Machine Learning for Computational Economics

Dejanir H. Silva
Purdue University

December 4, 2025

Contents

Chapter 5

The Deep Policy Iteration Method

In this chapter, we show how to use the machine-learning tools developed in Chapter 4 to solve high-dimensional dynamic programming problems in continuous time. We introduce the *Deep Policy Iteration* (DPI) method, a powerful technique for solving such problems by combining stochastic optimization, automatic differentiation, and neural-network function approximation. We will see how DPI can be applied to a wide range of economic and financial problems, including asset pricing, corporate finance, and portfolio choice.

5.1 The Dynamic Programming Problem

We consider a continuous-time optimal control problem in which an infinitely lived agent faces a Markovian decision process. The system's state is represented by a vector $\mathbf{s}_t \in \mathbb{R}^n$, and the control by a vector $\mathbf{c}_t \in \Gamma(\mathbf{s}_t) \subset \mathbb{R}^p$, where $\Gamma(\mathbf{s}_t)$ denotes the set of admissible controls at time t given state $\mathbf{s}_t$. Importantly, the feasible control set depends on the state. For instance, in the consumption–savings problem discussed in Chapter 3, the maximum feasible consumption at time t depends on the household's wealth, which is one of the state variables.

The agent's objective is to maximize the expected discounted value of the reward function $u(\mathbf{c}_t)$:

$$V(\mathbf{s}) = \max_{\{\mathbf{c}_t\}_{t=0}^{\infty}} \mathbb{E}\left[\int_0^{\infty} e^{-\rho t} u(\mathbf{c}_t)\, dt \,\middle|\, \mathbf{s}_0 = \mathbf{s} \right], \tag{5.1}$$

subject to the stochastic law of motion

$$d\mathbf{s}_t = \mathbf{f}(\mathbf{s}_t, \mathbf{c}_t)\, dt + \mathbf{g}(\mathbf{s}_t, \mathbf{c}_t)\, d\mathbf{B}_t, \tag{5.2}$$

$$\mathbf{c}_t \in \Gamma(\mathbf{s}_t), \quad \forall t \geq 0, \tag{5.3}$$

where $\mathbf{B}_t$ is an $m \times 1$ vector of independent Brownian motions, $\mathbf{f}(\mathbf{s}_t, \mathbf{c}_t) \in \mathbb{R}^n$ is the drift, and $\mathbf{g}(\mathbf{s}_t, \mathbf{c}_t) \in \mathbb{R}^{n \times m}$ is the diffusion matrix. Both the drift and the diffusion can depend on the control variables.

For most of this chapter, we focus on diffusion processes without jumps; extensions to jump processes are discussed later. The machine-learning tools introduced in Chapter 4 are particularly useful in this setting.

Interpretation. This general formulation encompasses a broad class of problems. It could describe a household maximizing lifetime utility, a firm choosing investment and financing policies, a regulator designing optimal policy, or a central bank managing welfare over time. As seen in Chapter 3, even derivative pricing problems can often be cast as optimal control problems in continuous time. The following sections present a unified approach for solving all such models.

The HJB Equation. As shown in Chapter 3, the optimal policy $\mathbf{c}(\mathbf{s})$ satisfies the Hamilton–Jacobi–Bellman (HJB) equation:

$$0 = \max_{\mathbf{c}\in\Gamma(\mathbf{s})}\Big[u(\mathbf{c}) + \nabla_{\mathbf{s}}V(\mathbf{s})^{\top}\mathbf{f}(\mathbf{s},\mathbf{c}) + \tfrac{1}{2}\operatorname{Tr}\big(\mathbf{g}(\mathbf{s},\mathbf{c})^{\top}\mathbf{H}_{\mathbf{s}}V(\mathbf{s})\,\mathbf{g}(\mathbf{s},\mathbf{c})\big)\Big], \tag{5.4}$$

where $\nabla_{\mathbf{s}}V(\mathbf{s})$ and $\mathbf{H}_{\mathbf{s}}V(\mathbf{s})$ denote, respectively, the gradient and Hessian of the value function. This equation follows from the multivariate version of Itô's lemma, applied to the stochastic process $V(\mathbf{s}_t)$. For completeness, the derivation of the multivariate Itô formula is provided in Appendix A.1.

Dealing with the Three Curses. As discussed in Chapters 1–3, dynamic programming methods face three intertwined computational obstacles– the so-called *three curses of dimensionality*: representation, optimization, and expectation. In low-dimensional settings, these challenges are manageable using the grid-based or collocation schemes introduced earlier. In higher dimensions, however, the same bottlenecks that limited discrete- and continuous-time methods reappear with full force.

(i) **Curse of representation:** as the number of state variables grows, storing and interpolating the value and policy functions on a grid becomes infeasible;

(ii) **Curse of optimization:** evaluating or solving the maximization step $\max_{\mathbf{c}}\{u(\mathbf{c})+\dots\}$ at each state point becomes increasingly expensive as the control space expands;

(iii) **Curse of expectation:** computing the expected continuation value $\mathbb{E}[V(\mathbf{s}')]$ involves integration over many stochastic dimensions, which scales exponentially in the number of shocks.

The remainder of this chapter shows how the *Deep Policy Iteration (DPI)* algorithm addresses each of these curses using modern machine-learning tools:

- Neural networks provide compact, differentiable representations of value and policy functions, circumventing the curse of representation;
- Automatic differentiation enables efficient computation of gradients and drift terms needed for the HJB, mitigating the curse of optimization;
- Stochastic optimization and simulation-based training replace explicit high-dimensional integration, alleviating the curse of expectation.

Together, these components allow DPI to solve high-dimensional continuous-time models that were previously computationally intractable.

5.2 Overcoming the Curse of Expectation: Itô's Lemma and Automatic Differentiation

One of the central challenges in solving high-dimensional dynamic programming problems is the computation of the expected continuation value. In discrete time, this requires evaluating high-dimensional integrals. In continuous time, these expectations enter the Hamilton–Jacobi–Bellman (HJB) equation through the infinitesimal generator of the value function. The key insight of continuous-time methods—and one that lies at the heart of the Deep Policy Iteration algorithm—is that these expectations can be replaced by derivatives via Itô's lemma. This allows us to transform the *curse of expectation* into a problem of efficient differentiation.

From integration to differentiation. In discrete time, the Bellman equation involves the expectation of the continuation value,

$$\mathbb{E}[V(\mathbf{s}')] = \int \cdots \int V(s + f(\mathbf{s}, \mathbf{c})\,\Delta t + g(\mathbf{s}, \mathbf{c})\,\sqrt{\Delta t}\,\mathbf{Z})\,\phi(\mathbf{Z})\,d\mathbf{Z}_1 \cdots d\mathbf{Z}_m,$$

where $\phi(\mathbf{Z})$ is the joint density of the shocks. In continuous time, the expected change in the value function can be written as

$$\mathbb{E}[dV(\mathbf{s})] = \nabla_{\mathbf{s}} V(\mathbf{s})^\top \mathbf{f}(\mathbf{s}, \mathbf{c}) + \tfrac{1}{2}\,\mathrm{Tr}\big[\mathbf{g}(\mathbf{s}, \mathbf{c})^\top \mathbf{H}_{\mathbf{s}} V(\mathbf{s}) \mathbf{g}(\mathbf{s}, \mathbf{c})\big]\,. \tag{5.5}$$

Hence, instead of computing high-dimensional integrals, we only need to compute derivatives of the value function.

Computational challenge. One could use finite differences to compute these derivatives, but this quickly becomes infeasible in higher dimensions. Suppose $n = 10$ and we use a grid of 100 points for each state variable. Then, just to store the grid in memory, we would need 10^{17} terabytes of RAM.

As discussed in Chapter 4, automatic differentiation (AD) provides an efficient way to compute derivatives by propagating local gradients through a computational graph. However, a naive use of AD can also be computationally expensive here. Computing the drift of $V(\mathbf{s})$ requires both the gradient and the Hessian matrix of $V(\mathbf{s})$, and the cost of forming the full Hessian increases rapidly with the number of state variables. Nested calls to an AD library to compute these second derivatives can be prohibitively slow.

Illustration. To see this, consider the simple example:

$$V(\mathbf{s}) = \sum_{i=1}^{n} s_i^2,$$

where s_i is the i-th state variable. Suppose there is a single shock ($m = 1$) and $n = 100$ state variables. We evaluate the drift at the point $\mathbf{s} = \mathbf{f}(\mathbf{s}) = \mathbf{g}(\mathbf{s}) = \mathbf{1}_{n\times 1}$, abstracting from controls for simplicity.

