Deep Learning for Corporate Finance Models: Brief Report (Part I)*

Zhaoxuan Wang

2026-02-01

1 Introduction

This report documents the implementation of deep learning methods for solving dynamic corporate finance models. The core problem is to find optimal investment and financing policies for firms facing adjustment costs, stochastic productivity shocks, and (in the extended model) risky debt with endogenous default and equilibrium risky interest rates (see Strebulaev and Whited 2012, sec. 3).

The conventional solution to this dynamic programming/optimal control problem is to discretize the state space and iterate on value function or policy function until convergence. However, this approach suffers from the curse of dimensionality and is computationally expensive. It is less accurate due to approximiation on discrete grids, and it quickly becomes intractable as the number of state variables and sample size increases.

In this report, I explore and apply the deep learning methods introduced in Maliar, Maliar, and Winant (2021) to solve this problem. The approach casts economic models into trainable objectives and uses deep neural networks to parametrize policy and value functions, trained via three complementary methods:

1. **Lifetime Reward (LR)**: Maximize discounted cumulative cash flows over T periods
2. **Euler Residual (ER)**: Minimize violations of the first-order optimality conditions
3. **Bellman Residual (BR)**: Actor-critic training to satisfy the Bellman equation and optimal policy

The application covers two types of model:

*Email - wxuan.econ@gmail.com or zxwang13@student.ubc.ca. This is a report prepared for the interview with JPMorgan MLCoE.

- **Basic Model**: Firm chooses capital investment $(k, z) \to k'$ subject to adjustment cost
- **Risky Debt Model**: Firm chooses capital investment and borrowing $(k, b, z) \to (k', b')$ with endogenous default decision and equilibrium risky interest rates

The complete codebase and lastest report release can be found on GitHub and are updated frequently. Key features and extensions will be added in the next 4-8 weeks:

- *February*: Introduce GMM and SMM estimation (part 2)
- *March*: Complete all bonus questions and extensions
- *April*: Test method on real-world data and final submission

> Usage
>
> To reproduce results in this report and understand API usage, please run the jupyter notebook `../report/part1_demo.ipynb`

1.1 Literature Review

In this section, I briefly review the relevant literature on deep learning for solving dynamic and structural finance/economic models. I focus on the practical side of this topic and do not aim to provide a comprehensive review of the whole field of dyanmic programming, deep learning, and corporate finance. Instead, I focus on the papers that are most relevant to this project.

During my work on this project, I have mostly used and referred to the following sources (in addition to the assigned readings) that are excellent for an overview of the fundamentals.

- **Dynamic programming**: Ljungqvist and Sargent (2018), lecture notes (UBC), QuantEcon
- **Deep RL**: Online lectures by Huggingface, OpenAI, and Stanford CS224

The basic problem is represented as a Bellman equation:

$$V(s, a) = \max_{a \sim \pi} \mathbb{E}_{s' \sim P}[r(s, a) + \beta V(s', a')]$$

where s are states following transition rule $s' \sim P(\cdot|s, a)$, a is the action taken by the agent, $r(s, a)$ is the reward function, $\beta \in (0, 1)$ is the discount factor, and $V(s', a')$ is the value function for the next period. The core of the problem is to find the optimal policy mapping $\pi : (s, a) \to a'$ that maximizes the Bellman equation.

1.1.1 Discrete-State Dynamic Programming

Conventional methods as introduced in Strebulaev and Whited (2012) to solve this problem is through **value function iterations (VFI)** and **policy function iterations (PFI)** (Howard's improvement method). In their simplest form, these methods often start by discretizing the state space into a grid of points, that is, all possible values of (s, a). The VFI algorithm relies on the property that the Bellman (functional) operator is a contraction mapping with with fixed point V^*, thus repeatedly applying the operator to a random initial V_0 will lead to V^*.

The basic **VFI algorithm** is as follows:

1. Initialize V_0 on the grid
2. For $j = 0, 1, 2, ...$ until convergence:

$$V_{j+1}(s, a) = \max_{a \sim \pi} \mathbb{E}_{s' \sim P} \left[r(s, a) + \beta V_j(s', a') \right]$$

The pros of this method is its simple and robust, but it can be very slow over large state grids because for each iteration, we need to evaluate V_j for all possible pairs of (s, a).

The **PFI algorithm** relies on a different structure:

1. Pick any policy π_j, solve for the on-policy value function V^{π_j} that satisfied the Bellman Equation
2. For the given V^{π_j}, search for a better policy π_{j+1} such that $V^{\pi_{j+1}} > V^{\pi_j}$
3. Repeat until the policy converged $|\pi_{j+1} - \pi_j| < \epsilon$

The underlying idea is to first find the optimal value for an arbitrary policy, and then update on the policy to see if we can find a better one that increases the Bellman equation value. In discrete state dynamic programming (DDP), the PFI algorithm is generally prefered because it is more accurate and often converge faster than the VFI because we solved for the *exact* policy π in step two.

The common shortcomings of these two methods are that they are both computationally expensive and cannot be applied to continuous state spaces. A natural mitigation to the "curse of dimensionality" associated with fixed grids is to use a **polynomial approximation** to the value function.

> Codes
>
> I implemented both VFI and PFI in Tensorflow and it can be found under `src/ddp`. The `QuantEcon` library also has a `DiscreteDP` module implemented using Numpy.

1.1.2 Projection Methods

Here I briefly introduce a variant of the project method reviewed in Ljungqvist and Sargent (2018). The core idea is that instead of a step function over a discrete grid, we can parameterize and approximate the value function by a weighted sum of orthogonal polynomials.

The canonical method uses **Chebyshev polynomials**, defined on the domain $[-1, 1]$ by:

$$T_n(x) = \cos(n \arccos x)$$

For example, the first few polynomials are $T_0(x) = 1$, $T_1(x) = x$, and $T_2(x) = 2x^2 - 1$.

Let $V(s, a)$ be the true value function we wish to find, defined over a domain of asset states $a \in [a_{min}, a_{max}]$ (holding the exogenous state s fixed or treating it separately). Since Chebyshev polynomials are defined on $[-1, 1]$, we first define a linear transformation to map the economic state variable a (e.g., capital) to the polynomial domain x:

$$x = 2\frac{a - a_{min}}{a_{max} - a_{min}} - 1$$

The approximation of the value function at iteration i, denoted $V_i(s, a)$, is defined as a linear combination of $n + 1$ basis functions with coefficients $\{\theta_0, \theta_1, \dots, \theta_n\}$:

$$V_i(s, a) \approx \hat{V}(x;) = \theta_0 + \sum_{j=1}^{n} \theta_j T_j(x)$$

Crucially, the grid points are not chosen uniformly. To minimize approximation error, the method evaluates the function at the zeros of the Chebyshev polynomial, known as Chebyshev nodes. For a grid of size N_g, the nodes x_g in the $[-1, 1]$ domain are calculated as:

$$x_g = \cos\left(\frac{2g - 1}{2N_g}\pi\right), \quad g = 1, \dots, N_g$$

To summarize, this method maps economic states to Chebyshev nodes: $(s, a) \to x$. Then we iterate on the Bellman equation using continuous $T(x)$ as an approximation to the value function, and update the coefficients $(\theta_0, \{\theta_j\}_{j=1}^n)$ to minimize the approximation error. Finally, when the coefficients converge, we obtained the optimal mapping: $(s, a) \to x \to V(s, a;)$ with fixed coefficients .

1.1.2.1 Polynomial Approximation Algorithm

The solution is reached by iterating on the Bellman equation using these continuous approximations.

1. **Initialization** Choose a degree of approximation n, a number of grid points $m \geq n+1$, and an initial guess for the coefficients $= \{\theta_0, \ldots, \theta_n\}$ (e.g., all zeros).
2. **Maximization** For each node a_k (corresponding to x_k), solve the optimization problem on the RHS of the Bellman equation using the continuous value function $\widehat{V}(s', a'; {}_{old})$ from the previous iteration to find the optimal policy a':

$$\tilde{y}_k = \max_{a'} \left\{ r(s,a) + \beta \widehat{V}(s', a'; {}_{old}) \right\}$$

3. **Updating Coefficients** Once we have the new optimized values $\tilde{y}_k$ at each node x_k, we compute the new coefficients ${}_{new}$ to fit these values. For example, using the least squares formula for Chebyshev polynomials:

$$\theta_j = \frac{\sum_{k=1}^{m} \tilde{y}_k T_j(x_k)}{\sum_{k=1}^{m} T_j(x_k)^2}$$

4. **Convergence** Update the coefficients and repeat Steps 2 and 3 until the coefficients converge (i.e., $||_{new} - {}_{old}|| < \epsilon$).

In practice, rather than iterating on the Bellman equation directly, one often approximates the policy $\pi(s, a; \theta)$ using Chebyshev polynomials:

$$\pi(s, a; \theta) \approx \sum \theta_j T_j(s, a)$$

1.2 Deep Learning Methods

1.2.1 Motivation

Although the projection method allows for continuous approximation, it shares the critical drawback of the grid-based methods (VFI, PFI): the **curse of dimensionality**. As the number of state variables increases, the number of grid points required for accurate approximation grows exponentially, making the method computationally intractable for high-dimensional problems.

For example, consider a model with three state variables (k, b, z), each can take 100 possible values (grid points), then the total number of points to be evaluated is $100^3 = 10^6$, which means

- VFI need to evaluate $V(k, b, z)$ and $V(k', b', z')$ for 10^6 grids each, so the total number of points to be evaluated is 10^{12} for a single iteration step
- PFI also need to evaluate 10^6 grids and invert a matrix with $10^6 \times 10^6$ elements for a single policy improvement step
- Projection method need to update 10^6 coefficients for a single iteration step using Newton's method (e.g., least squares), including inverting a $10^6 \times 10^6$ matrix

This relatively simple model soon becomes intractable, and this is the exact motivation for Maliar, Maliar, and Winant (2021) to develop a gradient-based deep learning methods that is efficient for high-dimensional problems.

1.2.2 Core Idea

Here I briefly introduce the general representation of the DL method by Maliar, Maliar, and Winant (2021) and postpone the details to later sections. The core idea is to parameterize the value function and policy function using neural networks, and then use gradient descent to minimize an objective function that enforces optimality.

