A Fast Deep Learning Solution for Market Clearing in Heterogeneous Agent Models

Yoshiya Yokomoto, Keio University

October 8, 2025

Abstract

This paper proposes a novel global solution method that sharply reduces the computational burden of heterogeneous agent models with aggregate uncertainty, where market prices are determined implicitly. The main bottleneck in such models is the repeated search for the market-clearing equilibrium price within each simulation period. To overcome this issue, the core of the method is to approximate the policy function with neural networks that incorporates the equilibrium price as a state variable. Once trained, this price-conditional policy function eliminates the need for repeated optimization during equilibrium searches, directly resolving the bottleneck. In addition, the neural network approach mitigates the curse of dimensionality from an expanded state space and is flexible enough to handle non-smooth policy functions, such as (S-s) rules arising from fixed adjustment costs.

Applying the method to the models of Khan and Thomas (2008) and Bloom et al. (2018) demonstrates substantial improvements in speed while preserving accuracy. For example, the Bloom et al. (2018) model, which requires six days under the Krusell-Smith method, can be solved in just 59 minutes. A further advantage is that the method preserves the skeleton of existing solution techniques like the Krusell-Smith algorithm, making it straightforward for researchers to understand and implement.

1 Introduction

This paper introduces a novel global solution method that delivers dramatic improvements in computational speed for solving heterogeneous agent models, particularly those where equilibrium prices must be determined implicitly rather than taken as a direct function of aggregates.

The classical benchmark by Krusell and Smith (1998), which has served as the foundation for many subsequent methodological papers, can be largely decomposed into two parts: solving the optimization problem and simulating the economy. However, in a broad class of heterogeneous agent models where the market-clearing price is determined implicitly—such as Krusell and Smith (1997), Khan and Thomas (2008), and Bayer et al. (2019)—an additional step arises in the simulation: root-finding to determine the equilibrium price. This feature, absent in the widely studied Krusell and Smith (1998) benchmark, makes such models significantly more challenging to solve. The root-finding procedure is itself highly demanding: for each candidate price, the optimal action for every agent must be recomputed. The need to repeat this optimization inside the price-search loop renders the simulation dramatically more expensive than solving the optimization problem itself. For instance, in Bloom et al. (2018), the 5,000-period simulation requires 434 minutes, compared to only 29 minutes for solving the optimization problem.

Hence, the key idea of the proposed method is to approximate the policy function with a neural network and make it directly conditional on the equilibrium price. In this way, the repeated re-optimization required within the price root-finding loop can be replaced by a simple forward pass of the policy network. While earlier studies also experimented with including prices as state variables[1], they typically relied on interpolation grids, which become unreliable under non-smooth policies and quickly prohibitive in higher-dimensional settings. By contrast, the neural network approximation avoids these pitfalls: it effectively captures non-linearities, and its flexible structure allows us to handle the non-smoothness of the policy function. Crucially, it is ideally suited to GPU acceleration. Millions of agents' decisions can be evaluated in parallel with virtually no extra cost, transforming the equilibrium search from a computational bottleneck into a tractable task. This transformation is the foundation for the dramatic speed improvements reported later in the paper.

This central insight underlies the broader contributions of my method, which not only delivers substantial speedups but also ensures accuracy, scalability, and applicability to more complex environments.

First, the method delivers dramatic improvements in computational speed, especially in the simulation step, while still preserving accuracy. In the relatively small-scale model of Khan and Thomas (2008), simulation time is reduced by more than two-thirds relative to the benchmark, yet the aggregate outcomes remain virtually indistinguishable. In the far more demanding model of Bloom et al. (2018), the improvement is even more striking: a simulation that takes 434 minutes under the Krusell-Smith method can be completed in just 1.3 minutes with my approach.

Second, the method tames the curse of dimensionality. In the Bloom et al. (2018) model, which expands the state space to millions of points, the time required to solve the Bellman equation remains essentially unchanged relative to the smaller Khan and Thomas (2008) case. By contrast, the KS method explodes in cost: what took only three seconds in the Khan-Thomas setting balloons to nearly 29 minutes in Bloom et al. (2018). My approach thus scales smoothly to higher dimensions

[1]The idea of including price as a state variable was previously introduced in Gomes and Michaelides (2008), Algan et al. (2014), and Favilukis et al. (2017). I will discuss the differences from these approaches below.

where traditional methods break down.

Third, the method extends the discrete-choice neural networks approach of Maliar and Maliar (2022) to solve a broad class of problems with non-smooth policies, such as those characterized by (S-s) adjustment rules. Models with fixed adjustment costs generate these non-smooth policy rules, rendering standard interpolation methods unreliable. This has forced previous work to abandon explicit policy functions and accept extremely slow simulation loops. By decomposing the firm's adjustment decision into a discrete choice (whether to act) and a conditional one (the magnitude of the action), my approach accommodates these non-smooth policies while delivering substantial speed improvements.

Fourth, the method scales impressively on modern GPUs. On an NVIDIA A100—accessible even in Google Colab—the Bloom et al. (2018) model with roughly 300 times its baseline state space can still be solved in about one-third of the time required by the KS method. This demonstrates that the approach is not only theoretically scalable but also practically feasible with widely available hardware. With GPUs offering even larger on-package memory, such as NVIDIA's H200 or Blackwell architectures, it becomes possible to handle vastly larger state spaces at realistic speeds, opening the door to applications that were previously computationally prohibitive.

Fifth, despite these advances, the method retains a relatively simple implementation. The skeleton of the Krusell-Smith algorithm is preserved, so the only substitution is to replace traditional value and policy functions with neural networks. Researchers familiar with KS-type methods can therefore adopt this approach without needing to learn an entirely new solution framework.[2]

Related Literature. First, my work builds on global solution methods for heterogeneous agent models with aggregate uncertainty, particularly the Krusell-Smith framework and its extensions. Den Haan (2010b) provides a survey of solution methods for the original Krusell and Smith (1998) model, while Terry (2017) offers a comprehensive survey specifically focused on solution methods for heterogeneous firm models such as Khan and Thomas (2008). A major challenge in heterogeneous agent models with implicit market clearing is the high computational cost of simulation, since root-finding must be performed in every period. Gomes and Michaelides (2008) and Favilukis et al. (2017) address this problem by treating the equilibrium price itself as a genuine state variable in both the policy and value functions. While this approach avoids repeated optimization during the root-finding, it comes with two important drawbacks: it increases the curse of dimensionality, and it cannot approximate models with non-smooth policy functions reliably.

As Terry (2017) and Bloom et al. (2018) emphasize, simulation is often crucial for estimation and calibration, making simulation speed improvements essential. Some methods, such as Algan et al. (2008) and Sunakawa (2020), have been developed to update forecasting rules without relying on simulation. In contrast, my approach does not aim to avoid simulation but rather to accelerate it by incorporating the equilibrium price into the policy function. Bakota (2023) proposes a different way to circumvent root-finding. Instead of computing the equilibrium price in every period, he updates the coefficients of the forecasting rules so that excess demand is driven to zero.

Second, my work is related to the Endogenous Gridpoint Method (EGM) (Carroll, 2006) and its extensions to non-smooth and non-concave problems. EGM is a powerful alternative to tackle the re-optimization problem that arises during root-finding: instead of searching for the opti-

[2]The main adjustments required are learning how to handle neural networks and performing computations in a vectorized manner rather than relying heavily on `for`-loops.

mal choice at each grid point, the method sets the grid over choice variables, while the grid for state variables is determined endogenously through the Euler equation and the budget constraint. However, the original EGM cannot be applied to non-smooth problems, because the first-order condition is no longer sufficient, and multiple optimal actions may exist. Several papers have proposed extensions to address this issue, such as Fella (2014) and Iskhakov et al. (2017). Although Fella (2014) can in principle be applied to models with fixed adjustment costs such as Bloom et al. (2018), it still requires searching for the optimal action in non-concave regions during simulation, which undermines its efficiency.

Third, while much of the existing literature applying deep learning to economics has focused on proposing new frameworks or exploring new possibilities (e.g., Fernández-Villaverde et al. (2020), Han et al. (2021), Azinovic et al. (2022), Kahou et al. (2021), Maliar et al. (2021), Kase et al. (2022), Maliar and Maliar (2022), Gu et al. (2023), Gopalakrishna et al. (2024), Valaitis and Villa (2024), Payne et al. (2025)), my contribution is different. I show a clear practical benefit: neural networks can deliver dramatic improvements in simulation speed for heterogeneous agent models with implicit market-clearing prices. Importantly, this benefit comes with only a simple modification of the familiar Krusell–Smith method, making the approach easy to adopt in practice.

Specifically, my training process is similar to Han et al. (2021), where the value and policy functions are trained separately, much like policy function iteration.[3] In the Bloom et al. (2018) application, I adopt the idea of Maliar and Maliar (2022), who apply classification techniques from the machine learning literature to discrete choice problems.

The simplicity of this framework makes it easy to incorporate techniques developed in recent years. For example, neural networks can be used for forecasting rules instead of the standard log-linear specification, as in Fernández-Villaverde et al. (2023). In addition, I employ the non-stochastic simulation method of Young (2010), following the original implementations in Khan and Thomas (2008) and Bloom et al. (2018). However, it is also possible to approximate the distribution with a finite number of agents and include it directly as a state variable, as in Maliar et al. (2021) and Azinovic et al. (2022). This approach has the advantage not only of reducing unnecessary state-space computations by using simulation-based states, but also of avoiding moment-based approximations. [4]

Organization of the paper. Section 2 applies my methodology to the heterogeneous firm model of Khan and Thomas (2008), where stochastic adjustment costs make policy rules smooth. This application highlights how My Method eliminates the costly re-optimization in simulation and thereby accelerates computation compared with the Krusell-Smith approach. Section 3 then turns to the more complex Bloom et al. (2018) model, which features a much larger state space and fixed adjustment costs that generate non-smoothness. There I describe how the method is extended to handle discrete, non-convex adjustment regimes and demonstrate its scalability to higher-dimensional settings.

[3] The objective function for the value function differs: they use cumulative utility and thereby avoid forecasting rules, whereas my method relies on the right-hand side of the Bellman equation.

[4] The generalized moments method proposed by Han et al. (2021) is also applicable in my setting, though they may be unstable due to the presence of forecasting rules.

2 Application to the Model of Khan and Thomas (2008)

In this section, I apply my proposed method to the Khan and Thomas (2008) model. Subsection 2.1 first reviews the Khan and Thomas (2008) model, then subsection 2.2 introduces the Krusell-Smith method and points out its computational problems, subsection 2.3 explains my method and subsection 2.4 presents computational results from applying my method to the Khan and Thomas (2008) model and compares them against a benchmark derived from the Khan and Thomas (2008) model code provided by Terry (2017).

2.1 Khan and Thomas (2008)

The production function employed by firms follows a standard Cobb-Douglas form:

$$y = z\epsilon k^{\alpha} N^{\nu}, \tag{1}$$

where y denotes output, z represents aggregate productivity, ϵ captures idiosyncratic firm-level productivity, k stands for capital, and N denotes labor input.
Firms face productivity shocks at both aggregate and individual levels. Specifically, the idiosyncratic productivity evolves according to an AR(1) process:

$$\log(\epsilon') = \rho_{\epsilon} \log(\epsilon) + \eta'_{\epsilon}, \quad \eta'_{\epsilon} \sim N(0, \sigma^2_{\eta_{\epsilon}}), \tag{2}$$

while aggregate productivity evolves similarly:

$$\log(z') = \rho_{z} \log(z) + \eta'_{z}, \quad \eta'_{z} \sim N(0, \sigma^2_{\eta_{z}}). \tag{3}$$

Capital evolves following the conventional law of motion:

$$k' = (1-\delta)k + i, \tag{4}$$

where δ is the depreciation rate, and i represents investment. Firms face fixed adjustment costs when altering their capital stock, which are given by a random draw ξ scaled by the equilibrium wage rate w, represented as:

$$\psi(w) = w\xi, \tag{5}$$

where ξ is an independently and identically distributed random variable drawn from distribution G over $[0, \bar{\xi}]$. As a result, we can summarize the distribution of firms over (ε, k) using the probability measure μ defined on the Borel algebra $\mathcal{S}$ for the product space $\mathbf{S} = \mathcal{E} \times \mathbf{R}_+$. The firm's optimization problem involves choosing employment n and investment k^* to maximize the lifetime profit, given their current states and adjustment costs[5]:

$$\begin{aligned} v^1(\varepsilon, k, \xi; z, \mu) = \max_{n, k^*} \Big[& z\varepsilon k^{\alpha} n^{\nu} - \omega(z, \mu)\, n + (1-\delta)k \\ & + \max \Big\{ -\xi\, \omega(z, \mu) + R(\varepsilon', k^*; z', \mu'), R(\varepsilon', (1-\delta)k; z', \mu') \Big\} \Big] \end{aligned} \tag{6}$$

[5] In the original Khan and Thomas (2008), they define a band around zero investment within which no adjustment cost is incurred. For comparability with Terry (2017), I omit this feature, following their specification.

$$R\left(\varepsilon', k'; z', \mu'\right) \equiv -k' + \mathbb{E}\left[d_j\left(z'; \mu\right) v^0\left(\varepsilon', k'; z', \mu'\right)\right],$$

$$v^0(\varepsilon, k; z, \mu) \equiv \int_0^{\bar{\xi}} v^1(\varepsilon, k, \xi; z, \mu) G(d\xi)$$

