Accelerating the Krusell-Smith Algorithm with Deep Learning

Yoshiya Yokomoto*

February 2026

Abstract

This paper accelerates the Krusell–Smith algorithm for heterogeneous-agent models using deep learning. The key bottleneck in the standard method is that simulation requires numerically finding the market-clearing price in every period, repeatedly solving agents' optimization problems for each price guess. I eliminate this bottleneck by approximating value and policy functions with neural networks that take the market-clearing price as an input. Once trained, the network evaluates policies for any price with a simple forward pass, removing repeated optimization during simulation. This greatly speeds up computation while preserving the Krusell–Smith structure. A further advantage is that neural networks can flexibly approximate non-smooth and discontinuous policies, such as S-s rules arising from fixed adjustment costs, where standard grid-based methods struggle. In the Bloom et al. (2018) model, the method reduces computation time from six days to 102 minutes with essentially identical accuracy.

Keywords. Neural networks; Deep learning; Heterogeneous agent models; Aggregate uncertainty

JEL Classification. C63, C68, E22, E32

---

*Keio University. Email: yoshiyayokomoto@gmail.com.
I am indebted to my adviser Tatsuro Senga for his invaluable guidance and support. I am also grateful for comments and suggestions from Ji Huang, Hidehiko Matsumoto, Jacob Røpke, Takashi Shiono, Yucheng Yang, Jan Žemlička and participants in the Deep Learning for Dynamic Stochastic Models Conference in Turin. This research is supported by a Koizumi Memorial Fund Subsidy. All remaining errors are mine.

1 Introduction

Heterogeneous firm models are widely used for studying the macroeconomic implications of micro-level heterogeneity among plants and firms; however, analyzing cyclical dynamics using this class of models with aggregate uncertainty poses a significant computational challenge. As established by Terry (2017), the Krusell-Smith (KS) algorithm is the superior method among alternatives—particularly for handling large shocks and non-linear dynamics—offering unmatched accuracy and flexibility for adapting to diverse macroeconomic heterogeneous agent models. This approach can capture complex phenomena like occasionally binding constraints, non-stationary dynamics, and multiple equilibria that simpler linearization methods miss. Ironically, this same generality makes it the most computationally resource-demanding method required to achieve such precision.[1]

This paper proposes a neural network–based Krusell–Smith (NN-KS) algorithm for solving heterogeneous agent models. In particular, I develop a global, non-linear solution method leveraging the merits of neural networks and deep learning techniques to overcome the speed bottleneck of the KS algorithm. It is an order of magnitude faster than the standard grid-based KS method, while preserving all the benefits achieved by the original approach—fully replicating the accuracy of the solution and its micro and macro implications. I demonstrate this by applying the NN-KS method to solve seminal heterogeneous firm models, such as those of Khan and Thomas (2008) and Bloom et al. (2018), and systematically compare my results with the comprehensive benchmarks provided by Terry (2017) across dimensions like simulation paths, business cycle moments, and forecast system accuracy.

Many economic models do not yield explicit formulae for agents' decisions, prices, and allocations. This difficulty is particularly acute for heterogeneous firm

[1] Terry (2017) systematically compares five solution methods for heterogeneous firm models—the traditional Krusell-Smith algorithm (Krusell and Smith (1998), Krusell and Smith (1997)), Parameterized Distributions (Algan et al., 2008), Explicit Aggregation (Den Haan and Rendahl, 2010), Projection plus Perturbation (Reiter, 2009), and Parameterization plus Perturbation (Winberry, 2018).

models because closed-form solutions for market-clearing prices cannot be derived. This is problematic for the Krusell-Smith algorithm because it relies on a long simulation of the stochastic economy wherein, in each period, we must find the market-clearing equilibrium prices (such as wages) numerically. This contrasts with standard incomplete market, household-centric heterogeneous agent models, where price determination is often simpler even under aggregate uncertainty.

In more detail, the standard approach requires solving the firm's decision problem given a guessed price in each simulation period. We then use the solved policy rules to obtain the distribution of firms and consequent aggregate quantities to check if market clearing is satisfied. Since the initial guess is rarely correct, we must iteratively repeat this process to find the market-clearing prices before moving to the next period. We then carry the resulting distribution of firms forward and restart the numerical search for the next period's prices, repeating this over the entire stochastic simulation of 5,000 to 10,000 periods. This process is incredibly demanding; for instance, solving the model in Bloom et al. (2018) can take six days to complete.

The novelty of the NN-KS method is that it resolves this exact bottleneck by rendering the repetitive solution of the firm's decision problem unnecessary during the price search. Specifically, I include prices directly as an argument in the firm's policy functions. By approximating these functions with a neural network, we can capture the complex, non-linear mapping from the idiosyncratic and aggregate state (including the price) to the firm's optimal decision. Once trained, evaluating the policy for any candidate price becomes a simple forward pass through the network. This allows the simulation loop to determine the market-clearing price quickly via the policy function, bypassing the time-consuming optimization problem required to derive decision rules at every guess. The Bloom et al. (2018) model, which requires six days under the standard Krusell-Smith method, can be solved in just 102 minutes—about 100 times faster—while maintaining the same level of accuracy in both micro and macro moments.

Beyond pure speed improvements, this approach overcomes the fundamental

limitations of traditional methods. The flexibility of neural networks allows us to approximate discontinuous policy functions, such as the (S, s)-type policies generated by fixed adjustment costs. [2] In particular, I decompose the firm's decision into a discrete choice (whether to adjust) and a conditional continuous choice (the adjustment magnitude) in the Bloom et al. (2018) setting.[3] Furthermore, the underlying neural network architecture is ideally suited for modern GPU hardware. This is particularly crucial for the simulation step as emphasized by Hatcher and Scheffel (2016), where the simultaneous evaluation of millions of agents through batch processing dramatically accelerates computation, ensuring the method remains tractable even as model complexity increases.

Despite these advances, the NN-KS method retains the familiar skeleton of the Krusell-Smith algorithm, making it straightforward for researchers to understand and implement. The only fundamental change is replacing traditional value and policy function approximations with neural networks—researchers familiar with the KS framework can readily adopt this approach without learning an entirely new solution paradigm.

Related Literature. The application of deep learning to economics has opened exciting new avenues for solving complex models and uncovering insights previously unattainable with traditional methods (Fernández-Villaverde et al., 2020; Han et al., 2021; Azinovic et al., 2022; Kahou et al., 2021; Maliar et al., 2021; Maliar and Maliar, 2022; Kase et al., 2022; Fernández-Villaverde et al., 2023; Gu et al., 2023; Valaitis and Villa, 2024; Payne et al., 2025; Azinovic-Yang and Žemlička,

[2] A key challenge in models with fixed adjustment costs is that optimal decision rules are inherently non-smooth and often take an (S, s)-type form. Standard interpolation schemes are unreliable in this setting, as interpolation schemes inherently smooth across the adjustment threshold and therefore fail to capture the discontinuous jump in the policy function. Moreover, policy functions may also be discontinuous with respect to the equilibrium price, which makes it difficult to include prices as state variables and approximate them using traditional grid-based approaches, such as Gomes and Michaelides (2008) and Favilukis et al. (2017).

[3] Fixed adjustment costs induce a discontinuous policy function, where the policy remains zero until a threshold is crossed and then suddenly jumps to a positive amount. Since standard neural networks are designed to approximate continuous functions, this decomposition is essential for accurately capturing such discontinuities.