- Parameterize value $V(s, a; \theta_V)$ and policy $\pi(s, a; \theta_\pi)$ as functions of trainable parameters θ
- In deep neural networks, θ are the weights and biases of the neurons
- Objective function $\mathcal{L}(\cdot; \theta)$ captures an economic optimality condition we want to enforce
- Algorithm use gradient descent to minimize the objective function and update the parameters θ

Personally, I find this approach directly connected to the canonnical projection method. Specifically, the polynomial approximation $V_i(s, a) \approx \theta_0 + \sum_{j=1}^{n} \theta_j T_j(x)$ can be seen as a specific type of neural network with a single hidden layer, a linear activation function, and transformed input $T : (s, a) \to x$.

The DL method has several key advantages over the conventional methods:

1. **Tractability**: Number of parameters in a neural network grows *linearly* with the dimensionality of the state space so training remains tractable even for high-dimensional problems
2. **Speed**: SGD-based training is typically much faster than the Newton's method (inverting huge matrices)
3. **Precision**: Neural networks with "deep" configs (multiple hidden layers and neurons) can "learn" to approximate kinks and discontinuities while still achieve high precision (e.g., minimizing residual below 10^{-4})
4. **Flexibility**: Neural networks can be easily extended to handle more complex economic models, such as models with more state action variables and complex constraints

1.2.3 Highlights of My Innovation

During my implementation of Maliar, Maliar, and Winant (2021), I found that the original design could be improved in several aspects:

- AiO cross-product loss vs MSE loss
- Ergodic set sampling v.s. "learning the corners"
- Handling kinks and discontinuities
- Bellman Residual method via Actor-Critic training
- Reproducibility and comparability across methods

Here I briefly summarize the key improvements/innovations I made on these points. More technical details are discussed in later sections.

1.2.3.1 AiO estimator vs MSE loss

Maliar, Maliar, and Winant (2021) propose an All-in-One (AiO) expectation operator to compute the expectation instead of the traditional deterministic quadrature. The idea is to draw two random i.i.d. shocks ϵ_1, ϵ_2 from the distribution of ϵ and compute the sample mean:

$$\mathbb{E}[f(\epsilon)]^2 \approx \frac{1}{N}\sum_{i=1}^{N}\left[f(\epsilon_{i,1}) \cdot f(\epsilon_{i,2})\right]$$

This estimator is unbiased because $\mathbb{E}[f_1 \cdot f_2] = \mathbb{E}[f_1] \cdot \mathbb{E}[f_2] = \mathbb{E}[f]^2$ when f_1, f_2 are i.i.d. draws.

However, this estimator induces unstability for the risky debt training, where f include a default indicator p_ϵ that is approximated with Gumbel-Sigmoid (see details in later section). I intentially introduces a uniform random noise in Gumbel-Sigmoid to force the network to "learn" around the default boundary, but this amplifies the unstability of the AiO expectation estimator.

To mitigate this risk, I also implement the standard Mean Square Error (MSE) loss as benchmark:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{1}{2}(f_{i,1}^2 \cdot f_{i,2}^2)\right] \to \mathbb{E}[f(\epsilon)]^2 + \text{Var}[f(\epsilon)]$$

This is a standard loss function used in deel reinforcement learning. Although it is biased (due to the variance term), it is stable in training and the variance term is usually small with i.i.d ϵ draws and large sample size.

Codes

My neural network trainers for ER and BR method has two options for loss functoin: `mse` and `crossprod` (AiO). I prefer `mse` for numerical stability.

1.2.3.2 Ergodic Set Sampling

Maliar, Maliar, and Winant (2021) propose to sample from the ergodic set of the state space, which is the set of states that are visited with positive probability in the long run. The main motivation is to reduce state space and focus only on the relavant states in training (what they called "where the solution lives").

Specifically, the authors propose a "continuous simulation with resampling" approach that repeat the following steps in each training iteration:

- Start with random initial states (k_0, z_0)
- Extract the policy $\pi : (k, z) \to k'$ from previous training
- Simulate a full time series rollout (k_t, z_t) over T periods
  - This forms the training set for LR method
- (For ER and BR only) Randomly re-shuffle and sample transition samples (k, z) from the simulated time series and take a random draw ϵ
  - This forms the training set for ER and BR

I argue that there are two critical risks with this approach:

1. "On-the-fly" data generation in training process
2. Network never explores the "corners" that are off-policy

The "On-the-fly" data generation essentially prevent us from replicating the results or evaluate performance across different training runs/methods. I address this issue by developing a specific RNG schedule that

- Seperate data generation and random draws from training
- Enforce full reproducibility given fixed master seed (which fixed dataset)

As for the "unvisited corners", some of them are irrelevant and can be ignored: for example, investment (depends on k') and shocks z are usually strongly correlated so there is no point to learn about the policies at corners where z is high and k' is low.

However, for important boundaries such as the default gate, i.e., firm defaults when continuation value $V < 0$, I specifically design the network to explore and learn around the boundary $V = 0$, otherwise once V turn negative for the first time, firm will stay default and the network will never learn how to recover (see the risky debt section for more details).

1.2.3.3 Kinks and Discontinuities

Maliar, Maliar, and Winant (2021) propose using Fischer-Burmeister (FB) function to handle the kinks in empirical risk/loss due to the max operators. Yet the FB function does not handle kinks and discontinuities in reward and value functions (Maliar, Maliar, and Winant (2021) consider a smooth and differentiable CRRA utility as their examples).

In our corporate finance models, there are kinks (e.g., default gate) and discontinuities (e.g., fixed adjustment cost) in the rewards (cash flow). Using discrete, hard indicator functions like $\mathbb{1}\{x < 0\}$ will kill the gradient of the component to lead to incorrect solutions. I carefully handle this issue using a temperature-annealing schedule that start with a smooth differentiable function to approximate the hard gate, and let this function converge to zero/one as the temperature goes to zero near the end of trainings (when policies and values are accurate).

1.2.3.4 Bellman Residual method with Actor-Critic

In Maliar, Maliar, and Winant (2021), the Bellman Residual (BR) method is implemented via minimizing a multi-task objective function:

$$\mathcal{L} = w_1 \mathcal{L}_{\text{BR}} + w_2 \mathcal{L}_{\text{FOC}} + w_3 \mathcal{L}_{\text{Envelop}}$$

where w exogenous weights and the second and third term are First Order Conditions or Envelope Conditions to enforce optimality of the policy. This framework has a number of weakness:

- Requires tuning hyperparameters w
- Requires reward function is differentiable (for FOC or Envelop Condition)
- If there are mistakes in hard-coded FOC and Envelop condition, the training will be wrong

To be fair, Maliar, Maliar, and Winant (2021) mentioned "direct optimization" as an alternative to the multi-task objective, but they did not provide detail guidelines on how to implement it.

I fill this gap by refining an Actor-Critic framework that implements the "direct optimization". The idea is very similar to the conventional policy function iteration method, where each training has two main steps:

- Critic: Find the fixed point value function V_π by minimize Bellman residual for a given armbitrary policy π
- Actor: Find the fixed point policy π by maximizing the Bellman equation given the learned value function V_π

Compared with the multi-objective loss, the Actor-Critic training is robust to kinks and discontinuities, requires no hyperparameter tuning or derivation of the FOCs. This makes it flexible to extend to different models. The implemention details are described in the sections later.

1.2.3.5 Reproducibility and Comparability

The last refinement is to make the training results fully replicable and comparable across different methods, training runs, and configurations. This is achieved through two features:

- Deteministic RNG schedule that governs data generation, Monte Carlo draws, re-shuffle and resampling (for ER and BR)
- Use time series rollout (LR training set) to generate the training data for ER and BR

This guarantees that a single pair of master seed (e.g., `(20,26)`) will guarantee the same training/validation/test data for all methods and runs. Details of the RNG schedule is described in later sections.

1.3 Notation

For the remainder of the report, I define the commonly used notations:

Trainable Parameters

- Policy network parameters: θ_{policy}
- Value network parameters: θ_{value}
- Price network parameters (risky debt only): θ_{price}

Conventions

- Current period variable: x
- Next period variable: x'
- Parameterized function (Neural Networks): $\Gamma(\cdot;\theta)$

State and Action Variables

- Exogenous productivity shock: $z > 0$
- Capital stock: $k \geq 0$
- Debt (borrowing): $b \geq 0$

AR(1) Shock Process

The log-productivity follows an AR(1) process:

$$\ln z' = (1-\rho)\mu + \rho \ln z + \sigma\varepsilon', \quad \varepsilon' \sim \mathcal{N}(0,1) \text{ i.i.d.}$$

The ergodic (unconditional) standard deviation is $\sigma_{\ln z} = \sigma/\sqrt{1-\rho^2}$.

Policies and Prices

- Basic model policy: $(k, z) \mapsto k'$
- Risky debt policy: $(k, b, z) \mapsto (k', b')$
- Risky debt bond price: $(k', b', z) \mapsto q(k', b', z) = 1/(1 + \tilde{r}(k', b', z))$

Training Methods

- **LR**: Lifetime Reward loss $\mathcal{L}_{\text{LR}}$
- **ER**: Euler Residual loss $\mathcal{L}_{\text{ER}}$
- **BR**: Bellman Residual with critic loss $\mathcal{L}^{\text{BR}}_{\text{critic}}$ and actor loss $\mathcal{L}^{\text{BR}}_{\text{actor}}$

---

2 Economic Models

2.1 Basic Model

State and Action

- State: (k, z) where k is capital stock and z is productivity
- Action: k' (next-period capital)

Investment

$$I = k' - (1-\delta)k$$

Investment can be positive (expansion) or negative (disinvestment).

Operating Profit

$$\pi(k, z) = z \cdot k^{\gamma}, \quad \gamma \in (0, 1)$$

where γ is the production technology parameter (decreasing returns to scale).

Capital Adjustment Cost

$$\psi(I, k) = \phi_0 \cdot \frac{I^2}{2k} + \phi_1 \cdot k \cdot \mathbf{1}\{I \neq 0\}$$

where ϕ_0 is the convex adjustment cost coefficient and ϕ_1 is the fixed adjustment cost coefficient. The indicator $\mathbf{1}\{I \neq 0\}$ triggers whenever the firm invests or disinvests.

