

AI for Structural Estimation*

Victor Duarte[†] Julia Fonseca[‡]

May 2026

Abstract

We develop a global method to solve and estimate dynamic equilibrium models that treats prices as pseudo parameters and market clearing as moment conditions, and reduces estimation time from days to minutes. Our approach leverages AI algorithms, software, and hardware, and has three building blocks. First, we extend the state space to include equilibrium prices and model parameters, which allows us to clear markets and estimate parameters by solving the model once. Second, we approximate the mapping between parameters and moments by training neural networks on model-simulated data, which act as closed-form expressions for moment conditions. Third, we use this mapping to estimate parameters by minimizing the distance between the model and data moments, and to find equilibrium prices by targeting a market-clearing imbalance of zero. We also use this mapping to assess identification globally, verifying if the estimation objective function has a unique minimum for each parameter. We illustrate our method by estimating a dynamic general equilibrium model of leverage and investment with three state variables, three controls, endogenous default, costly equity issuance, and non-convex adjustment costs. After four days, the traditional approach does not reach the loss we achieve in under 20 minutes. We build an AI agent that applies our method to new models from natural language prompts.

*We thank Diogo Duarte, Dan Green, Miles Kimball, Leonid Kogan, Jun Li, Adam McCloskey, Karel Mertens, Jonathan Parker, Alessandro Perri, Alex Richter, Dejanir Silva, David Thesmar, Adrien Verdelhan, Toni Whited, Mao Ye, and seminar participants at MIT, the NBER Summer Institute Big Data and High-Performance Computing meeting, the Dallas Fed, Duke Fuqua, the University of Colorado Boulder, and Wharton for helpful comments and discussions. This paper previously circulated as “Global Identification with Gradient-Based Structural Estimation.”

[†]Gies College of Business, University of Illinois Urbana-Champaign. Email: vduarte@illinois.edu

[‡]Gies College of Business, University of Illinois Urbana-Champaign and NBER. Email: juliaf@illinois.edu

1 Introduction

Structural estimation is a fundamental tool in many fields of economics and finance, from macroeconomics and industrial organization to asset pricing and corporate finance. A structural estimation procedure typically uses stochastic dynamic optimization and Monte Carlo simulation to obtain model-implied moments for a given set of parameters in an inner loop, and minimizes a function of the distance between model and data moments in an outer loop. In general equilibrium models, the optimization involves an additional root-finding step to find prices that clear markets. This nested problem is computationally intensive for at least two reasons. First, it requires solving a complex optimization problem for each set of model parameters being evaluated in the outer loop. Moreover, the optimization algorithm used to minimize the distance between the model-implied and data moments is oblivious to the mapping between the model parameters and moments, which precludes the use of efficient, gradient-based optimization.

This paper proposes a new approach to structural estimation that circumvents these bottlenecks. Our method has three building blocks. First, we train deep neural networks to approximate the value and policy functions, augmenting the state space by treating the model parameters and equilibrium prices as state variables. After solving the model a single time, we then have the solution as a function of all possible parameters and equilibrium prices, eliminating the need to solve a dynamic programming problem thousands of times. Second, separate neural networks learn the mapping between moments and parameters—which we refer to as the moment function—directly from simulated data. Third, we use the networks approximating the moment function to estimate parameters by minimizing a function of the distance between data and model-implied moments. Because these moment networks are differentiable, we can compute analytical gradients and Hessian matrices of this loss function and use highly efficient gradient-based optimization. We leverage this algorithm

further by using AI software (JAX) and hardware (GPU cloud computing).¹

We make four contributions. The first is to make structural estimation substantially faster and less computationally intensive than traditional methods. In our application, our method reduces computation time from days to minutes. Second, we make estimation in general equilibrium as tractable as in partial equilibrium with a novel approach that clears markets by matching moments. We add the price to the set of parameters being estimated and the market-clearing imbalance—the difference between the right- and left-hand-side of the market-clearing equation—to the set of moment conditions. We then solve for equilibrium prices during the estimation procedure by targeting a market-clearing imbalance of zero.

Our last two contributions derive from the moment networks. Because we learn the mapping between moments and parameters, we can globally minimize the distance between model-implied and data moments over all parameters but one. This gives us a measure of how the best achievable fit varies as one parameter changes and all others adjust optimally, which we refer to as “minimum loss functions.” Our third contribution is to develop an algorithm to progressively narrow the region of the parameter space across which we search, by using the minimum loss functions to focus on the region around the global minimum. This allows us to focus resources on improving the solution where it matters, which makes it feasible to solve and estimate a model as complex as the one in our application.

Our final contribution is a new global identification diagnostic, using minimum loss functions to verify if a unique global minimum exists for every parameter. By optimizing relative to all other parameters rather than holding them fixed at their estimated values, our approach accounts for the fact that perturbing the value of one parameter affects estimates of other parameters. Our approach isolates weak identification from misspecification by using parameter estimates from one estimation to simulate moments that we target in a

¹We also use JAX and GPU cloud computing when comparing our method to traditional approaches, thus isolating gains from the algorithm itself. For benchmarking of computational gains obtained in traditional dynamic programming from using AI software and hardware, see Duarte et al. (2020).

second estimation. We can then verify uniqueness in a version of the model that is correctly specified, and check whether the second estimation recovers the parameters used to produce the targeted moments.

We illustrate the method by applying it to a dynamic model of leverage and investment with endogenous default, cash holdings, costly equity issuance, and non-convex adjustment costs (Gao, Whited, and Zhang, 2021). We also estimate a general equilibrium extension with a representative consumer, in which the wage clears the labor market. The model has a non-differentiable, non-convex objective function with both discrete and continuous actions, which is substantially harder for gradient-based methods than the convex problems typically solved in the AI-solution-methods literature. Our method estimates either version of the model (partial or general equilibrium) within an hour, costing approximately two dollars in cloud computing. We benchmark our method against the traditional approach of searching across the parameter space with simulated annealing and solving the model with value function iteration (VFI) at each parameter vector, which we implement efficiently in JAX with GPU cloud computing. After 20 hours of optimization, the traditional approach arrives at an estimation loss that our method takes about 13 minutes to obtain. In the general equilibrium version, our method arrives at a lower estimation loss in under 20 minutes than the traditional approach achieves after four days.

To lower the cost of applying our method to new models, our replication package will include an open-source library and an AI agent that can implement models from a natural-language description of the economic environment. The researcher provides a description of the model, and the agent writes the implementation code. We have verified that the agent can apply our method to simple problems ranging from a deterministic cake-eating problem to a stochastic growth model with four estimated parameters. We have not yet tested more complex models, but if the agent’s capabilities do not already greatly exceed what our test cases require, we expect they soon will as LLMs continue to evolve and we add examples to

our library. We describe the DF Assistant in Appendix A.13.

Literature review. The method we develop in this paper is closely related to Simulated Method of Moments (SMM), which approximates model-implied moments using Monte Carlo simulations for each set of parameters being evaluated (McFadden, 1989; Pakes and Pollard, 1989; Lee and Ingram, 1991; Duffie and Singleton, 1993). Our approach differs from SMM in the method used to approximate the model-implied moments. Our method solves the model for all parameter values at once and learns a mapping between parameters and moments for an infinite set of parameters directly from simulated observations. This considerably reduces computing time, as it eliminates the need to solve the model multiple times to evaluate different sets of parameters. This approach is related to indirect inference, which matches the estimates of an auxiliary statistical model between simulated and real data (Gourieroux, Monfort, and Renault, 1993). Similarly, Bazdresch, Kahn, and Whited (2017) develops an indirect inference estimation procedure using the empirical policy function as the auxiliary model.

The idea of expanding the set of state variables to include model parameters in order to facilitate estimation was first used in economics by Norets (2012), which proposes a method using a single-layer neural network in the model solution and Markov chain Monte Carlo for estimation. We contribute to this work by embedding a new solution method using deep neural networks into an estimation procedure that also involves training a separate neural network to approximate the mapping between model parameters and moments. To the best of our knowledge, we are also the first to propose solving for market-clearing prices by treating prices as pseudo parameters during the structural estimation procedure and adding the market-clearing imbalance to the set of moment conditions, with a target of zero.

This paper is also related to a growing literature using machine learning tools to solve dynamic optimization problems, including Duarte, Duarte, and Silva (2024), Scheidegger and Billionis (2019), Maliar, Maliar, and Winant (2021), Azinovic, Gaegauf, and Schei-

degger (2022), Maliar and Maliar (2022), Duarte et al. (2023), Fernández-Villaverde et al. (2023), Yang et al. (2025). We relate to this literature as we also propose a method to solve dynamic optimization problems—which extends the continuous-time method proposed by Duarte, Duarte, and Silva (2024) to discrete-time, general equilibrium problems with non-convexities—and embed it in a fast and efficient estimation procedure. Although we focus on one particular solution method, in principle, our estimation procedure could be extended to incorporate any method capable of handling a large enough number of state variables so that model parameters can be included as states. For applications such as ours, the solution method must also accommodate a non-differentiable, non-convex objective function with both discrete and continuous actions, distinguishing our method from others that can handle high dimensionality.

We also relate to other papers that study the relationship between parameters and moments to conduct sensitivity analysis. Closest to this paper, Duarte, Duarte, and Silva (2024) develops a solution method that involves expanding the state space to include model parameters so that, once a solution is obtained, one can easily compute moments for a range of parameter values and assess sensitivity. We contribute to this work by extending their solution method to a substantially more challenging model in general equilibrium and discrete time, and embedding it in an efficient estimation procedure. We extend their method in four key ways. First, we develop a new approach to solve for equilibrium prices during the estimation procedure by targeting market clearing as a moment condition, which makes general equilibrium as tractable as partial equilibrium. Second, our method can handle non-differentiability and non-convexity in the objective function. Third, we adapt this continuous-time method to a discrete-time setting. Finally, we develop an iterative algorithm that focuses on the relevant range of the parameter space as training progresses, allowing us to more quickly obtain an accurate solution for this range.

Andrews, Gentzkow, and Shapiro (2017) proposes a local linear approximation of the

mapping between parameters and moments that is used to assess the sensitivity of estimates to misspecification, but not directly in the estimation procedure. Subsequent work by Catherine et al. (2023) uses model-implied moments obtained from multiple solutions of a dynamic optimization problem to approximate the moment function, which is then used to estimate parameters and conduct robustness checks. They use their estimate of the moment function to generalize the local approximation of Andrews, Gentzkow, and Shapiro (2017) by computing moments across the full range of possible values for one parameter, while fixing other parameters at their estimated values. Their method differs from traditional methods in that the model is solved multiple times to approximate the moment function instead of at each of the multiple iteration steps used to estimate parameters in traditional methods.

Our global identification diagnostic differs from these works in two important ways. First, we expand the set of state variables to include all model parameters, altogether eliminating the need to solve the model multiple times at any step of the estimation procedure. Second, our approach allows us to assess identification globally, without holding any parameters fixed, to determine whether the econometric objective function has a unique global minimum. Because our method is so efficient, we can afford to estimate the model twice, targeting simulated moments from the first estimation and checking whether we can recover the parameters in a second estimation.

Finally, this work is also related to a rich literature that structurally estimates corporate finance models (see Strebulaev and Whited (2012) for a survey). Closest to our work are Gao, Whited, and Zhang (2021)—who develop and solve the model in our application, and estimate the partial equilibrium version—and Ivanov, Pettit, and Whited (2025), which, to the best of our knowledge, is the only other paper in this literature that estimates a dynamic equilibrium model.

The rest of the paper is organized as follows. Section 2 describes the dynamic corporate finance model we use as an application. Section 3 presents our estimation method. Section

4 reports our main estimation results. Section 5 presents estimation results for the general equilibrium extension. Section 6 concludes. We provide a step-by-step description of the algorithm and practical guidance for implementation in Appendix A.

2 Application: A dynamic corporate finance model

We explain and illustrate our method with a dynamic model of leverage and investment with three state variables, three controls, endogenous default, costly equity issuance, and non-convex adjustment costs, similar to Gao, Whited, and Zhang (2021). We also consider a general equilibrium extension in which the wage clears the labor market.

2.1 Model description

Time is discrete. An infinitely-lived firm combines capital and labor to produce output using a stochastic technology subject to productivity shocks. Before production, it pays fixed operating costs and finances its activities with current profit flows, risky debt, external equity, and internal cash. There are two frictions associated with raising external financing: defaulting on debt incurs deadweight costs, and equity issuance incurs fees.

2.1.1 Technology and production

The firm operates a decreasing-returns technology with capital k and labor ℓ . Output is

$$y = z (k^\alpha \ell^{1-\alpha})^\theta, \quad (1)$$

where $\alpha \in (0, 1)$ is the capital share, $\theta \in (0, 1)$ governs the returns to scale, and z is a productivity shock. Log productivity follows an AR(1) process:

$$\log z' = \rho \log z + \sigma \varepsilon', \quad \varepsilon' \sim \mathcal{N}(0, 1). \quad (2)$$

The firm hires labor competitively at wage w . Since the production function has decreasing returns, the static labor choice has an analytical solution. Substituting the optimal labor demand yields a reduced-form operating surplus:

$$\pi(z, k) = zA_\pi k^\xi - c_f, \quad (3)$$

where $\xi \equiv \alpha\theta/\nu$, $\nu \equiv 1 - (1 - \alpha)\theta$, A_π collects wage- and technology-specific constants, and c_f is a fixed operating cost.²

2.1.2 Investment and adjustment costs

The firm chooses investment I each period. Capital accumulates according to

$$k' = (1 - \delta)k + I, \quad (4)$$

where δ is the depreciation rate. Investment is subject to both a convex adjustment cost $\frac{1}{2}\gamma_1 I^2/k$ and a fixed cost $\gamma_0 k \mathbf{1}\{I > 0\}$ that is proportional to capital and incurred whenever investment is positive. The convex component produces the standard q -theory smoothing of investment, while the fixed component generates lumpy investment and periods of inactivity.

2.1.3 Debt and default

The firm issues one-period debt at an endogenous price q that reflects default risk. The state variable for financing is net debt $b = b^{gross} - c$, where b^{gross} is gross debt per unit of capital and c is cash per unit of capital. Each period, the firm chooses next-period gross debt b' and cash c' , both expressed per unit of next-period capital k' .

The firm defaults when two conditions hold simultaneously: (i) the continuation equity value is negative, and (ii) liquidation proceeds do not cover the outstanding debt. Liquidation

²Note that z in Eq. (3) denotes the reduced-form productivity shifter, which absorbs the $1/\nu$ exponent from the labor substitution.

recovery is

$$R(k', c') = c'k' + \chi(1 - \delta)k', \quad (5)$$

where χ is the recovery rate. The bond price is the discounted expected repayment:

$$q(z, k', c', b') = \frac{1}{1 + r_f(1 - \tau)} \mathbb{E}_{z'|z} \left[(1 - \mathbf{1}_{def}) + \mathbf{1}_{def} \frac{R(k', c')}{b'k'} \right], \quad (6)$$

where r_f is the risk-free rate, τ is the corporate tax rate, and $\mathbf{1}_{def}$ indicates default.

2.1.4 Dividends and equity issuance

The firm's cash flow is determined in two subperiods. In the preproduction stage, the firm rolls over its debt and pays fixed operating costs. Sources minus uses of funds in this stage are

$$d_1 = q \cdot b' \cdot k' - b \cdot k - c_f. \quad (7)$$

If $d_1 > 0$, the firm carries this excess to the postproduction stage. If $d_1 < 0$, the firm issues equity at a cost of $\lambda_0 k + \lambda_1 |d_1|$ (a fixed plus proportional fee).

In the postproduction stage, the firm collects gross output net of investment and adjustment costs. Sources minus uses in this stage, including any preproduction excess carried forward, are

$$d_2 = zA_\pi k^\xi + \max\{d_1, 0\} - I - \frac{1}{2}\gamma_1 I^2/k - c'k'\iota_c - \gamma_0 k \mathbf{1}\{I > 0\}, \quad (8)$$

where $\iota_c = 1/(1 + r_c r_f(1 - \tau))$ adjusts for the opportunity cost of carrying cash. We set $r_c = 0$, so cash earns no interest and $\iota_c = 1$. If $d_2 > 0$, the firm distributes this amount to shareholders. If $d_2 < 0$, the firm issues equity, again incurring fixed and proportional costs.

The net payout to shareholders combines the cash flows from both subperiods, with

equity issuance costs subtracted wherever issuance occurs:

$$D = d_2 + d_1 \cdot \mathbf{1}\{d_1 < 0\} - (\lambda_0 k + \lambda_1 |d_1|) \cdot \mathbf{1}\{d_1 < 0\} - (\lambda_0 k + \lambda_1 |d_2|) \cdot \mathbf{1}\{d_2 < 0\}. \quad (9)$$

The first two terms give the firm's net cash position before issuance: d_2 already includes the rollover of any positive preproduction excess, and the $d_1 \cdot \mathbf{1}\{d_1 < 0\}$ term restores the negative preproduction cash flow that the rollover excludes. The last two terms are the fixed and proportional issuance costs incurred in each subperiod where the firm taps equity.

2.1.5 Bellman equation

The firm's equity value solves

$$V(z, k, b) = \max_{i, b', c'} \{D(z, k, b, i, b', c') + \beta \mathbb{E}_{z'|z} [\max\{V(z', k', b' - c'), 0\}]\}, \quad (10)$$

where $\beta = 1/(1 + r_f)$ and $k' = (1 + i - \delta)k$. The $\max\{V, 0\}$ term captures limited liability, allowing shareholders to walk away when equity value is negative. The bond-pricing equation (6) and the Bellman equation (10) must be solved jointly because q depends on V through the default rule, and V depends on q through the dividend.

2.2 General equilibrium extension

We also consider an extension of the model in which the wage is determined in equilibrium. We add a representative consumer with utility $\log(c_s) + \phi(1 - n_s)$, where c_s is consumption, n_s is labor supply, and ϕ governs the utility of leisure. Following Gao, Whited, and Zhang (2021), we set $\phi = 1.7$. The consumer's budget constraint is

$$c_s + q_d b'_d - b_d = w n_s + D, \quad (11)$$

where b_d is the consumer’s bond holdings and D denotes net distributions from firms (dividends minus equity issuance costs). The consumer saves by depositing funds with a competitive financial intermediary sector, which sets the safe deposit price q_d to earn zero profits. This zero-profit intermediary uses these funds to purchase the heterogeneous, risky corporate bonds at the firm-specific prices $q(z, k', c', b')$, diversifies idiosyncratic default risk, and guarantees full repayment of the safe deposits b_d to the consumer.

In equilibrium, the labor, bond, and output markets clear. The equilibrium wage w is the value that satisfies the aggregate resource constraint

$$Y - I - C - S = 0, \tag{12}$$

where Y is aggregate sales, I is aggregate investment, $C = w/\phi$ is aggregate consumption, and S is aggregate cash accumulation.

One of our main contributions is to show that we can clear markets during the estimation procedure by including the resource constraint imbalance in the set of targeted moments as an additional moment condition with a target of zero. We describe how the market-clearing condition is imposed within our estimation framework in Section 3.

3 Estimation method

This section gives an overview of our approach and key implementation decisions, with a comprehensive description and practical guidance reserved for Appendix A. Structural estimation typically requires solving a model many times. In the standard approach, an outer-loop optimizer proposes a candidate parameter vector, and then the algorithm solves the Bellman Eq. (10) for that parameter vector, simulates a panel of firms, computes moments, and evaluates how well those moments match the data. The optimizer then proposes a new candidate parameter vector, and the process repeats. With a model as the one described

in Section 2, each evaluation can take several minutes on a modern computer, and a global optimizer may require thousands of evaluations, so the total estimation time can stretch to weeks.

Our method eliminates this bottleneck by solving the model once for all parameter values simultaneously. The algorithm has three components. First, we train neural networks to approximate the value function $V(z, k, b)$ and the optimal controls (i, b', c') as functions of the parameters. Once trained, these networks instantly return the model solution for any parameter vector. Second, we use the trained networks to simulate data and compute moments for many parameter vectors, building a dataset that maps parameters to moments. Third, we train a separate set of neural networks to learn this mapping and use it to find the parameter values that best match the data. Because these networks are differentiable, we can use gradient-based optimization rather than derivative-free global search, which converges in seconds rather than minutes or hours.