Forecasting rules,

$$\mu' = \Gamma_\mu(z, \mu)$$

On the household side, a representative agent makes consumption, labor supply, and portfolio decisions. The household maximizes its lifetime utility:

$$W(\lambda; z_i, \mu) = \max_{c, n_h, \lambda'} u(c, 1 - n_h) + \beta \sum_{j=1}^{N_z} \pi_{ij} W(\lambda'; z_j, \mu'), \tag{7}$$

subject to:

$$\begin{aligned} & c + \int_S \rho_1\left(\varepsilon', k'; z, \mu\right) \lambda'\left(d\left[\varepsilon' \times k'\right]\right) \\ & \quad \leq \omega(z, \mu) n^h + \int_S \rho_0(\varepsilon, k; z, \mu) \lambda(d[\varepsilon \times k]) \end{aligned} \tag{8}$$

utility function:

$$u(c, 1 - n_h) = \log c + \phi(1 - n_h) \tag{9}$$

Using the aggregate quantities C and N to describe the market-clearing values of household consumption and hours, we can derive the following from the household's first-order condition and property of the Stochastic Discount Factor:

$$\omega(z, \mu) = \frac{D_2 U(C, 1 - N)}{D_1 U(C, 1 - N)} \tag{10}$$

$$d_j(z, \mu) = \frac{\beta D_1 U\left(C_j', 1 - N_j'\right)}{D_1 U(C, 1 - N)} \tag{11}$$

From the FOC with respect to C, Lagrange multiplier $p(z, \mu)$ is equal to the marginal utility of consumption and we rewrite the equations above:

$$p(z, \mu) = D_1 U(C, 1 - N) \tag{12}$$

$$\omega(z, \mu) = \frac{D_2 U(C, 1 - N)}{p(z, \mu)} = \frac{\phi}{p(z, \mu)} \tag{13}$$

$$d_j(z, \mu) = \frac{\beta p(z', \mu')}{p(z, \mu)} \tag{14}$$

Using the definitions for p and d_j above, and denoting V as the value function measured in units of household marginal utility $V \equiv pv$, the Bellman equation can be rewritten as:

$$V^1(\varepsilon, k, \xi; z, \mu) = \max_{n \in \mathbf{R}_+} (z \varepsilon k^\alpha n^\nu - \omega n + (1 - \delta) k) p$$

$$+\max\left\{-\xi\omega p+\max_{k^*\in\mathbf{R}_+}R\left(\varepsilon',k^*;z',\mu'\right),R\left(\varepsilon',(1-\delta)k;z',\mu'\right)\right\} \tag{15}$$

$$R\left(\varepsilon',k';z',\mu'\right)\equiv -k'p+\beta\mathbb{E}V^0\left(\varepsilon',k';z',\mu'\right),$$
$$V^0(\varepsilon,k;z,\mu)\equiv\int_0^{\bar{\xi}}V^1(\varepsilon,k,\xi;z,\mu)G(d\xi)$$

Forecasting rule,

$$\mu'=\Gamma_\mu(z,\mu),\quad p=\Gamma_p(z,\mu)$$

Note that the second maximization in (15) reflects the firm's choice between investing to move to the new capital level k^* or remaining at the depreciated level $(1-\delta)k$. Importantly, the choice of k^* depends only on (z,ε,μ) and **not** on the firm's individual capital k.[6] The threshold level of the adjustment cost ξ that determines whether the firm invests is given by:

$$\xi^*=\frac{R(\varepsilon,k^*;z,\mu')-R(\varepsilon,(1-\delta)k;z,\mu')}{p\omega} \tag{16}$$

Equilibrium Conditions. An equilibrium represents a set of firm value functions V^1,V^0, firm policies and adjustment thresholds k^*,n,ξ^*, prices $p(z,\mu),\omega(z,\mu)$, and mappings Γ_μ,Γ_p such that:

- Firm capital adjustment choices and policies conditional upon adjustment satisfy the Bellman equations defining V^1,V^0 above, and therefore firm capital transitions are given by

$$k'(\varepsilon,k,\xi;z,\mu)=\begin{cases}k^*(\varepsilon;z,\mu'), & \xi<\xi^*(\varepsilon,k;z,\mu)\\ (1-\delta)k, & \xi\geq\xi^*(\varepsilon,k;z,\mu)\end{cases} \tag{17}$$

- The distributional transition rule used in the calculation of expectations above by firms is consistent with the aggregate evolution of the distributional state

$$\begin{gathered}\mu'(\varepsilon',k')=\Gamma_\mu(z,\mu)=\iiint I_A(\varepsilon,k)d\mu(\varepsilon,k)dG(\xi)d\Phi(\eta'_\epsilon)\\ A(\varepsilon',k',\xi,\eta'_\epsilon;z,\mu)=\{(\varepsilon,k)\mid k'(\varepsilon,k,\xi;z,\mu)=k',\log(\varepsilon')=\rho_\epsilon\log(\varepsilon)+\eta'_\epsilon\},\\ \Phi(x)=\mathbb{P}(\eta'_\epsilon\leq x)\end{gathered} \tag{18}$$

- Aggregate output, investment, and labor are consistent with the current distribution μ and firm policies:

$$Y(z,\mu)=\iint z\varepsilon k^\alpha n(\varepsilon,k,\xi;z,\mu)^\nu d\mu(\varepsilon,k)dG(\xi) \tag{19}$$

$$I(z,\mu)=\iint(k'(\varepsilon,k,\xi;z,\mu)-(1-\delta)k)d\mu(\varepsilon,k)dG(\xi) \tag{20}$$

$$N(z,\mu)=\iint n(\varepsilon,k,\xi;z,\mu)d\mu(\varepsilon,k)dG(\xi)+\int G(\xi^*(\varepsilon,k;z,\mu))d\mu(\varepsilon,k) \tag{21}$$

- Aggregate consumption satisfies the resource constraint

$$C(z,\mu)=Y(z,\mu)-I(z,\mu) \tag{22}$$

[6] Since the adjustment cost $\xi\omega p$ is independent of k, the optimal choice of k^* is also independent of it.

- The households are on their optimality schedules for savings and labor supply decisions, i.e. the first order conditions defining marginal utility and wages hold, and the price mapping is consistent

$$p(z,\mu) = \Gamma_p(z,\mu) = \frac{1}{C(z,\mu)} \tag{23}$$

$$\omega(z,\mu) = \frac{\phi}{p(z,\mu)} \tag{24}$$

- Aggregate productivity z follows the assumed AR(1) process in logs.

2.2 Krusell-Smith (KS) Method

To make the problem tractable, we follow the approach of Krusell and Smith (1998) and approximate the infinite-dimensional distribution of firms, $\mu(\varepsilon, k)$, with the aggregate capital stock, K. The Bellman equation to be solved, which is integrated over the distribution of the adjustment cost ξ, can then be written as:

$$\begin{aligned} V^0(\varepsilon,k;z,K) = \max_{n\in\mathbf{R}_+} \left(z\,\varepsilon\,k^{\alpha}\,n^{\nu} \;-\; \omega\,n \;+\; (1-\delta)\,k\right) p \;-\; \omega\,p \int_0^{\bar{\xi}} \xi\, G(\mathrm{d}\xi) \\ +\; G(\xi^*(\varepsilon,k;z,K)) \max_{k^*\in\mathbf{R}_+} R\left(\varepsilon',k^*;z',K'\right) \\ +\; \left(1-G(\xi^*(\varepsilon,k;z,K))\right)\, R\left(\varepsilon',(1-\delta)\,k;z',K'\right) \end{aligned} \tag{25}$$

where the continuation value R is given by:

$$R\left(\varepsilon',k';z',K'\right) \equiv -k'p + \beta \mathbb{E} V^0\left(\varepsilon',k';z',K'\right) \tag{26}$$

Firms form expectations about future aggregate capital, K', and the current price, p, using perceived laws of motion, which are assumed to take a log-linear form:

$$\log(K') = a_K + b_K \log(K) \tag{27}$$

$$\log(p) = a_p + b_p \log(K) \tag{28}$$

This formulation can be understood as occurring before the adjustment cost ξ is realized, is realized, so the discrete choice of whether to invest (the second max operator in equation (15)) is replaced with its expectation. The choice of labor n is a static problem determined by its first-order condition. We only need the policy function for the second max operator in (25), which is derived by maximizing (26). Here, k is not entered as a state variable.

Algorithm 1: Krusell-Smith Method Procedure

```
Initialize: Set initial forecasting rule parameters and define the state grid;
repeat
    Step 1: Solve the Bellman Equation;
    Given forecasting rules, do value function iteration;
    Step 2: Simulate the Model;
    Run T periods simulations using the obtained value function;
    Step 3: Update Forecasting Rules;
    Update parameters by regressing simulation outcomes;
until convergence condition is satisfied;
Output: Value, policy function and converged forecasting rule parameters;
```

Algorithm 1 outlines the Krusell-Smith (KS) solution procedure. The process begins with an Initialization step, where the state space grid and the coefficients of the forecasting rules are defined. The idiosyncratic productivity shocks ε and the aggregate shock z are discretized using the Tauchen method Tauchen (1986).

Step1: Solve the Bellman Equation Given the forecasting rule, the Bellman equation is solved by some method, like Value Function Iteration(VFI). In this step, value function is computed for every combination of the state variables (ε, z, k, K). Throughout this step, the future aggregate capital K' and current price p are determined by the existing forecasting rules.

Step 2: Simulate the Model Once the value function converges in Step 1, the next phase is to simulate the model's economy over a T-period horizon (e.g., 2500 periods) using the obtained value function . This simulation step is critical for generating data to update the forecasting rules but is also the most computationally demanding part. While Step 1 might be relatively quick (e.g.,$\sim$2 seconds), the simulation can take significantly longer (e.g.,$\sim$3 minutes for 2500 periods) . The primary reason for this computational expense is the need to determine the market-clearing equilibrium price p in each period of the simulation. This contrasts with the original Krusell and Smith (1998) model where the interest rate could be directly inferred from aggregate capital K. To compute a p that satisfies equilibrium conditions, we need to solve the following equation:

$$\begin{aligned} V^0(\varepsilon, k; z, K, p) = \max_{n \in \mathbf{R}_+} \big(z\, \varepsilon\, k^{\alpha}\, n^{\nu} \;-\; \omega\, n \;+\; (1-\delta)\, k \big)\, p \;-\; \omega\, p \int_0^{\bar{\xi}} \xi\, G(\mathrm{d}\xi) \\ +\; G(\xi^*(\varepsilon, k; z, K)) \max_{k^* \in \mathbf{R}_+} R\big(\varepsilon', k^*; z', K', p\big) \\ +\; \big(1 - G(\xi^*(\varepsilon, k; z, K))\big)\; R\big(\varepsilon', (1-\delta)\, k; z', K', p\big) \end{aligned} \tag{29}$$

$$R\left(\varepsilon', k'; z', K', p\right) \equiv -k'p + \beta \mathbb{E} V^0\left(\varepsilon', k'; z', K'\right) \tag{30}$$

The value function on the left-hand side of the Bellman equation now includes the price p as an additional state variable. This is a crucial distinction. The price used during the root-finding search for equilibrium is a candidate value that generally differs from the price predicted by the forecasting rule in Equation (28). Consequently, the value of a firm cannot be uniquely determined by the state $(\varepsilon, k; z, K)$ and the forecasting rules (27)-(28) alone. It is therefore necessary to treat p as a state variable for the current period's problem. This leads to the formulation in Equation (29), where the future value function, $V^0(\varepsilon_m, k'; z_j, K')$, is taken as given. We have two options to compute the optimal action for (29).

The first, which I term the **"Without Policy"** approach, involves deriving the optimal action for every guess of the price, p, during the simulation. The second, the **"With Policy"** approach, pre-computes the policy function on a grid that also includes p, allowing for faster evaluation by interpolating over both p and aggregate capital, K. The **"Without Policy"** approach is more accurate and general; however, it is also computationally slow. The reason for this slowness is explained next.

Figure 1: Comparison of simulation with and without policy

Algorithm 2: Without Policy

```
for t = 1, ..., 2500 do
    while price not conv. do
        Guess p;
        for state (ε, k) do
            for cand. k' do
                Compute RHS of
                  Bellman;
            end
            Take arg max;
        end
        Aggregate → excess demand;
        Update price bracket;
    end
end
```

Algorithm 3: With Policy

```
for t = 1, ..., 2500 do
    while price not conv. do
        Guess p;
        for state (ε, k) do
            k' = g(ε; z, K, p);
        end
        Aggregate → excess demand;
        Update price bracket;
    end
end
```

Why the "Without Policy" loop is slow. Figure 1 contrasts two inner loops. In the left panel ("Without Policy") , every guess for p triggers a fresh arg max search over k' for each idiosyncratic state (ε, k), which dominates runtime in Step 2. Let N_s be the number of idiosyncratic states on the simulation path per period, N_a the number of k' candidates, and N_b the number of bisection steps to bracket p. The per-period work scales as

$$\text{cost}_{\text{withouy-policy}} \sim \mathcal{O}\big(N_b \cdot N_s \cdot N_a\big),$$

and as $\mathcal{O}(T \cdot N_b \cdot N_s \cdot N_a)$ over T periods.

With a price-conditioned policy. In the right panel ("With Policy"), the arg max is replaced by evaluating a price-conditioned policy $k' = g(\varepsilon, z, K, p)$, giving

$$\text{cost}_{\text{with-policy}} \sim \mathcal{O}\big(N_b \cdot N_s\big)$$

per period. However, for this to work during the simulation, g must be available and accurate regarding the time-varying aggregate state. In this model, the aggregates are (K, p), so g must be precomputed on a grid over (K, p) and then interpolated. This implies high-dimensional storage and interpolation: As the number of aggregate states increases, the policy function must be evaluated over higher-dimensional grids. This leads to high-dimensional interpolation, since interpolation requires considering multiple neighbors in each aggregate dimension, and the computational

cost grows quickly with dimensionality.