Cash Flow (Payout)

$$e(k, k', z) = \pi(k, z) - \psi(I, k) - I$$

Objective

The firm maximizes expected discounted lifetime cash flows:

$$\max_{\{k_{t+1}\}_{t=0}^{\infty}} \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t \cdot e(k_t, k_{t+1}, z_t)\right]$$

where $\beta = 1/(1+r)$ is the discount factor and r is the risk-free rate.

Bellman Equation

$$V(k, z) = \max_{k'} \{e(k, k', z) + \beta\, \mathbb{E}[V(k', z') \mid z]\}$$

2.2 Risky Debt Model

The risky debt model extends the basic model by allowing firms to borrow at an endogenous risky interest rate, with the option to default.

State and Action

- State: (k, b, z) where $b \geq 0$ is outstanding debt
- Action: (k', b') where $b' \in [0, b_{\max}]$ is new borrowing

Cash Flow

$$e(\cdot) = (1-\tau)\pi(k, z) - \psi(I, k) - I + q \cdot b' + \frac{\tau \tilde{r}\, b'}{(1+\tilde{r})(1+r)} - b$$

where:

- τ is the corporate tax rate
- b is repayment of last-period debt
- $q \cdot b' = b'/(1+\tilde{r})$ is proceeds from issuing new risky debt
- The third term is the tax shield from debt interest

External Financing Cost

When cash flow is negative, the firm must raise costly external equity:

$$\eta(e) = (\eta_0 + \eta_1|e|) \cdot \mathbf{1}\{e < 0\}$$

Endogenous Risky Interest Rate

The bond price $q = 1/(1+\tilde{r})$ is determined by the lender's zero-profit condition:

$$b'(1+r) = (1+\tilde{r})b' \, \mathbb{E}[1 - D \mid z] + \mathbb{E}[D \cdot R(k', b', z') \mid z]$$

where:

- LHS: Opportunity cost of lending at risk-free rate
- RHS: Expected return accounting for default probability and recovery

Endogenous Default

The firm defaults when its continuation (latent) value is negative:

$$D(k', b', z') = \mathbf{1}\{\tilde{V}(k', b', z') < 0\}$$

Shareholders walk away with zero under limited liability:

$$V(k', b', z') = \max\{0, \tilde{V}(k', b', z')\}$$

Recovery Under Default

$$R(k', z') = (1-\alpha)\left[(1-\tau)\pi(k', z') + (1-\delta)k'\right]$$

where $\alpha \in [0, 1]$ is the deadweight loss from liquidation.

Bellman Equation

$$\tilde{V}(k, b, z) = \max_{k', b'} \{e(\cdot) - \eta(e) + \beta \, \mathbb{E}[V(k', b', z') \mid z]\}$$

The Nested Fixed-Point Problem

A key computational challenge is that the latent value $\tilde{V}$ depends on the risky rate $\tilde{r}$, but solving for $\tilde{r}$ requires knowing the default probability $\mathbb{E}[D]$, which depends on $\tilde{V}$. Traditional methods solve this via nested iteration. The neural network approach trains policy, value, and pricing networks jointly, avoiding explicit nested loops.

---

3 Data Generation

This section describes how synthetic training data is generated with emphasis on reproducibility and numerical stability.

3.1 State Space Bounds

The state space bounds can be specified in two ways:

1. **Model-based (recommended)**: Specify bounds as multipliers on the steady-state capital k^*, which the framework converts to levels
2. **Direct specification**: Provide bounds directly in arbitrary units

Productivity Shock Bounds

Given AR(1) parameters (μ, σ, ρ), the ergodic distribution of $\ln z$ is truncated at m standard deviations:

$$\ln z \in [\mu - m \cdot \sigma_{\ln z},\ \mu + m \cdot \sigma_{\ln z}], \quad \sigma_{\ln z} = \frac{\sigma}{\sqrt{1-\rho^2}}$$

The default $m = 3$ covers the mass of the ergodic distribution.

Capital Stock Bounds

The frictionless steady-state capital (where marginal product equals user cost) is:

$$k^*(z) = \left(\frac{z \cdot \gamma}{r + \delta}\right)^{\frac{1}{1-\gamma}}$$

evaluated at the stationary mean productivity $z = e^{\mu}$.

Users specify bounds as multipliers on k^*:

$$k_{\min} = k^{\text{mult}}_{\min} \times k^*, \quad k_{\max} = k^{\text{mult}}_{\max} \times k^*$$

For example, multipliers $(0.2, 3.0)$ yield a state space from 20% to 300% of steady-state capital.

Debt Bounds

The maximum borrowing (debt) is capped by the collateral constraint defined as:

$$b' \leq (1-\tau)\pi(k', z_{\min}) + \tau\delta k' + s_{\text{liquid}} \cdot k'$$

where s_{liquid} is the liquidation fraction between 0 and 1. The RHS is the maximum liquidation value of the firm in the worst state $z_{\min}$. The lower bound is simply $b_{\min} = 0$.

It is important to note that in section 3.6 of Strebulaev and Whited (2012), the authors remove the collateral constraint and allow for $b' < 0$ to denote cash saving. Although this is helpful for theoretical analysis, it introduces serious numerical instability in training because - firm can easily leverage a huge b' that dominate cash flow and capital investment - the interest rate switch between a constant r and an endogenously-determined state variable $\tilde{r}$ around default boundary $b' = 0$

Therefore, I restored the collateral constraint and restrict training to the borrowing case $b' \geq 0$. It is straightforward to extend the model by introducing a new state variable for cash saving with risk-free rate r (or any other pricing schedule).

Validation Constraints

When using model-based bounds, the framework validates:

- $m \in (2, 5)$: Sufficient coverage without extreme outliers
- $k_{\min}^{\text{mult}} \in (0, 0.5)$: Allows starting below steady state
- $k_{\max}^{\text{mult}} \in (1.5, 5)$: Allows starting above steady state

These constraints prevent numerical overflow while permitting economically meaningful variation.

3.2 Input Normalization

Neural networks perform best with inputs in a standardized range. The framework normalizes all state variables to $[0, 1]$ internally using min-max normalization:

Capital: $\hat{k} = (k - k_{\min})/(k_{\max} - k_{\min})$

Debt: $\hat{b} = b/b_{\max}$

Log-productivity: $\widehat{\ln z} = (\ln z - \ln z_{\min})/(\ln z_{\max} - \ln z_{\min})$

This normalization:

- Prevents exploding gradients from large input magnitudes
- Ensures consistent scale across different parameterizations
- Is invertible, so outputs can be mapped back to economic levels

Networks accept inputs in levels and perform normalization internally. Outputs are also returned in levels via bounded sigmoid activations scaled to the appropriate range.

3.3 Dataset Structure

3.3.1 Training Set

The training set is a stream of batches $\{\mathcal{B}^j\}_{j=1}^{J}$ where each batch contains n i.i.d. samples:

$$\mathcal{B}^j = \left\{ \left(k_{0,i}, b_{0,i}, z_{0,i}, \{\varepsilon_{t,i}^{(1)}, \varepsilon_{t,i}^{(2)}\}_{t=1}^{T} \right) \right\}_{i=1}^{n}$$

Each sample i represents an independent firm with:

- Initial states (k_0, b_0, z_0) drawn uniformly from the state space
- Two independent shock sequences $\varepsilon^{(1)}, \varepsilon^{(2)}$ for the cross-product estimator

The initial debt b_0 is used only in the risky debt model.

3.3.2 Validation and Test Sets

- **Validation set**: Fixed dataset of size $N_{\text{val}} = 10n$, used for model selection and early stopping
- **Test set**: Fixed dataset of size $N_{\text{test}} = 50n$, used only for final evaluation

Both are generated from the same distribution but with different RNG seeds.

3.4 Main Path vs. Fork Path

For each sample, two shock realizations are generated to enable unbiased estimation of squared expectations.

Main Path (AR(1) Chain)

The main shock sequence $\{z_t^{(1)}\}$ forms a continuous AR(1) chain:

$$\ln z_{t+1}^{(1)} = (1-\rho)\mu + \rho \ln z_t^{(1)} + \sigma \varepsilon_{t+1}^{(1)}$$

This path is used for lifetime reward calculations in the LR method.

Fork Path (One-Step Branches)

The fork sequence $\{z_t^{(2)}\}$ branches from the main path at each period:

$$\ln z_{t+1}^{(2)} = (1-\rho)\mu + \rho \ln z_t^{(1)} + \sigma \varepsilon_{t+1}^{(2)}$$

Each fork $z_{t+1}^{(2)}$ is a one-step transition from the main path $z_t^{(1)}$, not a parallel chain.

```
Main (AR1 Chain):   z0 -> z1 -> z2 -> z3 -> ... -> zT
                      \     \     \     \
Fork (1-step AR1):    z1F   z2F   z3F   zTF
```

Usage by Method

- LR: Uses main path only (continuous chain for lifetime rewards)
- ER/BR: Uses cross-product of main and fork at each transition for variance reduction

3.5 Trajectory vs. Flattened Data

The data generator produces two formats optimized for different training methods.

3.5.1 Trajectory Format (for LR)

The LR method requires full trajectories to compute cumulative discounted rewards:

$$\text{Shape: } (N, T+1) \text{ for shock paths}$$

Each sample preserves the temporal sequence $(z_0, z_1, ..., z_T)$.

3.5.2 Flattened Format (for ER and BR)

The ER and BR methods operate on single-step transitions and require i.i.d. samples for valid stochastic gradient descent. A key design choice is that states (k, b) are sampled **independently** for each transition rather than extracted from simulated trajectories.

Rationale: At data generation time, no policy exists to generate the capital/debt sequence $(k_1, k_2, ..., k_T)$. Using an arbitrary behavioral policy would introduce bias. Instead, the framework samples (k, b) directly from the state space, treating each draw as from the ergodic distribution. This approach:

1. **Ensures i.i.d. samples**: Each transition is statistically independent, satisfying the assumptions of SGD
2. **Eliminates serial correlation**: Consecutive time steps in a trajectory are correlated; independent sampling removes this dependency
3. **Approximates ergodic coverage**: Uniform sampling over the bounded state space approximates draws from the ergodic distribution that would arise under an optimal policy

The flattened dataset has shape $(N \times T,)$ with each observation:

$$\text{Obs} = (k, b, z, z'_1, z'_2)$$

where (z'_1, z'_2) are the main and fork next-period shocks. After flattening, the dataset is randomly shuffled to further eliminate any residual structure.