2025). My contribution leverages these powerful neural network capabilities while maintaining a distinctly practical focus: I demonstrate that deep learning can dramatically accelerate the well-established Krusell-Smith method without altering its fundamental structure. This approach delivers order-of-magnitude speed improvements while requiring only the substitution of neural networks for traditional function approximations, making it immediately accessible to researchers already using KS-type algorithms.

The Krusell-Smith algorithm's accuracy comes at a severe computational cost, as Terry (2017) demonstrates.[4] Recent papers propose alternative ways to mitigate this bottleneck by advancing the simulation without imposing market clearing in every period. In particular, Bakota (2023) updates forecasting-rule coefficients using the Jacobian of cumulative excess demand in state-space representation, and Lee (2022) proposes a sequence-space method that derives implied prices from the simulation to update price guesses.[5]

My approach targets the same root-finding bottleneck but from a different angle: I still solve for the equilibrium price that clears the market (i.e., sets excess demand to zero) in each period, but I accelerate that search by conditioning firm policies directly on the candidate price and approximating these price-conditioned policies with neural networks. This transforms the expensive inner-loop re-optimization into a fast forward pass while preserving the essential KS structure. [6] Yang et al. (2025) adopts a similar strategy of treating prices as state

[4]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 these models is the high computational cost of simulation, as finding market-clearing prices requires repeatedly solving the firm's optimization problem in every period.

[5]These approaches share the view that the traditional per-period root-finding loop is the key computational bottleneck; they differ from the standard KS simulation in that excess demand need not be driven exactly to zero at each date before moving forward along the simulated path. Other approaches, such as Algan et al. (2008) and Sunakawa (2020), circumvent simulation entirely by aggregating individual policy functions to derive the law of motion for moments of the distribution.

[6]While earlier studies like Gomes and Michaelides (2008) and Favilukis et al. (2017) attempted to include prices directly using traditional interpolation methods, they faced exponentially increasing computational costs and unreliable interpolation in high dimensions, making the ap-

variables with neural networks, though they compute a Sequential Restricted Perceptions Equilibrium without the distribution included as a state variable, whereas I maintain the full rational expectations equilibrium.[7]

The NN-KS approach offers two crucial advantages that traditional methods cannot achieve. First, neural networks effectively tame the "curse of dimensionality" that hampers grid-based approximations when including prices as state variables. Second, the flexibility of neural network architectures allows us to handle the discontinuities inherent in lumpy investment models—a pervasive feature in heterogeneous firm models with fixed adjustment costs like Bloom et al. (2018). Building on Maliar and Maliar (2022)'s classification techniques, I decompose the firm's decision into a discrete choice (whether to adjust) and a conditional continuous choice (the adjustment magnitude), allowing the network to capture the discontinuities and (S, s)-type policies that cause traditional interpolation methods and even advanced techniques like the Endogenous Grid Method (Carroll, 2006) and its extension (Fella, 2014; Iskhakov et al., 2017) to fail or lose efficiency.[8]

Organization of the paper. Section 2 applies the proposed methodology to the heterogeneous firm model of Khan and Thomas (2008), where stochastic adjustment costs make policy rules smooth. This application highlights how the NN-KS 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.

proach impractical.

[7]As I explain later, I include moments of the distribution as state variables, as in Krusell and Smith (1998).

[8]There is no reason why Fella (2014) cannot in principle be applied to models with fixed adjustment costs such as Bloom et al. (2018), but it still requires searching for the optimal action in non-concave regions during simulation, which undermines its efficiency.

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

In this section, I apply the NN-KS method to the model of Khan and Thomas (2008). Subsection 2.1 reviews the model, while Subsection 2.2 introduces the Krusell–Smith method and discusses its computational limitations. Subsection 2.3 then describes the NN-KS method in detail. Finally, Subsection 2.4 presents the computational results and compares them with a benchmark obtained from the Khan and Thomas (2008) model implementation 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 $[0, \bar{\xi}]$. 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[9]:

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

$$R\left(\varepsilon', k'; z', \mu'\right) \equiv -k' + \mathbb{E}\Big[d_j\left(z; \mu\right) v^0\left(\varepsilon', k'; z', \mu'\right) \Big], \tag{7}$$

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

Forecasting rules,

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

In the equation (7), d_j is the stochastic discount factor.

On the household side, a representative agent makes decisions on consumption c, labor supply n_h, and portfolio λ' given the price ρ_0 for the current portfolio

[9]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.

shares and wage ω. The household maximizes its lifetime utility:

$$W(\lambda; z_i, \mu) = \max_{c, n_h, \lambda'} u(c, 1 - n_h) + \beta \sum_{j=1}^{N_z} \pi_{ij} W(\lambda'; z_j, \mu'), \tag{8}$$

subject to:

$$\begin{aligned} c + \int_S \rho_1 \left(\varepsilon', k'; z, \mu\right) \lambda' \left(d \left[\varepsilon' \times k'\right]\right) \\ \leq \omega(z, \mu) n^h + \int_S \rho_0(\varepsilon, k; z, \mu) \lambda(d[\varepsilon \times k]) \end{aligned} \tag{9}$$

utility function:

$$u(c, 1 - n_h) = \log c + \phi(1 - n_h) \tag{10}$$

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:

$$\omega(z, \mu) = \frac{D_2 U(C, 1 - N)}{D_1 U(C, 1 - N)} \tag{11}$$

$$d_j(z, \mu) = \frac{\beta D_1 U\left(C_j', 1 - N_j'\right)}{D_1 U(C, 1 - N)} \tag{12}$$

From the FOC with respect to C, the Lagrange multiplier $p(z, \mu)$ equals the marginal utility of consumption, so we can rewrite the equations above as:

$$p(z, \mu) = D_1 U(C, 1 - N) \tag{13}$$

$$\omega(z, \mu) = \frac{D_2 U(C, 1 - N)}{p(z, \mu)} = \frac{\phi}{p(z, \mu)} \tag{14}$$

$$d_j(z, \mu) = \frac{\beta p(z', \mu')}{p(z, \mu)} \tag{15}$$

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:

$$V^1(\varepsilon, k, \xi; z, \mu) = \max_{n \in \mathbf{R}_+} (z\varepsilon k^\alpha n^\nu - \omega n + (1-\delta)k)p$$
$$+ \max \left\{ -\xi \omega p + \max_{k^* \in \mathbf{R}_+} R\left(\varepsilon', k^*; z', \mu'\right), R\left(\varepsilon', (1-\delta)k; z', \mu'\right) \right\} \tag{16}$$

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

Forecasting rules,

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

Note that the second maximization in (16) 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.[10] The threshold level of the adjustment cost ξ that determines whether the firm invests is given by:

$$\xi^* = \frac{R(\varepsilon, k^*; z, \mu') - R(\varepsilon, (1-\delta)k; z, \mu')}{p\omega} \tag{17}$$

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 cap-

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

ital transitions are given by

$$k'(\varepsilon,k,\xi;z,\mu) = \begin{cases} k^*(\varepsilon;z,\mu'), & \xi < \xi^*(\varepsilon,k;z,\mu) \\ (1-\delta)k, & \xi \geq \xi^*(\varepsilon,k;z,\mu) \end{cases} \tag{18}$$