Table 5.1: Computational Cost of Numerical Derivatives

Method	FLOPs	Memory	Error
1. Finite differences	9,190,800	112,442,048	1.58%
2. Naive autodiff	2,100,501	25,673,640	0.00%
3. Analytical	20,501	44,428	0.00%
4. Hyper-dual Itô	599	6,044	0.00%

Notes: The table shows the computational cost for computing the drift of $V(s) = \sum_{i=1}^{100} s_i^2$, assuming $s_i = \mu_i = \sigma_i = 1$ for $i = 1, \ldots, 100$, using four different methods: 1) finite differences (with $h = 0.001$), 2) a naive use of automatic differentiation (where the Hessian is computed by nested calls to the Jacobian function), 3) using the analytical partial derivatives, and 4) the hyper-dual Itô's lemma method described in Proposition 5.1 combined with forward-mode automatic differentiation. The column FLOPs shows the number of floating point operations required by each approach. The column Memory is measured as bytes accessed. The column Error measures the absolute value of the relative error of each method in percentage terms.

Table 5.1 compares the number of floating-point operations (FLOPs) required to compute the drift of $V(\mathbf{s})$ using different methods. Finite differences are both memory-intensive and inaccurate, while a naive AD implementation—computing the Hessian via nested Jacobian calls—offers only limited improvement. We next introduce a more efficient method.

A hyper-dual approach to Itô's lemma. One of the key insights from Chapter 4 is that computing a *Jacobian–vector product* (JVP) is much more efficient than forming the full Jacobian. In particular, the cost of a JVP is independent of the number of state variables. In the absence of shocks, computing the drift of $V(\mathbf{s})$ amounts to evaluating a JVP:

$$\mathbb{E}[dV(\mathbf{s})] = \nabla_{\mathbf{s}} V(\mathbf{s})^\top \mathbf{f}(\mathbf{s})\, dt,$$

which can be efficiently computed using forward-mode AD.

However, when stochastic shocks are present, the drift also depends on quadratic forms involving the Hessian of $V(\mathbf{s})$. In this case, a JVP is no longer sufficient.

The idea of the *hyper-dual approach* is to extend dual numbers—the building block of forward-mode AD—to include not only the function value and tangent direction, but also the matrix of risk exposures associated with the diffusion term. We can represent a hyper-dual number as a triplet containing the value of the function, its tangent vector, and the diffusion matrix. The following proposition formalizes this idea.

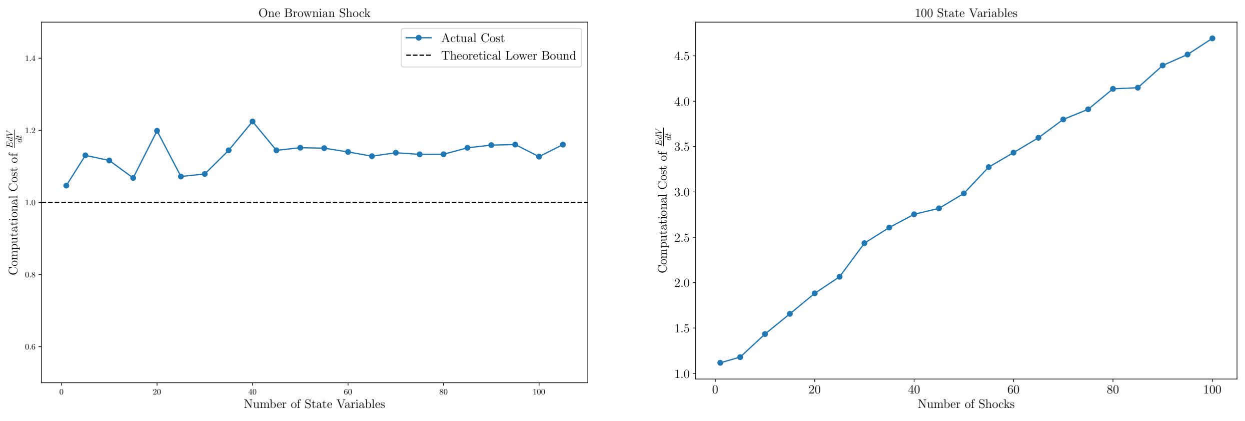
Proposition 5.1 (Hyper-dual Itô's lemma)**.** *For a given* $\mathbf{s}$*, define the auxiliary functions* $F_i : \mathbb{R} \to \mathbb{R}$ *as*

$$F_i(\epsilon; \mathbf{s}) = V\left(\mathbf{s} + \tfrac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\, \epsilon + \tfrac{\mathbf{f}(\mathbf{s})}{2m}\, \epsilon^2\right),$$

where $\mathbf{f}(\mathbf{s})$ *is the drift of* $\mathbf{s}$ *and* $\mathbf{g}_i(\mathbf{s})$ *is the i-th column of the diffusion matrix* $\mathbf{g}(\mathbf{s})$*. Then:*

1. ***Diffusion:*** *The* $1 \times m$ *diffusion matrix of* $V(\mathbf{s})$ *is*

$$\nabla V(\mathbf{s})^\top \mathbf{g}(\mathbf{s}) = \sqrt{2}\, [F_1'(0; \mathbf{s}), F_2'(0; \mathbf{s}), \ldots, F_m'(0; \mathbf{s})].$$



(a) (b)

Figure 5.1: Itô's lemma computational cost.

Notes: This figure shows how the cost of computing the drift of a function V scales with the number of state variables and Brownian shocks. Cost is measured as the execution time of $\frac{\mathbb{E}dV}{dt}(\mathbf{s})$ relative to that of $V(\mathbf{s})$. The left panel fixes $m = 1$ and varies n from 1 to 100; the right panel fixes $n = 100$ and varies m from 1 to 100. In this example, V is a two-layer neural network, and execution times are averaged over 10,000 runs on a mini-batch of 512 states.

2. ***Drift:*** *The drift of* $V(\mathbf{s})$ *is*

$$\mathcal{D}V(\mathbf{s}) = F''(0; \mathbf{s}),$$

where $F(\epsilon; \mathbf{s}) = \sum_{i=1}^{m} F_i(\epsilon; \mathbf{s})$.

Note

Automatic Itô calculus. In the same way automatic differentiation teaches the computer how to apply the rules of classical calculus, the hyper-dual approach effectively teaches it the rules of *stochastic calculus*. This enables the automatic computation of drifts and diffusions in continuous-time models. Appendix A.2.2 provides a detailed proof of Proposition 5.1.

The result in Proposition 5.1 reduces the problem of computing the drift of $V(\mathbf{s})$—which normally involves computing both a gradient and a Hessian—to evaluating the second derivative of a univariate function. Consistent with our discussion in Chapter 4, the cost of computing the drift is the same as evaluating $F(\epsilon)$, which involves evaluating $V(\mathbf{s})$ repeated m times (once per Brownian shock). Hence the computational complexity is

$$\mathcal{O}(m \times \text{cost}(V(\mathbf{s}))),$$

which is independent of the number of state variables n.

Julia implementation. It is straightforward to implement the hyper-dual approach to Itô's lemma in Julia. Listing 5.1 shows how to compute the drift of $V(\mathbf{s})$ using the `ForwardDiff.jl` package to compute the second derivative of the auxiliary function $F(\epsilon)$.

```julia
using ForwardDiff, LinearAlgebra
V(s) = sum(s.^2) # example function
n, m  = 100, 1 # number of state variables and shocks
s0, f, g = ones(n), ones(n), ones(n,m) # example values

# Analytical drift
∇f, H = 2*s0, Matrix(2.0*I, n,n) # gradient and Hessian
drift_analytical = ∇f'*f + 0.5*tr(g'*H*g) # analytical drift

# Hyper-dual approach
F(ϵ) = sum([V(s0 + g[:,i]*ϵ/sqrt(2) + f/(2m)*ϵ^2) for i = 1:m])
drift_hyper = ForwardDiff.derivative(ϵ ->
        ForwardDiff.derivative(F, ϵ), 0.0) # scalar 2nd derivative
```

Listing 5.1: Hyper-dual Itô's lemma implementation.

To verify correctness, we can compare the analytical and hyper-dual drifts:

```
Julia REPL

julia> drift_analytical, drift_hyper
(300.0, 300.0)
```

Listing 5.2: Comparison of analytical and hyper-dual drift.

These results confirm that the hyper-dual approach computes the drift exactly, matching the analytical benchmark.