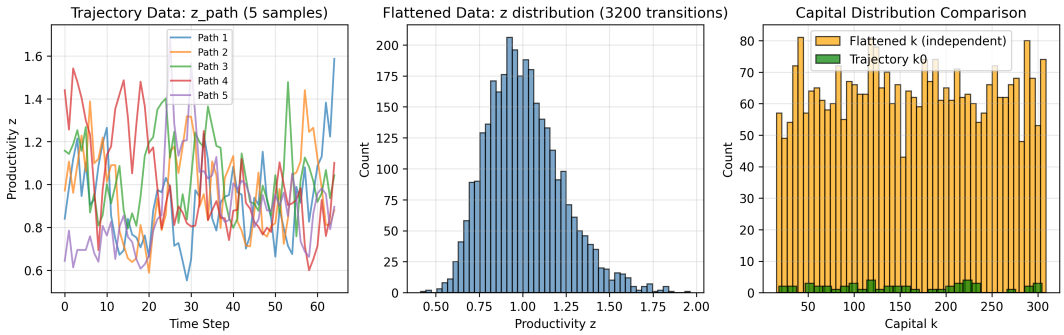

Figure 1: Comparison of trajectory and flattened data formats. Left: trajectory format preserves temporal structure for LR method. Right: flattened format with independent state sampling for ER/BR methods. *Example output from debug-mode training with* $T = 64$.

3.6 RNG Seed Schedule

Reproducibility requires deterministic random number generation. The framework uses TensorFlow's stateless random functions (`tf.random.stateless_*`) which produce identical outputs given identical seed pairs, regardless of call order.

Master Seed

A master seed pair $\mathbf{s}^{\text{master}} = (m_0, m_1)$ anchors all randomness.

Split Seeds

Disjoint seeds for each dataset:

$$\mathbf{s}^{\text{train}} = (m_0 + 100, m_1), \quad \mathbf{s}^{\text{val}} = (m_0 + 200, m_1), \quad \mathbf{s}^{\text{test}} = (m_0 + 300, m_1)$$

Variable IDs

Each random variable has a fixed ID:

Variable	ID
k_0	1
z_0	2
b_0	3
$\varepsilon^{(1)}$	4
$\varepsilon^{(2)}$	5

Training Seeds by Step

For training step j and variable x:

$$\mathbf{s}_{j,x}^{\text{train}} = (m_0 + 100 + \text{VarID}(x),\ m_1 + j)$$

This ensures each batch j receives unique, reproducible random draws.

Validation/Test Seeds

These are single fixed datasets (no step index):

$$\mathbf{s}_{x}^{\text{val}} = (m_0 + 200 + \text{VarID}(x),\ m_1)$$

$$\mathbf{s}_{x}^{\text{test}} = (m_0 + 300 + \text{VarID}(x),\ m_1)$$

This schedule guarantees:

1. **Reproducibility**: Identical seeds produce identical data across runs
2. **Common random numbers**: Different methods train on the same data, enabling fair comparison
3. **No leakage**: Train/validation/test sets use disjoint RNG streams

---

4 Network Architecture

4.1 Basic Model Networks

Policy Network

$$k' = \Gamma_{\text{policy}}(k, z; \theta_{\text{policy}})$$

- **Input**: (k, z) in levels, normalized internally to $[0, 1]$
- **Hidden layers**: 2 layers with 32 units each, SiLU activation

- **Output**: Sigmoid scaled to $[k_{\min}, k_{\max}]$:

$$k' = k_{\min} + (k_{\max} - k_{\min}) \cdot \sigma(\text{raw})$$

Value Network (BR method only)

$$V(k, z) = \Gamma_{\text{value}}(k, z; \theta_{\text{value}})$$

- **Input/Hidden**: Same as policy network
- **Output**: Linear (unbounded), since value can be any real number

4.2 Risky Debt Model Networks

Policy Network

$$(k', b') = \Gamma_{\text{policy}}(k, b, z; \theta_{\text{policy}})$$

- **Input**: (k, b, z) in levels, normalized internally to $[0, 1]$
- **Hidden layers**: Shared trunk with 2 layers, 32 units, SiLU activation
- **Output heads**:
  - $k' = k_{\min} + (k_{\max} - k_{\min}) \cdot \sigma(\text{raw}_k)$
  - $b' = b_{\max} \cdot \sigma(\text{raw}_b)$ (ensures $b' \geq 0$)

Value Network

$$\tilde{V}(k, b, z) = \Gamma_{\text{value}}(k, b, z; \theta_{\text{value}})$$

- **Output**: Linear (latent value can be positive or negative)

Price Network

$$q(k', b', z) = \Gamma_{\text{price}}(k', b', z; \theta_{\text{price}})$$

- **Output**: Sigmoid scaled to $(0, 1/(1+r)]$:

$$q = \frac{1}{1+r} \cdot \sigma(\text{raw})$$

  This ensures $\tilde{r} \geq r$ (risky rate at least equals risk-free rate).

4.3 Common Design Choices

Architecture

- Fully connected networks with configurable depth and width
- Default: 2 hidden layers, 32 units each
- SiLU (swish) activation: $\text{SiLU}(x) = x \cdot \sigma(x)$, which provides smoother gradients than ReLU

Input Processing

- Networks receive inputs in economic levels
- Internal min-max normalization to $[0, 1]$
- Economic primitives (π, ψ, e, etc.) computed using levels

Output Constraints

- Bounded outputs use sigmoid scaled to valid ranges
- Unbounded outputs (values) use linear activation

---

5 Basic Model Training

5.1 Lifetime Reward (LR) Method

The LR method directly maximizes expected discounted lifetime rewards.

Objective

$$\max_{\theta} \ \mathbb{E}_{k_0, z_0, \{\varepsilon_t\}} \left[\sum_{t=0}^{T-1} \beta^t e(k_t, k_{t+1}, z_t) + \beta^T V^{\text{term}}(k_T, z_T) \right]$$

Terminal Value

To correct for finite-horizon truncation, a terminal value approximates continuation:

$$V^{\text{term}}(k_T, z_T) = \frac{e(k_T, k_T, z_T)}{1 - \beta}$$

This assumes the policy has converged to steady state by period T, maintaining $k_{T+1} = k_T$ with replacement investment $I = \delta k_T$.

Empirical Loss

$$\mathcal{L}^{\text{LR}}(\theta) = -\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\left(\sum_{t=0}^{T-1}\beta^t e(k_{t,i}, k_{t+1,i}, z_{t,i}) + \beta^T V^{\text{term}}(k_{T,i}, z_{T,i})\right)$$

Algorithm

1. Load batch with initial states (k_0, z_0) and shock sequences $\{\varepsilon_t^{(1)}\}$
2. Simulate trajectory: $k_{t+1} = \Gamma_{\text{policy}}(k_t, z_t; \theta)$ for $t = 0, \ldots, T-1$
3. Compute discounted rewards and terminal value
4. Update θ via gradient descent on $\mathcal{L}^{\text{LR}}$

Key Points

- Uses trajectory data (main path only)
- Entire simulation is differentiable (gradients flow through time via backpropagation)
- Fork path not needed (no cross-product estimator)

5.2 Euler Residual (ER) Method

The ER method minimizes violations of the first-order optimality conditions.

Euler Equation

From the Lagrangian, the optimal policy satisfies:

$$1 + \psi_I(I_t, k_t) = \beta\,\mathbb{E}\left[\pi_k(k_{t+1}, z_{t+1}) - \psi_k(I_{t+1}, k_{t+1}) + (1-\delta)(1 + \psi_I(I_{t+1}, k_{t+1}))\right]$$

where subscripts denote partial derivatives.

Marginal Benefit Function

$$m(k', k'', z') = \pi_k(k', z') - \psi_k(I', k') + (1-\delta)(1 + \psi_I(I', k'))$$

where $I' = k'' - (1-\delta)k'$.

Unit-Free Residual

Dividing by the LHS gives a unit-free residual:

$$f = 1 - \beta \cdot \frac{m(k', k'', z')}{1 + \psi_I(I, k)}$$

At the optimum, $\mathbb{E}[f] = 0$.

Empirical Loss (Cross-Product)

$$\mathcal{L}^{\text{ER}} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} f_{i,1} \times f_{i,2}$$

Target Network for Stability

Computing $k'' = \Gamma_{\text{policy}}(k', z'; \theta)$ creates a recursive dependency. To stabilize training, a target network θ^- computes future actions:

$$k'' = \Gamma_{\text{policy}}(k', z'; \theta^-)$$

The target is updated via Polyak averaging: $\theta^- \leftarrow \nu\theta^- + (1-\nu)\theta$.

Algorithm

1. Load flattened batch (k, z, z'_1, z'_2)
2. Compute $k' = \Gamma_{\text{policy}}(k, z; \theta)$ and $I = k' - (1-\delta)k$
3. Compute $k''_\ell = \Gamma_{\text{policy}}(k', z'_\ell; \theta^-)$ for $\ell = 1, 2$
4. Compute residuals f_1, f_2 and cross-product loss
5. Update θ and Polyak-update θ^-

5.3 Bellman Residual (BR) Method

The BR method uses actor-critic training to satisfy the Bellman equation.

5.3.1 Critic Update

The critic (value network) is trained to satisfy:

$$V(k, z) = e(k, k', z) + \beta\, \mathbb{E}[V(k', z')]$$

Bellman Target

$$y_\ell = e(k, k', z) + \beta\, \Gamma_{\text{value}}(k', z'_\ell; \theta^-_{\text{value}}), \quad \ell = 1, 2$$

where actions come from the target policy $k' = \Gamma_{\text{policy}}(k, z; \theta^-_{\text{policy}})$.

The target is detached from the gradient (treated as a constant label).

Critic Loss

$$\mathcal{L}^{\text{BR}}_{\text{critic}} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (V_i - y_{i,1})(V_i - y_{i,2})$$

where $V_i = \Gamma_{\text{value}}(k_i, z_i; \theta_{\text{value}})$.