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

$$\begin{gathered} \mu'(\varepsilon',k') = \Gamma_\mu(z,\mu) = \iiint I_A(\varepsilon,k)d\mu(\varepsilon,k)dG(\xi)d\Phi(\eta'_\epsilon) \\ A(\varepsilon',k',\xi,\eta'_\epsilon;z,\mu) = \{(\varepsilon,k) \mid k'(\varepsilon,k,\xi;z,\mu) = k', \log(\varepsilon') = \rho_\epsilon \log(\varepsilon) + \eta'_\epsilon\}, \\ \Phi(x) = \mathbb{P}(\eta'_\epsilon \leq x) \end{gathered} \tag{19}$$

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

$$Y(z,\mu) = \iint z\varepsilon k^\alpha n(\varepsilon,k,\xi;z,\mu)^\nu d\mu(\varepsilon,k)dG(\xi) \tag{20}$$

$$I(z,\mu) = \iint (k'(\varepsilon,k,\xi;z,\mu) - (1-\delta)k)d\mu(\varepsilon,k)dG(\xi) \tag{21}$$

$$N(z,\mu) = \iint n(\varepsilon,k,\xi;z,\mu)d\mu(\varepsilon,k)dG(\xi) + \int G(\xi^*(\varepsilon,k;z,\mu))d\mu(\varepsilon,k) \tag{22}$$

- Aggregate consumption satisfies the resource constraint

$$C(z,\mu) = Y(z,\mu) - I(z,\mu) \tag{23}$$

- 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

$$p(z,\mu) = \Gamma_p(z,\mu) = \frac{1}{C(z,\mu)} \tag{24}$$

$$\omega(z,\mu) = \frac{\phi}{p(z,\mu)} \tag{25}$$

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

2.2 Krusell-Smith (KS) Method

To make the problem tractable, I 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:

$$\begin{aligned} V^0(\varepsilon,k;z,K) = \max_{n\in\mathbf{R}_+}\big(z\,\varepsilon\,k^\alpha\,n^\nu \;-\; \omega\,n \;+\; (1-\delta)\,k\big)\,p \;-\; \omega\,p\int_0^{\bar{\xi}}\xi\,G(\mathrm{d}\xi) \\ +\; G(\xi^*(\varepsilon,k;z,K))\max_{k^*\in\mathbf{R}_+} R\big(\varepsilon',k^*;z',K'\big) \\ +\; \big(1-G(\xi^*(\varepsilon,k;z,K))\big)\;R\big(\varepsilon',(1-\delta)\,k;z',K'\big) \end{aligned} \tag{26}$$

where the continuation value R is given by:

$$R\left(\varepsilon',k';z',K'\right) \equiv -k'p + \beta\mathbb{E}V^0\left(\varepsilon',k';z',K'\right) \tag{27}$$

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:

$$\log(K') = a_K + b_K\log(K) \tag{28}$$

$$\log(p) = a_p + b_p\log(K) \tag{29}$$

This formulation can be understood as occurring before the adjustment cost ξ is realized, so the discrete choice of whether to invest (the second max operator in equation (16)) 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 (26), which is derived by maximizing (27). Here, k is not included as a state variable for this policy function.

Algorithm 1: Krusell-Smith Method Procedure

1 Initialize: Set initial forecasting rule parameters and define the state grid;
2 **repeat**
3 **Step 1: Solve the Bellman Equation**;
4 Given forecasting rules, do value function iteration;
5 **Step 2: Simulate the Model**;
6 Run T periods simulations using the obtained value function;
7 **Step 3: Update Forecasting Rules**;
8 Update parameters by regressing simulation outcomes;
9 **until** *convergence condition is satisfied*;
10 **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 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.,∼2 seconds), the simulation can take significantly longer (e.g.,∼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:

$$\begin{aligned}\bar{V}^0(\varepsilon,k;z,K,p) = \max_{n\in\mathbf{R}_+}\big(z\,\varepsilon\,k^\alpha\,n^\nu \;-\; \omega\,n \;+\; (1-\delta)\,k\big)\,p \;-\; \omega\,p\int_0^{\bar{\xi}} \xi\,G(\mathrm{d}\xi)\\ +\; G(\xi^*(\varepsilon,k;z,K))\max_{k^*\in\mathbf{R}_+} R\big(\varepsilon',k^*;z',K',p\big)\\ +\; \big(1-G(\xi^*(\varepsilon,k;z,K))\big)\,R\big(\varepsilon',(1-\delta)\,k;z',K',p\big)\end{aligned} \tag{30}$$

$$R\left(\varepsilon',k';z',K',p\right) \equiv -k'p + \beta\mathbb{E}V^0\left(\varepsilon',k';z',K'\right) \tag{31}$$

The value function $\bar{V}^0$ 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 (29). Consequently, the value of a firm cannot be uniquely determined by the state $(\varepsilon, k; z, K)$ and the forecasting rules (28)-(29) 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 (30), 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 (30).

The first, which I term the **"Direct Maximization"** approach, involves deriving the optimal action for every guess of the price, p, during the simulation.[11] The second, the **"Interpolated-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 **"Direct Maximization"** approach is more accurate and general; however, it is also computationally slow. The reason for this slowness is explained next.

[11]We do not need a nested (concentric) iteration to compute the optimal decision in (30), because the continuation value on the right-hand side does not depend on the candidate price p. Therefore, we first solve (26) in Step 1 and then use the resulting value function as the continuation value in (30).

Figure 1: Comparison of simulation with Direct Maximization and Interpolated-Policy

Algorithm 2: Direct Maximization

```
1  for t = 1, . . . , 2500 do
2  |  while price not conv. do
3  |  |  Guess p;
4  |  |  for state (ε, k) do
5  |  |  |  for cand. k′ do
6  |  |  |  |  Compute RHS of
   |  |  |  |   Bellman;
7  |  |  |  end
8  |  |  |  Take arg max;
9  |  |  end
10 |  |  Aggregate → excess
   |  |   demand;
11 |  |  Update price bracket;
12 |  end
13 end
```

Algorithm 3: Interpolated-Policy

```
1  for t = 1, . . . , 2500 do
2  |  while price not conv. do
3  |  |  Guess p;
4  |  |  for state (ε, k) do
5  |  |  |  k′ = g(ε; z, K, p);
6  |  |  end
7  |  |  Aggregate → excess
   |  |   demand;
8  |  |  Update price bracket;
9  |  end
10 end
```

Why the "Direct Maximization" loop is slow. Figure 1 contrasts two inner loops. In the left panel ("Direct Maximization"), 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

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

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

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

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

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. Furthermore, as shown in Figure 4, when the policy function is strongly nonlinear in p, interpolation becomes unreliable.

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 is a deterministic adjustment cost, like Bloom et al. (2018), the policy function can be discontinuous. 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 "Direct Maximization" 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 The NN-KS Method

As seen in the previous section, the problem boils down to how to solve equation (30). The problem of the **"Direct Maximization"** approach 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 the NN-KS 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 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, the NN-KS method simply replaces the value and policy functions while preserving the skeleton of the Krusell-Smith algorithm, making it straightforward to understand and implement.

