorithms are implemented as a user-friendly open source Python package, where it is only necessary to specify high-level information such as the distribution of initial states and shocks, the transition function and the reward function.

Python package: github.com/NumEconCopenhagen/EconDLSolvers

[*] We are grateful for helpful comments from Marlon Azinovic, Anders Munk-Nielsen, Thomas Høgholm Jørgensen, Bertel Schjerning, Adam Jørgensen, Annasofie Olesen and Raphael Huleux. We thank Martin Andreas Kildemark for excellent research assistance. Our errors are naturally our own. Funding from Independent Research Fund Denmark (Grant 2065-00028B) and Riksbankens Jubileumsfond (Grant M23-0019) is gratefully acknowledged. Center for Economic Behavior and Inequality (CEBI) is a center of excellence at the University of Copenhagen, founded in September 2017, financed by a grant from the Danish National Research Foundation (Grant DNRF134).

[†] CEBI, Department of Economics, University of Copenhagen. E-mail: jeppe.druedahl@econ.ku.dk.

[‡] CEBI, Department of Economics, University of Copenhagen. E-mail: jro@econ.ku.dk.

1 Introduction

The economic literature on life-cycle consumption-saving models uses setups, where households make a small subset of the choices actual households make in real life. The curse of dimensionality implies it is not feasible to solve more realistic models. Following Fernández-Villaverde et al. (2024b), we hypothesize deep learning algorithms can tame the curse. We are the first to evaluate this claim in detail for large-scale consumption-saving models with both many states, many choices and many constraints. We focus on finite-horizon models, where the effect of the age structure must also be learned.

We first use the well-understood buffer-stock model to show that the deep learning algorithms can provide a good approximate solution to a very precisely solved model. We next use a consumption-saving model with multiple durable goods subject to convex adjustment costs and with irreversible investment. With one durable good the standard methods are faster, but not more accurate. With two durable goods it is a close race, and with three durable goods the deep learning algorithms clearly outperform the standard, but highly optimized, dynamic programming methods both in terms of speed and accuracy. To find an accurate solution using standard dynamic programming with three durable goods require many terabytes of RAM and takes days, while the deep learning algorithm take less than two hours on a single mid-range GPU.[1]

Our main contribution is to show that the deep learning algorithms opens up the possibility of solving much larger models than previously considered. Specifically, we solve a model with 8 durable goods. This implies that there together with non-durable consumption are 9 choices, and that there together with cash-on-hand and permanent income are 10 states. There is also 9 inequality constraints, one borrowing constraint and 8 no disinvestment constraints on the durable goods. We confirm multiple algorithms reach very similar solutions in terms of both the implied expected discounted utility across agents and the full distribution of economic

[1] As standard dynamic programming method we use the Endogenous Grid Method (EGM) of Carroll (2006) and Nested Endogenous Grid Method (NEGM) of Druedahl (2021). For efficiency these algorithms are written in C++ and use parallelization with 72 CPU cores (two Intel Xeon Gold 6254, 3.1 Ghz). For the deep learning we use a single Nvidia L40, which is mid-range GPU. The hardware costs are comparable at around $9000.

outcomes. This corroborates we have found the global solution. For standard dynamic programming the curse of dimensionality implies that the solution time when adding a new durable good grows with a factor higher than the number of grid points used for each durable good.[2] For the deep learning algorithms it also takes longer to achieve the same level of accuracy when adding a new durable good, but we find the increase in computational cost to be much lower.

Algorithms Following Maliar et al. (2021), we explore three types of deep learning algorithms:

1. DeepSimulate: Directly maximizing expected discounted utility.
2. DeepFOC: Solving first-order conditions.
3. DeepV: Solving for the value function using the Bellman equation.

The DeepSimulate and DeepFOC methods are generally the fastest and most accurate. But the DeepV remains interesting as it is the most versatile and can be extended to non-convex problems. We also show how using ensemble averaging and incorporating information from the first order conditions in DeepV can improve its performance considerably. We keep the hyperparameters of each of the algorithms fixed across the various models we solve with some minor exceptions. This indicates that the solution algorithms are rather stable and robust. DeepV typically requires more hyper-parameter tuning.

All the deep learning algorithms approximate the policy function, and where applicable, the value function, using deep neural networks. Each algorithm has two key steps: 1) Simulate data, and 2) update the network parameters. Using simulated data is a key part of avoiding the curse of dimensionality as this avoids using (tensor-product) grids, and implies that we only find the solution on the active part of the state space. Both the value-function approach, DeepV, and first-order condition approach, DeepFOC, need a full sample of states in all periods, which we generate

[2] The total number of grid points increases exactly with a factor of the number of grid points used for each durable good. But the workload in each grid point increases as the implied optimization problems and interpolations also grow in dimensionality.

by simulating with the current policy. In contrast, the approach that maximizes expected discounted utility directly, DeepSimulate, only needs a sample of shocks and initial states, which can be directly drawn.

There are both theoretical and practical benefits of using neural networks. Theoretically a number of universal function approximation theorems (Hornik et al., 1989) exist even for non-smooth functions. In practice it is central that neural networks using existing software allow for fast automatic differentiation of the outputs with respect to the the parameters (so-called weights and biases) given the inputs, and are optimized for using GPUs. We use networks with age as an inputs and solve simultaneously across all periods instead of using backward inductions. The benefit of this is that there are many symmetries and hidden patterns across age, which the neural network can learn quickly. A special feature of neural networks are that they allow us to choose a so-called final activation function which constrains the output domain. This e.g. allow us to enforce that the policy network never imply infeasible choices.

The parameter update requires specifying a loss-function, which is specific to the individual algorithm. For DeepSimulate we directly maximize the expected discounted utility, and this algorithm is thus the simplest. For DeepFOC the model solution is formulated as an equation system of first order optimality and slackness conditions, and thus requires that the first order conditions are sufficient. For DeepV the parameters in value network is chosen to minimize the error in the Bellman-equation, and then the parameters in the policy network is chosen to maximize the value-of-choice implied by the current approximation of the value network. The DeepV method is the most involved, but also the most versatile. We show how this method can also be used to solve a consumption-saving model where there is a non-convex adjustment for a durable good.

One benefit of our implementations of DeepFOC and DeepV compared to DeepSimulate is that they allow for easy off-policy exploration. This implies that the training sample will cover a larger part of the state-space. This is useful for ensuring that the policy function is also accurate for agents with e.g. very low or high wealth. It can also speed up convergence by faster exploration of the state space and help the algorithms escape local minima, where e.g. no one saves.

Code: Our algorithms are implemented in a user-friendly open source Python package, where it is only necessary to specify high-level information such as the distribution of initial states and shocks, the transition function and the reward function. Generic algorithms for simulating and solving the model can then be applied. The algorithms based on first order conditions also require these to be specified. The package is based on PyTorch, which is one of the leading deep learning packages, and also allow for using multiple GPUs to solve a single model. Code can be run online through Google Colab.

1.1 Related literature

The literature on solving *dynamic optimization problems* is vast both in economics and other fields such as operations research (see e.g. Stokey and Lucas, 1989; Rust, 1996; Judd, 1998; Bertsekas, 2012a,b; Puterman, 2014). The underlying assumption here is that the model structure, e.g. transition functions, is fully known to the agent facing the dynamic optimization problem, although there might be limited information about the value of current variables and uncertainty about their future value.

The computer science literature on *artificial intelligence* poses a more complex question: How can actions be taken in *real-time* in an *unknown environment*? The focus has therefore been on *reinforcement learning* where the goal is to gradually improve the choice of policy without ever actual finding *the* optimal solution.[3] Following Sutton and Barto (2018, p. 15) dynamic programming can be seen as special case of *reinforcement learning* (see also Bertsekas, 2019; Powell, 2022).[4] Even if decisions must not be made in real time and the environment is fully specified solution techniques from reinforcement learning can still be useful to alleviate or tame the *curse of dimensionality*, where the required solution time increases exponentially with the number of states, choices and shocks.

Standard dynamic programming algorithms are greedy in a double sense. In each iteration, they require a sweep through all of the state space grid, and at each grid point a numerical optimization problem must be solved fully.[5] To alleviate the curse

[3] Computer chess engines have for example by now far surpassed human capabilities. But the optimal behavior for chess players have not yet been found.

[4] Reinforcement learning is typically categorized as a form of *machine learning*. It is distinct from *supervised* machine learning as the goal is not to extrapolate or generalize from desired behavior known for a training set. It is also distinct from *unsupervised* machine learning as the goal is not just to find hidden structure, but actually maximizing some payoff.

[5] This is also the case for sparse-grids algorithms such as those in Brumm and Scheidegger (2017)

of dimensionality with reinforcement learning the fundamental idea is to use less, but better filtered, information in each step and represent value and policy functions in a very flexible manner, which can easily be updated.

A central breakthrough in *artificial intelligence* came with the use of *neural networks* with multiple *hidden layers* and *automatic differentiation* techniques. This is also referred to as *deep neural networks* and *deep learning* (LeCun et al., 2015; Goodfellow et al., 2016). In this paper, all algorithms are deep in the sense of using deep neural networks, and they draw inspiration from reinforcement learning. The combination is referred to as *deep reinforcement learning* (Schmidhuber, 2015; Mnih et al., 2015; Silver et al., 2016; Li, 2018). We in particular draw on the insights and code from Fujimoto et al. (2018) and Lillicrap et al. (2019) for problems with continuous controls.

Avoiding grids, and in particular tensor product grids, are central to alleviate the curse of dimensionality when the number of states are large. Using simulation ensures that the approximation of the policy function is good in the area of the state space where the agents are.

Economics literature. In economics, the literature on solving economic models with deep learning algorithms is growing rapidly. We focus on discrete time methods. Maliar et al. (2021) solve an infinite horizon consumption-saving model and a general equilibrium model with heterogeneous agents using the same three types of methods as us.[6] Generally, most focus has been given to general equilibrium problems, where the single-agent dynamic programming problem is simple, but the problem becomes high-dimensional due to many agents. Azinovic et al. (2022) focus on the system of first order equations approach and solve a general equilibrium model with overlapping generations and thus a life-cycle.[7] Kase et al. (2024) also focus on the equation system approach and solve and estimate a general equilibrium model with infinitely lived agents. Han et al. (2024) (DeepHAM) also solves a general equilibrium model with infinitely lived agents, but uses a version of the simulated utility

and Brumm et al. (2021). Endogenous grid point methods as in Carroll (2006), Druedahl and Jørgensen (2017) and Druedahl (2021) as been successful by minimizing or even removing the need for numerical optimization, but they do not alleviate the increase in computational complexity due to e.g. more states.

[6] See also Maliar and Maliar (2022) for an application with a discrete choice.

[7] Extensions are developed in Azinovic and Zemlicka (2023) and Nuno et al. (2024)

approach with a value function approximation for the terminal simulation period.[8]

Compared to the general equilibrium literature with deep learning this paper instead focuses on using deep learning for solving single-agent problems, which are high-dimensional and complex in themselves. Duarte et al. (2022), building on Sehnke et al. (2010), solves a complex life-cycle using a simulation based approach extended to allow for discrete choices by assuming all choices initially are probabilistic. In this paper, we instead consider multiple algorithms and multiple models and discuss accuracy, stability and speed of each algorithms in detail. Pascal (2024) solves a simple life-cycle with many dimensions due to a multi-dimensional exogenous income process using a equation based method and develops a bias-corrected Monte Carlo operator generalizing the all-in-one expectation operator in Maliar et al. (2021). This is complementary to this paper where, where we assume low dimensional shocks, which can be approximated with quadrature methods, but instead consider many choices and many endogenous states.

A literature on solving economic models with deep learning in continuous time is also developing. Both for single-agent problems (Gopalakrishna, 2021; Huang, 2022; Duarte et al., 2024) and multi-agent problems in general equilibrium (Gu et al., 2023; Huang, 2023; Gopalakrishna et al., 2024; Fan et al., 2024; Payne et al., 2024). It is beyond this paper to discuss the pros and cons of continuous versus discrete time.[9]

2 Notation and General Model Class

We consider finite-horizon models in discrete time indexed by $t \in \{0, 1, \ldots T-1\}$. To begin with, we assume all choices are continuous. In Section 7, we consider both discrete and continuous choices.

[8] The DeepHAM algorithm essentially builds upon the one proposed in Han and E (2016).

[9] The remaining literature on using machine learning to solve economic models is too vast to review here. The use of neural networks in economics goes back to Duffy and McNelis (2001). Valaitis and Villa (2024) use neural networks in a parameterized expectations algorithm. Fernández-Villaverde et al. (2023) and Fernández-Villaverde et al. (2024a) use neural networks for modeling the perceived law of motion. Ebrahimi Kahou et al. (2021) discuss how deep neural networks can exploit symmetry when solving high-dimensional economic models. Scheidegger and Bilionis (2019) compute global solutions to high-dimensional dynamic stochastic economic models on irregular state space geometries using Gaussian processes. Chen et al. (2023) solve a representative agent model with real-time deep reinforcement learning. Fernández-Villaverde et al. (2024b) discuss the potential of the deep learning approach in economics in general.

The terminology is as follows:

1. The *states* are s_t (continuous or discrete).
2. The continuous *actions* (or choices or controls) are $a_t \in \mathcal{A}(s_t)$ with *policy function* $\pi_t(s_t)$.
3. The *utility* (or reward) function is $u_t(s_t, a_t)$.
4. The *stochastic shocks* are z_t and drawn from F_t^z.
5. The *post-decision* states are $\overline{s}_t = \overline{\Gamma}_t\left(s_t, a_t\right)$.[10]
6. The *transition function* is $s_{t+1} = \Gamma_t\left(\overline{s}_t, z_{t+1}\right)$.
7. The *terminal post-decision value function* is $h(\overline{s}_{T-1})$.

The value function for a given policy is defined as:

$$\begin{aligned} v_t(s_t) &= u_t(s_t, a_t) + \begin{cases} \beta h(\overline{s}_t) & \text{if } t = T-1 \\ \beta \overline{v}_t\left(\overline{s}_t\right) & \text{else} \end{cases} \\ a_t &= \pi_t(s_t) \\ \overline{s}_t &= \overline{\Gamma}_t\left(s_t, a_t\right), \end{aligned} \tag{1}$$

where $\overline{v}_t\left(\overline{s}_t\right)$ is the post-decision value function for $t < T-1$

$$\overline{v}_t\left(\overline{s}_t\right) = \mathbb{E}_t\left[v_{t+1}\left(s_{t+1}\right)\right]. \tag{2}$$

The action-values (or values-of-choice) are

$$\begin{aligned} Q_t(s_t, a_t) = u_t(s_t, a_t) + \begin{cases} \beta h(\overline{s}_t) & \text{if } t = T-1 \\ \beta \overline{v}_t\left(\overline{s}_t\right) & \text{else} \end{cases} \\ \overline{s}_t = \overline{\Gamma}_t\left(s_t, a_t\right). \end{aligned} \tag{3}$$

[10] The use of post-decision states is useful for minimizing the number of times expectations must be computed, see e.g. Van Roy et al. (1997); Powell (2011); Bertsekas (2019). For other applications in economics see Hull (2015) and Druedahl (2021). Judd et al. (2017) also discusses how to use pre-computations to limit the cost of computing expectations.

such that the *optimal policy functions* solve

$$\pi^*(s_t) = \arg \max_{a_t \in \mathcal{A}(s_t)} Q_t(s_t, a_t)$$

The *expected discounted life-time* utility is

$$\begin{aligned} R(\pi) &= \mathbb{E}_0 \left[\sum_{t=0}^{T-1} \beta^t u_t(s_t, a_t) \right] \qquad (4) \\ s_0 &\sim F_0^s \\ a_t &= \pi_t(s_t) \\ \bar{s}_t &= \overline{\Gamma}_t(s_t, a_t) \\ z_{t+1} &\sim F_{t+1}^z \\ s_{t+1} &= \Gamma_t(\bar{s}_t, z_{t+1}), \end{aligned}$$

where F_0^s is the distribution of initial states.

Limitations. Our deep learning solutions will rely heavily on automatic differentiation. Therefore we should be able to calculate derivatives with respect to actions of e.g. current utility or the next period states. This does, however, not require that the utility and transition functions are everywhere differentiable. Lee et al. (2020) discuss the formal requirements for applicability of automatic differentiation. They show that assuming functions are almost everywhere differentiable is technically not quite enough, but we have found no examples of economic models where this matters.

3 Simple Test Case: The Buffer-Stock Model

We begin our analysis with a straightforward test case: the buffer-stock consumption-saving model. As a canonical example in economic theory, its relevance is well-established. The low dimensionality of the model allows for rapid solutions using traditional methods, suggesting that deep learning techniques may not offer a competitive advantage in this setting. Nevertheless, we consider this an essential starting point. Successfully solving this model is a prerequisite for tackling more complex models; failure here would imply significant challenges in solving more complicated models.

The *states* for the household are cash-on-hand, $m_t \geq 0$, and permanent income, $p_t \geq 0$. The household chooses consumption c_t. The *utility* function is

$$u_t\left(c_t\right) = \log\left(c_t\right). \tag{5}$$

The income function is

$$y_t(p_{t+1}, \psi_{t+1}) = \begin{cases} \kappa_t & \text{if } t \geq T^{\text{retired}} \\ \kappa_t \psi_{t+1} p_{t+1} & \text{else} \end{cases}, \quad \psi_{t+1} \sim F^{\psi}_{t+1} = \exp \mathcal{N}(-0.5\sigma_{\psi}^2, \sigma_{\psi}^2), \tag{6}$$

where κ_t captures the life-cycle profile and ψ_t is a log-normal transitory shock (with mean one). Next-period permanent income is

$$p_{t+1} = \Gamma^p_t(p_t, \xi_{t+1}) = p_t^{\rho_p} \xi_{t+1}, \;\; \xi_{t+1} \sim F^{\xi}_{t+1} = \exp \mathcal{N}(-0.5\sigma_{\xi}^2, \sigma_{\xi}^2), \tag{7}$$

where $\rho_p \in (0, 1]$ is the auto-regressive parameter and ξ_t is a log-normal permanent shock (with mean one). The savings rate is $a_t = 1 - \frac{c_t}{m_t}$. End of period savings is then $\overline{m}_t = \overline{\Gamma}^m_t(s_t, a_t) = a_t m_t$ Next-period cash-on-hand is

$$m_{t+1} = \Gamma^m_t(\overline{m}_t, p_{t+1}, \psi_{t+1}) = (1 + r)\bar{m}_t + y_t(p_{t+1}, \psi_{t+1}), \tag{8}$$

where $R > 0$ is the return factor.

The Bellman equation for this problem is:

$$\begin{aligned} v_t(m_t, p_t) &= \max_{c_t} u_t(c_t) + \beta \mathbb{E}_t \left[v_{t+1}(m_{t+1}, p_{t+1}) \right] && (9) \\ \text{s.t.} & && (10) \\ m_{t+1} &= \Gamma^m_t(\bar{m}_t, p_{t+1}, \psi_{t+1}) \\ p_{t+1} &= \Gamma^p_t(p_t, \xi_{t+1}) \\ \overline{m}_t &= a_t m_t \\ a_t &= 1 - \frac{c_t}{m_t} \\ \overline{m}_t &\geq 0 && (11) \end{aligned}$$

The problem can be formulated using post-decisions states and post-decision value function. The post decision states are end-of-period savings given by $\overline{m}_t = a_t m_t \geq 0$ and permanent income which is unaffected by the decision and simply remains p_t. The post-decision value function is

$$\begin{aligned}
\overline{v}_t(b_t, p_t) = \mathbb{E}_t \left[v_{t+1}(m_{t+1}, p_{t+1}) \right] \qquad (12)\\
m_{t+1} &= \Gamma_{m,t}(\bar{m}_t, p_{t+1}, \psi_{t+1})\\
p_{t+1} &= \Gamma_{p,t}(p_t, \xi_{t+1}).
\end{aligned}$$

The value function can then be formulated as

$$\begin{aligned}
v_t(m_t, p_t) &= \max_{c_t} u_t(c_t) + \beta \overline{v}_t(\bar{m}_t, p_t) \qquad &(13)\\
&s.t. &(14)\\
\bar{m}_t &= a_t m_t\\
a_t &= 1 - \frac{c_t}{m_t}\\
b_t &\geq 0. &(15)
\end{aligned}$$

For the initial states, we have

$$m_0 \sim F_0^m = \exp \mathcal{N}(-0.5\sigma_{m0}^2 + \log \mu_{m0}, \sigma_{m0}^2). \qquad (16)$$

$$p_0 \sim F_0^p = \exp \mathcal{N}(-0.5\sigma_{p0}^2 + \log \mu_{p0}, \sigma_{p0}^2). \qquad (17)$$

The calibration is given in Appendix C.

Extended model We also consider an extended version of the model with additional fixed states. Specifically, we modify this model to incorporate heterogeneous income risk, introducing heterogeneity in three parameters: $\sigma_{\psi,i}$, $\sigma_{\xi,i}$. and ρ_p These parameters are now part of the state space, resulting in a five-dimensional state space.

For the simulations, we draw the values of these parameters from uniform distributions as follows:

$$\sigma_{\psi,i} \sim F_{\psi} = U(\sigma_{\psi,low}, \sigma_{\psi,high}) \tag{18}$$

$$\sigma_{\xi,i} \sim F_{\xi} = U(\sigma_{\xi,low}, \sigma_{\xi,high}) \tag{19}$$

$$\rho_{p,i} \sim F_{\rho_p} = U(\rho_{p,low}, \rho_{p,high}). \tag{20}$$

4 Deep Learning Algorithms

In this section, we introduce three deep learning (DL) algorithms for solving finite horizon models: DeepSimuate, DeepFOC and DeepV. *DeepSimuate* is based purely on simulation, *DeepFOC* is based on an equation system of first order equations, and *DeepV* is value-function based. While we use the buffer-stock model from Section 3 as our primary example, the notation and discussion is consistent with the broad class of finite-horizon models from Section 2.

The algorithms we consider differ from traditional dynamic programming (DP) algorithms in two key ways:

1. Value and policy functions are approximated as deep neural networks trained on simulated training samples instead of being defined on a grid.

2. Policy functions are found globally by minimizing an appropriate loss function instead of using a local numerical optimizer for each grid point.

Neural networks are used for their strong function approximation capabilities exploiting symmetries and hidden patterns, and fast training using existing software. Avoiding grids, and in particular tensor product grids[11], are central to alleviate the curse of dimensionality when the number of states and choices are large. Avoiding calling a local numerical optimizer (or equation-solver) removes the central computational bottleneck in many standard algorithms.[12] A final computational bottleneck is the computation of expected (i.e. continuation) values by numerical integration.

[11] See Judd et al. (2014), Brumm and Scheidegger (2017) and Brumm et al. (2021) for applications of (adaptive) sparse grids techniques in economics.

[12] This is also achieved by endogenous grid based methods building on Carroll (2006). See e.g. Maliar and Maliar (2013), Druedahl and Jørgensen (2017) and Druedahl (2021).

The algorithms considered below can use both quadrature (which is efficient with a smaller number of shocks) and Monte Carlo (which is efficient with a larger number of shocks).

Appendix B contains an introduction to the theory of neural networks, and how to work with them efficiently using PyTorch in Python. Here, it suffices to note that neural networks are a mapping from a set of inputs to a set of outputs, which using so-called *final activation functions* can be enforced to lie in certain output range (e.g. $\in [0,1]$ or > 0). Theoretically a number of *universal function approximation theorems* (Hornik et al., 1989) exist. In practice it is central that neural networks allow for fast automatic differentiation of the outputs w.r.t. the parameters (so-called weights and biases) given the inputs. This is also called *backpropagation* or *reverse mode automatic differentiation*.

Policy functions. In all the algorithms the policy function π is approximated with a neural network with parameters, θ_π,

$$\pi_t\left(s_t\right) \approx \pi\left(t, s_t; \theta_\pi\right). \tag{21}$$

where t and the state-vector are the inputs, and the output is the vector of actions. We refer to this as the *policy network*. When the value function is needed it will be approximated in a similar way as a *value network*.

Simulation. We assume that initial states are drawn from the distribution F_0^s, and the shocks are drawn from F_t^z each period. A simulation sample induced by policy π with parameters θ_π with N individuals, indexed by $i \in \{0, \ldots N-1\}$, over T periods, indexed by $t \in \{0, \ldots, T-1\}$, is denoted,

$$\mathcal{S}\left(\theta_\pi\right) = \left\{s_{t,i}, \bar{s}_{t,i}, a_{t,i}, z_{t,i}, \; \forall t, i\right\} \tag{22}$$

and can be found by simulation as

$$\begin{aligned}
s_{0,i} &\sim F_0^s \\
a_{t,i} &= \pi(t, s_{t,i}; \theta_\pi) \\
\bar{s}_{t,i} &= \overline{\Gamma}_t(s_{t,i}, a_{t,i}) \\
\text{if } t < T-1:\ s_{t+1,i} &= \Gamma_t\left(\bar{s}_{t,i}, z_{t+1,i}\right),\ \ z_{t+1,i} \sim F_{t+1}^z.
\end{aligned}$$

The expected discounted utility, or expected reward, induced by policy π with parameters θ_π is

$$\begin{aligned}
R\left(\theta_\pi\right) = \mathbb{E}_0 &\left[\sum_{t=0}^{T-1} \beta^t u_t(s_t, a_t)\right] \qquad (23) \\
s_0 &\sim F_0^s \\
a_t &= \pi(t, s_t; \theta_\pi) \\
\bar{s}_t &= \overline{\Gamma}_t\left(s_t, a_t\right) \\
z_{t+1} &\sim F_{t+1}^z \\
s_{t+1} &= \Gamma_t\left(\bar{s}_t, z_{t+1}\right).
\end{aligned}$$

Denoting the training sample of initial states and shocks by $\underline{\mathcal{S}} = \{s_{0,i}, z_{1,i}, \ldots, z_{T,i}, \forall i\}$, the sample equivalent of the expected reward is

$$\begin{aligned}
R\left(\theta_\pi; \underline{\mathcal{S}}\right) = &\left[\frac{1}{N}\sum_{i=0}^{N-1}\sum_{t=0}^{T-1} \beta^t u_t(\tilde{s}_{t,i}, \tilde{a}_{t,i})\right] \qquad (24) \\
\tilde{s}_{0,i} &= s_{0,i} \\
\tilde{a}_{t,i} &= \pi(t, \tilde{s}_{t,i}; \theta_\pi) \\
\tilde{\bar{s}}_{t,i} &= \overline{\Gamma}_t\left(\tilde{s}_{t,i}, \tilde{a}_{t,i}\right) \\
\tilde{s}_{t+1,i} &= \Gamma_t\left(\tilde{\bar{s}}_{t,i}, z_{t+1,i}\right),
\end{aligned}$$

where $\tilde{x}$ denotes x is computed within the simulation and is not an input from the training sample of initial states and shocks.

Next. We now turn to explaining each of the algorithms in turn. This fundamentally boils down to specifying an approach to minimizing some *loss function* to update the parameters in the policy network, θ_π.

4.1 DeepSimuate

The simplest and most direct approach is *DeepSimuate*, which is only based on simulation. In each iteration a training sample of initial states and shocks are drawn, and the success of the current guess of the policy network is evaluated in terms of the implied expected reward.

DeepSimuate is described in detail in Algorithm 1.