Benchmark. We can benchmark the performance of the hyper-dual approach to Itô's lemma by comparing the execution time of alternative methods. Table 5.1 shows that finite differences are both memory-intensive and inaccurate, while a naive AD implementation—computing the Hessian via nested Jacobian calls—offers only limited improvement. The third row reports results using analytical partial derivatives, while the fourth row uses the hyper-dual Itô's lemma method. The hyper-dual approach proves to be the most efficient, even outperforming the case with analytical derivatives in both speed and memory usage.

Figure 5.1 illustrates how the execution time of computing the drift scales with the number of state variables and Brownian shocks. Panel (a) fixes the number of shocks at one and varies the number of state variables from 1 to 100, while Panel (b) fixes the number of state variables at 100 and varies the number of shocks from 1 to 100. In this example, V is a two-layer neural network, and execution times are averaged over 10,000 runs on a mini-batch of 512 states.

The figure shows that the execution time of the hyper-dual Itô's lemma method is roughly independent of the number of state variables, and increases linearly with the number of Brownian shocks. This stands in sharp contrast to the discrete-time approach, where the expected continuation value must be computed via numerical quadrature, whose cost grows exponentially with the number of shocks. The fact that computational cost is independent of the number of state variables and increases only linearly with the number of shocks demonstrates how this method overcomes the *curse of expectation*.

5.3 The Deep Policy Iteration Algorithm

In this section, we introduce the Deep Policy Iteration (DPI) algorithm for solving dynamic programming problems in continuous time. Our objective is to compute the value function $V(\mathbf{s})$ and policy function $\mathbf{c}(\mathbf{s})$ satisfying the coupled functional equations:

$$0 = HJB(\mathbf{s}, \mathbf{c}(\mathbf{s}), V(\mathbf{s})), \qquad \mathbf{c}(\mathbf{s}) = \arg\max_{\mathbf{c}\in\Gamma(\mathbf{s})} HJB(\mathbf{s}, \mathbf{c}, V(\mathbf{s})), \tag{5.6}$$

where

$$HJB(\mathbf{s}, \mathbf{c}, V) = u(\mathbf{c}) - \rho V + \underbrace{(\nabla_{\mathbf{s}} V)^\top \mathbf{f}(\mathbf{s}, \mathbf{c}) + \tfrac{1}{2}\operatorname{Tr}\big[\mathbf{g}(\mathbf{s}, \mathbf{c})^\top \mathbf{H}_{\mathbf{s}} V(\mathbf{s})\, \mathbf{g}(\mathbf{s}, \mathbf{c})\big]}_{F''(0;\mathbf{s},\mathbf{c})}. \tag{5.7}$$

The drift term in the HJB can be computed efficiently using the auxiliary function $F(\cdot)$ defined in Proposition 5.1.

Overcoming the curse of representation. To solve for $V(\mathbf{s})$ and $\mathbf{c}(\mathbf{s})$ numerically, we must represent them on a computer. A traditional approach is to discretize the state space and interpolate between grid points, leading to a piecewise-linear approximation. This corresponds to a parametric representation with parameters $\boldsymbol{\theta}_V$ and $\boldsymbol{\theta}_C$ for the value and policy functions, respectively: $V(\mathbf{s}_i; \boldsymbol{\theta}_V) = \boldsymbol{\theta}_{V,i}$ and $\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C) = \boldsymbol{\theta}_{C,i}$. However, as the dimensionality of the state space grows, the number of grid points explodes exponentially—an expression of the first curse of dimensionality. We observed a similar limitation for spectral methods in Chapter 3, where the number of basis coefficients also grows rapidly in multiple dimensions.

In Chapter 4, we showed that such grid-based approximations can be viewed as shallow neural networks with fixed breakpoints. Neural networks generalize this idea by learning flexible breakpoints and nonlinear combinations of basis functions. A deep neural network (DNN) can approximate complex value and policy functions with relatively few parameters, making it an effective representation even in high-dimensional settings. We therefore represent $V(\mathbf{s})$ and $\mathbf{c}(\mathbf{s})$ using DNNs parameterized by $\boldsymbol{\theta}_V$ and $\boldsymbol{\theta}_C$, respectively.

Overcoming the curse of optimization. We now turn to the challenge of training the DNNs to satisfy the functional equations above. A key difficulty lies in performing the maximization step efficiently, without resorting to costly root-finding procedures at every state point. Our approach combines *generalized policy iteration* (see, e.g., Sutton and Barto 2018) with deep function approximation, alternating between policy evaluation and policy improvement. This leads to the *Deep Policy Iteration (DPI)* algorithm.

We describe the algorithm in three stages: (i) sampling, (ii) policy improvement, and (iii) policy evaluation.

Tip

Simplifying assumptions. For clarity, we make several simplifying assumptions that can be relaxed in practice. First, we adopt plain stochastic gradient descent (SGD) for parameter updates, although any of the optimizers discussed in Chapter 4 (e.g., Adam, RMSProp) could be used instead. Second, we perform exactly one iteration of policy evaluation and policy improvement at each update. Third, we use a quadratic loss function for the policy evaluation step.

Step 1: Sampling. We begin by sampling a mini-batch of states $\{\mathbf{s}_i\}_{i=1}^{I}$ from the state space. This batch can be drawn from a uniform distribution within plausible state-space bounds, or from an estimated ergodic distribution based on previous iterations.

Step 2: Policy Improvement. The policy improvement step, represented by the second equation in Eq. (5.6), involves solving an optimization problem for every state in the mini-batch. This step can be computationally demanding and lies at the heart of the *curse of optimization.*

Note

Generalized policy iteration. In Chapter 3, we introduced the policy function iteration (PFI) algorithm, which alternates between two steps: *policy evaluation* and *policy improvement.* In the policy evaluation step, we solve for the new value function $V_{n+1}(\mathbf{s})$ given the policy $\mathbf{c}_n(\mathbf{s})$. In the policy improvement step, we solve for the new policy $\mathbf{c}_{n+1}(\mathbf{s})$ given the current value function $V_{n+1}(\mathbf{s})$. We repeat this process until convergence.
However, when the initial guess for V is far from optimal, fully solving the maximization problem at each iteration is inefficient. This motivates an *approximate policy improvement* step that performs only a single gradient-based update in the direction of improvement.

To implement this approximate step, for each state $\mathbf{s}_i$ in the mini-batch we start from the current policy estimate $\mathbf{c}_{0,i} \equiv \mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C^{j-1})$ and perform one gradient ascent step on $HJB(\mathbf{s}_i, \mathbf{c}, \boldsymbol{\theta}_V^{j-1})$:

$$\mathbf{c}_{1,i} = \mathbf{c}_{0,i} + \nabla_{\mathbf{c}} HJB(\mathbf{s}_i; \mathbf{c}_{0,i}, \boldsymbol{\theta}_V^{j-1}). \tag{5.8}$$

(The learning rate is normalized to 1 without loss of generality; see discussion below.)

We then treat $(\mathbf{s}_i, \mathbf{c}_{1,i})_{i=1}^{I}$ as a mini-batch of training data, and update the policy network so that its output $\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C)$ matches these improved controls. The corresponding loss function is a quadratic penalty:

$$\mathcal{L}(\boldsymbol{\theta}_C) = \frac{1}{2I} \sum_{i=1}^{I} \|\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C) - \mathbf{c}_{1,i}\|^2. \tag{5.9}$$

Differentiating with respect to the network parameters yields

$$\nabla_{\boldsymbol{\theta}_C} \mathcal{L}(\boldsymbol{\theta}_C) = \frac{1}{I} \sum_{i=1}^{I} \mathbf{J}_{\boldsymbol{\theta}_C} \mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C)^\top \big(\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C) - \mathbf{c}_{1,i}\big), \tag{5.10}$$

where $\mathbf{J}_{\boldsymbol{\theta}_C}\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C)$ is the Jacobian of the policy network with respect to its parameters.

Evaluating the gradient at $\boldsymbol{\theta}_C^{j-1}$ and using $\mathbf{c}_{0,i} - \mathbf{c}_{1,i} = -\nabla_{\mathbf{c}} HJB(\mathbf{s}_i; \mathbf{c}_{0,i}, \boldsymbol{\theta}_V^{j-1})$, we obtain

$$\nabla_{\boldsymbol{\theta}_C}\mathcal{L}(\boldsymbol{\theta}_C^{j-1}) = -\frac{1}{I}\sum_{i=1}^{I} \mathbf{J}_{\boldsymbol{\theta}_C}\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C^{j-1})^\top \nabla_{\mathbf{c}} HJB(\mathbf{s}_i, \mathbf{c}_{0,i}, \boldsymbol{\theta}_V^{j-1}) \tag{5.11}$$

$$= -\frac{1}{I}\sum_{i=1}^{I} \nabla_{\boldsymbol{\theta}_C} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j-1}, \boldsymbol{\theta}_V^{j-1}). \tag{5.12}$$

This leads to the following policy update:

Policy improvement step.

$$\boldsymbol{\theta}_C^{j} = \boldsymbol{\theta}_C^{j-1} + \eta_C \frac{1}{I}\sum_{i=1}^{I} \nabla_{\boldsymbol{\theta}_C} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j-1}, \boldsymbol{\theta}_V^{j-1}), \tag{5.13}$$

where η_C is the learning rate controlling the step size in parameter space. Equation (5.13) corresponds to a single gradient-ascent step on the HJB objective with respect to $\boldsymbol{\theta}_C$.