Interpolation and non-smoothness. A practical complication arises from the fact that the policy function k' is often not smooth in the aggregate states. In the Khan and Thomas (2008) model, the value function and policy function are smooth, thanks to stochastic adjustment cost. Hence, we can have the policy function interpolated for K and p. However, if it's a deterministic adjustment cost, the value function is not smooth and concave. Generally, models with fixed adjustment costs or discrete choices can exhibit kinks or discontinuities in the value function. Similarly, borrowing constraints or occasionally binding constraints generate non-differentiabilities. These sources of irregularity imply that direct interpolation of k' with respect to aggregate states and p may be unreliable. To obtain accurate policies, we therefore resort to the "Without policy" formulation, where the policy function is recomputed at each iteration. While this ensures consistency, it comes at the cost of substantially slower computation.

2.3 My Method

As seen in the previous section, the problem boils down to how to solve equation (29). The problem of the "Without policy" version lies in the need to recompute optimal actions every time the price is updated. Therefore, this issue can be resolved by constructing a policy function that includes the equilibrium price itself as an input (state variable). In my method, I apply neural networks as an approximator for the policy function. This price-conditional policy function, once trained, can directly output optimal actions for any given price encountered during the simulation's bisection search, thereby eliminating the costly re-optimization bottleneck. The adoption of deep neural networks is particularly advantageous here due to their proven ability to approximate highly complex, non-linear functions in high-dimensional settings and efficient computation for the price dimension. In addition, my proposed method simply replaces the value and policy functions while preserving the skeleton of the Krusell-Smith method, making it straightforward to understand and implement.

Overall Solution Algorithm. My proposed solution method, detailed in Algorithm 4, is an iterative procedure. It starts by **initializing** the neural networks for the value function (V^0_{nn}), policy function (g_{nn}), and forecasting rules (Γ_p, Γ_μ), along with optimizers and an initial training dataset $\mathcal{D}_0$. The algorithm then enters a **Main Loop**, which iterates until the forecasting rules for the aggregate price and capital achieve convergence. Within each iteration of this main loop, two key phases are executed sequentially:

First, a **Value & Policy Iteration** phase (Step 1 in Algorithm 4) solves for the optimal policy and value functions consistent with the current forecasting rules. This phase itself is an inner loop that continues until the change in the right-hand side of the Bellman equation (denoted 'rhs' in the algorithm) is below a threshold ϵ. Inside this inner loop, the policy networks g_{nn} are trained by the **TrainPolicyNetwork** procedure, and subsequently, the value network V^0_{nn} is trained to minimize the Bellman error via the **TrainValueNetwork** procedure, both using the current training data $\mathcal{D}$. When training the policy network with **TrainPolicyNetwork**, the parameters of the value network V^0_{nn} are held fixed, and conversely, when training the value network with **TrainValueNetwork**, the parameters of the policy network g_{nn} are kept constant. In essence, this alternating update process is analogous to traditional Policy Iteration.

Algorithm 4: Solution Algorithm of the Proposed method

Data: Models $V_{nn}^0, g_{nn}, \Gamma_p, \Gamma_\mu$, initial dataset $\mathcal{D}_0$, optimizers, schedulers
Result: Trained models $V_{nn}^0, g_{nn}, \Gamma_p, \Gamma_\mu$

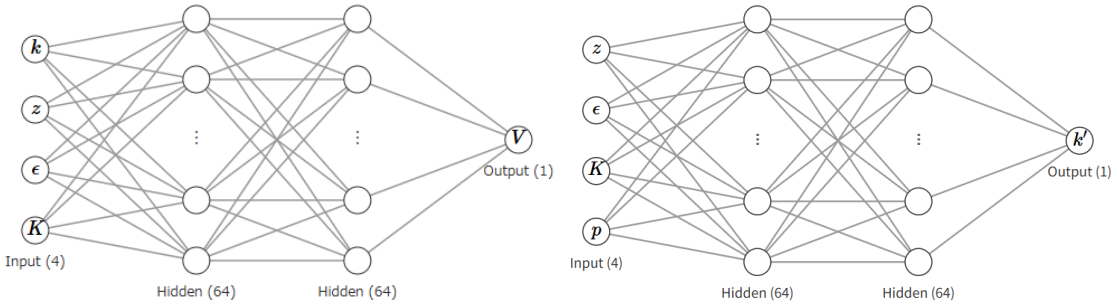
Initialization:
Initialize $V_{nn}^0, g_{nn}, \Gamma_p, \Gamma_\mu$, optimizers, schedulers
Set $\mathcal{D} \leftarrow \mathcal{D}_0$.
Main Loop (repeat until forecasting rules converge):
while $\|\Gamma_p - \Gamma_p^{old}\|_\infty > \delta_1$ *or* $\|\Gamma_\mu - \Gamma_\mu^{old}\|_\infty > \delta_1$ **do**
    1. Value & Policy Iteration (repeat until RHS change $\leq \epsilon$):
    Initialize rhs_{prev} and rhs.
    while $\|rhs - rhs_{prev}\|_\infty > \epsilon$ **do**
        $rhs_{prev} \leftarrow rhs$.
        $rhs \leftarrow$ TrainPolicyNetwork$(g_{nn}, V_{nn}^0, \mathcal{D})$
        $loss_V \leftarrow$ TrainValueNetwork$(V_{nn}^0, g_{nn}, \mathcal{D})$
    end
    2. Simulation and Forecast Update:
    $\mathcal{D}_{new} \leftarrow$ SimulateModel$(g_{nn}, \Gamma_p, \Gamma_\mu)$
    $\Gamma_p^{old} \leftarrow \Gamma_p$
    $\Gamma_\mu^{old} \leftarrow \Gamma_\mu$
    $\Gamma_p \leftarrow$ UpdatePriceForecast$(\Gamma_p, \mathcal{D}_{new})$
    $\Gamma_\mu \leftarrow$ UpdateCapitalForecast$(\Gamma_\mu, \mathcal{D}_{new})$
    $\mathcal{D} \leftarrow$ UpdateTrainingData$(\mathcal{D}, \mathcal{D}_{new})$
end
return $V_{nn}^0, g_{nn}, \Gamma_p, \Gamma_\mu$

Second, once the policy and value functions have stabilized for the current forecasting rules, a **Simulation and Forecast Update** phase (Step 2) is performed. The economic model is simulated with the converged policy g_{nn} to generate a new dataset of time series data, $\mathcal{D}_{new}$. This new data is then used to update the forecasting rule networks Γ_p and Γ_μ (through **UpdatePriceForecast** and **UpdateCapitalForecast**). Finally, the main training dataset $\mathcal{D}$ is augmented with this new simulation data using **UpdateTrainingData**[7].

The main loop then repeats, taking the newly updated forecasting rules into the next **Value & Policy Iteration** phase, until the convergence criterion for the forecasting rules (e.g., $\|\Gamma_p - \Gamma_p^{old}\|_\infty \leq \delta_1$) is met. The algorithm then returns the converged neural networks.

[7]Here I assume that the training dataset is constructed from simulated values of the aggregate variable K in order to enable more efficient learning. Alternatively, one could simply take grid points for K; in that case, updating the training dataset would not be necessary.

Figure 2: Neural Network Approximations



(a) Value Function Approximation

(b) Policy Function Approximation

Neural Network Approximations and Loss Functions. Specifically, the value and policy functions in equation (29) are approximated with neural networks as follows:

$$V^0(\varepsilon, k; z, \mu) \approx V^0_{nn}(\varepsilon, k; z, K), \tag{31}$$

$$k^* \approx g_{nn}(\varepsilon; z, K, p). \tag{32}$$

Figures 2a and 2b show snapshots of the value and policy functions, respectively. The neural networks have 2 hidden layers with 64 neurons per layer. The value function takes the state variables $(\varepsilon, k; z, K)$ directly as input and outputs the value. The policy function g_{nn} takes $(\varepsilon; z, K, p)$ as its inputs. My method incorporates the *current* price p as an additional, direct input to this policy function. This explicit conditioning on p is crucial: it allows the neural network to learn the optimal k' for any given p encountered during the simulation's price bisection search, thereby avoiding the need for repeated re-optimization. [8]

The policy function $g_{nn}(\varepsilon; z, K, p)$ is trained to output the capital stock k' that maximizes the equation (30). This is achieved by minimizing the **Policy Function Loss**:

$$\mathcal{L}_{policy} = -\frac{1}{N}\sum_{i=1}^{N}\left[-k'_{g_{nn}(\varepsilon_i; z_i, K_i, p_{error,i})} p_{error,i} + \beta \mathbb{E}_i V^0\left(\varepsilon'_i, k'_{g_{nn}(\varepsilon_i; z_i, K_i, p_{error,i})}; z'_i, K'_i\right)\right] \tag{33}$$

To calculate (33), the key inputs are the perturbed price $p_{error,i}$. The perturbed price $p_{error,i}$ is constructed by adding a uniform noise term $\boldsymbol{\epsilon}_i \in [-\boldsymbol{\delta}, \boldsymbol{\delta}]$ to the price generated from $\Gamma_p(z_i, K_i)$:

$$p_{error,i} = \Gamma_p(z_i, K_i) + \boldsymbol{\epsilon}_i. \tag{34}$$

This noise injection during training aims to make the policy function robust to the price variations encountered during the bisection search within the simulation. The range parameter $\boldsymbol{\delta}$ is chosen to cover the typical fluctuations in price guesses observed during this bisection process.[9] In Khan and Thomas (2008), I set $\boldsymbol{\delta} = 0.15$ to match the magnitude of price variation observed in their

[8] An important distinction from having a policy without neural networks is that we do not need to compute for every grid point of p, need to compute the value only just around the predicted one. This is possible thanks to the grid-off ability of neural networks.

[9] Note that the policy function is trained to be consistent with both (25)–(26) and (29)–(30). In (25)–(26), the policy should be optimized given the forecasting rules (27) and (28). Since p_{error} is constructed around the forecasting rule for the price, the loss function (33) implicitly covers the optimization problem in (25)–(26).

simulations. The parameters of g_{nn} are optimized using the ADAM optimizer to minimize $\mathcal{L}_{policy}$.

The value function V^0_{nn} is trained to satisfy the Bellman equation across the state space. The **Value Function Loss**, $\mathcal{L}_v$, quantifies the deviation from this ideal:

$$\mathcal{L}_v = \frac{1}{N}\sum_{i=1}^{N}\Big(V^0_{nn}(\varepsilon_i, k_i; z_i, K_i) - \text{RHS}_i\Big)^2. \tag{35}$$

This loss measures the mean squared error between the current network's output $V^0_{nn}(\varepsilon_i, k_i; z_i, K_i)$ and a target value, RHS_i. The term RHS_i is constructed from the right-hand side of the Bellman equation (25), representing the expected discounted value if the firm is at state (ε_i, k_i) in aggregate conditions (z_i, K_i) and follows the policy g_{nn}:

$$\begin{aligned} RHS_i = \max_{n_i \in \mathbf{R}_+} & \left(z_i\, \varepsilon_i\, k_i^{\alpha}\, n_i^{\nu} \;-\; \omega_i\, n_i \;+\; (1-\delta)\, k_i\right) p_i \;-\; \omega_i\, p_i \int_0^{\bar{\xi}} \xi\, G(\mathrm{d}\xi) \\ & +\; G(\xi^*(\varepsilon_i, k_i; z_i, K_i))\, R\big(\varepsilon_i', g_{nn}(\varepsilon_i; z_i, K_i, p_i); z_i', K_i'\big) \\ & +\; \big(1 - G(\xi^*(\varepsilon_i, k_i; z_i, K_i))\big)\; R\big(\varepsilon_i', (1-\delta)\, k_i; z_i', K_i'\big) \end{aligned} \tag{36}$$

Here, in the last term of (36), the argument of policy is $p \neq p_{error}$.

Unlike standard Value Function Iteration (VFI), where the right-hand side of the Bellman equation directly becomes the next iteration's value function, here the neural network V^0_{nn} is trained via gradient descent to minimize this squared difference $\mathcal{L}_V$. This process iteratively adjusts the network's parameters so that its output $V^0_{nn}(\text{state}_i)$ more closely aligns with the target RHS_i, effectively pushing the network to learn a representation that satisfies the Bellman optimality condition. [10]

2.4 Results

In this section, I compare the performance of my proposed method with the benchmark Krusell-Smith (KS) method. For the KS method, I use the Fortran code provided by Terry (2017). My method is implemented in Python using the PyTorch library. The parameter settings and histogram grid specifications are identical. Computations were performed on a system with a Core(TM) i7-12700F 2.10 GHz CPU and a GeForce RTX 3080 GPU. I focus on several important dimensions: (i) computation time, (ii) the Bellman equation error, (iii) the dynamics of macroeconomic variables in an unconditional simulation, (iv) key microeconomic moments of the investment rate, and (v) the accuracy of the forecast rules.

Computational Speed Comparison. First, I present computation speed. Table 1 compares computation time. The VFI and simulation times reported in the table are measured during the first iteration of the outer loop. For the simulations, both methods use a simulation length of 2,500 periods. It is worth noting that, from the second iteration onward, the VFI step in My Method benefits from using the previous iteration's results as a warm start, significantly reducing the computation time to approximately 30 seconds per iteration. Reported total times are the cumulative results of six iterations of the outer loop.