5.3.2 Actor Update

The actor (policy network) maximizes the Bellman RHS:

$$\max_{\theta_{\text{policy}}} \mathbb{E}\left[e(k, k', z) + \beta V(k', z')\right]$$

Actor Loss

$$\mathcal{L}^{\text{BR}}_{\text{actor}} = -\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left[e(k_i, k'_i, z_i) + \beta \Gamma_{\text{value}}(k'_i, z'_{i,1}; \theta_{\text{value}})\right]$$

where $k'_i = \Gamma_{\text{policy}}(k_i, z_i; \theta_{\text{policy}})$ uses the current (not target) policy.

Only the main shock z'_1 is used (sample mean estimator, not cross-product).

5.3.3 Algorithm

1. Load flattened batch (k, z, z'_1, z'_2)
2. **Critic step** (repeat N_{critic} times per actor step):
   - Compute target y_ℓ using target networks
   - Update θ_{value} to minimize $\mathcal{L}^{\text{BR}}_{\text{critic}}$
   - Polyak-update θ^-_{value}
3. **Actor step**:
   - Compute $k' = \Gamma_{\text{policy}}(k, z; \theta_{\text{policy}})$
   - Update θ_{policy} to minimize $\mathcal{L}^{\text{BR}}_{\text{actor}}$
   - Polyak-update θ^-_{policy}

5.4 Gradient Flow and Target Networks

A common source of bugs in implementing ER and BR methods is incorrect handling of gradient flow. This section clarifies when to detach gradients and why target networks with Polyak averaging are essential.

5.4.1 The Problem: Recursive Dependencies

In both ER and BR methods, the loss depends on future quantities computed by the same network being trained:

- **ER**: The residual f depends on $k'' = \Gamma_{\text{policy}}(k', z'; \theta)$, where k' itself depends on θ
- **BR Critic**: The target y depends on $V(k', z') = \Gamma_{\text{value}}(k', z'; \theta_{\text{value}})$
- **BR Actor**: The objective depends on $V(k', z')$ evaluated at the actor's chosen k'

If gradients flow through these future quantities naively, training becomes unstable: the network chases a moving target that changes with every parameter update.

5.4.2 Solution: Target Networks with Gradient Detachment

The solution has two components:

1. **Target networks** (θ^-): Maintain a slowly-updating copy of the trainable parameters
2. **Gradient detachment**: Stop gradients from flowing through the target computation

Why both are needed: Detachment alone (using the current θ but stopping gradients) still causes the target to shift every step. Target networks provide a stable reference that changes slowly via Polyak averaging.

5.4.3 Polyak Averaging

After each training step, the target network is updated as a weighted average:

$$\theta^- \leftarrow \nu\,\theta^- + (1-\nu)\,\theta$$

where $\nu \in (0.99, 0.999)$ is the averaging coefficient. With $\nu = 0.995$ (default), the target network tracks the current network but with significant lag, providing stability.

5.4.4 Gradient Flow Rules by Method

ER Method

Table 2: Basic Model

Computation	Network	Gradient?	Rationale
$k' = \Gamma_{\text{policy}}(k, z; \theta)$	Current	Yes	This is what we're optimizing
$k'' = \Gamma_{\text{policy}}(k', z'; \theta^-)$	Target	No	Stable reference for future action

The residual $f(k, k', k'', z')$ has gradients only w.r.t. k'. If we accidentally used the current policy for k'', the gradient would try to adjust the policy to make k'' satisfy the Euler equation—but k'' is a future decision that should be determined by the *current* state (k', z'), not by backpropagation.

BR Critic Update

Table 3: Basic Model

Computation	Network	Gradient?	Rationale
$V = \Gamma_{\text{value}}(k, z; \theta_{\text{value}})$	Current	Yes	This is what we're fitting
$k' = \Gamma_{\text{policy}}(k, z; \theta^{-}_{\text{policy}})$	Target	No	Fixed action for target
$y = e + \beta \Gamma_{\text{value}}(k', z'; \theta^{-}_{\text{value}})$	Target	No	Fixed label (regression target)

Table 4: Risky Debt Model

Computation	Network	Gradient?	Rationale
$V = \Gamma_{\text{value}}(k, b, z; \theta_{\text{value}})$	Current	Yes	This is what we're fitting
$(k', b') = \Gamma_{\text{policy}}(k, b, z; \theta^{-}_{\text{policy}})$	Target	No	Fixed action for Bellman target
$q^{\text{target}} = \Gamma_{\text{price}}(k', b', z; \theta^{-}_{\text{price}})$	Target	No	Fixed price for stable cash flow e
$q = \Gamma_{\text{price}}(k', b', z; \theta_{\text{price}})$	Current	Yes	This is what we're fitting
$y = e(q^{\text{target}}) - \eta + \beta V^{-}_{\text{eff}}$	Target	No	Fixed label (regression target)

The critic update is a regression: fit $V(k, z)$ to the target y. The target must be treated as a constant label (like supervised learning), otherwise the critic could "cheat" by adjusting the target rather than improving its prediction. For risky debt, the price network plays a dual role: (1) the *target* price q^{target} computes the cash flow e for the Bellman target, ensuring stability; (2) the *current* price q enters the pricing residual loss, which trains the price network to satisfy lender zero-profit.

BR Actor Update

Table 5: Both Basic and Risky Debt Model

Computation	Network	Gradient?	Rationale
$k' = \Gamma_{\text{policy}}(k, z; \theta_{\text{policy}})$	Current	Yes	This is what we're optimizing
$V(k', z') = \Gamma_{\text{value}}(k', z'; \theta_{\text{value}})$	Current	No	Frozen value landscape

The actor optimizes policy to maximize $e(k, k', z) + \beta V(k', z')$. Gradients flow through k' (to improve the action) but not through θ_{value} (the value function is held fixed during the actor step). If gradients flowed into θ_{value}, the actor could artificially inflate V rather than finding genuinely better actions.

5.4.5 Common Mistakes (that I've made and corrected)

1. **Forgetting to detach targets**: Results in unstable training where loss oscillates or diverges. The network optimizes by moving the target rather than improving predictions.
2. **Using current network for future actions**: In ER, computing k'' with the current policy instead of the target creates a circular dependency that destabilizes learning.
3. **Allowing gradients through value in actor update**: The actor manipulates the value function instead of finding better policies. Symptom: actor loss decreases but policy quality doesn't improve.
4. **Not using Polyak averaging**: Hard target updates (periodically copying $\theta \to \theta^-$) cause discontinuous jumps in the target, leading to training instability.

5.4.6 Implementation Pattern (TensorFlow)

```
# Critic update: detach target
with tf.GradientTape() as tape:
    V_pred = value_net(k, z)                          # Trainable
    k_next = target_policy_net(k, z)                  # Detached (no tape)
    V_next = target_value_net(k_next, z_next)         # Detached (no tape)
    y = tf.stop_gradient(e + beta * V_next)           # Explicit stop
    loss = cross_product_loss(V_pred, y)
grads = tape.gradient(loss, value_net.trainable_variables)

# Actor update: gradient through action, not through value
with tf.GradientTape() as tape:
    k_next = policy_net(k, z)                         # Trainable
    V_next = value_net(k_next, z_next)                # Used but frozen
    loss = -tf.reduce_mean(e + beta * V_next)
grads = tape.gradient(loss, policy_net.trainable_variables)  # Only policy
```

The key insight: anything computed *outside* the `GradientTape` context is automatically detached. Target networks are never inside the tape.

5.5 Handling Discontinuities

Several model components involve discontinuous indicator functions that have zero gradient almost everywhere, blocking learning signals.

5.5.1 Sources of Discontinuity

1. **Fixed adjustment cost**: $\mathbf{1}\{I \neq 0\}$ triggers for any nonzero investment
2. **External financing cost**: $\mathbf{1}\{e < 0\}$ triggers for negative cash flow
3. **Default indicator**: $\mathbf{1}\{\tilde{V} < 0\}$ triggers when continuation value is negative

5.5.2 Sigmoid Smoothing with Annealing

Replace hard indicators with temperature-controlled sigmoids that:

- Provide gradient signal during early training (high temperature)
- Converge to true indicators as temperature decreases

Adjustment Cost Gate

$$\sigma\left(\frac{|I/k| - \epsilon}{\tau}\right) \to \mathbf{1}\{|I/k| > \epsilon\} \quad \text{as } \tau \to 0$$

Normalizing by k prevents saturation for large investments.

External Financing Gate

$$\sigma\left(-\frac{e/k + \epsilon}{\tau}\right) \to \mathbf{1}\{e/k < -\epsilon\} \quad \text{as } \tau \to 0$$

Default Gate (Gumbel-Sigmoid)

The default boundary requires exploration of both solvent and default regions. A deterministic sigmoid risks getting stuck (e.g., learning "never default"). The Gumbel-Sigmoid adds stochastic exploration:

$$p = \sigma\left(\frac{-\tilde{V}/k + \log u - \log(1-u)}{\tau}\right), \quad u \sim \text{Uniform}(0,1)$$

The noise $\log u - \log(1-u)$ (logistic distribution) forces the network to explore around the default boundary when τ is large.

5.5.3 Annealing Schedule

Temperature decays exponentially:

$$\tau_j = \tau_0 \cdot d^j$$

where τ_0 is the initial temperature (default: 1.0), d is the decay rate (default: 0.995), and j is the iteration.

Annealing stops at $\tau_{\min}$ (default: 10^{-4}), after which the temperature is held constant. The number of decay steps is:

$$N_{\text{decay}} = \frac{\log(\tau_{\min}) - \log(\tau_0)}{\log(d)}$$

A stabilization buffer (default: 25%) allows fine-tuning at low temperature:

$$N_{\text{anneal}} = \lceil N_{\text{decay}} \cdot (1 + \text{buffer}) \rceil$$

---

6 Risky Debt Training

6.1 Method Selection

The risky debt model can in principle use LR, ER, or BR methods, but practical considerations favor BR.

LR Method: Applicable but challenging. The nested fixed-point problem (risky rate depends on value, which depends on risky rate) is implicit in the lifetime simulation. The network must simultaneously learn the policy and the equilibrium pricing, which can be unstable without the explicit critic structure.

ER Method: Problematic. Deriving Euler equations for the risky debt model requires differentiating through the default indicator and bond pricing equilibrium. The resulting conditions are complex, and the indicator discontinuities are harder to handle without the value function providing a stable learning target.