Importantly, as we describe in Section 3.4, the same framework naturally accommodates general equilibrium. Equilibrium prices can be included as pseudo parameters to be estimated, with market-clearing conditions imposed as additional moment conditions. Our approach thus embeds the model solution step, including solving for equilibrium prices to clear markets, into the estimation procedure.

3.1 Block 1: Solving the model for all parameter values at once

Our procedure for approximating the value and policy functions extends the method of Duarte, Duarte, and Silva (2024) to discrete-time, general equilibrium problems with non-convexities. A key feature of this approach is that, instead of learning the value function for one particular set of parameters, we train a neural network to approximate the value function as a function of parameters as well as state variables. This means that we compute value functions for every possible combination of parameters in a given (infinite) set. In

the machine learning literature, the value function as a function of parameters and state variables is known as a universal value function (Schaul et al., 2015).

Expanding the set of state variables to include model parameters leads to a substantial increase in the state space. In our application, we add eight parameters to three state variables, increasing the dimension of the input space from three to eleven. Solving this problem requires an algorithm that can handle relatively large state spaces, which generally precludes traditional grid-based methods and polynomial approximations. There are many other methods for solving dynamic optimization problems with large state spaces, including ones that use machine learning tools to do so (e.g., Scheidegger and Billionis, 2019; Maliar et al., 2021; Azinovic et al., 2022; Maliar and Maliar, 2022; Duarte et al., 2023; Fernández-Villaverde et al., 2023), but we are not aware of any capable of accommodating a non-differentiable, non-convex objective function with both discrete and continuous actions. However, in principle, another such method could replace the procedure we describe in this subsection.³

Our approach is to use neural networks to approximate the value and policy functions in a policy iteration algorithm (e.g., Sutton and Barto, 1998). Policy iteration algorithms start from an arbitrary guess of the policy function and then iterate between (1) estimating the corresponding value function and (2) using the estimated value function to construct a new estimate of the policy function. In what follows, we describe each of the two steps in turn.

³ Note that replacing this procedure with a method that cannot handle the expansion of the state space, such as VFI, would eliminate the computational advantage of our method relative to the traditional approach. Without expanding the state space to include all model parameters, one would simply shift the computational bottleneck of solving the model thousands of times from one step of the algorithm (evaluating candidate parameter vectors in an outer loop) to another (producing enough simulated data to train moment networks).

3.1.1 Policy evaluation

For a given policy function, the associated value function satisfies the Bellman equation (10).

We use a neural network to approximate this value function, writing

$$V(z, k, b, \boldsymbol{\beta}) \approx V(z, k, b, \boldsymbol{\beta}; \Theta_V), \quad (13)$$

where $\boldsymbol{\beta} = (\theta, \rho, \sigma, \delta, \gamma_1, \gamma_0, \chi, c_f)$ is the vector of model parameters and Θ_V denotes the collection of adjustable weights of the neural network (see Appendix A.1 for background). Because the parameters play a fundamentally different role than the state variables—they define the economic environment, while the states describe the firm’s current position within that environment—we treat them differently in the network architecture. Rather than feeding parameters and states into the network as a single input vector, we allow the parameters to control how the network processes the states. For instance, when productivity is very persistent (ρ close to 1), the current productivity level z is highly informative about the future, and the value function should be more sensitive to z . Our architecture lets the parameters modulate this sensitivity at every layer of the network, so that a single network can represent a different value function for each parameter configuration. We describe the details of this architecture in Appendix A.2.2.

For a given policy function, we obtain the implied value function by adjusting network weights Θ_V to minimize the difference between the left-hand side and the right-hand side of the Bellman equation (10). Specifically, we define the mean squared Bellman residual as

$$L_V(\Theta_V) = \mathbb{E} \left[\left(V(z, k, b, \boldsymbol{\beta}; \Theta_V) - D - \beta \mathbb{E}_{z'|z} [\max\{V(z', k', b' - c', \boldsymbol{\beta}; \Theta_V), 0\}] \right)^2 \right], \quad (14)$$

where D is the dividend under the current policy and $k' = (1 + i - \delta)k$ is next-period capital. The outer expectation is taken over the joint distribution of states and parameters from which we sample during training. The inner expectation $\mathbb{E}_{z'|z}$ integrates over next-

period productivity using the model’s law of motion and is computed using Gauss-Hermite quadrature with five nodes, a standard numerical integration method that provides exact integration for polynomials up to degree nine.⁴ An additional complication, which is common to models with endogenous asset prices, is that the bond price depends on the value function through the default probability, creating a circular dependence. We break this circularity by computing the bond price using a lagged copy of the value function network, which we update at regular intervals during training (see Appendix A.3.4).

Minimizing Eq. (14) is a standard supervised learning problem and can be solved with stochastic gradient descent. Starting from a random initial guess of the weights Θ_V , gradient descent updates the weights in the direction that most quickly reduces the loss. The insight of stochastic gradient descent is to circumvent the computationally costly step of computing the outer expectation in Eq. (14) by approximating it with a random sample. We draw an i.i.d. sample of N state-parameter pairs, sampling parameters β uniformly from within parameter bounds (which narrow over time as described in Section 3.5) and, for each parameter vector, sampling states (z, k, b) uniformly from the parameter-implied state bounds. We approximate the loss as

$$\widehat{L}_V(\Theta_V) = \frac{1}{N} \sum_{i=1}^N (V_i - D_i - \beta \mathbb{E}_{z'|z_i}[\max\{V'_i, 0\}])^2, \quad (15)$$

where $V_i = V(z_i, k_i, b_i, \beta_i; \Theta_V)$ and $V'_i = V(z', k'_i, b'_i - c'_i, \beta_i; \Theta_V)$. We update the network weights by performing one step of stochastic gradient descent and advance to the policy improvement step.

⁴ The details of the quadrature formula are in Appendix A.3.5. An alternative, used in the continuous-time method of Duarte, Duarte, and Silva (2024), takes a second-order Taylor expansion, which is exact for diffusion processes but may introduce error in discrete time.

3.1.2 Policy improvement

Given the value function V , we would like to obtain a new estimate of the policy function as

$$(i, b', c') = \arg \max_{i, b', c'} \{D(z, k, b, i, b', c') + \beta \mathbb{E}_{z'|z}[\max\{V(z', k', b' - c', \beta; \Theta_V), 0\}]\}. \quad (16)$$

This optimization problem is generally computationally costly. Instead of computing the exact maximum at each state, we use neural networks to approximate each of the three policy functions and optimize with respect to their weights. Denoting the policy network weights by Θ_π , we draw a sample of N state-parameter pairs as in the policy evaluation step and minimize

$$\widehat{L}_\pi(\Theta_\pi) = -\frac{1}{N} \sum_{i=1}^N (D_i + \beta \mathbb{E}_{z'|z_i}[\max\{V'_i, 0\}]), \quad (17)$$

holding the value network weights Θ_V fixed. Minimizing \widehat{L}_π is equivalent to maximizing the right-hand side of the Bellman equation at the sampled points. We update the policy network weights by performing one step of stochastic gradient descent and return to the policy evaluation step. The algorithm cycles between these two steps 500 times, gradually improving both the value and policy functions. We refer to one complete cycle of 500 such alternating steps as an epoch. We discuss the number of steps per epoch and other hyperparameter choices in Appendix A.12.

3.1.3 Smooth approximation

The dividend function D in our model includes indicator functions for equity issuance costs and fixed adjustment costs, which introduce non-differentiabilities. Similarly, the endogenous default threshold introduces non-differentiability through the bond price. These prevent gradients from flowing through these regions during training. We address this by replacing the indicator functions with smooth approximations during training, so that the network can learn through the kinks, which we discuss in Appendix A.4. To ensure accuracy, we refine this approximation during the simulation block of computations, described below.

3.2 Block 2: Simulating moments

The computations performed in this block are conceptually simple but computationally intensive. Our goal is to construct a dataset to use in the estimation procedure. For each sampled parameter vector, we use the refined policy functions to simulate a panel of 5,000 firms over 300 time periods, discarding the first 200 periods as burn-in. At each period, the firm’s state (z, k, b) evolves according to the model’s law of motion: productivity follows the Tauchen discretization of the AR(1) process in Eq. (2), capital accumulates as in Eq. (4), and the controls are determined by the policy function. Firms whose continuation value falls below zero default and are replaced with new draws of (k, b) , keeping the same productivity z . From each simulated panel, we compute the 11 targeted moments described in Section 4 below. We provide more detail on the moment simulation procedure in Appendix A.6.

This data collection process runs on three GPUs that operate independently of each other, and asynchronously with the neural network training in Block 1. While the value and policy networks continue to improve on one GPU, the simulator evaluates the latest networks, refines them on the grid as described below, simulates panels, and accumulates moment observations on three other GPUs. The output is a dataset that pairs each parameter vector with its model-implied moments. We describe this asynchronous execution in detail in Appendix A.11.

3.2.1 Grid refinement

To remove the smooth approximation described in Section 3.1.3 and improve accuracy, we refine the solution each time a new parameter vector is evaluated. We evaluate the value and policy networks on a discrete grid and run a few steps of policy iteration using the exact dividend function. Each step consists of a policy improvement step, in which we search locally around the current policy for better controls, and a policy evaluation step, in which we compute the value function implied by the current policy, rather than iterating the

Bellman equation to convergence. Because of the limited liability constraint, this requires a short iterative procedure that converges in a few steps (Appendix A.5). Because the bond price depends on the value function through the default probability, we alternate between updating the value function and recomputing bond prices.

Incorporating a few steps of refinement on a grid into our approach is still orders of magnitude faster than solving our model with traditional VFI for two reasons. First, the refinement algorithm itself is more efficient than VFI. It adapts the policy iteration approach used in AI to a grid, computing the implied value function for a given policy rather than iterating the Bellman equation to convergence. Second, the neural networks are already very close to the solution, so only a few steps of this refinement are needed.

In applications with smooth objective functions, this refinement step should not be necessary and may even reduce accuracy, since a grid is also an approximation. We illustrate and discuss the implementation of this refinement step in Appendix A.5.

3.3 Block 3: Estimating parameters

In this block, we use the dataset described in Section 3.2 above to train a separate neural network to approximate the mapping between model parameters and moments, which we refer to as the “moment function.” We then describe how we use the moment function for estimation.

Let $\boldsymbol{\beta} = (\theta, \rho, \sigma, \delta, \gamma_1, \gamma_0, \chi, c_f)$ be the vector of model parameters and let m_j denote the j -th simulated moment. We are interested in learning the function

$$g_j(\boldsymbol{\beta}) \equiv \mathbb{E}[m_j | \boldsymbol{\beta}]. \tag{18}$$

We search for a neural network $g_j(\boldsymbol{\beta}; \Theta_j)$ that approximately minimizes

$$\mathcal{L}(\Theta_j) = \mathbb{E}[(m_j - g_j(\boldsymbol{\beta}; \Theta_j))^2], \quad (19)$$

where Θ_j are the weights of the network for moment j . We draw an i.i.d. sample of pairs $\{(\boldsymbol{\beta}_i, m_{i,j})\}_{i=1}^{N_d}$ from the dataset constructed in Block 2 and approximate the loss as

$$\widehat{\mathcal{L}}(\Theta_j) = \frac{1}{N_d} \sum_{i=1}^{N_d} (m_{i,j} - g_j(\boldsymbol{\beta}_i; \Theta_j))^2. \quad (20)$$

As before, this optimization can be done with stochastic gradient descent. We train separate networks for each moment, each with three hidden layers of 32 units. To avoid overfitting, we use 10-fold cross-validation: for each moment, we partition the training data into 10 folds and train 10 networks, each holding out one fold for validation (Appendix A.7). This produces 10 separate approximations of the moment function, which we use to assess sensitivity.

Once training is complete, the networks $g_j(\boldsymbol{\beta}; \Theta_j)$ have learned a differentiable mapping between parameters and the moments of interest. This has two major implications for structural estimation. First, it is a fast way to evaluate moments for different sets of parameters—evaluating all 11 moments takes milliseconds rather than the minutes required for a full model solution and simulation. Second, because the networks are differentiable, we can compute the analytical Jacobian $\partial g/\partial \boldsymbol{\beta}$, allowing us to use efficient gradient-based optimization.

We estimate the model parameters by minimizing a weighted distance between data moments and model-implied moments:

$$\widehat{\boldsymbol{\beta}} = \arg \min_{\boldsymbol{\beta}} d(\boldsymbol{\beta}) \equiv (\widehat{\boldsymbol{m}} - g(\boldsymbol{\beta}))' W (\widehat{\boldsymbol{m}} - g(\boldsymbol{\beta})), \quad (21)$$

where $\widehat{\boldsymbol{m}}$ is the vector of data moments, $g(\boldsymbol{\beta})$ stacks the 11 moment network outputs, and

W is a positive definite weighting matrix. We use as our weighting matrix the inverse of the moment variance-covariance matrix. Following the approach of Erickson and Whited (2002), we stack the influence functions for all moments and covary them to compute the moment variance-covariance matrix, clustering at the firm level. This choice of weighting matrix has been shown to result in better finite sample performance (Bazdresch, Kahn, and Whited, 2017).

We use a Levenberg-Marquardt algorithm with 30 random starting points to avoid local minima. Levenberg-Marquardt combines gradient descent with the Gauss-Newton method, acting more like gradient descent when far from the solution and more like Gauss-Newton near it. Note that it is only possible to use an efficient gradient-based algorithm like Levenberg-Marquardt because the moment network approximation provides the analytical Jacobian. With each of the 10 cross-validation folds producing a separate moment function, we obtain 10 parameter estimates, whose variation provides a measure of the sensitivity of the results to the moment-network approximation. We describe this procedure in more detail in Appendix A.8.

3.4 General equilibrium extension

In the general equilibrium version of the model, described in Section 2.2, the wage w must satisfy the resource constraint $Y - I - C - S = 0$. In a standard estimation procedure, this requires three nested loops. An outer-loop optimizer proposes candidate parameter vectors. For each candidate, a root-finding algorithm searches for the equilibrium wage. At each candidate wage, the researcher solves the firm's dynamic problem with value function iteration and simulates the model at the solution to evaluate the resource constraint. This makes estimation substantially more computationally intensive in general equilibrium than in partial equilibrium, often prohibitively so.

Our method eliminates these loops. Instead, we include the wage w in the parameter

vector alongside the structural parameters, so that the neural networks learn the model solution for all combinations of $(\theta, \rho, \sigma, \delta, \gamma_1, \gamma_0, \chi, c_f, w)$ simultaneously. We then add the market-clearing imbalance—the “resource gap” $Y - I - C - S$ —as an additional targeted moment, with a target value of zero. We scale the resource gap by aggregate sales Y so that this moment is on a similar scale as the others. The estimation procedure in Block 3 finds the wage that clears the market jointly with the structural parameters that match the data moments, in a single optimization step. As a result, this general equilibrium extension takes about as long to estimate as the partial equilibrium model. We describe the modifications required to accommodate this extension, including how we construct the weighting matrix, in Appendix A.10.

3.5 *Minimum loss functions and adaptive shrinkage*

The algorithm starts by sampling parameters uniformly across wide bounds that span the range of economically plausible values. As training progresses, it narrows these bounds to focus computational effort on the region around the best achievable fit for each parameter. To guide this narrowing for a given parameter, we minimize the loss function over all other parameters, which gives us the best achievable fit for that parameter as a function of parameter values.

Specifically, for each parameter β^j in the parameter vector $\boldsymbol{\beta}$, we minimize the estimation objective over the remaining parameters $\boldsymbol{\beta}^{-j}$. That is, we evaluate

$$L(\beta^j) = \min_{\boldsymbol{\beta}^{-j}} (\hat{m} - g(\beta^j, \boldsymbol{\beta}^{-j}))' W (\hat{m} - g(\beta^j, \boldsymbol{\beta}^{-j})). \quad (22)$$

The resulting curve shows how the best achievable moment fit varies with β^j , accounting for the fact that other parameters adjust in response.

We call $L()$ “minimum loss functions”⁵ and use them to progressively shrink the parameter bounds, so as to focus computations in the relevant range. After a certain amount of training, we attempt to shrink the bounds after each training epoch. We do so by attempting to restrict the interval to the region around the global minimum of the minimum loss function, while still retaining any plausible parameter vector. For parameters whose minimum loss function is flat or differs substantially across the 10 folds, the bounds do not shrink. Similarly, if tightening bounds would exclude parameter vectors recently deemed plausible by the optimizer, the bounds do not shrink. We describe this procedure in detail in Appendix A.9.3.

3.6 *Global identification diagnostic*

In addition to guiding the adaptive shrinkage of parameter bounds, the minimum loss functions defined in Eq. (22) provide a diagnostic for global identification. To evaluate whether a parameter is identified by the set of moments, we examine its minimum loss function $L()$. A sharp, unique minimum indicates that the parameter is identified by the set of moments used in the estimation, as there is only a single value of the parameter that minimizes the distance between model and data moments. In contrast, a flat curve indicates that many parameter values can achieve a similar fit, suggesting weak identification. By minimizing the distance function with respect to all other parameters instead of holding them fixed at their estimated values, this procedure accounts for the fact that changing the value of one parameter can change the optimal values of other parameters.

When computing $L(\beta^j)$ for this diagnostic, we use simulated moments at our parameter estimates β^{j*} as the target \hat{m} , instead of the data moments themselves. By replacing the data moments with model-implied moments at the estimated parameters, the model becomes

⁵An analogous concept in statistics is a profile likelihood curve, which traces the maximized likelihood as a function of one parameter with all others optimized out. We use the term “minimum loss function” because we profile the GMM objective rather than a likelihood.

correctly specified, and we can isolate weak identification from model misspecification. When we instead target data moments, a flat minimum loss function can reflect that the moments are not sufficiently informative about a parameter or that the model does not closely reproduce the data. In addition to assessing the shape of minimum loss functions, this allows us to test the model’s ability to recover the estimated parameter from the moments they imply, which is also informative about whether the parameters are identified by this set of moments. We further detail this diagnostic in Appendix A.9.3.

3.7 Summary of the implementation

The full algorithm proceeds as follows. On one GPU, the value and policy networks train continuously, drawing random batches of states and parameters from the current bounds and cycling through policy evaluation and policy improvement steps. On three other GPUs, the simulator draws parameter vectors, evaluates the latest networks on the discrete grid, refines the solution, simulates firm panels, and computes moments. The training and data collection processes run concurrently.

Periodically, the accumulated moment data are used to retrain the networks that approximate the moment functions, and the Levenberg-Marquardt optimizer produces updated parameter estimates. The algorithm computes minimum loss functions and uses them to narrow the parameter bounds for the next round of training and simulation. This iterative tightening concentrates computational resources on the relevant region of the parameter space.

We describe each implementation step in detail and provide practical guidance for researchers adapting this method to other applications in Appendix A.

4 Main results

In this section, we apply the estimation method described in Section 3 to the partial equilibrium version of the model in Section 2. We estimate eight parameters jointly: the returns-to-scale parameter θ , the autocorrelation ρ and standard deviation σ of the productivity process, the depreciation rate δ , the convex and fixed capital adjustment costs γ_1 and γ_0 , the default recovery rate χ , and the fixed operating cost c_f . The remaining parameters—the risk-free rate r_f , the capital share α , the equity issuance costs λ_0 and λ_1 , and the tax rate τ —are set externally following Gao, Whited, and Zhang (2021). We set $r_f = 0.02$, $\alpha = 0.3$, $\tau = 0.2$, $\lambda_0 = 0.007$, and $\lambda_1 = 0.054$. In the partial equilibrium specification, the wage is fixed at $w = 1$.

We target 11 moments. The first eight are the means and standard deviations of cash, debt, investment, and operating income, all expressed as a ratio to total assets. We also include the serial correlation of operating income. The last two moments are regression coefficients motivated by the approach in Bazdresch, Kahn, and Whited (2017), who propose the idea of matching empirical estimates of the model policy functions. Namely, the last two moments are the regression coefficients from regressing the change in cash (scaled by total assets) on net debt and lagged operating income, both expressed as ratios to total assets.