The main part of the algorithm is the loss function in equation (25). This is just the negative of the expected reward. Evaluating the loss function is straightforward, but keeping track of the derivative $\partial L/\partial\theta_\pi$ is complicated. Mathematically it is,

$$\frac{\partial L}{\partial\theta_\pi} = -\frac{\partial a_0}{\partial\theta_\pi}\left(\frac{\partial u_0}{\partial a_0} + \beta\frac{\partial s_1}{\partial a_0}\left(\frac{\partial u_1}{\partial s_1} + \frac{\partial a_1}{\partial s_1}\frac{\partial u_1}{\partial a_1}\right) + \beta^2\ldots\right) - \frac{\partial a_1}{\partial\theta_\pi}\left(\ldots\right) - \ldots$$

Each parameter in θ_π has a direct impact on the actions in each period and therefore also the utility. Indirectly it also impacts utility in future periods by affecting the states in future periods. Despite this complexity modern automatic differentiation can efficiently keep track of the derivative.

We use the loss function to make a single update of the parameters in the policy network. This is referred to as an epoch. In principle, we could make multiple updates, but because it is cheap to draw a new training sample containing fully unused information, a single epoch is efficient.

Every Δ_R iterations we compute an out-of-sample expected reward using the large fixed sample $\underline{\mathcal{S}}^*$. This is useful for inspecting the progress of the solution algorithm, but also costly which is why we set $\Delta_R \gg 1$. In practice, we either run the deep learning algorithms for a fixed number of iterations or a fixed number of minutes. We introduce a formal convergence criterion below.

The main benefit of the algorithm is that the use of Monte Carlo implies $\partial L_\pi/\partial\theta_\pi$ is estimated unbiased and quite precisely even at the beginning of training. This however requires us to use large enough training samples to dampen the variance from the sampling. In practice $N = 5,000$ seems enough. Updating is done with some form of gradient descent as discussed below.

Another advantage of *DeepSimuate* is that it has a fairly limited set of hyperparameters, which makes it easier to do hyperparameter optimization. The most important

Algorithm 1 DeepSimuate

```
Draw fixed sample of initial states and shocks, $\underline{\mathcal{S}}^*$.
Draw random initial policy parameter values, $\theta_\pi^{-1}$.
For $K$ iterations indexed by $k = 0, 1, \dots, K-1$ do:
    1. Draw a random training sample, $\underline{\mathcal{S}}^k$, of initial states and shocks.
    2. Update policy network to $\theta_\pi^k$ for one epoch using the negative expected reward
       as the loss-function
           $L_\pi(\theta_\pi; \underline{\mathcal{S}}^k) = -R\left(\theta_\pi; \underline{\mathcal{S}}\right).$    (25)
    3. If $k$ is divisible by $\Delta_R$ calculate current out-of-sample expected reward, $R^k =$
       $R\left(\theta_\pi^k, \underline{\mathcal{S}}^*\right)$
```

hyperparameters are the learning rate for θ_π in step 2 and how many agents we simulate N. We use a fixed learning rate of 10^{-3}. The parameters in the initial policy neural network are drawn randomly following the PyTorch default of uniform draws around 0.[13]

One disadvantage of *DeepSimuate* is that the algorithm runs »on policy«, and adding noise can thus *not* easily be used for escaping local minima (see further discussion below). Any information in first order conditions can also not be utilized. Finally, the algorithm cannot be used for models with discrete choices. Even if discrete choice is modeled as selecting a probability, it fails because the derivative vanishes in the Monte Carlo simulation, where no agents are precisely on the boundary between two outcomes.[14]

Literature The *DeepSimuate* algorithm is related to traditional reinforcement learning methods such as REINFORCE (see Sutton and Barto (2018)). A key difference is that in those approaches it is assumed that the transition functions are not known, and thus can not be differentiated. Then it is only the sampled returns, which are

[13] The PyTorch default is to draw parameters uniformly in the interval $\left(-\frac{1}{\sqrt{N^{input}}}, \frac{1}{\sqrt{N^{input}}}\right)$ where N^{input} is the dimension of the input-vector.

[14] In infinite horizon models $T \to \infty$ an additional problem is truncation of the simulation. In this case Han et al. (2024) introduce a terminal value function.

used for updating the policy parameters.

Duarte et al. (2022) employs an algorithm similar to the idea here, but uses exploration in parameter space inspired by Sehnke et al. (2010). This entails specifying the policies as stochastic in way such that analytical derivatives partly exist, and then anneal toward the deterministic solution. This broadens the scope of the algorithm to models with discrete choices. In our experience it has been hard to implement a stable version of such an algorithm with a high likelihood of convergence for the models we consider.

4.2 Beyond simulation

The next two algorithms, *DeepFOC* and *DeepV*, are more involved than *DeepSimuate*. They do, however, share the generic form given in Algorithm 2, which uses the same termination criterion as *DeepSimuate*. This generic algorithm introduces two new concepts from the reinforcement learning literature:

1. **Explorative policy function** $\breve{\pi}(t, s_t, \epsilon_t; \theta_\pi)$ with exploration shocks ϵ_t. In the buffer-stock model we use

$$\begin{aligned} \breve{\pi}(t, s_t, \epsilon_t; \theta_{\pi,t}) &= \min\left\{\max\left\{\pi\left(t, s_t; \theta_\pi\right) + \epsilon_t, 0\right\}, 1\right\} \\ \epsilon_t &\sim F_\epsilon\left(\sigma_\epsilon\right) = \mathcal{N}(0, \sigma_\epsilon), \end{aligned} \tag{26}$$

   such that we ensure that the savings-rate remains bounded in $[0;1]$. The benefit of adding noise both stems from introducing a smoothing element early on in the training when there can be erratic changes in the policy, and from helping the model escape degenerate policy functions, where e.g. no one saves, and other local minima. Another advantage is that it enhances the precision of the solution at the edges of the simulated area, which, in the buffer-stock model, translates to more accurate solutions for the wealthiest individuals. However, a drawback is that for a fixed number of points, increased exploration reduces density in the critical regions of the state space. We allow the scale parameter for the exploration shocks to differ across iterations, σ_ϵ^k, as exploration is more likely to be useful early on.

2. **Replay buffer**: The sample simulated each iteration is saved in a replay buffer, and we the draw a random training batch from this replay buffer to train

the neural networks on. Initially the replay buffer is just the training sample $\mathcal{B} = \mathcal{S}^0$. The buffer is updated by replacing some existing elements with new elements from the newly simulated sample $\mathcal{S}^k$. The buffer's memory determines how many iterations of simulated samples it can hold, operating on a first-in, first-out principle.[15] The replay buffer reduces algorithm volatility by making the algorithm less reliant on the latest sample, though at a cost of speed since the latest simulation is closer to the model implied true distribution. Mnih et al. (2015) argues that using a replay buffer reduces the correlation of observations within training samples which reduces variance of parameter updates.

The specific algorithms differ by how the policy parameters are updated in step 3 using a random training batch from the replay buffer.

4.3 DeepFOC

We will now turn to using first-order conditions for finding θ_π. This approach is the dominant approach in the literature on using deep learning for solving economic models. We will use Azinovic et al. (2022) and Maliar et al. (2021) as the main references here. From the buffer-stock model we can derive the Euler-equation as

$$u_t'(c_t) \geq \beta R \mathbb{E}_t[u_{t+1}'(c_{t+1})]. \tag{27}$$

The Euler-equation holds with $=$ for $c_t \in [0, m_t)$. For now, we assume that the policy network is constructed such that only feasible choices, $c_t \in [0, m_t]$, are made. We discuss this further below.

Fischer-Burmeister (FB). Maliar et al. (2021) suggests encoding the inequality constraint as an equality constraint by using a Fischer-Burmeister-function:

$$f_{FB}(x, y) = \sqrt{x^2 + y^2} - (x + y). \tag{28}$$

[15] We sample from the replay buffer across agents such that if we sample a given agent, we sample all time-periods for that agent. Replay buffers have been used successfully in many reinforcement learning papers, see Mnih et al. (2015) and Lillicrap et al. (2019) for some applications. We rely on the implementation in Fujimoto et al. (2018).

Algorithm 2 Generic solution algorithm

```
Draw sample of fixed initial states and shocks, $\underline{\mathcal{S}}^*$.
Draw random initial parameter values, $\theta_\pi^{-1}$.
Initialize buffer memory $\mathcal{B} = \emptyset$
For a maximum of $K$ iterations indexed by $k = 0, 1, \ldots, K-1$ do:
    1. Simulate random training sample with exploration $\mathcal{S}^k$ by
           $s_{0,i} \sim F_0$
           $a_{t,i} = \breve{\pi}(t, s_{t,i}, \epsilon_{t,i}; \theta_\pi^{k-1})$
           $\epsilon_{t,i} \sim F_\epsilon\left(\sigma_k\right)$
           $\bar{s}_{t,i} = \bar{\Gamma}_t(s_{t,i}, a_{t,i})$
           if $t < T$ : $s_{t+1,i} = \Gamma_t\left(\bar{s}_{t,i}, z_{t+1,i}\right)$,  $z_{t+1,i} \sim F_{t+1}$,
       for all $i \in \mathcal{N}$ and $t \in \mathcal{T}$.
    2. Update replay buffer $\mathcal{B}$ with $\mathcal{S}^k$ (first in, first out).
    3. Call DeepFOC or DeepV: Update policy network to $\theta_\pi^k$ using random training
       batch $\mathcal{B}^k$ from $\mathcal{B}$
    4. If $k$ is divisible by $\Delta_R$ calculate out-of-sample expected reward, $R^k =
       R\left(\theta_\pi^k, \underline{\mathcal{S}}^*\right)$.
```

This function is zero when one input is zero and the other is non-negative, and otherwise it is positive,

$$f_{FB}(x,0) \begin{cases} = 0 & \text{if } x \geq 0 \\ > 0 & \text{else} \end{cases}$$
$$f_{FB}(0,y) \begin{cases} = 0 & \text{if } y \geq 0 \\ > 0 & \text{else} \end{cases}.$$

Let us use the buffer-stock case as an example. The household in that model is constrained when

$$c_t = m_t \text{ and } u_t'(c_t) > \beta(1+r)\mathbb{E}_t\left[u_{t+1}'(c_{t+1})\right],$$

or *unconstrained* when

$$c_t < m_t \text{ and } u_t'(c_t) = \beta(1+r)\mathbb{E}_t\left[u_{t+1}'(c_{t+1})\right].$$

If we check whether an agent is constrained by computing $m_t - c_t$, then the equation we are trying to solve is:

$$f_{FB}\left(u_t'(c_t) - \beta(1+r)\mathbb{E}\left[u_{t+1}'(c_{t+1})\right], m_t - c_t\right) = 0. \tag{29}$$

Karush–Kuhn–Tucker (KKT). An alternative approach is used in Azinovic et al. (2022). They instead solve for the KKT-conditions explicitly. The result is two conditions. Firstly, the Euler-equation which now includes a KKT-multiplier

$$u_t'(c_t) = \beta(1+r)\mathbb{E}_t\left[u_{t+1}'(c_{t+1})\right] + \lambda_t. \tag{30}$$

Secondly, a complementary slackness constraint:

$$\lambda_t(m_t - c_t) = 0. \tag{31}$$

There is also an inequality constraint on $\lambda_t \geq 0$ (dual feasibility). We ensure this is always fulfilled by considering the multiplier λ_t to be an extra output from the policy network π similar to an additional action, and use a *soft*plus final activation function, $\log(1+\exp(x))$, to get $\lambda_t \geq 0$.

We now have an additional equation to solve although each equation is individually simpler than the FB-equation that combines these equations. As we have two equations to solve in the buffer-stock model we are minimizing

$$\kappa_{\text{FOC}}\frac{1}{(T-1)N}\sum_{t=0}^{T-2}\sum_{i=0}^{N-1}(\text{FOC})^2+\kappa_{\text{slack}}\frac{1}{(T-1)N}\sum_{t=0}^{T-2}\sum_{i=0}^{N-1}(\text{slackness})^2$$

where $\kappa_{\text{FOC}} > 0$ and $\kappa_{\text{slack}} > 0$ are exogenous weights and

$$\begin{aligned}\text{FOC} &= u_t'(c_{t,i}) - \beta(1+r)\mathbb{E}_t\left[u_{t+1}'(c_{t+1,i})\right] - \lambda_{t,i}\\ \text{slackness} &= \lambda_{t,i}(m_{t,i}-c_{t,i}),\end{aligned}$$

Note that we have no condition in $T-1$ since consuming everything, $c_t = m_t$, will always be optimal in the last period. In other models there might be an active choice in the final period.

Loss function Regardless of the approach, we can specify a general algorithm by stating we want to solve some series of equations $g_{t,l} \in \{g_{t,0}, g_{t,1}, \ldots, g_{t,G-1}\}$ such that $g_{t,0} = g_{t,1} = \ldots g_{t,G-1} = 0$ in each time period t where G is the number of equations. These equations will potentially depend on the current and next-period states as well as the current and next-period policy:

$$g_{t,l}(s_{t,i}, \pi(t, s_{t,i}; \theta_\pi), \tilde{s}_{t+1,i}, \pi(t+1, \tilde{s}_{t+1,i}; \theta_\pi))$$

except for the last period where only the current state and choice matters,

$$g_{T-1,l}\left(s_{T-1,i}, \pi(T-1, s_{T-1,i}; \theta_\pi)\right)$$

. If using the KKT approach the multipliers are considered an output of the network. In the buffer-stock model with the Fischer-Burmeister approach $G = 1$ and we only have

$$g_{t,0}\left(s_t, \pi(s_t), s_{t+1}, \pi(s_{t+1})\right) = f_{FB}\left(u_t'(c_t) - \beta(1+r)\mathbb{E}[u_{t+1}'(\tilde{c}_{t+1})], m_t - c_t\right).$$

This requires us to compute an expectation. This can be done with either quadrature or Monte Carlo methods. We consider models with a small number of shocks, where

Algorithm 3 DeepFOC

Algorithm 2 with the updating in step 3 as below:

1. Update policy network to θ_π^k using the loss function

$$L_\pi\left(\theta_\pi; \mathcal{B}^k\right) = \sum_{l=0}^{G} \frac{1}{\left|\mathcal{B}^k\right|} \sum_{i,t,s_{t,i}\in\mathcal{B}^k} \kappa_l \begin{cases} g_{T-1,l}^2 & \text{if } t = T-2 \\ g_{t,l}^2 & \text{else} \end{cases} \tag{32}$$

$$g_{T-1,l} = g_{T-1,l}\left(s_{T-1,i}, \pi(T-1, s_{T-1,i}; \theta_\pi)\right)$$

$$g_{t,l} = g_{t,l}\left(s_{t,i}, \pi(t, s_{t,i}; \theta_\pi), \tilde{s}_{t+1,i}, \pi(t+1, \tilde{s}_{t+1,i}; \theta_\pi)\right) \tag{33}$$

$$\tilde{a}_{t,i} = \pi(t, s_{t,i}; \theta_\pi) \tag{34}$$

$$\tilde{\tilde{s}}_{t,i} = \overline{\Gamma}_t(t, s_{t,i}; \theta_\pi)$$

$$\tilde{s}_{t+1,i} = \Gamma_t\left(\tilde{\tilde{s}}_{t,i}, \tilde{z}_{t+1,i}\right), \;\; \tilde{z}_{it+1} \sim F_{t+1}^z.$$

   for up to $\#_\pi$ epochs, or until the policy loss as not improved for Δ_π epochs.

2. Return θ_π^k from the epoch with the smallest policy loss.

quadrature is efficient. See Pascal (2024) for a bias-corrected Monte Carlo operator, building on the insights in Maliar et al. (2021), which is useful when there are many shocks.

We can now state DeepFOC in Algorithm 3, where we weight the equation errors with the non-negative weigth $\kappa_l \geq 0$. We allow for multiple epochs, where i) the data is passed through the loss-function, ii) derivatives are calculated and iii) the parameters are updated using some form of gradient descent, but terminate early if the loss is not becoming smaller.

Compared to the other algorithms, DeepFOC relies significantly more on human input in deriving the equation system. Central advantages are, however, that only a single neural network is used and model specific information from the first order conditions are utilized. Furthermore, the policy loss is typically rather straightforward to interpret in terms of how precise the achieved solution is. This is harder with DeepSimuate or with the value-based approach introduced below.

Computationally a central disadvantage is that to find $\partial L/\partial\theta_\pi$ requires applying the expectation operator. Keep tracking of the derivatives inside the expectations operator and then summing up with the appropriate weights is costly. For a given time budget DeepFOC will then be able to make fewer iterations.

A final disadvantage for DeepFOC is that it relies on the first order conditions being not only *necessary*, but also *sufficient*. This is e.g. rarely the case with discrete choices. In general, DeepFOC is the most limited of the algorithms we use in this paper. While DeepSimuate also cannot handle discrete choices it can in principle handle cases where FOCs are only necessary but not sufficient.

4.4 DeepVPD

We now turn to the value-function based algorithm *DeepV*. This is the most versatile algorithm, but also the most complicated. As our baseline *DeepV* algorithm we choose to approximate the post-decision value function instead of the standard pre-decision value function. To be precise, we call this sub-algorithm *DeepVPD,* where PD is for post-decision. Approximating the post-decision value function has a number of benefits. The post-decision value function is firstly generally more smooth than the value function itself, and secondly it yields a simpler loss function for the policy network, where there is no longer a need to keep track of derivatives inside an expectation operator.

Specifically, we introduce the post-decision value network

$$\overline{v}_t(\overline{s}_t) \approx \overline{v}(t, \overline{s}_t, \theta_{\overline{v}}). \tag{35}$$

To update the parameter in this value network, we need a *target*. We call the value and policy networks used to compute the target for the *target networks* and denote their parameters by $\check{\theta}_\pi$ and $\check{\theta}_{\overline{v}}$. The value network must then be a good approximation of

$$\mathbb{E}_t[v_{t+1}(s_{t+1})] = \mathbb{E}_t \left[u_{t+1}(s_{t+1}, \pi(t+1, s_{t+1}, \check{\theta}_\pi)) + \begin{cases} h(\overline{s}_{T-1}) & \text{for } t+1 = T-1 \\ \beta \overline{v}(t+1, \overline{s}_{t+1}, \check{\theta}_{\overline{v}}) & \text{else} \end{cases} \right],$$

subject to the relevant transition functions and where $h(\overline{s}_{T-1})$ is the terminal post-decision value function, which in the buffer-stock model is just zero. The expectation on the right hand side is again calculated with some form of numerical integration as in *DeepFOC*.

The parameters in the policy network are updated to maximize the value-of-choice, like in *DeepSimuate*, but which can now be stated in terms of the post-decision value function as

$$u_t(s_t, a_t) + \begin{cases} h(\bar{s}_{T-1}) & \text{for } t+1 = T-1 \\ \beta \bar{v}(t, \bar{s}_t, \theta_{\bar{v}}) & \text{else} \end{cases}$$

The detailed formulations of the algorithm is given in Algorithm 4. In step 1, we compute the value target using the policy and value target networks with parameters $\check{\theta}_{\pi}^{k-1}$ and $\check{\theta}_{\bar{v}}^{k-1}$, where k denote the current iteration. Step 2 is then a standard function approximation problem. Since we have $\partial \tilde{\bar{v}}_{t,i} / \partial \theta_{\bar{v}} = 0$ we do not need to take derivatives inside the expectation operator in the value update, which would have been a significant computational cost. At the end of step 2 the target value parameters are updated as an average of the old target parameters and the new parameters $\theta_{\bar{v}}$ with a weight of $\tau \in (0, 1]$. The use of target parameters helps with avoiding the fundamental »moving target problem« that arises with such an algorithm.[16] If $\tau = 1$ all weight is put on the new parameters. We set $\tau = 0.20$.

In step 3 we turn to the update of policy parameters, and again make a smoothed update of the policy target parameters. We skip this step for the first $\underline{k}_{\pi} > 0$ iterations. This is useful as the value function approximation initially might be too imprecise for policy optimization to meaningful. When simulating the training sample, actions are then either implied by the randomly initialized policy network, or by directly drawing exogenous stochastic actions. Note that $\tilde{\bar{v}}$ is not considered fixed in this step. By using a post-decision value function we avoid taking expectations inside the policy update resulting in a significant speedup. Taking expectations inside the policy update is unavoidable if you approximate the value function itself rather than the post-decision value function as shown in Appendix A.

Compared to standard dynamic programming solution methods for finite-horizon models it should be noted that we solve simultaneously across all periods and not sequentially backwards. This effectively utilizes the symmetry of solutions across periods.[17]

Relative to *DeepFOC* the main disadvantage is that information from the first order

[16] This modification has been used successfully in the reinforcement learning literature such as Mnih et al. (2015)

[17] Introducing period-specific neural networks would be computationally costly. An intermediate approach could be to have a single neural network, but still update it through backward propagation. A danger then, however, is that the networks are then always over-fitted to the current period.

Algorithm 4 DeepVPD

```
Draw random initial parameter values, $\theta_{\overline{v}}^{-1}$.
Initialize $\check{\theta}_{\pi}^{-1} = \theta_{\pi}^{-1}$ and $\check{\theta}_{\overline{v}}^{-1} = \theta_{\overline{v}}^{-1}$.
Algorithm 2 with the updating in step 3 as:
  1. Compute the value target for $t < T-1$

     $\tilde{\overline{v}}_{t,i} = \mathbb{E}_t \left[ u_{t+1}\left(\tilde{s}_{t+1,i}, \tilde{a}_{t+1,i}\right) + \begin{cases} h(\tilde{\overline{s}}_{t+1,i}) & \text{if } t = T-2 \\ \overline{v}(t+1, \tilde{\overline{s}}_{t+1,i}; \check{\theta}_{\overline{v}}^{k-1}) & \text{else} \end{cases} \right]$

     where

     $\tilde{s}_{t+1,i} = \Gamma_t(\overline{s}_{t,i}, \tilde{z}_{t+1,i}), \;\; \tilde{z}_{t+1,i} \sim F_{t+1}^{z}$
     $\tilde{a}_{t+1,i} = \pi(t+1, \tilde{s}_{t+1,i}; \check{\theta}_{\pi}^{k-1})$
     $\tilde{\overline{s}}_{t+1,i} = \overline{\Gamma}_{t+1}(\tilde{s}_{t+1,i}, \tilde{a}_{t+1,i})$

  2. Update the value network to $\theta_{\overline{v}}^{k}$ using the loss function for $t < T-1$

     $L_{\overline{v}}\left(\theta_{\overline{v}}; \mathcal{B}^k\right) = \frac{1}{|\mathcal{B}^k|} \sum_{i,t,\overline{s}_{it} \in \mathcal{B}^k} \left(\overline{v}\left(t, \overline{s}_{t,i}; \theta_{\overline{v}}\right) - \tilde{\overline{v}}_{t,i}\right)^2 \qquad (36)$

     for up to $\#_{\overline{v}}$ epochs, or until the value loss has not improved for $\Delta_{\overline{v}}$ epochs.
     (a) Return $\theta_{\overline{v}}^{k}$ from the epoch with the lowest loss.
     (b) Update value target network: $\check{\theta}_{\overline{v}}^{k} = \tau\theta_{\overline{v}}^{k} + (1-\tau)\check{\theta}_{\overline{v}}^{k-1}$.

  3. If $k > \underline{k}_{\pi}$ : Update the policy network to $\theta_{\pi}^{k}$

     $L_{\pi}(\theta_{\pi}; \mathcal{B}^k) = -\frac{1}{|\mathcal{B}^k|} \sum_{i,t,s_{t,i} \in \mathcal{B}^k} \left(u_t(s_{t,i}, \tilde{a}_{t,i}) + \tilde{\overline{v}}_{t,i}\right) \qquad (37)$
     $\tilde{a}_{t,i} = \pi(t, s_{t,i}; \theta_{\pi})$
     $\tilde{\overline{s}}_{t,i} = \overline{\Gamma}_t(s_{t,i}, \tilde{a}_{t,i})$.
     $\tilde{\overline{v}}_{t,i} = \begin{cases} h(\tilde{\overline{s}}_{t,i}) & \text{if } t = T-1 \\ \beta\overline{v}\left(t, \tilde{\overline{s}}_{t,i}; \theta_{\overline{v}}^{k}\right) & \text{else} \end{cases}$

     for up to $\#_{\pi}$ epochs, or until the policy loss as not improved for $\Delta_{\pi}$ epochs.
     (a) Return $\theta_{\pi}^{k+1}$ from the epoch with the lowest loss.
     (b) Update target policy network $\check{\theta}_{\pi}^{k} = \tau\theta_{\pi}^{k} + (1-\tau)\check{\theta}_{\pi}^{k-1}$.
```

conditions are not used. On the other hand it is plug-n'-play like *DeepSimuate* requiring no mathematical derivation beyond specifying the Bellman-equation. An interesting advantage of the *DeepV* algorithms are that it can also be used with discrete choices. We return to this in Section 7.

FOC. One approach to incorporate information from the first order conditions in *DeepVPD* is to assume the value network has additional outputs in the form of derivatives. In particular for the buffer-stock model, we need a second output of the value network to approximate in step 1,

$$\frac{\partial \overline{v}_t(\overline{s}_t)}{\partial \overline{m}_t} = \mathbb{E}_t \left[u'_{t+1}(s_{t+1}, \pi(t+1, s_{t+1}, \theta_\pi) \right]$$

which it is straightforward to incorporate in the loss function in equation (36). In step 2, the policy loss function in equation (37) must be extended with a Fischer-Burmeiser term checking whether the FOC is fulfilled

$$f_{FB}\left(u'_t(c_t) - \beta R \frac{\overline{v}_t(\overline{s}_t)}{\partial b_t}, m_t - c_t \right)$$

Alternatively, a KKT approach can be used instead. In larger models this might be easier. Here the policy network is assumed to output both choices and multipliers, and the loss function for the value function incorporate approximating all relevant derivatives. In the policy step a term with FOC-errors and the slackness constraint is introduced. Details are in Appendix A.

Averaging. Our experience tell us that one limitation of the *DeepVPD* algorithm is that the value function does not become smooth enough, but continue to have small wiggles late in the training. An extension of the algorithm handles this by introducing multiple value networks with different random initializations.[18] To train the policy network an average of the multiple value networks is used, which averages out the wiggles. This can be used simultaneously with adding the FOCs to the loss function. Details are in Appendix A.

[18] This is called ensemble learning in the computer science literature.

DeepV and DeepQ. Appendix A specifies how the algorithm would look like if we approximated the pre-decision instead of the post-decision value function. It also discusses how our algorithm compares to the canonical actor-critique algorithms, which we refer to as *DeepQ*.

4.5 Implementation

We now turn to discussing our main implementation choices. The full details are given in Appendix B.

4.5.1 Choice of actions

In the buffer-stock model, we chose the action to be the savings-rate instead of consumption itself. This section discusses that choice.

When using $a_t = c_t$ we would typically employ a non-negative final activation function such as the *softplus* or the *ReLU*. An issue with using c_t is that the policy can predict consumption that violates the borrowing constraint during training. This is easy to handle in *DeepFOC* as the borrowing constraint is handled explicitly, but in other approaches this will require using penalty functions.

Letting the action be the savings savings rate such that $c_t = (1 - a_t)m_t$ has one clear advantage: We can use a *sigmoid* final activation function to bound the predictions between 0 and 1 resulting in no violations of the borrowing constraint. The disadvantage is that borrowing-constrained agents will choose $a_t = 0$. At $a_t = 0$ the derivative of the *sigmoid* is also zero which slows down training and during early training you can risk getting caught somewhere where $a_t \approx 0$ everywhere in which case it will take a long time for the parameters to adjust away from that region of the parameter-space. This implies an advantage of using $a_t = c_t$: Agents never want to choose $c_t = 0$ and thus we never have the problem of evaluating the policy in a region where the gradient is zero.

We found it easier to use the savings-rate approach, but both are valid.