[10] Using the same network V^0_{nn} for both the current value estimates and the future values in RHS_i makes the targets move along with the predictions, which destabilizes training. To address this, we adopt the target network technique from reinforcement learning. Further implementation details are provided in the Appendix.

As shown in the table, the KS Method requires very little time for VFI but spends the majority of its computation on simulation, resulting in a total time of about 24 minutes. By contrast, my method initially incurs a higher cost in the VFI step, but the simulation is much faster. Because the VFI step quickly accelerates after the first iteration due to warm starts, the overall computation time is substantially reduced, with my method completing six iterations in 14 minutes compared to 24 minutes for the KS method. In this model, using a CPU or GPU does not affect the simulation time because the number of points in the histogram is just 250.

Table 1: Computation time comparison

Method	VFI Time (sec)	Simulation Time (sec)	Total Time (min)
KS Method (Terry (2017))	3	193	24
My Method	123	68	14

Note: VFI time refers to the value function iteration during the *first iteration* of the outer loop, while simulation time corresponds to a 2,500-period simulation in that iteration. The total time aggregates six outer-loop iterations.

Bellman Equation Error. Table 2 reports the results of a Bellman error comparison, where the error is measured as $\log V' - \log V$ for a set of state points. Both methods calculate this error across the same 250 grid points, constructed by taking 5 points for the aggregate shock z, 5 for the idiosyncratic shock ε, 10 for individual capital k, and 10 for the aggregate capital K. The table shows the **average** Bellman error over these 250 points. As shown in Table 2, My Method achieves a lower average Bellman error (0.0015) compared to the KS Method (0.0085), indicating that my neural network approximation yields a value function that satisfies the Bellman equation more accurately on average across the sampled state space.

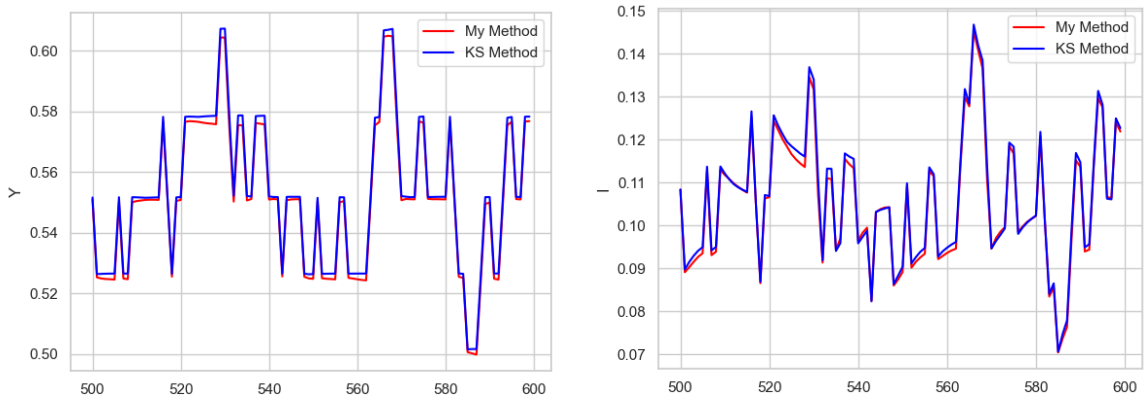
Table 2: Bellman equation error

Method	**Average Bellman Error** ($\log V' - \log V$)
KS Method (Terry (2017))	0.0085
My Method	0.0015

Note: Errors are computed at 250 state points, using 5 grid points for the aggregate shock z, 5 for the idiosyncratic shock ε, 10 for individual capital k, and 10 for aggregate capital K. This choice of grid sizes follows the implementation in Terry (2017), ensuring comparability across methods.

Unconditional Simulation Comparison. I next evaluate the performance of the approximate policies in a long-run (unconditional) simulation. Figure 3 compares the time paths of aggregate output and investment under the KS Method and My Method for the same sequence of shocks. The aggregate dynamics generated by My Method closely track those of the KS benchmark. Thus, despite relying on neural network approximations of the policy and value functions, My Method reproduces the dynamics of aggregate output and investment with high accuracy.

Figure 3: Comparison of unconditional simulation paths



(a) Aggregate output (b) Aggregate investment

Note: The figure compares unconditional simulation paths of the KS and My methods, using the same realization of aggregate shocks.

Microeconomic Investment-Rate Moments. Table 3 reports key microeconomic investment-rate moments, comparing the benchmark KS method with my proposed methods. My methods generate micro-level investment statistics that are quantitatively close to those produced by the KS benchmark.

Table 3: Microeconomic investment-rate moments

	KS Method	My Method
$\frac{i}{k}$	0.0947	0.1042
$\sigma\left(\frac{i}{k}\right)$	0.2597	0.3362
$\mathbb{P}\left(\frac{i}{k}=0\right)$	0.7693	0.7635
$\mathbb{P}\left(\frac{i}{k}\geq 0.2\right)$	0.1724	0.1606
$\mathbb{P}\left(\frac{i}{k}\leq -0.2\right)$	0.0280	0.0311
$\mathbb{P}\left(\frac{i}{k}>0\right)$	0.1890	0.1961
$\mathbb{P}\left(\frac{i}{k}<0\right)$	0.0417	0.0403

Note: Each row reports the mean value, across periods, of the indicated microeconomic moment of the cross-sectional distribution of investment rates i/k. The first row gives the average investment rate, the second its cross-sectional standard deviation, the third the probability of investment inaction, the fourth (fifth) the probability of positive (negative) investment spikes larger in magnitude than 20%, and the sixth (seventh) the probability of strictly positive (negative) investment rates. All statistics are computed from a 2,500-period unconditional simulation of the model, after discarding the first 500 periods as burn-in. The same realization of aggregate shocks is used across methods to ensure comparability.

Forecast-System Accuracy. Table 4 reports measures for the forecast of the aggregate price p and next-period aggregate capital K', comparing My Method with the KS benchmark. The KS Method attains slightly better accuracy in some metrics, but my regression-based approach delivers nearly identical performance across all measures. This indicates that the policy networks are sufficiently

Table 4: Accuracy of the forecasting rules

	p (%)		K' (%)	
	KS Method	My Method	KS Method	My Method
Den Haan Statistics				
Max	0.11	0.49	0.39	0.89
Mean	0.06	0.08	0.25	0.23
Root Mean Squared Error (RMSE)				
RMSE	0.05	0.09	0.05	0.09
Forecast Regression R^2				
R^2	0.9998	0.9994	0.9995	0.9996

Note: The top panel reports Den Haan (2010a) statistics: the maximum and mean percentage differences between realized values and dynamic forecasts. The middle panel reports the root mean squared error (RMSE), in percentages, based on static forecasts. The bottom panel reports the R^2 values from forecast regressions, reflecting the explanatory power of the forecast rules conditional on aggregate productivity. All statistics are computed using the same exogenous aggregate productivity series across methods. Values for the KS Method are reproduced from Terry (2017).

well-trained so as not to generate unstable dynamics, while preserving the accuracy of the forecasting system.

3 Application to the Model of Bloom et al. (2018)

In this section, I demonstrate that the efficiency of my method becomes even more pronounced in larger models. I apply my method to the model of Bloom et al. (2018). The basic structure of this model is similar to that of Khan and Thomas (2008), but Bloom et al. (2018) additionally incorporates both aggregate and idiosyncratic stochastic volatility shocks for firm productivity, as well as nontrivial capital and labor adjustment costs, which introduce non-smoothness into policy function. Below, I provide a more detailed overview of the production environment and demonstrate how my neural-network-based solution can handle the non-smoothness and large state space more efficiently than traditional methods.

3.1 Model

The production side closely follows Khan and Thomas (2008): each firm produces with a Cobb–Douglas technology in capital and labor.

$$y = z\varepsilon k^{\alpha} n^{\nu}, \tag{37}$$

Productivity Processes. Both aggregate and idiosyncratic productivity follow persistent stochastic processes with time-varying volatility. Specifically:

- **Aggregate Productivity:**

$$\log(z') = \rho_z \log(z) + \sigma_z u'_z, \quad u'_z \sim N(0,1) \tag{38}$$

- **Idiosyncratic Productivity:**

$$\log(\varepsilon') = \rho_\varepsilon \log(\varepsilon) + \sigma_\varepsilon u'_\varepsilon, \quad u'_\varepsilon \sim N(0,1) \tag{39}$$

The time-varying volatilities, σ_z and σ_ε capture switches between "low-uncertainty" and "high-uncertainty" regimes, typically modeled as states governed by a two-state Markov chain. These aggregate uncertainty states $(\sigma_z, \sigma_\varepsilon)$ become part of the aggregate state space.

Capital Dynamics and Adjustment Costs. Capital evolves according to the standard law of motion $k' = (1 - \delta_k)k + i$. Investment is subject to adjustment costs AC^k, which include a fixed cost proportional to output and a term capturing partial irreversibility:

$$\text{AC}^k = \mathbf{1}_{|i|>0}\, y\, F^K + S|i|\, \mathbf{1}_{i<0} \tag{40}$$

Here, $\mathbf{1}_{(\cdot)}$ is an indicator function, F^K is the fixed disruption cost parameter, and S is the resale loss fraction for disinvested capital.

Labor Dynamics and Adjustment Costs. Labor (hours worked) also faces adjustment frictions and evolves as:

$$n = (1 - \delta_n)\, n_{-1} + s, \tag{41}$$

where n_{-1} is the previous period's labor, δ_n is an exogenous separation rate and s represents net hiring (or firing). Labor adjustment costs AC^n include a fixed cost proportional to output and a variable cost related to hiring/firing flow:

$$\text{AC}^n = \mathbf{1}_{|s|>0}\, y\, F^L + |s|\, H\, w \tag{42}$$

where F^L is the fixed disruption cost for labor adjustment, second term is a linear hiring/firing cost, which is expressed as a fraction of the aggregate wage w. Crucially, because labor adjustment depends on the previous period's labor input, n_{-1} is an additional endogenous state variable for the firm.

Firm Problem and State Space. Following the approach in Khan and Thomas (2008), we define the $p(z, \sigma_z, \sigma_\varepsilon, \mu)$ as the household's marginal utility of current consumption. The wage $w(z, \sigma_z, \sigma_\varepsilon, \mu)$ is also derived from the household's first-order conditions. This definition of p allows us to re-define the firm's value function in terms of these marginal utility units, $\tilde{V} \equiv pV$. The Bellman equation for $\tilde{V}$ can then be expressed as follows:

$$\tilde{V}(k, n_{-1}, \varepsilon; z, \sigma_z, \sigma_\varepsilon, \mu) = \max_{i,n} \Big\{ p(z, \sigma_z, \sigma_\varepsilon, \mu) \Big[y - w(z, \sigma_z, \sigma_\varepsilon, \mu) n - i - \text{AC}^k - \text{AC}^n \Big]$$

$$+ \beta \mathbb{E}\left[\tilde{V}(k', n, \varepsilon'; z', \sigma_z', \sigma_\varepsilon', \mu')\right]\Bigg\}. \tag{43}$$

This value function $\tilde{V}$ depends on the firm's individual state (k, n_{-1}, ε) and the aggregate state $(z, \sigma_z, \sigma_\varepsilon, \mu)$, where μ represents the distribution of firms over their idiosyncratic states and $\sigma_z, \sigma_\varepsilon$ represent the current volatility regime. Solving the model requires tracking the evolution of the distribution μ and the equilibrium price p via forecasting rules:

$$\begin{aligned} \mu' &= \Gamma_\mu(z, \sigma_z, \sigma_\varepsilon, \mu), \\ p &= \Gamma_p(z, \sigma_z, \sigma_\varepsilon, \mu). \end{aligned} \tag{44}$$

Bloom et al. (2018) solve this model using the Krusell-Smith method, approximating the infinite-dimensional distribution μ with aggregate capital K. The forecasting rules (44) are thus typically approximated as functions of the aggregate state variables $(z, \sigma_z, \sigma_\varepsilon)$ and aggregate capital K, i.e., $K' = \Gamma_K(z, \sigma_z, \sigma_\varepsilon, K)$ and $p = \Gamma_p(z, \sigma_z, \sigma_\varepsilon, K)$.

In the original code, the firm's problem is solved by discretizing the state space. The number of grid points for each state variable, corresponding to the order of individual states (k, n_{-1}, ε) and aggregate states relevant for the firm's decision and forecasting $(z, \sigma_z, \sigma_\varepsilon, K)$, is shown below:

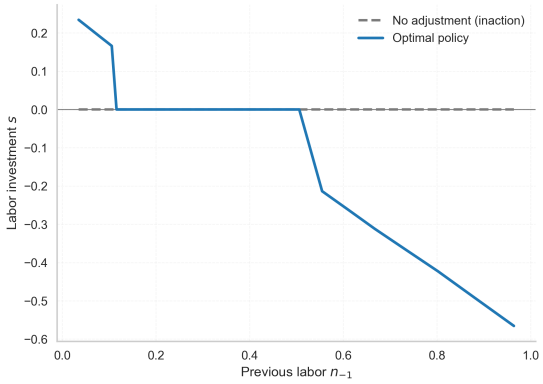
$$\underbrace{91}_{\substack{\text{grid points} \\ \text{for } k}} \times \underbrace{37}_{\substack{\text{grid points} \\ \text{for } n_{-1}}} \times \underbrace{5}_{\substack{\text{grid points} \\ \text{for } \varepsilon}} \times \underbrace{5}_{\substack{\text{grid points} \\ \text{for } z}} \times \underbrace{2}_{\substack{\text{states} \\ \text{for } \sigma_z}} \times \underbrace{2}_{\substack{\text{states} \\ \text{for } \sigma_\varepsilon}} \times \underbrace{10}_{\substack{\text{grid points} \\ \text{for } K}} = 3{,}367{,}000 \quad \text{points.}$$

Here, σ_z and σ_ε represent the two possible volatility regimes (e.g., high/low), and the 10 points for K represent the discretization of aggregate capital used to approximate the distribution μ in the forecasting rules. This large state space poses a significant computational challenge.