We start by showing that our method can accurately recover known parameters in a Monte Carlo exercise. We then illustrate the shrinkage algorithm described in Section 3.5. Next, we estimate the partial equilibrium model using Compustat data and compare the performance of our method with the standard approach of combining value function iteration with a gradient-free global optimizer. Finally, we use the global diagnostic developed in Section 3.6 to assess identification.

4.1 *Recovering known parameters*

To assess the performance of our method, we start by showing the results of an exercise in which the true parameter values are known. We solve the model using value function iteration for a given set of parameters, use this solution to produce simulated data, compute the 11 targeted moments, and then use the algorithm described in Section 3 to estimate the model targeting these moments. The advantage of this exercise is that we know the true parameters that generated the underlying data and can thus assess our method’s ability to recover them. This is useful for building confidence that our method can arrive at a known solution, though estimating parameters with a correctly specified data-generating process is an inherently simpler exercise than the estimation with Compustat data in Section 4.2 below.

We repeat this exercise multiple times to ensure that our method performs well across the parameter space. We draw parameter vectors from a uniform distribution over the bounds reported in Table A1. For each parameter vector, we solve the model using value function iteration, simulate data using the policy functions, compute the targeted moments, and estimate the model using our method and these targets. As we discuss in Section 4.2 below, the recovery rate χ is generally weakly identified in this model. When a simulation has no defaulting firms, neither our method nor the standard approach can reliably recover χ because it is not identified by this set of moments. We thus filter parameter vectors in this exercise to exclude simulations in which no firm defaults, so as to isolate the performance of our method from whether the model is identified, and sample 40 such parameter vectors. In Section 4.2.3, we show that the estimated parameters fall within a low-default region, and we cannot accurately recover the estimated χ .

We start by showing that we recover the true moments. In Figure 1, we show scatter plots of true moments and moments implied by our method, with each panel representing a different moment. The x-axis of each panel represents the targeted moments implied by

each parameter vector, and the y-axis represents the simulated moments at the parameter estimates obtained with our method. The 45-degree line is shown in solid black, so that if the fit were perfect, all dots would lie on that line. The fit is excellent, with an R^2 of 1.00 for all 11 moments. This close fit of the moments is important since errors in moment conditions would pose challenges for the estimation.

Next, we show that we also obtain a close fit of the true parameters. In Figure 2, we show scatter plots of the true parameters and the parameters we estimate, with each panel representing a different parameter. In each panel, the x-axis is the set of randomly drawn parameters, and the y-axis is the set of parameter estimates obtained with our method, with the 45-degree line shown in black. Our parameter estimates are a close fit to the true parameters, with $R^2 \geq 0.97$ for seven of the eight parameters.

4.1.1 Adaptive shrinkage

Next, we illustrate our adaptive shrinkage algorithm, which progressively narrows the parameter bounds during training to concentrate computational effort near the parameter estimates, as described in Section 3.5. Figure 3 shows the distribution of sampled parameters at four training epochs for the exercise described above. Panels (a) and (b) show histograms of the fixed adjustment cost γ_0 and the autocorrelation of productivity ρ , with the red vertical line denoting the true parameter value.

Shrinking does not start until epoch 200, so the snapshot from epoch 50 shows uniform sampling across the initial parameter bounds. After epoch 200, as training progresses, the distributions tighten around the true values. They do so considerably more quickly for γ_0 than for ρ , for which sampling is still not concentrated around the true parameter value by epoch 300 (shown in green) and is still wider than the γ_0 interval at the end of training (epoch 400, shown in blue). This reflects the fact that ρ is not as well identified as γ_0 by this set of moments, and the safeguards in our algorithm that prevent shrinking for weakly

identified parameters (see Appendix A.9.2). Panel (c) shows the joint distribution of ρ and γ_0 , with the true values marked by a red circle, illustrating how the algorithm simultaneously narrows both dimensions of the parameter space.

4.2 Estimation with Compustat data

Next, we use data from Compustat between 1970 and 2019 to construct data moments and use the algorithm of Section 3 to estimate the model targeting these moments.

4.2.1 Data construction

We use data from the Compustat annual database between 1970 and 2019. We restrict our sample to firms headquartered in the U.S. and exclude firms in the financial (SIC codes 6000–6999), regulated (SIC codes 4900–4999), and quasi-governmental (SIC codes 9000+) sectors. We exclude observations with missing data in any variable, as well as those with total assets below \$10 million, with negative sales, or with sales or asset growth exceeding 200%. We require each firm to have at least four consecutive annual observations.

We define our variables to be conceptually close to their model counterparts. Cash, debt, operating income, and investment are given by Compustat items CH, DLC+DLTT, OIBDP, and CAPX, respectively, and each is scaled by total assets (AT). In our simulated data, cash, debt, operating income, and investment are given by $c/(k+c)$, $b/(k+c)$, $\pi(z,k)/(k+c)$, and $(k' - (1-\delta)k)/k$, respectively (Appendix A.6.2).

We winsorize all ratios at the 1st and 99th percentiles by year. Because firms in the model are ex-ante homogeneous, we remove firm fixed effects from the standard deviation and autocorrelation moments by demeaning variables at the firm level and adding back the sample average.

4.2.2 Estimation results

We estimate the model using the algorithm described in Section 3. We show the results of our estimation in Table 1, with panel (a) reporting parameter estimates and panel (b) reporting the moment fit. Our estimates are precise for all parameters other than the recovery rate χ . We obtain very similar estimates to Gao, Whited, and Zhang (2021) despite small differences in the data moments, except for the fixed cost c_f and the recovery rate χ . As we discuss in Section 4.2.3 below, the latter is weakly identified by this set of moments, as Gao, Whited, and Zhang (2021) also notes.

The key advantage of our method is computational efficiency, which we illustrate by comparing our estimation run time against the traditional approach of solving the model with VFI and using simulated annealing to search over the parameter space. We implement this in JAX and run it in the same GPU cloud computing service we use for our method, described in Appendix A.12.10. We thus isolate efficiency gains arising from our algorithm and not our use of AI software and hardware.⁶

We report this comparison in Figure 4. The x-axis is run time in minutes, and the y-axis is the running best of the estimation loss function, averaged across 4 seeds. We report the average loss for both methods starting at 3 minutes, because our method spends its first minutes training the value, policy, and moment networks before producing its first parameter estimate, and therefore does not output a loss before the 3-minute mark. By the time the first estimate is produced, the moment networks already provide a good approximation of the mapping between parameters and moments, so our method’s first reported loss is already low.

We run our method for 90 minutes and simulated annealing for 1,200 minutes, or 20 hours. The dashed blue line indicates that training has ended, and the loss remains constant

⁶For benchmarking of computational gains obtained in traditional dynamic programming from using AI software and hardware, see Duarte et al. (2020).

thereafter. Our method converges in less than an hour, while simulated annealing has still not converged after 20 hours. At the end of the 20-hour run, simulated annealing arrives at a loss that our method takes less than 13 minutes to obtain.

4.2.3 Global identification

Finally, we illustrate our global identification diagnostic. As we describe in Section 3.6 and in Appendix A.9.3, we start by using the vector of parameter estimates from Section 4.2.2 to simulate moments. We then estimate the model again targeting these model-implied moments, and minimize the distance between model and target moments for all parameters but one to compute minimum loss functions. Our baseline diagnostic uses simulated moments at the estimated parameters as the targets, rather than the data moments themselves, so as to isolate identification from model misspecification. We then check whether the resulting minimum loss functions have a unique global minimum for each parameter, and whether the second estimation recovers the parameters from the first estimation, which were used to produce the targeted moments.

In Figure 5, we show minimum loss functions for four parameters: the fixed adjustment cost γ_0 (panel (a)), the depreciation rate δ (panel (b)), the recovery rate χ (panel (c)), and the autocorrelation of the productivity process ρ (panel (d)). The solid red line denotes the parameters from the estimation in Section 4.2.2 above, which are used to simulate moments that are then targeted in a new estimation. The dashed red line denotes the estimates obtained by targeting the model-implied moments. The x-axis spans the initial parameter bounds, with the black dashed vertical lines and shaded region marking the narrowed bounds after adaptive shrinkage (Section 3.5).

The top row (γ_0 and δ) shows examples of parameters that are well identified by this set of moments. This can be seen by their sharp minimum loss functions, each with a clear global minimum, and by the fact that the second estimation perfectly recovers the parameters used

to produce the targeted moments. In contrast, the recovery rate χ in panel (c) is clearly not identified by this set of moments. Not only is the minimum loss function flat, but we also fail to get close to the parameter value used to generate the moments. The autocorrelation of productivity in panel (d) also has a relatively flat minimum loss function, suggesting weak identification, but we recover an estimate that is close to the parameter value that was used to produce the targeted moments. Note that the bounds for the parameters in the bottom row were not shrunk, illustrating the safeguards that prevent narrowing of the bounds for weakly identified parameters (see Appendix A.9.2 for details).

For completeness, we also report the minimum loss functions obtained targeting the data moments rather than model-implied moments at the estimated parameter vector. Appendix Figure B2 shows the results of this exercise for the same set of parameters as Figure 5. The minimum loss functions for the parameters that the exercise above suggests are well identified by this set of moments (γ_0 and δ) now look less compelling. This highlights the usefulness of separating model misspecification from weak identification by targeting model-implied moments.

5 General equilibrium extension

In this section, we estimate the general equilibrium extension of the model described in Section 2.2 using the approach outlined in Section 3.4. With the standard approach, estimation of a general equilibrium model requires an additional root-finding algorithm to solve for the equilibrium wage for each candidate parameter vector. Instead, we solve the model for all possible values of model parameters and equilibrium wages, then find the parameters that minimize the distance between model and data moments and the wage that clears markets with fast, gradient-based optimization.

We estimate the same eight model parameters and target the same 11 moments as in Section 4. We include the equilibrium wage w as a ninth pseudo parameter in the estimation

and add the (scaled) resource constraint imbalance $(Y - I - C - S)/Y$ as a twelfth moment with a target value of zero. We describe all changes to the estimation procedure relative to the partial equilibrium case in Appendix A.10.

We start by showing that our method can accurately recover known parameters and equilibrium wages in a Monte Carlo exercise, similar to the one in Section 4.1. We then estimate the general equilibrium model using Compustat data and, as before, compare our performance against the standard approach of combining value function iteration with simulated annealing.

5.1 *Recovering known parameters*

We draw 40 parameter vectors from a uniform distribution over the parameter bounds reported in Table A1, as in Section 4.1. For each parameter vector, we solve the model using value function iteration and find the market-clearing wage using Brent’s method, a root-finding algorithm that combines bisection with interpolation. We then simulate data from this solution, compute the targeted moments, and estimate the model using our method and these targets.

Figure 6 is analogous to Figure 1, and shows scatter plots of the 11 data moments for the estimation of the general equilibrium model. The fit is again excellent, with an R^2 of 1.00 for all 11 moments. In Figure 7, we show scatter plots of the true and estimated structural parameters. The fit is very good for most parameters, with $R^2 \geq 0.97$ for all but one. We obtain a worse fit for the autocorrelation of productivity ρ , which, as we show in Section 4.2.3 above, is not as well identified as the others.⁷

In Figure 8, we show the equilibrium wage recovered by our method against the equilibrium wage obtained with the traditional root-finding approach. The R^2 is 0.99, confirming

⁷We also show that the recovery rate χ is not identified in the absence of default, but this Monte Carlo exercise restricts to simulations where at least one firm defaults, as described in Section 4.1.

that our novel approach of treating the wage as a pseudo parameter and targeting the market-clearing imbalance as a moment condition accurately recovers equilibrium prices.

5.2 *Estimation with Compustat data*

Finally, we estimate the general equilibrium model targeting the Compustat data moments described in Section 4.2.1 and a market-clearing imbalance of zero. We show results in Table 2, with panel (a) reporting parameter estimates and panel (b) reporting the moment fit. Our estimates are again fairly precise, with the exception of the recovery rate χ . Among the moments we report in panel (b) is the scaled resource gap $(Y - I - C - S)/Y$, with our simulated gap being around -1%. As we explain in Appendix A.10, one can obtain tighter market clearing, meaning a gap that is closer to zero, by increasing the weight placed on this moment. We do not tune this weight because our imbalance of 1% is within the tolerance used to solve for the wage in the traditional nested approach (Gao, Whited, and Zhang, 2021).

We again compare our estimation run time against the traditional approach of solving the model with VFI and using simulated annealing to search over the parameter space, implemented in JAX and run with GPU cloud computing. We report this comparison in Figure 9, running our method for 90 minutes and simulated annealing for 1,200 minutes, or 20 hours. Our method converges within an hour and simulated annealing remains far from our estimation loss after 20 hours. Our method takes just over 13 minutes to arrive at the estimation loss that simulated annealing obtains after 20 hours of optimization. In Appendix Figure B3, we run simulated annealing for 100 hours, or over four days, at which point it is still above the loss we obtain in 17 minutes.

6 Conclusion

We propose a highly efficient method to estimate dynamic equilibrium models by expanding the set of state variables to include all model parameters and approximating the mapping between parameters and model-implied moments directly from simulated observations. We also introduce a new approach to solve for general equilibrium prices by treating them as pseudo parameters and targeting the market-clearing imbalance as additional moment conditions. In our application, this approach makes estimation in general equilibrium as tractable as in partial equilibrium. After moment networks are trained on simulated observations, the estimation can be carried out as if we had the moment conditions in closed form. These same moment networks can also be used to assess identification globally, by verifying whether the estimation objective function has a unique minimum for each parameter.

We illustrate our approach by solving and estimating a dynamic equilibrium model of leverage and investment with three state variables, three controls, endogenous default, costly equity issuance, and non-convex adjustment costs, similar to Gao, Whited, and Zhang (2021). Our method reduces the estimation time from days to minutes, taking less than 20 minutes to arrive at the estimation loss that the traditional approach obtains in days.

To lower the cost of adopting our method, our replication package will provide a library and an AI agent that can implement new models from a description of the economic environment in natural language. We have verified that the DF Assistant can apply our method to simple dynamic programming and estimation problems. We have yet to test more complex models, and it is possible that the agent’s capabilities already far exceed what our test cases require. If not, we expect they soon will as LLMs improve and we add examples to our library.

References

- Andrews, Isaiah, Matthew Gentzkow, and Jesse M. Shapiro. Measuring the sensitivity of parameter estimates to estimation moments. *The Quarterly Journal of Economics*, 132(4):1553–1592, 06 2017. ISSN 0033-5533. doi: 10.1093/qje/qjx023. URL <https://doi.org/10.1093/qje/qjx023>.
- Azinovic, Marlon, Luca Gaegauf, and Simon Scheidegger. Deep equilibrium nets. *International Economic Review*, 63(4):1471–1525, 2022. doi: <https://doi.org/10.1111/iere.12575>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/iere.12575>.
- Bazdresch, Santiago, R. Jay Kahn, and Toni M. Whited. Estimating and testing dynamic corporate finance models. *The Review of Financial Studies*, 31(1):322–361, 07 2017. ISSN 0893-9454. doi: 10.1093/rfs/hhx080. URL <https://doi.org/10.1093/rfs/hhx080>.
- Catherine, Sylvain, Mehran Ebrahimian, Mohammad Fereydounian, David Sraer, and David Thesmar. Robustness checks in structural analysis. Technical report, Working paper, 2023.
- Duarte, Victor, Diogo Duarte, Julia Fonseca, and Alexis Montecinos. Benchmarking machine-learning software and hardware for quantitative economics. *Journal of Economic Dynamics and Control*, 111:103796, 2020. ISSN 0165-1889. doi: <https://doi.org/10.1016/j.jedc.2019.103796>. URL <https://www.sciencedirect.com/science/article/pii/S0165188919301939>.
- Duarte, Victor, Julia Fonseca, Aaron Goodman, and Jonathan Parker. Simple allocation rules and optimal portfolio choice over the lifecycle. Technical report, Working paper, 2023.
- Duarte, Victor, Diogo Duarte, and Dejanir Silva. Machine learning for continuous-time finance. *Review of Financial Studies*, 37(11):3217–3271, 2024.
- Duffie, Darrell, and Kenneth J. Singleton. Simulated moments estimation of markov models

- of asset prices. *Econometrica*, 61(4):929–952, 1993. ISSN 00129682, 14680262. URL <http://www.jstor.org/stable/2951768>.
- Erickson, Timothy, and Toni M. Whited. Two-step GMM estimation of the errors-in-variables model using high-order moments. *Econometric Theory*, 18(3):776–799, 2002. ISSN 02664666, 14694360. URL <http://www.jstor.org/stable/3533649>.
- Fernández-Villaverde, Jesús, Samuel Hurtado, and Galo Nuño. Financial frictions and the wealth distribution. *Econometrica*, 91(3):869–901, 2023. doi: <https://doi.org/10.3982/ECTA18180>. URL <https://onlinelibrary.wiley.com/doi/abs/10.3982/ECTA18180>.
- Gao, Xiaodan, Toni M. Whited, and Na Zhang. Corporate money demand. *The Review of Financial Studies*, 34(4):1834–1866, 2021. doi: 10.1093/rfs/hhaa083.
- Gourieroux, C., A. Monfort, and E. Renault. Indirect inference. *Journal of Applied Econometrics*, 8:S85–S118, 1993. ISSN 08837252, 10991255. URL <http://www.jstor.org/stable/2285076>.
- Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.
- Ivanov, Ivan T., Luke Pettit, and Toni M. Whited. Taxes depress corporate borrowing: Evidence from private firms. *The Review of Economic Studies*, 2025. doi: 10.1093/restud/rdaf094.
- Kingma, Diederik P., and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.
- Lee, Bong-Soo, and Beth Ingram. Simulation estimation of time-series models. *Journal of Econometrics*, 47(2-3):197–205, 1991. URL <https://EconPapers.repec.org/RePEc:eee:econom:v:47:y:1991:i:2-3:p:197-205>.
- Maliar, Lilia, and Serguei Maliar. Deep learning classification: Modeling discrete labor

- choice. *Journal of Economic Dynamics and Control*, 135:104295, 2022. ISSN 0165-1889. doi: <https://doi.org/10.1016/j.jedc.2021.104295>. URL <https://www.sciencedirect.com/science/article/pii/S016518892100230X>.
- Maliar, Lilia, Serguei Maliar, and Pablo Winant. Deep learning for solving dynamic economic models. *Journal of Monetary Economics*, 122:76–101, 2021. ISSN 0304-3932. doi: <https://doi.org/10.1016/j.jmoneco.2021.07.004>. URL <https://www.sciencedirect.com/science/article/pii/S0304393221000799>.
- McFadden, Daniel. A method of simulated moments for estimation of discrete response models without numerical integration. *Econometrica*, 57(5):995–1026, 1989. ISSN 00129682, 14680262. URL <http://www.jstor.org/stable/1913621>.
- Norets, Andriy. Estimation of dynamic discrete choice models using artificial neural network approximations. *Econometric Reviews*, 31(1):84–106, 2012. doi: 10.1080/07474938.2011.607089. URL <http://dx.doi.org/10.1080/07474938.2011.607089>.
- Pakes, Ariel, and David Pollard. Simulation and the asymptotics of optimization estimators. *Econometrica*, 57(5):1027–1057, 1989. ISSN 00129682, 14680262. URL <http://www.jstor.org/stable/1913622>.
- Perez, Ethan, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. FiLM: Visual reasoning with a general conditioning layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
- Schaul, Tom, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *Proceedings of the 32nd International Conference on Machine Learning, ICML’15*, pages 1312–1320. JMLR.org, 2015.
- Scheidegger, Simon, and Ilias Bilionis. Machine learning for high-dimensional dynamic stochastic economies. *Journal of Computational Science*, 33:68–82, 2019. ISSN 1877-7503.