Note

Normalization. If we had introduced a learning rate η' in Eq. (5.8), its effect would simply multiply η_C in Eq. (5.13). Because only the product $\eta_C\eta'$ matters for the update, we can normalize $\eta' = 1$ without loss of generality.

Step 3: Policy Evaluation. We now update the value function given the new policy parameters $\boldsymbol{\theta}_C^{j}$. We present two alternative update rules, each with distinct trade-offs.

The first rule mirrors the iterative policy evaluation procedure in Algorithm 2. Analogous to the explicit finite-difference method in Chapter 3, we consider a *false-transient* formulation that iterates the value function backward in (pseudo-)time:

$$\frac{V(\mathbf{s}; \boldsymbol{\theta}_V^{j}) - V(\mathbf{s}_i; \boldsymbol{\theta}_V^{j-1})}{\Delta t} = HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V^{j-1}). \tag{5.14}$$

Instead of discretizing the spatial derivatives as in finite differences, we obtain them analytically through automatic differentiation of the neural network $V(\mathbf{s}; \boldsymbol{\theta}_V)$, potentially using the hyper-dual Itô method from Proposition 5.1.

Rather than iterating until convergence, we interpret this expression as defining a *target* for the next value update:

$$V(\mathbf{s}; \boldsymbol{\theta}_V^{j}) = V(\mathbf{s}_i; \boldsymbol{\theta}_V^{j-1}) + HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V^{j-1})\,\Delta t, \tag{5.15}$$

and train the value network by minimizing the quadratic loss

$$\mathcal{L}(\boldsymbol{\theta}_V) = \frac{1}{2I}\sum_{i=1}^{I}\big(V(\mathbf{s}_i; \boldsymbol{\theta}_V) - V(\mathbf{s}_i; \boldsymbol{\theta}_V^{j-1}) - HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V^{j-1})\Delta t\big)^2. \tag{5.16}$$

Evaluating the gradient at $\boldsymbol{\theta}_V^{j-1}$ yields

$$\nabla_{\boldsymbol{\theta}_V}\mathcal{L}(\boldsymbol{\theta}_V^{j-1}) = -\frac{\Delta t}{I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V^{j-1})\nabla_{\boldsymbol{\theta}_V} V(\mathbf{s}_i; \boldsymbol{\theta}_V^{j-1}), \tag{5.17}$$

and the corresponding update rule is:

Policy evaluation step 1.

$$\boldsymbol{\theta}_V^{j} = \boldsymbol{\theta}_V^{j-1} + \eta_V \frac{\Delta t}{I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V^{j-1})\nabla_{\boldsymbol{\theta}_V} V(\mathbf{s}_i; \boldsymbol{\theta}_V^{j-1}). \tag{5.18}$$

Alternatively, we can proceed as in the implicit finite-difference method, and evaluate the HJB equation at the new value function parameters $\boldsymbol{\theta}_V^{j}$:

$$\frac{V(\mathbf{s}; \boldsymbol{\theta}_V^{j}) - V(\mathbf{s}_i; \boldsymbol{\theta}_V^{j-1})}{\Delta t} = HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V^{j}). \tag{5.19}$$

We can define our loss function as minimizing the mean-squared error of the equation above. As in the implicit finite-difference method in Chapter 3, we can take the limit as $\Delta t \to \infty$, so the loss function becomes simply the mean-squared error of the HJB residuals:

$$\mathcal{L}(\boldsymbol{\theta}_V) = \frac{1}{2I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V)^2, \tag{5.20}$$

whose gradient is

$$\nabla_{\boldsymbol{\theta}_V}\mathcal{L}(\boldsymbol{\theta}_V) = \frac{1}{I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V)\nabla_{\boldsymbol{\theta}_V} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V). \tag{5.21}$$

Evaluating at $\boldsymbol{\theta}_V^{j-1}$ gives:

Policy evaluation step 2.

$$\boldsymbol{\theta}_V^{j} = \boldsymbol{\theta}_V^{j-1} - \eta_V \frac{1}{I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V^{j-1})\nabla_{\boldsymbol{\theta}_V} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j}, \boldsymbol{\theta}_V^{j-1}). \tag{5.22}$$

Tip

The trade-off between the two update rules. In the reinforcement learning literature, methods that minimize the Bellman residual directly are known to converge more stably but often more slowly than iterative policy evaluation. Moreover, Eq. (5.22) requires relatively costly third-order derivatives, since $\nabla_{\boldsymbol{\theta}_V} HJB$ involves the Hessian of V. As a rule of thumb, it is preferable to start with Eq. (5.18) for speed, and switch to

Eq. (5.22) if instability or divergence is observed.

We can now summarize the complete algorithm:

Algorithm 4: Deep Policy Iteration (DPI)

```
1 Initialize parameters $\boldsymbol{\theta}_V^0$ and $\boldsymbol{\theta}_C^0$;
2 for $j = 1, 2, \ldots$ do
3     Sample a mini-batch of states $\{\mathbf{s}_i\}_{i=1}^I$;
      // Policy improvement (actor update)
4     Update $\boldsymbol{\theta}_C$ using Eq. (5.13);
      // Policy evaluation (critic update)
5     Update $\boldsymbol{\theta}_V$ using Eq. (5.18) or Eq. (5.22);
```

Note

Actor–Critic Interpretation. The Deep Policy Iteration (DPI) algorithm mirrors the *actor–critic* architecture from reinforcement learning. The **actor** corresponds to the policy network $\mathbf{c}(\mathbf{s}; \boldsymbol{\theta}_C)$, which proposes actions (controls) given the current state. The **critic** corresponds to the value network $V(\mathbf{s}; \boldsymbol{\theta}_V)$, which evaluates those actions by estimating the value function through the HJB residual. In each iteration:

- the **actor update** (policy improvement step, Eq. (5.13)) adjusts $\boldsymbol{\theta}_C$ to increase the value estimated by the critic;

- the **critic update** (policy evaluation step, Eq. (5.18) or Eq. (5.22)) refines $\boldsymbol{\theta}_V$ so that the critic better approximates the value implied by the current policy.

This alternating structure allows DPI to learn both the optimal policy and its associated value function jointly, just as actor–critic methods do in modern reinforcement learning, but here grounded in continuous-time economic dynamic programming.

5.4 Applications

In this section, we apply the Deep Policy Iteration (DPI) algorithm to solve a variety of economic and financial problems. We consider three canonical domains of finance—asset pricing, corporate finance, and portfolio choice—to demonstrate how the DPI algorithm can be adapted to different environments. The different applications illustrates how the DPI algorithm can be applied to solve a variety of economic and financial problems

5.4.1 Asset Pricing I: The Two-Trees Model

We start with a classic asset-pricing problem—the two-trees model from Chapter 3. Although this model can be solved analytically, it provides a transparent benchmark to illustrate how the DPI algorithm operates in practice and how each of its components fits together. We then extend the model to the multi-asset setting, where the dimensionality of the problem grows rapidly.

The two-trees model. As shown in Chapter 3, the solution of the two-trees model boils down to solving a boundary-value problem for the price–consumption ratio v_t. The pricing condition for a log investor implies

$$v_t = \mathbb{E}_t \left[\int_0^\infty e^{-\rho s} s_{t+s} \, ds \right], \tag{5.23}$$

where the relative share process s_t evolves as

$$ds_t = -2\sigma^2 s_t(1-s_t)\big(s_t - \tfrac{1}{2}\big)\, dt + \sigma s_t(1-s_t)(dB_{1,t} - dB_{2,t}). \tag{5.24}$$

Since s_t is Markov, we can write $v_t = v(s_t)$. The stationary HJB equation for $v(s)$ is

$$\rho v = s - v_s \, 2\sigma^2 s(1-s)\big(s - \tfrac{1}{2}\big) + \frac{1}{2} v_{ss} \left(2\sigma^2 s^2 (1-s)^2\right), \tag{5.25}$$

subject to the boundary conditions $v(0) = 0$ and $v(1) = 1/\rho$.