Overall Solution Algorithm. The 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,

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 $V_{nn}^0, g_{nn}, \Gamma_p, \Gamma_\mu$

```
1  Initialization:
2  Initialize V_nn^0, g_nn, Γ_p, Γ_μ, optimizers, schedulers
3  Set D ← D_0.
4  Main Loop (repeat until forecasting rules converge):
5  while ||Γ_p − Γ_p^old||_∞ > δ_1 or ||Γ_μ − Γ_μ^old||_∞ > δ_1 do
6      1. Value & Policy Iteration (repeat until RHS change ≤ ϵ):
7      Initialize rhs_prev and rhs.
8      while ||rhs − rhs_prev||_∞ > ϵ do
9          rhs_prev ← rhs.
10         rhs ← TrainPolicyNetwork(g_nn, V_nn^0, D)
11         loss_V ← TrainValueNetwork(V_nn^0, g_nn, D)
12     end
13     2. Simulation and Forecast Update:
14     D_new ← SimulateModel(g_nn, Γ_p, Γ_μ)
15     Γ_p^old ← Γ_p
16     Γ_μ^old ← Γ_μ
17     Γ_p ← UpdatePriceForecast(Γ_p, D_new)
18     Γ_μ ← UpdateCapitalForecast(Γ_μ, D_new)
19     D ← UpdateTrainingData(D, D_new)
20 end
21 return V_nn^0, g_nn, Γ_p, Γ_μ
```

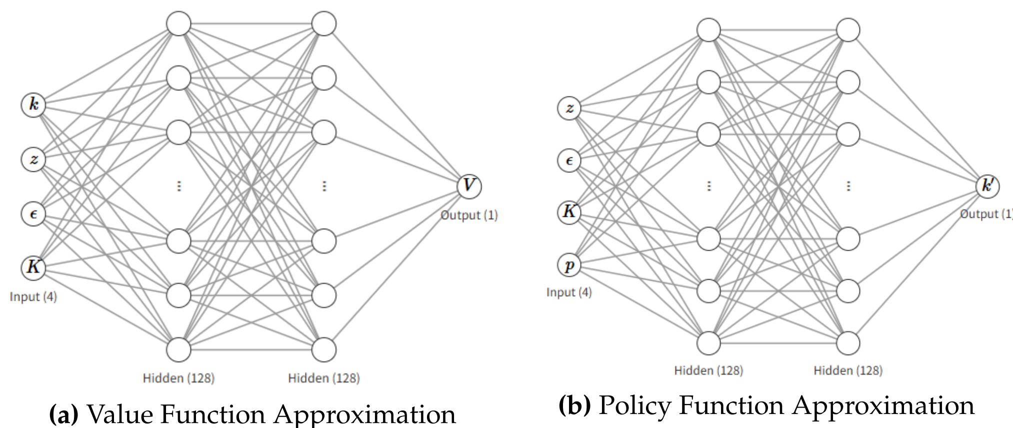
both using the current training data $\mathcal{D}$. When training the policy network with **TrainPolicyNetwork**, the parameters of the value network V_{nn}^0 are held fixed, and conversely, when training the value network with **TrainValueNetwork**, the parameters of the policy network g_{nn} are kept constant. In essence, this alternating update process is analogous to traditional Policy Iteration.

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**[12].

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.

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 (30) are approximated with neural networks as follows:

$$V^0(\varepsilon, k; z, \mu) \approx V^0_{nn}\left(\varepsilon, k; z, K\right), \tag{32}$$

$$k^* \approx g_{nn}\left(\varepsilon; z, K, p\right). \tag{33}$$

[12]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.

Figures 2a and 2b show snapshots of the value and policy functions, respectively. The neural networks have 2 hidden layers with 128 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. The NN-KS 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. [13]

Training of the policy function The policy function $g_{nn}(\varepsilon; z, K, p)$ is trained to output the capital stock k' that maximizes the equation (31). This is achieved by minimizing the **Policy Function Loss**:

$$\mathcal{L}_{policy} = -\frac{1}{N}\sum_{i=1}^{N}\left[-k'_{g_{nn}(\varepsilon_i; z_i, K_i, p_{error,i})} p_{error,i} + \beta \mathbb{E}_i V^0 \left(\varepsilon'_i, k'_{g_{nn}(\varepsilon_i; z_i, K_i, p_{error,i})}; z'_i, K'_i \right) \right] \tag{34}$$

To calculate (34), 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 $\Gamma_p(z_i, K_i)$:

$$p_{error,i} = \Gamma_p(z_i, K_i) + \boldsymbol{\epsilon}_i. \tag{35}$$

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.[14] In Khan and Thomas (2008), I set $\boldsymbol{\delta}$ =

[13] An important distinction from the interpolated-policy functions 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.

[14] Note that the policy function is trained to be consistent with both (26)–(27) and (30)–(31). In (26)–(27), the policy should be optimized given the forecasting rules (28) and (29). Since p_{error} is constructed around the forecasting rule for the price, the loss function (34) implicitly covers the

0.05 to match the magnitude of price variation observed in their simulations. The parameters of g_{nn} are optimized using the ADAM optimizer to minimize $\mathcal{L}_{policy}$.

Training of the value function 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:

$$\mathcal{L}_v = \frac{1}{N}\sum_{i=1}^{N}\Big(V^0_{nn}(\varepsilon_i, k_i; z_i, K_i) - \mathrm{RHS}_i\Big)^2. \tag{36}$$

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 (26), 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}:

$$\begin{aligned} RHS_i = \max_{n_i \in \mathbf{R}_+} &\big(z_i\,\varepsilon_i\,k_i^{\alpha}\,n_i^{\nu} \;-\; \omega_i\,n_i \;+\; (1-\delta)\,k_i\big)\,p_i \;-\; \omega_i\,p_i\int_0^{\bar{\xi}} \xi\,G(\mathrm{d}\xi) \\ &+\; G(\xi^*(\varepsilon_i, k_i; z_i, K_i))R\big(\varepsilon_i', g_{nn}(\varepsilon_i; z_i, K_i, p_i); z_i', K_i'\big) \\ &+\; \big(1 - G(\xi^*(\varepsilon_i, k_i; z_i, K_i))\big)\;R\big(\varepsilon_i', (1-\delta)\,k_i; z_i', K_i'\big) \end{aligned} \tag{37}$$

Here, the p is not 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 $V^0_{nn}(\mathrm{state}_i)$ more closely aligns with the target RHS_i, effectively pushing the network to learn a representation that satisfies the Bellman optimality condition. [15]

optimization problem in (26)–(27).

[15] 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, I adopt the target network technique from reinforcement learning.

2.4 Results

In this section, I compare the performance of the proposed NN-KS method with the benchmark Krusell-Smith (KS) method. For the KS method, I use the Fortran code provided by Terry (2017). The NN-KS 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 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) business cycle statistics, and (v) the accuracy of the forecast rules.

Table 1: Computation time comparison

Method	VFI Time (sec)	Simulation Time (sec)	Total Time (min)
KS Method (Terry (2017))	3	193	24
NN-KS Method	221	63	18

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.

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. Reported total times are the cumulative results of six iterations of the outer loop. 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, the NN-KS method initially incurs a higher cost in the VFI step, but the simulation is much faster (63 seconds versus 193 seconds). Importantly, this comparison is based on the same CPU hardware, so the speedup reflects the algorithmic improvement rather than a hardware advantage.

Moreover, the benchmark simulation time of 193 seconds corresponds to the original Fortran code running with OpenMP parallelization enabled. Without OpenMP, the same simulation takes 801 seconds. This highlights that, even though the benchmark is implemented in optimized Fortran (which has an inherent speed advantage over Python/PyTorch), the NN-KS simulation is still about three times faster on the same machine.