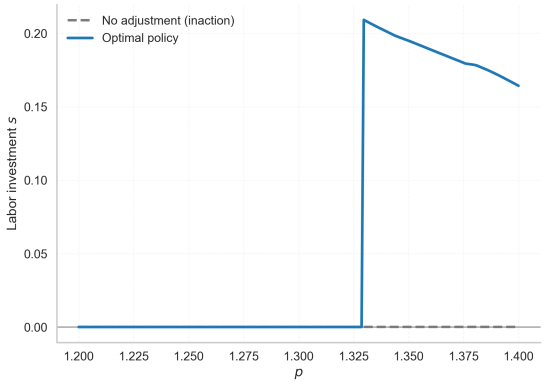
Non-smooth Adjustment Costs. The model includes several features that make the firm's decision-making process non-smooth. Specifically, there are fixed costs for adjusting capital or labor, a penalty for selling capital (partial irreversibility), and linear hiring/firing costs. Together, these create kinks and sudden jumps in the firm's optimal policy. A firm will only pay a fixed cost to change its capital or labor if the benefit from that change is large enough. If the desired adjustment is small, the firm will choose to do nothing, a behavior known as *inaction*. In this case, the firm simply allows its capital to depreciate and its workforce to shrink through natural employee separations. This leads to a classic (S–s) adjustment rule:

Figure 4 illustrates this (S–s) pattern for labor investment. **Figure (a)** shows how labor adjustment changes depending on the firm's previous labor level (n_{t-1}), while **Figure (b)** shows the adjustment as a function of the price (p). In both plots, a wide region of inaction is clearly visible (the flat line at zero). This period of inaction is broken by sharp, sudden jumps when a threshold is crossed. These features demonstrate the non-smooth nature of the policy. Because of these jumps, we cannot use standard solution methods that rely on smooth interpolation to approximate the firm's decisions.

Figure 4: Evidence of an S–s type adjustment rule in labor investment



(a) Labor adjustment as a function of previous labor n_{t-1}



(b) Labor adjustment as a function of price p

3.2 Decomposing the Single max: Discrete and Conditional Policies

Following and extending the discrete choice approach of Maliar and Maliar (2022), I address the non-smoothness induced by fixed and partially irreversible adjustment costs (i.e., the S–s rule) by decomposing the policy into two neural networks. The first is a **discrete policy function** g_d, which selects whether to adjust capital and/or labor. The second is a **conditional policy function** g_c, which, given the discrete choice, determines the continuous policies (k', n). Both networks are parameterized by neural networks. This decomposition turns the original problem of a single maximization into a two-stage decision: (i) a discrete adjustment decision, (ii) a conditional continuous choice.

Necessity of two policy functions. The reason for using two separate policy functions is that a single policy function, like the one shown in Figure 2b for the Khan and Thomas (2008) model, generating both k' and n cannot adequately capture the S–s rule. For example, if the optimal decision is not to invest in capital, the policy must return exactly $(1 - \delta_k)k$. Even a slight deviation from this value would trigger a fixed adjustment cost. However, neural networks generally cannot reproduce such an exact value, making it impossible for a single network to fully represent the discontinuous structure of the decision problem. An alternative approach could be to impose a threshold rule: if investment is below a certain level, it is treated as zero investment. Yet this method is problematic in cases like Bloom et al. (2018), where the grid range for labor n is very narrow (from 0.035 to 0.95). In such settings, a small misclassification caused by thresholding leads to significant distortions, making the threshold rule unreliable. This limitation motivates the decomposition into a discrete adjustment policy g_d and a conditional continuous policy g_c.

From a single maximization to four discrete policy branches. With fixed and partially irreversible adjustment costs, the original Bellman equation (43)—a *single* maximization over continuous controls (k', n)—can be equivalently rewritten as a maximum over four **discrete policy branches**, corresponding to the binary decision of whether to adjust capital and/or labor:

$$\tilde{V}(k, n_{-1}, \varepsilon; z, \sigma_z, \sigma_\varepsilon, \mu) = \max\{\, R^{00},\ R^{10},\ R^{01},\ R^{11} \,\}. \tag{45}$$

Each R^{ab} is the Bellman RHS evaluated under a particular branch specification (a indicates capital adjustment, b labor adjustment; 0 = no, 1 = yes). Let $x \equiv (k, n_{-1}, \varepsilon; z, \sigma_z, \sigma_\varepsilon, \mu)$ and $k_0 \equiv (1-\delta_k)k$, $n_0 \equiv (1-\delta_n)n_{-1}$. Then, the four branches are:

$$R^{00} = p\Big[y(k, n_0, \varepsilon; z) - wn_0\Big] + \beta\,\mathbb{E}\Big[\tilde{V}\big(k_0,\ n_0,\ \varepsilon';\ z', \sigma_z', \sigma_\varepsilon', \mu'\big)\Big], \quad \text{(no investment, no labor adj.)} \tag{46}$$

$$R^{10} = \max_{k'}\ p\Big[y(k, n_0, \varepsilon; z) - wn_0 - i - \mathrm{AC}^k(i)\Big] + \beta\,\mathbb{E}\Big[\tilde{V}\big(k',\ n_0,\ \varepsilon';\ z', \sigma_z', \sigma_\varepsilon', \mu'\big)\Big], \quad \text{(capital adj., no labor adj.)} \tag{47}$$

$$R^{01} = \max_{n}\ p\Big[y(k, n, \varepsilon; z) - wn - \mathrm{AC}^n(s)\Big] + \beta\,\mathbb{E}\Big[\tilde{V}\big(k_0,\ n,\ \varepsilon';\ z', \sigma_z', \sigma_\varepsilon', \mu'\big)\Big], \quad \text{(no capital adj., labor adj.)} \tag{48}$$

$$R^{11} = \max_{k', n}\ p\Big[y(k, n, \varepsilon; z) - wn - i - \mathrm{AC}^k(i) - \mathrm{AC}^n(s)\Big] + \beta\,\mathbb{E}\Big[\tilde{V}\big(k',\ n,\ \varepsilon';\ z', \sigma_z', \sigma_\varepsilon', \mu'\big)\Big], \quad \text{(capital adj., labor adj.)} \tag{49}$$

Thus, the *outer* maximization $\max\{\cdot\}$ (45) corresponds to the **discrete policy** $g_d(x, p)$, that is, the choice of which adjustment regime to follow. Within each branch, the *inner* maximization delivers the branch-specific **conditional policy** $g_c(x, p|ab)$ for (k', n).

Discrete policy. The discrete policy network g_d maps the state (x, p) into a probability distribution over the four adjustment regimes $ab \in \{00, 10, 01, 11\}$ as illustrated in Figure 5a. Formally,

$$\pi_{ab}(x, p) = g_d(x, p), \qquad \sum_{ab} \pi_{ab}(x, p) = 1,$$

where $\pi_{ab}(x, p)$ denotes the predicted probability of choosing branch ab. At evaluation time, the branch is selected greedily as

$$ab^* = \arg\max_{ab} \pi_{ab}(x, p).$$

This discrete policy g_d therefore selects whether capital and/or labor are adjusted. Inaction is implemented exactly: if ab^* specifies no capital adjustment ($a = 0$) or no labor adjustment ($b = 0$), the corresponding decision is forced to $(k', n) = (k_0, n_0)$ along that dimension, regardless of the conditional policy output.

Branch-specific conditional policies. For each branch $ab \in \{00, 10, 01, 11\}$, the continuous decision rule is represented by the **conditional policy network**, which maps the state (x, p) and the

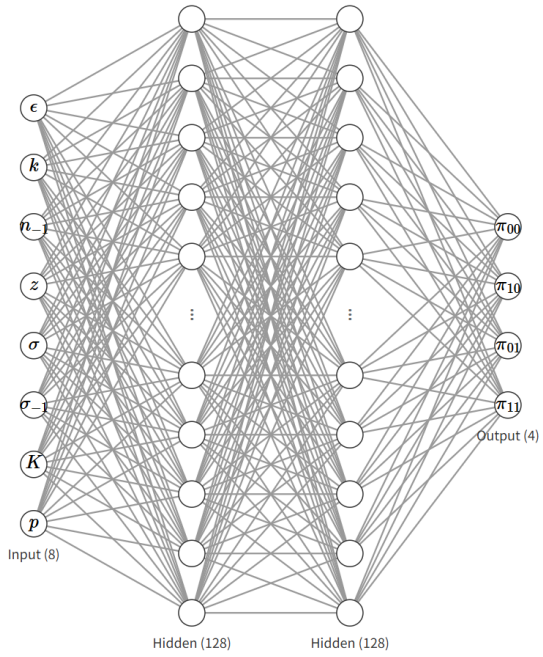
branch indicator ab into a policy pair (k', n). Formally:

$$\begin{aligned}
&\textbf{Branch } 00 \textbf{ (inaction):} && (k', n) = g_c(x, p \mid 00) = (k_0,\, n_0). \\
&\textbf{Branch } 10 \textbf{ (capital-only):} && (k', n) = g_c(x, p \mid 10) = \big(\arg\max_{k'}\{\text{RHS in (47)}\},\, n_0\big). \\
&\textbf{Branch } 01 \textbf{ (labor-only):} && (k', n) = g_c(x, p \mid 01) = \big(k_0,\, \arg\max_{n}\{\text{RHS in (48)}\}\big). \\
&\textbf{Branch } 11 \textbf{ (both):} && (k', n) = g_c(x, p \mid 11) = \arg\max_{k', n}\{\text{RHS in (49)}\}.
\end{aligned}$$

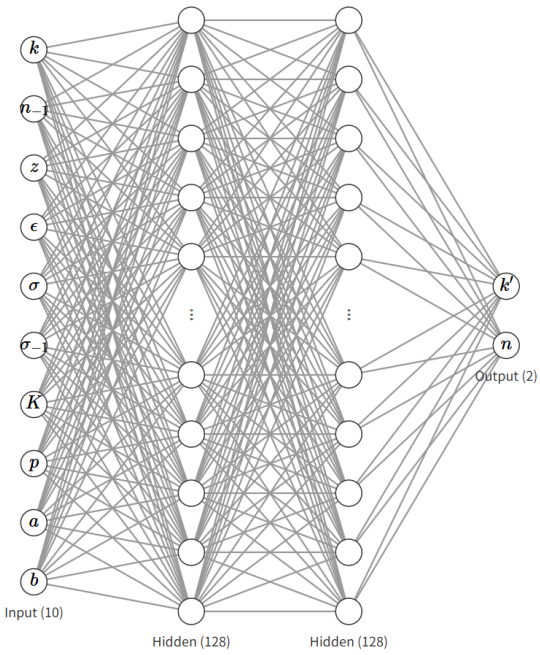
Once a branch ab is chosen, the **conditional policy** implements the corresponding continuous choice:

$$(k', n) = g_c(x, p \mid ab^*(x, p)).$$

Figure 5: Neural network architectures for discrete and conditional policies.



(a) Discrete policy network g_d

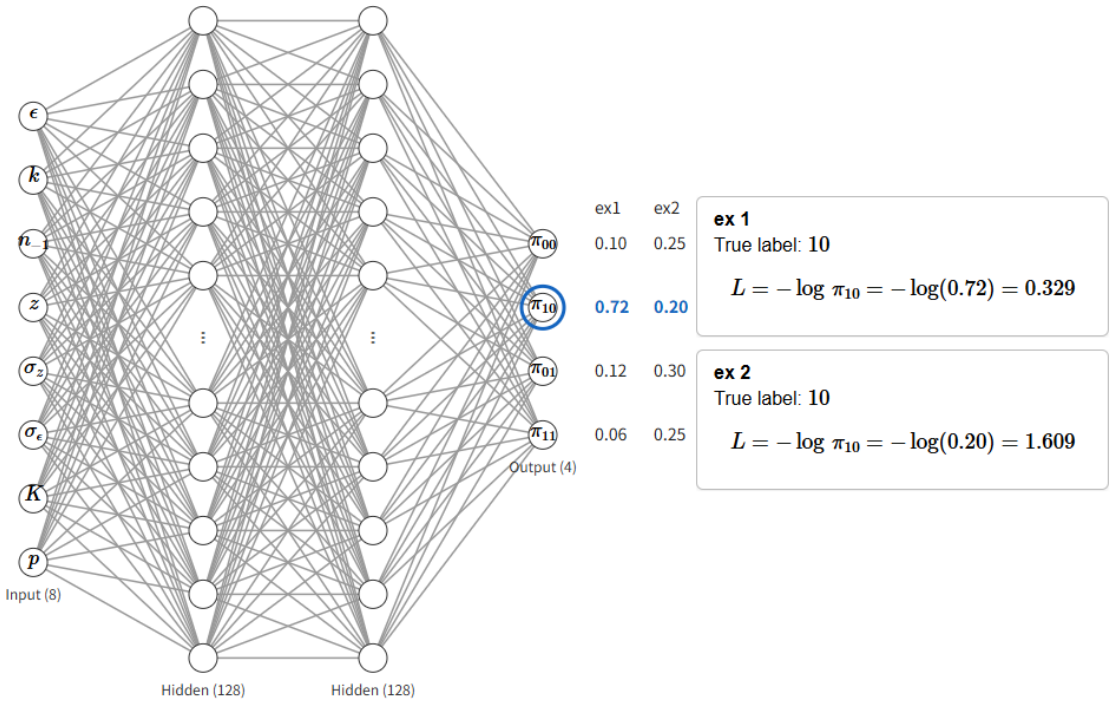


(b) Conditional policy network g_c

Figure 5 illustrates the two complementary networks. The discrete policy network g_d (panel a) takes the firm's state as input and classifies it into one of the four adjustment regimes. The conditional policy network g_c (panel b) then receives both the state and the chosen regime, and outputs the branch-specific continuous policy (k', n). This decomposition ensures that non-smoothness across regimes (due to fixed costs and irreversibilities) is handled by g_d, while smooth optimization within each regime is captured by g_c.