Although this one-dimensional problem is straightforward to solve using finite differences or collocation methods, we solve it here using the DPI framework to illustrate the workflow.

Model setup. We start by defining a model struct that contains both the parameters and the functions for the drift and diffusion of the state variable s.

```julia
@kwdef struct TwoTrees
  ρ::Float64 = 0.04
  σ::Float64 = sqrt(0.04)
  μ::Float64 = 0.02
  μₛ::Function = s -> @. -2 * σ^2 * s * (1-s) * (s-0.5)
  σₛ::Function = s -> @. sqrt(2) * σ * s * (1-s)
end;
```

Listing 5.3: Model struct for the two-trees model.

The drift and diffusion functions use Julia's broadcast operator `@.` to apply the transformation elementwise to batched inputs. This allows the code to handle multiple state draws simultaneously, which will be useful during training.

Hyper-dual Itô's lemma. We next implement the hyper-dual approach to Itô's lemma to compute the drift of the value function efficiently.

```julia
# Hyper-dual approach to Ito's lemma
function drift_hyper(V::Function, s::AbstractMatrix, m::TwoTrees)
  F(ϵ) = V(s + m.σₛ(s)/sqrt(2)*ϵ + m.μₛ(s)/2*ϵ^2)
  ForwardDiff.derivative(ϵ->ForwardDiff.derivative(F, ϵ), 0.0)
end;
```

Listing 5.4: Hyper-dual approach to Itô's lemma.

The implementation in Listing 5.4 is remarkably compact: two lines of code suffice to compute the drift of any function $V(\mathbf{s})$ using Proposition 5.1. To validate the implementation, we test it against the analytical drift; the results match to machine precision.

```julia
# Small test: exact vs. automatic differentiation
rng = Xoshiro(0)
s   = rand(rng, 1, 1000)
# Exact drift for test function
V_test(s) = sum(s.^2, dims = 1)
drifts_exact = map(1:size(s, 2)) do i
    ∇V, H = 2 * s[:,i], 2 * Matrix(I,length(s[:,i]),length(s[:,i]))
    ∇V' * m.μₛ(s[:,i]) + 0.5 * tr(m.σₛ(s[:,i])' * H * m.σₛ(s[:,i]))
end'
drifts_hyper = drift_hyper(V_test, s, m)
errors = maximum(abs.(drifts_exact - drifts_hyper))
```

Listing 5.5: Test of the hyper-dual approach to Itô's lemma.

Neural-network representation. We represent the value function with a neural network.

```julia
### Defining the neural net
model = Chain(
  Dense(1 => 25, Lux.gelu),
  Dense(25 => 25, Lux.gelu),
  Dense(25 => 1)
)
```

Listing 5.6: Neural network for the value function.

The model summary confirms the network's structure:

Julia REPL

```
julia> model
Chain(
  layer_1 = Dense(1 => 25, gelu_tanh),          # 50 parameters
  layer_2 = Dense(25 => 25, gelu_tanh),         # 650 parameters
  layer_3 = Dense(25 => 1),                     # 26 parameters
)  # Total: 726 parameters, plus 0 states.
```

Training setup. We initialize the parameters and optimizer state using the Adam optimizer with a learning rate of 0.001.

```julia
# Initialization
ps, ls = Lux.setup(rng, model) ▷ f64   # parameters/layer states
opt = Adam(1e-3)                      # optimizer
os = Optimisers.setup(opt, ps)        # optimizer state

# Loss function
function loss_fn(ps, ls, s, target)
  return mean(abs2, model(s, ps, ls)[1] - target)
end

# Target
function target(v, s, m; Δt = 0.2)
  hjb = s + drift_hyper(v, s, m) - m.ρ * v(s)
  return v(s) + hjb * Δt
end
```

Listing 5.7: Initialization of parameters and optimizer state.

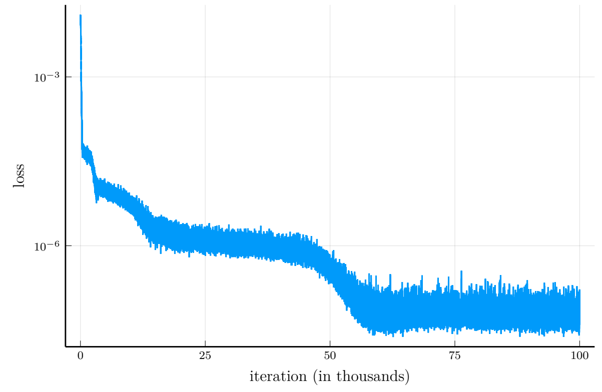
We adopt the first update rule for the policy evaluation step (Eq. (5.18)), which avoids the need for mixed-mode automatic differentiation.

Tip

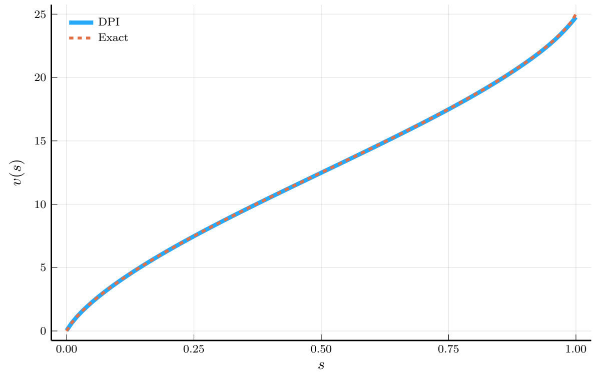
Mixed-mode automatic differentiation. Implementing the second update rule (Eq. (5.22)) typically requires mixed-mode AD: forward-mode for the second derivative of the auxiliary function in Proposition 5.1, and reverse-mode for the parameter gradients $\nabla_{\boldsymbol{\theta}_V}$. While `Zygote.jl` does not differentiate through dual-number arithmetic, a mixed approach using `Reactant.jl` can achieve this. See the documentation of `Lux.jl` for details on how to use mixed-mode AD. In the next section, we discuss an alternative approach to sidesteps the need for mixed-mode AD.

Training the network. We now train the neural network by sampling batches of states, computing the HJB residual, and minimizing the corresponding quadratic loss.

```julia
# Training loop
loss_history = Float64[]
for i = 1:40_000
    s_batch = rand(rng, 1,  128)
    tgt     = target(s-> model(s, ps, ls)[1], s_batch, m, Δt = 1.0)
    loss    = loss_fn(ps, ls, s_batch, tgt)
    grad    = gradient(p -> loss_fn(p, ls, s_batch, tgt), ps)[1]
    os, ps  = Optimisers.update(os,ps, grad)
    push!(loss_history, loss)
end
```



(a) Training loss over iterations.



(b) DPI prediction vs. analytical solution.

Figure 5.2: Training progress and fitted price–consumption ratio for the two-trees model.

Listing 5.8: Training the neural net.

Results. Figure 5.2 shows the training loss and the fitted price–consumption ratio. The network fits the analytical solution with high precision: the loss decreases smoothly over iterations, and the fitted function tracks the true $v(s)$ almost perfectly.

Important

Handling boundary conditions. Notice that we did not explicitly impose the boundary conditions during training. In this model, the PDE for the value function is *degenerate* at the boundaries: the drift and/or diffusion vanish at $s = 0$ and $s = 1$. As a result, the boundary values are endogenously pinned by the PDE itself. This is a common feature of models with heterogeneity. The DPI algorithm handles this automatically, provided that boundary states are properly represented in the training samples.

Discussion. This example demonstrates the DPI workflow in its simplest form. We used the hyper-dual Itô method to compute the drift efficiently, represented the value function with a neural network, and trained it using gradient descent. Even though the two-trees model is one-dimensional, the same structure generalizes seamlessly to high-dimensional settings with multiple states and shocks. We next illustrate this by extending the model to a multi-tree (Lucas orchard) economy.

5.4.2 Asset Pricing II: Lucas Orchard

We now extend the two-trees model to a multi-tree economy, known as the *Lucas orchard model* (Martin 2013). By varying the number of trees, we can examine how the DPI algorithm scales with the dimensionality of the state space and compare its performance with existing numerical methods in the literature.

The model. Consider a representative investor with log utility who can invest in a riskless asset and N risky assets. Each risky asset i pays a continuous dividend stream $D_{i,t}$ that follows a geometric Brownian motion:

$$\frac{dD_{i,t}}{D_{i,t}} = \mu_i\, dt + \sigma_i\, dB_{i,t}, \qquad i = 1, 2, \ldots, N, \tag{5.26}$$

where each $B_{i,t}$ is a Brownian motion satisfying $dB_{i,t}\, dB_{j,t} = 0$ for $i \neq j$.