It is worth noting that, from the second iteration onward, the VFI step in the NN-KS method benefits from using the previous iteration's results as a warm start, significantly reducing the computation time to approximately 30 seconds per iteration. Because the VFI step quickly accelerates after the first iteration due to warm starts, the overall computation time is substantially reduced, with the NN-KS method completing six iterations in 18 minutes compared to 24 minutes for the KS method. In this model, using a CPU or GPU does not affect the simulation time with my computer because the number of points in the histogram is just 250.

Table 2: Bellman equation error

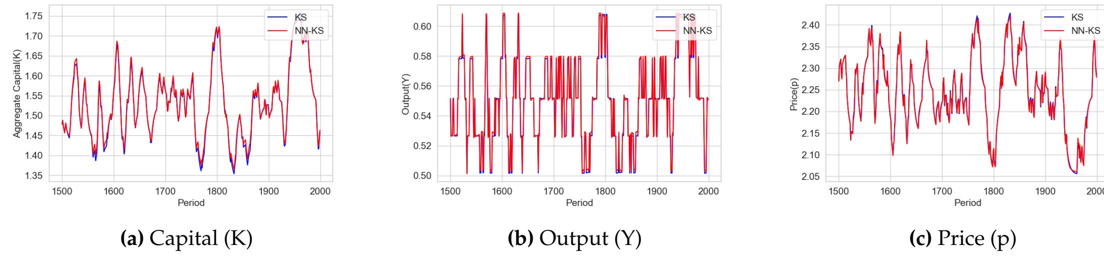
Method	**Average** ($\|V' - V\|$)	**Maximum** ($\|V' - V\|$)
KS Method (Terry (2017))	0.0473	0.0474
NN-KS Method	0.0065	0.0498

Note: Errors are computed at 2500 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.

Bellman Equation Error. Table 2 reports the results of a Bellman error comparison, where the error is measured as the difference between the right-hand side and the left-hand side of the Bellman equation in levels, i.e., $|V' - V|$, evaluated at a set of state points. Both methods compute this error over the same 2500 grid points, constructed using 5 points for the aggregate shock z, 5 for the idiosyncratic shock ε, 10 for individual capital k, and 10 for aggregate capital K. The table

reports both the **average** and the **maximum** Bellman errors across these points. As shown in Table 2, the NN-KS method attains a lower average Bellman error (0.0065) than the KS method (0.0473), while the maximum errors are comparable (0.0498 for NN-KS versus 0.0474 for KS).

Figure 3: Comparison of unconditional simulation paths



(a) Capital (K) **(b)** Output (Y) **(c)** Price (p)

Note: The figure compares unconditional simulation paths of the KS and the NN-KS method, using the same realization of aggregate shocks.

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 capital, output, and price under the KS method and the NN-KS method for the same sequence of shocks. The aggregate dynamics generated by the NN-KS method closely track those of the KS benchmark with high accuracy.

Business cycle statistics. Table 3 reports business cycle statistics computed from long-run unconditional simulations under the KS and NN-KS methods. All variables are HP-filtered with smoothing parameter $\lambda = 100$, and the first 500 periods are discarded as burn-in. The table presents three sets of moments: standard deviations, standard deviations relative to output, and correlations with output. Overall, the NN-KS method closely replicates the business cycle properties generated by the KS benchmark. While the unconditional standard deviations of investment and labor are slightly higher under NN-KS, the correlations with output are almost identical across the two methods.

Table 3: Business cycle statistics: KS vs NN-KS (HP filter, $\lambda = 100$)

	Output	Investment	Labor	Consumption
Standard deviations (%)				
KS	2.6299	10.3433	1.7981	1.0163
NN-KS	2.6840	10.7591	1.8882	1.0109
(run s.d.)	0.0095	0.0790	0.0153	0.0034
Ratio to output				
KS	1.0000	3.9329	0.6837	0.3864
NN-KS	1.0000	4.0085	0.7034	0.3766
(run s.d.)	–	0.0154	0.0032	0.0023
Correlations with output				
KS	1.0000	0.9765	0.9645	0.8850
NN-KS	1.0000	0.9746	0.9616	0.8587
(run s.d.)	–	0.0003	0.0006	0.0037

Note: All variables are HP-filtered with smoothing parameter $\lambda = 100$, and statistics are computed after discarding the first 500 simulation periods (burn-in). For NN-KS, entries report the average across 5 independent runs; the standard deviations across runs are reported in the (run s.d.) rows.

Forecast-System Accuracy. Table 4 reports measures for the forecast of the aggregate price p and next-period aggregate capital K', comparing the NN-KS with the KS benchmark. The KS method attains better accuracy in almost all metrics. Although the Max value of the Den Haan Statistics shows a relatively larger value in the NN-KS method, my approach delivers nearly identical performance across other measures. This indicates that the policy networks are sufficiently well-trained 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 the NN-KS method becomes even more pronounced in larger models. I apply the proposed 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

Table 4: Accuracy of the forecasting rules

	p (%)		K' (%)	
	KS	NN-KS	KS	NN-KS
Den Haan Statistics				
Max	0.11	0.39	0.38	0.90
Mean	0.05	0.06	0.23	0.15
Root Mean Squared Error (RMSE)				
RMSE	0.05	0.06	0.05	0.06
Forecast Regression R^2				
R^2	0.9998	0.9997	0.9995	0.9999

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).

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.

$$y = z\varepsilon k^{\alpha} n^{\nu}, \tag{38}$$

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

- **Aggregate Productivity:**

$$\log(z') = \rho_z \log(z) + \sigma_z u'_z, \quad u'_z \sim N(0,1) \tag{39}$$

- **Idiosyncratic Productivity:**

$$\log(\varepsilon') = \rho_\varepsilon \log(\varepsilon) + \sigma_\varepsilon u'_\varepsilon, \quad u'_\varepsilon \sim N(0,1) \tag{40}$$

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:

$$\mathrm{AC}^k = \mathbf{1}_{|i|>0}\, y\, F^K + S|i|\, \mathbf{1}_{i<0} \tag{41}$$

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:

$$n = (1 - \delta_n)\, n_{-1} + s, \tag{42}$$

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:

$$\mathrm{AC}^n = \mathbf{1}_{|s|>0}\, y\, F^L + |s|\, H\, w \tag{43}$$

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 redefine 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:

$$\tilde{V}(k, n_{-1}, \varepsilon; z, \sigma_z, \sigma_\varepsilon, \mu) = \max_{i,n} \Bigg\{ p(z, \sigma_z, \sigma_\varepsilon, \mu) \Big[y - w(z, \sigma_z, \sigma_\varepsilon, \mu) n - i - \mathrm{AC}^k - \mathrm{AC}^n \Big] + \beta \mathbb{E} \Big[\tilde{V}(k', n, \varepsilon'; z', \sigma_z', \sigma_\varepsilon', \mu') \Big] \Bigg\}. \tag{44}$$

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:

$$\begin{aligned} \mu' &= \Gamma_\mu(z, \sigma_z, \sigma_\varepsilon, \mu), \\ p &= \Gamma_p(z, \sigma_z, \sigma_\varepsilon, \mu). \end{aligned} \tag{45}$$

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 (45) 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.[16] 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),[17] 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.