4.5.2 Time and input transform

A central question is how to include time in the state-space. The straightforward approach would be to include t as a value directly as an input. This is cheap in terms of the number of network parameters, but in practice we have found it hard to use. Instead we include dummies for each time period in the networks. This lets the optimizer choose parameters, which create differences in policy across periods. For the policy network e.g.

$$\begin{aligned} \pi_t(s_t) &\approx \pi(t_{dummy}, s_t; \theta_\pi) \\ t_{dummy} =&\{\mathbf{1}(t=0), \mathbf{1}(t=1), \ldots, \mathbf{1}(t=T-1)\} \end{aligned}$$

This introduces T inputs in the network. This is costly in parameters compared to just introducing t directly as a continuous variable, but in practice it is much easier to train.

It is typically also advisable to ensure the inputs are scaled similarly, which however is already the case in the buffer-stock model. Other transformations can in principle also be applied, such that the inputs are not just the states, but also some transform there of. We have not yet found this helpful in practice.

4.5.3 Hyperparameters

A key aspect of deep learning algorithms is setting hyperparameters. There are many parameters and we often have at best heuristics for setting them. Here we briefly detail some of key drivers behind setting parameters.

- **Neural network structure.** A key choice is the depth of the network (how many hidden layers?) as well as the width (how many nodes in each layer?). We always use two layers and our baseline is 500 nodes in each layer. The total number of parameters is given by

  $$\underbrace{i \times n_1 + \sum_{k=1}^{K-1} n_k \times n_{k+1} + n_K \times o}_{\text{weigths}} + \underbrace{\sum_{k=1}^{K} n_k}_{\text{biases}}$$

  where i is number of inputs, o is the number of outputs: o, K is number of hidden layers, and n_k is number of neurons in hidden layer k. With 3 inputs and 1 output we have $3 \cdot 500 + 2 \cdot 500^2 + 1 \cdot 500 + 4 \cdot 500 = 254.000$ parameters. The next key choice is the hidden and final activation functions. We use *ReLU* for the all activation functions as a baseline, and we use *sigmoid* final actions functions to bound approximations in $[0;1]$.

- **Training sample.** A few hyper-parameters are key here. One is the number of sampled agents, N^{train}, in training. This choice depends on the algorithm. With *DeepSimulate* this number should be high as we also rely on N^{train} when taking expectations. We have $N^{\text{train}} = 5,000$. In the other-algorithms higher N^{train} is strictly about more coverage in the state-space. A high N^{train} can be extremely costly in terms of memory especially when we need to take expectations.[19]. We set $N^{\text{train}} = 150$ and assume the replay buffer has size $N^{\text{buffer}} = 8 \cdot N^{\text{train}}$ and as the batch size we use $N^{\text{batch}} = N^{\text{train}}$. Large buffer memory is inefficient as we are often using poorer samples of the true distribution from previous iterations. But training in the beginning especially for DeepFOC can be volatile and tends to explode. This can be avoided by using a larger memory which ensures that the sampled distribution is more stable across iterations.

  A second set of parameter(s) is the sequence $\sigma_{\epsilon,k}$ that determines the degree of exploration (for DeepFOC and DeepVPD) when generating the training sample. High $\sigma_{\epsilon,k}$ is very effective in the beginning of training when the simulated sample is a poor approximation of the true distribution anyway. In the later stages of training it is less valuable to have high ϵ as it dilutes the coverage in the interesting part of the state-space. Nonetheless it might still be valuable to have exploration both for taking expectations near the borders of the simulation and for stability. We set $\sigma_{\epsilon,k} = 0.1$ throughout in the buffer-stock model.

  For DeepVPD we only start training the policy function after 10 iterations.

- **Optimizers and learning rates.** We generally use the ADAM-optimizer and an exponential learning rate scheduler. We initially set the learning rate to 10^{-3} and for DeepFOC and DeepV we let it decay with a rate of 0.999 each iteration with a minimum value of 10^{-5}.

- **Early epoch termination:** For the policy updates we allow up to 15 epochs and terminate if there has been no improvement for 5 epochs. This strikes a balance between exploiting the information in the current sample and the cost

[19] This can, however, be alleviated by mini-batching if needed.

of simulating a new training sample. For the value updates we allow for up to 50 epochs and terminate if there has been no improvement for 10 epochs. We allow for more value updates as this is a faster function approximation problem.

- **Target Networks.** *DeepV* use target-networks to improve stability and precision. The key parameter here is τ which is the target adjustment rate. A low τ is very stable and very slow and vice versa for high τ. We set $\tau = 0.20$.

The full list of hyperparameters and our baseline choices are presented in Appendix Table B.1.

4.6 Convergence

In standard dynamic programming finite-horizon models are solved with backwards induction. Therefore the question of when to terminate is side-stepped. In particular, backwards induction terminates even if the grids are too rough to provide a detailed solution. To ensure the solution is precise enough it must be verified that it does not change with finer grids, but this is seldom done in practice. With the deep learning methods the issue of when to terminate is unavoidable.

We suggest to regularly calculate an out-of-sample expected reward R while solving as specified in the algorithms above. We can stop when the expected reward no longer improves »substantially«. Two issues with this is that it can be hard to interpret the size of changes in R, and that R can deviate substantially downwards for a longer period of iterations. To mitigate we do as follow each time the iteration counter is divisible by Δ_R:

1. If R is the best so far calculate the expected reward $R(\omega)$ for a grid of transfers $\omega \leq 0$ to an initial state (e.g. cash-on-hand).
2. Find the implied transfer for all previous best R so far using inverse interpolation of $R(\omega)$.
3. Stop if no best R in the last $\mathcal{T}$ minutes has a transfer $> \zeta$ bp.

We choose $\mathcal{T} = 20$ and $\zeta = 1$. We evaluate the performance of this convergence criterion in our advanced test case in Section 6.

5 Results: Buffer-Stock Model

In this section, we evaluate the speed, accuracy and stability of the deep learning algorithms when solving the buffer-stock model.

We use a mid-range L40 GPU (~9000$) to solve the model. We compare with a standard dynamic programming solution using a parallelized version of the very efficient Endogenous Grid Method (EGM) from Carroll (2006) in C++, which avoids any numerical optimization or root-finding. The model was solved using 72 threads on two Intel Xeon Gold 6254 3.1 GhZ (also ~9000$ in total).[20]

Additional details on the EGM are in Appendix C. We use the baseline hyperparameters from Appendix Table B.1.

Basic speed and accuracy As a simple measure of accuracy, we look at the *average realized discounted sum of utility* implied be each solution method using the same draw of initial states and shocks. We use this to compute an equivalent variation in terms of a transfer which ex ante would make agents indifferent between keep using the EGM solution and switching to a specific deep learning solution. A positive transfer implies that agents must be paid to be indifferent between EGM and DL implying that DL has a more accurate solution and vice versa for negative transfers. Specifically, we calculate the average realized discounted sum of utility for each algorithm using a sample of 100,000 agents in terms of initial states and shocks across all periods. For EGM we also compute the average realized discounted sum of utility adding a series of transfers to initial cash-on-hand. To find the equivalent variation for each deep learning algorithm we interpolate the inverse of the EGM discounted utility for each transfer level.

We let the deep learning algorithms run for 60 minutes. The plots in Figure 1 shows time on the x-axis and the implied transfer in basis points (bp.) of average initial cash-on-hand on the y-axis. Note that both axis have log-scales. A transfer in the range -10 to $+10$ basis points are thus all very small. The vertical line is the EGM solution time.

[20] This somewhat understates the monetary cost as substantial amounts of RAM is also needed for multi-dimensional models. For the extended buffer-stock model with two endogenous states and three fixed states we use 100 GB of RAM. On the other hand, the GPU cannot be used without some CPU.

We start with the baseline model with just cash-on-hand and permanent income as the states, i.e. any additional fixed states in panel (a). Here the EGM solution takes less than a hundredth of a minute (less than a second). DeepFOC gets within 10 bp. of the EGM solution in 1 minute and gets slightly better after about 10 minutes. DeepSimulate is slower getting within the 10 bp. range after about 6-7 minutes and is very close to 0 bp. after the full 60 minutes. DeepVPD is clearly the slowest algorithm only getting close to the 10 bp. range after the full 60 minutes. In panel (c) we see the effect of extending DeepVPD with i) three value networks instead of one, and ii) including the FOC for consumption in the policy loss. This DeepVPD++ algorithm is about as good as DeepSimulate.[21] This implies that the extra computational costs of both having more and more complex value networks and a more complex policy loss is small compared to the gain in accuracy in this model.

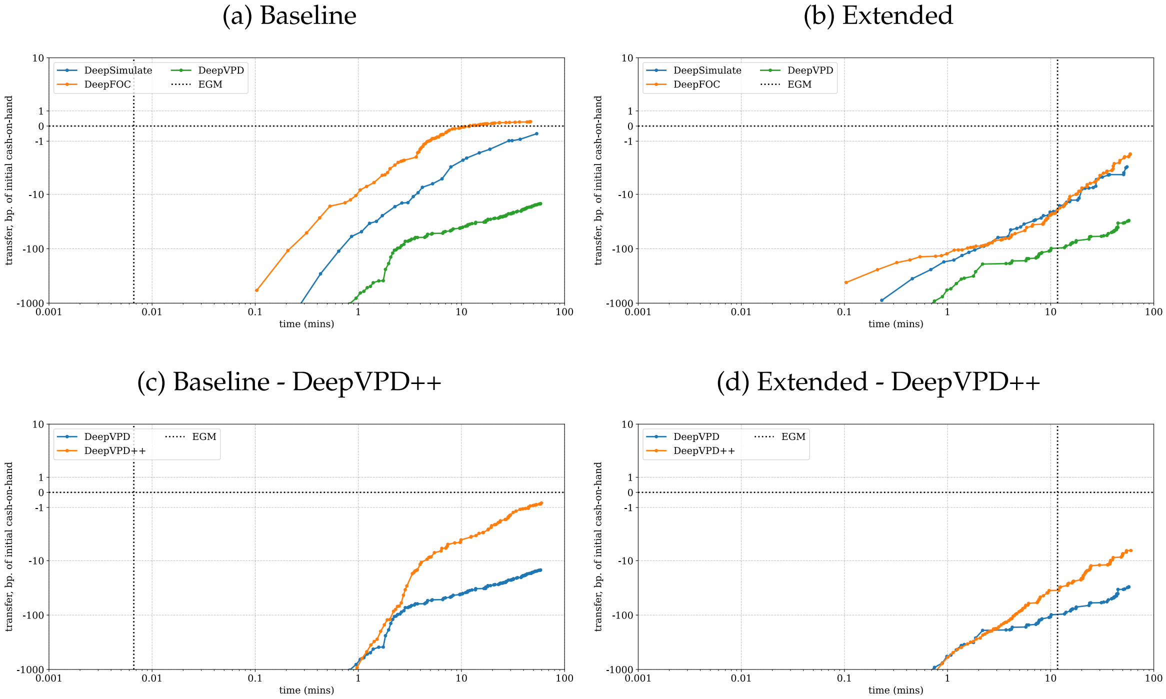
When we add three additional states the solution time of EGM increases substantially as seen in panel (b). We use 15 grid points in each dimension and therefore the solution time increases with a factor of 15^3 to a bit more than 10 minutes.[22] The time-accuracy paths of the deep learning algorithms change much less which is the first indication that they can tame the curse of dimensionality in this class of models. The required transfer for a given solution time is a bit worse, but both DeepFOC and DeepSimulate reach $<$ 10 bp. transfer at about 20 minutes, while DeepVPD++ as seen in panel (d) takes a bit longer.

The exact comparison to the EGM solution is not too important. We have tried to make the comparison fair in terms of using hardware with a similar cost and optimizing the code for EGM. But it could still e.g. be argued that EGM could suffice with fewer grid points[23] or be implemented faster on a GPU. Despite of this it would, however, still be that case that the solution time of EGM would increase exponentially with the number of dimensions.

[21] Appendix Figure C.2 shows the effect of including each of these extensions separately.

[22] The solution time increase with a bit more than a factor of 15^3 because each interpolation required now is over more dimensions.

[23] Choosing grid sizes and placement of grid points is like choosing hyperparameters for the deep learning algorithms. We have prioritized using hyperparameter, which works across multiple models, and have therefore not optimized them for the buffer-stock model in particular. In this sense it would be unfair to optimize the choice of grids for EGM.



(a) Baseline

(b) Extended

(c) Baseline - DeepVPD++

(d) Extended - DeepVPD++

Figure 1: Buffer-Stock Model: Speed and accuracy

Notes: The x-axis shows the time in minutes, while the y-axis shows the transfer required to make agents indifferent between the EGM solution and the DL solution. Both axes are in log scale. We only compute the transfer each 10th iteration in DeepFOC and DeepVPD and each 50th iteration in DeepSimulate. We only show the transfer when the algorithm improves in terms of average discounted utility. The vertical line indicates the solution time for EGM. DeepVPD++ (in panel b and d) is the baseline DeepVPD algorithm extended with i) the value networks instead of one, and ii) the FOC for consumption in the policy loss.

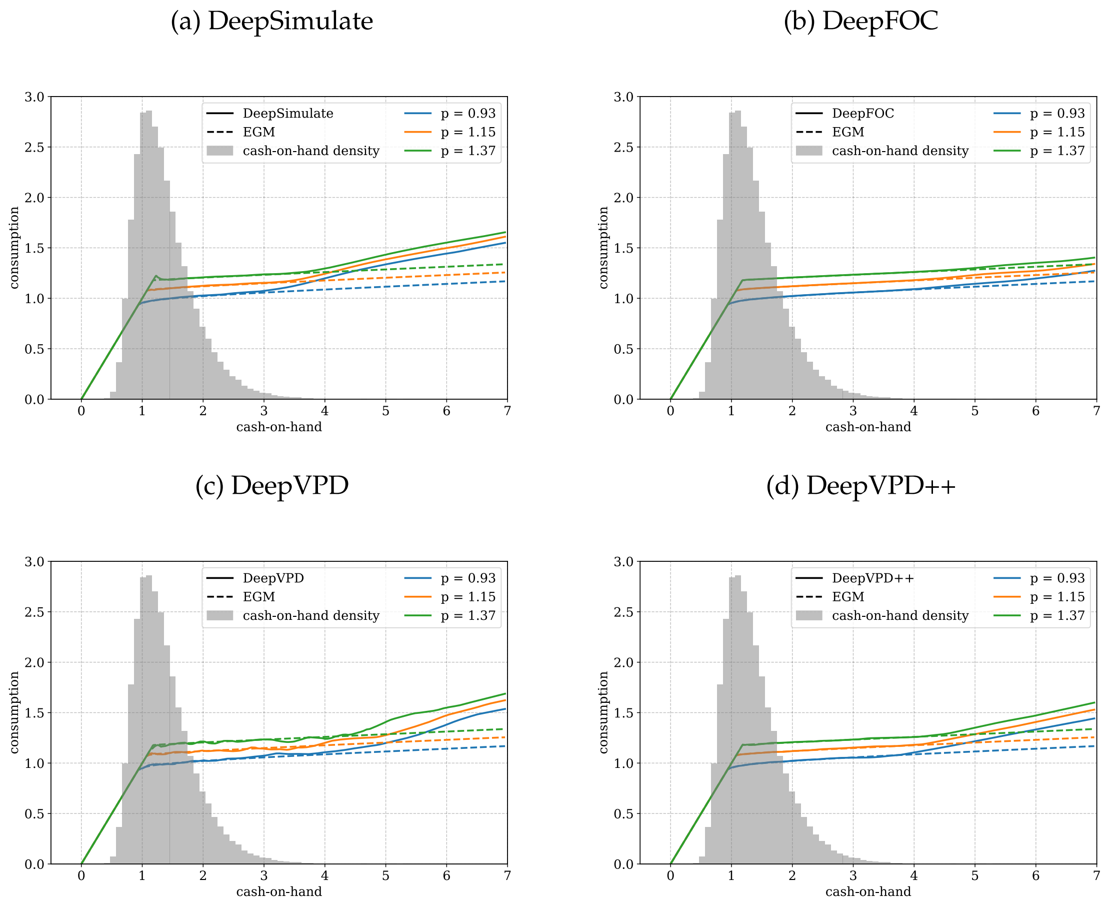
Consumption functions In Figure 2, we plot the implied consumption functions for the deep learning algorithms and EGM together with the histogram of cash-on-hand. We see that the deep learning algorithms work well for the active part of the state space agents actually visits in the simulation. Beyond this the solution quickly deteriorates. This is not a bug, but a feature. In higher dimensional models this is exactly what helps the deep learning algorithm to tame the curse of dimensionality.

The most imprecise consumption function is DeepVPD, where the consumption function is rather wiggly. But this is smoothed out with DeepVPD++. Appendix Figure C.6 shows the smoothing comes from both the averaging across multiple networks and adding the FOC for consumption in the policy loss.

A few details are worth pondering for intuition. If we focus on panel (a) which shows the policy from DeepSimulate. When $p = 1.37$ (green line) the solution is slightly off near the borrowing constrained area. This is due to the fact that the solution lies on a two-dimensional distribution and not the one-dimensional distribution shown by the histogram. There are only few individuals with high p and low m which means that this imprecision is due to low density in this area of the distribution. We also see that the green line matches the DP-solution for higher values of cash-in-hand compared to the blue line ($p = 0.93$). This is again due to the fact m and p are positively correlated and we have more density where both p and m are high compared to the area where p is low and m is high.

Comparing DeepVPD and DeepFOC with DeepSimulate shows that they seem to approximate the policy better for higher values of cash-on-hand. This is partially due to exploration - adding noise to actions implies that the variance of the cash-on-hand distribution increases such that we also find a good policy function for very wealthy individuals.

Exploration Figure 3 shows what happens when the scale of the exploration shocks is increased from 0.1 to 0.4. The right hand side figure shows that the right tail of the distribution of cash-on-hand in training expands considerably. The left hand side figure show this implies a good approximation for the consumption function even for larger values of cash-on-hand. Appendix Figure C.3 shows the convergence properties of the algorithms are almost unaffected by adding more exploration noise. We have, however, experienced that too little exploration can create convergence problems, where the algorithm gets stuck in e.g. a degenerate solution, where nobody


(a) DeepSimulate

(b) DeepFOC

(c) DeepVPD

(d) DeepVPD++

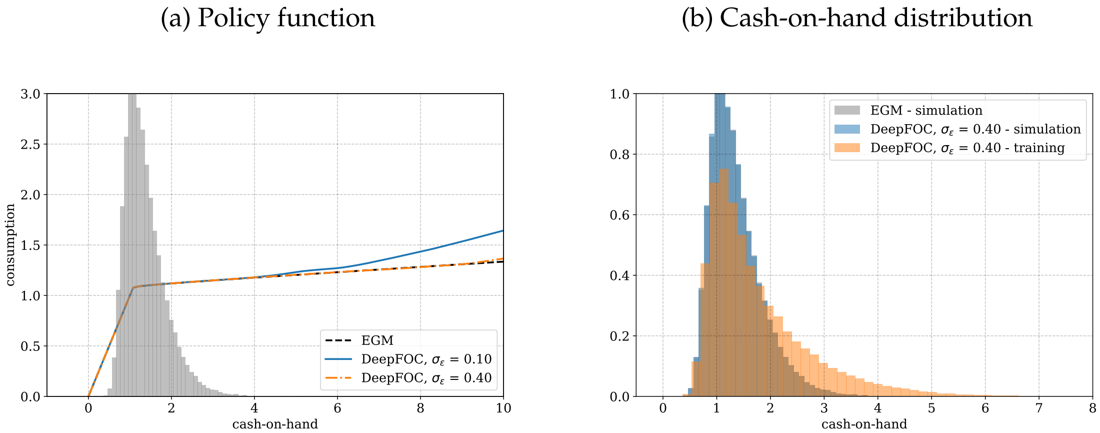
Figure 2: Buffer-Stock Model: Consumption Function (baseline)

Notes: The x-axis is a range of cash-on-hand values. The y-axis is a range of consumption values. The grey histogram is the cash-on-hand density from simulating the model with the EGM-policy. The full and dashed lines show the DL and EGM policies respectively for different levels of permanent income p.

saves. As mentioned above too much exploration can also be a problem as it dilutes the density of points in the relevant area of the state-space, though it is not a problem here.

Life-cycle profiles Figure 4 shows the implied life-cycle profiles for cash-on-hand for the extended model.[24] Economically speaking this is the most relevant measure of accuracy. We build economic models to explain the real world, and if our solution

[24] Results are similar for the baseline model and produced in the replication package.



(a) Policy function

(b) Cash-on-hand distribution

Figure 3: Buffer-Stock Model: Exploration (DeepFOC)

Notes: Panel (a): See Figure 2. The full and dashed lines show the DL and EGM policies respectively for different levels of exploration noise σ_ϵ. Panel (b): The grey histogram shows the cash-on-hand density from simulating with the EGM-policy. The orange histogram shows the cash-on-hand density from training samples generated when obtaining the exploratory DeepFOC policy function with $\sigma_\epsilon = 0.4$. The blue histogram shows the cash-on-hand density when simulating with the obtained exploratory DeepFOC policies.

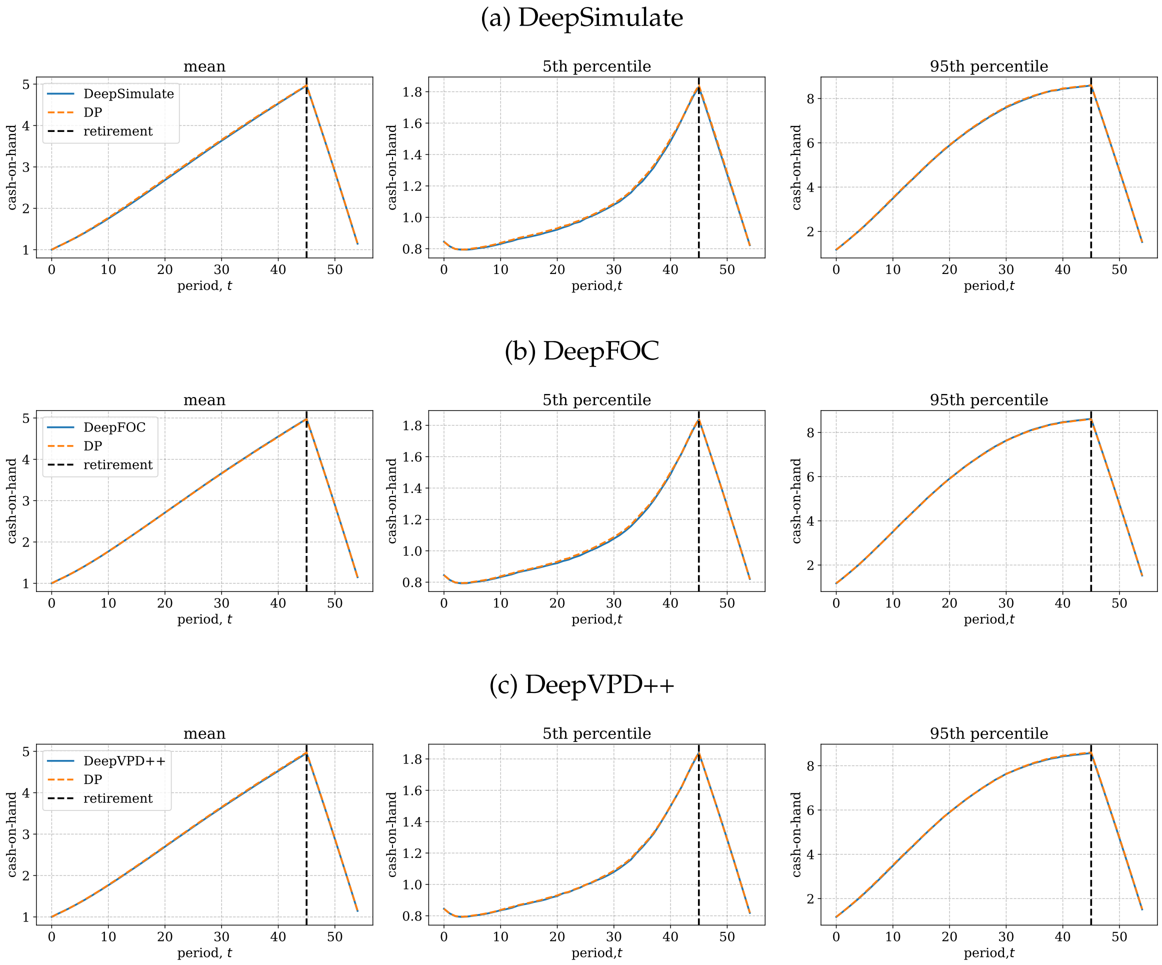
method is imprecise in terms of what it implies for the real world, policy analysis is not warranted. All three algorithms perform well. Appendix Figure C.4 and C.5 show the CDFs of cash-on-hand and the MPC over the life-cycle. Again all methods perform well.

Euler-errors A more technical measure of accuracy is Euler-errors. Euler errors are computed as follows:

$$\sum_{t=0}^{T-2}\sum_{i=0}^{N-1} \mathbf{1}(a_{t,i} > 10^{-3}) \left(\frac{(u_t')^{-1}\left(\beta R \mathbb{E}_t\left[u_{t+1}'(c_{t+1,i})\right]\right)}{c_{t,i}} - 1 \right) \tag{38}$$

where $\mathbf{1}(a_{it} > 10^{-3})$ is an indicator checking that individuals are not borrowing-constrained and $(u')^{-1}$is the inverse marginal utility of consumption. Figure 5 reports $\log_{10}$ of the absolute value of the Euler-error for all employed households.[25] A -2 is thus interpreted as a 10^{-2}, i.e. 1%, error as a share of consumption. The

[25] For retired household there is no uncertainty and the EGM method therefore delivers an exact zero Euler-error, which implies the $\log_{10}$ cannot be taken.

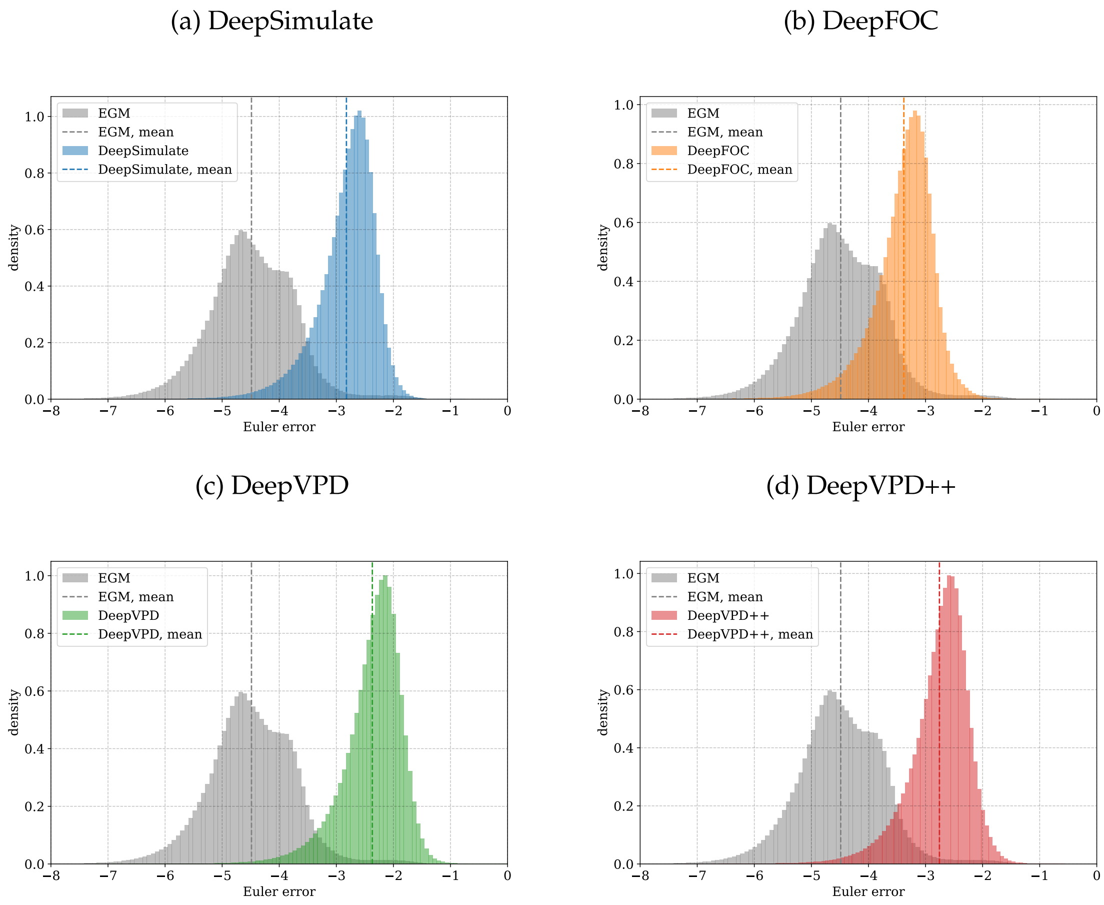

(a) DeepSimulate

(b) DeepFOC

(c) DeepVPD++

Figure 4: Buffer-stock model: Life-cycle profile of cash-on-hand (extended)

Notes: The first column of panels shows the mean of cash-on-hand through the life-cycle from the EGM simulation as well as for each of the DL-algorithms. A vertical dashed line shows when retirement occurs. The second and third columns show the simulated cash-on-hand for the 5th and 95th percentiles respectively.