BR Method (Recommended): The actor-critic structure naturally decomposes the equilibrium conditions:

- Critic learns value and pricing to satisfy Bellman and zero-profit conditions
- Actor optimizes policy given the learned value/pricing functions

This decomposition handles the nested fixed-point problem without explicit iteration and provides stable learning targets for each component.

6.2 BR Method for Risky Debt

6.2.1 Pricing Loss

The lender's zero-profit condition determines bond prices. Define the pricing residual:

$$f_\ell = q \cdot b' \cdot (1 + r) - [p_\ell \cdot R(k', z'_\ell) + (1 - p_\ell) \cdot b']$$

where:

- $q = \Gamma_{\text{price}}(k', b', z; \theta_{\text{price}})$ is the predicted bond price
- p_ℓ is the Gumbel-Sigmoid default probability at shock z'_ℓ
- $R(k', z')$ is the recovery value

The MSE pricing loss is defined as

$$\mathcal{L}^{\text{price}}_{\text{MSE}} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{1}{2}(f^2_{i,1} + f^2_{i,2})$$

6.2.2 Critic Update

The critic target incorporates limited liability via the smooth effective value:

$$V_{\text{eff}} = (1 - p) \cdot \tilde{V}$$

where p is the default probability. As $\tau \to 0$, this converges to $\max\{0, \tilde{V}\}$.

Bellman Target

$$y_\ell = e(\cdot) - \eta(e) + \beta \cdot V_{\text{eff}}(\Gamma_{\text{value}}(k', b', z'_\ell; \theta^-_{\text{value}}))$$

Combined Critic Loss

$$\mathcal{L}_{\text{critic}} = \omega_1 \mathcal{L}^{\text{BR}} + \mathcal{L}^{\text{price}}$$

where ω_1 is exogenous weight to balance the two loss components (tuned to match scales). I set $ _1 (0.1, 0.5) $ to prioritize the zero-profit condition.

6.2.3 Actor Update

The actor maximizes expected continuation value:

$$\mathcal{L}^{\text{BR}}_{\text{actor}} = -\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left[e(\cdot) - \eta(e) + \beta \cdot V_{\text{eff}}(\Gamma_{\text{value}}(k'_i, b'_i, z'_{i,1}; \theta_{\text{value}})) \right]$$

The smooth V_{eff} (rather than hard max) is essential here. A hard max has zero gradient in the default region, providing no signal for the actor to adjust (k', b') to exit default.

6.2.4 Algorithm Summary

1. **Critic step** (repeat N_{critic} times):
   - Compute actions $(k', b') = \Gamma_{\text{policy}}(k, b, z; \theta^{-}_{\text{policy}})$
   - Compute Bellman targets y_ℓ and pricing residuals f_ℓ
   - Update $(\theta_{\text{value}}, \theta_{\text{price}})$ on combined loss
   - Polyak-update target networks
2. **Actor step**:
   - Compute actions using current policy
   - Update θ_{policy} to minimize actor loss
   - Polyak-update target policy
3. **Annealing**: Decay temperature τ for default and other gates
4. **Stopping**: Check convergence after annealing completes

---

7 Convergence and Early Stopping

Since we generally do not have closed-form solutions to benchmark against, I design the following convergence/stopping criteria to ensure that the training is sufficient and the solutions are valid.

The stopping logic is hierarchical: first ensure annealing is complete, then check method-specific convergence on the validation set.

7.0.1 Annealing Gatekeeper

If current step $< N_{\text{anneal}}$, ignore all early stopping triggers. This ensures the soft gates have sharpened before evaluating convergence. For example, failure to do so may kill the "inaction" region of investment policy due to fixed adjustment cost.

7.0.2 Method-Specific Criteria

Computed on the validation set to avoid overfitting.

LR Method (Relative Improvement Plateau)

Stop when relative improvement over a window s falls below threshold:

$$\frac{|\bar{\mathcal{L}}^{\text{LR}}(j) - \bar{\mathcal{L}}^{\text{LR}}(j-s)|}{|\bar{\mathcal{L}}^{\text{LR}}(j-s)|} < \epsilon_{\text{LR}}$$

where $\bar{\mathcal{L}}$ is a moving average.

ER Method (Absolute Threshold)

Stop when loss is near zero:

$$\mathcal{L}^{\text{ER}} < \epsilon_{\text{ER}}$$

The ER loss has a known optimum of zero (Euler equation holds exactly).

BR Method (Dual Convergence)

Both conditions must be satisfied:

1. Critic accuracy: $\mathcal{L}^{\text{BR}}_{\text{critic}} < \epsilon_{\text{critic}}$
2. Actor stability: Relative improvement in actor objective $< \epsilon_{\text{actor}}$

7.0.3 Patience Counter

A patience counter tracks consecutive checks where criteria are met:

- Increment if criteria satisfied
- Reset to zero if criteria violated
- Stop when counter exceeds patience limit (default: 5)

7.0.4 Training Logs

Figures Figure 2 and Figure 3 show the example training loss curves and diagnostic metrics for

- LR, ER, and BR methods on the basic model
- BR method for risky debt model

Note that the curves are from demo run and does not represent convergened result.

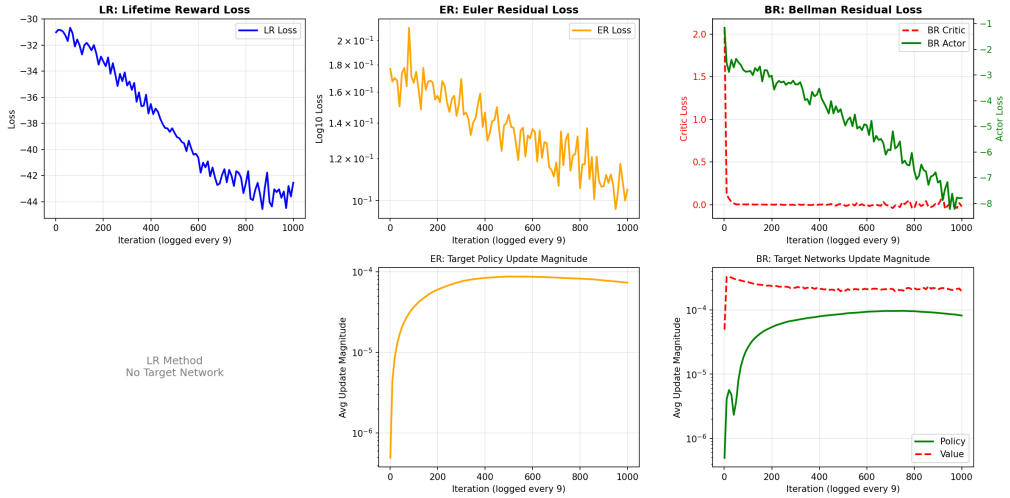

Figure 2: Training loss curves for LR, ER, and BR methods on the basic model. LR minimizes negative lifetime reward; ER and BR minimize squared residuals (shown in log scale). *Example output from debug-mode training; production results will differ.*

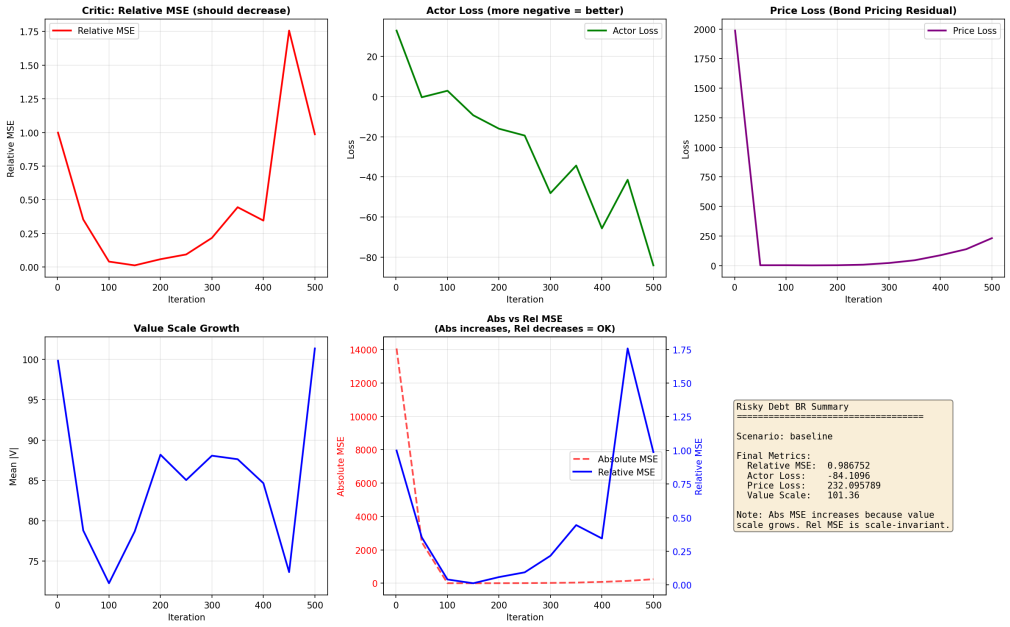

Figure 3: Training loss curves for LR, ER, and BR methods on the risky debt model. LR minimizes negative lifetime reward; ER and BR minimize squared residuals (shown in log scale). *Example output from debug-mode training; production results will differ.*

8 Implementation Details

8.1 Centralized Configuration

The codebase separates configuration into two layers to prevent circular imports and parameter drift.

Technical Defaults (`src/_defaults.py`)

A leaf module with zero imports containing all training hyperparameters:

- Learning rates, batch sizes, iteration counts
- Temperature annealing parameters
- Early stopping thresholds
- Polyak averaging coefficients

Both trainer configurations and annealing schedules import from this single source.

Economic Parameters (`src/economy/parameters.py`)

Frozen dataclasses for domain knowledge:

- `ShockParams`: AR(1) parameters (ρ, σ, μ)
- `EconomicParams`: Production and cost parameters $(\theta, r, \delta, \phi_0, \phi_1, ...)$

The `frozen=True` setting prevents accidental mutation. Changes require explicit `with_overrides()` calls that log all modifications.