As in the two-trees model, it is convenient to express the system in terms of the *dividend shares*

$$s_{i,t} = \frac{D_{i,t}}{C_t}, \qquad C_t = \sum_{i=1}^{N} D_{i,t},$$

where C_t is aggregate consumption. Appendix A.3 derives the law of motion for the vector of shares $\mathbf{s}_t = (s_{1,t}, \ldots, s_{N,t})^\top$:

$$d\mathbf{s}_t = \boldsymbol{\mu}_s(\mathbf{s}_t)\, dt + \boldsymbol{\sigma}_s(\mathbf{s}_t)\, d\mathbf{B}_t, \tag{5.27}$$

where $\boldsymbol{\mu}_s(\mathbf{s}_t)$ and $\boldsymbol{\sigma}_s(\mathbf{s}_t)$ are the drift and diffusion of the vector of dividend shares, respectively.

Valuation. Let $v_{i,t} \equiv P_{i,t}/C_t$ denote the price–consumption ratio of asset i. The pricing condition for a log investor is analogous to the two-trees case:

$$v_{i,t} = \mathbb{E}_t\left[\int_0^\infty e^{-\rho s} s_{i,t+s}\, ds\right]. \tag{5.28}$$

Because the process for each share $s_{i,t}$ depends on the entire vector $\mathbf{s}_t$, the price–consumption ratio for asset i must be a function of all dividend shares, $v_{i,t} = v_i(\mathbf{s}_t)$.

HJB equation. The stationary HJB equation for $v_i(\mathbf{s})$ is

$$\rho\, v_i(\mathbf{s}) = s_i + \nabla_{\mathbf{s}} v_i(\mathbf{s})^\top \boldsymbol{\mu}_s(\mathbf{s}) + \frac{1}{2}\operatorname{Tr}\left[\boldsymbol{\sigma}_s(\mathbf{s})^\top \mathbf{H}_{\mathbf{s}} v_i(\mathbf{s}) \boldsymbol{\sigma}_s(\mathbf{s})\right], \tag{5.29}$$

subject to the boundary conditions $v_i(0) = 0$ and $v_i(1) = 1/\rho$.

Dimensionality of the state space. The state vector $\mathbf{s}$ lies in the $(N-1)$-dimensional simplex:

$$\mathcal{S} = \left\{ \mathbf{s} \in [0,1]^N \;:\; \sum_{i=1}^{N} s_i = 1 \right\}.$$

In principle, one could eliminate one of the shares—for instance, $s_1 = 1 - \sum_{i=2}^{N} s_i$—to reduce the number of state variables to $N-1$. In practice, however, the DPI algorithm can handle high-dimensional state spaces directly, since the neural-network representation of $v_i(\mathbf{s})$ does not rely on a tensor grid. Hence, it is often simpler and computationally convenient to retain all N shares as independent state variables. The resulting redundancy does not affect numerical stability and has negligible cost.

Julia implementation. We now implement the Lucas orchard model in Julia. The workflow of the DPI algorithm for the Lucas orchard model is virtually identical to that for the simple two-trees model.

As usual, we start by defining a model struct that contains both the parameters and the functions for the drift and diffusion of the state variable s.

```julia
@kwdef struct LucasOrchard
    ρ::Float64 = 0.04
    N::Int = 10
    σ::Vector{Float64} = sqrt(0.04) * ones(N)
    μ::Vector{Float64} = 0.02 * ones(N)
    μc::Function = s -> μ' * s
    σc::Function = s -> [s[i,:]' * σ[i] for i in 1:N]
    μₛ::Function = s ->  s .* (μ .- μc(s)- s.*σ.^2 .+
        sum(σc(s)[i].^2 for i in 1:N)) # drift of s
    σₛ::Function = s -> [s .* ([j == i ? σ[i] : 0 for j in 1:N] .-
        σc(s)[i]) for i in 1:N] # diffusion of s
end;
```

Listing 5.9: Model struct for the Lucas orchard model.

The drift and diffusion functions are written using Julia's broadcast operator `@.`, which efficiently applies operations elementwise over batched inputs. For a mini-batch of size B, the input $\mathbf{s}$ is an $N \times B$ matrix, where N is the number of assets and each column corresponds to a draw from the Dirichlet distribution. The drift function returns an $N \times B$ matrix of the same shape, while the diffusion function returns a vector of length N, where each element is an $N \times B$ matrix representing the exposures to the corresponding Brownian motion.

Listing 5.10 shows how to instantiate the model.

```julia
1  # Instantiate the model
2  m           = LucasOrchard(N = 10) # number of assets
3  rng, d      = MersenneTwister(0), Dirichlet(ones(m.N))
4  s_samples   = rand(rng, d, 1_000) # N x 1_000 matrix
5  vcat(m.μₛ(s_samples), m.σₛ(s_samples)...) # N*(N+1) x 1_000 matrix
```

Listing 5.10: Instantiation of the Lucas orchard model.

The vector of state variables $\mathbf{s}$ is sampled from the Dirichlet distribution, as $\mathbf{s}$ is non-negative and sums to one. Line 5 stacks the drift and diffusion of the state variable s into a single matrix of size $N(N+1) \times B$, allowing us to visually inspect the resulting dynamics.

Tip

Dirichlet distribution. The Dirichlet distribution is a probability distribution over the simplex, that is, the set of all vectors $\mathbf{x}$ of non-negative real numbers that sum to one. It generalizes the beta distribution to multiple dimensions, with pdf

$$f(\mathbf{x}; \boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{i=1}^{N} x_i^{\alpha_i - 1},$$

where $B(\boldsymbol{\alpha})$ is the multivariate beta function. By varying the concentration parameters $\boldsymbol{\alpha}$, we can control how dispersed or concentrated the draws are across the simplex. In the Lucas orchard model, using a symmetric Dirichlet with $\alpha_i = 1$ generates uniform coverage of the state space.

Hyper-dual Itô's lemma. We next implement the hyper-dual approach to Itô's lemma to compute the drift of the value function.

```julia
# Hyper-dual approach to Ito's lemma
function drift_hyper(V::Function, s::AbstractMatrix,
    m::LucasOrchard)
  N, σs, μs = m.N, m.σₛ(s), m.μₛ(s) # Preallocations
    F(ϵ) = sum(V(s .+ σs[i] .* (ϵ / sqrt(2)) .+
        μs .* (ϵ^2 / (2 * N))) for i in 1:N)
  return ForwardDiff.derivative(ϵ ->
        ForwardDiff.derivative(F, ϵ), 0.0)
end
```

Listing 5.11: Hyper-dual approach to Itô's lemma.

The implementation in Listing 5.11 for the Lucas orchard model is virtually identical to the implementation in Listing 5.4 for the two-trees model, but now we are dealing with the case of multiple state variables and Brownian motions.

Tip

The importance of preallocations. Relative to the two-trees model, we now preallocate arrays for the drift and diffusion of the state variable $\mathbf{s}$, rather than constructing them inside loops over $i = 1, \ldots, N$. In a low-dimensional example, this difference is negligible, but in higher-dimensional problems (large N or large batch size B), preallocations avoid repeated memory allocation and garbage collection, which can otherwise dominate runtime in tight training loops.

Having implemented the drift computation efficiently, we are now ready to embed it within the DPI algorithm to train the neural networks that approximate the value functions $v_i(\mathbf{s})$ for each asset.

Neural-network representation. Next, we represent the value function with a neural network.

```julia
### Defining the neural net
model = Chain(
  Dense(10 => 25, Lux.gelu),
  Dense(25 => 25, Lux.gelu),
  Dense(25 => 1)
)
```

Listing 5.12: Neural network for the value function.

We use essentially the same architecture as in the two-trees model, but now the input is the 10-dimensional vector of state variables **s** representing the dividend shares of each asset. The network has two hidden layers with 25 units each and GELU activations. The output is a scalar corresponding to the value of the first tree.