(a) DeepSimulate

(b) DeepFOC

(c) DeepVPD

(d) DeepVPD++

Figure 5: Buffer-stock model: Euler-errors

Notes: All panels show the density of log10 euler-errors from the EGM-solution in the grey histogram. The other histograms show the density of log10 euler-errors from DL policies. The vertical lines show the mean of log10 euler-errors for EGM and DL.

ranking across the deep learning algorithms are clear. DeepFOC is best with a mean below -3, DeepSimulate is next with a mean just above -3 and DeepVPD is last with a mean around -2.5, though with a substantial improvement in DeepVPD++ to slighly above -3. The EGM solution is superior to all methods in terms of the Euler-errors as it directly used the Euler-equation even though it is actually implying slightly lower expected discounted utility on average as discussed above. As DeepFOC is also targeting the euler-errors it makes sense that it would be the best among the DL solutions.

Summary Table 1 show a summary summary of the results for the baseline model.

The upper part of the table reports the life-time reward, the transfer and the average log10 Euler-error, which we have already discussed.

The lower part of the table reports a decomposition of the total time and for the deep learning algorithms. Over 70% of the time is spent updating neural networks, primarily within highly optimized PyTorch routines like backpropagation. This suggests limited potential for further speed gains through micro-optimization of the code. The cost of computing the training samples is not more than 10 percent and especially low for DeepSimulate, where only the initial states and shocks are drawn. The computational cost of computing the life-time reward out-of-sample is of a similar size, the sample is much bigger, but also calculated infrequently.

DeepSimulate use far the most iterations as there is always only a single policy epoch. DeepFOC has the fewest iterations, also fewer than DeepVPD despite no value iterations. The reason is that each epoch is more expensive in DeepFOC as it needs to keep track of derivatives while computing expectations. The average number of policy epochs is below the maximum of 15 in DeepFOC, but is at the maximum for DeepVPD.

Going from DeepVPD to DeepVPD++ the extra steps in each iteration implies about a third fewer iterations. As the average number of policy epochs falls slightly the total number of policy updates thus decreases similarly. Many more value epochs are, however, completed as we sum across the multiple value networks. We can have this increase in the number of value epochs without a larger reduction in the number of iterations because each value epoch is very fast, and because GPU utilization improves. Appendix Tables C.6-C.6 show more detailed results for adding either multiple value networks or the FOC for consumption in the policy loss.

Table 1: Buffer-Stock Model – Baseline: Summary.

	EGM	DeepSimulate	DeepFOC	DeepVPD	DeepVPD++
Life-time reward, R	2.0616	2.0616	2.0617	2.0600	2.0616
Transfer (bp.)	0	-1	0	-15	-1
Mean log10 Euler error	-4.5	-3.0	-3.8	-2.4	-3.0
Total time (m)	0.01	60.01	60.01	60.01	60.01
update NN	-	93.1%	75.3%	71.7%	79.9%
simulation: training sample	-	2.3%	8.2%	10.1%	6.2%
simulation: out-of-sample R	-	2.3%	7.2%	9.1%	6.8%
Iterations	1	17118	10050	12062	9128
Avg. policy epochs	-	1.0	10.1	15.0	14.5
Avg. value epochs	-	0.0	0.0	19.8	53.0

Notes: Summary of results for the baseline buffer-stock model.

Table 2 shows a summary of the results for the extended model. The results are parallel to the baseline model.

Table 2: Buffer-Stock Model – Extended: Summary.

	EGM	DeepSimulate	DeepFOC	DeepVPD	DeepVPD++
Life-time reward, R	2.8601	2.8598	2.8599	2.8569	2.8594
Transfer (bp.)	0	-3	-2	-30	-7
Mean log10 Euler error	-4.5	-2.8	-3.4	-2.4	-2.8
Total time (m)	11.66	60.01	60.01	60.01	60.03
update NN	-	92.4%	80.6%	74.2%	82.4%
simulation: training sample	-	1.9%	6.0%	8.8%	4.8%
simulation: out-of-sample R	-	2.8%	6.5%	6.4%	6.4%
Iterations	1	15756	8872	11252	7431
Avg. policy epochs	-	1.0	13.8	15.0	14.3
Avg. value epochs	-	0.0	0.0	32.1	96.6

Notes: Summary of results for the extended buffer-stock model.

6 Advanced Test Case: Multiple Durable Goods

While the buffer-stock model is a fine test-case due to its canonical status it is too low-dimensional to fully take advantage of the gains from using deep learning. In this section, we set up a more complicated model.

6.1 Model

We consider a model where households choose both non-durable consumption c_t, and durable consumption $d_{t,j}$ for $D \geq 1$ durable goods indexed by $j \in \{0, 1, \ldots, D-1\}$. The adjustment of the durable consumption goods are subject to convex adjustment costs. The states are then cash-on-hand m_t, permanent income p_t and the durable goods possessed at the beginning of the period $n_{t,j}$.

This model is useful as it can be scaled up by increasing D. As D increases both the number of states and actions increase resulting in a model of growing complexity. When convenient we use boldface variables to denote vectors of durable goods, i.e. $\boldsymbol{d}_t = [d_{t,0}, d_{t,1}, \ldots, d_{t,D-1}]$ and $\boldsymbol{n}_t = [n_{t,0}, nt_{,1}, \ldots, n_{t,D-1}]$.

The per period utility is

$$u_t(c_t, \boldsymbol{d}_t) = \frac{\left(c_t^{1-\sum_{j=0}^{D-1} \omega_j} \left(\prod_{j=0}^{D-1} \left(d_{t,j} + \underline{d} \right)^{\omega_j} \right) \right)^{1-\rho}}{1-\rho} \tag{39}$$

$$\omega_j \in (0,1), \sum_{j=0}^{D-1} \omega_j \in (0,1),$$

where $\underline{d}$ is a small number, such that $d_{t,j} = 0$ does not imply infinitely negative utility.

We maintain the income function from the buffer-stock model and the transition function for permanent income

$$y_{t+1}(p_{t+1}, \psi_{t+1}) = \begin{cases} \kappa_t & \text{if } t \geq T^{\text{retired}} \\ \kappa_t \psi_{t+1} p_{t+1} & \text{else} \end{cases}, \ \ \psi_{t+1} \sim F_{t+1}^{\psi} = \exp \mathcal{N}(-0.5\sigma_{\psi}^2, \sigma_{\psi}^2), \tag{40}$$

$$p_{t+1} = \Gamma_{p,t}(p_t, \xi_{t+1}) = p_t^{\rho_p} \xi_{t+1}, \ \ \xi_{t+1} \sim F_{t+1}^{\xi} = \exp \mathcal{N}(-0.5\sigma_{\xi}^2, \sigma_{\xi}^2). \tag{41}$$

The amount purchased of each durable is denoted

$$\Delta_{t,j} = d_{t,j} - n_{t,j}. \tag{42}$$

And we assume that durable investment is non-negative; there is no resale market for durable goods:

$$\Delta_{t,j} \geq 0. \tag{43}$$

This assumption is just to make the problem slightly more interesting. Agents become hesitant to invest too much as to avoid being stuck with a large durable wealth stock if their income should suddenly fall.

We assume quadratic adjustment costs for each durable

$$\Lambda(\Delta_{t,j}) = \nu \Delta_{t,j}^2. \tag{44}$$

The cost parameter τ is identical for all j, but we can easily generalize to a case where the adjustment costs differ across the different durable goods.

The post-decision level of assets then is

$$\bar{m}_t = m_t - c_t - \sum_{j=0}^{D-1} \Delta_{t,j} + \Lambda(\Delta_{t,j}). \tag{45}$$

Agents cannot borrow:

$$\bar{m}_t \geq 0. \tag{46}$$

The transition for cash-on-hand is

$$m_{t+1} = R\bar{m}_t + y_t(p_{t+1}, \psi_{t+1}). \tag{47}$$

The durable goods depreciate with a rate of δ_j such that

$$n_{t+1,j} = (1 - \delta_j)d_{t,j}. \tag{48}$$

Finally, for the initial states, we assume:

$$m_0 \sim F_{s0} = \exp \mathcal{N}(-0.5\sigma_{m0}^2 + \log \mu_{m0}, \sigma_{m0}^2). \tag{49}$$

$$p_0 \sim F_{p0} = \exp \mathcal{N}(-0.5\sigma_{p0}^2 + \log \mu_{p0}, \sigma_{p0}^2). \tag{50}$$

$$n_{0,j} \sim F_{n0,j} = \exp \mathcal{N}(-0.5\sigma_{n0,j}^2 + \log \mu_{n0,j}, \sigma_{n0,j}^2). \tag{51}$$

Above we allowed durable to differ in two aspects: utility weights ω_j and depreciation rates δ_j. For simplicity we assume $\omega_j = \omega$.

6.2 Policy network

As with the buffer-stock model, we can approximate the policy function in a number of ways. As before, we prefer to enforce the inequality constraints directly in the network structure. Specifically, we assume that the final activation functions are *sigmoid* and thus $a_{t,k} \in [0;1]$ for $k \in \{0, 1, \dots D\}$.

First we find consumption as

$$c_t = a_{tD} m_t.$$

We find $d_{t,0}$,$d_{t,1}$, . . . ,$d_{t,D-1}$ sequentially starting with $m_t^0 = m_t - c_t$ as

$$\begin{aligned} \overline{\Delta}_{t,j} &= \max \left\{ \Delta_{t,j} \,|\, \Delta_{t,j} + \nu \Delta_{t,j}^2 \le m_t^j \right\} \\ d_{tj} &= n_{tj} + a_{tj} \overline{\Delta}_{tj} \\ m_t^{j+1} &= m_t^j - \left(\Delta_{tj} + \nu \Delta_{tj}^2 \right), \end{aligned} \tag{52}$$

where $\overline{\Delta}_{tj}$ is the maximum for how much can be purchased of durable good j conditional on the purchase of up to durable good $j-1$ without breaking the budget constraint.

Using this sequential procedure ensures no infeasible predictions, which avoids the need for penalty functions to handle the inequality constraints.

6.3 DeepFOC

With DeepFOC we need to derive first order conditions. We derive the full KKT conditions for solving this model in Appendix E. λ_t is the multiplier for the borrowing constraint and $\mu_{t,j}$ are the multipliers for the non-negative investment constraint on the durable goods.

The complementary slackness conditions are

$$\lambda_t b_t = 0 \tag{53}$$

$$\mu_{tj}\Delta_{t,j} = 0. \tag{54}$$

The first-order conditions are

$$u_c'(c_t, \boldsymbol{d}_t) = \beta R \mathbb{E}_t\left[u_c'(c_{t+1}, \boldsymbol{d}_{t+1})\right] + \lambda_t \tag{55}$$

$$\begin{aligned} u_{d_j}'(c_t, \boldsymbol{d}_t) = {} & (1 + \Lambda'(\Delta_{t,j}))u_c'(c_t, \boldsymbol{d}_t) \\ & - (1-\delta)\beta\mathbb{E}_t[(1 + \Lambda'(\Delta_{t+1,j}))u_c'(c_{t+1}, \boldsymbol{d}_{t+1})] \\ & + (1-\delta)\beta\mathbb{E}_t[\mu_{t+1,j}] - \mu_{t,j}. \end{aligned} \tag{56}$$

When using DeepFOC, we assume that both the actual choices and the KKT multipliers are outputs of the policy network, and minimize the errors in both the first order conditions and the complementary slackness conditions.

In the buffer-stock model we encoded the inequality constraints using the Fischer-Burmeister (FB) approach instead, but this is much harder when there are multiple inequality constraints to handle.

6.4 Results: DP vs DL

We now turn to evaluating the speed, accuracy and stability of the deep learning algorithms when solving the durable goods model.

We use the same hardware as in the simple test case. We compare with a standard dynamic programming solution using a parallized version of the efficient Nested Endogenous Grid Method (NEGM) from Druedahl (2021) in C++, which avoids any numerical optimization or root-finding for the consumption choice, but use numerical optimization for the remaining choices as in value function iteration. Details on NEGM is in Appendix E. We consider two different sizes of grids

NEGM+: $\#_p = 100, \#_m = 150, \#_n = 150$

NEGM-: $\#_p = 50, \#_m = 50, \#_n = 50$

where $\#_x$ is the number of grid points for variable x.

We use the baseline hyperparameters in Appendix Table B.1 with the following exceptions

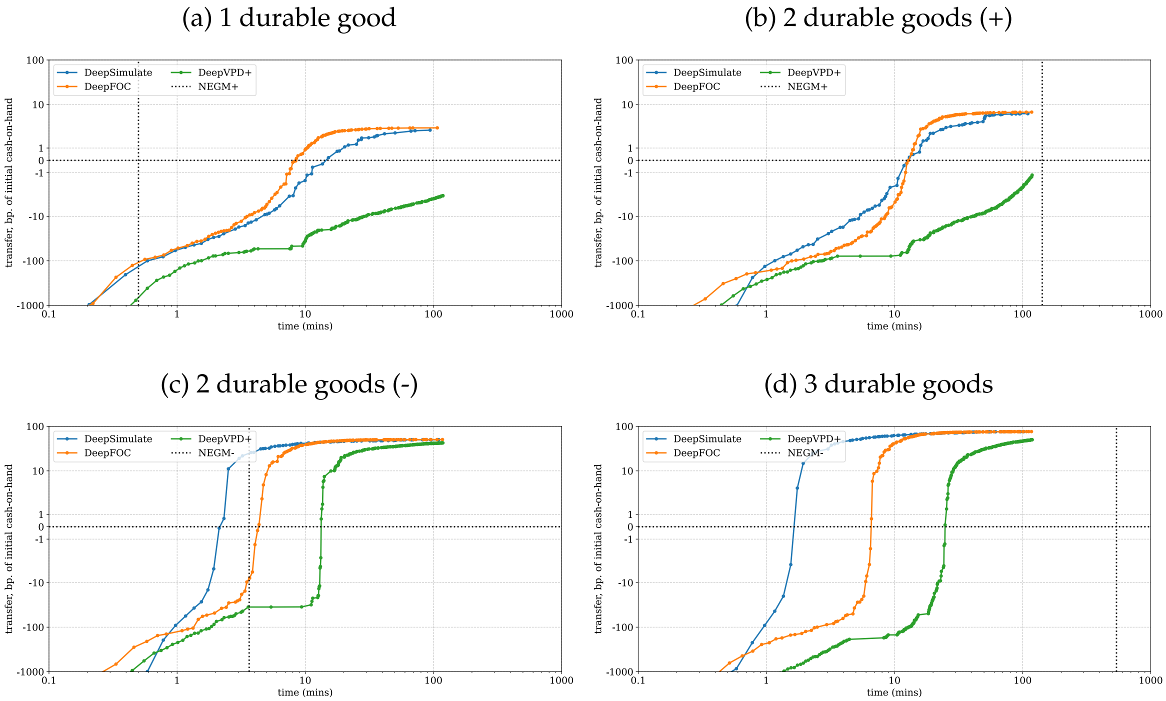
> DeepSimulate: Unchanged.
>
> DeepFOC: We use 700 neurons in the policy network to allow for approximation of not just the growing number of actions, but also the growing number of multipliers.
>
> DeepVPD: We only focus on DeepVPD+ with 3 value networks. We use a faster decay of the learning at 0.999 instead of 0.9999 for both the policy and value parameters. We set target network adjustment parameter $\tau = 1.0$.
>
> Exploration: With 3 durable the exploration scale factor σ_ϵ is reduced to 0.05 for each action.

Figure 6 shows the speed and accuracy for the deep learning algorithms for models with 1, 2 and 3 durable goods. We can solve the model with standard dynamic programming for up to 3 durable goods. With 1 durable good we use the fine grids in NEGM+. With 2 durable goods we use both the rough and the fine grids in NEGM- and NEGM+. With 3 durables we only use the rough grids in NEGM- as the RAM requirement here already exceeds 500 Gb.

We now discuss the timing and accuracy results for the different specifications one at a time. Appendix Tables D.1-D.1 contains detailed additional results.

1 durable good. With one durable good (upper left) the deep learning algorithms reach the accuracy of the EGM solution, but is much slower. DeepSimulate and DeepFOC are within a 10 bp. transfer of the NEGM in around 10 minutes, which DeepVPD takes more than an hour. Looking both the life-cycle profiles (Appendix Figure D.5) and Euler-errors (Appendix Figure D.5) the solutions are very similar. Appendix Figure D.2 confirms the policy function of DeepSimulate aligns with the NEGM solution, if anything the deep learning solution is less wiggly.

2 durable goods. With two durable goods (upper right) the deep learning algorithms are only slightly slower to get to the same level of relative accuracy (same



(a) 1 durable good

(b) 2 durable goods (+)

(c) 2 durable goods (-)

(d) 3 durable goods

Figure 6: Durables Model: Speed and accuracy for $D = 1, 2$ or 3

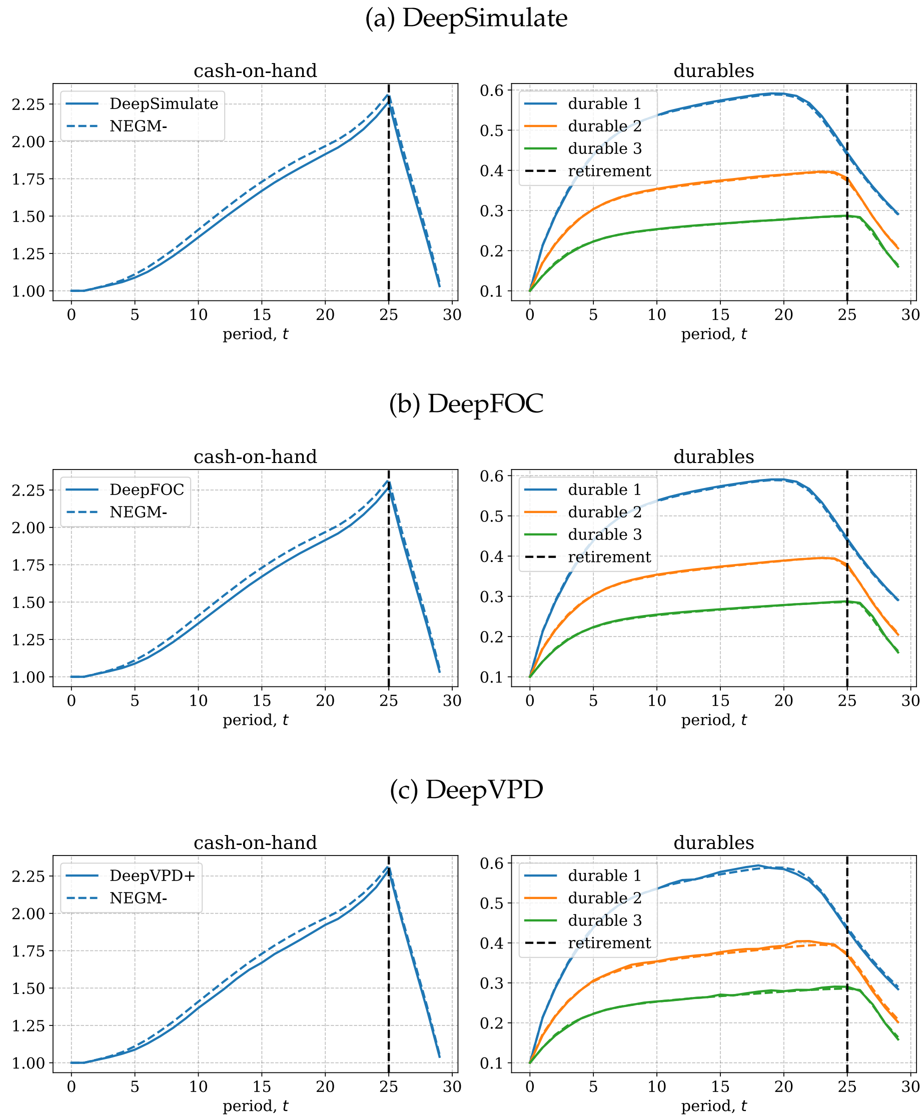
Notes: See Figure 1. + indicates that we use fine grids and - indicates that we use rough grids when using NEGM.

transfer), but the solution time for the NEGM model skyrockets from 0.5 minutes to 140 minutes. The increase in the solution time for NEGM+ is above a factor $\#_n = 150$, which is the number of grid points for each durable goods. The reason is that for each grid point more work now needs to be done, specifically solving a two dimensional instead of a one dimensional optimization problem, and interpolate in one more dimension.

It is impressive that the deep learning algorithms handles the extra dimension so well. DeepFOC is somewhat slowed down by the fact that an extra durable good choice also implies more constraints and multipliers, which is must handle explicitly. Therefore DeepSimulate outperforms DeepFOC with two durable goods. DeepVPD still does not quite reach the accuracy of the NEGM solution, but is within 10 bp. before an hour. Figure D.4 shows that the implied life-cycle profiles are very similar across the different solution methods. Life-cycle profiles (Appendix Figure D.6) looks very similar across solution methods, and the Euler-errors (see Appendix Figure D.6) are consistently smaller for the deep learning methods, except for DeepVPD+ where they are of similar size as with NEGM.

If we use rough grids for NEGM we can get the solution time down to less than 4 minutes (lower left). But the lower accuracy implies that DeepSimulate reaches a zero-transfer before that. All the deep learning algorithms reach transfer clearly above 10 bp. of initial cash-on-hand indicating the NEGM solution is not precise enough. In sum the deep learning algorithms seems approximately as good or better than NEGM with 2 durable goods.

3 durable goods. The story is repeated with three durable goods (lower right). The grids for the NEGM solution are now the rough ones, but the solution time still grows with a factor of more than $\#_n = 50$ to above 500 minutes (almost 9 hours) because of the extra dimension. The deep learning solution all reach same level of accuracy in less than 20 minutes, and they all find substantially better solutions within a hour. DeepSimulate only needs a few minutes to reach the same level of accuracy. Figure 7 shows that the solution methods imply similar life-cycle profiles, but the rough DP solution overstates the accumulation of cash-on-hand. In terms of the Euler-errors in Figure 8 the story is the same as for 2 durables, DeepSimulate and DeepFOC gives smaller errors than EGM, and DeepVPD+ gives errors of similar size.

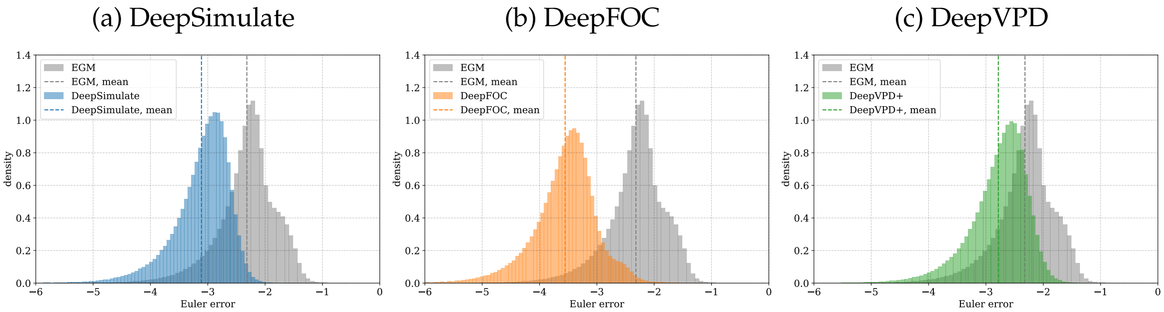

(a) DeepSimulate

(b) DeepFOC

(c) DeepVPD

Figure 7: Durables Model (3 durables): Life-cycle profiles

Notes: The first column shows mean simulated cash-on-hand obtained from NEGM- policies and DL policies through the life-cycle. The second column shows mean simulated stock of durables through the life-cycle


(a) DeepSimulate

(b) DeepFOC

(c) DeepVPD

Figure 8: Durables Model (3D): Euler-errors

Notes: See Figure 5

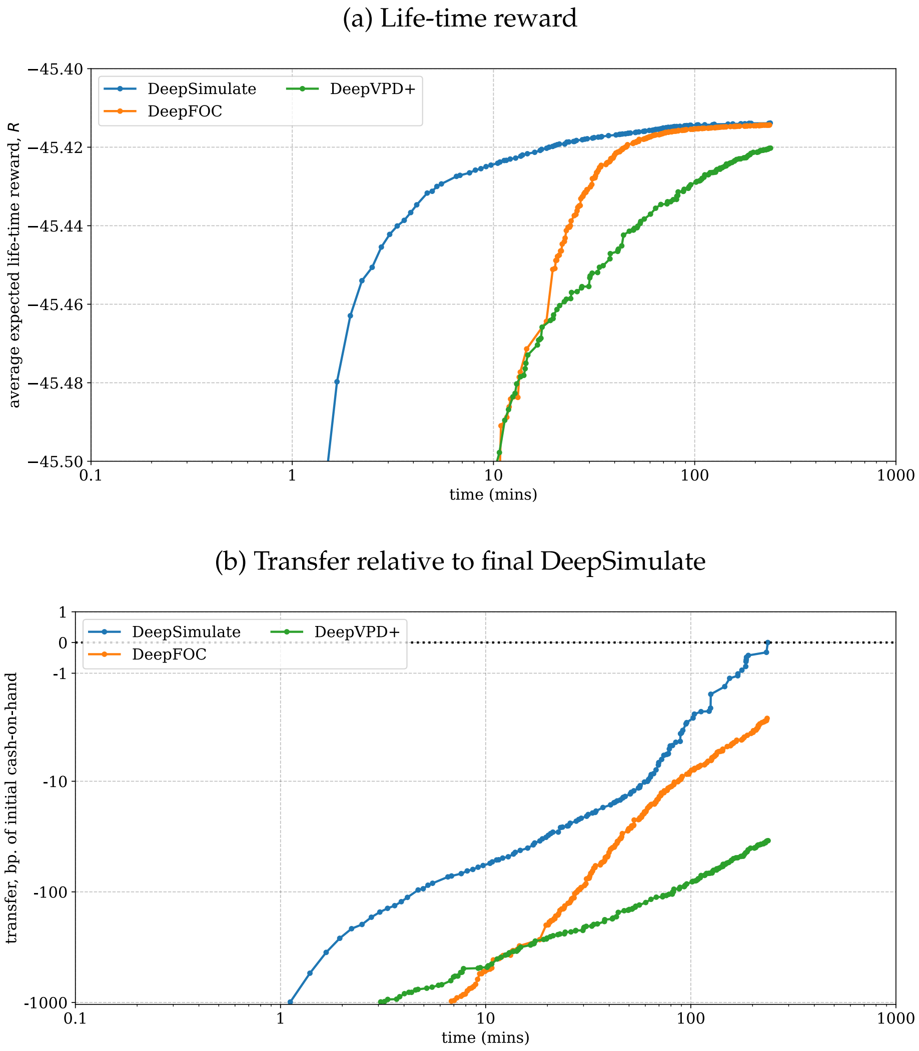
6.5 Far beyond the current limit: 8 durable goods

Finally, we consider a model with 8 durable good and thus a total of 10 state variables. Here the standard dynamic programming solution methods no longer apply in practice due to the curse of dimensionality. We change the hyperparameters relative to the baseline in Appendix Table B.1 as follows:

> DeepSimulate: Nothing is changed.
>
> DeepFOC: 1000 neurons in each policy layer.
>
> DeepVPD: 1000 neurons in each value layer. We use 4 value networks instead of 3. We set target network adjustment parameter $\tau = 0.8$.
>
> Exploration: σ_ϵ is reduced to 0.03 for each action. Set to 0.06 in DeepVPD.

