Solving and Estimating Heterogeneous Agent Models Using Machine and Reinforcement Learning *

Yuanzhe Liu†

November 11, 2024

Abstract

In this paper, I introduce an innovative approach leveraging Deep Reinforcement Learning techniques to solve and estimate heterogeneous models. Contrasted with conventional solution methods, the deep learning approach offers a global solution while retaining the entirety of nonlinearity. Moreover, it exhibits remarkable scalability, making it suitable for handling models with hundreds or even thousands of state variables. I also explore the integration of this novel solution method with amortized likelihood-free Bayesian inference, opening up new possibilities for advanced probabilistic modeling and estimation.

Keywords: ML, NN, RL, Heterogeneous agent model

JEL Codes: C63, E21, C11

*I am grateful for continuous support and helpful comments from my advisor Peter Rupert, as well as from Kieran Walsh,and Yueyuan Ma. I am also grateful to all the participants of the UCSB Macro group; special thanks to Xander Abajian and Yang Gao. All errors are my own.

†yliu793@ucsb.edu

1 Introduction

The heterogeneous agent model has been a workhorse in macroeconomics,widely used to study the distribution of wealth, income, consumption, inequality, and the transmission of aggregate shocks. Traditionally, approximate aggregation Krusell and Smith (1998) and perturbation methods Reiter (2009); Auclert et al. (2021) have been the primary solution techniques for these models. However, these approaches have several limitations. First, solving such models requires considerable manual effort, including deriving Euler equations, discretizing the state space, and performing linearization. Second, scaling these methods to models with hundreds or even thousands of state variables is particularly challenging, especially when dealing with multiple sources of heterogeneity and aggregate shocks. Third, traditional methods typically yield only local solutions rather than guaranteed global solutions. This can be a significant drawback when nonlinearities play a crucial role or when the study's objective is to understand how nonlinearities influence the model's behavior. In such cases, traditional methods may not be the most suitable choice.

In this paper, I demonstrate how reinforcement learning (RL) can be adapted to solve heterogeneous agent models. I show that naively applying existing RL algorithms does not work effectively and propose a method to tailor commonly used RL algorithms for solving economic models. I show that this approach can reliably find the global solution without imposing additional assumptions and can easily scale to models with hundreds or thousands of state variables. Furthermore, I show how the learned policy function allows for the straightforward computation of the model's impulse response function (IRF) and enables counterfactual policy analysis, which is not feasible with traditional methods.

While RL is a powerful tool, it offers specific advantages when applied to solving economic models. For instance, unlike traditional RL applications such as game playing, economic models typically do not face issues of sparse rewards or limited exploration of the action space, making RL less likely to get stuck in local minima. However, RL is not a panacea and comes with its own limitations. A particularly notable challenge is that most meaningful

economic models are stochastic, which poses significant difficulties for RL algorithms. In Section 2, I will examine the root causes of this issue and propose strategies to address it. It is one of the core contributions of this paper.

There are several other challenges RL might face while solving economic models. Convergence issues, RL is not always guaranteed to find the optimal policy. It can sometimes get stuck in local minima for different reasons, failing to reach the global solution. Design and Initialization Sensitivity, the success of RL heavily depends on the careful design of the model, the algorithm, and the initialization. Poor design choices can hinder the agent's ability to learn effectively and find the global solution. Despite these challenges, with well-designed models and algorithms, reinforcement learning remains a powerful tool for solving complex economic models that are otherwise intractable using traditional methods.

Reinforcement learning (RL) is an unsupervised machine learning technique that has been successfully applied to various domains, including game playing, robotics, and autonomous driving. In many cases, such as Go and multiplayer online battle arena (MOBA) games, reinforcement learning has outperformed the best human players. The agent learns by interacting with an environment and receiving rewards or penalties based on its actions. The goal is to develop a policy that maximizes the agent's discounted cumulative reward over time. RL agents learn through trial and error, exploring the environment and adjusting their actions based on the outcomes they experience. Most RL algorithms rely on estimating the value of different state-action pairs from past experiences to guide future decisions.

Most problems in reinforcement learning (RL) can be formulated as a Markov decision process (MDP). An MDP provides a mathematical framework that describes the interaction between an agent and its environment. It consists of a set of states, a set of actions, a transition function that defines how the environment evolves in response to the agent's actions, and a reward function that assigns rewards to state-action pairs. This is very similar to how we build most economic models. In an economic model, the agent is household or firm. They make decisions based on the state of the economy and receive rewards or punishments

based on their actions. The goal of the agent is to maximize their expected discounted utility over time. Most economic models can be formulated as a MDP and therefore can be solved using reinforcement learning.

The key distinction between reinforcement learning and dynamic programming lies in how the solution is obtained. In dynamic programming, the model is solved using techniques such as backward induction or value function iteration. This approach starts from the final (terminal) period and works backward to determine the value function at each preceding period, by maximizing the expected value of the next period's value function. In reinforcement learning, the model is solved through trial and error. The agent begins with a random policy and gradually improves it by interacting with the environment and adjusting the policy based on the rewards it receives. The critical difference is that in dynamic programming, the transition function and reward function are assumed to be known in advance. In contrast, reinforcement learning requires the agent to learn these functions from simulated or observed data through interaction with the environment.

Traditional methods, such as dynamic programming, have several limitations when applied to models with large state spaces, like heterogeneous agent models. In these cases, the state space becomes too large to efficiently solve the model using dynamic programming. This challenge, known as the curse of dimensionality, makes it computationally infeasible to explore every possible state-action combination.

In contrast, reinforcement learning (RL) offers a promising alternative for handling large state spaces. RL algorithms can efficiently explore vast environments and, crucially, have the potential to find a global solution, while traditional methods typically only identify local solutions. The perturbation and projection method is one of the most commonly used techniques for solving heterogeneous agent models, but it also has limitations—it cannot guarantee a global solution and requires assumptions to simplify the model and make it tractable. In comparison, reinforcement learning can find global solutions without relying on these assumptions.

After solving the model, we may wish to estimate it using data. However, since the policy and value function are approximated using neural networks, directly computing the likelihood function is not feasible for Bayesian or maximum likelihood approaches. To address this challenge, one of my contributions in this paper is integrating amortized likelihood-free Bayesian inference with RL as a method for model estimation.

Literature Review

The most widely used method for solving heterogeneous agent models with aggregate shocks was introduced by Krusell and Smith (1998), followed by many extensions. A common feature of these approaches is their attempt to represent the cross-sectional distribution of agents using a small number of statistics, thereby reducing the dimensionality of the state space.

Reiter (2009) proposed a projection and perturbation method to solve heterogeneous agent models. However, this method requires linearization around the stationary steady state. When the model includes multiple aggregate shocks or a multidimensional cross-sectional distribution, the method becomes slower, and if the model is highly nonlinear in aggregate shocks, it becomes less accurate. Building on Reiter (2009), Auclert et al. (2021) introduced a new approach by perturbing the model to first order in aggregates. Rather than representing the equilibrium as a system of linear equations in state space, they express it as a system of linear equations in sequence space. However, their method still relies on linearization, meaning it cannot guarantee a global solution.

Han et al. (2021) proposed a novel approach to solving heterogeneous agent models using deep learning. In their method, a neural network approximates both the value function and the policy function. They employ fictitious play to iteratively update these functions. Their results demonstrate that the method can reliably find the global solution and scale efficiently to models with hundreds or thousands of state variables. Similarly, Azinovic et al. (2022) developed a deep learning-based method to solve heterogeneous agent models.

Their approach uses a neural network to approximate the value function and applies Euler equations as constraints during the value function update process.

what exactly are we expecting from a good solution method? A good solution method should meet the following criteria as discussed in Han et al. (2021):

- **Efficiency**: This is necessary in order to use the method for calibration, estimation, and further quantitative analysis.
- **Reliability**: In particular, it should be applicable beyond the local perturbation regime if nonlinearity matters, e.g. ZLB.
- **Generality**: The method should be applicable to a wide variety of different HA models. Multiple shocks and multidimensional cross-sectional distribution should be handled without much modification.