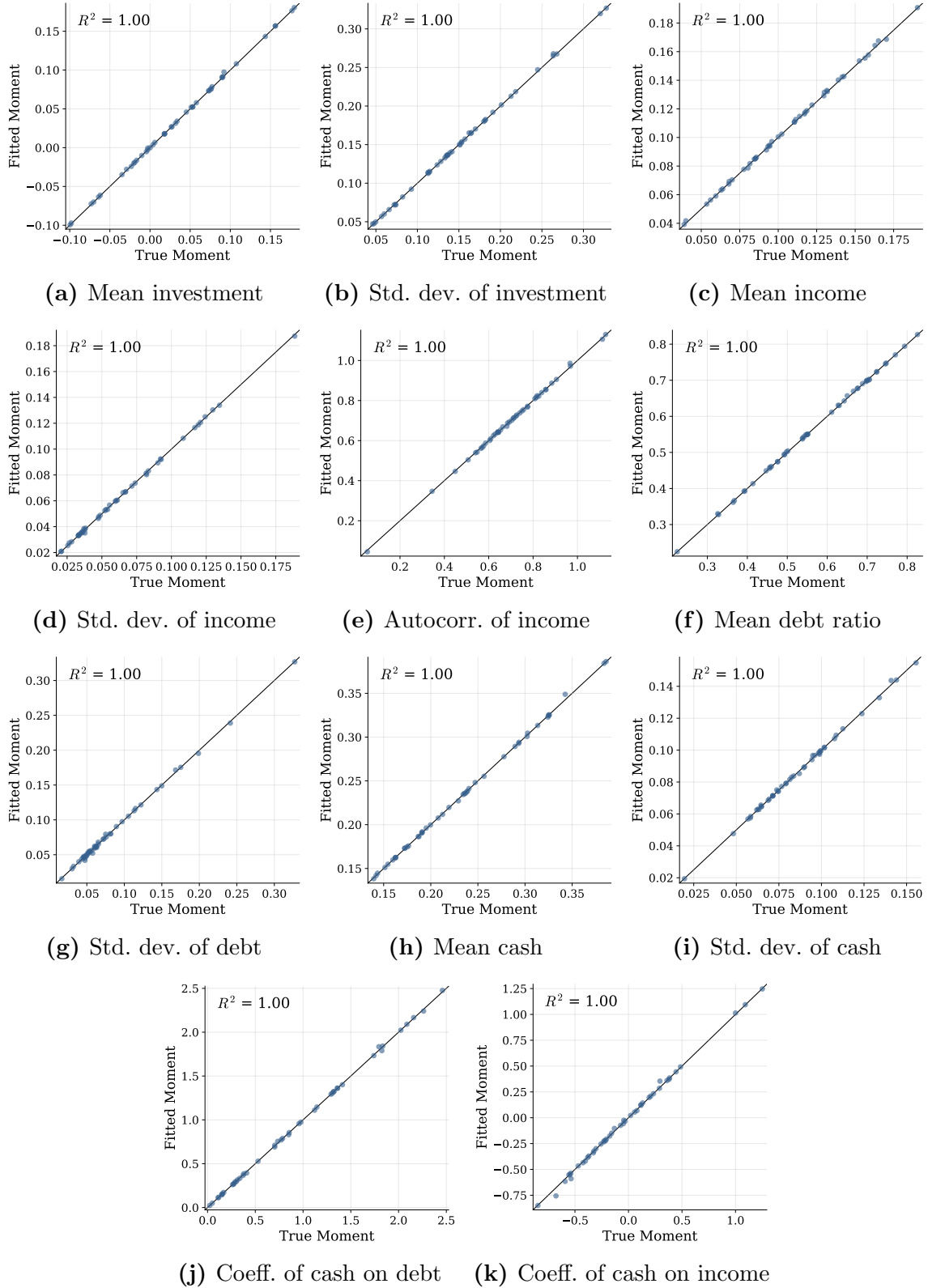
doi: <https://doi.org/10.1016/j.jocs.2019.03.004>. URL <https://www.sciencedirect.com/science/article/pii/S1877750318306161>.

Strebulaev, Ilya A., and Toni Whited. Dynamic models and structural estimation in corporate finance. *Foundations and Trends(R) in Finance*, 6(1–2):1–163, 2012. URL <https://EconPapers.repec.org/RePEc:now:fntfin:0500000035>.

Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

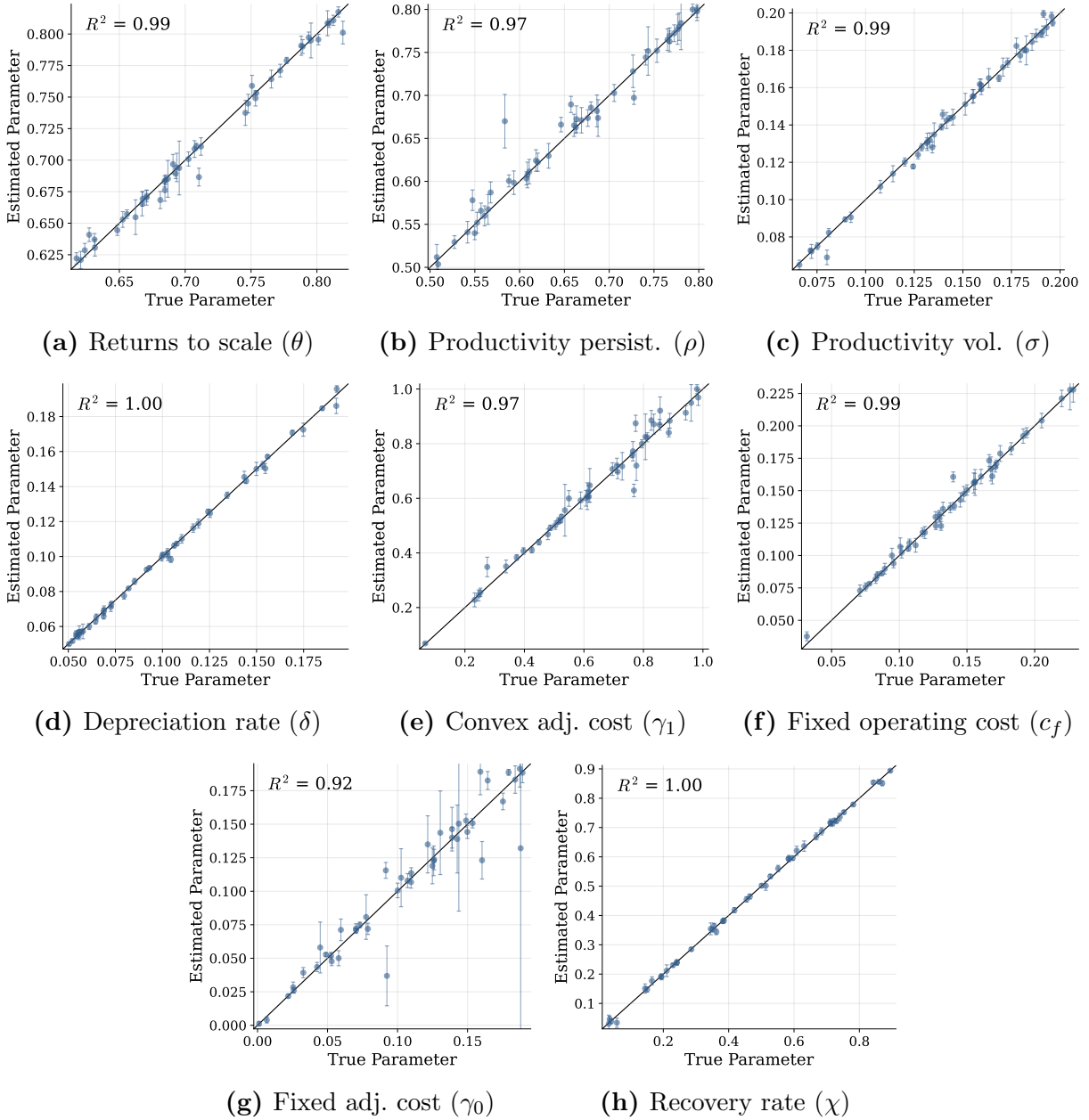
Yang, Yucheng, Chiyuan Wang, Andreas Schaab, and Benjamin Moll. Structural reinforcement learning for heterogeneous agent macroeconomics. Technical report, Working paper, 2025.

Figure 1: True vs. Fitted Moments: Partial Equilibrium Model



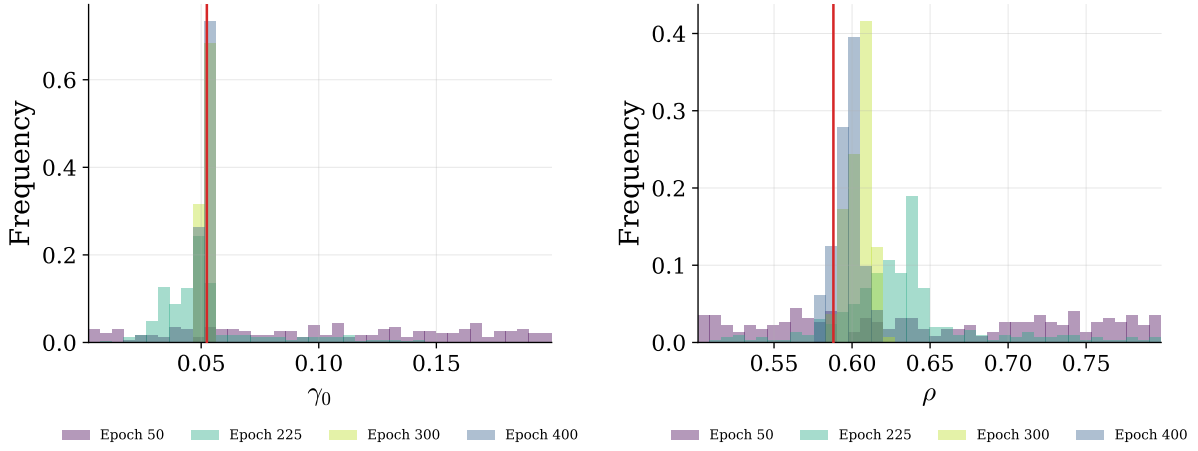
Each panel corresponds to one of 11 targeted moments, described in Section 4. We draw 40 parameter vectors and, for each, solve the model using value function iteration and simulate moments, which we report in the x-axis. We then estimate the model using our method (Section 3) targeting these moments, then simulate moments using our solution at the estimated parameters and report those in the y-axis. The solid line is the 45-degree line.

Figure 2: True vs. Estimated Parameters: Partial Equilibrium



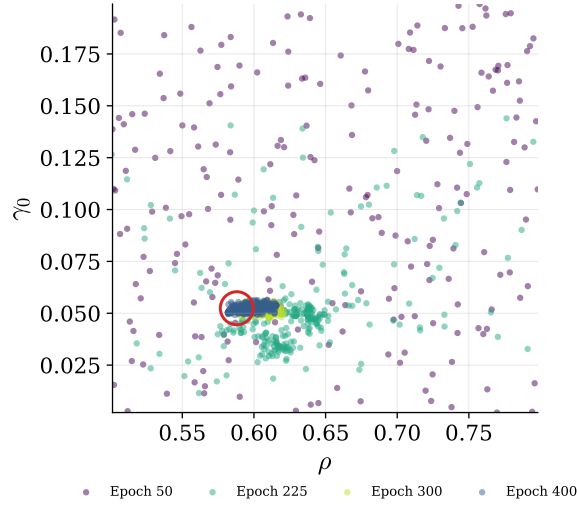
Each panel corresponds to one of 8 estimated parameters, described in Section 4. We draw 40 parameter vectors (the x-axis) and, for each, solve the model using value function iteration and simulate moments. We then estimate the model using our method (Section 3) targeting these moments, and report our parameter estimates on the y-axis. Bands denote 95% confidence intervals. The solid line is the 45-degree line

Figure 3: Adaptive Shrinkage



(a) Fixed adj. cost (γ_0) histogram

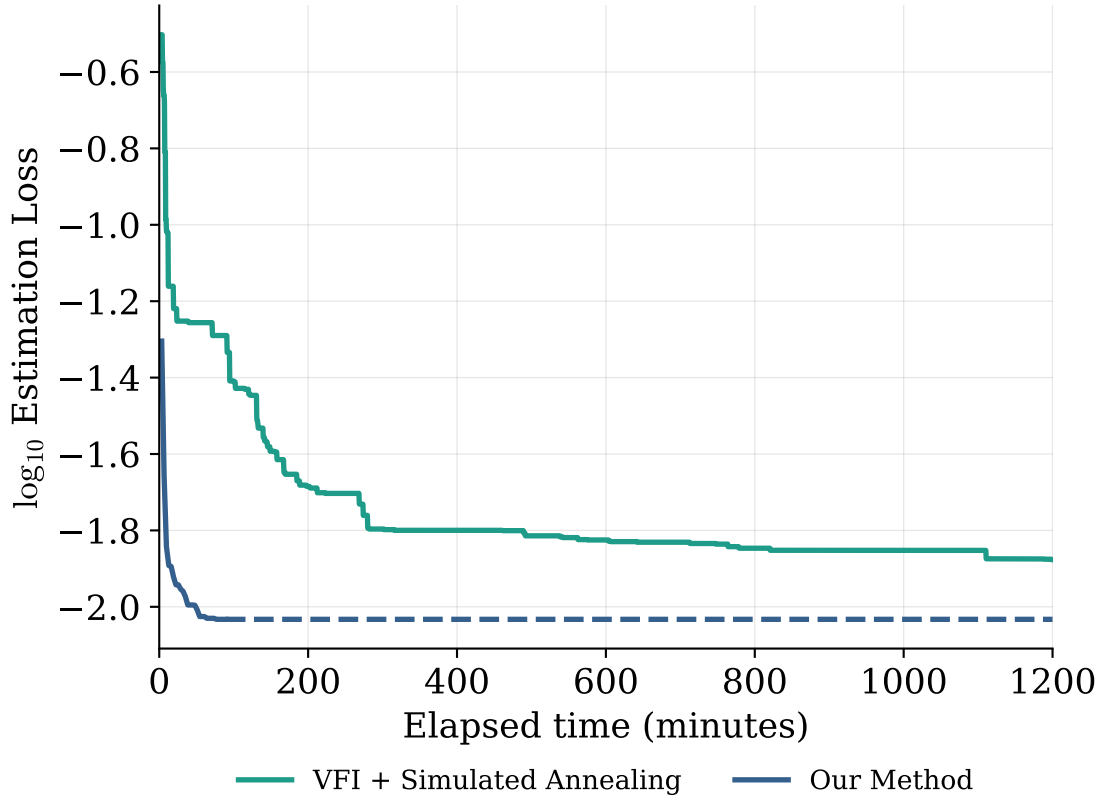
(b) Autocorr. of productivity (ρ) histogram



(c) Joint distribution of γ_0 and ρ

This figure illustrates the adaptive shrinkage algorithm, which narrows the parameter space during training (Section 3.5). Panels (a) and (b) show histograms of sampled parameter values for the fixed adjustment cost γ_0 and the autocorrelation of the productivity process, respectively, at four training epochs. The red vertical line denotes the true parameter value. Panel (c) shows the joint distribution of ρ and γ_0 at the same four epochs, with the red circle marking the true values.

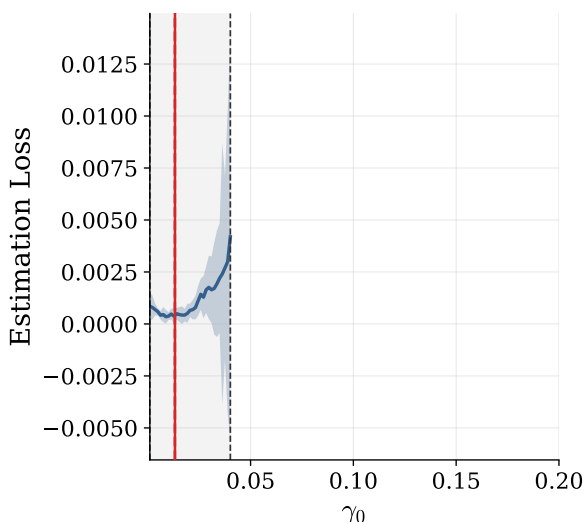
Figure 4: Time to Solution: Partial Equilibrium Model



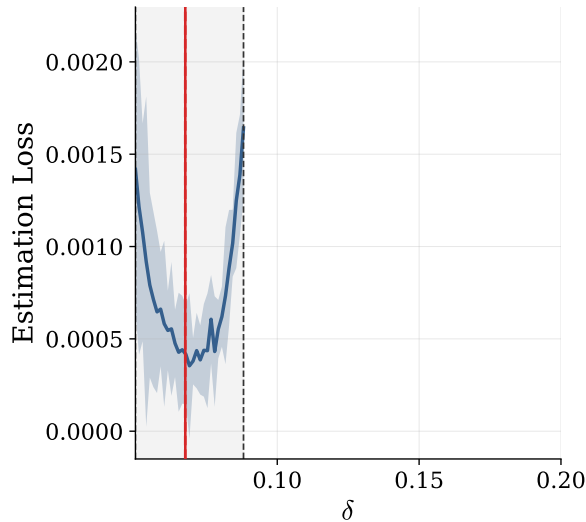
This figure compares the estimation loss over run time in minutes for our method (blue) and value function iteration combined with simulated annealing (green) for the partial equilibrium version of the model described in Section 2. Both methods target the same Compustat moments and use the same weighting matrix. We report the average of the running best loss across 4 seeds. We run our method for 90 minutes and simulated annealing for 20 hours, with our method’s line becoming dashed after the end of training.