8.2 Numerical Stability

Several design choices ensure stable training:

1. **Input normalization**: All states mapped to $[0, 1]$ before network processing
2. **Bounded outputs**: Sigmoids scaled to valid ranges prevent invalid states
3. **Logit clipping**: Soft gates clip extreme logits (default: ± 20) to prevent saturation
4. **Normalized indicators**: Ratios like I/k and e/k prevent scale-dependent gate behavior

8.3 Reproducibility

1. **Stateless RNG**: All randomness uses `tf.random.stateless_*` functions
2. **Seed scheduling**: Deterministic mapping from master seed to per-variable, per-step seeds
3. **Configuration hashing**: Cache files include MD5 hash of configuration for invalidation
4. **Metadata logging**: All generated datasets store their configuration for verification

8.4 Checkpointing

The `src/utils/checkpointing.py` module provides save/load functionality for training results. Keras models (policy, value, price networks and their targets) are serialized as `.keras` files, while training history and configuration dataclasses are stored as JSON metadata. This allows resuming analysis without re-running expensive training. The `save_training_result()` and `load_training_result()` functions handle serialization transparently, and `save_all_results()` / `load_all_results()` support batch operations across multiple experiments with automatic index generation.

---

9 Results and Post-training Evaluation

This section reports training results (demo) and discuss strategies to verify the correcteness and effectiveness of the solutions.

> **Note on Figures**
>
> The results below are from debug-mode training with small batch sizes and limited iterations. They demonstrate the methodology but not converged solutions. None of the reported runs triggered the convergence criteria. Production-quality results will be generated after full training runs.

> **Demo Training Configurations**
>
> I run the demo training with 2020 Macbook Pro (M1) and the following configs: Sample size $= N \times T$ with horizon $T = 64$, $N = 50$. Training batch size $= 64$ and maximum number of iteration $= 500$. Demo run takes about 45-50 minutes on my laptop. The results can be reproduced by running `src/report/part1_demo.ipynb`.

9.1 Verification Strategies

In principle, the convergence/stopping rule should ensures the effectiveness of the solution. In addition, I propose three post-training verification strategies to validate the correctness of the solutions.

1. **Analytical benchmark**: Special case with closed-form analytical solutions (e.g., basic model with no adjustment costs)

2. **Cross-method consistency**: Policies learned by LR, ER, and BR should be highly similar (if not exactly the same) when trained on the same data
3. **Economic sanity checks**: Policies should exhibit expected qualitative behavior (e.g., comparative statics)

9.2 Analytical Benchmark

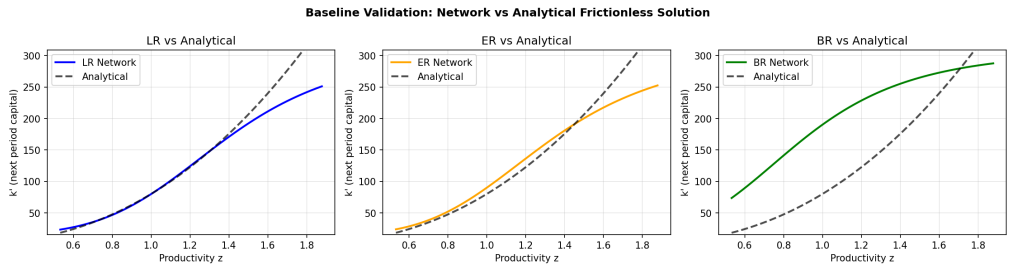

Figure 4: Validation against analytical solution at baseline (no adjustment costs). Learned policies from all three methods are compared against the closed-form Euler solution. Close agreement validates the implementation. *Debug-mode results; full training will improve accuracy.*

In the basic model, with zero adjustment costs ($\psi = 0$), the Euler equation simplifies to:

$$\mathbb{E}[\gamma z'(k')^{\gamma-1} \mid z] = r + \delta$$

Using properties of the lognormal distribution:

$$\mathbb{E}[z' \mid z] = \exp\left((1-\rho)\mu + \rho \ln z + \frac{1}{2}\sigma^2\right)$$

This yields the analytical policy:

$$k'(z) = \left[\frac{\gamma \exp\left((1-\rho)\mu + \rho \ln z + \frac{1}{2}\sigma^2\right)}{r+\delta}\right]^{\frac{1}{1-\gamma}}$$

Key properties of this benchmark:

- k' is independent of current capital k
- k' increases in z when $\rho > 0$

Learned policies should match the analytical solution in the no-adjustment-cost case (Figure 4). Note that the BR approach typically perform poorly when iteration is small because it converge relatively slower.

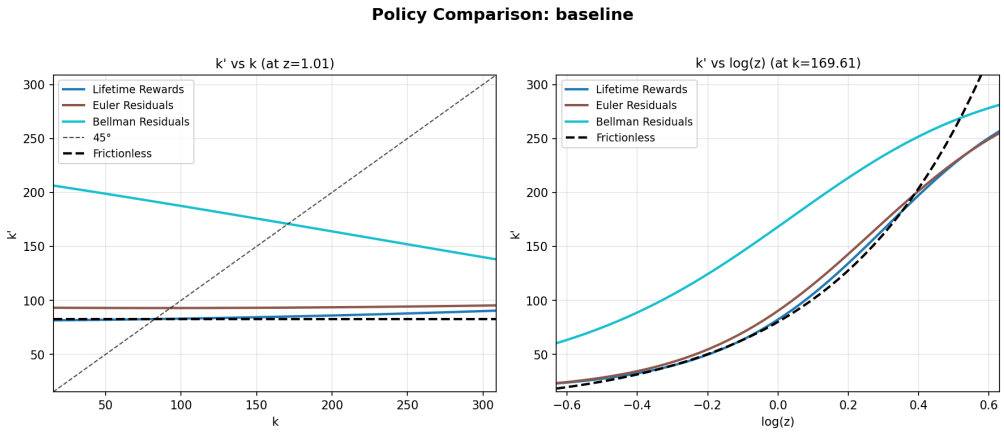

Figure 5: Policy comparison at baseline parameters (no adjustment costs). Frictionless dash line represent analytical solution

9.2.1 2D Policy Slice

For the basic model, I visualize policies along two dimensions:

- k' vs. k at fixed $\ln z = \mu$ (capital dynamics)
- k' vs. $\ln z$ at fixed $k = k^*$ (response to productivity)

The result should also confirms that k' is independent of k so that the policy function is a flat line. The 45 degree line represent steady state capital k^*. See Figure 5.

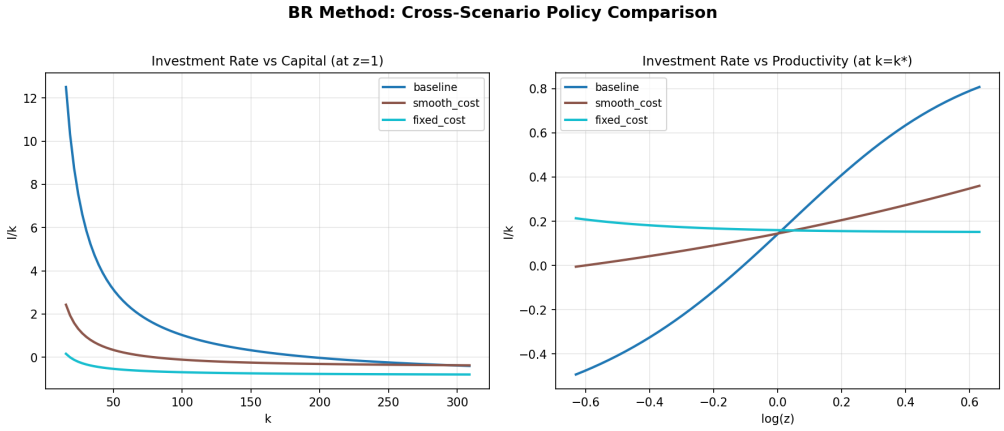

Figure 6: Optimal investment rate comparison at baseline parameters (no adjustment costs). Frictionless dash line represent analytical solution

A more economically meaningful plot is to compute the implied investment rate $\frac{I}{k} = \frac{k' - (1-\delta)k}{k}$

following Strebulaev and Whited (2012). This measure is unit-free and is comparable across methods/configs. See Figure 6.

One feature we should expect is that the baseline (frictionless) investment rate should be **smoothly** decreasing in k and increasing with productivity shocks $\log z$. In contrast, non-zero smooth (convex) and fixed adjustment cost will compress the investment rate and induce potential inaction regions. see Figure 6

9.3 Basic Model Results

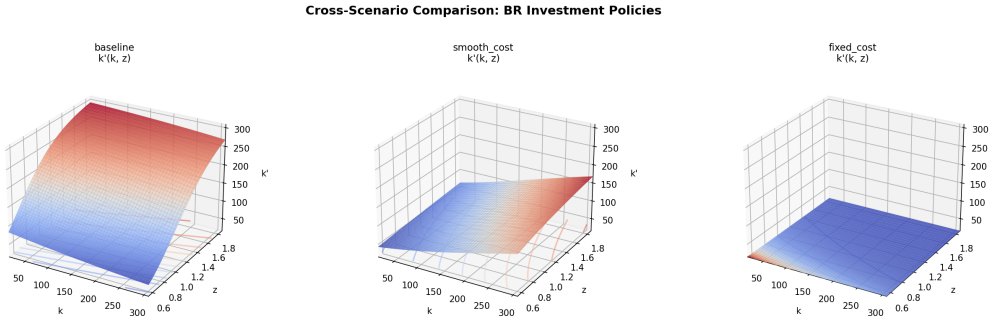

Figure 7: 3D visualization of optimal capital policy $k'(k, z)$ across all three scenarios under the basic model. The surface shows how policy responds jointly to current capital and productivity.

The cleanest way to visualize the result is probably to plot the 3D policy surfaces that maps (k, z) to k'. This figure also highlights how adjustment cost affects the optimal investment policies. See Figure 7.

- Baseline frictionless: smooth surface increasing with both states
- Smooth/convex adjustment cost: "flatten" surface because investments are costly
- Fixed adjustmetn cost: could trigger inaction regions because firm only investment when return pass a fixed threshold

9.4 Risky Debt Model Results

The risky debt model produces joint policies for capital and borrowing that maps current period (k, b, z) to (k', b').

Start with a clean 3D plot in Figure 8. Compared to the basic model, I specified non-negative adjustment costs and costly external financing (equity injection). Thus the results represent a frictional benchmark.