In this paper, I will show that reinforcement learning can meet all these criteria and provide a powerful alternative to traditional solution methods for heterogeneous agent models.

The rest of the paper is organized as follows. Section 2 introduces the reinforcement learning (RL) method applied to solve the model. In Section 3, I present the results obtained using the RL method and display the impulse response function (IRF). Section 4 explains how to use simulation-based inference to perform Bayesian estimation of the model. Finally, Section 5 concludes the paper and discusses directions for future research.

2 Reinforcement Learning

Reinforcement Learning agents learns by trail and error. They interact with the environment and receive rewards or penalties based on their actions. The goal of the agent is to learn a policy that maximizes its cumulative reward over time. Most RL algorithms does not require external data, they learn from the data they generate by interacting with the environment/simulator.

For example, consider the game of Snake, where the goal is to eat as many apples as possible without hitting the walls or the snake's own body. In this scenario, the agent is the snake, the environment is the game board, and the rewards are the number of apples the snake consumes. If the snake hits the wall or itself, it receives a penalty. The snake learns how to navigate the board and collect apples by exploring its environment and adjusting its actions based on the rewards it receives. Its policy consists of the rules it follows to decide how to move. By trying different moves and observing the outcomes, the snake gradually improves its strategy.

However, applying RL to economic models requires careful consideration of the model's structure and the agent's decision-making process. Otherwise, the agent may struggle to learn an optimal policy and stuck in a local minimum. One possible outcome of the game of Snake example is that the agent learns to just turning around in a circle, which is not the optimal policy because it does not eat any apples. It learns to do so because it is the easiest way to avoid hitting the wall or itself. Therefore, it is important to carefully design the model, the algorithm, and the initialization to ensure that reinforcement learning can find the global solution. This is less of a concern in most economic models because the agent will face a server utility loss if it consume to little.

Reinforcement Learning (RL) consists of four essential elements, as shown in Figure 1: Agents, Environment, Actions, and Rewards. In the context of the Krusell and Smith (1998) model, the agents are the households, the environment corresponds to the transition function that governs how the state variables evolve over time, the actions are the households' consumption and saving decisions, and the rewards is the utility derived from consumption. The agent's objective is to learn a policy that maximizes its expected cumulative reward over time.

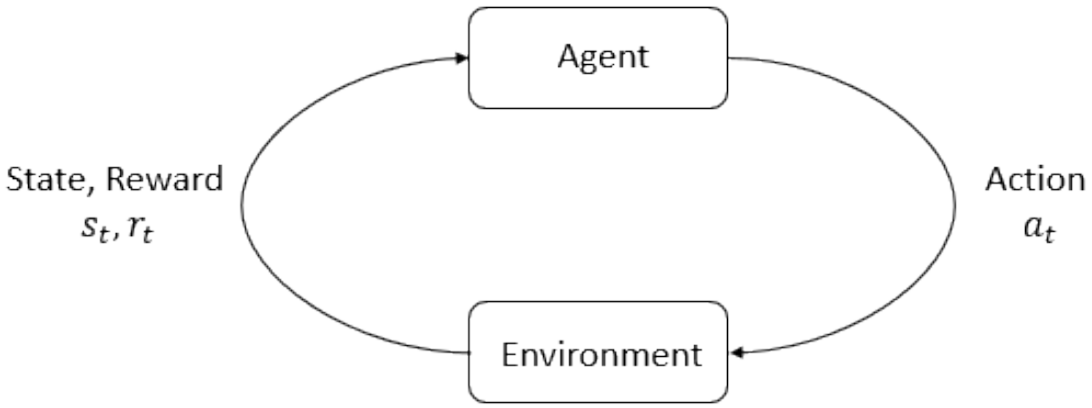

Figure 1: Reinforcement Learning Framework

Reinforcement Learning (RL) has proven to be a powerful approach for decision-making in Markov environments, achieving remarkable results in areas such as game playing, robotic control, and autonomous driving. This paper focuses on model-free policy gradient methods, which are widely applied in continuous action spaces to solve various problems. A key challenge for this class of methods lies in the high variance of the gradient estimator. When the environment is stochastic, the variance of the gradient estimator becomes even higher. This makes policy learning particularly challenging, as the agent struggles to discern whether a poor outcome is due to taking a suboptimal action or merely bad luck.

To address this issue, one approach involves using a stochastic policy function to explore the value of different actions. The data used in most online reinforcement learning algorithms is simulated using the policy function to interact with the environment. Therefore, if the policy function is stochastic then for a given state, there could be many different actions taken during simulation. This exploration strategy can help the agent learn to distinguish between suboptimal actions and bad stochastic shocks.

However, in economic models, it is often difficult to justify the assumption that the agent is making random decisions. Alternatively, a deterministic policy function can be employed to select actions, which aligns more closely with economic theory. Nevertheless, learning such a policy is challenging due to the high variance of the gradient estimator.

In this paper, I propose a novel approach to address this challenge. I introduce a method that combines the benefits of both stochastic and deterministic policies, leveraging the strengths of each to improve learning efficiency.

In this paper, we focus on discrete time Markov decision process (MDP), defined as follows: $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \rho_0, \mu, \beta)$ where $\mathcal{S} \subseteq \mathbb{R}^n$ is the state space, $\mathcal{A} \subseteq \mathbb{R}^m$ is the action space, $\mathcal{P}$ is the transition probability function, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to [0, 1]$, ρ_0 is the initial state distribution, μ is the reward function, and β is the discount factor. The goal of the agent is to learn a policy π that maximizes the expected cumulative reward over time. The policy is a mapping from states to actions, $\pi : \mathcal{S} \to \mathcal{A}$.

Through out this paper, there will be two types of *value function*. The first type is the value function $V^\pi(s)$, which represents the expected cumulative reward starting from state s and following policy π. It is the same as what we have in the economics literature. $V^\pi(s_t) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \beta^t \mu_t(a_t) \mid s_t\right]$. The second type is the state-action value function $Q^\pi(s_t, a_t)$, which represents the expected cumulative reward starting from state s_t, taking action a_t, and then following policy π. $Q^\pi(s_t, a_t) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \beta^t \mu_t(a_t) \mid s_t, a_t\right]$. The purpose of having two types of value function is that the state-action value function is easier to learn than the value function and it can be used to update the policy function. When update the policy function, we use the gradient calculated from the state-action value function.

2.1 Policy Gradient Methods

Policy gradient methods are a class of reinforcement learning techniques that update the parameters of a function approximator (in this paper, a neural network) to increase the value function. Let $J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s)$ be the performance measure, where v_{π_θ} represents the true value function for the policy function π_θ. Here, θ denotes the parameter of the policy function. For instance, when approximating the policy function using a neural network (NN), θ corresponds to the weights and biases of the network. More details about NN can be found in Goodfellow et al. (2016); Zhang et al. (2023). The task is to update θ to maximize the

value function. The policy gradient theorem states that the gradient of the expected return with respect to the policy parameters is proportional to the sum of rewards, weighted by the gradient of the log probability of the chosen action. Mathematically, the policy gradient theorem can be expressed as:

$\nabla_\theta J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a Q^\pi(s,a) \nabla \pi(a \mid s, \boldsymbol{\theta})$

Here, $Q^\pi(s,a)$ represents the action-value function, which denotes the expected cumulative future rewards an agent can obtain by starting from state s, taking action a, and then following a specific policy π_θ in this context. More details and the proof of the policy gradient theorem can be found in Sutton and Barto (2018). The distribution $\mu()$ here is the on-policy distribution under π. It can be seen as the ergodic set of the economy. All parts of $\nabla_\theta J(\boldsymbol{\theta})$ can be estimated from simulated data.