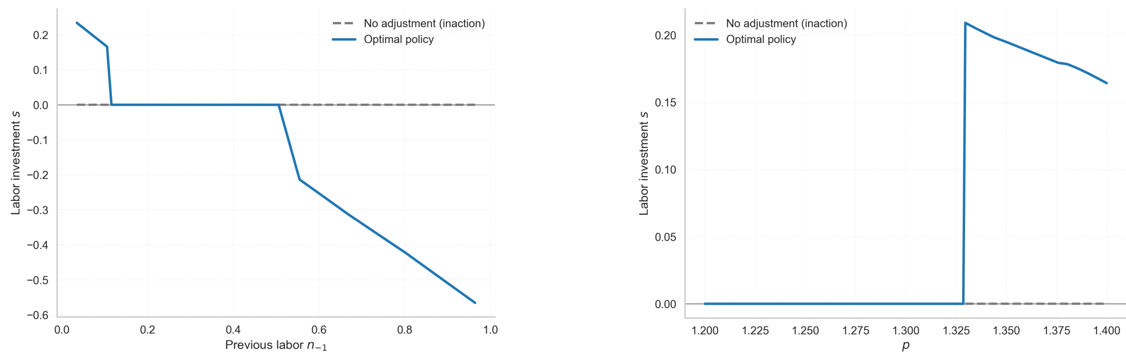
Non-smooth Adjustment Costs. The model includes several features that make the firm's decision-making process non-smooth. Specifically, there are fixed costs for adjusting capital or labor, a penalty for selling capital (partial irreversibility),

[16] In the original implementation, the grid over capital and labor is constructed so that adjacent grid points correspond exactly to depreciated values. In particular, the grid is defined so that $k_1 = (1 - \delta_k)k_2$ for neighboring points. Therefore, if the firm is currently at k_2, choosing k_1 on the grid corresponds exactly to a "no-adjustment" decision, since capital simply depreciates without paying the fixed cost.

[17] In line with the original implementation, I impose $\sigma_z = \sigma_\varepsilon$ and include the lagged uncertainty state σ_{-1} in the state space.

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: 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

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, standard solution methods that rely on smooth interpolation are not applicable for approximating the firm's policy functions. Indeed, the original code of Bloom et al. (2018) employs a grid-based solution method without interpolation for this reason.

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 firm's optimal level of investment. 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 (whether to adjust), (ii) a conditional continuous choice (the adjustment magnitude).

Necessity of two policy functions. As Figure 4 illustrates, the optimal policy exhibits an (S, s)-type inaction region and a jump discontinuity at the adjustment boundary. This discontinuity already makes a single-network representation problematic: standard feedforward neural networks with common (continuous) activation functions implement continuous mappings, and classical universal-approximation results guaranteeing uniform approximation are stated for continuous targets.[18] Therefore, 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. [19]

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

[18] See, e.g., Cybenko (1989) and Hornik (1991).

[19] If the optimal decision is not to invest, the policy function must return exactly $(1 - \delta_k)k$. In a continuous neural-network approximation, however, producing this exact value at the inaction point is practically impossible. Even an arbitrarily small approximation error implies a strictly positive or negative deviation from $(1 - \delta_k)k$, which can numerically induce a positive investment level. As a result, the algorithm may mistakenly select "adjustment" and incur the fixed cost, even though the true optimum is to choose no adjustment and keep capital at $(1 - \delta_k)k$.

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 (44)—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:

$$\tilde{V}(k, n_{-1}, \varepsilon; z, \sigma_z, \sigma_\varepsilon, \mu) = \max\{\, R^{00},\ R^{10},\ R^{01},\ R^{11}\,\}. \tag{46}$$

Each R^{ab} is the right-hand-side of the bellman equation 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:

$$R^{00} \;=\; p\Big[y(k, n_0, \varepsilon; z) - w n_0\Big] + \beta\, \mathbb{E}\Big[\tilde{V}\big(k_0,\ n_0,\ \cdot\big)\Big], \quad \text{(no investment, no labor adj.)} \tag{47}$$

$$\begin{aligned} R^{10} \;=\; & \max_{k'}\ p\Big[y(k, n_0, \varepsilon; z) - w n_0 - i - \mathrm{AC}^k(i)\Big] \\ & \qquad + \beta\, \mathbb{E}\Big[\tilde{V}\big(k',\ n_0,\ \cdot\big)\Big], \quad \text{(capital adj., no labor adj.)} \end{aligned} \tag{48}$$

$$\begin{aligned} R^{01} \;=\; & \max_{n}\ p\Big[y(k, n, \varepsilon; z) - w n - \mathrm{AC}^n(s)\Big] \\ & \qquad + \beta\, \mathbb{E}\Big[\tilde{V}\big(k_0,\ n,\ \cdot\big)\Big], \quad \text{(no capital adj., labor adj.)} \end{aligned} \tag{49}$$

$$\begin{aligned} R^{11} \;=\; & \max_{k', n}\ p\Big[y(k, n, \varepsilon; z) - w n - i - \mathrm{AC}^k(i) - \mathrm{AC}^n(s)\Big] \\ & \qquad + \beta\, \mathbb{E}\Big[\tilde{V}\big(k',\ n,\ \cdot\big)\Big], \quad \text{(capital adj., labor adj.)} \end{aligned} \tag{50}$$

Thus, the *outer* maximization $\max\{\cdot\}$ (46) corresponds to the **discrete policy** $g_d(x, p)$ (whether to adjust), that is, the choice of which adjustment regime to follow. Within each branch, the *inner* maximization delivers the branch-specific **conditional continuous policy** $g_c(x, p|ab)$ (the adjustment magnitude) 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 continuous 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).

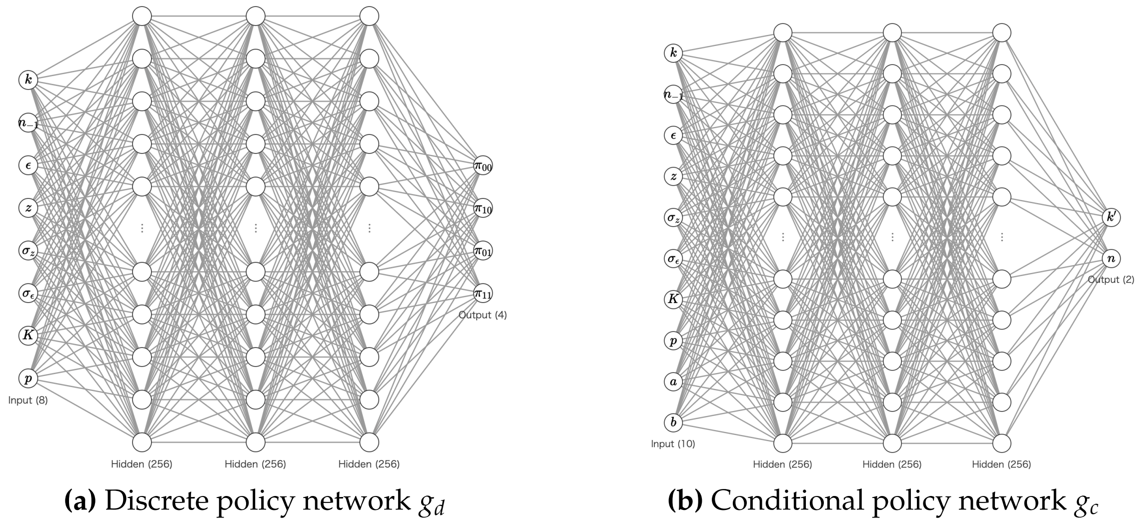
Formally:

$$
\begin{aligned}
&\textbf{Branch } 00 \textbf{ (inaction):} && (k', n) = g_c(x, p \mid 00) = (k_0,\, n_0). \\
&\textbf{Branch } 10 \textbf{ (capital-only):} && (k', n) = g_c(x, p \mid 10) = \big(\arg\max_{k'}\{\text{RHS in (48)}\},\, n_0\big). \\
&\textbf{Branch } 01 \textbf{ (labor-only):} && (k', n) = g_c(x, p \mid 01) = \big(k_0,\, \arg\max_{n}\{\text{RHS in (49)}\}\big). \\
&\textbf{Branch } 11 \textbf{ (both):} && (k', n) = g_c(x, p \mid 11) = \arg\max_{k', n}\{\text{RHS in (50)}\}.
\end{aligned}
$$