The model summary confirms the network's structure:

Julia REPL

```
julia> model
Chain(
    layer_1 = Dense(10 => 25, gelu_tanh),      # 275 parameters
    layer_2 = Dense(25 => 25, gelu_tanh),      # 650 parameters
    layer_3 = Dense(25 => 1),                  # 26 parameters
)         # Total: 951 parameters,
          #        plus 0 states.
```

Training setup. We initialize the parameters and optimizer state using the Adam optimizer with a learning rate of 10^{-3}. The `loss_fn` function computes the mean squared deviation between the model's prediction and the target, while the `target` function constructs the target value according to the false-transient update rule from Equation (5.18). Because we are focusing on the value of the first tree, the first term in line 14 selects the first component of the state vector **s**.

```julia
1  # Initializaion
2  ps, ls = Lux.setup(rng, model) |> f64
3  opt = Adam(1e-3)
4  os = Optimisers.setup(opt, ps)
5  
6  # Loss function
7  function loss_fn(ps, ls, s, target)
8    return mean(abs2, model(s, ps, ls)[1] - target)
9  end
10 
11 # Target
12 function target(v, s, m; Δt = 0.2)
```

```julia
    v̄  = v(s)
    hjb = s[1,:]' + drift_hyper(v, s, m) - m.ρ * v̄
    return v̄  + hjb * Δt, mean(abs2, hjb)
end
```

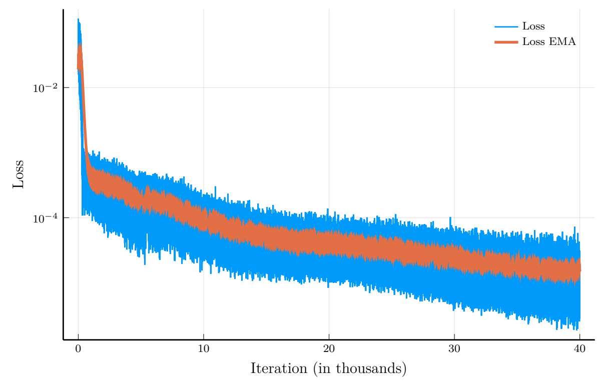
Listing 5.13: Initialization of parameters and optimizer state.

Training the network. We now train the neural network by sampling batches of states, computing the HJB residual, and minimizing the corresponding quadratic loss.

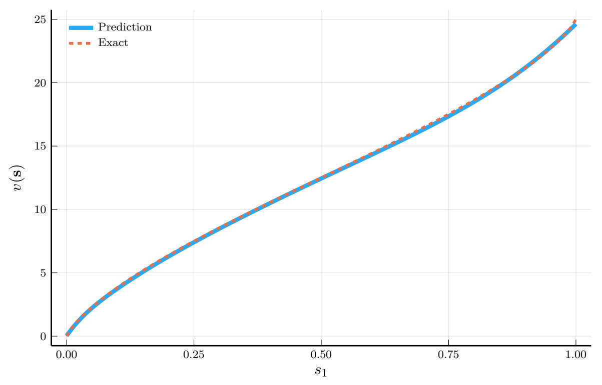
```julia
# Training parameters
max_iter, Δt = 40_000, 1.0
# Sampling interior and boundary states
d_int  = Dirichlet(ones(m.N))            # Interior region
d_edge = Dirichlet(0.05 .* ones(m.N))  # Boundary region
# Loss history and exponential moving average loss
loss_history, loss_ema_history, α_ema = Float64[], Float64[], 0.99
# Training loop
p = Progress(max_iter; desc="Training...", dt=1.0) #progress bar
for i = 1:max_iter
    if rand(rng) < 0.50
        s_batch = rand(rng, d_int, 128)
    else
        s_batch = rand(rng, d_edge, 128)
    end
    v(s)       = model(s, ps, ls)[1] # define value function
    tgt, hjb_res = target(v, s_batch, m, Δt = Δt) #target/residual
    loss, back = Zygote.pullback(p -> loss_fn(p,ls,s_batch,tgt), ps)
    grad       = first(back(1.0)) # gradient
    os, ps     = Optimisers.update(os,ps, grad) # update parameters
    loss_ema   = i==1 ? loss : α_ema*loss_ema + (1.0-α_ema)*loss
    push!(loss_history, loss)
    push!(loss_ema_history, loss_ema)
    next!(p, showvalues = [(:iter, i),("Loss", loss),
        ("Loss EMA", loss_ema), ("HJB residual", hjb_res)])
end
```

Listing 5.14: Training the neural net.

After defining the training parameters, we specify the state distributions from which we sample the training data. We use two Dirichlet distributions: one for the interior region and one for the boundary region of the simplex. For the interior region, we use a Dirichlet distribution with all parameters equal to one, which corresponds to the uniform distribution over the simplex. For the boundary region, we use a Dirichlet distribution with all parameters equal to 0.05, which is highly concentrated near the edges of the simplex. At each iteration,



(a) Training loss.



(b) Two-trees prediction.

Figure 5.3: Lucas orchard training and two-trees special case.

we sample from the interior region with probability 0.5 and from the boundary region with probability 0.5. This ensures that the model learns both the interior dynamics and the boundary behavior, which are critical for stability in degenerate PDEs.

To monitor progress, we use the `ProgressMeter.jl` package to display a live progress bar. The progress bar reports the iteration count, the instantaneous loss, its exponential moving average, and the HJB residual. Apart from the sampling and progress display, the overall training loop is nearly identical to the one used for the two-trees model in Section 5.4.1.

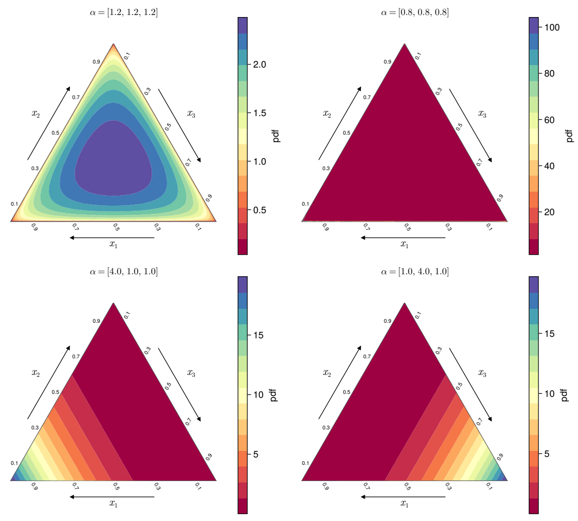
Panel (a) of Figure 5.3 shows the evolution of the training loss and its exponential moving average. The loss decreases smoothly over iterations, while the moving average tracks it closely, confirming stable convergence. The loss in Figure 5.3 is computed on the training set. To evaluate generalization, we next assess model performance on held-out test sets, and analyze how the computational cost and accuracy scale with the number of trees N.

Test sets. We next evaluate the model's performance on out-of-sample test sets.

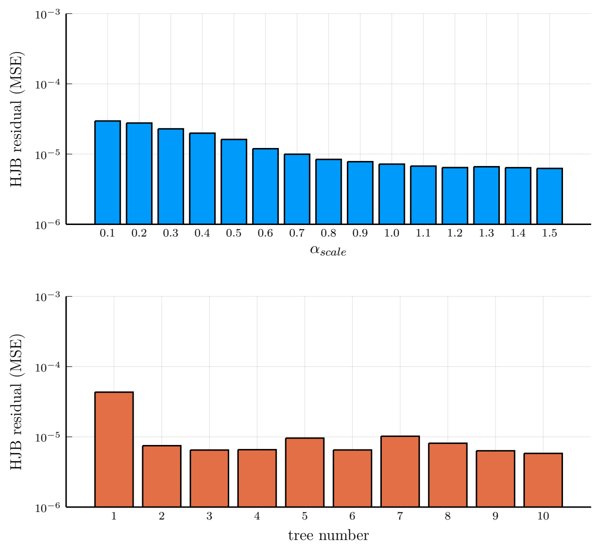
Our first test set is the two-trees special case. This corresponds to an extremely asymmetric configuration of the state vector $\mathbf{s} = (s_1, 1 - s_1, 0, \dots, 0)$, which lies outside the region used for training. Although the Lucas orchard model in general has no closed-form analytical solution, for this specific configuration the model should reproduce the price–consumption ratio from the two-trees model. Panel (b) of Figure 5.3 confirms that the network's prediction coincides almost perfectly with the analytical benchmark, illustrating the model's ability to generalize beyond the training data.

Our second test set draws states from a symmetric Dirichlet distribution with parameters $\boldsymbol{\alpha} = \alpha_{\text{scale}}(1, 1, \dots, 1)$, where α_{scale} controls the concentration of points within the simplex. Panel (a) of Figure 5.4 illustrates the corresponding Dirichlet densities for different values of α_{scale}. When $\alpha_{\text{scale}} > 1$, samples are concentrated near the center of the simplex; when $\alpha_{\text{scale}} < 1$, samples are concentrated near the edges. The top panel of Figure 5.4(b) shows the mean-squared error (MSE) of the HJB residuals as α_{scale} ranges from 0.1 to 1.5. The residuals remain uniformly small across all regions of the simplex, with slightly better performance when points cluster near the center.

Our final test set draws from an asymmetric Dirichlet distribution where the j-th element



(a) Dirichlet densities.