There are various policy gradient methods, such as REINFORCE, PPO, DDPG, and TRPO. In this paper, I use the Deep Deterministic Policy Gradient (DDPG) algorithm (Algorithm 1) to solve the heterogeneous agent model. DDPG is an off-policy actor-critic algorithm that uses a deterministic policy to select actions. It is well-suited for continuous action spaces and has been shown to be effective across many problems. Off-policy means that the policy used to generate the data differs from the policy used to update the value function. This feature is useful in practice, as it allows us to reuse data collected from previous policies, reduced correlations between samples, and improve sample efficiency. If we use a single trajectory of simulation to update the policy, the data will be highly correlated since we use the same policy function, which will make the gradient estimator have high variance. The intuition is that correlated sample will not provide much information compare to uncorrelated sample. If one decision is bad, the next decision is likely to be bad as well. Therefore, the policy updating process will be more effective if we use less correlated samples.

The actor-critic structure of DDPG means that the algorithm learns both a policy (actor) and a action-value function (critic) simultaneously. The actor updates the policy using the gradient of the value function, which provides a more stable estimate of the policy gradient.

An example of an on-policy method is the REINFORCE algorithm. In REINFORCE, the policy is updated using the gradient of the expected return with respect to the policy parameters. The expected return is estimated by running a simulation over many periods and computing the sum of rewards. The policy is then updated using the gradient of the log probability of the action, multiplied by the return. In contrast, for off-policy methods, we simulate the environment to collect data and store them in a reply buffer, and then sample from this reply buffer to estimate the gradient.

Algorithm 1: Deep Deterministic Policy Gradient

Initial policy and action-value function parameters θ and ϕ, and a reply buffer $\mathcal{D}$;

Pre train the policy function using supervised learning;

while *Not Converge* **do**

Observe state s and select action $a = \text{clip}\left(\pi_\theta(s) + \epsilon, a_{\text{Low}}, a_{\text{High}}\right)$, where $\epsilon \sim \mathcal{N}$;

Execute a ;

Observe next state s', utility r ;

Store (s, a, r, s') in reply buffer $\mathcal{D}$;

Update s to s' ;

if *it is time to update* **then**

Randomly sample a batch $B = \{(s, a, r, s')\}$ from $\mathcal{D}$;

$y\left(s'\right) = r + \beta Q_{\phi_{\text{targ}}}\left(s', \pi_{\theta_{\text{targ}}}\left(s'\right)\right)$;

$\phi = \phi - \nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s'\}\in B} \left(Q_\phi(s,a) - y\left(s'\right)\right)^2$;

$\theta = \theta + \nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi\left(s, \pi_\theta(s)\right)$;

$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$

$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$

end

end

The key idea of Deep Deterministic Policy Gradient (DDPG) is to use a deterministic policy to select actions and learn a value function that estimates the expected return of the policy. The policy is updated using the gradient of the value function, which provides a more stable estimate of the policy gradient. The value function is updated according to the Bellman equation, similar to how it is done in Q-learning.

Since DDPG trains a deterministic policy in an off-policy fashion, it may initially fail to explore a wide enough variety of actions, potentially getting stuck in a local solution. To address this, exploration noise is added to the action selection process Silver et al. (2014) and Lillicrap et al. (2015). This noise encourages the policy to explore more diverse actions. As the policy becomes more confident over time, the noise is annealed (gradually reduced) to focus more on exploitation. I also pre-train the policy function using a supervised learning algorithm to provide a better starting point for the RL algorithm. This stabilizes the training process by reducing the chances of poor initial performance. The initial policy function is trained under the assumption that each agent consumes only their labor income and interest income, meaning their capital stock remains unchanged over time. Although this is not the optimal policy, it serves as a reasonable baseline from which the RL algorithm can make further improvements.

2.2 Model

Here is a brief review of Krusell and Smith (1998), rephrased using the language of reinforcement learning. In this model, we assume there is a continuum (infinite number) of households in the economy. Each household faces a decision-making problem that can be described by a Bellman equation, which represents the household's dynamic optimization problem.

$$v(k, l; \lambda, z) = \max_{c,k'} \{u(c) + \beta E \left[v \left(k', l'; \lambda', z'\right) \mid (\lambda, z)\right]\} \tag{1}$$

where the maximization is subject to

$$c + k' = \tilde{r}(K, N, z)k + w(K, N, z)l + (1 - \delta)k \tag{2}$$

$$\tilde{r} = \tilde{r}(K, N, z) = z\alpha \left(\frac{K}{N}\right)^{\alpha-1} \tag{3}$$

$$w = w(K, N, z) = z(1 - \alpha) \left(\frac{K}{N}\right)^{\alpha} \tag{4}$$

$$\lambda' = H(\lambda, z) \tag{5}$$

where (K, N) denotes aggregate capital and labor defined as:

$$K = \int k\lambda(k, l)dkdl \tag{6}$$

$$N = \int l\lambda(k, l)dkdl. \tag{7}$$

Here, $\lambda(k, l)$ represents the joint distribution of capital (k) and labor (l) across households, while $H(\cdot)$ in Equation 5 denotes the transition kernel. This distribution is a random function influenced by the aggregate shock (common noise) z. Typically, we assume that z follows an AR(1) process: $z' = (1 - \rho_z)\mu_z + \rho_z z + \epsilon_z$ where ϵ_z is drawn from a Gaussian distribution. Similarly, labor productivity s is also assumed to follow an AR(1) process.

Households choose their consumption c for the current period and savings k' for the next period, subject to the budget constraint in Equation 2. Here, $\tilde{r}$ denotes the interest rate, and w represents the wage. On the right-hand side of Equation 2, the budget constraint accounts for capital income, labor income, and the value of undepreciated capital from the previous period. The parameter δ represents the depreciation rate of capital.

The equilibrium is defined as follows:

1. Given the price functions, 3 and 4, and $H(\cdot)$, the value function solves the Bellman equation 1 and the optimal decision rule is $f(k, l, \lambda, z)$. ($f(k, l, \lambda, z)$ returns c and k')

2. The decision rule $f(k, l, \lambda, z)$ and the processes for l and z imply that today's distribution $\lambda(k, l)$ is mapped into tomorrow's $\lambda'(k, l)$ by H (I think is part is similar to the forward backward phrase in the mean field game literature.)

In the original paper by Krusell and Smith (1998), instead of tracking the full distribution λ, which has infinite dimensions, they approximate it by tracking only the first moment of λ. This simplification works remarkably well within their model. However, relying solely on the first moment may not be effective in other models where the entire distribution matters significantly. Important implications of the model may also be lost if we ignore higher-order moments or other features of the distribution.

The intuition behind why this method works is that, when households make consumption and saving decisions, they need to forecast the next period's interest rate, $\tilde{r}'$. To make an accurate forecast, it is sufficient to know only the first moment of the distribution λ since the marginal propensity to save is almost the same across the distributution. While some researchers have attempted to incorporate higher moments into the method, the improvements in accuracy have been marginal.

This model provides a useful benchmark to explore whether RL can be effectively applied to this class of models. For more complex models, tracking only the first moment may no longer be sufficient, especially when the dynamics of aggregate capital K are highly nonlinear. Later, I will discuses the potential of using RL to solve more complex models, for example, when there are multiple asset types such as risk free bond which by definition the aggregate is zero.

2.3 Implementation details

In the Krusell-Smith (KS) economy, each agent observes the state of the economy in each period, including the interest rate (r), wage (w), total factor productivity (TFP), and the cross-sectional distribution of asset holdings and labor endowments. Based on this information, they make consumption and saving decisions and then transition to the next period.

We simulate 100 KS agents. For $i \in \{1, 2, 3...100\}$

$s_{t,i} = \{k_{t,1}, k_{t,2}..., k_{t,100}; l_{t,i}, z_t\}$

$a_{t,i} = \pi_\theta(s_{t,i})$

$u_{t,i} = log(a_{t,i})$

$k_{t+1,i} = (1 + r_t) * k_{t,i} + w_t * l_{t,i} - a_{t,i}$

Once every agent has made their decision, we draw a new TFP shock, z_{t+1}, and transition to the next period.