In Figure 9 we illustrate the speed and accuracy of each algorithm. In panel (a) we directly show the average discounted utility. In panel (b) we compute transfers relative to the final DeepSimulate solution. We run each algorithm for four hours. After about 100 minutes DeepSimulate and DeepFOC have reached very similar solutions, which does change much there thereafter, less than 10 bp. The DeepVPD+ solution gets closer and closer to this solution but remains above 10 bp. away.

Figure 10 confirms the life-cycle profiles are similar, especially for DeepSimulate and DeepFOC. Figure 11 shows the cross-sectional correlation of cash-on-hand and each durable over the life-cycle. As the initial states and the shocks are the same equivalent solutions should give a correlation of one.



(a) Life-time reward

(b) Transfer relative to final DeepSimulate

Figure 9: Durables Model: Speed and accuracy for $D = 8$

Notes: See Figure 1. In panel (a) we directly show the average discounted utility. In panel (b) we compute transfers relative to the final DeepSimulate solution.

The Euler-errors in Figure 12 continuous to be small for all three solutions methods. For DeepVPD the mean is lowest at -2.6, for DeepSimulate it is -3, while it for DeepFOC gets to -3.4 despite a somewhat large right tail.

In Figure 13, we plot the policy functions for DeepSimulate. The consumption function and durable purchase functions looks smooth and reasonable. The ordering of the purchase of durable goods is fully aligned in terms of higher purchase of the durable goods with lower depreciation rates for an equal utility weight.

In sum, it seems that all algorithms find quite similar solutions which, we argue, is an indication that we have found a global optimum as it is unlikely that both DeepSimulate and DeepFOC have found almost the exact same wrong solution.

6.6 Convergence

So far we have just let the deep learning algorithms run for a fixed number of minutes. We now consider the formal convergence criterion form Section 4.

Figure 14 evaluates the performance of this convergence criterion for the model with 8 durable.[26] The grey box illustrates the convergence criterion. If the box is entered from below there have been less than 20 minutes with improvements less than 1 bp. If it enters from the left there have been more than 20 minutes with improvements less than 1 bp. We see that DeepSimulate and DeepFOC has reached convergence, while DeepVPD has not. This seems reasonable. In panel (d) we plot the policy loss in DeepFOC, which can also be used for a convergence criterion as it is equation errors (though not-out-of-sample).

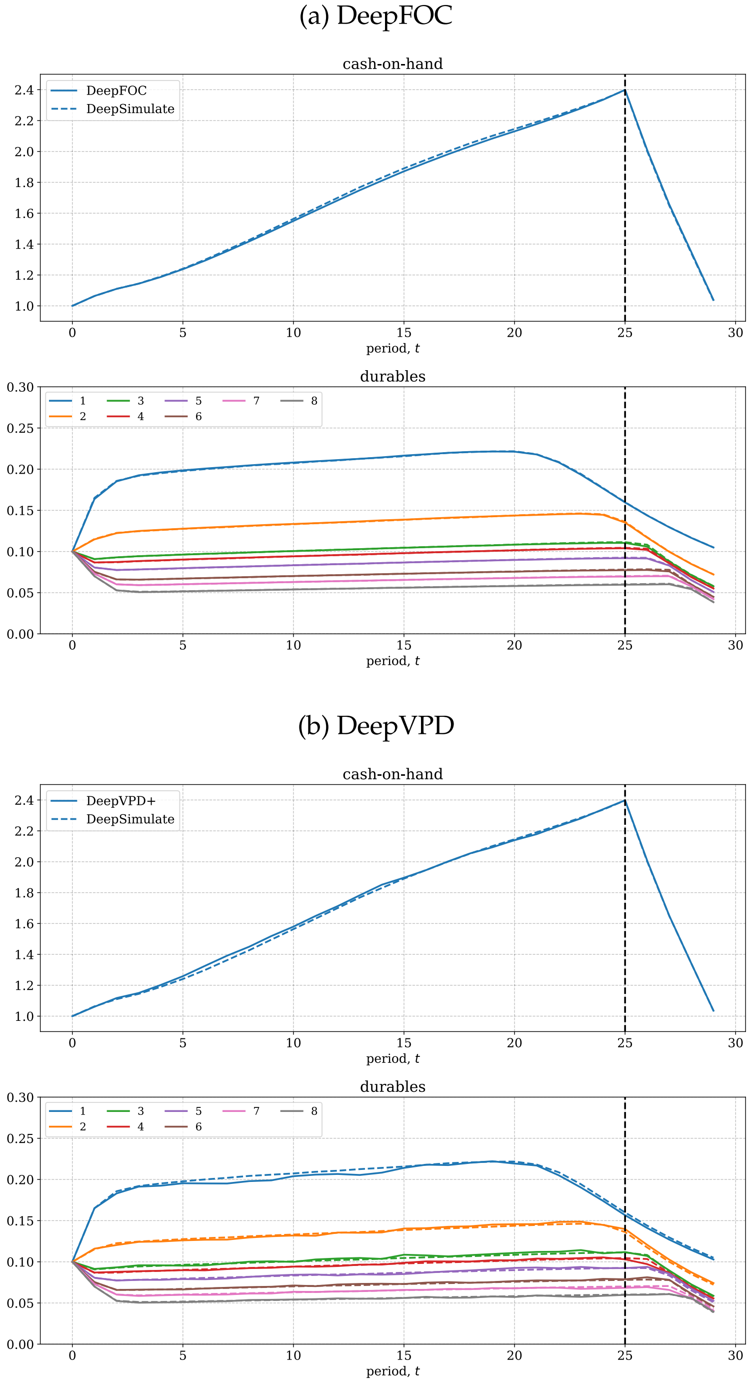
7 Non-Convex Models: Discrete and Continuous Choices

7.1 Model

We consider a model with a non-durable good and a durable good subject to a non-convex adjustment cost. Time is discrete and indexed by $t \in \{0, 1, \dots T-1\}$.

In the beginning of each period the state variables for the agent is permanent income

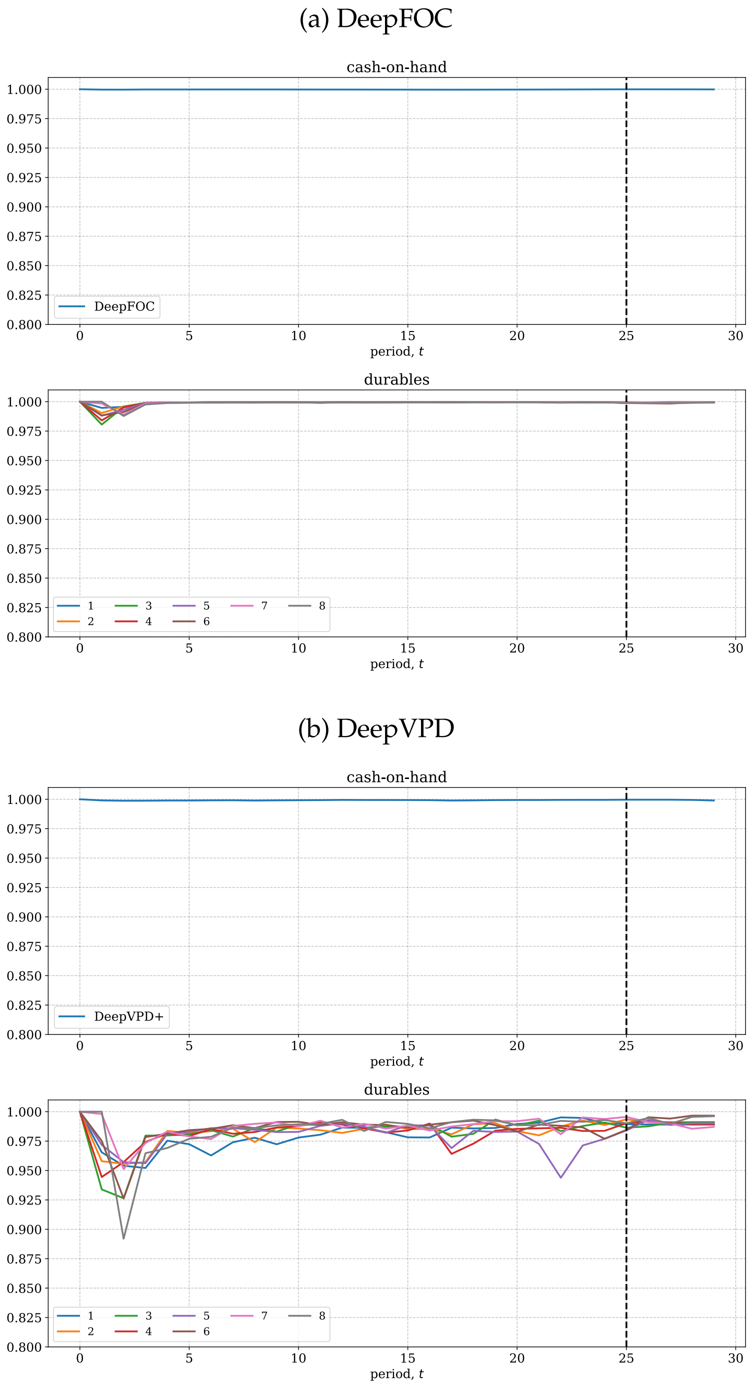
[26] Results for all other models and their various specifications is available in the replication package.


(a) DeepFOC

(b) DeepVPD

Figure 10: Durables Model (8 durables): Life-cycle profiles

Notes: Shows the average life-cycle profiles for cash-on-hand and each durable.


(a) DeepFOC

(b) DeepVPD

Figure 11: Durables Model (8 durables): Cross-sectional life-cycle correlation

Notes: Shows the cross-sectional correlation of cash-on-hand and each durable over the life-cycle. As the initial states and the shocks are the same, equivalent solutions should give a correlation of one.

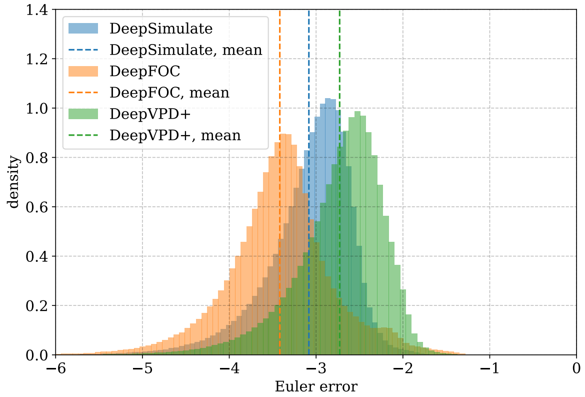


Figure 12: Durables Model (8 durables): Euler-errors

Notes: Shows the Euler-errors for each solution method.

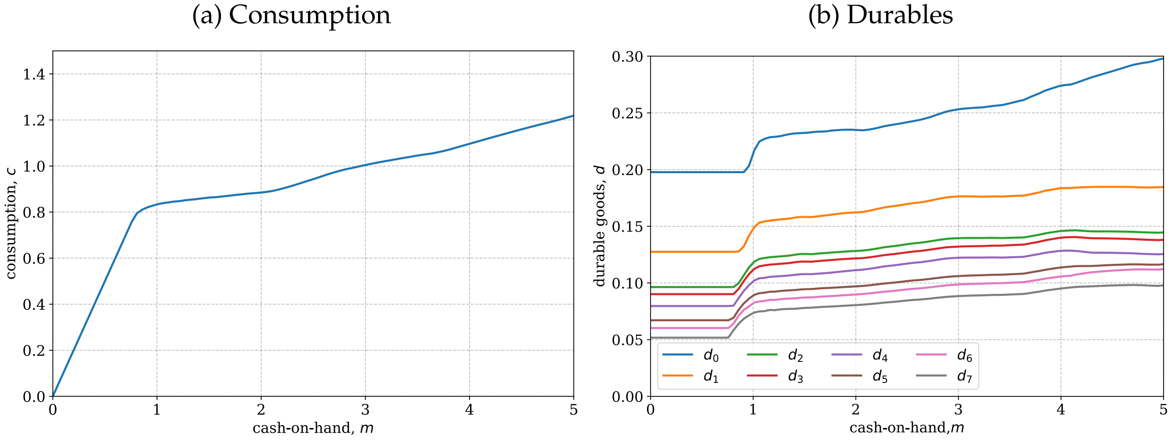
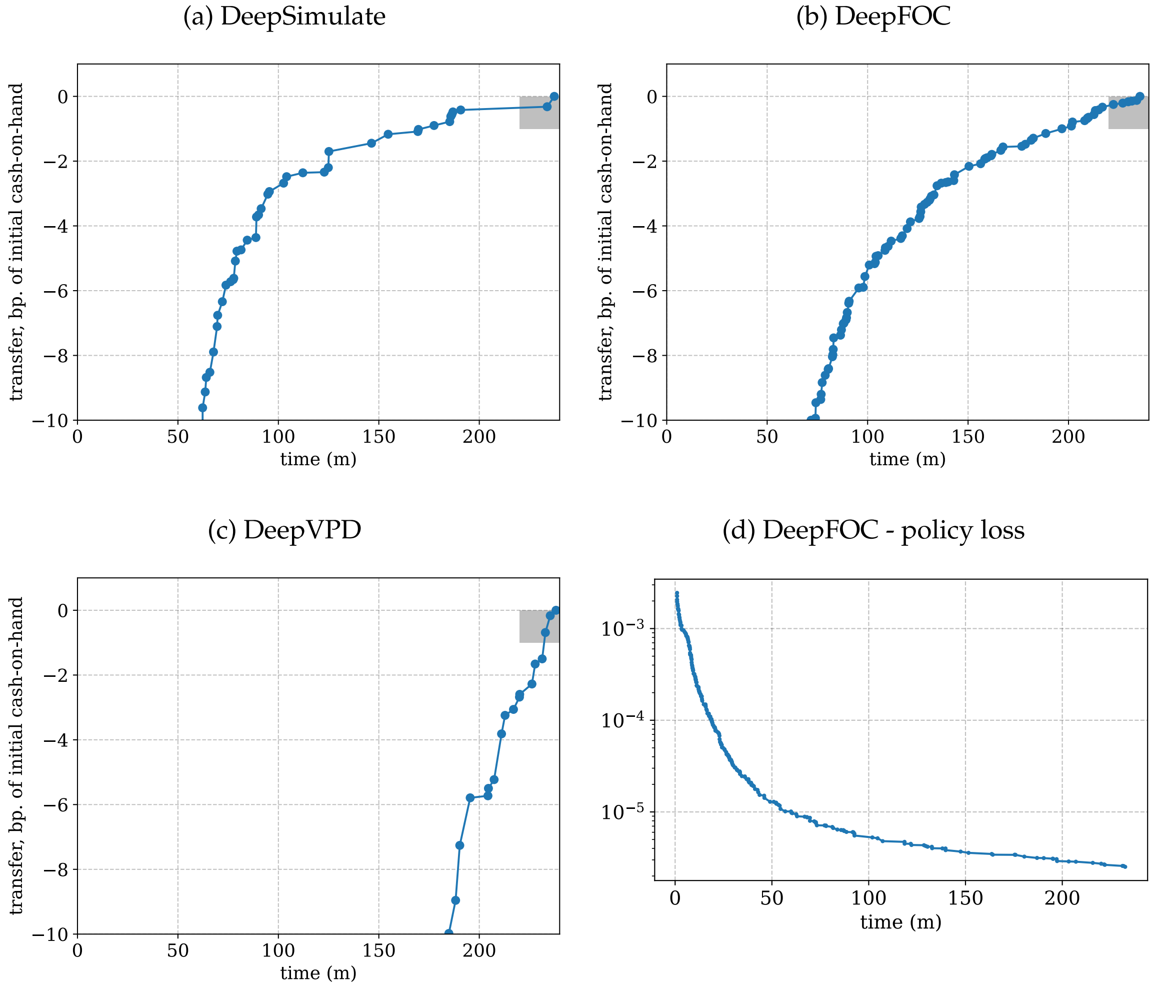


Figure 13: Durables Model (8 durables): Policy functions for DeepSimulate

Notes: Shows the policy functions obtained from DeepSimulate


(a) DeepSimulate

(b) DeepFOC

(c) DeepVPD

(d) DeepFOC - policy loss

Figure 14: Durables Model:Convergence

Notes: Panels (a)-(c) show the transfer of the so-far best *R* relative to the final solution. The grey box illustrates the convergence criterion. If the box is entered from below there has been less than 20 minutes with improvements less than 1 bp. If it enters from the left there has been more than 20 minutes with improvements less than 1 bp. In panel (d) we plot the policy loss in DeepFOC (though not-out-of-sample).

p_t, the stock of the durable good n_t and cash-on-hand m_t. The agent makes a choice between *adjusting* or *keeping* his or her durable stock. If adjusting the agent must sell the current durable stock with a proportional loss of $\kappa \in (0,1)$. Cash-on-hand afterwards is $x_t = m_t + (1-\kappa)n_t$. At the end of each period, the agent has a stock of the durable good d_t and assets b_t. Permanent income follows an AR(1) process, the durable good depreciates with a rate of $\delta \in (0,1)$ and the return factor for savings is $R > 0$. The agent has CRRA preferences over a Cobb-Douglas aggregate of non-durable and durable consumption.

$$u(c_t, d_t) = \frac{\left(c_t^{\alpha} d_t^{1-\alpha}\right)^{1-\rho}}{1-\rho} \qquad \alpha \in (0,1), \rho > 0, \rho \neq 1 \tag{57}$$

The overarching value function is

$$\begin{aligned} v_t(p_t, n_t, m_t) &= \max\left\{v_t^{\text{keep}}(p_t, n_t, m_t) + \varepsilon_t^{\text{keep}}, v_t^{\text{adj}}(p_t, x_t) + \varepsilon_t^{\text{adj}}\right\} \\ x_t &= m_t + (1-\kappa)n_t \end{aligned} \tag{58}$$

where ε_t are extreme value shocks with scale parameter σ_ε. Defining

$$\begin{aligned} \overline{v}_t &= \max\left\{v_t^{\text{keep}}, v_t^{\text{adj}}\right\} \\ \tilde{v}_t^{\text{adj}} &= \exp\left(\frac{v_t^{\text{adj}} - \overline{v}_t}{\sigma_\varepsilon}\right) \\ \tilde{v}_t^{\text{keep}} &= \exp\left(\frac{v_t^{\text{keep}} - \overline{v}_t}{\sigma_\varepsilon}\right) \end{aligned}$$

the closed form expression for the overarching value function and the adjustment probability is

$$v_t(p_t, n_t, m_t) = \overline{v}_t + \sigma_\varepsilon \log\left(\exp\left(\frac{\tilde{v}_t^{\text{keep}}}{\sigma_\epsilon}\right) + \exp\left(\frac{\tilde{v}_t^{\text{adj}}}{\sigma_\epsilon}\right)\right) \tag{59}$$

$$\text{Prob}[d_t \neq n_t | p_t, n_t, m_t] = \frac{\exp\left(\frac{\tilde{v}_t^{\text{adj}}}{\sigma_\epsilon}\right)}{\exp\left(\frac{\tilde{v}_t^{\text{keep}}}{\sigma_\epsilon}\right) + \exp\left(\frac{\tilde{v}_t^{\text{adj}}}{\sigma_\epsilon}\right)} \tag{60}$$

The post-decision value function is

$$
\begin{aligned}
\bar{v}_t(p_t, d_t, b_t) &= \begin{cases} 0 & \text{if } t = T-1 \\ \mathbb{E}_t\left[v_{t+1}(p_{t+1}, n_{t+1}, m_{t+1})\right] & \text{else} \end{cases} \qquad (61) \\
& p_{t+1} = p_t^{\eta}\xi_{t+1}, \;\; \xi_{t+1} \sim \exp\mathcal{N}\left(-\frac{1}{2}\sigma_{\xi}^2, \sigma_{\xi}^2\right) \\
& n_{t+1} = (1-\delta)d_t \\
& m_{t+1} = Ra_t + p_{t+1}
\end{aligned}
$$

The problem for the *keepers* are

$$
\begin{aligned}
v_t^{\text{keep}}(p_t, n_t, m_t) &= \max_{c_t} u(c_t, n_t) + \beta\bar{v}_t(p_t, n_t, b_t) \qquad (62) \\
& \text{s.t.} \\
& b_t = m_t - c_t
\end{aligned}
$$

The problem for the *adjusters* are

$$
\begin{aligned}
v_t^{\text{adj}}(x_t) &= \max_{c_t, d_t} u(c_t, d_t) + \beta\bar{v}_t(p_t, n_t, b_t) \qquad (63) \\
& \text{s.t.} \\
& b_t = x_t - c_t - d_t
\end{aligned}
$$

which can alternatively be written

$$
\begin{aligned}
v_t^{\text{adj}}(x_t) &= \max_{d_t} u(c_t, d_t) + \beta\bar{v}_t(p_t, n_t, b_t) \\
& \text{s.t.} \\
& c_t = c_t^{\text{keep},\star}(p_t, n_t, x_t - d_t) \\
& b_t = x_t - c_t - d_t
\end{aligned}
$$

or even just

$$
v_t^{\text{adj}}(x_t) = \max_{d_t} v_t^{\text{keep}}(p_t, d_t, x_t - d_t)
$$

The calibration is shown in Appendix E.

7.2 Solution

To solve the with *Deep Learning* the discrete choice implies that of our three algorithms only *DeepVPD* can be applied. We let the policy network have 3 outputs, a_t^0, a_t^1, and a_t^2, and we use a *sigmoid* for the final activation so all outputs are in $[0;1]$. The continuous choices across discrete choices then are

$$
\begin{aligned}
c_t^{\text{keep}} &= (1-a_t^0)m_t && (64)\\
c_t^{\text{adj}} &= a_t^2(1-a_t^1)x && (65)\\
d_t^{\text{adj}} &= (1-a_t^2)(1-a_t^1)x_t && (66)
\end{aligned}
$$

where a_t^0 is the savings rate for the keepers, a_t^1 is the savings rate for the adjusters, and a_t^2 is the expenditure share on non-durable consumption for the adjusters.

To make the discrete choice, we first calculate the implied value-of-choices as

$$
\begin{aligned}
v_t^{\text{keep}}(p_t, n_t, m_t) &= u\left(c_t^{\text{keep}}, n_t\right) + \bar{v}(t, p_t, n_t, b_t^{\text{keep}}; \theta_w)\\
b_t^{\text{keep}} &= m_t - c_t^{\text{keep}}\\
v_t^{\text{adj}}(p_t, n_t, m_t) &= u\left(c_t^{\text{adj}}, d_t^{\text{adj}}\right) + \bar{v}(t, p_t, d_t^{\text{adj}}, b_t^{\text{adj}}; \theta_w)\\
b_t^{\text{adj}} &= m_t + (1-\kappa)n_t - c_t^{\text{adj}} - d_t^{\text{adj}}
\end{aligned}
$$

When solving we use these to calculate choice probabilities using equation (60). In the simulation, we draw the extreme value shocks and the agent simply adjust if

$$
v_t^{\text{adj}}(p_t, x_t) + \varepsilon_t^{\text{adj}} > v_t^{\text{keep}}(p_t, n_t, m_t) + \varepsilon_t^{\text{keep}}
$$

7.3 Results

For comparison we used a dynamic programming VFI-solution that follows the same strategy as the Deep Learning approach. We set $T = 15$. Comparing the DP and DL solutions using the transfer measure gives a transfer of -2 bp. of initial cash-on-hand which indicates that the DP solution is slighty better but the difference is very minor. Figure 15 shows simulated outcomes for the DL and DP solutions. We see very similar profiles for consumption and durables stock. Slight differences in the adjuster share results in differences in cash-on-hand. As a proof-of-concept ex-

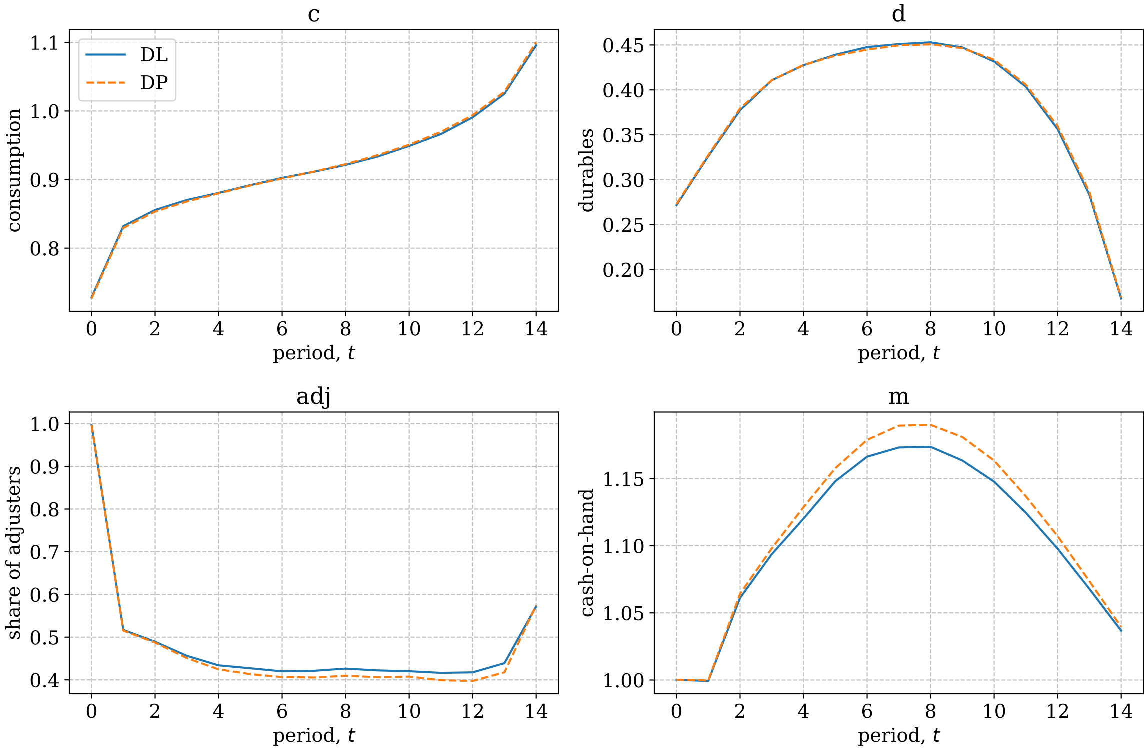


Figure 15: Non-Convex Durables Model: Life cycles profiles

Notes: Panel Each figure compares the DL solution (blue line) with the DP-solution (dashed orange line). Upper left shows average consumption. Upper right shows average durables stock. Lower left shows adjuster share. Lower right shows average cash-on-hand.

ercise this shows that it is possible to solve problems with non-convexities with DL using a version of the VPD algorithm.