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 choice in the Krusell and Smith (1998) model with indivisible labor. In this paper the objective is to train the network to learn 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, a firm will choose the branch that maximizes the RHS. Formally,

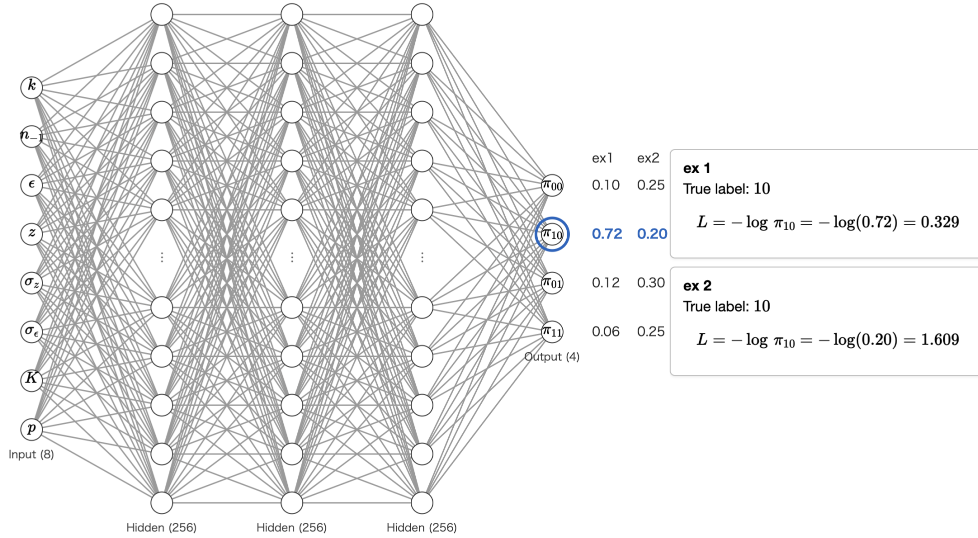
$$y^*(x, p_{\text{error}}) \in \underset{ab \in \{00,10,01,11\}}{\operatorname{argmax}} R^{ab}(x, p_{\text{error}}).$$

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

Figure 6: Training of the Discrete policy network



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 $(k, n_{-1}, \varepsilon, z, \sigma_z, \sigma_\varepsilon, K, p_{\text{error}})$ 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:

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

The input to g_c consists of the state (x, p_{error}) and the binary indicators (a, b) corresponding to the branch $ab = g_d(x, p_{\text{error}})$.[20] This setup allows the network to learn conditional responses across different regimes of inaction and adjustment.

Value network and schedule. We train the value function by minimizing the difference between the value function and the right-hand side of the Bellman equation, as in equation (36) of Khan and Thomas (2008) setting. Using this policy, we construct the target for training the value function:

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

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:

$$\textsc{Conditional Policy} \rightarrow \textsc{Discrete Policy} \rightarrow \textsc{Value Network}.$$

3.3 Result

In this section, I compare the performance of the NN-KS with the results obtained using the code provided by Bloom et al. (2018). In Appendix B.1, I evaluate the accuracy of the discrete policy function, and in Appendix B.2, I provide a graphical comparison of policy functions between the KS method and the NN-KS method across aggregate and idiosyncratic productivity states. 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 H100 GPU for validation. The results are mostly

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

based on single precision (FP32). [21] It is important to note that the benchmark results for Bloom et al. (2018) reported here are based on a slightly modified version of their original code, specifically concerning tracking the distribution in the simulation. Consequently, these results may differ from those reported in the published version of Bloom et al. (2018). [22]

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	29	434	9212 ($\approx$ 6 days)
NN-KS	23	2.0	102

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.

Computation and Simulation Speed. Table 5 presents the VFI time, simulation time and total time for each method. As the table shows, the NN-KS achieves a significant reduction in computation time compared to the KS method. In total, the NN-KS method completes in about 102 minutes, whereas the KS method requires roughly six days.[23] The total time highly depends on the quality of initial coefficients of forecasting rules, but it mostly requires around 20 iterations with their original initial values. Simulations were run for 5,000 periods for the both methods. It is also worth noting that, while my simulations are GPU-accelerated, running them on the CPU alone increases the simulation time to approximately 38

[21] The code is implemented primarily in single precision (FP32), with double precision used only for updating the distribution and aggregating variables in the simulation.

[22] 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.

[23] When run on an NVIDIA H100 GPU, runtime is approximately one hour. NVIDIA H100 GPUs are accessible through Google Colab.

minutes. Relative to KS, this corresponds to about a $434/38 \approx 11$ times speedup even without GPU acceleration (algorithmic gain). GPU acceleration then delivers an additional $38/2 \approx 19$ times speedup, reducing the simulation time further from the CPU-only baseline to the GPU baseline (hardware gain).

For NN-KS, the VFI step in the first outer-loop iteration is the most time-consuming because the value and policy networks must be learned essentially from scratch. From the second iteration onward, however, training becomes much faster because the algorithm warm-starts from the networks obtained in the previous iteration, and the VFI time typically falls to around 1–2 minutes per iteration.[24]

Sources of Speed Improvement and Scalability. The dramatic reduction in simulation time, particularly evident in Table 5, stems from two key features of the NN-KS 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 ma-

[24]This VFI time can vary substantially with how the neural network is initialized; with a better choice of initial values, it is possible to reduce the first-iteration time to only a few minutes. The results reported in Table 5 intentionally initialize only the value network using a simple linear function to ensure a fair comparison across methods, even though the true value function is highly non-linear; the policy network is initialized using the He initialization.

Table 6: Computation time for **1,000-period** simulations (256 × 3 layers).

	NN-KS			KS
Scale factor (Total points)	CPU (i7-12700F)	GPU (RTX 3080)	GPU (H100)	CPU (i7-12700F)
1× (16,835)	460 s	29 s	12 s	5,208 s
5× (84,175)	2,300 s	101 s	31 s	—
10× (168,350)	5,600 s	185 s	52 s	—
20× (336,700)	9,500 s	360 s	99 s	—
40× (673,400)	15,600 s	630 s	170 s	—
80× (1,346,800)	25,200 s	1020 s	290 s	—

Note: CPU times (NN-KS) are measured directly only for 1×, then others are scaled from 50-period runs. GPU times (NN-KS) are measured directly up to 10×, then scaled in the same way. For the KS method, only the CPU time at 1× is available (5,208 s). The RTX 3080 has 10 GB of GPU memory, while the H100 provides 80 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.

trix multiplications with the 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.