$s_{t+1,i} = \{k_{t+1,1}, k_{t+1,2}..., k_{t+1,100}; l_{t+1,i}, z_{t+1}\}$

Store $\{s_{t,i}, a_{t,i}, u_{t,i}, s_{t+1,i}\}$ in the memory reply buffer and update policy and action-value function according to Algorithm 1.

The equilibrium conditions discussed earlier are satisfied implicitly through the simulation and data collection process. The transition function of the aggregate state is generated by the simulation which is governed by the policy function. The wage and interest rate are calculated using the first-order condition from the firm's profit maximization problem. The value function is updated via minimizing the Bellman error as discussed in Algorithm 1, and the policy function is adjusted to maximize the value function. This algorithm is repeated until the policy converges. Although we do not explicitly impose the equilibrium conditions in the algorithm, they are met through the simulation process. Table 1 shows the parameter values used in the model, while Table 2 shows the parameters of the learning algorithm. Some Parameters are then estimated using Bayesian estimation. The details of the estimation process are described in Section 4.

Table 1: Model Parameter Values

Parameter	**Description**	**Value**
β	Discount Factor	0.98
μ_z	Mean of TFP	1
ρ_z	Persistence of TFP process	0.95

Table 1 – Continued

Parameter	Description	Value
ϵ_z	st of TFP process	0.02
μ_s	Mean of labor	0.117
ρ_s	Persistence of labor process	0.95
ϵ_s	st of labor shock	0.05
α	Weight of capital in production function	0.33
δ	Depreciation rate of Capital	0.025
σ	utility	1

Table 2: DDPG Parameter Values

Parameter	Description	Value
lr_{actor}	Learning rate of policy function	$5e^{-4}$
lr_{critic}	Learning rate of action-value function	$1e^{-3}$
B	Batch size	64
S_θ	Structure of NN_{policy}	$[256, 128, 32]$
S_ϕ	Structure of NN_{value}	$[256, 128, 32]$
τ	soft update	$1e^{-3}$
ϵ	Exploration noise	0.001

We simulate 100 agents to represent the distribution of agents in the economy. However, they share the same policy and value function. The reason is that there is no ex-ante heterogeneity in the model, and the only difference between agents is their asset holding and labor endowment. Therefore, it is sufficient to learn one policy and value function for all agents. Even there are Ex-ante heterogeneity, we can still use the same policy and value function for all agents if we add the agent's type as an input to the policy function. This can actually stabilizes training process because there are less parameters to be learned given the fact that RL algorithms are typically data inefficient.

To make the training process more stable and ensure that the RL algorithm converges to the global solution, three key features are incorporated into the algorithm. First, we pre-

train the policy function using a supervised learning algorithm to provide a better starting point for the RL algorithm. This stabilizes the training process by reducing the chances of poor initial performance. The initial policy function is trained under the assumption that each agent consumes only their labor income and interest income, meaning their capital stock remains unchanged over time. Although this is not the optimal policy, it serves as a reasonable baseline from which the RL algorithm can make further improvements. The first one hundred iterations is supervised pre-training. Once the pre-training is done, we switch to the RL training process while adding exploration noise to the action selection process. The noise is annealed over time to reduce the amount of exploration as the policy becomes closer to the optimal policy.

The second modification is instead of directly inputting the entire joint distribution of capital and labor into the policy function, we use a summary network to condense the distribution into a few key statistics. This summary network is shared between the policy function and the value function, promoting more effective learning and stability during training. The intuition behind this modification is that the information from the cross-sectional distribution should be consistent for both the policy and value functions. Additionally, this approach reduces the dimensionality of the state space, making the training process more efficient. It also make the algorithm permutation invariant, which is important in the context of the heterogeneous agent model since the order of agents should not affect the policy function. We could have input the entire distribution into the policy function, however, it would have made the training process much slower and less stable because the policy function would have to learn the fact that the order does not matter.

I use Φ_{θ_s} to denote the summary network. The input of the summary network is the individual asset holding and labor endowment, and the output is the summary statistics. The *mean* of the summary statistics,S_t, are then input into the policy function and the value function.

$$\Phi_{\theta_s}(k_{t,i}, l_{t,i}) = s_{t,i} \tag{8}$$

$$S_t = \frac{1}{N} \sum_{i=1}^{N} \Phi_{\theta_s}(k_{t,i}, l_{t,i}) \tag{9}$$

Han et al. (2021) also employed a similar approach in their work. They used a neural network to represent a generalized moment in contrast to Krusell and Smith (1998) which only keep track of the first moment. The output of the summary network is not necessarily one dimension but can be multiple dimensions. The summary network is meant to capture the key information of the cross-sectional distribution. In practice, I found that setting the output dimension to be 4 works well for the Krusell-Smith model.

Another important consideration when using RL to solve economic models is that most economic models are inherently stochastic. Although the policy gradient theorem in Sutton and Barto (2018) applies to both deterministic and stochastic environments, the majority of RL studies primarily evaluate algorithm performance in deterministic settings. For instance, the widely used Gymnasium environment Towers et al. (2024) is deterministic. Similarly, in the snake game example discussed earlier, the environment is deterministic: the agent's actions, such as turning left or right, deterministically move the snake in the corresponding direction, with no uncertainty in the process.

In contrast, economic models like the Krusell-Smith model involve stochastic elements, such as total factor productivity (TFP) shocks and labor endowment processes. In these cases, the agent must learn the *expected value* of the policy function under stochastic conditions. This differs from deterministic environments, where the expectation operator in the Bellman equation can be safely omitted, particularly when using a deterministic policy function. Addressing this stochasticity poses a unique challenge in applying RL to economic models.

In practice, making RL algorithms work effectively in stochastic environments is challeng-

ing. The variance of the gradient estimate can be very high due to stochastic environment, leading to unstable training and poor convergence. In other words, agents may struggle to distinguish between bad luck due to randomness in the environment and a suboptimal policy.

To address this issue, stochastic policy functions are often used to explore the value of different actions. However, in economic models, it is often difficult to justify the assumption that agents make random decisions. High variance in the gradient estimates can lead to algorithm instability. When the variance of the policy gradient estimator is high, it is likely that, at some point during training, the policy function will be updated in a direction that worsens performance. Even a single misstep in the update process can cause the policy function to diverge rapidly, resulting in an unstable training process. Since the training data is generated by the policy function, a diverging policy function can produce data that is far outside the ergodic set of the economy, exacerbating divergence and destabilizing the training further.

One of the key contributions of this paper is the introduction of a novel simulation strategy to generate data for the RL algorithm. In our benchmark Krusell-Smith model, the state variable consists of two components: (1) the stochastic component, including TFP and labor endowment of each agent, and (2) the deterministic component, which is the asset-holding distribution. The stochastic component often leads to high gradient variance, while the deterministic component is governed by the policy function. The simulation strategy I use is to generate multiple trajectories of the stochastic part of the state variable and then use the deterministic policy function to generate the asset holding distribution for the next period. This way, we collect data that is more informative to learn the expected value of the policy function. The intuition is that the agent could see on average how good the future state is. This way, the policy function will be more stable and the training process will be more efficient.

Another set of crucial parameters to tune for stability and efficiency are the buffer size and the update frequency of the policy and value functions. The buffer size refers to the

number of data points stored in the replay buffer. When the buffer is full, older entries are replaced using a first-in, first-out (FIFO) rule. The buffer size significantly influences the quality of gradient estimates. If the buffer size is too small, the agent will primarily rely on very recent data, which may result in a poor representation of the ergodic set. This can lead to catastrophic forgetting, where the agent forgets how to respond to certain state variables if it has not encountered them for a long time. Conversely, if the buffer size is too large, the agent may rely on outdated data, which can reduce the efficiency of the training process.