8 Conclusion

TBD.

References

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

Azinovic, M. and Zemlicka, J. (2023). Intergenerational Consequences of Rare Disasters. Working Paper.

Bertsekas, D. (2012a). *Dynamic Programming and Optimal Control: Volume I*. Athena Scientific.

Bertsekas, D. (2012b). *Dynamic Programming and Optimal Control: Volume II; Approximate Dynamic Programming*. Athena Scientific.

Bertsekas, D. (2019). *Reinforcement Learning and Optimal Control*. Athena Scientific.

Brumm, J., Krause, C., Schaab, A., and Scheidegger, S. (2021). Sparse grids for dynamic economic models. *SSRN Electronic Journal*.

Brumm, J. and Scheidegger, S. (2017). Using Adaptive Sparse Grids to Solve High-Dimensional Dynamic Models. *Econometrica*, 85(5):1575–1612.

Carroll, C. D. (2006). The Method of Endogenous Gridpoints for Solving Dynamic Stochastic Optimization Problems. *Economics Letters*, 91(3):312–320.

Chen, M., Joseph, A., Kumhof, M., Pan, X., and Zhou, X. (2023). Deep Reinforcement Learning in a Monetary Model. arXiv:2104.09368 [econ, q-fin, stat].

Druedahl, J. (2021). A Guide on Solving Non-Convex Consumption-Saving Models. *Computational Economics*, 58(3):747–775.

Druedahl, J. and Jørgensen, T. H. (2017). A general endogenous grid method for multi-dimensional models with non-convexities and constraints. *Journal of Economic Dynamics and Control*, 74:87–107.

Duarte, V., Duarte, D., and Silva, D. H. (2024). Machine Learning for Continuous-Time Finance. *The Review of Financial Studies*, 37(11):3217–3271.

Duarte, V., Fonseca, J., Goodman, A. S., and Parker, J. A. (2022). Simple Allocation Rules and Optimal Portfolio Choice Over the Lifecycle. Working Paper 29559, National Bureau of Economic Research.

Duffy, J. and McNelis, P. D. (2001). Approximating and simulating the stochastic growth model: Parameterized expectations, neural networks, and the genetic algorithm. *Journal of Economic Dynamics and Control*, 25(9):1273–1303.

Ebrahimi Kahou, M., Fernández-Villaverde, J., Perla, J., and Sood, A. (2021). Exploiting Symmetry in High-Dimensional Dynamic Programming.

Fan, B., Qiao, E., Jiao, A., Gu, Z., Li, W., and Lu, L. (2024). Deep Learning for Solving and Estimating Dynamic Macro-finance Models. *Computational Economics*.

Fernández-Villaverde, J., Hurtado, S., and Nuño, G. (2023). Financial Frictions and the Wealth Distribution. *Econometrica*, 91(3):869–901.

Fernández-Villaverde, J., Marbet, J., Nuño, G., and Rachedi, O. (2024a). Inequality and the zero lower bound. *Journal of Econometrics*, page 105819.

Fernández-Villaverde, J., Nuño, G., and Perla, J. (2024b). Taming the Curse of Dimensionality: Quantitative Economics with Deep Learning. Technical Report w33117, National Bureau of Economic Research, Cambridge, MA.

Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods. arXiv:1802.09477 [cs, stat].

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Gopalakrishna, G. (2021). ALIENs and Continuous Time Economies. Working Paper.

Gopalakrishna, G., Gu, Z., and Payne, J. (2024). Asset Pricing, Participation Constraints, and Inequality. Working Paper.

Gu, Z., Laurière, M., Merkel, S., and Payne, J. (2023). Deep Learning Solutions to Master Equations for Continuous Time Heterogeneous Agent Macroeconomic Models. Working Paper.

Han, J. and E, W. (2016). Deep Learning Approximation for Stochastic Control Problems. arXiv:1611.07422 [cs, math, stat].

Han, J., Yang, Y., and E, W. (2024). DeepHAM: A Global Solution Method for Heterogeneous Agent Models with Aggregate Shocks. Working Paper.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

Huang, J. (2022). A Probabilistic Solution to High-Dimensional Continuous-Time Macro and Finance Models.

Huang, J. (2023). Breaking the Curse of Dimensionality in Heterogeneous-Agent Models: A Deep Learning-Based Probabilistic Approach.

Hull, I. (2015). Approximate dynamic programming with post-decision states as a solution method for dynamic economic models. *Journal of Economic Dynamics and Control*, 55:57–70.

Judd, K. L. (1998). *Numerical Methods in Economics*. MIT Press.

Judd, K. L., Maliar, L., and Maliar, S. (2017). How to Solve Dynamic Stochastic Models Computing Expectations Just Once. *Quantitative Economics*, 8(3).

Judd, K. L., Maliar, L., Maliar, S., and Valero, R. (2014). Smolyak method for solving dynamic economic models: Lagrange interpolation, anisotropic grid and adaptive domain. *Journal of Economic Dynamics and Control*, 44:92–123.

Kase, H., Melosi, L., and Rottner, M. (2024). Estimating Nonlinear Heterogeneous Agents Models with Neural Networks. Working Paper.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Lee, W., Yu, H., Rival, X., and Yang, H. (2020). On correctness of automatic differentiation for non-differentiable functions. *Advances in Neural Information Processing Systems*, 33:6719–6730.

Li, Y. (2018). Deep Reinforcement Learning: An Overview. *arXiv:1701.07274 [cs]*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2019). Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*.

Maliar, L. and Maliar, S. (2013). Envelope condition method versus endogenous grid method for solving dynamic programming problems. *Economics Letters*, 120(2):262–266.

Maliar, L. and Maliar, S. (2022). Deep learning classification: Modeling discrete labor choice. *Journal of Economic Dynamics and Control*, 135:104295.

Maliar, L., Maliar, S., and Winant, P. (2021). Deep learning for solving dynamic economic models. *Journal of Monetary Economics*, 122:76–101.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Nuno, G., Renner, P., and Scheidegger, S. (2024). Monetary Policy with Persistent Supply Shocks. Working Paper.

Pascal, J. (2024). Artificial neural networks to solve dynamic programming problems: A bias-corrected Monte Carlo operator. *Journal of Economic Dynamics and Control*, 162:104853.

Payne, J., Rebei, A., and Yang, Y. (2024). Deep Learning for Search and Matching Models. SSNN.

Powell, W. B. (2011). *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley, Hoboken, N.J, 2nd edition edition.

Powell, W. B. (2022). *Reinforcement Learning and Stochastic Optimization: A Unified Framework for Sequential Decisions*. Wiley. Google-Books-ID: CtCvzgEACAAJ.

Puterman, M. L. (2014). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Rust, J. (1996). Chapter 14 Numerical dynamic programming in economics. volume 1, pages 619–729. Elsevier.

Scheidegger, S. and Bilionis, I. (2019). Machine learning for high-dimensional dynamic stochastic economies. *Journal of Computational Science*, 33:68–82.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

Sehnke, F., Osendorfer, C., Rückstieß, T., Graves, A., Peters, J., and Schmidhuber, J. (2010). Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559.

Shi, R. A. (2022). Can an AI agent hit a moving target? arXiv:2110.02474 [cs, econ].

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Stokey, N. L. and Lucas, R. E. (1989). *Recursive methods in economic dynamics*. Harvard University Press.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition.

Valaitis, V. and Villa, A. T. (2024). A machine learning projection method for macro-finance models. *Quantitative Economics*, 15(1):145–173. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.3982/QE1403.

Van Roy, B., Bertsekas, D. P., Lee, Y., and Tsitsiklis, J. N. (1997). A neuro-dynamic programming approach to retailer inventory management. In *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, volume 4, pages 4052–4057. IEEE.

A Appendix: Algorithms

A.1 DeepV

The *DeepV* algorithm relies on approximating the regular value function with a neural network as $v\left(t, s_{t,i}; \theta_v\right)$, instead of the post-decision value function as in *DeepV-PD*. DeepV is explained in Algorithm A.1. The computation of the policy loss $L_\pi(\theta_\pi; \mathcal{B}^k)$ now includes the computationally costly expectation of the approximated next period value function. This was not necessary in *DeepV-PD*, where the neural network directly approximated the expected next period value $\overline{v}(t, \bar{s}_t, \theta_{\overline{v}})$.

A.2 DeepQ

To explain the relation of our *DeepV* and *DeepV-PD* algorithms to deep reinforcement learning literature, we compare them to a version of the »Deep Deterministic Policy Gradient (DDPG)« algorithm from Lillicrap et al. (2019) with a few changes. We call this *DeepQ*. The goal in *DeepQ* is to find the policy function by approximating an action-value function $Q(s_t, a_t; \theta_Q)$ as a neural network with parameters θ_Q alongside the approximated policy.

The Q-value relates to the state-value $v_t(s_t)$ by the relation:

$$v_t(s_t) = \max_{a_t} Q_t(s_t, a_t). \tag{69}$$

We can also write the Bellman equation as:

$$Q_t(s_t, a_t) = u(s_t, a_t) + \begin{cases} h(\bar{s}_t) & \text{if } t = T-1 \\ \beta \mathbb{E}_t[\max_{a_{t+1}} Q_{t+1}(s_{t+1}, a_{t+1})] & \text{else} \end{cases}. \tag{70}$$

To update the parameters in the action- value network, we need a *target*. We call the action-value and policy networks used to compute the target for the *target networks* and denote their parameters by $\check{\theta}_\pi$ and $\check{\theta}_\theta$. The action-value network must then be a good approximation of the right hand-side of (70). The parameters in the policy network, θ_π are updated to maximize the value-of-choice directly using the action-value function.

The full details are in Algorithm (A.2). DeepQ generally features fast updates of

Algorithm A.1 DeepV

Draw random initial parameter values, θ_v^{-1}.
Initialize $\check{\theta}_\pi^{-1} = \theta_\pi^{-1}$ and $\check{\theta}_v^{-1} = \theta_v^{-1}$.
Algorithm 2 with the updating in step 3 as:

1. Compute the value target

$$\begin{aligned}
\tilde{a}_{t,i} &= \pi(t, s_{t,i}; \check{\theta}_\pi^{k-1}) \\
\tilde{\bar{s}}_{t,i} &= \bar{\Gamma}_t(s_{t,i}, \tilde{a}_{t,i}) \\
\tilde{s}_{t+1,i} &= \Gamma_t(\tilde{\bar{s}}_{t,i}, \tilde{z}_{t+1,i}), \;\; \tilde{z}_{t+1,i} \sim F_{t+1} \\
\tilde{v}_{t,i} &= \begin{cases} u(s_{t,i}, \tilde{a}_{t,i}) + h(\tilde{\bar{s}}_{t,i}) & \text{if } t = T \\ u(s_{t,i}, \tilde{a}_{t,i}) + \beta \mathbb{E}_t \left[v\left(t+1, \tilde{s}_{t+1,i}; \check{\theta}_v^{k-1}\right) \right] & \text{else} \end{cases}
\end{aligned}$$

2. Update the value network to θ_v^k using the loss function

$$L_w\left(\theta_w; \mathcal{B}^k\right) = \frac{1}{\left|\mathcal{B}^k\right|} \sum_{i,t,s_{it} \in \mathcal{B}^k} \left(v\left(t, s_{t,i}; \theta_w\right) - \tilde{v}_{t,i}\right)^2 \tag{67}$$

   for up to $\#_v$ epochs, or until the value loss has not improved for Δ_v epochs.

   (a) Return $\theta_{\bar{v}}^k$ from the epoch with the lowest loss.

   (b) Update value target network: $\check{\theta}_v^k = \tau \theta_v^k + (1 - \tau) \check{\theta}_v^{k-1}$.

3. If $k > \underline{k}_\pi$:Update the policy network to θ_π^k

$$L_\pi(\theta_\pi; \mathcal{B}^k) = -\frac{1}{\left|\mathcal{B}^k\right|} \sum_{i,t,s_{t,i} \in \mathcal{B}^k} \left(u(s_{t,i}, \tilde{a}_{t,i}) + \tilde{\bar{v}}_{t,i} \right) \tag{68}$$

$$\begin{aligned}
\tilde{a}_{t,i} &= \pi(t, s_{t,i}; \theta_\pi) \\
\tilde{\bar{s}}_{t,i} &= \Gamma_t(s_{t,i}, \tilde{a}_{t,i}) \\
\tilde{\bar{v}}_{t,i} &= \begin{cases} h(\tilde{\bar{s}}_{t,i}) & \text{if } t = T - 1 \\ \beta \mathbb{E}_t \left[v(t+1, \tilde{s}_{t+1,i}; \theta_v^k \right] & \text{else} \end{cases} \\
&\text{if } t < T - 1 : \; \tilde{s}_{t+1,i} = \Gamma_t(\bar{s}_{t,i}, \tilde{z}_{t+1,i}), \;\; \tilde{z}_{t+1,i} \sim F_{t+1},
\end{aligned}$$

   for up to $\#_\pi$ epochs, or until the policy loss as not improved for Δ_π epochs.

   (a) Return θ_π^{k+1} from the epoch with the lowest loss.

   (b) Update target policy network $\check{\theta}_\pi^k = \tau \theta_\pi^k + (1 - \tau) \check{\theta}_\pi^{k-1}$.

both θ_Q and θ_π as the loss-function gradients are quite simple. Using the exploratory policy-function where we simulate with added noise in the actions is quite important when doing DeepQ. If we only relied on shocks for exploring the state-action space then we would have almost perfect correlation between the variation in state- and action-spaces. A problem with DeepQ is that the Q-functions quickly becomes a fairly complicated object to approximate precisely as it is considerably more high-dimensional than the state-value function $v_t(s_t)$ and post-decision-value function $\overline{v}_t\left(\bar{s}_t\right)$.

Model-free reinforcement learning. In model-free reinforcement learning the transition functions $\bar{\Gamma}(s_t, a_t)$ and $\Gamma(\bar{s}_t, z_{t+1})$ and the shock distribution F_{t+1} are unknown, but can still be evaluated, e.g. by making an actual real-time action. Using $s_{t+1,i}$ from the input data, the loss function is then written

$$
\begin{aligned}
L_Q(\theta_Q) &= -\frac{1}{NT}\sum_{t=0}^{T-2}\sum_{i=0}^{N-2}\left(Q(s_{t,i}, a_{t,i}) - \tilde{Q_{t,i}}\right)^2 \qquad (73) \\
\tilde{Q_{t,i}} &= u(s_{t,i}, a_{t,i}) + \begin{cases} h(\bar{s}_{t,i}) & \text{if } t = T-1 \\ \beta Q(t+1, s_{t+1,i}, \tilde{a}_{t+1,i}; \check{\theta}_Q) & \text{else} \end{cases} \\
\tilde{a}_{t+1,i} &= \pi(t+1, s_{t+1,i}; \check{\theta}_\pi),
\end{aligned}
$$

where the expectation $\mathbb{E}_t[Q(t+1, \tilde{s}_{t+1,i}, \tilde{a}_{t+1,i}; \check{\theta}_Q)]$ is replaced with $Q(t+1, s_{t+1,i}, \tilde{a}_{t+1,i}; \check{\theta}_Q)$. Instead of handling the expectation explicitly, the averages across T and N are now also doing a form of Monte Carlo integration of the expectation. In economics we typically assume the model structure is known and should thus use it to maximize the precision and speed of the chosen solution algorithm. There are some cases where one may want to maintain that the transitions functions are unknown. An example could be agent-based models such as Chen et al. (2023). Another case is learning in real-time learning as in Shi (2022).

A.3 Extending DeepV-PD with FOC multiple value networks

We now consider our combined extensions of the DeepV-PD algorithms.

Firstly, we assume that we instead of one value network have J value networks indexed by $j \in \{0, \ldots, J-1\}$, which are initialized independently and updated inde-

Algorithm A.2 DeepQ

Draw random initial parameter values, θ_Q^{-1}.
Initialize $\check{\theta}_\pi^{-1} = \theta_\pi^{-1}$ and $\check{\theta}_Q^{-1} = \theta_Q^{-1}$.
Algorithm 2 with the updating in step 3 as:

1. Compute action-value target for $t < T - 1$

$$\tilde{Q}_{t,i} = \begin{cases} u(s_{t,i}, a_{t,i}) & t = T - 2 \\ u(s_{t,i}, a_{t,i}) + \beta \mathbb{E}_t[Q(t+1, \tilde{s}_{t+1,i}, \tilde{a}_{t+1,i}; \theta_Q^{\text{target}})] & \text{else} \end{cases}$$

   where

$$\begin{aligned} \bar{s}_{t,i} &= \bar{\Gamma}(s_{t,i}, a_{t,i}) \\ \tilde{s}_{t+1,i} &= \Gamma_t(\bar{\tilde{s}}_{t,i}, \tilde{z}_{t+1,i}), \;\; \tilde{z}_{t+1,i} \sim F_{t+1} \\ \tilde{a}_{t+1,i} &= \pi(t+1, \tilde{s}_{t+1,i}; \check{\theta}_\pi^{k-1}) \end{aligned}$$

2. Update the value network to θ_Q^k using the loss function for $t < T - 1$

$$L_Q\left(\theta_Q; \mathcal{B}^k\right) = \frac{1}{|\mathcal{S}|} \sum_{i,t,s_{t,i},a_{t,i} \in \mathcal{S}} \left(Q\left(t, s_{t,i}, a_{t,i}; \check{\theta}_Q^{k-1}\right) - \tilde{Q}_{t,i} \right)^2 \tag{71}$$

   for up to $\#_Q$ epochs, or until the value loss has not improved for Δ_Q epochs.

   (a) Return θ_Q^k from the epoch with the lowest loss.

   (b) Update value target network: $\check{\theta}_Q^k = \tau\theta_Q^k + (1-\tau)\check{\theta}_Q^{k-1}$.

3. If $k > \underline{k}_\pi$: Update the policy network to θ_π^k

$$\begin{aligned} L_\pi(\theta_\pi; \mathcal{B}^k) &= -\frac{1}{|\mathcal{B}^k|} \sum_{i,t,s_{i,t} \in \mathcal{S}} Q(t, s_{ti}, \tilde{a}_{ti}) \qquad (72) \\ \tilde{a}_{t,i} &= \pi(t, s_{t,i}; \theta_\pi). \end{aligned}$$

   for up to $\#_\pi$ epochs, or until the policy loss as not improved for Δ_π epochs.

   (a) Return θ_π^{k+1} from the epoch with the lowest loss.

   (b) Update target policy network $\check{\theta}_\pi^k = \tau\theta_\pi^k + (1-\tau)\check{\theta}_\pi^{k-1}$.

pendently but using the same data.

Secondly, we assume that the first order conditions in vector form can be written on the form

$$g\left(s_t, a_t, \overline{q}_t\right) = 0$$

where $\overline{q}_t = \overline{q}(\overline{s}_t)$ contains the relevant post-decision marginal values. We define $\overline{q}$ implicitly from q and $\overline{q}_h$ as

$$\overline{q}(\overline{s}_t) = \begin{cases} \overline{q}_h(\overline{s}_t) & \text{if } t = T-1 \\ \mathbb{E}_t\left[q(t+1, s_{t+1}, a_{t+1})\right] & \text{else} \end{cases}$$

If using a KKT approach multipliers are considered as outputs of the policy network. Instead of approximating only the post-decision value function, we now approximate the combined post-decision value and marginal value function

$$\overline{vq}\left(t, \overline{s}_{t,i}; \theta_{\overline{vq},j}\right),$$

where we refer to the sub-outcomes as $\overline{v}\left(t, \overline{s}_{t,i}; \theta_{\overline{vq},j}\right)$ and $\overline{q}\left(t, \overline{s}_{t,i}; \theta_{\overline{vq},j}\right)$.

The details are in Algorithm (A.3). We introduce the positive weight parameters $\kappa_{\overline{q}} > 0$ and $\kappa_{\text{FOC}} > 0$, and the positive diagonal weight matrix Ω_{FOC}. If $J = 1$ and $\kappa_{\overline{q}} = \kappa_{\text{FOC}} \to 0$ we are back at the baseline DeepV-PD algorithm from Section (4).

Algorithm A.3 DeepVPD – FOC and multiple value networks

Draw random initial parameter values, $\theta_{\overline{vq},j}^{-1}$ for $j \in \{0, \dots J-1\}$.
Initialize $\check{\theta}_{\pi}^{-1} = \theta_{\pi}^{-1}$ and $\check{\theta}_{\overline{vq},j}^{-1} = \theta_{\overline{vq},j}^{-1}$.
Algorithm 2 with the updating in step 3 as:

1. Compute the value targets for $t < T-1$

$$\tilde{\overline{v}}_{t,i,j} = \mathbb{E}_t \left[u\left(\tilde{s}_{t+1,i}, \tilde{a}_{t+1,i}\right) + \begin{cases} h(\tilde{\overline{s}}_{t+1,i}) & \text{if } t = T-2 \\ \overline{v}(t+1, \tilde{\overline{s}}_{t+1,i}; \check{\theta}_{\overline{vq},j}^{k-1}) & \text{else} \end{cases} \right]$$
$$\tilde{\overline{q}}_{t,i,j} = \mathbb{E}_t \left[q(t+1, \tilde{s}_{t+1,i}, \tilde{a}_{t+1,i}, \check{\theta}_{\overline{vq},j}^{k-1}) \right]$$

   where

$$\begin{aligned} \tilde{s}_{t+1,i} =& \Gamma_t(\overline{s}_{t,i}, \tilde{z}_{t+1,i}), \;\; \tilde{z}_{t+1,i} \sim F_{t+1} \\ \tilde{a}_{t+1,i} =& \pi(t+1, \tilde{s}_{t+1,i}; \check{\theta}_{\pi}^{k-1}) \\ \tilde{\overline{s}}_{t+1,i} =& \overline{\Gamma}_{t+1}(\tilde{s}_{t+1,i}, \tilde{a}_{t+1,i}) \end{aligned}$$

2. Update the value networks to $\theta_{\overline{vq},j}^{k}$ using the loss function for $t < T-1$

$$L_{\overline{vq}}\left(\theta_{\overline{vq},j}; \mathcal{B}^k\right) = \frac{1}{|\mathcal{B}^k|} \sum_{i,t,\overline{s}_{it} \in \mathcal{B}^k} \left(d\tilde{\overline{v}}_{t,i,j}\right)^2 + \kappa_{\overline{q}} \left(d\tilde{\overline{q}}_{t,i,j}\right)^T \left(d\tilde{\overline{q}}_{t,i,j}\right) \tag{74}$$
$$d\tilde{\overline{v}}_{t,i,j} = \overline{v}\left(t, \overline{s}_{t,i}; \theta_{\overline{vq},j}\right) - \tilde{\overline{v}}_{t,i,j}$$
$$d\tilde{\overline{q}}_{t,i,j} = \overline{q}\left(t, \overline{s}_{t,i}; \theta_{\overline{vq},j}\right) - \tilde{\overline{q}}_{t,i,j}$$

   for up to $\#_{\overline{vq}}$ epochs, or until the value loss has not improved for $\Delta_{\overline{vq}}$ epochs.

   (a) Return $\theta_{\overline{vq},j}^{k}$ from the epoch with the lowest loss.

   (b) Update value target network: $\check{\theta}_{\overline{vq},j}^{k} = \tau\theta_{\overline{vq},j}^{k} + (1-\tau)\check{\theta}_{\overline{vq},j}^{k-1}$.

3. If $k > \underline{k}_{\pi}$: Update the policy network to θ_{π}^{k}

$$L_{\pi}(\theta_{\pi}; \mathcal{B}^k) = -\frac{1}{|\mathcal{B}^k|} \sum_{i,t,s_{t,i} \in \mathcal{B}^k} \text{value}_{t,i} + \kappa_{\text{FOC}} \text{FOC}_{t,i}^{T} \Omega_{\text{FOC}} \text{FOC}_{t,i} \tag{75}$$
$$\text{value}_{t,i} = u(s_{t,i}, \tilde{a}_{t,i}) + \tilde{\overline{v}}_{t,i}$$
$$\text{FOC}_{t,i} = g\left(s_{it}, a_{t,i}, \tilde{\overline{q}}_{t,i}\right)$$
$$\tilde{a}_{t,i} = \pi(t, s_{t,i}; \theta_{\pi})$$
$$\tilde{\overline{s}}_{t,i} = \overline{\Gamma}_t(s_{t,i}, \tilde{a}_{t,i}).$$
$$\tilde{\overline{v}}_{t,i} = \begin{cases} h(\tilde{\overline{s}}_{t,i}) & \text{if } t = T-1 \\ \beta \frac{1}{J} \sum_{j=0}^{J-1} \overline{v}\left(t, \tilde{\overline{s}}_{t,i}; \theta_{\overline{vq},j}^{k}\right) & \text{else} \end{cases}$$
$$\tilde{\overline{q}}_{t,i} = \begin{cases} \overline{q}_h(\tilde{\overline{s}}_{t,i}) & \text{if } t = T-1 \\ \frac{1}{J} \sum_{j=0}^{J-1} \overline{q}\left(t, \tilde{\overline{s}}_{t,i}; \theta_{\overline{vq},j}^{k}\right) & \text{else} \end{cases}$$

B Appendix: Implementation

B.1 Neural networks

This paper uses deep neural networks as function approximators of policy functions. A neural network is a function with parameters θ mapping N real inputs to M real outputs, $\mathbb{R}^N \to \mathbb{R}^M$.

The neural network is structured into a series of layers: An *input layer*, an *output layer* and a series of *hidden layers*. Each of these layers is firstly characterized by its number of *neurons*. In the input- and output layers the number of neurons equal N and M respectively. In the hidden-layers, the number of neurons is a hyperparameter chosen by the user, here called N_l for the l'th layer. Secondly, each layer has a non-linear *activation function* $\sigma_l : \mathbb{R} \to \mathbb{R}$. The typical choice of σ is the rectified linear units(ReLU):

$$\sigma(z) = \max(0, z)$$

The final activation function σ_{L+1} can be used to limit the output space of the neural net. If it is ReLU then the neural network can only output non-negative values.

The mapping of a neural network with L hidden layers where $X = [x_0, \dots x_{N-1}]$ is the input vector, is

$$\begin{aligned}
a_i^0 &= \sigma_0 \left(\sum_{j=0}^{N-1} w_{i,j}^0 x_j + b_i^0 \right) \quad i \in [0, \dots N-1] \\
a_i^1 &= \sigma_1 \left(\sum_{j=0}^{N_0^{neurons}-1} w_{i,j}^1 a_j^0 + b_i^1 \right) \quad i \in [0, \dots N_1^{neurons} - 1] \Big) \\
&\vdots \\
a_i^L &= \sigma_L \left(\sum_{j=0}^{N_0^{neurons}-1} w_{i,j}^1 a_j^0 + b_i^1 \right) \quad i \in [0, \dots N_L^{neurons} - 1] \Big) \\
y_i = a_i^{L+1} &= \sigma_{L+1} \Big(\sum_{j=0}^{M-1} w_{i,j}^L a_j + b_i^{L+1} \Big) \quad i \in [0, \dots M-1] \Big)
\end{aligned}$$

where $w_{i,j}^l$ are the weights in layer l mapping to neuron i and is coefficient to input j. b_i^l is a constant in layer l mapping to neuron i and is called a bias. Together

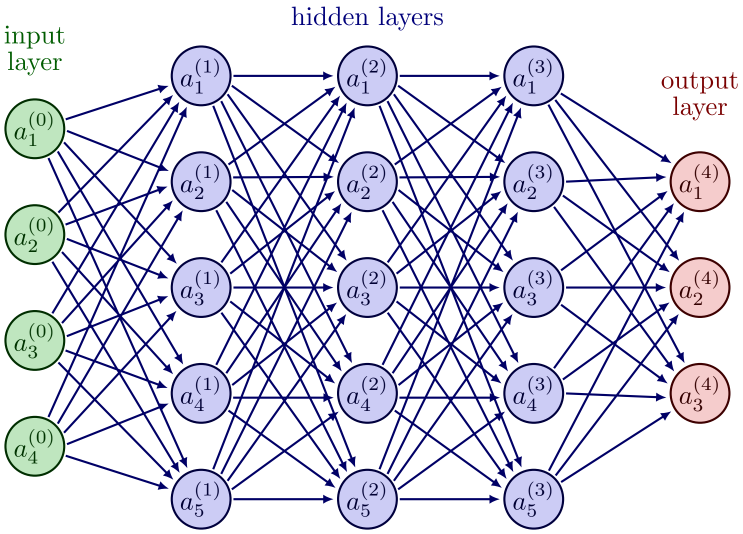


Figure B.1: Neural Network

$\theta = \left\{ \left\{ w_{i,j}^l \right\}, \left\{ b_i^l \right\} \right\}$. Graphically the neural network can be represented as in Figure (B.1), where each circle is a neuron and the arrows represents the weights.