Figure 5: Identification Diagnostic: Minimum Loss Functions with Model-Implied Targets

Examples of well identified parameters

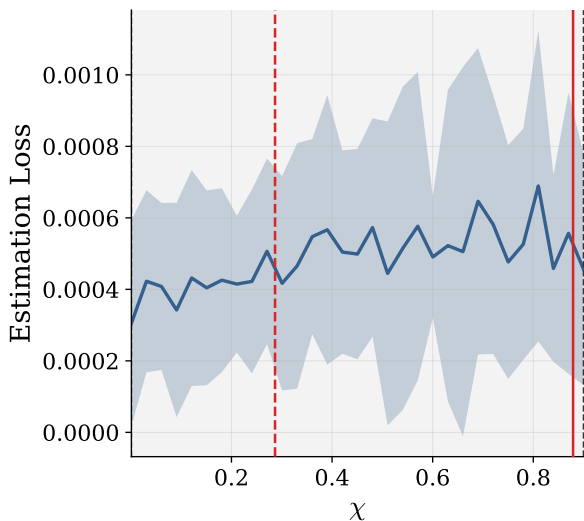


(a) Fixed adj. cost (γ_0)

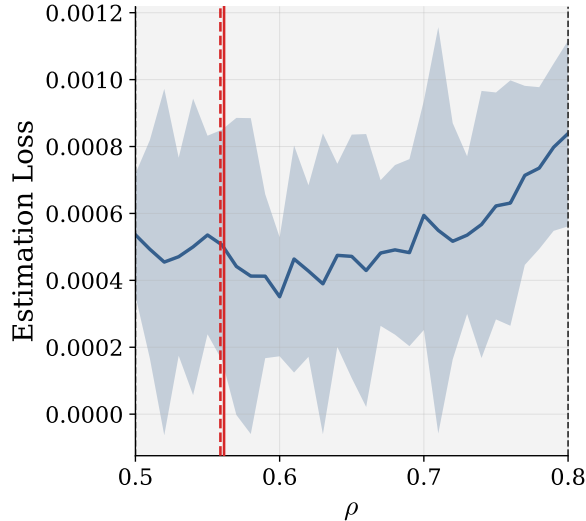


(b) Depreciation rate (δ)

Examples of weakly identified parameters



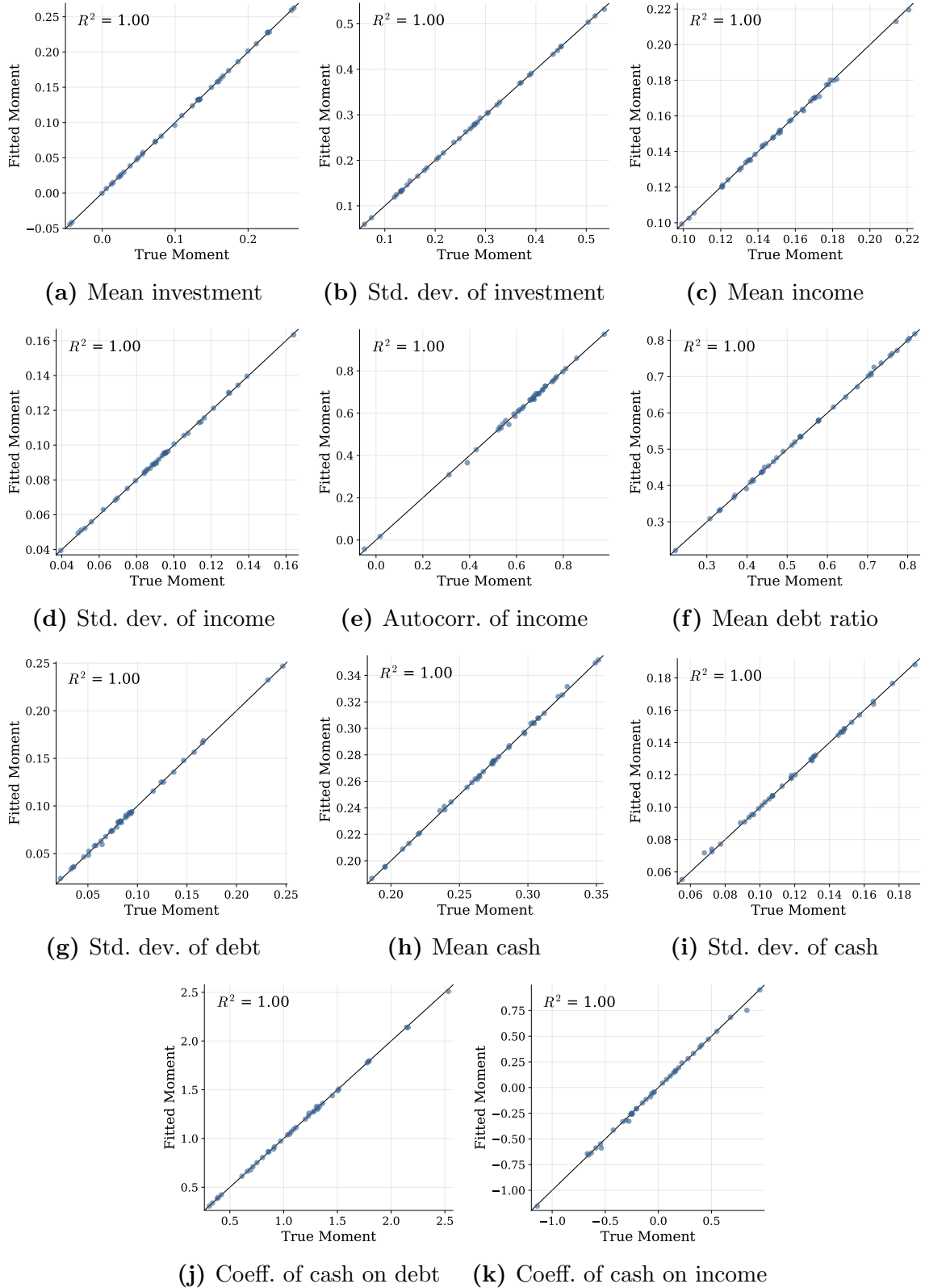
(c) Recovery rate (χ)



(d) Autocorr. of productivity (ρ)

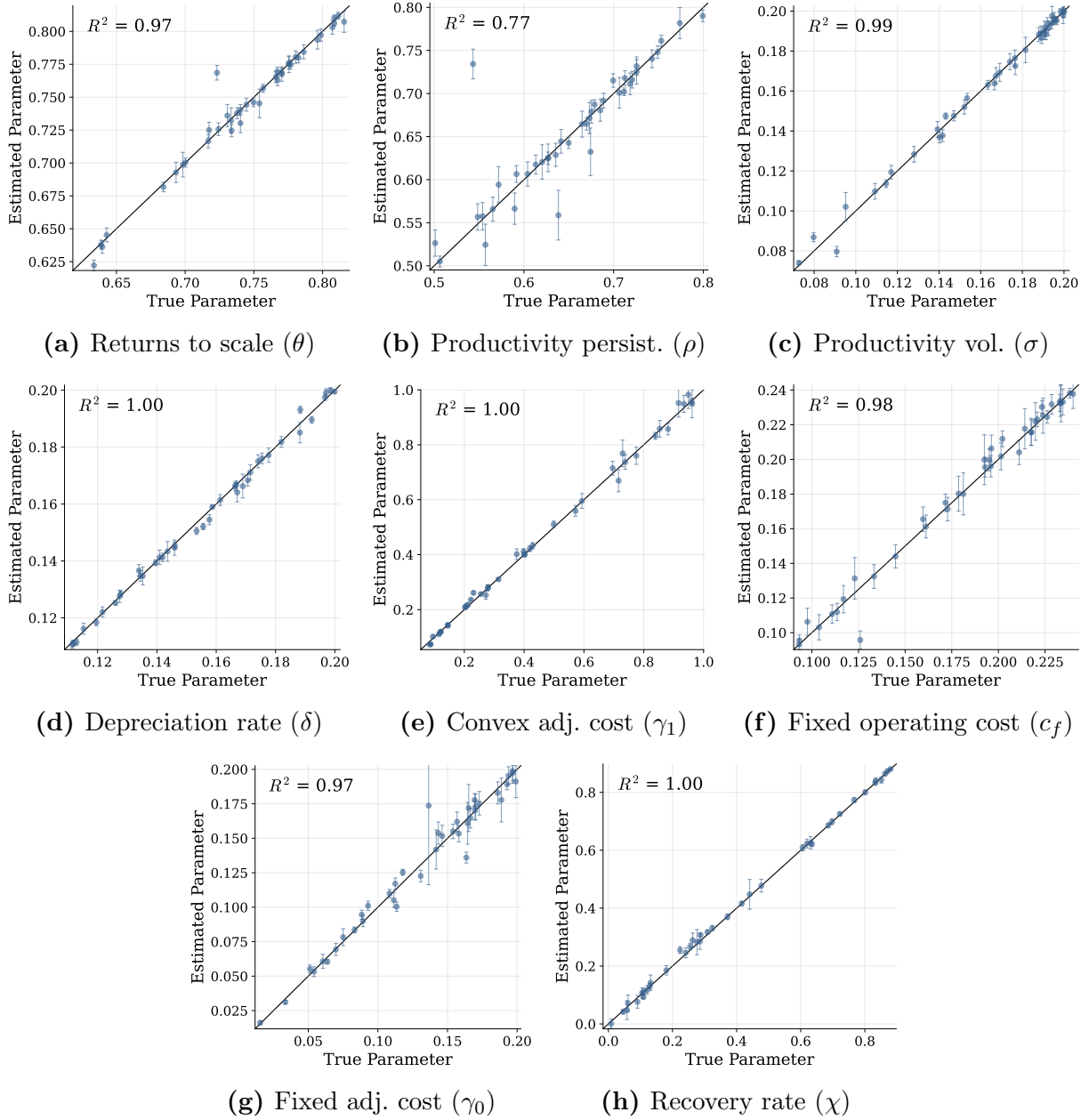
This figure shows the minimum loss function, described in Section 3.6, for four selected parameters: the fixed adjustment cost γ_0 , the depreciation rate δ , the recovery rate χ , and the autocorrelation of the productivity process ρ . In each panel, we fix the parameter on the x-axis and minimize the estimation loss function over all other parameters. The targeted moments are model-implied moments simulated at the partial equilibrium estimate $\hat{\beta}$ from Section 4.2, reported in Table 1. The solid line is the median across 10 folds, and the shaded band shows ± 2 standard deviations across the 10 folds. The red solid line denotes the parameter used to produce the model-implied moments used as targets and the red dashed line denotes the recovered parameter. The x-axis spans the initial parameter bounds, with the black dashed vertical lines and shaded region marking the narrowed bounds after adaptive shrinkage (Section 3.5).

Figure 6: True vs. Fitted Moments: General Equilibrium Model



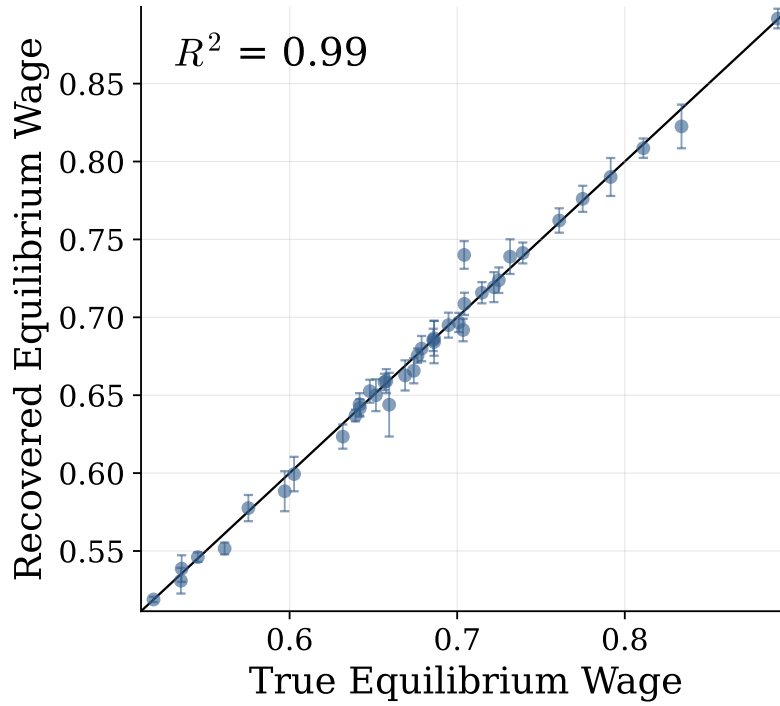
Each panel corresponds to one of 11 targeted data moments for the general equilibrium model described in Section 2.2. We draw 40 parameter vectors and, for each, solve the model using value function iteration to obtain target moments, which we report on the x-axis. We then estimate the model using our method (Section 3) targeting these moments, then simulate moments using our solution at the estimated parameters and report those on the y-axis. The solid line is the 45-degree line.

Figure 7: True vs. Estimated Parameters: General Equilibrium



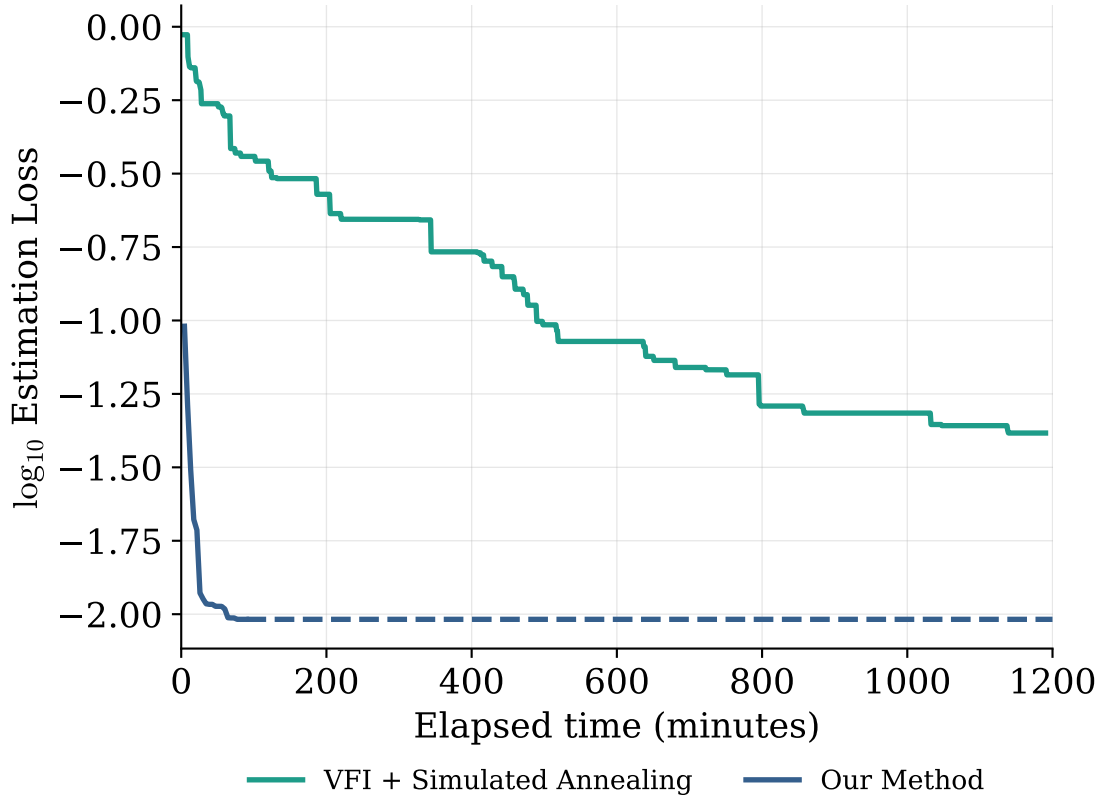
Each panel corresponds to one of 8 estimated parameters for the general equilibrium model described in Section 2.2. We draw 40 parameter vectors (the x-axis) and, for each, solve the model using value function iteration and simulate moments. We then estimate the model using our method (Section 3) targeting these moments, and report our parameter estimates on the y-axis. Bands denote 95% confidence intervals. The solid line is the 45-degree line.

Figure 8: True vs. Recovered Equilibrium Wage



This figure shows the equilibrium wage recovered by our method for the general equilibrium model described in Section 5. We draw 40 parameter vectors and, for each, solve the model using value function iteration, using a root-finding algorithm to solve for the market-clearing wage and reporting these equilibrium wage values on the x-axis. The y-axis reports the wage recovered using our new approach, which treats the wage as a pseudo parameter and the market-clearing imbalance as a moment condition during the estimation procedure, with a target imbalance of zero (Section 3.4). Bands denote 95% confidence intervals. The solid line is the 45-degree line.

Figure 9: Time to Solution: General Equilibrium Model



This figure compares the estimation loss over run time in minutes for our method (blue) and value function iteration combined with simulated annealing (green) for the general equilibrium version of the model described in Section 2.2. Both methods target the same Compustat moments and use the same weighting matrix. We report the average of the running best loss across 4 seeds. We run our method for 90 minutes and simulated annealing for 20 hours, with our method’s line becoming dashed after the end of training.

Table 1: Partial Equilibrium Model Estimation

<i>Panel A: Parameter estimates</i>							
θ	ρ	σ	δ	γ_0	γ_1	χ	c_f
0.796	0.597	0.187	0.066	0.014	0.945	0.003	0.028
(0.002)	(0.007)	(0.002)	(0.000)	(0.000)	(0.014)	(0.024)	(0.001)

<i>Panel B: Targeted moments</i>		
Moment	Data	Model
Mean investment rate	0.065	0.068
Standard deviation of investment rate	0.045	0.052
Mean operating income	0.095	0.159
Standard deviation of operating income	0.100	0.040
Serial correlation of operating income	0.495	0.505
Mean debt ratio	0.271	0.305
Standard deviation of debt ratio	0.148	0.111
Mean cash ratio	0.096	0.033
Standard deviation of cash ratio	0.080	0.028
Coefficient of cash on net debt	0.094	0.078
Coefficient of cash on income	0.039	0.123

This table reports parameters and moments from the estimation of the partial equilibrium model described in Section 2. Panel A reports parameter estimates with standard errors in parentheses. θ is returns to scale parameter, ρ is autocorrelation of the productivity process, σ is the standard deviation of the productivity process, δ is the depreciation rate, γ_0 is the fixed capital adjustment cost, γ_1 is the convex capital adjustment cost, χ is the default recovery rate, and c_f is the fixed operating cost. Panel B compares data moments from Compustat (1970–2019) with model-implied moments at the estimated parameters.

Table 2: General Equilibrium Model Estimation

Panel A: Parameter estimates

θ	ρ	σ	δ	γ_0	γ_1	χ	c_f	w
0.804	0.515	0.196	0.065	0.011	0.941	0.831	0.196	0.500
(0.002)	(0.007)	(0.001)	(0.000)	(0.000)	(0.018)	(0.044)	(0.004)	(0.002)

Panel B: Targeted moments

Moment	Data	Model
Mean investment rate	0.065	0.066
Standard deviation of investment rate	0.045	0.050
Mean operating income	0.095	0.153
Standard deviation of operating income	0.100	0.038
Serial correlation of operating income	0.495	0.431
Mean debt ratio	0.271	0.320
Standard deviation of debt ratio	0.148	0.132
Mean cash ratio	0.096	0.032
Standard deviation of cash ratio	0.080	0.024
Coefficient of cash on net debt	0.094	0.044
Coefficient of cash on income	0.039	0.101
Resource gap	0.000	-0.012

This table reports parameters and moments from the estimation of the general equilibrium model described in Section 2.2. Panel A reports parameter estimates with standard errors in parentheses. θ is returns to scale parameter, ρ is autocorrelation of the productivity process, σ is the standard deviation of the productivity process, δ is the depreciation rate, γ_0 is the fixed capital adjustment cost, γ_1 is the convex capital adjustment cost, χ is the default recovery rate, and c_f is the fixed operating cost. Panel B compares data moments from Compustat (1970–2019) with model-implied moments at the estimated parameters.

Online Appendix for

“AI for Structural Estimation”

Victor Duarte and Julia Fonseca

A Algorithm and implementation details

This appendix provides a full description of the algorithm summarized in Section 3 and provides practical guidance for implementation. Section A.1 provides background on neural networks and automatic differentiation, and fixes notation. Sections A.2 and A.3 describe the value and policy network architecture and training. Section A.4 explains the smooth approximation of non-differentiable payoffs, and Section A.5 describes the grid refinement step. Section A.6 details the simulation and moment computation. Section A.7 describes the moment function approximation, Section A.8 covers the estimation procedure, and Section A.9 explains the minimum loss functions and adaptive bound shrinkage. Section A.10 describes the modifications needed for general equilibrium estimation. Section A.11 describes the asynchronous execution across GPUs, and Section A.12 discusses hyperparameters and other implementation choices. Section A.13 described an AI agent that applies our method to new models from natural language prompts.

A.1 Background: neural networks and gradient-based optimization

A.1.1 Neural networks

A neural network is a parametric function built by composing simple operations called layers. Each layer applies a linear transformation followed by a nonlinear function. Given an input

vector $x \in \mathbb{R}^{d_{in}}$, a single layer computes

$$h = \phi(Wx + c), \tag{23}$$

where $W \in \mathbb{R}^{d_{out} \times d_{in}}$ is a matrix of weights, $c \in \mathbb{R}^{d_{out}}$ is a vector of biases, and ϕ is a nonlinear function applied element-wise. The linear transformation $Wx + c$ is a rotation, stretching, and shifting of the input. The nonlinear function ϕ , called the activation function, introduces the curvature that allows the network to represent nonlinear relationships.

A network with L hidden layers composes L such operations in sequence, with the output of layer l becoming the input of layer $l + 1$. The final layer is a linear transformation without an activation function, producing the network’s scalar or vector output. We use the SiLU activation $\phi(x) = x \cdot \sigma(x)$ throughout, where $\sigma(x) = 1/(1 + e^{-x})$ is the logistic sigmoid, because SiLU is smooth and has gradients everywhere. We discuss the choice of activation function in Section A.12.

The adjustable quantities in the network—all weight matrices W_1, \dots, W_L and bias vectors c_1, \dots, c_L —are collectively called the network’s “weights” or “parameters.” To avoid confusion with model parameters $(\theta, \rho, \sigma, \delta, \gamma_1, \gamma_0, \chi, c_f)$, we refer to the neural network’s adjustable quantities as weights throughout. The key property that makes neural networks useful is universal approximation: with enough layers and units per layer, they can approximate any continuous function to arbitrary accuracy (Hornik et al., 1989). In our application, we use them to approximate value functions, policy functions, and the mapping from model parameters to moments.

A.1.2 Training neural networks

Training a neural network means adjusting the network’s weights to minimize a loss function that measures the network’s error on a specific task. For example, if the task is to approximate the value function, the loss function measures the extent to which the current network

violates the Bellman equation.

Training proceeds iteratively. At each step, we:

1. Draw a random subset of inputs (a “mini-batch”) from the training distribution.
2. Evaluate the loss function on the mini-batch.
3. Compute the gradient of the loss with respect to every weight in the network. This gradient tells us which direction to adjust each weight to reduce the loss.
4. Update each weight by taking a small step in the direction of steepest descent, given by the negative of the gradient of the loss. The size of the step is controlled by the learning rate (Section A.12.4).