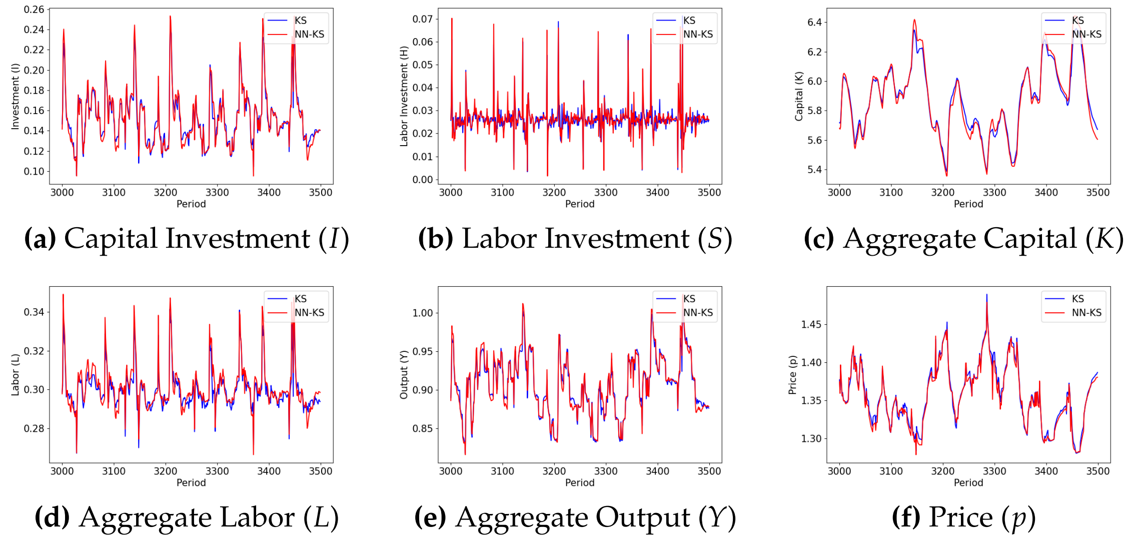
Scalability Results. Table 6 reports computation times for 1,000-period simulations under different scale factors of the distribution grid.[25] Both CPU and GPU performance are reported for neural network configurations (256 × 3 layers). The results clearly demonstrate the advantage of GPU acceleration: while CPU run-

[25]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.

times increase rapidly with scale, GPUs (both RTX 3080 and H100) remain orders of magnitude faster. For instance, in the 80× setting with more than 1.3 million grid points, the CPU runtime rises to nearly 25,200 seconds, whereas the GPU runtime remains around 1,000 seconds; on the H100, it drops further to just 290 seconds. [26]

Unconditional Simulation Comparison. Figure 7 reports unconditional simulation paths for aggregate variables over periods 3000–3500, computed using the policies from the final outer-loop iteration for both methods. While small differences remain, the NN-KS paths closely match those under the KS benchmark. This evidence suggests that the NN-KS method converges to a solution that is sufficiently close to the one obtained by the KS method.

Figure 7: Comparison of unconditional simulation paths



(a) Capital Investment (I) **(b)** Labor Investment (S) **(c)** Aggregate Capital (K)

(d) Aggregate Labor (L) **(e)** Aggregate Output (Y) **(f)** Price (p)

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

[26] At 100× and larger scales, the RTX 3080 reaches its memory limit and the runtime increases sharply, whereas the H100 continues to scale smoothly up to roughly 800×. This underscores both the gains from GPU acceleration and the importance of on-package memory capacity when running very large-scale simulations.

Table 7: HP-filtered business-cycle statistics

	Output	Investment	Labor	Consumption
Standard deviations (%)				
KS	1.9784	10.0256	2.4187	0.8733
NN-KS	2.0303	10.5243	2.5980	0.8710
(run s.d.)	0.0191	0.1052	0.0767	0.0034
Ratio to output				
KS	1.0000	5.0676	1.2226	0.4414
NN-KS	1.0000	5.1835	1.2795	0.4288
(run s.d.)	–	0.0225	0.0277	0.0029
Correlations with output				
KS	1.0000	0.9502	0.8200	0.2780
NN-KS	1.0000	0.9513	0.8250	0.2303
(run s.d.)	–	0.0010	0.0026	0.0248

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. NN-KS statistics are reported as averages over 5 independent runs. The third row in each block ("run s.d.") reports the across-run standard deviation computed from these 5 runs.

Business-Cycle Properties. Table 7 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 the NN-KS method. Overall, the NN-KS method replicates the business cycle properties well: while the volatilities of capital and labor investment are slightly larger under NN-KS, the correlations with output are closely matched.

Internal Accuracy of Forecast Systems. Table 8 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 regressions. Both the NN-KS method and the benchmark KS method achieve high accuracy. For p, the NN-KS method performs slightly worse than KS across all three metrics (higher Den Haan statistics and RMSE, and a slightly lower R^2). For K', however, the NN-KS method per-

Table 8: Accuracy of the forecasting rules

	p (%)		K' (%)	
	KS	NN-KS	KS	NN-KS
Den Haan Statistics				
Maximum	3.52	3.62	5.89	6.47
Mean	0.87	0.88	1.75	1.88
Root Mean Squared Error				
RMSE	0.45	0.47	0.56	0.25
Forecast Regression R^2				
R^2	0.978	0.975	0.980	0.997

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 the NN-KS method.

forms better: it has lower RMSE and a higher R^2, although its Den Haan statistics are slightly higher. Appendix A.2 provides detailed forecast accuracy statistics broken down by aggregate shock state for the Bloom et al. (2018) model application (see Table 10).

4 Conclusion

This paper proposes an NN-KS method that embeds neural networks in the Krusell–Smith framework by (i) approximating value and policy functions with the equilibrium price as a state variable and (ii) handling non-smooth policies with a discrete–continuous decision structure. This removes repeated optimization during price search while capturing (S, s)-type kinks, yielding large speed gains with GPU acceleration.

Applications to Khan and Thomas (2008) and Bloom et al. (2018) show that the

method preserves accuracy while sharply reducing computation time (e.g., six days to 102 minutes in Bloom et al. (2018)). The approach generalizes to other heterogeneous agent models with implicit market-clearing prices, including settings with multiple prices, while preserving the familiar KS structure.

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.

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

Azinovic-Yang, M. and J. Žemlička (2025). Deep learning in the sequence space.

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

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.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems 2*, 303–314.

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.

Den Haan, W. J. and P. Rendahl (2010). Solving the incomplete markets model with aggregate uncertainty using explicit aggregation. *Journal of Economic Dynamics and Control 34*(1), 69–78.

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. (2025). Deep learning for solving economic models. Technical report, National Bureau of Economic Research.

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

Fernández-Villaverde, J., G. Nuño, and J. Perla (2024). Taming the curse of dimensionality: quantitative economics with deep learning. Technical report, National Bureau of Economic Research.

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.

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*.

Hatcher, M. C. and E. M. Scheffel (2016). Solving the incomplete markets model in parallel using gpu computing and the krusell–smith algorithm. *Computational Economics 48*(4), 569–591.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks 4*(2), 251–257.

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.

Lee, H. (2022, Dec). Repeated transition method and the nonlinear business cycle with the corporate saving glut. MPRA Paper 115887, University Library of Munich, Germany.

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*.

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

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.

Winberry, T. (2018). A method for solving and estimating heterogeneous agent macro models. *Quantitative Economics 9*(3), 1123–1151.

Yang, Y., C. Wang, A. Schaab, and B. Moll (2025). Structural reinforcement learning for heterogeneous agent macroeconomics.