Following Hornik et al. (1989) a number of universal function approximation theorems exist for the *feedforward neural network* structure used here. Results exists even for non-smooth and non-continuous functions (i.e. »local features«), which for example polynomials have problems fitting. It is also generally believed that oscillation problems when interpolating and extrapolating are smaller with neural networks than with polynomials.

B.2 PyTorch

We implement all algorithms in PyTorch. Pytorch is a library for deep learning that allows for easy computation on both GPUs and CPUs available in Python and C++. PyTorch is a very flexible library allowing for highly customized neural network architectures.

PyTorch features an automatic differentiation package 'autograd' which computes derivatives of the neural network parameters through back-propagation. The basic functionality of the autograd library is illustrated in Listing 1. Listing 2 shows how to setup a neural network. Finally, listing 3 show how to initialize the neural network and perform a single training step.

```python
import torch

def f(x):
    if x[0] > 1:
        return x**2
    else:
        return x

x = torch.tensor([2.0],requires_grad=True) # leave
y = f(x) # root
y.backward() # automatic differentiation
print(f’{x = }’) # -> tensor([2.0])
print(f’{x.grad = }’) # tensor([4.0])
```

Listing 1: Example: Automatic Differentiation

```python
class NN(torch.nn.Module):
    def __init__(self):

        super(NN, self).__init__()
        self.layers =  torch.nn.ModuleList(4)
        self.layers[0] = torch.nn.Linear(1,100)
        self.layers[1] = torch.nn.Linear(100,100)
        self.layers[2] = torch.nn.Linear(100,100)
        self.layers[3] = torch.nn.Linear(100,1)

    def forward(self,x):

        x = torch.F.relu(self.layers[0](x))
        x = torch.F.relu(self.layers[1](x))
        x = torch.F.relu(self.layers[2](x))
        return self.layers[3](x)
```

Listing 2: Example: Neural Network

```python
# a. setup
f_NN = NN()
opt = torch.optim.Adam(f_NN.parameters(),lr=1e-3)

# b. training step (given sample)
```

```
opt.zero_grad() # zero gradients
pred_y = f_NN(sample.x) # forward pass
loss = F.mse_loss(pred_y,sample.y) # calculate loss
loss.backward() # backward pass
opt.step() # optimization step
```

Listing 3: Example: Setup and update step

B.3 Hyperparameters

Table B.1: Baseline Hyperparameters

Variable Name	**Description**	**DeepFOC**	**DeepSimulate**	**DeepVPD**
Nneurons_policy	Neurons in the policy network	500 for all elements	500 for all elements	500 for all elements
policy_activation_intermediate	Activation function for policy network intermediate layers	relu	relu	relu
policy_activation_final	Activation function for policy network final layer	sigmoid	sigmoid	sigmoid
Nneurons_value	Neurons in the value network	-	-	500 for all elements
value_activation_intermediate	Activation function for value network intermediate layers	relu	relu	relu
N_value_NN	Number of value networks	-	-	-
learning_rate_policy	Initial learning rate for the policy network	0.001	0.001	0.001
learning_rate_policy_decay	Decay rate for policy learning rate	0.9999	1.0	0.9999
learning_rate_policy_min	Minimum learning rate for the policy network	1e-05	1e-05	1e-05
learning_rate_value	Initial learning rate for the value network	-	-	0.001
learning_rate_value_decay	Decay rate for value learning rate	-	-	0.9999
learning_rate_value_min	Minimum learning rate for the value network	-	-	1e-05
epsilon_sigma	Initial exploration noise, σ_ϵ	0.1 for all elements	-	0.1 for all elements
epsilon_sigma_decay	Decay rate for exploration noise	1.0	-	1.0
epsilon_sigma_min	Minimum exploration noise	0.0 for all elements	-	0.0 for all elements
do_exo_actions_periods	Periods with exogenous actions	-1	-1	-1
K	Maximum number of iterations before termination, K	100000	100000	100000
K_time	Maximum number of minutes before termination	60	60	60
sim_R_freq	Simulation frequency, Δ_R	10	50	10
Delta_transfer	Transfer tolerance for improvement	0.0001	0.0001	0.0001
Delta_time	Time tolerance for improvement	20	20	20
K_time_min	Minimum number of minutes before termination	inf	inf	inf
terminate_on_policy_loss	Terminate if policy loss is below tolerance	False	False	False
tol_policy_loss	Tolerance for policy loss	0.0001	0.0001	0.0001
N	Sample size, N^{train}	150	5000	150
buffer_memory	Size of replay buffer	8	1	8
batch_size	Batch size for training	150	-	150
epoch_termination	Epoch termination condition	True	-	True
Nepochs_policy	Number of epochs for policy network, $\#_\pi$	15	-	15
Delta_epoch_policy	Epoch increment for policy network, Δ_π	5	-	5
epoch_policy_min	Minimum epochs for policy network	5	-	5
Nepochs_value	Number of epochs for value network, $\#_{\overline{v}}$	-	-	50
Delta_epoch_value	Number of epochs with no improvement before termination	-	-	10
epoch_value_min	Minimum epochs for value network	-	-	10
start_train_policy	Epoch to start training policy network	-1	-	10
tau	Target smoothing coefficient, τ	-	-	0.2
use_target_policy	Use target policy network	-	-	True
use_target_value	Use target value network	-	-	True
clip_grad_policy	Gradient clipping for policy network	100.0	100.0	100.0
clip_grad_value	Gradient clipping for value network	-	-	100.0
use_input_scaling	Use input scaling	False	False	False
scale_vec_states	State scaling vector for states	-	-	-
scale_vec_states_pd	State scaling vector for post decesion states	-	-	-
min_actions	Minimum action values	-	-	-
max_actions	Maximum action values	-	-	-
dtype	Data type	torch.float32	torch.float32	torch.float32
use_FOC	Use analytical First Order Conditions	-	-	False
Nquad	Number of quadrature points	16	16	16
Ngpus	Number of GPUs used	1	1	1

Notes: Shows the chosen baseline hyperparameters for each of the deep learning solution algorithms.

C Appendix: Buffer-Stock Model

C.1 Calibration

The calibration is:

- **Preferences:** $\beta = \frac{1}{1.01}$
- **Income process:** $\sigma_\psi = 0.1$,$\sigma_\xi = 0.1$,$\rho_p = 0.95$, κ_t in Figure C.1
- **Assets market:** $r = 0.01$
- **Initial states:** $\mu_{s0} = 1$,$\sigma_{s0} = 0.1$, $\mu_{p0=1}$=1, $\sigma_{p0} = 0.1$
- **Time:** $T = 55$, $T^{\text{retired}} = 45$

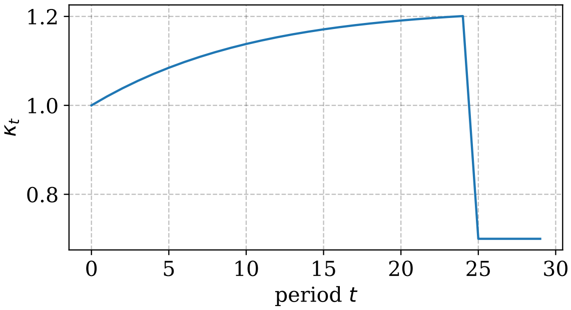


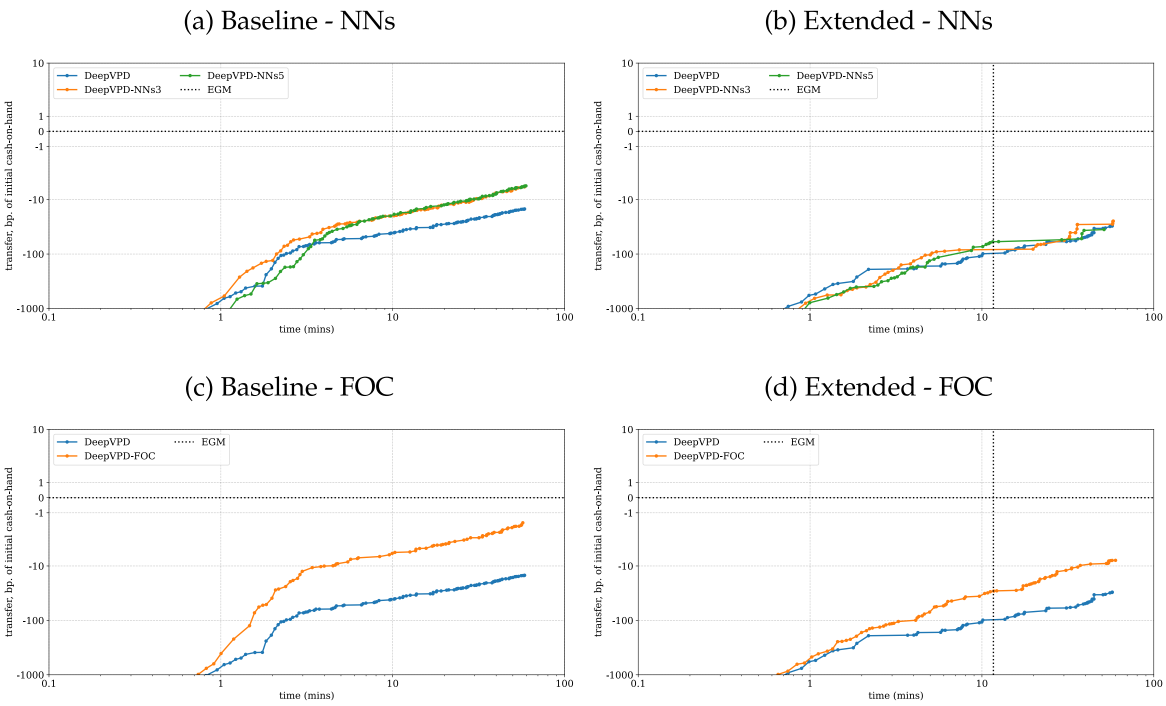
Figure C.1: Life-cycle component of income, κ_t

Notes: Shows how income absent any shocks evolves over the life-cycle

C.2 Extended model

The calibration is

- $\sigma_{\psi,low} = 0.05$, $\sigma_{\xi,high} = 0.15$
- $\sigma_{\xi,low} = 0.05$, $\sigma_{\xi,high} = 0.15$
- $\rho_{p,low} = 0.7$, $\rho_{p,high} = 0.99$


(a) Baseline - NNs

(b) Extended - NNs

(c) Baseline - FOC

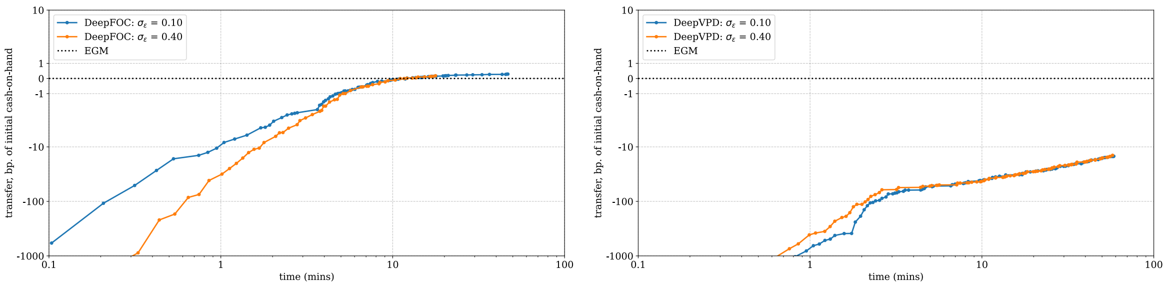
(d) Extended - FOC

Figure C.2: Buffer-Stock Model: Speed and accuracy with DeepV-PD extensions

Notes: See Figure 1.

C.3 Dynamic programming solution

This section describes our approach to solve the buffer-stock model using standard dynamic programming methods. We use the Endogenous Grid Method from Carroll (2006) modified to interpolate the policy back on an exogenous cash-in-hand grid. We rely on linear interpolation on monotone fixed grids. We do not have equidistant grid points as for some grids (cash-in-hand for example) we use more points towards the bottom of the grid. The solution algorithm is written in C++. To speed up the algorithm, we use parallelization across 72 threads.

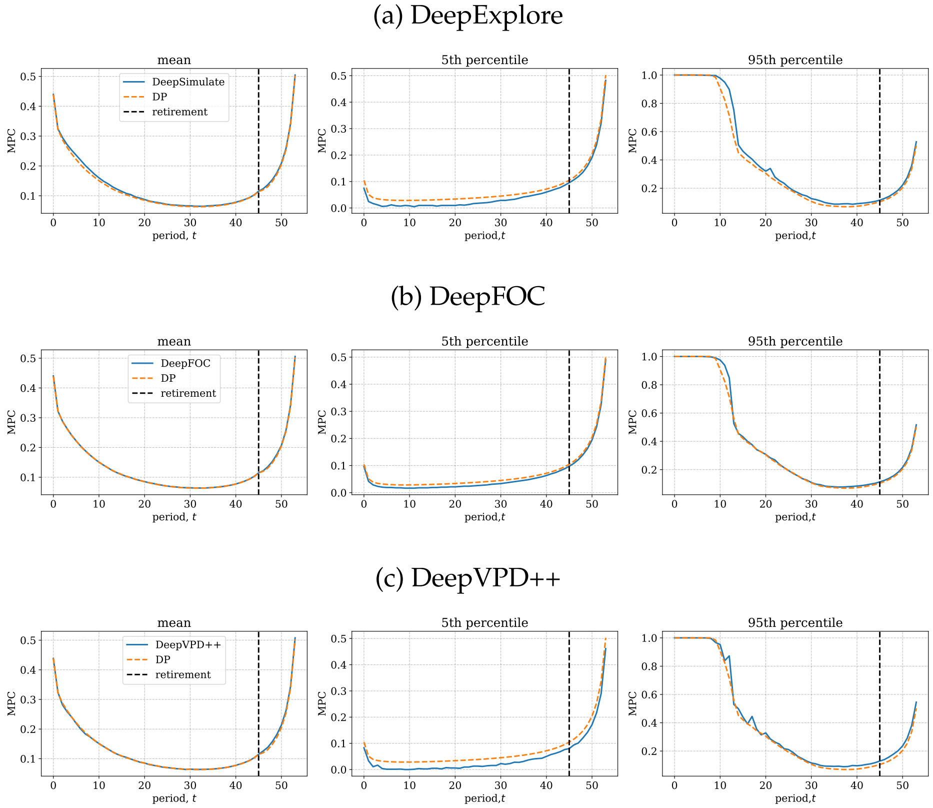


(a) Baseline: DeepFOC

(b) Extended: DeepFOC

Figure C.3: Buffer-stock model: Convergence with more exploration

Notes: See Figure 1.



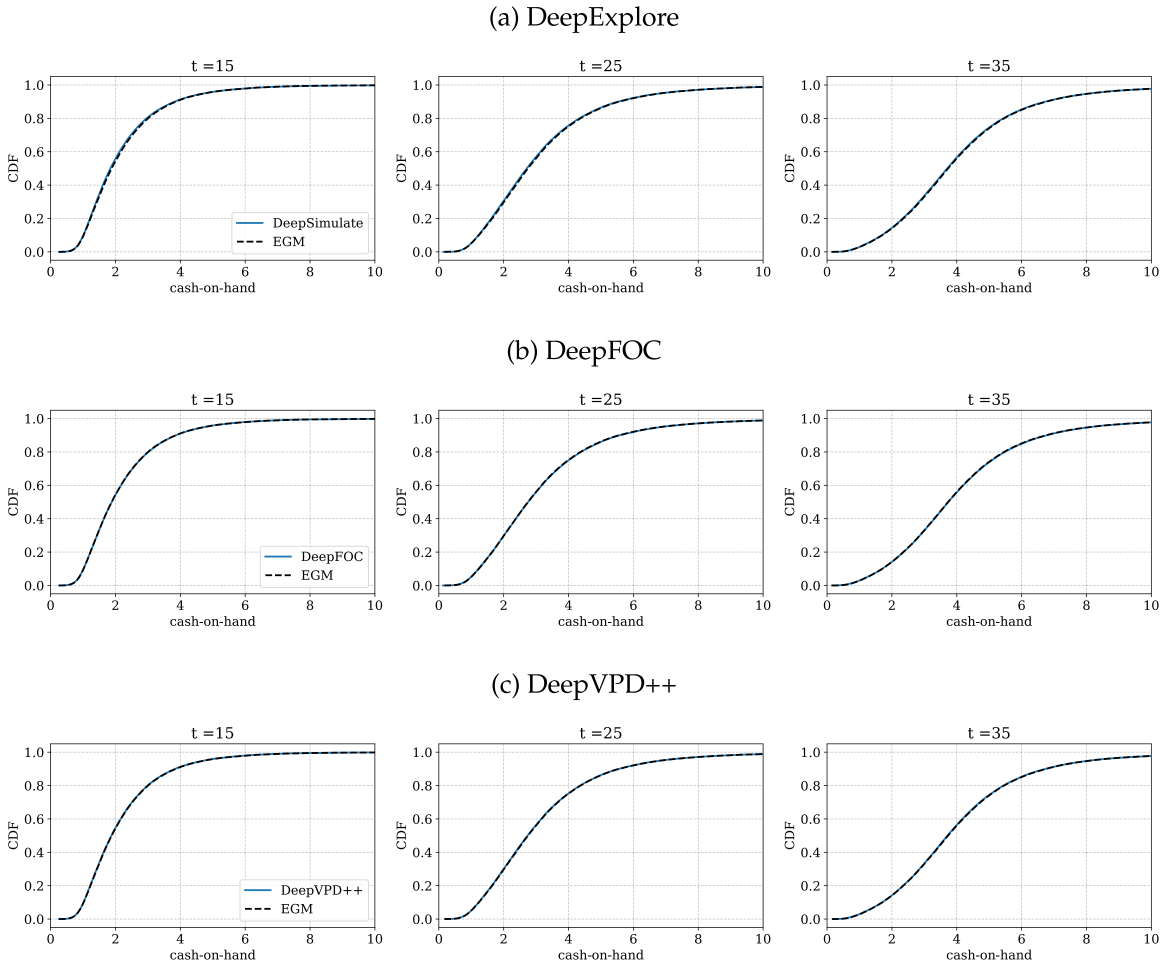
(a) DeepExplore

(b) DeepFOC

(c) DeepVPD++

Figure C.4: Buffer-stock model: Life-cycle profile of MPC (extended)

Notes: Shows mean MPC over the life-cycle and MPC for 5th and 95th percentiles for different DL algorithms compared to EGM. Last period of life is not included.



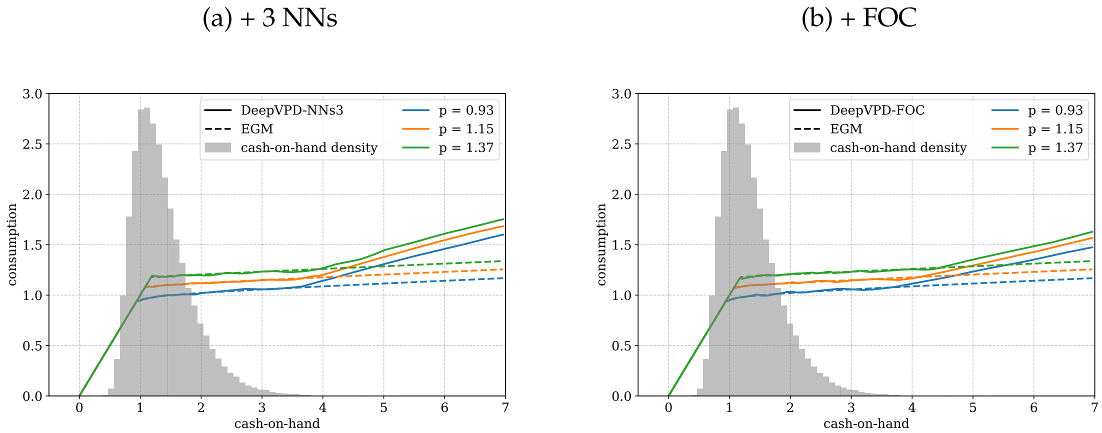
(a) DeepExplore

(b) DeepFOC

(c) DeepVPD++

Figure C.5: Buffer-stock model: Life-cycle profile of CDF of cash-on-hand (extended)

Notes: Shows the CDF of cash-on-hand at different points in the life-cycle


(a) + 3 NNs

(b) + FOC

Figure C.6: Buffer-stock model: Consumption Function (DeepVPD extensions)

Notes: See Figure 2

C.4 More results

Table C.6: Buffer-Stock Model – Baseline: DeepV-PD Extensions.

	EGM	Baseline	+NNs3	+FOC	+FOC+NNs3
Life-time reward, R	2.0616	2.0600	2.0610	2.0615	2.0616
Transfer (bp.)	0	-15	-6	-2	-1
Mean log10 Euler error	-4.5	-2.4	-2.6	-2.8	-3.0
Total time (m)	0.01	60.01	60.01	60.01	60.01
update NN	-	71.7%	79.5%	74.3%	79.9%
simulation: training sample	-	10.1%	6.3%	9.6%	6.2%
simulation: out-of-sample R	-	9.1%	7.3%	7.6%	6.8%
Iterations	1	12062	9222	12079	9128
Avg. policy epochs	-	15.0	14.8	14.7	14.5
Avg. value epochs	-	19.8	58.0	18.0	53.0

Notes: Shows extensions of the DeepV-PD algorithm.

Table C.6: Buffer-Stock Model – Extended: DeepV-PD Extensions.

	EGM	Baseline	+NNs3	+FOC	+FOC+NNs3
Life-time reward, *R*	2.0616	2.0600	2.0610	2.0615	2.0616
Transfer (bp.)	0	-15	-6	-2	-1
Mean log10 Euler error	-4.5	-2.4	-2.6	-2.8	-3.0
Total time (m)	0.01	60.01	60.01	60.01	60.01
update NN	-	71.7%	79.5%	74.3%	79.9%
simulation: training sample	-	10.1%	6.3%	9.6%	6.2%
simulation: out-of-sample R	-	9.1%	7.3%	7.6%	6.8%
Iterations	1	12062	9222	12079	9128
Avg. policy epochs	-	15.0	14.8	14.7	14.5
Avg. value epochs	-	19.8	58.0	18.0	53.0

Notes: Shows extensions of the DeepV-PD algorithm.

D Appendix: Multiple Durable Goods

D.1 Calibration

- **Time:** $T = 30$, $T^{\textbf{retired}} = 25$.
- **Preferences:** $\beta = \frac{1}{1.01}, \rho = 2.0$, $d_j = 1.01 \quad \forall j \in \{0, \ldots, D-1\}$, $\omega_j = \frac{0.2}{D}$. Our choice of ω_j means that regardless of D the utility weight on non-durable consumption c_t is 0.8.
- **Income process:** $\sigma_\psi = 0.1, \sigma_\xi = 0.1, \rho_p = 0.95$, κ_t in Figure D.1.
- **Assets market:** $R = 1.01$, $\tau = 0.1$. The depreciation rates are:

$$
\begin{aligned}
D = 1 &: \delta_j \in \{0.2\} \\
D = 2 &: \delta_j \in \{0.15, 0.2\} \\
D = 3 &: \delta_j \in \{0.1, 0.15, 0.2\} \\
D = 8 &: \delta_j \in \{0.1, 0.15, 0.19, 0.2, 0.22, 0.25, 0.27, 0.3\}.
\end{aligned}
$$

- **Initial states:** $\mu_{s0} = 1$, $\sigma_{s0} = 0.1$, μ_{p0}=1, $\sigma_{p0} = 0.1$, $\mu_{n0,j} = 0.1$, $\sigma_{n0,j} = 0.01$.

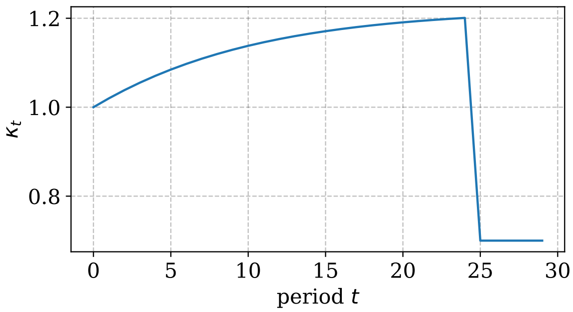


Figure D.1: Life-cycle component of income, κ_t

Notes: Shows how income absent any shocks evolves over the life-cycle

D.2 DeepFOC: Derivation of FOCs

In this section, we derive the FOCs we use when solving the model with multiple durable goods using DeepFOC. We focus on the case with only one durable good to keep notation simple, but extending the derivation to multiple durable goods is straightforward.