The update frequency of the policy and value functions is equally important. If the update frequency is too high, the policy function becomes volatile, destabilizing the training process. If the update frequency is too low, training progresses too slowly. In this paper, I use a buffer size of 10^7, striking a balance between training efficiency and policy stability. I update the policy and value functions once per iteration. Additional parameter choices are summarized in Table 2.

In the Appendix A I use a very simple RBC model to show that the source of explosive training process is the high policy gradient variance caused by the stochastic environment. The only in the RBC model is the TFP shock. I show that if we turn off the TFP shock then the vanilla RL algorithm learns the optimal policy without any problem. However, if we turn on the TFP shock then the vanilla RL learned policy function will be very volatile.

3 RL Result

The learning rate for the policy function $5e^{-4}$, the learning rate for the action-value function $1e^{-3}$, the batch size $|B|= 10^4$, the target update rate $\tau = 0.01$, and the exploration noise $\epsilon = 0.01$. I use a neural network with three hidden layers of 128 units each and Tanh activation functions to approximate the policy function and the action-value function. THe batch size is unusually large

The following figures show the results of the RL method. Figure 2 shows the distribution

of wealth in the economy after half a million iterations.

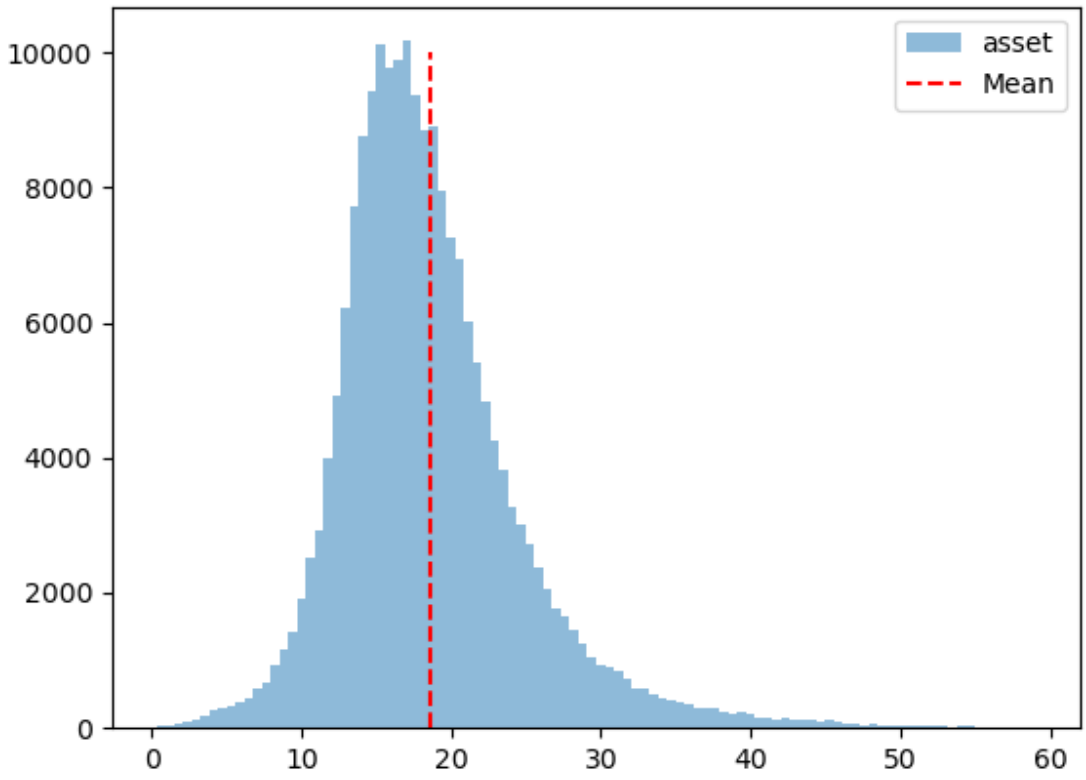

Figure 2: Wealth distribution after 500,000 iterations.

The following figures illustrate the consumption and saving policy functions obtained using the RL algorithm. Figure 3 shows the consumption function, while Figure 4 presents the saving function. Both figures closely resemble the policy functions obtained through the traditional method. As noted in the original paper, the policy functions are nearly linear because the marginal propensity to consume (MPC) remains almost constant across the wealth distribution.

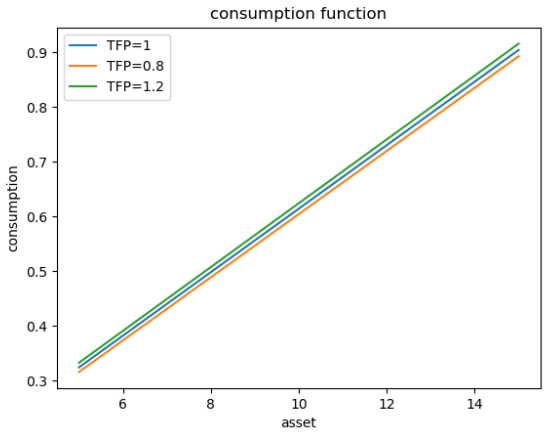

Figure 3: consumption function.

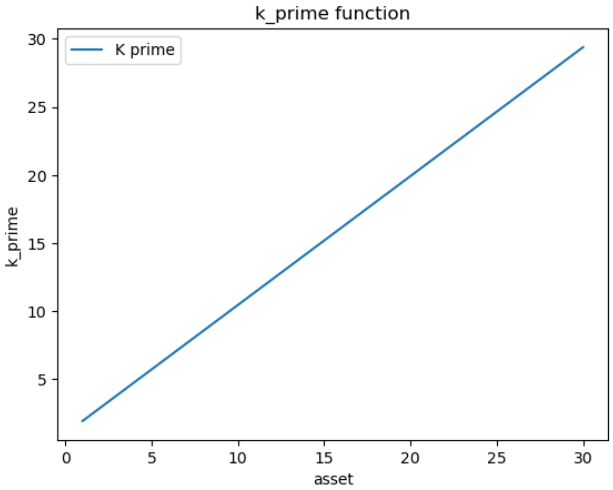

Figure 4: k'

Figure 5 shows the impulse response function of a transfer policy. The transfer policy is a policy that transfers a fixed amount of wealth from the rich to the poor. The social planner take one unit of wealth from agents whose wealth is at the top one third and give it to agents whose wealth is at the lower one third. The figure 5 shows the effect of this transfer policy on middle class who are in the middle one third of the wealth distribution. The original method proposed by Krusell and Smith (1998) states that only the mean of the distribution matters and the MPC is almost the same across the wealth distribution, therefore, the transfer policy should not have any impact on the middle class.

It is very easy to obtain the IRF of the model using the policy function learned by the RL algorithm. We can simply simulate this transfer for many time and then compute the average of the outcome. If we were to use the traditional method, we would have to solve the model for each period and then compute the outcome. The RL method is much faster and more efficient than the traditional method when it comes to compute the IRF.