Training the Discrete Policy. The training of the discrete policy g_d, which determines which branch $ab \in \{00, 10, 01, 11\}$ to select, is handled as a typical **classification problem** in deep learning. Maliar and Maliar (2022) introduce this approach for modeling employment–unemployment

Figure 6: Training of the Discrete policy network



choice in the Krusell and Smith (1998) model with indivisible labor. In this paper the objective is to train the network a mapping from the economic state (x, p_{error}) to one of the four discrete choices. Here, p_{error} is used again to handle the range of bisection, in the same spirit as in the previous section for the Khan and Thomas (2008) model.

To perform this supervised learning, we first generate the **correct labels** y^* for the network to imitate, derived directly from the economic model. Based on the principle of optimality in dynamic programming, an agent will choose the branch that maximizes the RHS. Formally,

$$y^*(x, p_{\text{error}}) \in \underset{ab \in \{00,10,01,11\}}{\text{argmax}} R^{ab}(x, p_{\text{error}}).$$

Here, $R^{ab}(x, p_{\text{error}})$ is the one-period return plus the continuation value, evaluated under the policy of branch ab. The branch that yields the highest value becomes the optimal action for that state, i.e., the "correct label" $y^*(x, p_{\text{error}})$. Using these labels, the discrete policy g_d is trained to minimize the **cross-entropy loss**:

$$\mathcal{L}_{\text{disc}} = -\frac{1}{N} \sum_{i=1}^{N} \log \pi_{y_i^*}(x_i, p_{\text{error},i}),$$

where $\pi_{y^*}(x, p_{\text{error}})$ denotes the predicted probability of choosing branch ab. If the model assigns high probability to the correct branch (e.g. $\pi_{y^*} = 0.99$), the penalty $-\log(0.99)$ is nearly zero. Conversely, if the predicted probability is small (e.g. $\pi_{y^*} = 0.01$), the penalty $-\log(0.01)$ is very large, prompting a strong correction. This procedure is equivalent to **maximum likelihood estimation**, as it maximizes the likelihood of the correct labels given the data.

Figure 6 illustrates this training mechanism. The panel shows the discrete policy network g_d, which takes the state $(\varepsilon, k, n, z, \sigma_z, \sigma_\varepsilon, K, p_{\text{error}})$ as input and outputs probabilities over the four

branches $(\pi_{00}, \pi_{10}, \pi_{01}, \pi_{11})$. The panel presents two examples where the true optimal branch is 10. When the model assigns a high probability to branch 10 (e.g. $\pi_{10} = 0.72$), the loss is small (0.329). When the probability is low (e.g. $\pi_{10} = 0.20$), the loss is much larger (1.609). This visualization highlights how the cross-entropy loss penalizes low confidence in the correct branch and drives g_d toward imitating the optimal discrete decision rule.

Training the Conditional Policy. Conditional on the branch ab predicted by the discrete policy $g_d(x, p_{\text{error}})$, the conditional policy $g_c(x, p_{\text{error}} \mid ab)$ is trained to maximize the sampled RHS:

$$\mathcal{L}_{\text{cont}} \;=\; -\frac{1}{N}\sum_{i=1}^{N} R^{\,g_d(x_i,p_{\text{error},i})}\big(x_i, p_{\text{error},i}\big).$$

The input to g_c consists of the state (x, p_{error}) and the binary indicators (a, b) corresponding to the branch $ab = g_d(x, p_{\text{error}})$.[11] This setup allows the network to learn conditional responses across different regimes of inaction and adjustment.

Value network and schedule. Given the set of state variables (x_i, p_i) and K_i' from the forecasting rules (43), the conditional policy provides

$$(k_i', n_i) \;=\; g_c\big(x_i, p_i \mid g_d(x_i, p_i)\big).$$

Using this policy, we construct the target for training the value function:

$$RHS_i \;=\; p_i\Big[y_i - wn_i - (k_i' - k_{0,i}) - \text{AC}_i^k - \text{AC}_i^n\Big] \;+\; \beta\,\tilde{V}_{nn}(x_i'),$$

where $\tilde{V}_{nn}(x_i')$ is the value function evaluated at the next-period state.
The value network $\tilde{V}_{nn}$ is trained by minimizing the mean-squared error against this target.

$$\mathcal{L}_{\text{val}} \;=\; \frac{1}{N}\sum_{i=1}^{N}\Big(\tilde{V}_{nn}(x_i) - RHS_i\Big)^2.$$

Training alternates across the three components in sequence:

$$\text{CONDITIONAL POLICY} \;\rightarrow\; \text{DISCRETE POLICY} \;\rightarrow\; \text{VALUE NETWORK}.$$

3.3 Result

In this section, I compare the performance of my method with the results obtained using the code provided by Bloom et al. (2018). The computations were carried out on a system equipped with a Core™ i7-12700F 2.10 GHz CPU, a GeForce RTX 3080 GPU, and an A100 GPU for validation. The results are based on single precision (FP32). As discussed in Appendix B, the differences from double precision are slight. It is important to note that the benchmark results for Bloom et al. (2018) reported here are based on a modified version of their original code, specifically concern-

[11] In principle, one could randomize (a, b) during training. However, for efficiency, I use the gate output g_d to avoid training on unlikely adjustment regimes.

ing tracking the distribution in the simulation. Consequently, these results may differ from those reported in the published version of Bloom et al. (2018). [12]

Computation and Simulation Speed. Table 5 presents the VFI time, simulation time and total time for each method. As the table shows, my method achieves a significant reduction in computation time compared to the KS Method. In total, my method completes in about 59 minutes, whereas the KS Method requires roughly six days. The total time highly depends on the quality of initial coefficients of forecasting rules, but it mostly requires around 20 to 25 iterations with their original initial values. Simulations were run for 5,000 periods for the KS Method and My Method. It is also worth noting that, while my simulations are GPU-accelerated, running them on the CPU alone increases the simulation time to approximately 12 minutes. Unlike the results for the Khan and Thomas (2008) model in the previous section, I observe a substantial difference in the Value Function Iteration (VFI) time as well. This suggests that applying neural networks as an approximator is effective.

Table 5: Speed comparison for the Bloom et al. (2018) model

Method	VFI Time (min)	Simulation Time (min)	Total Time (min, 21 outer loops)
KS Method	29	434	9212 ($\approx$ 6 days)
My Method	2.2	1.3	59

Note: VFI time refers to the time required to solve the Bellman equation. Simulation time is measured for 5,000 periods during the first simulation. Total time is approximated as $(\text{VFI Time} + \text{Simulation Time}) \times 21$, but in practice it is shorter since VFI uses the previous value function, forecasting rules improve, and both steps speed up as the outer loop progresses.

Sources of Speed Improvement and Scalability. The dramatic reduction in simulation time, particularly evident in Table 5, stems from two key features of my method. Firstly, as discussed previously, incorporating the equilibrium price p as a state variable into the policy function eliminates the need for repeated optimization of firm actions within the price bisection loop in each simulation period. Once trained, the policy network provides an instantaneous mapping from the state (including the guessed price) to the optimal action, bypassing the computationally intensive maximization step.

Secondly, I exploit GPU acceleration for the simulation. In the Bloom et al. (2018) model, the distribution grid (ε, k, n_{-1}) for firms consists of 16,835 (5,91,37) points at baseline. During each root-finding step, policies must be computed for all grid points, and once the root-finding concludes, the distribution mass must also be updated. Given the large number of points involved, efficient parallel computation is indispensable.

This is precisely where the neural-network parameterization of the policy function becomes advantageous. A forward pass amounts to nothing more than matrix multiplications with the

[12] In their original code, they set a threshold of 1e-4 on the mass of firms in the distribution grid to determine which points to update. They ignore points with a mass below the threshold to reduce computation time. In the modified version, I update all points in the distribution grid regardless of their mass, and the results reported in this paper are based on this modified version. One could replicate the result by setting `disttol` = 0.0 and `GEerrorswitch` = 2 in their code.

Table 6: Computation time (in seconds) for **1,000-period** simulations.

Scale factor (Total points)	128 × 2 layers			128 × 3 layers		
	CPU (i7-12700F)	GPU (RTX 3080)	GPU (A100)	CPU (i7-12700F)	GPU (RTX 3080)	GPU (A100)
1× (16,835)	1,430 s	15 s	15 s	2,080 s	17 s	16 s
5× (84,175)	7,000 s	40 s	35 s	9,200 s	53 s	46 s
10× (168,350)	12,200 s	71 s	62 s	19,300 s	97 s	84 s
20× (336,700)	25,600 s	136 s	118 s	35,600 s	187 s	164 s
40× (673,400)	50,000 s	260 s	220 s	61,500 s	310 s	285 s
80× (1,346,800)	90,000 s	470 s	370 s	104,000 s	580 s	480 s
100× (1,683,500)	113,200 s	**2,360 s**	470 s	132,000 s	**2,300 s**	590 s

Note: CPU times are measured directly up to 5×, then scaled from 50-period runs. GPU times are measured directly up to 20×, then scaled in the same way. A100 results are measured on Google Colab. The RTX 3080 has 10 GB of GPU memory, while the A100 provides 40 GB. The scale factor is defined relative to the histogram size used in the original Bloom et al. (2018) code, which is based on the state variables (k, n_{-1}, ε). At baseline (1×), the histogram consists of 16,835 points with dimensions $(91, 37, 5)$. Scaling increases the grid along k and n_{-1} while holding ε fixed; for example, at 10× the histogram has dimensions $(91 \times 5,\ 37 \times 2,\ 5)$. Since the number of histogram points determines simulation speed, the aggregate variables are not altered when applying the scale factor.

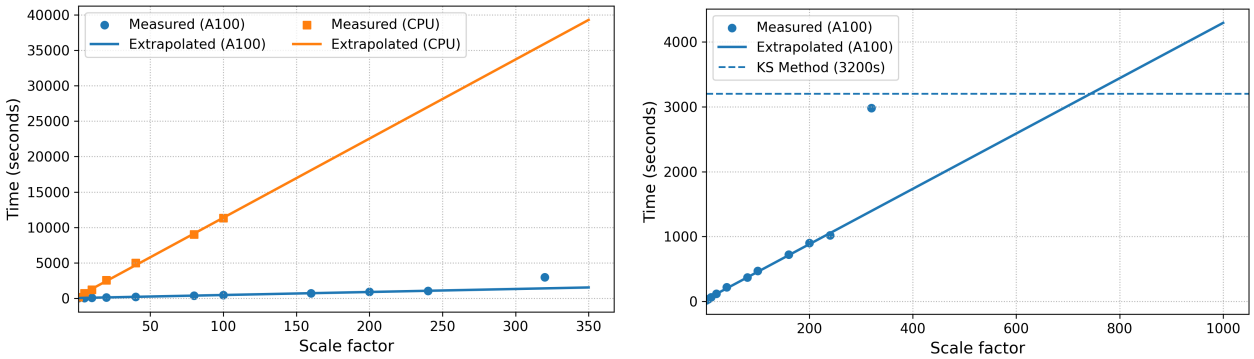
learned parameters, an operation that is highly GPU-efficient. All state vectors across the distribution can be stacked into a single batch and passed through the network simultaneously, yielding the policies for all firms in one step. In Algorithm 3, this already eliminates the costly *argmax* loop; with GPU batch evaluation, even the explicit loop over (ε, k) grid points disappears.

Scalability Results. Table 6 reports computation times for 1,000-period simulations under different scale factors of the distribution grid. Both CPU and GPU performance are reported for two neural network configurations (128 × 2 layers and 128 × 3 layers). The results clearly demonstrate the advantage of GPU acceleration: while CPU runtimes increase rapidly with scale, GPUs (both RTX 3080 and A100) achieve near-linear scaling and remain orders of magnitude faster. For instance, at the 80× case with more than 1.3 million grid points, CPU times reach nearly 90,000 seconds, whereas GPU times remain below 500 seconds. At the 100× case, however, the RTX 3080 hits its memory ceiling, and runtime rises sharply to over 2,000 seconds, whereas the A100 continues to scale smoothly. This highlights both the effectiveness of GPU acceleration and the importance of on-package memory capacity for handling very large-scale simulations.

Figures 7(a)(b) provide a visual representation of these scalability patterns. (1) Figure 7a compares CPU and GPU performance directly. The GPU runs are consistently much faster than the CPU, and the gap widens as the grid size increases. (2) Figure 7b focuses on the A100. The figure reports measured runtimes up to 320×, which reaches the practical memory limit of about 40,GB on Google Colab's A100. The fitted line is drawn using all measured points except for the 320× case, which is excluded from the extrapolation. While the linear extrapolation line demonstrates that runtime scales almost proportionally with the number of histogram points, the measured 320× case shows a sharp slowdown (about 2,980 seconds) relative to the extrapolation. This illustrates how performance deteriorates once the GPU memory ceiling is approached. With GPUs that offer larger on-package memory, linear scaling extends much further.[13] Extrapolation suggests that at around 750× (12,626,250 points), runtime would match the KS Method baseline of about 3,200 seconds, consistent with near-linear scaling when memory is not the bottleneck. In

[13]For example, NVIDIA's H200 provides 141 GB HBM3e memory with 4.8 TB/s bandwidth. The Blackwell-based DGX B200 system includes 1,440 GB total GPU memory across 8 GPUs.