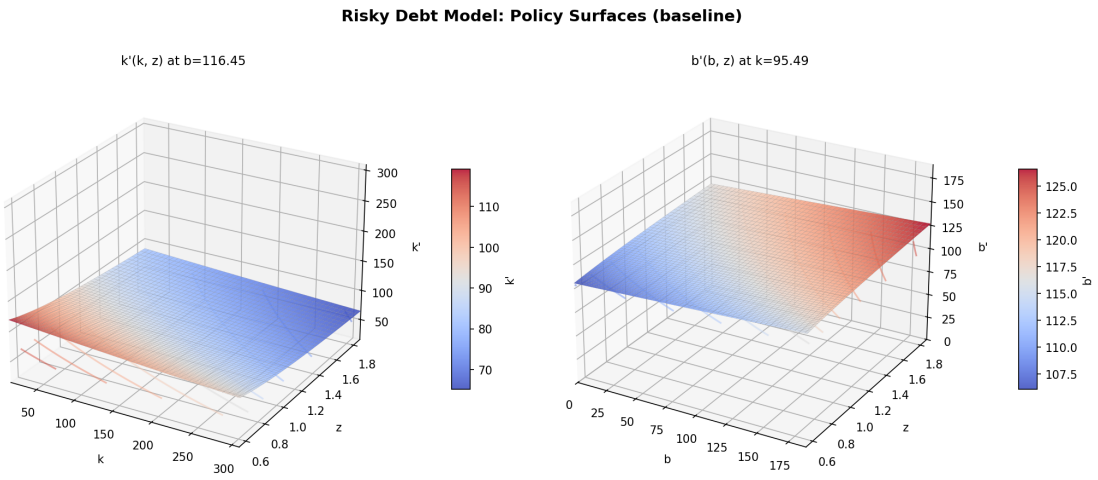

Figure 8: 3D policy surfaces for the risky debt model showing $k'(k, b, z)$ and $b'(k, b, z)$ at fixed productivity levels.

Unlike the basic model, the optimal policies in the risky debt model are in 4D space, so I need to fix one of the current state. On the left panel, I plot k' against (k, z) fixing current debt b at its steady state level. On the right panel, I plot b' against (k, z) fixing current capital k at its steady state level.

If we'd like to focus on comparative statics, the Figure 9 plots the 2D slices of optimal policies against current state variables fixing the other state variables at their steady state levels.

10 Test Suite

I implement a comprehensive test suite to validate the correctness of economic primitives, network architectures, training algorithms, and integration across components. This section describes the testing methodology and coverage.

10.1 Unit Tests

Unit tests verify individual functions and modules in isolation.

Economic Primitives (`tests/economy/`)

Tests validate parameter dataclasses, state space bounds, and economic logic:

- `test_parameters.py`: Verifies `EconomicParams` and `ShockParams` validation (e.g., $\rho \in [-1, 1]$, $\delta > 0$) and TensorFlow tensor conversion

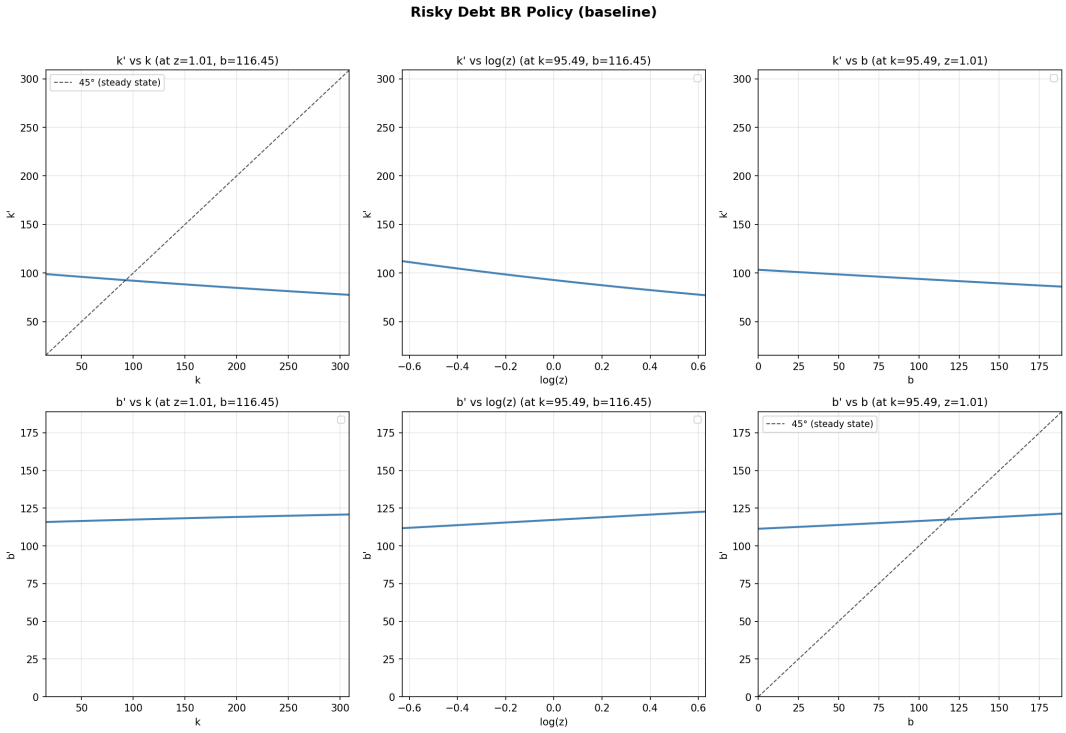

Figure 9: Risky debt BR policy: optimal capital k' and borrowing b' as functions of current state. Left panels show response to capital; right panels show response to productivity.

- `test_natural_bounds.py`: Tests ergodic log-z bounds computation from AR(1) parameters, steady-state capital k^*, and collateral-constrained debt bounds
- `test_logic.py`: Validates production function, adjustment costs, cash flow, Euler equation components (χ, m, residual), recovery values, and pricing residuals. Critically, tests verify gradient flow through economic functions for differentiability

Loss Functions (`tests/trainers/test_losses.py`)

Each loss function is tested against manual calculations:

- LR loss: $\mathcal{L}^{\text{LR}} = -\frac{1}{N}\sum_i \sum_t \beta^t e_t$ verified with known discount sequences
- ER AiO loss: $\text{mean}(f_1 \cdot f_2)$ tested with explicit residual pairs
- BR critic loss: Cross-product $(V - y_1)(V - y_2)$ validated with hand-computed deltas
- Price loss: Zero-profit residual formula verified component-by-component

Gradient Flow (`tests/trainers/test_gradient_flow.py`)

These tests enforce the gradient blocking rules from Section 5.5:

1. *Critic block isolation*: Verifies `tf.stop_gradient` on policy outputs prevents $\nabla_{\theta_{\text{policy}}} \mathcal{L}_{\text{critic}} = 0$
2. *Actor gradient existence*: Confirms policy receives gradients through value network in actor update
3. *Target detachment*: Validates RHS continuation term has no gradient (treated as constant label)
4. *Limited liability*: Tests ReLU subgradient ($\nabla = 1$ for $\tilde{V} > 0$, $\nabla = 0$ for $\tilde{V} < 0$)

These tests prevent subtle bugs where networks "cheat" by manipulating targets instead of improving predictions.

Data Generation (`tests/economy/test_data_generator.py`, `test_rng.py`)

Reproducibility tests verify:

- Identical master seeds produce identical train/validation/test data across runs
- Disjoint RNG streams ensure no data leakage between splits
- Trajectory format shapes $(N, T+1)$ and flattened format shapes $(N \times T,)$ are correct
- AR(1) rollout correctness: $z_{t+1} = \exp((1-\rho)\mu + \rho \ln z_t + \sigma\varepsilon_{t+1})$

Network Architectures (`tests/networks/test_networks.py`)

Tests verify:

- Output bounds: Policy outputs in $[k_{\min}, k_{\max}]$, price outputs in $(0, 1/(1+r)]$
- Input normalization: States mapped to $[0, 1]$ internally
- Shape consistency across batch dimensions
- Weight initialization (target networks start identical to current networks)

10.2 Integration Tests

Integration tests verify end-to-end behavior of complete training pipelines.

Trainer Integration (`tests/trainers/test_br_trainers.py`, `test_er_trainer.py`, `test_risky_br_trainers.py`)

These tests simulate complete training loops:

- Multiple training steps execute without numerical instability (NaN/Inf)
- Polyak averaging updates all target networks at configured rate ν
- `n_critic_steps` parameter controls critic-to-actor update ratio
- TensorFlow Dataset pipelines integrate correctly with trainer APIs
- Annealing schedule decreases temperature across iterations

DDP Solver Validation (`tests/ddp/test_ddp_debt.py`)

The discrete-state DDP solver provides ground truth for neural network validation:

- *Geometric series test*: With constant reward R, VFI must converge to $V^* = R/(1-\beta)$
- *Bellman operator tests*: Discounting, max optimization, limited liability, and probability mixing verified independently
- *Equilibrium pricing*: Solvent firms receive risk-free rate; defaulting firms receive recovery-adjusted prices

Cross-Method Consistency (`tests/trainers/test_basic_with_debt_data.py`)

Tests verify that Basic model trainers correctly ignore the debt dimension when data includes b, enabling a shared data pipeline for both Basic and Risky Debt models.

10.3 Test Coverage Summary

Module	Unit Tests	Integration	Key Invariants
Economy	6 files	—	Parameter validation, bounds, gradients
Networks	1 file	—	Output bounds, normalization, shapes
Trainers	7 files	4 files	Loss formulas, gradient flow, target networks
DDP	4 files	2 files	VFI convergence, equilibrium pricing
Utils	1 file	—	Annealing decay, indicator functions

The test suite contains approximately 260 individual tests. All tests use `pytest` with deterministic seeds for reproducibility.

11 References

Ljungqvist, Lars, and Thomas J. Sargent. 2018. *Recursive Macroeconomic Theory*. Fourth Edition. Cambridge, Massachusetts: MIT Press.

Maliar, Lilia, Serguei Maliar, and Pablo Winant. 2021. "Deep Learning for Solving Dynamic Economic Models." *Journal of Monetary Economics* 122 (September): 76–101. https://doi.org/10.1016/j.jmoneco.2021.07.004.

Strebulaev, Ilya A, and Toni M Whited. 2012. "Dynamic Models and Structural Estimation in Corporate Finance." *Foundations and Trends in Finance*.