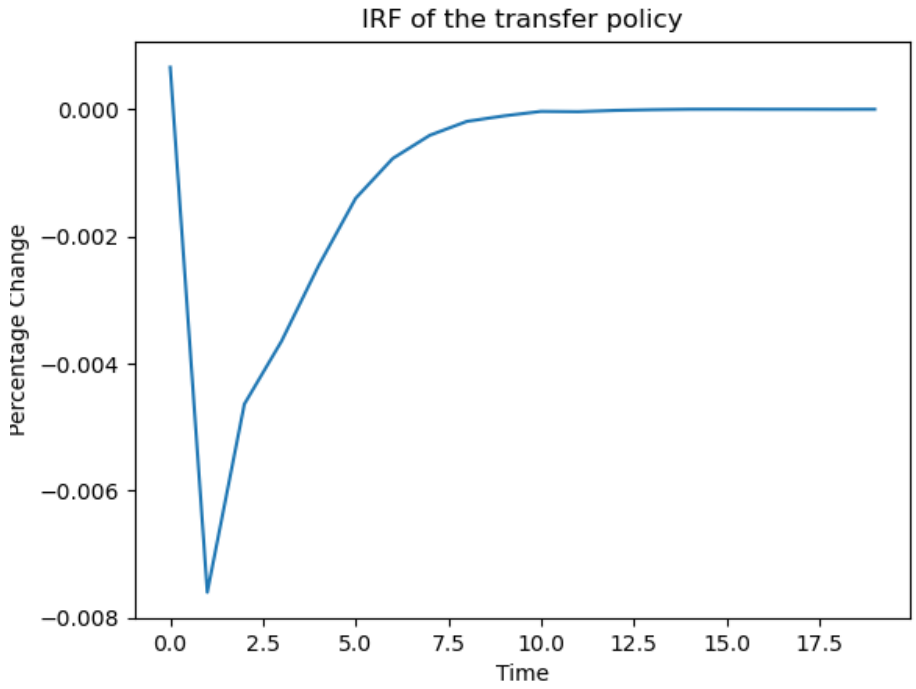

Figure 5: Impulse response function of a transfer policy.

4 Simulation Based inference

After solving the model, we may want to estimate some parameters using real data. However, because the policy function is approximated using neural networks, we must rely on likelihood-free methods for estimation, as the likelihood function is not analytically available.

Traditionally, when models are solved by linearizing around their steady state, methods like those in Herbst and Schorfheide (2016) can be employed. These methods reformulate the model in state-space form, use the Kalman filter to compute the likelihood function, and apply sampling techniques, such as Metropolis-Hastings, to estimate the posterior distribution of the parameters. However, this approach is not feasible in our case due to the neural network approximation, which renders the likelihood function unavailable.

Another limitation of the traditional method is that it primarily relies on aggregate variables—such as GDP, aggregate consumption, and aggregate investment—as demonstrated

in Smets and Wouters (2007). In heterogeneous agent models, however, the cross-sectional distribution of variables, such as capital, bonds, and consumption, carries critical information about the model parameters. The traditional approach neglects this rich cross-sectional information because it does not account for these distributions, further limiting its applicability when the likelihood function is not available.

Liu and Plagborg-Møller (2023) proposed a method for incorporating the cross-sectional distribution of variables—such as consumption, savings, and wealth—into Bayesian estimation. This method involves solving for the conditional probability of the cross-sectional distribution given the aggregate state variables. Similarly, Kase et al. (2022) developed a technique that leverages information from cross-sectional distributions, using machine learning methods to approximate the particle filter for estimation.

An additional advantage of likelihood-free methods is that, when combined with the RL-based solution approach, they enable an end-to-end framework for solving and estimating the model. During estimation, the model must be solved multiple times for different parameter values. With this approach, each solved solution can serve as the starting point for the next iteration, significantly reducing computational time and resource requirements.

4.1 SBI method

When using RL or other methods involving neural networks to approximate the policy and value functions, an analytical expression for the likelihood function given observations is not available, rendering conventional Bayesian inference methods inapplicable. Instead, an alternative class of likelihood-free estimation methods has been developed in the literature, with Approximate Bayesian Computation (ABC) being a popular choice. ABC is a simulation-based method that approximates the posterior distribution of parameters by comparing simulated data with observed data.

The basic idea of ABC is as follows: for a given set of parameters drawn from the prior, we simulate the model over many periods and then compare the simulated data with

the observed data. The distance between the simulated and observed data serves as a measure of likelihood for those parameters. If this distance exceeds a specified threshold, the parameters are rejected; otherwise, they are accepted. The accepted parameters are then used to construct an approximate posterior distribution.

Another way of seeing this is to think of this process as of obtaining the joint distribution. $p(X, \theta) = p(\theta) * p(X|\theta)$. The posterior distribution of the parameters is then given by $p(\theta|X) = \frac{p(X,\theta)}{p(X)}$. The likelihood function $p(X|\theta)$ is not available analytically so does the joint distribution. If we draw many sets of parameters from the prior and simulate the model for many periods, we can obtain the joint distribution. The posterior distribution of the parameters is then given by the joint distribution divided by the marginal distribution of the data $p(X)$.

Papamakarios and Murray (2018) proposed a new method to estimate the posterior distribution of the parameters using normalizing flow. Normalizing flow is a generative model that learns the distribution of the data by transforming a simple distribution into a complex distribution. The key idea of normalizing flow is to learn a series of invertible transformations that map a simple distribution to a complex distribution. The parameters of the transformations are learned using a neural network. The normalizing flow model can then be used to sample from the posterior distribution of the parameters. The advantage of normalizing flow is that it can capture complex dependencies in the data and can be trained efficiently using stochastic gradient descent. Normalizing flow has been shown to be effective in a wide range of problems, including density estimation, generative modeling, and variational inference.

Radev et al. (2023) developed a more advanced method that amortized the estimation process. That is for any given observations, without the need to re-train the model, we can obtain the posterior distribution of the parameters simply by evaluating the trained NN. At its core, Radev et al. (2023) includes three neural networks: a summary network for encoding data into embeddings, a posterior network for amortized posterior inference, and

a likelihood network to approximate likelihood functions. This combined approach allows for efficient estimation of marginal likelihoods and posterior predictive distributions, both of which are crucial for Bayesian model validation and comparison. Ultimately, Radev et al. (2023) represents a flexible and fully amortized approach that enhances the efficiency and practicality of Bayesian inference, making it suitable for complex probabilistic models and computationally demanding simulations.

When we estimate heterogeneous agent models, we would like to include the cross section distributions such as the distribution of wealth, consumption, and saving. The traditional method proposed by Herbst and Schorfheide (2016) is to linearize the model around its steady state and then express the likelihood function. However, this method is not feasible in this case because the likelihood function is not available. The amortized inference method allows us to include the cross section distribution into the estimation. We first pass the data through a summary network to obtain the summary statistics. The summary statistics are then input into the posterior network to obtain the posterior distribution of the parameters.

However, the original Radev et al. (2023) paper did not address identification issues, which are crucial in the context of economic models. There are two primary reasons why some parameters may be difficult to identify. First, the likelihood function may be very flat, making it challenging to distinguish between different parameter values. Second, the available data may not be sufficiently informative to identify certain parameters. Although identification is beyond the scope of this paper, it is an important issue that warrants further investigation in future research.

The following algorithm 2 shows the training process of the SBI method. The algorithm is based on the original paper by Radev et al. (2023).

Algorithm 2: SBI: Training

Initialize the prior $p(\theta)$ and function $f_\Phi()$;

while *Not Converge* **do**

Sample $\{\theta_i\}_{i=1}^{i=N}$;

Simulate data $\{X_i\}_{i=1}^{i=N}$ from the model using $\{\theta_i\}_{i=1}^{i=N}$;

Pass $\{\theta_i, X_i\}_{i=1}^{i=N}$ through $f_\Phi()$ to get $\{z_i\}_{i=1}^{i=N}$;

Calculate the loss function Kullback-Leibler divergence between the empirical distribution of z and standard normal distribution;

Update Φ to minimize the loss function;

end

In this framework, $p(\theta)$ represents the prior distribution of the parameters, and $f_\Phi()$ is the neural network that approximates the invertible function mapping a standard normal distribution to the posterior distribution of the parameters. The loss function is the Kullback-Leibler (KL) divergence between the posterior distribution of the parameters and the standard normal distribution. The neural network is trained to minimize this KL divergence. The standard normal distribution is chosen because it is straightforward to sample from and has a simple functional form, but other distributions could be used as long as they are easy to sample from.