Figure 7: Scalability of simulation runtime (FP32)



(a) CPU vs GPU (A100)

(b) A100 scaling

Note: Results are based on single precision (FP32). (a) Comparison of CPU (i7-12700F) and GPU (A100), showing much better scalability on the GPU. (b) A100 runtimes scale roughly linearly with problem size until the 40 GB memory limit is approached. The fitted line is based on all cases except the 320× run, which slows down due to memory pressure. The dashed line indicates the KS Method baseline (3,200s). Extrapolation suggests that at about 750× scale, the runtime would match the KS Method with CPU.

this case, once the aggregate variables are taken into account, the overall state space amounts to roughly 2.4 billion points.

Policy Micro Accuracy Check. Table 7 evaluates the micro-level accuracy of the *discrete policy* g_d. Recall from Section 3.2 that the Bellman problem can be decomposed into four mutually exclusive adjustment regimes, indexed by $ab \in \{00, 10, 01, 11\}$: no adjustment (00), capital-only adjustment (10), labor-only adjustment (01), and joint adjustment (11). The table reports the unconditional frequency of each branch under My Method and the KS Method, together with their Difference. The close alignment of branch frequencies demonstrates that the discrete classifier g_d successfully reproduces the micro-level allocation patterns implied by the KS Method, with only minor deviations across regimes.

Table 7: Micro-level accuracy of the discrete policy g_d

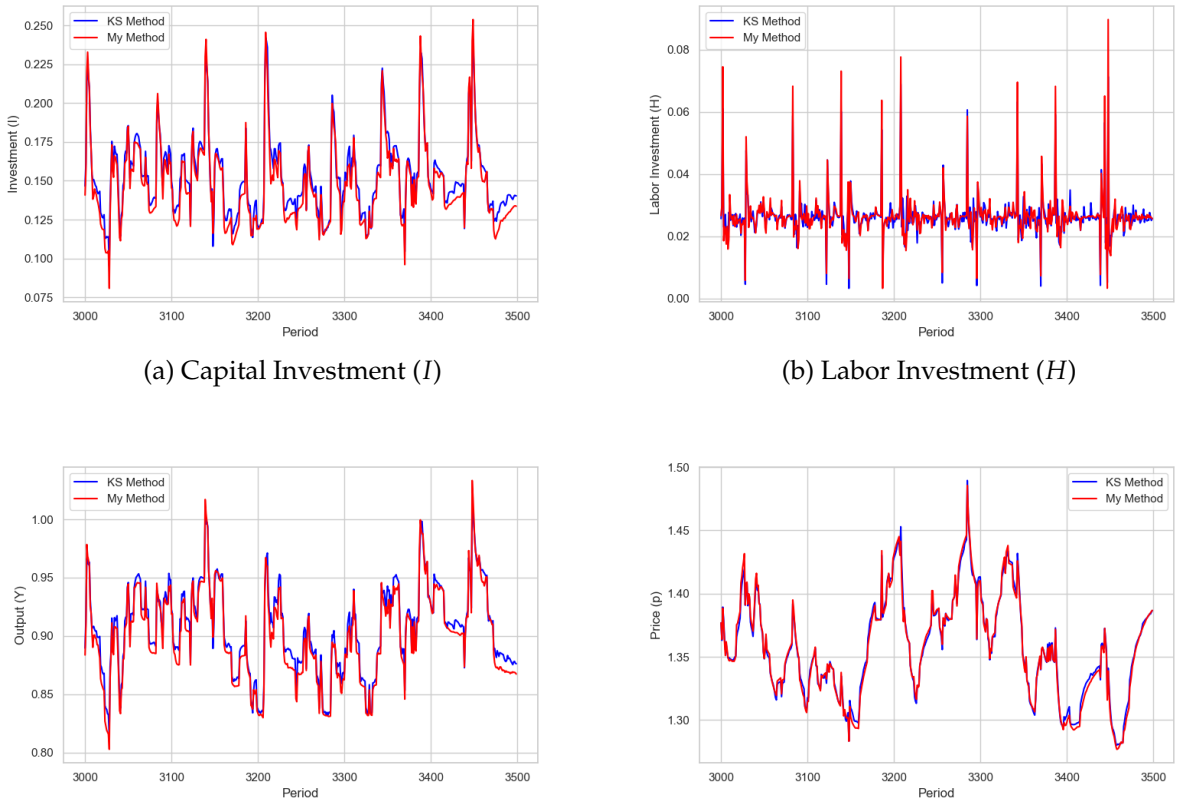
Branch ab	My Method	KS Method	Difference
00 (no adj.)	0.2165	0.1947	0.0218
10 (capital)	0.3082	0.3327	−0.0245
01 (labor)	0.2272	0.2186	0.0086
11 (both)	0.2481	0.2540	−0.0059

Note: Entries are unconditional branch shares under My Method and the KS Method, with their Difference reported in the last column. The evaluation is based on all 336,700 combinations of state variables obtained by fixing aggregate capital K and enumerating over the full grid for the remaining states ($n_{-1}, \varepsilon, z, \sigma_z, \sigma_\varepsilon$, and k).

Unconditional Simulation Comparison. Figure 8 reports unconditional simulation paths for aggregate variables over periods 3000–3500. The results confirm that, beyond computational gains, the new method replicates the qualitative and quantitative dynamics of the baseline model in un-

conditional simulations. Minor differences appear at high frequency, but the aggregate behavior remains well aligned. This provides evidence that the approximation introduced by my method preserves the economic content of the model, while enabling a substantial reduction in runtime documented in the previous section.

Figure 8: Unconditional simulation comparison



(a) Capital Investment (I)

(b) Labor Investment (H)

(c) Output (Y)

(d) Price (p)

Note: The figure compares unconditional simulation paths over periods 3000–3500. Blue lines denote the KS Method and red lines denote My Method.

Business-Cycle Properties. Table 8 reports the HP-filtered standard deviation, the ratio to output, and the correlation with output for key macroeconomic variables. I compare the results obtained from the KS Method with those from my method.[14] Overall, the results closely match those of the KS Method.

Internal Accuracy of Forecast Systems. Table 9 presents the internal accuracy of the forecasting rules for the aggregate price p and next-period aggregate capital K'. I report the Den Haan statistics (maximum and mean), the root mean squared error (RMSE), and the R^2 from forecast

[14] Because of the characteristics of GPU computation (often referred to as non-determinism), it is difficult to reproduce exactly the same results across runs. The results of my method are therefore reported as the average over 10 runs. The standard deviations of the statistics in Table 8 are 0.09, 0.61, 0.07, and 0.14 for Output, Investment, Consumption, and Hours, respectively.

Table 8: HP-filtered business-cycle statistics

	KS Method			My Method		
	Std	Ratio	Corr	Std	Ratio	Corr
Output	1.978	1.000	1.000	1.906	1.000	1.000
Investment	10.026	5.068	0.950	10.418	5.463	0.934
Consumption	0.873	0.441	0.278	0.872	0.457	0.243
Hours	2.419	1.223	0.820	2.139	1.122	0.820

Note: Statistics are based on the same realization of aggregate and stochastic volatility shocks, discarding the first 500 periods. Each series is expressed in log, the HP filter with smoothing parameter $\lambda = 1600$ is applied, and the cyclical component is multiplied by 100 to express deviations in percent.

regressions. Both My Method and the benchmark KS Method achieve high accuracy in forecasting the aggregate price and next-period capital. The Den Haan statistics, RMSE, and R^2 values are comparable, with My Method often showing slightly better performance than the KS Method. Appendix A.2 provides detailed forecast accuracy statistics broken down by aggregate shock state for the Bloom et al. (2018) model application (see Table 11).

Table 9: Accuracy of the forecasting rules

	p (%)		K' (%)	
	KS	My Method	KS	My Method
Den Haan Statistics				
Maximum	3.52	3.11	5.89	5.82
Mean	0.87	1.01	1.75	2.21
Root Mean Squared Error				
RMSE	0.45	0.47	0.56	0.26
Forecast Regression R^2				
R^2	0.978	0.976	0.980	0.996

Note: The table reports Den Haan statistics (maximum and mean), root mean squared error (RMSE, %), and R^2 from forecast regressions, comparing the KS benchmark and my method.

4 Conclusion

This paper develops a novel global solution method that addresses the significant computational burden of heterogeneous agent models with aggregate uncertainty, particularly those where equilibrium prices are determined implicitly. The method integrates deep learning techniques into the widely-used Krusell-Smith framework through two core innovations. First, it approximates the value and policy functions with neural networks, crucially including the equilibrium price as an explicit state variable in the policy function. Second, to handle models with non-smooth policies arising from features like fixed adjustment costs, it introduces a specialized architecture that decomposes the firm's problem into a discrete choice over adjustment regimes and a continuous choice within that regime. This dual approach directly tackles the primary computational bottlenecks. The price-conditional policy function bypasses the need for repeated, costly optimization during the search for market-clearing prices in each simulation period. Furthermore, the discrete-continuous policy structure effectively captures the kinks and discontinuities inherent in models with (S-s) type rules—a challenge for traditional grid-based interpolation methods. Combined with GPU acceleration, this strategy yields dramatic reductions in computation time.

The method's practical benefits are demonstrated through applications to the models of Khan and Thomas (2008) and the more complex model of Bloom et al. (2018). The results show substantial speed improvements; for instance, the Bloom et al. (2018) model, which requires six days to solve with the standard method, is solved in just 59 minutes. This efficiency is achieved while maintaining high standards of accuracy, validated by Bellman equation errors, the replication of key micro- and macroeconomic moments, and the internal consistency of the forecasting rules.

While this paper focuses on heterogeneous firm models, the method's scope is much broader, applying generally to any heterogeneous agent model with implicit market-clearing conditions. It can readily accommodate settings with multiple market clearing prices—often seen in multi-asset or HANK frameworks—simply by including all relevant prices as inputs to the neural networks. Indeed, the larger and more complex the model, the more pronounced the advantages of this approach become.

The computational gains offered by this method have significant implications for macroeconomic research. By making the analysis of richer, more realistic models computationally feasible, it allows for the inclusion of features like multiple shocks, non-convexities, and complex adjustment costs that were previously prohibitive. This facilitates more rapid policy experiments, extensive sensitivity analysis, and robust model calibration. Crucially, by preserving the overall structure of the Krusell-Smith algorithm, the method remains accessible and straightforward for researchers to implement.

References

Algan, Y., O. Allais, and W. J. Den Haan (2008). Solving heterogeneous-agent models with parameterized cross-sectional distributions. *Journal of Economic Dynamics and Control 32*(3), 875–908.

Algan, Y., O. Allais, W. J. Den Haan, and P. Rendahl (2014). Solving and simulating models with heterogeneous agents and aggregate uncertainty. *Handbook of computational economics 3*, 277–324.

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

Bakota, I. (2023). Market clearing and krusell-smith algorithm in an economy with multiple assets. *Computational Economics 62*(3), 1007–1045.

Bayer, C., R. Luetticke, L. Pham-Dao, and V. Tjaden (2019). Precautionary savings, illiquid assets, and the aggregate consequences of shocks to household income risk. *Econometrica 87*(1), 255–290.

Bloom, N., M. Floetotto, N. Jaimovich, I. Saporta-Eksten, and S. J. Terry (2018). Really uncertain business cycles. *Econometrica 86*(3), 1031–1065.

Carroll, C. D. (2006). The method of endogenous gridpoints for solving dynamic stochastic optimization problems. *Economics letters 91*(3), 312–320.

Den Haan, W. J. (2010a). Assessing the accuracy of the aggregate law of motion in models with heterogeneous agents. *Journal of Economic Dynamics and Control 34*(1), 79–99.

Den Haan, W. J. (2010b). Comparison of solutions to the incomplete markets model with aggregate uncertainty. *Journal of Economic Dynamics and Control 34*(1), 4–27.

Favilukis, J., S. C. Ludvigson, and S. Van Nieuwerburgh (2017). The macroeconomic effects of housing wealth, housing finance, and limited risk sharing in general equilibrium. *Journal of Political Economy 125*(1), 140–223.

Fella, G. (2014). A generalized endogenous grid method for non-smooth and non-concave problems. *Review of Economic Dynamics 17*(2), 329–344.

Fernández-Villaverde, J., S. Hurtado, and G. Nuno (2023). Financial frictions and the wealth distribution. *Econometrica 91*(3), 869–901.

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

Gomes, F. and A. Michaelides (2008). Asset pricing with limited risk sharing and heterogeneous agents. *The Review of Financial Studies 21*(1), 415–448.

Gopalakrishna, G., J. Payne, and Z. Gu (2024). Asset pricing, participation constraints, and inequality. *Participation Constraints, and Inequality (November 8, 2024)*.

Gu, Z., M. LauriÃ, S. Merkel, J. Payne, et al. (2023). Deep learning solutions to master equations for continuous time heterogeneous agent macroeconomic models. Technical report.

Han, J., Y. Yang, et al. (2021). Deepham: A global solution method for heterogeneous agent models with aggregate shocks. *arXiv preprint arXiv:2112.14377*.