This procedure is called stochastic gradient descent (SGD). Each step adjusts thousands of weights simultaneously, and after many steps—typically thousands to millions—the network converges to a good approximation of the target function. We use the Adam variant of SGD, which adapts the step size for each weight based on the history of past gradients, improving convergence speed (Kingma and Ba, 2015). We discuss the choice of optimizer in Section A.12.

A.1.3 Automatic differentiation

The gradient computation in step 3 above is performed by automatic differentiation. In traditional numerical methods, computing derivatives requires either symbolic calculus, which is impractical for complex functions, or finite differences, which requires perturbing each input and re-evaluating. With automatic differentiation, the software records every arithmetic operation performed during the evaluation of the loss function and then applies the chain rule to compute exact derivatives with respect to all weights in a single backward pass.

The cost of this backward pass is roughly the same as the cost of evaluating the function itself. This means that computing the gradient of a loss function with respect to thousands

of network weights costs about twice as much as simply evaluating the loss. This is a dramatic improvement over finite differences, which would require one evaluation per weight. Automatic differentiation is what makes gradient-based training of neural networks practical, and it is also what allows us to compute analytical gradients of the estimation objective in Section 3.3.

A.2 Value and policy networks

In standard value function iteration, the value function $V(z, k, b)$ is stored as a three-dimensional array on a discrete grid. In our approach, V is represented by a neural network that takes $(\log z, k, b)$ as inputs and returns a scalar value. Similarly, the three policy functions—investment $i(z, k, b)$, next-period gross debt $b'(z, k, b)$, and next-period cash $c'(z, k, b)$ —are each represented by a separate neural network. As described in Section 3 and in Section A.2.2 below, these networks also take as inputs the model parameters.

A.2.1 Input normalization

Neural networks work best when their inputs are on a common scale. We normalize each input to the interval $[-1, 1]$ using a linear transformation:

$$\tilde{x} = \frac{2(x - x_{\min})}{x_{\max} - x_{\min}} - 1, \quad (24)$$

where $[x_{\min}, x_{\max}]$ are the bounds for that variable.

Initial bounds for state variables. The bounds for state variables depend on the model parameters. We initialize training with the following bounds:

- *Log productivity:* $\log z \in \left[-m \cdot \frac{\sigma}{\sqrt{1-\rho^2}}, +m \cdot \frac{\sigma}{\sqrt{1-\rho^2}} \right]$, where $m = 2.5$. These are the unconditional ± 2.5 standard deviation bounds from the Tauchen approximation of the AR(1) process in Eq. (2). They depend on two parameters, σ and ρ .

- *Capital*: The bounds are derived from the steady-state capital stock at the extremes of productivity. Let $k_{ss}(z)$ denote the capital level at which the marginal product of capital equals $\delta + r_f$, evaluated at z_{\min} and z_{\max} . The capital bounds are $k \in [k_{ss}(z_{\min}), k_{ss}(z_{\max})]$. These depend on $\delta, r_f, \theta, \alpha, \rho$, and σ .
- *Net debt*: The bounds are $b \in [-1, 2]$, where negative values represent net cash positions and positive values represent net debt.

Because some of the state bounds depend on the parameters, the same raw state value maps to different normalized inputs for different parameter vectors. This is appropriate since the economically relevant range of states varies with the parameters.

Initial bounds for parameters. The model parameters are normalized to $[-1, 1]$ based on their respective bounds, using the same linear transformation (24). We initialize the algorithm with the following parameter bounds:

Appendix Table A1: Initial parameter bounds

Parameter	Lower bound	Upper bound
Returns to scale (θ)	0.60	0.82
Autocorrelation of productivity (ρ)	0.50	0.80
Standard deviation of productivity (σ)	0.05	0.20
Depreciation (δ)	0.05	0.20
Convex adjustment cost (γ_1)	0.05	1.00
Fixed operating cost (c_f)	0.001	0.25
Fixed adjustment cost (γ_0)	0.001	0.20
Recovery rate (χ)	0.001	0.90

This table reports parameter bounds at the start of training. These bounds narrow during training via the adaptive shrinkage procedure described in Section A.9.

These bounds narrow over time as the adaptive shrinkage procedure described in Section A.9 concentrates the parameter space around the estimated values. The normalization is recomputed whenever the bounds change, so the network always receives inputs in $[-1, 1]$.

A.2.2 *Conditioning on parameters*

The value and policy networks take both state variables $(\log z, k, b)$ and model parameters $(\theta, \rho, \sigma, \delta, \gamma_1, \gamma_0, \chi, c_f)$ as inputs. A simple approach would concatenate all eleven normalized inputs into a single vector and feed it into the network, as in Duarte, Duarte, and Silva (2024). However, this treats state variables and parameters symmetrically, when these inputs have different economic roles. The parameters define the economic environment—for instance, how persistent productivity is or how quickly capital depreciates—while the states describe the firm’s position within that environment. For instance, how persistent productivity is (which is governed by ρ) could change how sensitive the value function should be to the current productivity level, given that it changes how informative current productivity is about future productivity.

We reflect this in our network architecture by allowing parameters to modulate how the network processes states. The state variables enter the main body of the network (the “trunk”), and the parameters control how each hidden layer in the trunk processes information from state variables. Specifically, for each hidden layer l , a small auxiliary network maps the normalized parameter vector to a pair of vectors (γ_l, δ_l) —a scale and a shift—each of the same dimension as the hidden layer. The hidden layer output is:

$$h_{l+1} = \phi(\gamma_l \odot (W_l h_l + c_l) + \delta_l), \quad (25)$$

where \odot denotes element-wise multiplication, W_l and c_l are the weight matrix and bias of layer l , and ϕ is the activation function. The intuition is that γ_l controls the gain of each unit in the layer (amplifying or dampening signals) and δ_l controls the offset (shifting activation

thresholds). For instance, when the auxiliary network receives a high value of the persistence of productivity shocks ρ , it can learn to set higher gains on the units that process productivity information, amplifying the sensitivity of the value function to the current productivity z . Changing the parameters thus changes how the layer responds to states, effectively giving a different value function for each parameter vector, while sharing the trunk architecture.

This mechanism is called Feature-wise Linear Modulation (FiLM; Perez et al., 2018). Each auxiliary network has one hidden layer with 32 units and takes the normalized parameter vector as input. It outputs a vector of length $2 \times d_{hidden}$, where d_{hidden} is the number of units per hidden layer, which is split into γ_l and δ_l . Each hidden layer in the trunk has its own auxiliary network, so the conditioning can differ across layers. As we discuss in Section A.12.1, FiLM allows us to use a smaller network architecture.

A.2.3 Network architecture

The value network has three hidden layers with 128 units each, a choice that we discuss in Section A.12.1. The policy network consists of three separate sub-networks, one for each control (i, b', c') , with the same architecture as the value network. Each sub-network outputs a single scalar, which is mapped to the feasible range using a sigmoid transformation:

$$a = a_{\min} + (a_{\max} - a_{\min}) \cdot \sigma(r), \tag{26}$$

where r is the raw network output, σ is the logistic sigmoid, and $[a_{\min}, a_{\max}]$ are the feasibility bounds for that control at the current state and parameter values. Because the sigmoid maps any real number into $(0, 1)$, the output a always lies strictly within $[a_{\min}, a_{\max}]$.

For the gross debt control b' , we subtract 4 from the raw output before applying the sigmoid: $b' = b'_{\min} + (b'_{\max} - b'_{\min}) \cdot \sigma(r - 4)$. This biases the initial policy toward low debt levels since $\sigma(-4) \approx 0.018$, which improves training stability in the early stages when the bond pricing function is not yet well learned.

A.3 Training the value and policy networks

Training proceeds by policy iteration, alternating between policy evaluation and policy improvement. Each training epoch consists of 500 gradient steps (see Section A.12). At each step, we perform the following:

A.3.1 Step 1: Sampling states and parameters

We draw a mini-batch of $N = 8,192$ state-parameter combinations. Parameter vectors $(\theta, \rho, \sigma, \delta, \gamma_1, \gamma_0, \chi, c_f)$ are drawn uniformly from the current parameter bounds. For each parameter vector, we compute the parameter-dependent state bounds, as described in Section A.2, and draw $(\log z, k, b)$ uniformly within those bounds.

A.3.2 Step 2: Policy evaluation

At each sampled point, we evaluate the Bellman residual, meaning the difference between the left-hand side and the right-hand side of the Bellman equation:

$$R_i = V(z_i, k_i, b_i) - D_i - \beta \mathbb{E}_{z'|z_i}[\max\{V(z', k'_i, b'_i - c'_i), 0\}], \quad (27)$$

where D_i is the dividend at the current state under the current policy, and $k'_i = (1 + i_i - \delta)k_i$ is next-period capital.

We update the value network weights to reduce the mean squared residual:

$$L_V = \frac{1}{N} \sum_{i=1}^N R_i^2. \quad (28)$$

The gradient of L_V with respect to the value network weights is computed by automatic differentiation and flows through both the $V(z_i, k_i, b_i)$ term on the left and the $\mathbb{E}[\max\{V(\cdot), 0\}]$ term on the right. We take one step of the Adam optimizer with learning rate 10^{-3} , a choice that we discuss in Section A.12.

A.3.3 Step 3: Policy improvement

Using the updated value function, we adjust the policy network weights to choose better controls at each sampled point. The loss is the negative of the Bellman right-hand side:

$$L_\pi = -\frac{1}{N} \sum_{i=1}^N (D_i + \beta \mathbb{E}_{z'|z_i}[\max\{V(z', k'_i, b'_i - c'_i), 0\}]). \quad (29)$$

Minimizing L_π means maximizing expected lifetime value. The gradient flows through the policy networks—which determine i_i , b'_i , and c'_i , and hence D_i and the next-period states—but treats the value network weights as fixed. We take one step of Adam with learning rate 10^{-3} (Section A.12).

After steps 2 and 3, we return to step 1 with a fresh mini-batch and repeat. One epoch consists of 500 such cycles. We discuss our choice of the number of cycles in an epoch in Section A.12.

A.3.4 The bond pricing circularity

In the model of Section 2, the value function V and the bond price q are jointly determined. V depends on q through dividends (since $d_1 = q \cdot b' \cdot k' - b \cdot k - c_f$), and q depends on V through the default probability (since default occurs when $V < 0$). We handle this circularity by computing the bond price using a “target network”—a network whose weights are held fixed for all gradient steps within that epoch. At the start of the next epoch, the target network is updated to the current value network weights, and again held fixed for all steps within that epoch. This means the bond price changes slowly relative to the value function, giving the value function time to adjust before the bond price is recomputed. The target network is used only for bond pricing, with the continuation value $\mathbb{E}[\max\{V(\cdot), 0\}]$ in the Bellman residual using the current value network weights.

During training, we compute the default probability in the bond pricing equation using a

quadratic approximation in the productivity shock ε around $\varepsilon = 0$, using the first and second derivatives obtained by automatic differentiation. We then compute the default probability $\Pr(V' < 0)$ analytically from the roots of this quadratic and the standard normal CDF, without requiring an inner quadrature loop. We use this approximation only during neural network training. The grid refinement step described in Section A.5 computes bond prices exactly on a discrete grid, increasing the accuracy of this approximation.

A.3.5 Computing expectations with Gauss-Hermite quadrature

The conditional expectation $\mathbb{E}_{z'|z}[f(z')]$ for the AR(1) process $\log z' = \rho \log z + \sigma\varepsilon$, $\varepsilon \sim \mathcal{N}(0, 1)$, can be written as an integral against the standard normal density:

$$\mathbb{E}_{z'|z}[f(z')] = \int_{-\infty}^{\infty} f(e^{\rho \log z + \sigma\varepsilon}) \frac{e^{-\varepsilon^2/2}}{\sqrt{2\pi}} d\varepsilon.$$

Gauss-Hermite quadrature provides a set of Q nodes $\{x_q\}$ and weights $\{w_q\}$ that approximate integrals against the weight function e^{-x^2} . The standard Gauss-Hermite nodes are defined for this weight function, so we substitute $\varepsilon = \sqrt{2}x$ to match the standard normal density. The resulting approximation is:

$$\mathbb{E}_{z'|z}[f(z')] \approx \frac{1}{\sqrt{\pi}} \sum_{q=1}^Q w_q f\left(e^{\rho \log z + \sigma\sqrt{2}x_q}\right). \quad (30)$$

With $Q = 5$ nodes, this is exact for polynomial integrands up to degree 9, which is accurate enough for the value functions in our application, given the refinement step. We discuss our choice of Q in Section A.12.

A.4 Smooth approximation of non-differentiable payoffs

The dividend function D includes indicator functions: $\mathbf{1}\{d_1 < 0\}$ for equity issuance in the first subperiod, $\mathbf{1}\{I > 0\}$ for fixed adjustment costs, and $\mathbf{1}\{d_2 < 0\}$ for equity issuance in

the second subperiod. The gradient of an indicator function is zero everywhere except at the threshold, where it is undefined. This means that when the network produces a dividend that is, say, slightly above the equity issuance threshold, the gradient provides no signal about how to adjust. The network cannot learn that a small change in the control would trigger or avoid issuance costs.

We solve this by replacing each indicator with a smooth approximation during training:

$$\mathbf{1}\{x < 0\} \approx \sigma(-x/\tau), \tag{31}$$

$$\max(0, x) \approx \tau \log(1 + e^{x/\tau}), \tag{32}$$

$$\min(0, x) \approx -\tau \log(1 + e^{-x/\tau}), \tag{33}$$

where σ is the logistic sigmoid and $\tau > 0$ is a temperature parameter. As $\tau \rightarrow 0$, all three approximations converge pointwise to the exact functions. The smooth indicator $\sigma(-x/\tau)$ transitions gradually from 1 to 0 over a region of width approximately 4τ centered at $x = 0$, so the gradient is nonzero near the threshold, allowing the network to learn.

We set the temperature to $\tau = 10^{-3}$. A smaller τ produces a tighter approximation to the exact piecewise function and thus a more accurate approximation, but the gradient signal can vanish for points that are not very close to the kink. A larger τ spreads the gradient signal over a wider region, making training easier, but at the cost of a less accurate approximation during training. Because the grid refinement step (Section A.5 below) uses the exact, non-smooth dividend function, the accuracy of the final solution is less sensitive to the choice of temperature.

A.5 *Grid refinement*

Before simulating moments for each sampled parameter vector, we refine the neural network solution on a discrete grid. This removes the dependence on the smooth approximations

used during training, described in Section A.4 above, and makes the solution as accurate as value function iteration. As we discuss in Section A.12, we do not recommend this step when the objective function is smooth, as neural networks generally provide a more accurate approximation than a grid in those circumstances.

The refinement procedure for a given parameter vector is:

1. *Evaluate the neural networks on the grid.* Compute $V(z_i, k_j, b_l)$ and the policies $(i, b', c')(z_i, k_j, b_l)$ at every point on the discrete grid. This gives a grid-based value function and policy initialized at the neural network’s output.
2. *Run policy iteration.* Each step consists of:
 - (a) *Policy improvement.* For each state grid point (z_i, k_j, b_l) , we search for the control combination (k', b', c') that maximizes $D + \beta \mathbb{E}_{z'|z}[\max\{V', 0\}]$ (the right-hand-side of Bellman Eq. (10)), using the exact, non-smoothed dividend function. Rather than searching over the entire control grid, we construct a candidate set from the current policy at the grid point, the policies at its immediate neighbors in each dimension of the (z, k, b) grid, and small perturbations (± 1 grid index) around the current policy. This produces 9 candidates per control dimension, or $9^3 = 729$ combinations to evaluate at each grid point. Because the neural network provides an excellent starting point, this local search is sufficient to find the optimal policy.
 - (b) *Policy evaluation.* Given the improved policy, solve for the value function that satisfies the Bellman equation under that fixed policy. Because of the $\max\{V, 0\}$ operator, this is a nonlinear fixed-point problem. We solve it using a semismooth Newton method. At each iteration, we set $\max\{V, 0\} = V$ for grid points where $V > 0$ and $\max\{V, 0\} = 0$ for grid points where $V \leq 0$. With this partition fixed, the Bellman equation becomes linear in V and can be solved using the generalized minimal residual method (GMRES), a standard iterative linear solver. We then

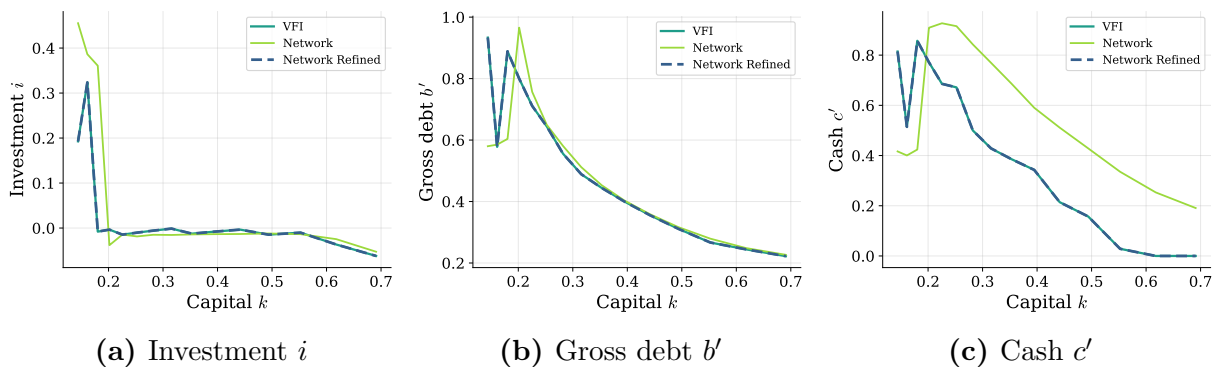
recompute the partition based on the new solution and repeat until it stabilizes.

3. *Recompute bond prices.* The bond pricing Eq. (6) depends on the value function through the default probability. After updating V , we recompute $q(z, k', c', b')$ on the full grid.
4. *Repeat.* Go back to step 2 with the updated bond prices.

We perform six rounds of steps 2–4. We use the same number of grid points as Gao, Whited, and Zhang (2021), with larger grids for controls than for states: 11 for productivity, 15 for capital, 35 for net debt, 81 for next-period capital, 91 for gross debt, and 71 for cash.

Figure A1 illustrates this refinement step. Each panel plots a policy function with respect to capital, with productivity fixed to the median and net debt near zero. In each panel, the green solid line shows the VFI solution, while the yellow-green solid line shows the neural network solution before the refinement step. The neural networks representing the policy functions get close to the solution, but differences in accuracy remain. As we discuss in Section A.12.1 below, we choose a smaller network architecture because of this refinement step, but we could obtain a more accurate network solution by increasing the number of units. We choose not to because our smaller network is enough to give us a very good starting point. After 6 rounds of the grid-optimization procedure described above, we arrive at the blue dashed line, which is indistinguishable from the VFI solution.

Appendix Figure A1: Refinement Step Illustration



Each panel shows a one-dimensional slice of a policy function at $z = 1$ (median productivity) and $b = -0.03$ (approximately zero net debt), as a function of capital k . The green solid line is the value function iteration (VFI) solution. The yellow-green solid line is the neural network solution before grid refinement. The blue dashed line is the network solution after grid refinement (Section 3.2.1).

A.6 Simulation and moment computation

For each parameter vector, we simulate a panel of $N_f = 5,000$ firms over $T = 300$ time periods, discarding the first $T_0 = 200$ periods as burn-in to approximate the ergodic distribution. We discuss these choices in Section A.12.

A.6.1 Panel simulation

Productivity z evolves on a discrete Markov chain constructed via the Tauchen approximation of the AR(1) process (2). At each period, the firm's state is (z, k, b) , where z takes one of 11 discrete values and (k, b) are continuous.

The firm's controls are determined by the refined grid-based policy functions. Because z is discrete, no interpolation is needed in the productivity dimension: we simply look up the policy at the current z grid point. For (k, b) , which are continuous and generally do not fall exactly on grid points, we use bilinear interpolation: we find the four nearest grid points in the (k, b) plane and take a weighted average of their policy values, with weights proportional to the proximity to each grid point.