In each iteration, we sample parameters from the prior, train the model using these sampled parameters, and simulate the model using the trained RL solver. The simulated data is then used to train the neural network $f_\Phi()$. This training process is repeated until the neural network converges. However, this process is computationally intensive because, for each set of sampled parameters, we must first train the RL solver and then simulate the model. Although we can use the trained model from the previous iteration as the starting point for the next, the data collection process remains time-consuming.

One solution to this problem is to train a surrogate model so that we only train one big

RL solver for many different parameters. This can be done by adding the parameters as pusedo-state variables to the policy and value function. This way, we can train the model once and then use it to simulate the model for many different parameters. This is similar to the idea of amortized inference. However, this method is not working well in practice. The reason is that the policy function is very

The when inference, we can use the trained network to obtain the posterior distribution of the parameters. The following algorithm 3 shows the inference process of the SBI method.

Algorithm 3: SBI: Inference

Get the observed data x^o;

for *i in 1,2,3 ... N* **do**

  Sample z_i from $\mathcal{N}_D(\mathbf{0}, \mathbb{I})$;

  Pass $\{z_i; x^o\}$ through $f_\Phi^{-1}()$ to get θ_i;

end

Return $\{\theta_i\}_{i=1}^{i=N}$

The algorithm is based on the original paper by Radev et al. (2023). The algorithm takes the observed data x^o as input and returns the posterior distribution of the parameters. The algorithm samples z from the standard normal distribution and then passes it through the inverse of the neural network $f_\Phi()$ to obtain the posterior distribution of the parameters. This process is repeated for a large number of samples to obtain an accurate estimate of the posterior distribution. The posterior distribution can then be used to make inferences about the parameters and to compute the posterior predictive distribution of the model.

4.2 Estimation Result

The Krusell-Smith model has two sources of uncertainty: the TFP shock and the labor shock. The TFP shock follows an log $AR(1)$ process with a mean μ_{tfp}, a persistence ρ_{tfp}, and a standard deviation denoted by σ_{tfp}. The labor shock follows an log $AR(1)$ process

with a mean μ_l, a persistence ρ_l, and a standard deviation ρ_l.

I estimate the parameters of the above two log $AR(1)$ using the SBI method. The prior of the parameters is set to be uniform and the range is given in Table 5. The rest parameters is set to be the same as specified in Table 1. The estimation is done using the normalizing flow method. The details of the estimation process are described in the algorithm 2 and 3.

Table 3: Prior Range Values

Parameter	**Lower Bound**	**Upper Bound**
μ_{tfp}	-0.1	0.1
ρ_{tfp}	0.9	0.999
σ_{tfp}	0.01	0.05
μ_l	-0.1	0.1
ρ_l	0.9	0.999
σ_l	0.01	0.05

To verify the estimation result, I compare the posterior distribution of the parameters with the true parameter values I used to generate the observation.

Table 4: Prior Range Values

Parameter	**True Value**
μ_{tfp}	0.0
ρ_{tfp}	0.95
σ_{tfp}	0.03
μ_l	0.0
ρ_l	0.95
σ_l	0.03

The estimation result is shown in Figure 7. The orange vertical line represents the true parameter value that I used to generate the observation. The posterior distribution

is centered around the true value, indicating that the estimation is accurate. The width of the posterior distribution reflects the uncertainty in the estimation. The narrower the distribution, the more certain we are about the parameter value. The posterior distribution can be used to make inferences about the parameters and to compute the posterior predictive distribution of the model.

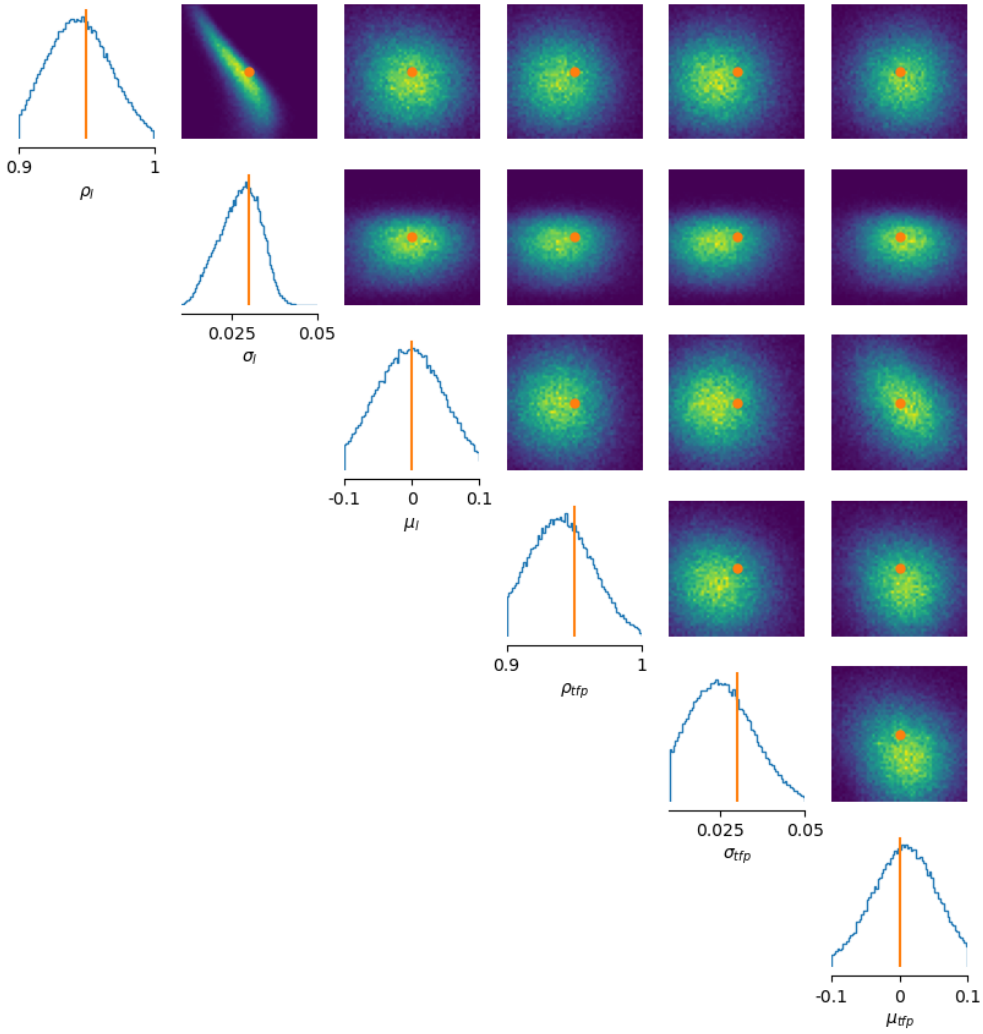

Figure 6: Posterior distribution of the parameters.

In Figure 7, the diagonal shows the marginalized posterior distribution of each parameter. The off-diagonal shows the joint posterior distribution of each pair of parameters. The shade of the color represents the density of the posterior distribution. The brighter the color, the higher the density.

Table 5: Posterior

Parameter	Posterior Mean	Posterior Var
μ_{tfp}	-0.003	2.05×10^{-03}
ρ_{tfp}	0.9445	4.68×10^{-04}
σ_{tfp}	0.0271	3.50×10^{-05}
μ_l	0.0061	1.90×10^{-03}
ρ_l	0.9411	4.57×10^{-04}
σ_l	0.0258	7.56×10^{-05}

5 Conclusion

In this paper, I introduce a new pipeline for solving and estimating heterogeneous agent models. The pipeline combines the Reinforcement Learning (RL) method for solving the model and the Simulation-Based Inference (SBI) method for estimating it. The RL method is efficient, reliable, and flexible, capable of scaling up to models with hundreds or even thousands of state variables. Moreover, it finds the global solution without additional assumptions. The SBI method is a powerful tool for estimating heterogeneous agent models by incorporating cross-sectional distributions of variables. Together, this pipeline provides a promising approach for solving and estimating heterogeneous agent models and can be applied to calibration, estimation, and further quantitative analysis.