The Bellman equation for the model is

$$
\begin{aligned}
v_t(m_t, p_t, n_t) &= \max_{c_t, d_t} u(c_t, d_t) + \beta \mathbb{E}_t[v_{t+1}(m_{t+1}, p_{t+1}, n_{t+1})] \\
&\text{s.t} \\
\Delta_t &= d_t - n_t \\
b_t &= m_t - \Delta_t - \Lambda(\Delta_t) - c_t \\
m_{t+1} &= Rb_t + y_{t+1}(p_{t+1}, \psi_{t+1}) \\
p_{t+1} &= p_t^{\rho_p} \xi_{t+1} \\
n_{t+1} &= (1-\delta)d_t \\
b_t &\geq 0 \\
\Delta_t &\geq 0.
\end{aligned}
$$

We can rewrite this as a KKT-problem with two complementary slackness constraints:

$$
\begin{aligned}
v_t(m_t, p_t, n_t) &= \max_{c_t, d_t, \lambda_t, \mu_t} u(c_t, d_t) + \beta \mathbb{E}_t[v_{t+1}(m_{t+1}, p_{t+1}, n_{t+1})] + \lambda_t b_t + \mu_t \Delta_t \\
&\text{s.t.} \\
\Delta_t &= d_t - n_t \\
b_t &= m_t - \Delta_t - \Lambda(\Delta_t) - c_t \\
m_{t+1} &= Rb_t + y_{t+1}(p_{t+1}, \psi_{t+1}) \\
p_{t+1} &= p_t^{\rho_p} \xi_{t+1} \\
n_{t+1} &= (1-\delta)d_t \\
\lambda_t b_t &= 0 \\
\mu_t \Delta_t &= 0.
\end{aligned}
$$

D.2.1 Consumption

The FOC wrt. to c_t is

$$\begin{aligned}
0 &= u'_c(c_t, d_t) + \beta \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial m_{t+1}} \frac{\partial m_{t+1}}{\partial b_t} \frac{\partial b_t}{\partial c_t} \right] + \lambda_t \frac{\partial b_t}{\partial c_t} \\
&= u'_c(c_t, d_t) - \beta R \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial m_{t+1}} \right] - \lambda_t. \qquad (76)
\end{aligned}$$

The envelope condition entails

$$\begin{aligned}
\frac{\partial v_t(m_t, p_t, n_t)}{\partial m_t} &= \beta \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial m_{t+1}} \frac{\partial m_{t+1}}{\partial m_t} \right] + \lambda_t \\
&= \beta R \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial m_{t+1}} \right] + \lambda_t.
\end{aligned}$$

Combining (76) with this gives

$$\frac{\partial v_t}{\partial m_t} = u'_c(c_t, d_t). \qquad (77)$$

Inserting this into (76) gives

$$u'_c(c_t, d_t) = \beta R \mathbb{E}_t \left[u'_c(c_{t+1}, d_{t+1}) \right] + \lambda_t. \qquad (78)$$

D.2.2 Durable good

The FOC wrt. to d_t is

$$\begin{aligned}
0 &= u'_d(c_t, d_t) \\
&\quad + \beta \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial m_{t+1}} \frac{\partial m_{t+1}}{\partial b_t} \frac{\partial b_t}{\partial d_t} \right] \\
&\quad + \beta \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial n_{t+1}} \frac{\partial n_{t+1}}{\partial d_t} \right] \\
&\quad + \lambda_t \frac{\partial b_t}{\partial d_t} + \mu_t \\
&= u'_d(c_t, d_t) \\
&\quad - \beta R \left(1 + \Lambda'(\Delta_t)\right) \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial m_{t+1}} \right] \\
&\quad + \beta(1-\delta) \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial n_{t+1}} \right] \\
&\quad - \left(1 + \Lambda'(\Delta_t)\right) \lambda_t + \mu_t.
\end{aligned} \tag{79}$$

The envelope condition entails

$$\begin{aligned}
\frac{\partial v_t}{\partial n_t} &= \beta \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial m_{t+1}} \frac{\partial m_{t+1}}{\partial b_t} \frac{\partial b_t}{\partial n_t} \right] + \lambda_t \frac{\partial b_t}{\partial n_t} - \mu_t \\
&= \beta R \left(1 + \Lambda'(\Delta_t)\right) \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial m_{t+1}} \right] + (1 + \Lambda'(\Delta_t))\lambda_t - \mu_t.
\end{aligned} \tag{80}$$

Combining this with (77) and then (78) gives

$$\begin{aligned}
\frac{\partial v_t}{\partial n_t} &= \left(1 + \Lambda'(\Delta_t)\right) \left(\beta R \mathbb{E}_t \Big[u'_c(c_{t+1}, d_{t+1}) \Big] + \lambda_t \Big) - \mu_t \right] \\
&= \left(1 + \Lambda'(\Delta_t)\right) u'_c(c_t, d_t) - \mu_t
\end{aligned} \tag{81}$$

Inserting this and (77) into (79) gives

$$\begin{aligned}
u'_d(c_t, d_t) &= \left(1 + \Lambda'(\Delta_t)\right) (\beta R \mathbb{E}_t[u'_c(c_{t+1}, d_{t+1})] + \lambda_t) \\
&\quad - \beta(1-\delta) \mathbb{E}_t[\left(1 + \Lambda'(\Delta_t)\right) u'(c_{t+1}, d_{t+1})] \\
&\quad + \beta(1-\delta) \mathbb{E}_t[\mu_{t+1}] - \mu_t.
\end{aligned} \tag{82}$$

Inserting 78 into this gives:

$$
\begin{aligned}
u_d'(c_t, d_t) = &\left(1 + \Lambda'(\Delta_t)\right) u_c'(c_t, d_t) \\
&- \beta(1-\delta)\mathbb{E}_t[(1 + \Lambda'(\Delta_{t+1}))u_c'(c_{t+1}, d_{t+1})] \\
&+ \beta(1-\delta)\mathbb{E}_t[\mu_{t+1}] - \mu_t.
\end{aligned} \tag{83}
$$

Let us briefly consider the interpretation of (83).

1. The left hand side is simply marginal utility of the durable, $u_d'(c_t, d_t)$.
2. The first term $\left(1 + \Lambda'(\Delta_t)\right) u_c'(c_t, d_t)$ states that if we disregarded the future ($\beta = 0$) then we would equalize the marginal rate of substitution with relative prices as usual.
3. The second term $-\beta(1-\delta)\mathbb{E}_t[(1 + \Lambda'(\Delta_{t+1}))\, u_c'(c_{t+1}, d_{t+1})]$ captures that investment into the durable today persists into the next-period allowing for further consumption at $t+1$ giving an incentive for investment beyond simply equalizing marginal rates of substitution with relative prices at t.
4. The third term $\beta(1-\delta)\mathbb{E}_t[\mu_{t+1}]$ captures that by investing today you tighten the non-negativity constraint on investment at $t+1$ causing you to reduce investment in the durable good.
5. Finally, the term $-\mu_t$ captures the value of relaxing the constraint at time t.

D.3 DeepVPD with FOC

We can write the Bellman-equation in terms of the post-decision value function as

$$
\begin{aligned}
v_t(m_t, p_t, n_t) = &\max_{c_t, d_t, \lambda_t, \mu_t} u(c_t, d_t) + \beta \bar{v}_t(\bar{m}_t, p_t, d_t) + \lambda_t \bar{m}_t + \mu_t \Delta_t \\
&\text{s.t.} \\
&\bar{m}_t = m_t - \Delta_t - \Lambda(\Delta_t) - c_t \\
&\Delta_t = d_t - n_t \\
\lambda_t \bar{m}_t &= 0 \\
\mu_t \Delta_t &= 0,
\end{aligned} \tag{84}
$$

where the post-decision value function is

$$\begin{aligned} \bar{v}_t(\bar{m}_t, p_t, d_t) &= \mathbb{E}_t[v_{t+1}(m_{t+1}, p_{t+1}, n_{t+1})] \\ m_{t+1} &= R\bar{m}_t + y_{t+1}(p_{t+1}, \psi_{t+1}) \\ p_{t+1} &= p_t^{\rho_p} \xi_{t+1} \\ n_{t+1} &= (1-\delta)d_t. \end{aligned} \tag{85}$$

The first order conditions can be written:

$$u_c'(c_t, d_t) = \beta \frac{\partial \bar{v}_t}{\partial \bar{m}_t} + \lambda_t \tag{86}$$

$$u_d'(c_t, d_t) = \left(1 + \Lambda'(\Delta_t)\right) \beta \frac{\partial \bar{v}_t}{\partial \bar{m}_t} - \beta \frac{\partial \bar{v}_t}{\partial d_t} + \left(1 + \Lambda'(\Delta_t)\right) \lambda_t - \mu_t = \left(1 + \Lambda'(\Delta_t)\right) u_c'(c_t, d_t) - \beta \frac{\partial \bar{v}_t}{\partial d_t} - \mu_t. \tag{87}$$

The post-decision value function can also be written as

$$\bar{v}_t(\bar{m}_t, p_t, d_t) = \mathbb{E}_t \left[u(c_{t+1}, d_{t+1}) + \begin{cases} 0 & \text{if } t+1 = T-1 \\ \beta \bar{v}_{t+1}(\bar{m}_{t+1}, p_{t+1}, d_{t+1}) & \text{else} \end{cases} \right]. \tag{88}$$

In the *DeepVPD with FOCs* algorithm, we calculate parts of the RHS and then train a value network to approximate this. It is straightforward to do same for the derivative $\partial \bar{v}_t / \partial \bar{m}_t$ as

$$\frac{\partial \bar{v}_t}{\partial \bar{m}_t} = \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial m_{t+1}} \frac{\partial m_{t+1}}{\partial \bar{m}_t} \right] = R\mathbb{E}_t \left[u_c'(c_{t+1}, d_{t+1}) \right], \tag{89}$$

where we are equation (77). It is a bit more complicated with $\partial \bar{v}_t / \partial d_t$. We have

$$\begin{aligned} \frac{\partial \bar{v}_t}{\partial d_t} &= \mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial n_{t+1}} \frac{\partial n_{t+1}}{\partial d_t} \right] \\ &= (1-\delta)\mathbb{E}_t \left[\frac{\partial v_{t+1}}{\partial n_{t+1}} \right] \\ &= (1-\delta)E_t \Big[\left(1 + \Lambda'(\Delta_{t+1})\right) \beta \frac{\partial \bar{v}_{t+1}}{\partial \bar{m}_{t+1}} + \lambda_{t+1} \left(1 + \Lambda'(\Delta_{t+1})\right) - \mu_{t+1} \Big] = (1-\delta)E_t \Big[\left(1 + \Lambda'(\Delta_{t+1})\right) \end{aligned} \tag{90}$$

where we are using the envelope condition in (80). We value network now has $2 + D$ outputs. The first output is for the post-decision value itself, the second output is $\frac{\partial \bar{v}_t}{\partial \bar{m}_t}$ and the remaining D outputs are for $\frac{\partial \bar{v}_t}{\partial d_t}$ (one for each durable).
The loss-function for the policy network then is

$$
\begin{aligned}
L = & -\left(u(c_t, d_t) + \beta \bar{v}_t(\bar{m}_t, d_t, p_t)\right. \\
& + \left(u_c'(c_t, d_t) - \beta \frac{\partial \bar{v}_t}{\partial \bar{m}_t} - \lambda_t\right)^2 + \lambda_t \bar{m}_t \\
& + \left(u_d'(c_t, d_t) - \left(\left(1 + \Lambda'(\Delta_t)\right) u_c'(c_t, d_t) - \beta \frac{\partial \bar{v}_t}{\partial d_t} - \mu_t\right)\right) + \mu_t \Delta_t.
\end{aligned}
$$

D.4 More results

Table D.1: Durables Model: Dynamic Programming Solutions.

	1D-	1D	1D+	2D-	2D	2D+	3D-
R	-31.6505	-31.6429	-31.6423	-35.1661	-35.1576	-35.1569	-36.7986
Euler-error	-2.31	-2.37	-2.38	-2.33	-2.37	-2.37	-2.32
Time (m)	0.06	0.26	0.50	3.65	46.08	142.29	540.18
Nn	50	100	150	50	100	150	50
Nm	50	100	150	50	100	150	50
Np	50	100	100	50	100	100	50
Nm-pd	300	300	300	300	300	300	300
Nm (keep)	300	300	300	300	300	300	300
GB	0.58	1.68	3.22	10.43	151.73	502.73	570.65

Notes: Shows summary of results for Nested Endogenous Grid Method (NEGM) for the model with multiple durable goods.

Table D.1: Durables model: Summary (1D).

	NEGM+	DeepSimulate	DeepFOC	DeepVPD+
Life-time reward, R	-31.6423	-31.6418	-31.6417	-31.6430
Transfer (bp.)	0	3	3	-3
Euler-error	-2.38	-3.14	-3.78	-2.69
Total time (m)	0.50	120.00	120.01	120.01
update NN	-	91.1%	82.7%	88.3%
simulation: training sample	-	5.4%	4.8%	3.4%
simulation: out-of-sample R	-	1.0%	4.9%	2.6%
Iterations	-	35008	17964	14595
Avg. policy epochs	-	1.0	13.4	13.0
Avg. value epochs	-	0.0	0.0	140.1

Notes: Shows summary of results for the model with 1 durable good.

Table D.1: Durables model: Summary (2D).

	NEGM+	DeepSimulate	DeepFOC	DeepVPD+
Life-time reward, R	-35.1569	-35.1556	-35.1555	-35.1572
Transfer (bp.)	0	6	7	-1
Euler-error	-2.37	-3.13	-3.67	-2.83
Total time (m)	142.29	120.01	120.01	120.01
update NN	-	95.1%	83.2%	88.1%
simulation: training sample	-	1.4%	4.7%	3.9%
simulation: out-of-sample R	-	1.4%	6.2%	3.5%
Iterations	-	34716	15659	14372
Avg. policy epochs	-	1.0	13.3	11.7
Avg. value epochs	-	0.0	0.0	141.4

Notes: Shows summary of results for the model with 2 durable goods.

Table D.1: Durables model: Summary (3D).

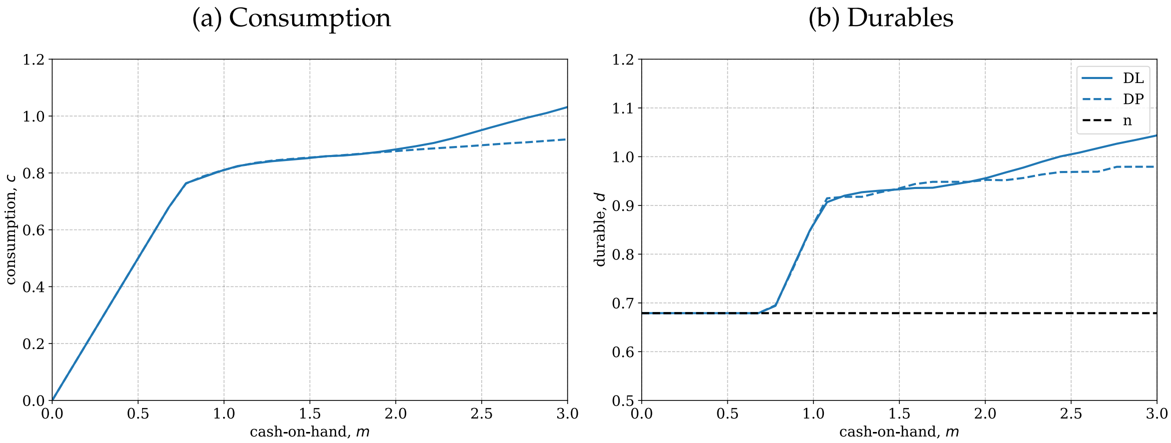
	NEGM-	DeepSimulate	DeepFOC	DeepVPD+
Life-time reward, R	-36.7986	-36.7822	-36.7822	-36.7878
Transfer (bp.)	0	76	76	50
Euler-error	-2.32	-3.11	-3.55	-2.78
Total time (m)	540.18	120.00	120.01	120.01
update NN	-	94.6%	77.4%	86.4%
simulation: training sample	-	1.3%	9.4%	4.1%
simulation: out-of-sample R	-	1.9%	6.9%	4.8%
Iterations	-	34441	13397	13856
Avg. policy epochs	-	1.0	13.1	13.5
Avg. value epochs	-	0.0	0.0	140.1

Notes: Shows summary of results for the model with 3 durable goods.

Table D.1: Durables model: Summary (8D).

	NEGM	DeepSimulate	DeepFOC	DeepVPD+
Life-time reward, R	-	-45.4138	-45.4144	-45.4202
Transfer (bp.)	-	-	-	-
Euler-error	-	-3.08	-3.42	-2.73
Total time (m)	-	240.01	240.01	240.01
update NN	-	96.2%	74.3%	80.1%
simulation: training sample	-	1.0%	14.1%	14.3%
simulation: out-of-sample R	-	1.6%	9.1%	3.0%
Iterations	-	46928	15739	23860
Avg. policy epochs	-	1.0	13.4	15.0
Avg. value epochs	-	0.0	0.0	109.5

Notes: Shows summary of results for the model with 8 durable goods.


(a) Consumption

(b) Durables

Figure D.2: Durables Model (1 durable): Policy functions

Notes: Shows policy functions for the one durable model obtained from DP and DL

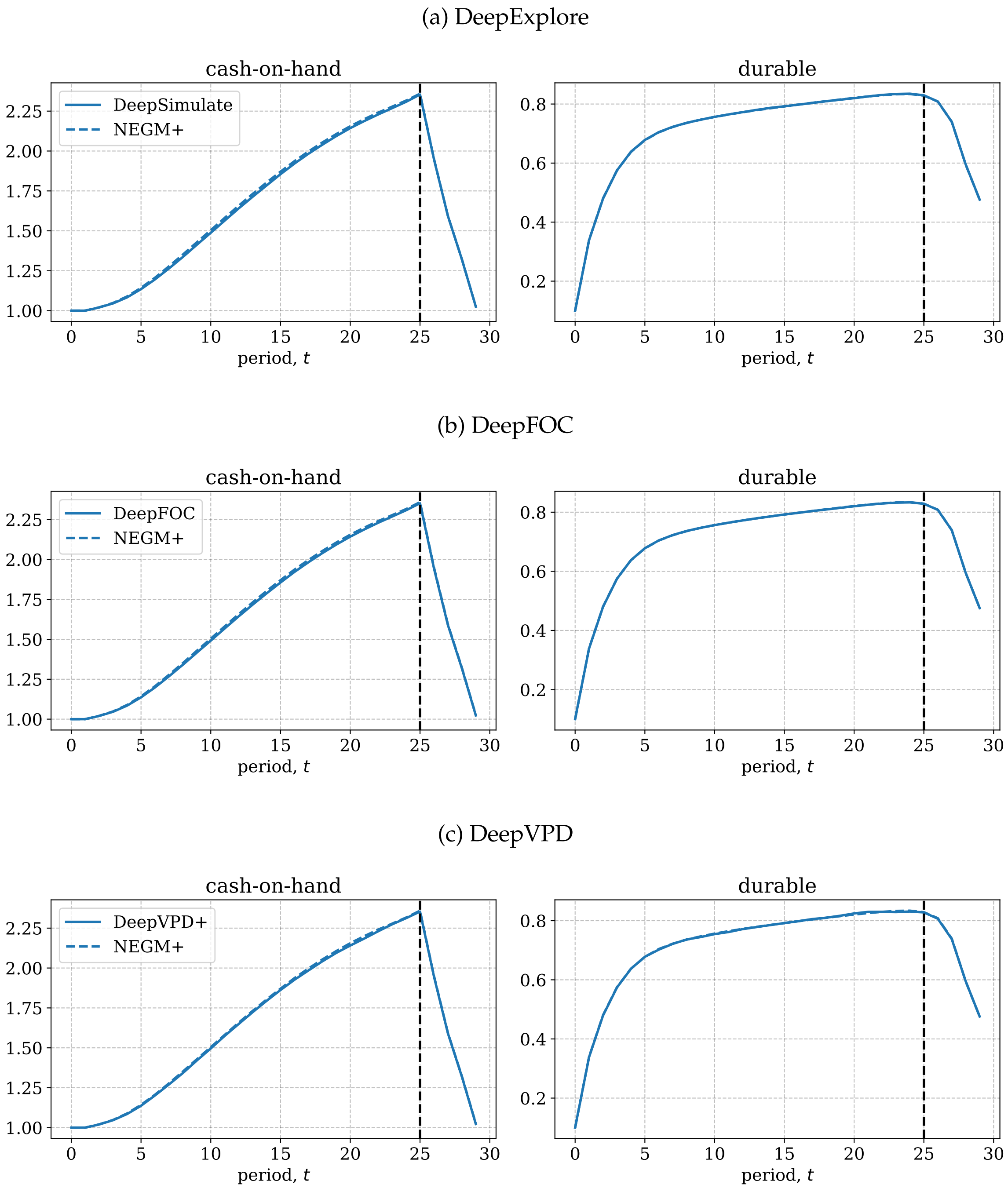
E Appendix: Non-Convex Durable Model

E.1 Calibration

- **Time:** $T = 15$,
- **Preferences:** $\beta = 0.965$,$\alpha = 0.9$, $\underline{\mathrm{d}} = 0.01$, $\rho = 2$.
- **Income process:** $\sigma_\psi = 0.1$,$\sigma_\xi = 0.1$,$\rho_p = 0.95$.
- **Assets market:** $r = 0.03$, $\kappa = 0.1$, $\delta = 0.15$.
- **Initial states:** $\mu_{s0} = 1$, $\sigma_{s0} = 0.1$, μ_{p0}=1, $\sigma_{p0} = 0.1$, $\mu_{n0} = 0.0$, $\sigma_{n0} = 0.01$.
- **Taste shocks:** $\sigma_\epsilon = 0.1$

E.2 Dynamic programming solution: details

The DP-solution follows roughly the same format as the DL-solution with the exception that we do not solve the keeper and adjuster problems simultaneously. The algorithm is a standard backwards induction loop with the following elements in each period:

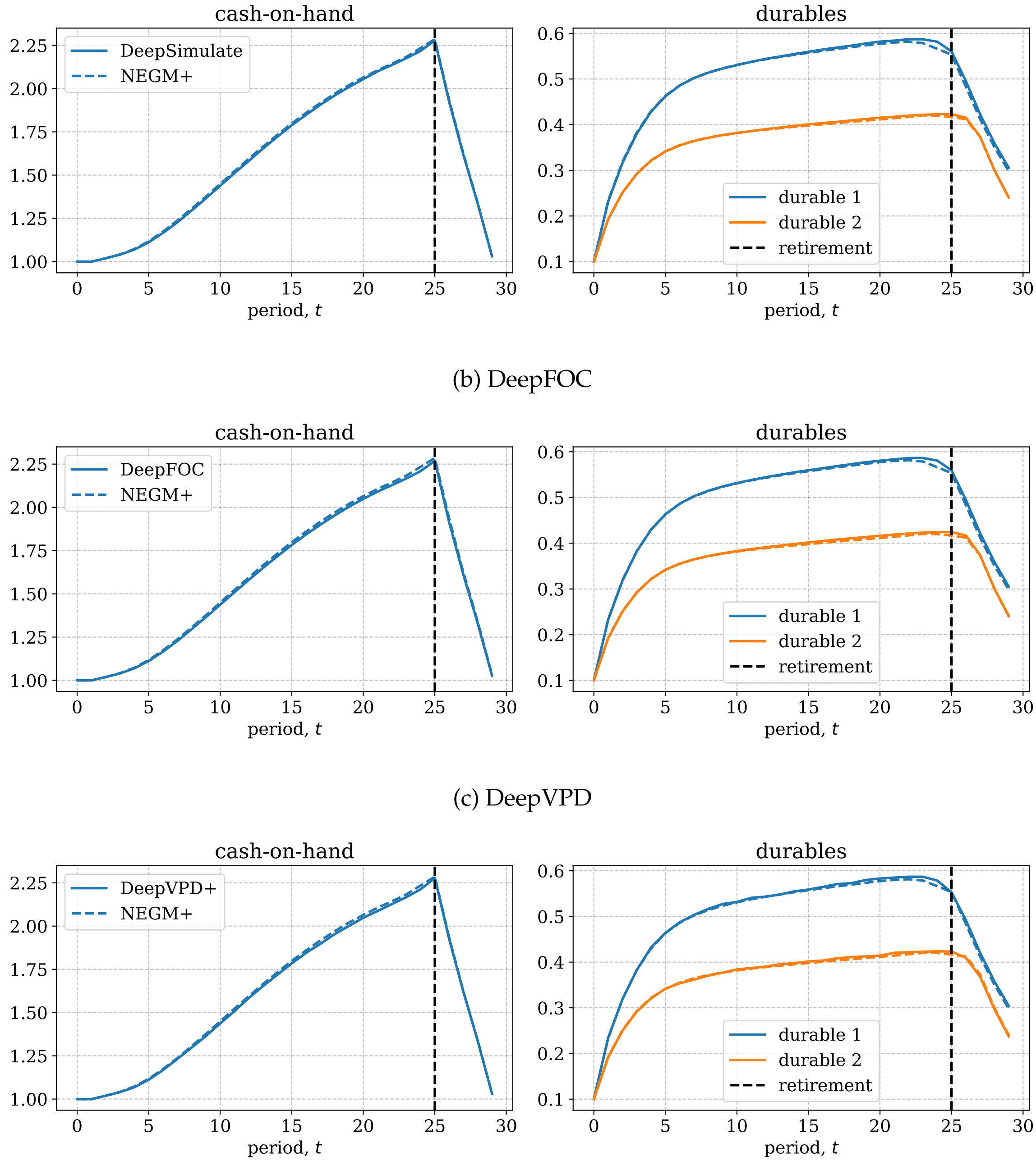

(a) DeepExplore

(b) DeepFOC

(c) DeepVPD

Figure D.3: Durables Model (1 durable): Life-cycle profiles

Notes: See Figure 7



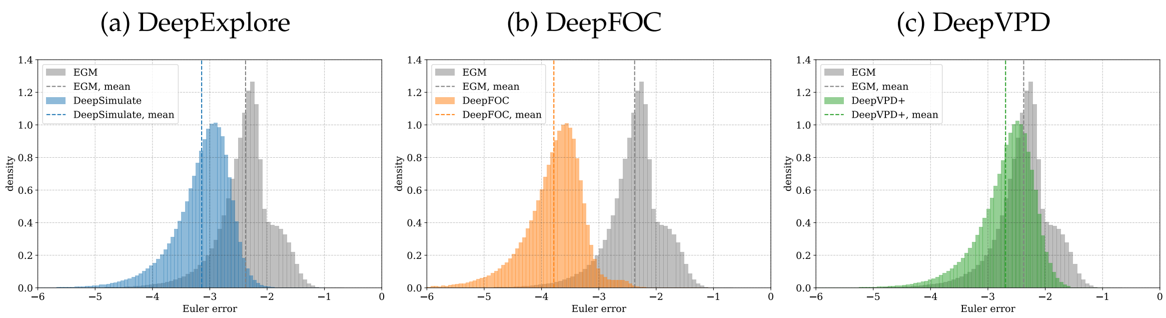
(a) DeepExplore

(b) DeepFOC

(c) DeepVPD

Figure D.4: Durables Model (2 durables): Life-cycle profiles

Notes: See Figure 7

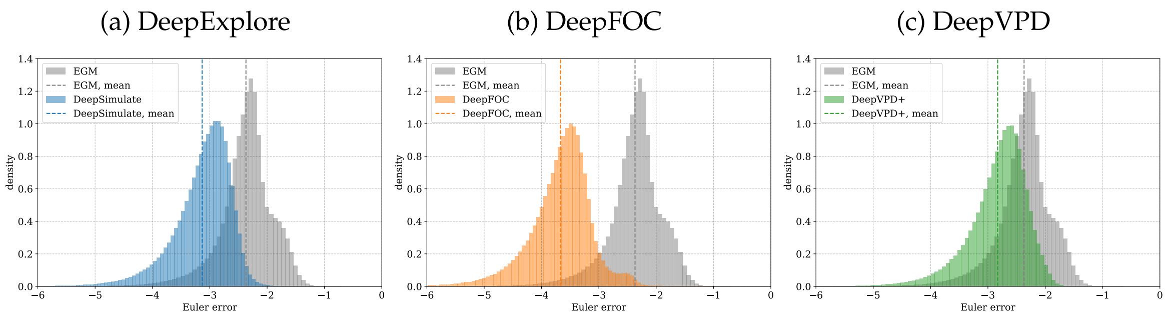

(a) DeepExplore

(b) DeepFOC

(c) DeepVPD

Figure D.5: Durables Model (1 durable): Euler-errors

Notes: See Figure 8


(a) DeepExplore

(b) DeepFOC

(c) DeepVPD

Figure D.6: Durables Model (2 durables): Euler-errors

Notes: See Figure 8

1. If not last-period: Compute post-decision value function on grids of post-decision states: $\bar{v}(p_t, \bar{m}_t, d_t)$

2. Solve keeper problem:

$$\begin{aligned} \max_{a_t^0} u(c_t, n_t) + \beta \bar{v}_t(p_t, n_t, \bar{m}_t) \\ c_t = m_t(1 - a_t^0) \\ \bar{m}_t = a_t^0 m_t \end{aligned}$$

3. Solve adjuster problem:

$$\begin{aligned} \max_{a_t^1, a_t^2} u(c_t, d_t) + \beta \bar{v}_t(p_t, n_t, \bar{m}_t) c_t = m_t(1 - a_t^0) \bar{m}_t = a_t^0 m_t \\ x_t = m_t + (1 - \kappa) n_t \\ \bar{m}_t = a_t^1 x_t \\ c_t = (1 - a_t^1) a_t^2 x_t \\ d_t = (1 - a_t^1)(1 - a_t^2) x_t \end{aligned}$$