Capital accumulates as $k' = (1 + i - \delta)k$, and the next-period net debt state is $b^{gross'} - c'$, where $(i, b^{gross'}, c')$ are the interpolated policy outputs.

Default and reseeding. When a firm's continuation value $V(z, k, b)$ falls below zero, the firm defaults. We replace defaulting firms by drawing new values of (k, b) from random grid points, keeping the current productivity state z , which is exogenous and does not reset upon default. This ensures that the simulated panel always has N_f active firms and that the cross-sectional distribution reflects the model's ergodic distribution after burn-in.

A.6.2 Moment construction

After discarding the burn-in, we construct the simulated panel as a sequence of consecutive pairs. For each pair of adjacent periods, we denote current-period variables without primes and next-period variables with primes. We denote level variables with uppercase letters, with the lowercase counterparts denoting ratios to capital, as defined in Section 2.¹ We then define the following variables:

$$\text{Investment rate: } i = (K' - (1 - \delta)K)/K, \quad (34)$$

$$\text{Operating income: } inc = (zA_\pi K^\xi - c_f)/(K + C), \quad (35)$$

$$\text{Debt ratio: } d = B^{gross'}/(K' + C'), \quad (36)$$

$$\text{Cash ratio: } cash = C'/(K' + C'), \quad (37)$$

$$\text{Cash saving: } \Delta c = (C' - C)/(K' + C'), \quad (38)$$

$$\text{Net debt: } net = B/(K + C). \quad (39)$$

All moments are computed over observations in which the firm is not in default in either period (i.e., $V > 0$ and $V' > 0$). The 11 targeted moments are:

¹Specifically, $C = c \cdot k$, $B = b \cdot k$, $C' = c' \cdot k'$, and $B' = b' \cdot k'$.

1. Mean investment rate.
2. Standard deviation of investment rate.
3. Mean operating income.
4. Standard deviation of operating income.
5. Autocorrelation of operating income, computed as the OLS slope from regressing next-period inc' on current-period inc , both centered by their sample means over good observations.
6. Mean debt ratio.
7. Standard deviation of debt ratio.
8. Mean cash ratio.
9. Standard deviation of cash ratio.
10. OLS slope on net debt in the cash-saving regression, a multivariate OLS regression of cash saving Δc on net debt net and operating income inc , with all three variables centered by their respective sample means.
11. OLS slope on operating income in the cash-saving regression.

A.7 Moment function approximation

The simulation step described in Section A.6 above produces a dataset of parameter-moment pairs, with 11 simulated moments for each sampled parameter vector. We use this dataset to train neural networks that learn the mapping from parameters to moments directly, so that evaluating moments at a new parameter vector takes milliseconds instead of the minutes required for a full model solution and simulation. We refer to the moment function approximations as moment networks or moment-surrogate networks.

A.7.1 *Moment network architecture*

We train one network per moment, for a total of 11 networks. Each is a feedforward network with three hidden layers of 32 units each, using the SiLU activation function. The input is the eight-dimensional normalized parameter vector, and the output is a scalar (the predicted moment value). These networks are much smaller than the value and policy networks because the mapping from parameters to moments is lower-dimensional and smoother than the value function. We discuss this choice of architecture in Section A.12.

A.7.2 *Cross-validation*

To prevent overfitting and quantify approximation uncertainty, we use K -fold cross-validation. We partition the dataset into $K = 10$ roughly equal subsets, or “folds.” For each moment and each fold, we train a separate network using the other $K - 1$ folds and hold out the remaining fold for validation. This produces $M \times K = 11 \times 10 = 110$ networks in total.

There are two practical benefits of this approach. First, the held-out fold provides an out-of-sample measure of prediction accuracy (R^2) for each moment, which we monitor during training. Second, the K networks per moment produce K different parameter estimates in the estimation step, and the variation across these estimates measures how sensitive the results are to the specific training data.

A.7.3 *Training*

All 110 moment networks are trained simultaneously. Training uses the Adam optimizer with learning rate 10^{-4} and batch size 256, for 200 passes of stochastic gradient descent (SGD) over the accumulated dataset. We discuss these hyperparameter and optimizer choices in Section A.12. The training dataset is capped at 10,000 observations. If the accumulated dataset is larger, we use the 10,000 most recent observations. The loss function for moment

j and fold k is the mean squared prediction error:

$$L_{j,k} = \frac{1}{|\mathcal{T}_k|} \sum_{i \in \mathcal{T}_k} (m_{i,j} - g_j(\boldsymbol{\beta}_i))^2, \quad (40)$$

where \mathcal{T}_k is the training set for fold k , $m_{i,j}$ is the simulated j -th moment for parameter vector $\boldsymbol{\beta}_i$, and $g_j(\cdot)$ is the j -th moment-surrogate network.

A.8 Estimation

A.8.1 The optimization problem

For each cross-validation fold k , we minimize the weighted distance between data moments \hat{m} and the predictions of fold k 's moment-surrogate networks:

$$\hat{\boldsymbol{\beta}}_k = \arg \min_{\boldsymbol{\beta}} (\hat{m} - m_k(\boldsymbol{\beta}))' W (\hat{m} - m_k(\boldsymbol{\beta})), \quad (41)$$

where $m_k(\boldsymbol{\beta})$ stacks the 11 moment predictions from fold k 's networks and W is the weighting matrix.

A.8.2 Levenberg-Marquardt algorithm

We solve Eq. (41) using the Levenberg-Marquardt (LM) algorithm, a standard method for nonlinear least-squares problems. LM maintains a parameter vector and iteratively updates it using the Jacobian matrix $J = \partial m_k / \partial \boldsymbol{\beta}$, the 11×8 matrix of partial derivatives of moments with respect to parameters. At each iteration, LM computes a direction in which to adjust the parameters by solving a linear system that combines information from the Jacobian with a regularization term that controls the step size. When the current point is far from the solution, the regularization makes LM behave like gradient descent, taking small, safe steps. When the current point is close, the regularization shrinks, and LM behaves like the Gauss-Newton method, with faster convergence.

The Jacobian J is computed exactly by automatic differentiation through the moment-surrogate networks. This is an important advantage over standard SMM. Because the networks are differentiable, the optimizer has access to analytical gradients and can converge in a few iterations rather than performing hundreds of function evaluations. We set convergence tolerances to 10^{-10} for the function value, step size, and gradient norm, with a maximum of 20 iterations per run.

A.8.3 Multiple restarts

To avoid local minima, we run LM from 30 random starting points and select the solution with the lowest objective value. Each starting point is drawn from a standard normal distribution in the unconstrained parameter space, as described below. With $K = 10$ folds and 30 restarts per fold, the estimation step runs 300 LM optimizations. Because evaluating and differentiating the moment-surrogate networks is nearly instantaneous, the entire estimation step takes seconds.

A.8.4 Enforcing parameter bounds

Each parameter β^j must lie within its bounds $[\underline{\beta}^j, \bar{\beta}^j]$. We enforce this by reparametrizing:

$$\beta^j = \underline{\beta}^j + (\bar{\beta}^j - \underline{\beta}^j) \cdot \sigma(x^j), \quad (42)$$

where $x^j \in \mathbb{R}$ and σ is the logistic sigmoid. The optimizer searches over $x = (x^1, \dots, x^8) \in \mathbb{R}^8$, and the sigmoid guarantees that $\beta^j \in (\underline{\beta}^j, \bar{\beta}^j)$. The Jacobian of the moment function with respect to x is obtained by the chain rule: $\partial m / \partial x = (\partial m / \partial \beta) \cdot (\partial \beta / \partial x)$, where $\partial \beta^j / \partial x^j = (\bar{\beta}^j - \underline{\beta}^j) \cdot \sigma(x^j)(1 - \sigma(x^j))$.

A.8.5 Weighting matrix

We construct the weighting matrix W as the inverse of the moment variance-covariance matrix, estimated from the data using influence functions following Erickson and Whited

(2002). For each observation i and each moment j , the influence function ψ_{ij} captures the contribution of observation i to the sampling variability of moment j . The influence functions for our 11 moments are:

- *Means*: $\psi_{ij} = x_{ij} - \hat{\mu}_j$.
- *Standard deviations*: $\psi_{ij} = [(x_{ij} - \hat{\mu}_j)^2 - \hat{\sigma}_j^2] / (2\hat{\sigma}_j)$.
- *Autocorrelation*: $\psi_i = \tilde{x}_i \tilde{\varepsilon}_i / \hat{Q}$, where \tilde{x}_i is the centered lagged profitability, $\tilde{\varepsilon}_i = \tilde{y}_i - \hat{\rho} \tilde{x}_i$ is the OLS residual, and $\hat{Q} = \frac{1}{N_p} \sum \tilde{x}_i^2$.
- *Regression coefficients*: $\psi_i = \left(\frac{1}{N_{reg}} X'X \right)^{-1} x_i \varepsilon_i$, where x_i is the 2×1 vector of centered regressors and ε_i is the OLS residual.

Observations that do not contribute to a moment (e.g., observations without valid lags for ρ_{inc}) receive $\psi_{ij} = 0$.

We stack the influence functions into an $N \times 11$ matrix Ψ . To account for within-firm serial correlation, we sum the influence functions across all observations belonging to each firm g , producing a $G \times 11$ matrix of firm-level totals, where G is the number of firms. The cluster-robust covariance matrix is

$$\hat{\Sigma}_{jl} = \frac{1}{N_j N_l} \sum_{g=1}^G \left(\sum_{i \in g} \psi_{ij} \right) \left(\sum_{i \in g} \psi_{il} \right), \quad (43)$$

where N_j and N_l are the effective sample sizes for moments j and l , which may differ across moments because some moments (e.g., the autocorrelation and regression slopes) require lagged observations and therefore use fewer data points. The weighting matrix is $W = \hat{\Sigma}^{-1}$.

In practice, we work with the Cholesky factor $W^{1/2}$ such that $W = (W^{1/2})'W^{1/2}$, and normalize it by its median column norm. This normalization does not affect the parameter estimates, since the minimizer of $(W^{1/2}r)'(W^{1/2}r)$ is invariant to a scalar rescaling of $W^{1/2}$, but it improves numerical conditioning by bringing the objective function to order one.

A.9 Minimum loss functions and adaptive shrinkage

A.9.1 Computing minimum loss functions

For each parameter β^j , the minimum loss function measures how well the model can fit the data when β^j is held fixed at a particular value and all other parameters are free to adjust:

$$L(\beta^j) = \min_{\boldsymbol{\beta}^{-j}} (\hat{m} - m(\beta^j, \boldsymbol{\beta}^{-j}))' W (\hat{m} - m(\beta^j, \boldsymbol{\beta}^{-j})), \quad (44)$$

where $\boldsymbol{\beta}^{-j}$ denotes all parameters other than β^j . We evaluate this at 31 evenly spaced values of β^j across its current bounds. At each grid point, we run the full Levenberg-Marquardt estimation, with 30 restarts and $K = 10$ folds, over $\boldsymbol{\beta}^{-j}$, and take the median across folds. We also compute, at each grid point, the standard deviation of the minimum loss across the 10 folds. Whenever we report minimum loss functions, we report bands that span ± 2 of these pointwise standard deviations.

A.9.2 Adaptive bound shrinkage

We use minimum loss functions to narrow the parameter bounds during training, focusing subsequent computation on the region near the estimated parameters. We start shrinking only after a warm-up period of 200 training epochs, so that the value, policy, and moment networks are reasonably accurate before the bounds tighten.

Per-parameter level rule. For each parameter β^j , we find the value β^{j*} that minimizes its minimum loss function $L(\beta^j)$. Given a tolerance $\Delta > 0$, we construct a new interval for β^j as the contiguous region around β^{j*} where $L(\beta^j) \leq L(\beta^{j*}) + \Delta$. This level rule ensures parameters with sharp minimum loss functions get narrow intervals, while ones with flatter curves get wider intervals. To be conservative, we apply this rule to the minimum loss functions of each of the 10 validation folds, and for each parameter take the interval spanning the lowest and the highest endpoints across the 10 folds. We discuss our process for choosing

the tolerance Δ below, which involves safeguards that prevent shrinking when parameters are weakly identified, and retain parameter vectors that have recently been deemed plausible by the optimizer.

Identification guard. Before applying the level rule above to a parameter, we check whether its minimum loss function is sharp and precise enough to merit shrinking. We do so by comparing the minimum loss $L(\beta^{j*})$ to the 90th-percentile loss. If the 90th-percentile loss is not at least three standard deviations above the minimum loss, the bounds for the parameter do not shrink. The standard deviation is computed by taking the standard deviation across the 10 validation folds at each grid point and taking the median of those pointwise standard deviations. Before computing the standard deviation, we recenter each fold’s minimum loss function around its own minimum so as to ignore level differences across folds, since level differences do not affect which value of β^j minimizes the loss. The identification guard prevents shrinkage of the parameter space for parameters that are weakly identified or whose moment networks are still inaccurate.

Containment guard. A candidate parameter region is admissible only if it still contains two sets of parameter values. First, among the 500 most recent sampled parameters, the 50 whose simulated moments are closest to the target moments, where closeness is measured in the same weighted-GMM norm used to estimate β . Second, every Levenberg-Marquardt estimate produced in the current estimation step, across all 30 restarts and 10 cross-validation folds. A candidate region that excludes any parameter vector in these two sets fails the containment guard. This prevents the shrinkage algorithm from excluding plausible parameters.

Choosing the tolerance. The tolerance Δ controls how aggressively the bounds shrink, with smaller values of Δ producing tighter bounds. We do not pick Δ directly because it is measured in the same units as the loss function, and is thus hard to interpret and sensitive to the scale of the loss. Instead, we pick the volume of the new parameter region as a fraction v of the pre-shrink volume, where volume is the product of interval widths across all parameters

that pass the identification guard. For each candidate v , we find the Δ that produces a region whose volume is v times the previous volume. Starting from $v = 0.80$ (a 20% volume reduction), we search for the smallest $v \in [0.05, 1.0]$ that satisfies the containment guard. If no v in this range satisfies the containment guard, the parameter bounds are left unchanged for this round.

After shrinkage, the narrower bounds replace the original bounds. All subsequent operations use the narrower bounds, focusing computation on the region near the estimates.

A.9.3 Global identification diagnostic

We also use minimum loss functions for all parameters to assess identification, verifying if each has a unique global minimum. If the minimum loss function $L(\beta^j)$ for a parameter is flat, there are two possible explanations: the moments may not pin down β^j (weak identification) or the model may not reproduce the data moments at any parameter value (model misspecification). To isolate identification, we compute $L(\beta^j)$ in Eq. (44) above using as the target simulated moments at the parameter estimate $\hat{\beta}$ from Section 4.2, rather than the data moments. Specifically, we simulate moments at $\hat{\beta}$ using the simulation process described in Section A.6, producing a vector \tilde{m} of model-implied moments. We then compute the minimum loss function with \tilde{m} as the target,

$$L(\beta^j) = \min_{\beta^{-j}} (\tilde{m} - g(\beta^j, \beta^{-j}))' W (\tilde{m} - g(\beta^j, \beta^{-j})), \quad (45)$$

following the procedure in Section A.9. We use 31 grid points across the current bounds for β^j , 30 Levenberg-Marquardt restarts over the remaining parameters β^{-j} , and 10 cross-validation folds at each grid point. Note that this procedure estimates the model again, targeting the model-implied moments, allowing us to also verify if we can recover the correct parameters. Our identification diagnostic is thus twofold: we visually assess whether the minimum loss function has a unique minimum, and whether the second estimation recovers

the parameter vector used to construct the targeted moments—namely, the parameter vector obtained in the first estimation.

A.10 General equilibrium extension

In the partial equilibrium version, we estimate eight parameters and target 11 moments. The general equilibrium extension, described in Section 2.2, adds the wage w to the parameter vector and the market-clearing imbalance (the “resource gap” $Y - I - C - S$) to the targeted moments. We solve for the wage that clears markets by targeting an imbalance of zero. This section describes how our procedure changes in this case.

A.10.1 Parameter vector and bounds

The parameter vector expands to $(\theta, \rho, \sigma, \delta, \gamma_1, \gamma_0, \chi, c_f, w)$. We bound the wage between 0.5 and 1.5. Because A_π depends on the wage, the capital bounds described in Section A.2 also depend on w and are recomputed for each sampled parameter vector. The neural networks now receive a 9-dimensional (rather than 8-dimensional) parameter input through the FiLM conditioning mechanism. The network architecture is otherwise the same.

A.10.2 Targeted moments

The number of targeted moment increases from 11 to 12. The additional moment is the market-clearing imbalance $Y - I - C - S$, which we normalize by aggregate sales to put it on the same scale as other moments across the parameter space and improve numerical stability. Specifically:

$$\text{Resource Gap} = \frac{Y - I - C - S}{Y}, \quad (46)$$

where Y is aggregate sales, I is aggregate investment, $C = w/\phi$ is aggregate consumption, and S is aggregate cash savings. All aggregates are computed as cross-sectional means from the simulated firm panel. We target a value of zero for this moment, so that the wage clears

the resource constraint.

A.10.3 Weighting matrix

The weighting matrix for the 11 data moments is constructed from influence functions as described in Section A.8. The gap moment has no empirical counterpart, so there is no sampling variance from which to construct an influence function. We therefore extend the 11×11 partial equilibrium weighting matrix by appending a row and column of zeros with a unit diagonal entry for the gap. Because the partial equilibrium weighting matrix has been normalized to have a median column norm of 1, a weight of 1 on the market-clearing imbalance gives this moment a weight on the same scale as the data moments. Researchers who find that markets are not clearing tightly enough can increase this weight to place more emphasis on the gap relative to the data moments.

A.10.4 Changes to the algorithm

In the training step (Block 1), the neural networks now condition on 9 additional states, the 8 parameters plus the wage. In the simulation step (Block 2), the moment computation includes the resource gap. In the estimation step (Block 3), the moment-surrogate networks learn 12 mappings instead of 11, and the Levenberg-Marquardt optimizer searches over 9 parameters instead of 8. The gap moment, targeted at zero, forces the optimizer to find a wage w that clears the resource constraint simultaneously with the structural parameters that match the 11 data moments.

A.11 Asynchronous execution

The algorithm runs two processes concurrently on four GPUs:

- *GPU 1 (training)*. Runs the value and policy network training loop described in Section A.3. Each epoch consists of 500 gradient steps. The training runs continuously, cycling through epochs.

- *GPUs 2–4 (data collection)*. Draw batches of parameter vectors from the current bounds, evaluate the latest value and policy networks on the discrete grid, refine the grid-based solution (Section A.5), simulate firm panels (Section A.6), and compute moments.

The training and collection processes share the neural network weights. When GPUs 2–4 each begin evaluating a new batch of parameter vectors, they each read the most recent value and policy network weights from GPU 1. The four GPUs operate independently, with none waiting for the others to finish before proceeding. Because GPU 1 is continuously training these networks, the solution available to GPUs 2–4 improves over time, and with it the quality of the simulated moments.

At the end of every epoch, we execute two sets of computations, all described in detail above. First, we retrain the moment networks on the dataset accumulated by GPUs 2–4. Second, we evaluate the moment network accuracy by computing R^2 on held-out data, running the Levenberg-Marquardt estimation to produce updated parameter estimates, and evaluating the minimum loss functions. After the warm-up period of 200 epochs, we also attempt to shrink parameter bounds every epoch, but often do not shrink due to our safeguards against tightening bounds for weakly identified parameters and excluding parameter vectors that recent estimation rounds suggest are plausible.

A.12 Hyperparameters and other implementation choices

Table A2 reports our hyperparameters and optimizer choices. In this section, we discuss the reasoning behind these implementation choices and offer guidance for researchers applying the method to other models.