The RL method has many potential applications. For example, it can be used to solve models with multiple assets and multiple shocks. Krusell and Smith (1997) extended the basic model to include a risk-free bond, where the mean return is zero. This scenario poses challenges for traditional methods like those proposed by Krusell and Smith (1998). If the model includes multiple, correlated asset classes, the complexity increases, even for the RL method, as the RL agent must learn the correlations between asset classes and additional elements such as bond prices. A potential solution to this complexity is multi-agent RL, where a portfolio manager agent manages multiple asset classes, and a broker

agent determines the bond price.

When estimating the model, the first question to address is whether the parameters of interest can be identified. There are two primary reasons why some parameters may lack identification. First, the likelihood function may be very flat, making it difficult to distinguish between different parameter values. In other words, small changes in parameters may lead to minimal changes in the likelihood of observing the same data. Second, the data may not be informative enough to identify certain parameters. This could be due to noisy data or the absence of specific data points. Identification is a critical issue that should be further explored in future research.

Appendices

A Benchmark

In this section, we use a simple RBC model to demonstrate that the performance of the RL approach. The RBC model is a standard model in macroeconomics that is used to study the business cycle. The model consists of a representative agent who maximizes utility subject to a budget constraint. The agent chooses consumption and saving to maximize utility subject to the budget constraint. The agent's utility function is given by $U(c) = \log(c)$, where c is consumption. The agent's budget constraint is given by $c + k' = (1 - \delta)k + y$, where k is capital, k' is next period's capital, δ is the depreciation rate of capital. The agent's production function is given by $y = zk^{\alpha}$, where y is output, z is total factor productivity which follows an $AR(1)$ process, and α is the capital share of output. The agent's Euler equation is given by $\frac{1}{c} = \beta E\left[\frac{1}{c'}(1 - \delta + \alpha z k^{\alpha-1})\right]$, where β is the discount factor, c' is next period's consumption, and E is the expectation operator. The agent's capital accumulation decision is given by $k' = (1 - \delta)k - c$.

The golden rule of solve this class of simple model is using value function iteration. The value function iteration is a dynamic programming method that solves for the value function of the agent by iterating on the Bellman equation. The Bellman equation is given by $V(k, z) = \max_{c,k'} \log(c) + \beta E[V(k', z')|z]$. We first discrete the sate variable k around its steady state ,and discrete z using method proposed by Rouwenhorst (1995). Then we iterate on the Bellman equation until convergence. The policy function is then obtained by taking the argmax of the Bellman equation. The policy function is the optimal decision rule of the agent. Value function iteration is a standard method in macroeconomics that is used to solve dynamic stochastic general equilibrium (DSGE) models. It is guaranteed to find the global solution to the model. However, it is computationally expensive and can be slow for large models.

We discrete k into 6000 grid points and z into 31 grid points. We measure the performance of the solution method using the mean squared error of the euler equation. The mean squared error of the euler equation is given by $\frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{c_i} - \beta E\left[\frac{1}{c'}(1-\delta+\alpha z k^{\alpha-1})\right]\right)^2$, where N is the number of grid points, c_i is the consumption decision of the agent at grid point i, and c' is the next period's consumption decision. The mean squared error of the euler equation is a measure of how well the policy function satisfies the euler equation. The lower the mean squared error, the better the policy function. The MSE of the value function iteration is around 10^{-7}. The RL method I proposed in this paper has a MSE of around 10^{-}, which is in the same ballpark to the value function iteration method. Value function iteration is guaranteed to find the global solution to the model, therefore, in terms of accuracy the RL method is as good as the value function iteration method. However, the RL method is much faster and more efficient (less memory usage) than the value function iteration method.

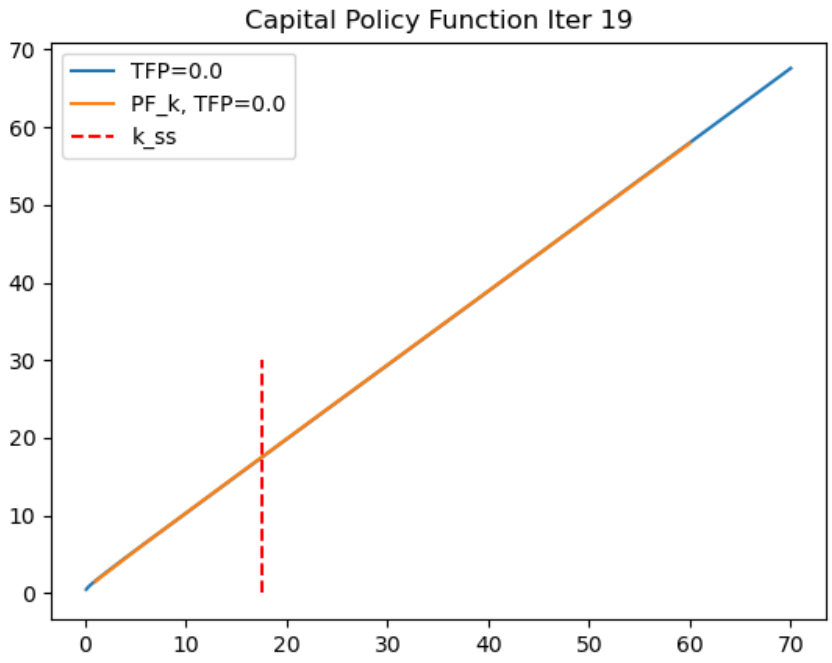

Figure 7

References

Auclert, A., Bardóczy, B., Rognlie, M., and Straub, L. (2021). Using the sequence-space jacobian to solve and estimate heterogeneous-agent models. *Econometrica*, 89(5):2375–2408.

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

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

Han, J., Yang, Y., and E, W. (2021). Deepham: A global solution method for heterogeneous agent models with aggregate shocks.

Herbst, E. P. and Schorfheide, F. (2016). *Bayesian estimation of DSGE models*. Princeton University Press.

Kase, H., Melosi, L., and Rottner, M. (2022). *Estimating nonlinear heterogeneous agents models with neural networks*. Centre for Economic Policy Research.

Krusell, P. and Smith, Jr., A. (1998). Income and wealth heterogeneity in the macroeconomy. *Journal of Political Economy*, 106(5):867–896.

Krusell, P. and Smith, A. A. (1997). Income and wealth heterogeneity, portfolio choice, and equilibrium asset returns. *Macroeconomic dynamics*, 1(2):387–422.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Liu, L. and Plagborg-Møller, M. (2023). Full-information estimation of heterogeneous agent models using macro and micro data. *Quantitative Economics*, 14:1–35.

Papamakarios, G. and Murray, I. (2018). Fast ϵ-free inference of simulation models with bayesian conditional density estimation.

Radev, S. T., Schmitt, M., Pratz, V., Picchini, U., Köthe, U., and Bürkner, P.-C. (2023). Jana: Jointly amortized neural approximation of complex Bayesian models. In Evans, R. J. and Shpitser, I., editors, *Proceedings of the Thirty-Ninth Conference on Uncertainty in Artificial Intelligence*, volume 216 of *Proceedings of Machine Learning Research*, pages 1695–1706. PMLR.

Reiter, M. (2009). Solving heterogeneous-agent models by projection and perturbation. *Journal of Economic Dynamics and Control*, 33(3):649–665.

Rouwenhorst, K. G. (1995). Asset pricing implications of equilibrium business cycle models. *Frontiers of business cycle research*, 1:294–330.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In Xing, E. P. and Jebara, T., editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China. PMLR.

Smets, F. and Wouters, R. (2007). Shocks and frictions in us business cycles: A bayesian dsge approach. *American Economic Review*, 97(3):586–606.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., Cola, G. D., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, H., and Younis, O. G. (2024). Gymnasium: A standard interface for reinforcement

learning environments.
Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2023). *Dive into Deep Learning*. Cambridge University Press. https://D2L.ai.