Iskhakov, F., T. H. Jørgensen, J. Rust, and B. Schjerning (2017). The endogenous grid method for discrete-continuous dynamic choice models with (or without) taste shocks. *Quantitative Economics 8*(2), 317–365.

Kahou, M. E., J. Fernández-Villaverde, J. Perla, and A. Sood (2021). Exploiting symmetry in high-dimensional dynamic programming. Technical report, National Bureau of Economic Research.

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

Khan, A. and J. K. Thomas (2008). Idiosyncratic shocks and the role of nonconvexities in plant and aggregate investment dynamics. *Econometrica 76*(2), 395–436.

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

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

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

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

Payne, J., A. Rebei, and Y. Yang (2025). Deep learning for search and matching models. *Available at SSRN 5123878*.

Sunakawa, T. (2020). Applying the explicit aggregation algorithm to heterogeneous macro models. *Computational Economics 55*(3), 845–874.

Tauchen, G. (1986). Finite state markov-chain approximations to univariate and vector autoregressions. *Economics letters 20*(2), 177–181.

Terry, S. J. (2017). Alternative methods for solving heterogeneous firm models. *Journal of Money, Credit and Banking 49*(6), 1081–1111.

Valaitis, V. and A. T. Villa (2024). A machine learning projection method for macro-finance models. *Quantitative Economics 15*(1), 145–173.

Young, E. R. (2010). Solving the incomplete markets model with aggregate uncertainty using the krusell–smith algorithm and non-stochastic simulations. *Journal of Economic Dynamics and Control 34*(1), 36–41.

A Detailed Forecast Accuracy

A.1 Detailed Forecast Accuracy Statistics: Khan and Thomas (2008) Model

This section provides detailed forecast accuracy statistics for the forecasting rules used in the Khan and Thomas (2008) model application, broken down by aggregate productivity state (A_t). Table 10 compares the benchmark KS Method with my method.

Table 10: Shock-specific forecast accuracy

	Price p		Capital K'	
Statistic	KS	My Method	KS	My Method
Den Haan Statistics (%)				
Maximum	0.11	0.49	0.38	0.89
Mean	0.05	0.08	0.23	0.23
Root Mean Squared Error (RMSE) (%)				
$A = A_1$	0.06	0.12	0.06	0.12
$A = A_2$	0.05	0.11	0.05	0.10
$A = A_3$	0.05	0.05	0.05	0.04
$A = A_4$	0.04	0.08	0.05	0.11
$A = A_5$	0.04	0.09	0.05	0.12
Forecast Regression R^2				
$A = A_1$	1.0000	0.9934	1.0000	0.9987
$A = A_2$	1.0000	0.9942	1.0000	0.9990
$A = A_3$	1.0000	0.9990	1.0000	0.9998
$A = A_4$	1.0000	0.9963	1.0000	0.9982
$A = A_5$	1.0000	0.9948	1.0000	0.9980

Note: The table reports forecasting accuracy statistics conditional on aggregate shock A_t. Den Haan statistics (maximum and mean), root mean squared errors (RMSE), and forecast regression R^2 are shown separately for price p and next-period capital K'. All values are expressed as percentage points of log deviations (original values multiplied by 100).

A.2 Detailed Forecast Accuracy Statistics: Bloom et al. Model

This section provides detailed forecast accuracy statistics for the Bloom et al. (2018) model application, broken down by the aggregate state (A, S, S_{-1}), representing discretized grid points for aggregate productivity, current uncertainty, and lagged uncertainty. Table 11 compares the benchmark Krusell-Smith (KS) method with my method. RMSE values are multiplied by 100.

Table 11: Shock-specific forecast accuracy

Aggregate State	Price p				Capital K			
(A, S, S_{-1})	KS		My Method		KS		My Method	
	RMSE (%)	R^2	RMSE (%)	R^2	RMSE (%)	R^2	RMSE (%)	R^2
Den Haan Statistics								
Maximum	3.52		3.11		5.82		5.25	
Mean	0.87		1.01		2.21		2.22	
(1,0,0)	0.61	0.83	0.64	0.80	0.35	0.98	0.32	0.98
(1,0,1)	0.16	0.97	0.12	0.99	0.03	1.00	0.06	0.99
(1,1,0)	0.30	0.94	0.27	0.93	0.47	0.95	0.52	0.98
(1,1,1)	0.31	0.92	0.34	0.92	0.12	1.00	0.14	1.00
(2,0,0)	0.55	0.85	0.59	0.84	0.34	0.98	0.35	0.98
(2,0,1)	0.16	0.99	0.29	0.97	0.08	1.00	0.15	1.00
(2,1,0)	0.39	0.89	0.26	0.91	0.25	0.98	0.25	0.98
(2,1,1)	0.35	0.94	0.30	0.96	0.11	1.00	0.16	1.00
(3,0,0)	0.45	0.92	0.53	0.87	0.30	0.99	0.28	0.99
(3,0,1)	0.18	0.99	0.31	0.98	0.09	1.00	0.17	1.00
(3,1,0)	0.21	0.98	0.27	0.97	0.44	0.98	0.40	0.98
(3,1,1)	0.33	0.96	0.27	0.86	0.11	1.00	0.18	1.00
(4,0,0)	0.53	0.87	0.60	0.85	0.34	0.99	0.30	0.99
(4,0,1)	0.21	0.98	0.11	0.99	0.08	1.00	0.19	0.99
(4,1,0)	0.24	0.97	0.34	0.94	0.44	0.97	0.38	0.99
(4,1,1)	0.36	0.93	0.25	0.96	0.12	1.00	0.21	0.99
(5,0,0)	0.57	0.86	0.57	0.87	0.33	0.99	0.30	0.98
(5,0,1)	0.21	0.97	0.09	0.99	0.17	1.00	0.25	1.00
(5,1,0)	0.31	0.96	0.40	0.92	0.11	1.00	0.08	1.00
(5,1,1)	0.32	0.94	0.27	0.96	0.11	1.00	0.22	1.00

Note: Table reports the forecasting accuracy statistics conditional on (A_t, S_t, S_{t-1}). Den Haan statistics (maximum and mean), root mean squared errors (RMSE, %), and forecast regression R^2 are shown separately for price p and next-period capital K'. All values are expressed as percentage points of log deviations (original values multiplied by 100).

B Numerical Precision (FP32 vs FP64)

This appendix documents how floating-point precision affects my results and when FP32 (single precision) can be safely used. Floating-point numbers are an approximate way of representing real numbers on a computer. Their accuracy depends on the *machine epsilon* (the smallest gap between two numbers that the system can tell apart): about 10^{-7} for FP32 and about 10^{-16} for FP64. Arithmetic is not exact—$(a + b) + c$ may give a different answer than $a + (b + c)$—and this problem is worse when calculations are split across many processors. Adding very small numbers to very large ones can also cause the small terms to disappear (a problem known as cancellation).

These effects are usually minor in small or medium problems, but they become important when a model must add up many very small contributions.

Why FP32 can be risky in large-scale discretizations. The simulation uses histogram-based transition of a cross-sectional distribution. When the state space becomes very large, the mass at each grid point may become extremely small. Under FP32, many of these masses are close to the effective numerical resolution; consequently, (i) they can be rounded to zero during updates, (ii) their contributions may vanish in parallel reductions, and (iii) sparsity patterns can change with thread scheduling. This affects equilibrium price calculation, and the error can accumulate over time along the simulated path.

What the data say in my setting. Despite the above risk, for the Bloom et al. (2018) scale considered here, FP32 performs well. Table 12 compares HP-filtered business-cycle statistics under the KS benchmark, my method in FP32, and my method in FP64 (computed on an NVIDIA A100). Figure 9 overlays the simulated time series of key aggregates under FP32 and FP64 on the same shock realization. The paths largely overlap; while there are minor discrepancies around sharp movements, their magnitude is small enough that it is difficult to attribute them cleanly to GPU non-determinism (e.g., reduction order and kernel scheduling) versus precision per se. Overall, the macro dynamics and forecasting accuracy are robust to FP32 at the Bloom et al. (2018) scale in my implementation.

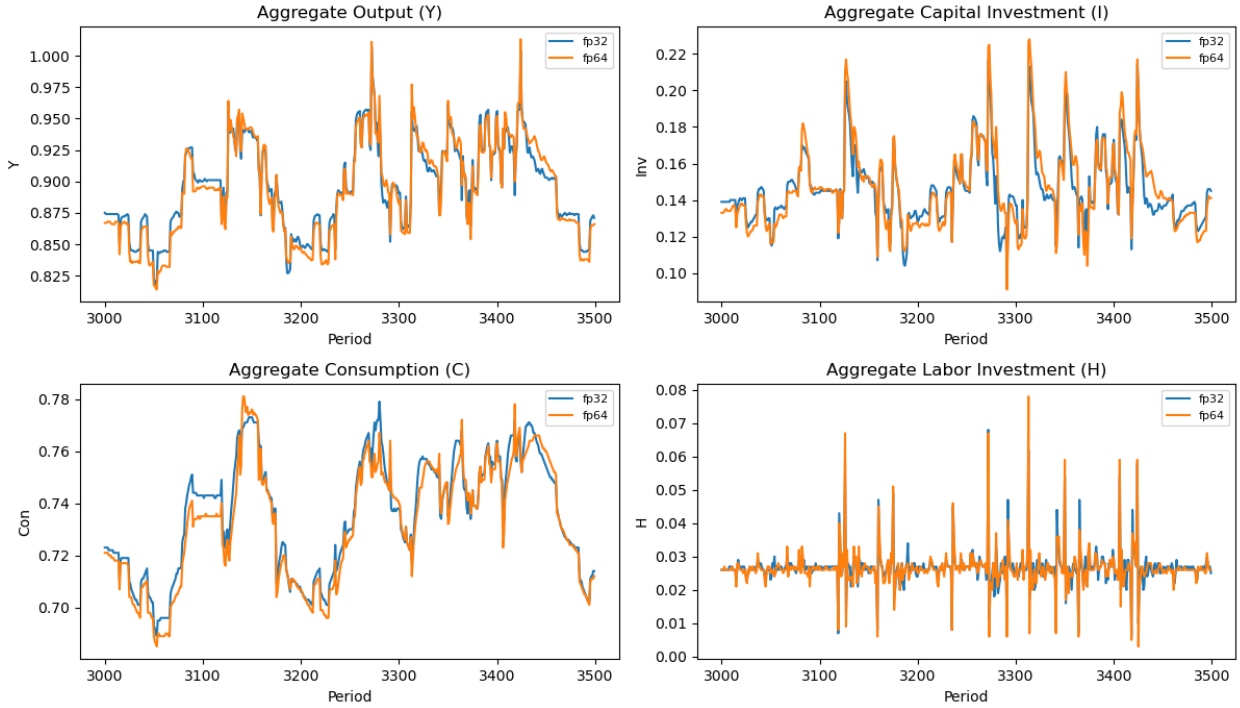
Hardware-dependent throughput. While FP64 is numerically safer for very large histograms, its throughput depends strongly on hardware. On my A100 setup, FP64 achieved comparable effective runtime to FP32 for the experiments reported here (owing to high FP64 throughput and memory bandwidth), so using FP64 did not reduce practical speed for these runs. By contrast, on an RTX 3080 the FP64 path ran at roughly 1/20 of the FP32 speed in my tests. Thus, the recommended precision is hardware- and scale-dependent.

Table 12: HP-filtered business-cycle statistics

	KS Method			My Method (FP32)			My Method (FP64)		
	Std	Ratio	Corr	Std	Ratio	Corr	Std	Ratio	Corr
Output	1.978	1.000	1.000	1.906	1.000	1.000	1.892	1.000	1.000
Investment	10.026	5.068	0.950	10.418	5.463	0.934	10.049	5.311	0.927
Consumption	0.873	0.441	0.278	0.872	0.458	0.243	0.915	0.484	0.259
Hours	2.419	1.223	0.820	2.139	1.122	0.820	2.290	1.210	0.799

Note: Statistics are based on the same realization of aggregate and stochastic volatility shocks, discarding the first 500 periods. Each series is expressed in log, the HP filter with smoothing parameter $\lambda = 1600$ is applied, and the cyclical component is multiplied by 100 to express deviations in percent. The FP64 results were computed on an NVIDIA A100 GPU.

Figure 9: FP32 vs. FP64: Overlay of simulated aggregate series



C Deep Learning Hyperparameters

This section details the hyperparameters used for training the neural network approximations of the value function (V^0_{nn}) and policy function (g_{nn}) in the two main model applications presented in the paper. The specific settings for the Khan and Thomas (2008) and Bloom et al. (2018) models are summarized in Table 13.

Table 13: Neural Network Hyperparameters

Hyperparameter	Khan-Thomas (2008)	Bloom et al. (2018)
Network Architecture (Value/Policy)	2 hidden layers, 64 neurons each	2 hidden layers, 128 neurons each
Optimizer	ADAM	ADAM
Learning Rate Schedule	Decay $1 \times 10^{-3} \to 1 \times 10^{-5}$	Decay $1 \times 10^{-3} \to 1 \times 10^{-5}$
Batch Size (Value Function)	256	256
Batch Size (Policy Function)	256	256
Outer Loop Iterations (Max)	6	21
Target Network Update Rate (τ)	0.05	0.05

The learning rate was gradually decayed over the training steps within each outer loop iteration, starting from 1×10^{-3} down to 1×10^{-5}. The target network update rate τ corresponds to the parameter in the soft update rule $\theta_{target} \leftarrow \tau\theta_{main} + (1 - \tau)\theta_{target}$.