Appendix Table A2: Hyperparameter settings

<i>Panel A: Value and policy networks</i>		<i>Panel B: Moment-surrogate networks</i>	
Hidden layers	3	Hidden layers	3
Units per layer	128	Units per layer	32
Activation	SiLU	Activation	SiLU
FiLM generator	1 layer, 32 units	Optimizer	Adam
Optimizer	Adam	Learning rate	10^{-4}
Learning rate	10^{-3} (w/ decay)	Mini-batch size	256
Mini-batch size	8,192	SGD passes per update	200
Gradient steps per epoch	500	Cross-validation folds (K)	10
		Max training observations	10,000
<i>Panel C: Quadrature and grid</i>		<i>Panel D: Estimation</i>	
Gauss-Hermite nodes	5	Optimizer	Levenberg-Marquardt
State grid ($n_z \times n_k \times n_b$)	$11 \times 15 \times 35$	Restarts per fold	30
Control grid ($n_{k'} \times n_{b'} \times n_{c'}$)	$81 \times 91 \times 71$	Convergence tolerance	10^{-10}
Tauchen std. devs. (m)	2.5	Max iterations per restart	20
<i>Panel E: Simulation</i>		<i>Panel F: Min. loss functions and shrinkage</i>	
Firms	5,000	Grid points per parameter	31
Periods	300	Target volume fraction	0.80
Burn-in	200		

This table reports the hyperparameter settings used in our implementation. All hyperparameters are discussed in Section A.12.

A.12.1 Network architecture

The value and policy networks each have three hidden layers with 128 units. The moment networks are smaller, with three layers of 32 units, because they approximate a smoother, lower-dimensional mapping. More units typically increase accuracy, but increase the number of network weights and thus slow down computations.

Other deep learning approaches often use more than 128 units for value and policy networks, and we choose smaller networks for two reasons. The first is FiLM, which we discuss above in Section A.2.2. If we were to concatenate the model parameters and state variables into a single input vector, as in Duarte, Duarte, and Silva (2024), the network would have to learn from scratch how each parameter changes the mapping from states to values and policies, which would require a lot of capacity. FiLM reduces this burden by using

a small auxiliary network to determine how the model parameters should modulate the way that the main network processes the state variables. This means that the main network can be smaller while still remaining flexible enough to accurately represent value and policy functions across the parameter space.

The second reason is that we refine the solution with a few steps of optimization on a grid prior to simulation (Section A.5), which means that we can afford for the value and policy networks to be slightly less accurate approximations. Researchers adopting our method to smooth problems who decide to skip the refinement step may need to increase the number of units.

In general, we recommend that researchers start with a similar network architecture—or a smaller one for simpler applications—and monitor the Bellman residual during training. If the residual plateaus at a high level, the network may be too small, and we recommend increasing the number of units. We recommend keeping the number of layers at two or three, as is standard in the reinforcement learning literature. We use and recommend powers of 2 for the number of units (32, 64, 128, . . .) because GPU hardware processes these dimensions more efficiently.

A.12.2 Activation function

We use the SiLU activation function $\phi(x) = x \cdot \sigma(x)$ for all networks, which is common in economics and finance applications (e.g. Duarte, Duarte, and Silva, 2024). The most common activation function in the broader deep learning literature is ReLU, $\phi(x) = \max(0, x)$, but it has a non-differentiable kink at zero and a zero gradient for all negative inputs. In our application, these properties can lead to units that stop learning because they are stuck in the zero-gradient region. Other smooth activations—such as softplus, $\phi(x) = \log(1 + e^x)$, or tanh—work similarly well as SiLU in our experience.

A.12.3 Choice of optimizer

We use the Adam (Kingma and Ba, 2015) optimizer, which is the standard choice for neural network training. Plain stochastic gradient descent uses the same learning rate for every weight in the network, which can be inefficient when some weights need large updates and others need small ones. Adam maintains a running average of past gradients and past squared gradients for each weight individually, and use these to scale the learning rate. Weights that have been receiving consistently large gradients get smaller steps to avoid overshooting, while weights that have been receiving small or noisy gradients get larger steps to speed up progress. This per-weight adaptation generally leads to faster and more stable convergence than plain stochastic gradient descent. We recommend Adam as a starting point for researchers adopting our method.

While the value and policy functions are continuously being trained on fresh data, moment networks are trained on a fixed dataset, which can make them more prone to overfitting. We do not find that to be an issue in our application, but researchers who do should consider using AdamW for the moment networks. The difference between Adam and AdamW is that AdamW adds weight decay, a form of regularization that penalizes large network weights and reduces overfitting.

A.12.4 Learning rate

For the value and policy networks, we start with a learning rate of 10^{-3} and decrease by 1% per epoch, with a floor of 10^{-6} , similarly to Duarte, Duarte, and Silva (2024). A learning rate of 10^{-3} is the default for Adam and works well for this application, and the decay helps stabilize convergence. Early in training, when the networks are far from the solution, a higher learning rate allows for more rapid progress. As training progresses and the networks approach the solution, a lower rate prevents overshooting in response to feedback loops, such as the one between the value function and bond prices and, in the general equilibrium

extension, between the value function and the wage.

We use a lower constant learning rate of 10^{-4} for the moment network to reduce overfitting. A lower rate can help prevent overfitting by taking smaller steps rather than overresponding to idiosyncratic features of the training data, especially when combined with a fixed number of training steps. This is particularly important for the moment networks, which are trained repeatedly on the same accumulated dataset rather than on fresh samples drawn at each step. We do not use decay in the moment networks because it is a simpler problem, which is stable even without decay.

As a general recommendation, we suggest starting with a learning rate of 10^{-3} for the value and policy networks, and a lower rate of 10^{-4} for the moment networks. If training is unstable, with the loss function oscillating rather than decreasing steadily, we recommend introducing learning rate decay for the value and policy networks.

A.12.5 Batch size and training steps

Each epoch consists of 500 gradient steps with a mini-batch of 8,192 state-parameter pairs. We choose 500 steps because, in our hardware, 500 steps takes about a second, and so the code outputs convergence metrics every second. In our experience, the number of steps in an epoch is not an important hyperparameter, with different steps producing similar results.

The mini-batch size should be large enough to provide a representative sample of the state-parameter space at each step. With 11 input dimensions, 8,192 points provide reasonable coverage. Increasing the batch size improves the gradient estimate but slows each step. For models with more state variables or parameters, a larger batch size may be needed to maintain good coverage.

A.12.6 Quadrature

We use five Gauss-Hermite nodes to approximate the conditional expectation in the Bellman equation. With $Q = 5$ nodes, Gauss-Hermite quadrature is exact for polynomial integrands of degree $2Q - 1 = 9$. This is sufficient for our application, especially since we refine the neural network solution with a few steps of optimization on a grid (Section A.5). More nodes may be needed for models with highly nonlinear value functions, or when researchers choose to skip the refinement step. A simple diagnostic is to increase Q and check that the solution is stable. For models with non-normal shocks, researchers can replace the Gauss-Hermite nodes and weights with a quadrature rule appropriate for the shock distribution.

A.12.7 Simulation panel

We simulate 5,000 firms for 300 periods, discarding the first 200 as burn-in, as in Gao, Whited, and Zhang (2021). The burn-in must be long enough for the simulated cross-section to approximate the model's ergodic distribution. The appropriate length depends on how persistent the model's state variables are, and we find that 200 periods is conservative in our application. We recommend varying the burn-in length and checking whether the simulated moments change.

The number of firms controls the precision of the simulated moments at each parameter vector. Increasing it reduces simulation noise but increases computation time. In our application, 5,000 firms produce moments that are precise enough for the moment-surrogate networks to learn an accurate mapping.

A.12.8 Grid for refinement

The grid dimensions we use for state variables (11 for productivity, 15 for capital, 35 for net-debt) and controls (81 for next-period capital, 91 for gross debt, and 71 for cash) are comparable to those in standard VFI implementations of similar models (e.g., Gao, Whited,

and Zhang, 2021). Finer grids improve accuracy but increase the cost of each refinement and simulation step. For models with additional state or control variables, this step will be more computationally costly. In those situations, researchers should consider increasing the number of units in the value and policy networks to improve the accuracy of those approximations and reduce the number of refinement steps needed.

Researchers adopting our method to applications with smooth objective functions should generally skip the refinement step altogether. This refinement ensures that the solution is as accurate as value function iteration, and is helpful in our application, given that the objective function is non-differentiable and non-convex. For smooth problems, neural networks are typically more accurate than a grid.

A.12.9 Estimation restarts

We use 30 random starting points for Levenberg-Marquardt. In our application, the majority of restarts converge to the same solution, suggesting that the estimation objective has a well-defined global minimum for the moments we target. This implies that fewer restarts could likely suffice, but the computational cost of additional restarts is small since evaluating the moment-surrogate networks is nearly instantaneous.

A.12.10 Software and hardware

Our implementation uses JAX, a Python framework for GPU-accelerated array computation. Two features of JAX are central to our method: just-in-time compilation, which converts Python functions into optimized machine code via XLA (Accelerated Linear Algebra) and eliminates interpreter overhead, and automatic differentiation, which computes exact gradients of the training loss and the estimation objective. Modern GPU-accelerated computing frameworks, including JAX, can also automatically detect available hardware. Our code assigns computations to multiple GPUs when available and falls back to either one GPU or the CPU otherwise, which increases computation time but does not change the algorithm or

require any modifications.

We run the algorithm on four RTX5090 GPUs. The asynchronous execution described in Section A.11 uses one GPU for network training and three for simulation. We rent GPU time from vast.ai, where the cost of one hour of computations with four RTX5090 GPUs is approximate two dollars at the time of this writing. Similar cloud GPU providers include Lambda, CoreWeave, and RunPod.

A.13 DF Assistant: implementation from natural language

To lower the cost of applying our method to new models, our replication package will include an AI agent that can apply our approach to new models from a natural-language prompt. A researcher provides a description of the model in natural language and mathematical notation, and the agent outputs GPU-compatible implementation code. The agent is a structured protocol that instructs a large language model (LLM) on how to use our open-source library, relying on examples of correctly implemented problems. Currently, the examples are a series of tutorial notebooks that walk through our method, starting with a moment-network approximation of a simple AR(1) process, then introducing minimum loss functions and our shrinkage algorithm, our smoothing procedure, and building up to the model in our application. These tutorial notebooks will also be included in our replication package.

A.13.1 Model description

The researcher provides the agent with a description of the economic environment. For a decision problem, the description should include the state variables, actions, parameters with their values or bounds, shocks and their distributions, transition laws, the objective function, the discount factor, feasibility constraints, a reference calibration, numerical settings such as grid sizes and convergence tolerances, and the solution method (currently, only value function iteration and our deep policy iteration approach are supported). For estimation, the

description should also specify which parameters are to be estimated and their bounds, the target moments with their economic definitions, and the weighting matrix. If the description is incomplete, the DF Assistant makes decisions and documents them.

A.13.2 The protocol

The DF Assistant protocol guides the LLM through a structured workflow, starting with reading the library source code. It reads the examples and uses them as references for correct usage patterns. The LLM then extracts the economic content from the model description, determines whether the model requires solving a decision problem, estimation, or both, and maps each element of the specification to the corresponding component of the library to write a complete, working script.

The protocol enforces several safeguards to prevent hallucination and other failures, such as imposing tracing of every function call to the library source code or an existing example. Additional rules prevent common errors in GPU-accelerated code, such as using the wrong numerical library inside compiled functions or reusing random number keys in vectorized computations. The protocol also requires a self-review step, checking that all files are complete, all functions are defined, all imports are present, and the script will run from top to bottom in a fresh environment.

A.13.3 Test cases

We have so far tested the DF Assistant on models with known analytical solutions, allowing us to verify that the implementation is correct. The first two test cases focus on the library’s two solvers, value function iteration and deep policy iteration:

1. A deterministic cake-eating problem.
2. A consumption-savings problem with a two-state Markov income process.

The next four test the estimation procedure—including training moment networks, comput-

ing minimum loss functions, and shrinking parameter bounds—on stochastic processes with known parameters:

3. Estimating the innovation scale σ of an AR(1) process by targeting the unconditional second moment $E[x_t^2]$.
4. Estimating the moving-average coefficient θ of an MA(1) process by targeting the lag-1 autocovariance.
5. Estimating the persistence probability p of a two-state Markov chain with Gaussian measurement noise by targeting the observed autocorrelation.
6. Jointly estimating two coefficients of a quadratic regression from two cross-sectional moments.

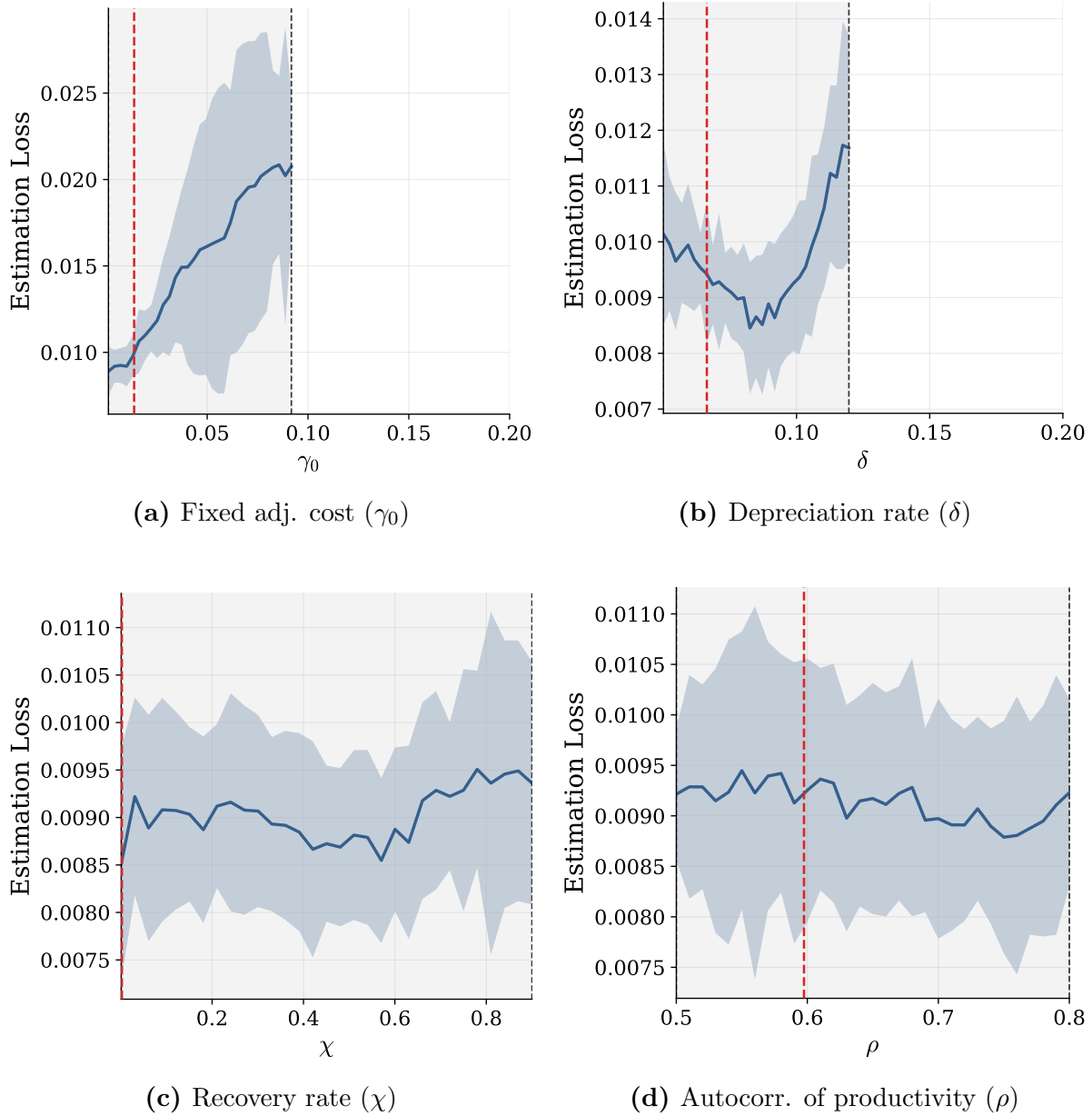
The last test case combines both, solving and estimating a simple decision problem:

7. Jointly solving and estimating a Brock-Mirman growth model with log utility and AR(1) productivity, estimating four parameters by targeting moments of log output.

In each test case, the agent correctly applied our library, and the code ran without modification. We have yet to test the agent on more complex models, and it is possible that its capabilities already far exceed what our test cases require. If not, we expect they soon will, as LLMs continue to improve and we add more examples to our library.

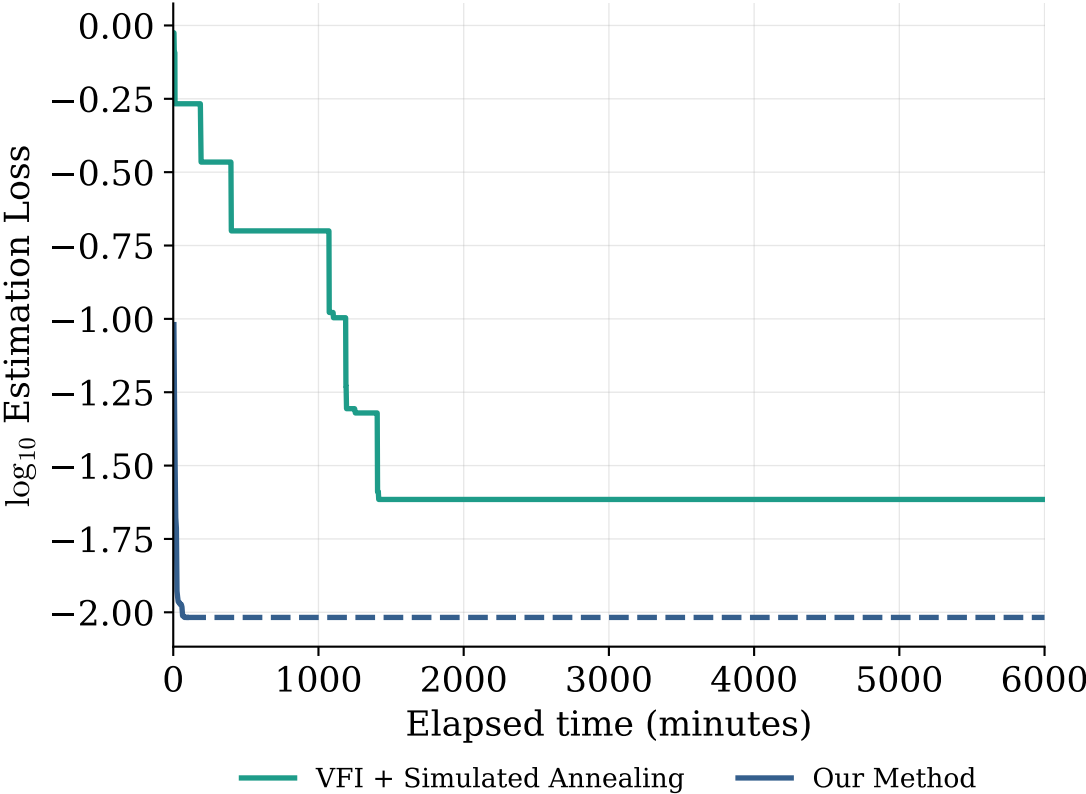
B Additional figures

Appendix Figure B2: Minimum Loss Functions with Data-Implied Targets



This figure shows the minimum loss function, described in Section 3.6, for four selected parameters: the fixed adjustment cost γ_0 , the depreciation rate δ , the recovery rate χ , and the autocorrelation of the productivity process ρ . In each panel, we fix the parameter on the x-axis and minimize the estimation loss function over all other parameters. The targeted moments are the Compustat data moments from Section 4.2. The solid line is the median across 10 folds, and the shaded band shows ± 2 standard deviations across the 10 folds. The red vertical dashed line denotes the parameter estimate. The x-axis spans the initial parameter bounds, with the black dashed vertical lines and shaded region marking the narrowed bounds after adaptive shrinkage (Section 3.5).

Appendix Figure B3: Time to Solution for General Equilibrium Model: 100-Hour Run



This figure compares the estimation loss over run time in minutes for our method (blue) and value function iteration combined with simulated annealing (green) for the general equilibrium version of the model described in Section 2.2. Both methods target the same Compustat moments and use the same weighting matrix. We run our method for 90 minutes and simulated annealing for 100 hours, with our method’s line becoming dashed after the end of training, and report the running best loss.