(b) Orchard residuals.

Figure 5.4: Dirichlet simplex visualizations and Lucas orchard residual diagnostics.

of $\boldsymbol{\alpha}_j$ equals 4.0 and the remaining elements equal 1.0. This shifts mass toward one vertex of the simplex, allowing us to test the network's performance near highly skewed states. The bottom two plots of Figure 5.4(a) display the resulting densities for the first and second tree, respectively, while the bottom panel of Figure 5.4(b) reports the HJB residual MSE for $j = 1, \ldots, 10$. Residuals remain low for all trees, confirming that the network generalizes well to asymmetric configurations of the state space.

Overall, these tests demonstrate that the DPI-trained neural network achieves excellent accuracy and generalization across a wide range of states.

Comparison with other methods. We next compare the DPI algorithm with classical numerical methods for solving high-dimensional models, using the Lucas orchard economy as a benchmark. In particular, we focus on the performance of each method as we increase the number of trees—that is, the dimensionality of the state space.

Finite-difference schemes become computationally infeasible beyond a few dimensions, and Chebyshev collocation on full tensor-product grids also suffers from exponential growth in cost. We therefore compare the DPI algorithm to a sparse-grid version of the Chebyshev collocation method, known as the *Smolyak method* (Smolyak 1963).

Note

The Smolyak Sparse Grid Method. The Smolyak method is a sparse-grid technique for approximating multivariate functions with high accuracy while mitigating the exponential growth in computational cost that arises with tensor-product grids. Originally proposed by Smolyak (1963), it combines one-dimensional interpolation or quadrature formulas of varying precision to construct a multi-dimensional approximation that is

both adaptive and efficient.
The key idea is to build a d-dimensional interpolant not on the full tensor grid (which scales as S^d for S points per dimension) but on a carefully selected subset of grid points:

$$\mathcal{A}(q,d) = \sum_{|\boldsymbol{i}| \le q+d-1} \left(\Delta_{i_1} \otimes \cdots \otimes \Delta_{i_d}\right),$$

where q controls the order (or "level") of the approximation and Δ_{i_j} denotes the incremental contribution of the i_j-th one-dimensional rule. As q increases, the approximation becomes more accurate, but the number of grid points grows only polynomially in d rather than exponentially.
In economics and finance, the Smolyak method has become one of the workhorses for solving high-dimensional dynamic models (Judd et al. 2014; Brumm and Scheidegger 2017). It is commonly used with Chebyshev or Clenshaw–Curtis nodes to approximate value or policy functions, and with quadrature rules to approximate expectations.
Despite its efficiency relative to full tensor grids, the Smolyak method still faces practical limitations:

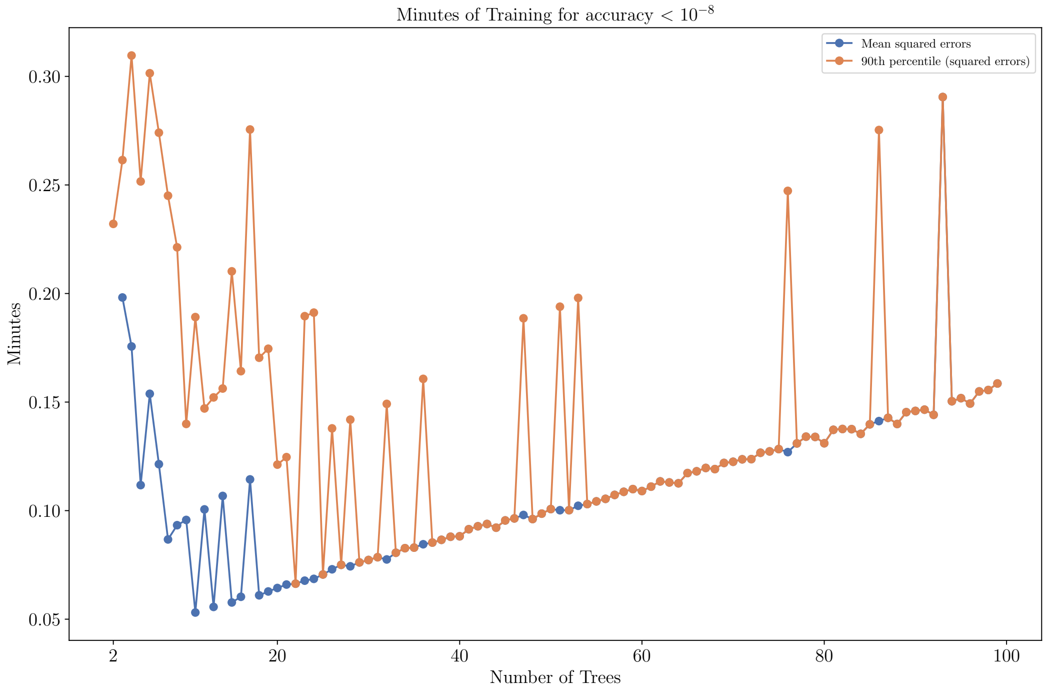
- The number of grid points grows rapidly with the approximation order q and the dimension d;
- The resulting system of equations can become ill-conditioned, especially for high-order polynomials;
- Sparse-grid interpolation can be difficult to parallelize efficiently in very high-dimensional settings.

By contrast, the DPI algorithm replaces the global polynomial approximation with a neural-network representation that adapts its capacity to the local complexity of the value function and scales linearly with the number of parameters rather than exponentially with the dimension of the state space.

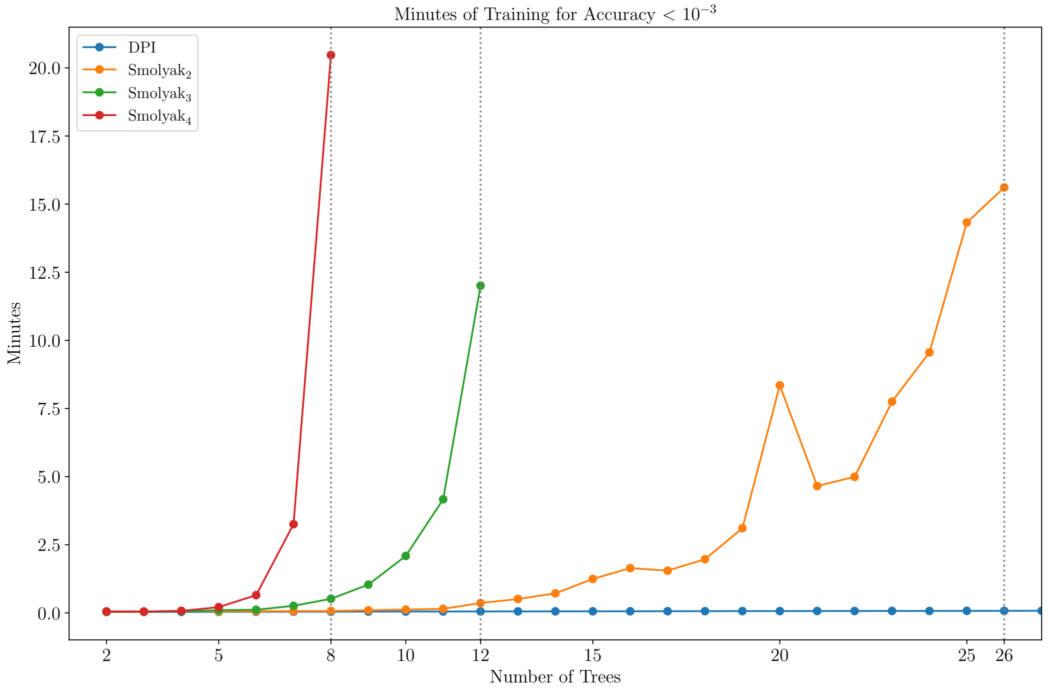
To assess performance, we solve the Lucas orchard model for an increasing number of trees. For each economy, we measure the time-to-solution until the mean-squared error (MSE) of the HJB residuals falls below 10^{-8}. As a stricter criterion, we also record the time required for the 90th percentile of the squared errors to fall below 10^{-8}.

Panel (a) of Figure 5.5 reports the results. The DPI method achieves accurate solutions extremely quickly, even in high-dimensional settings. Increasing the number of trees has only a modest effect on computational cost. For instance, in an economy with 100 trees, the DPI algorithm reaches an MSE of 10^{-8} in less than one minute.

Figure 5.5: Accuracy and Time-to-Solution in a Lucas Orchard Economy



(a) Time to solution.



(b) Smolyak methods and DPI algorithm MSEs.

Notes. Panel (a) shows the time-to-solution of the DPI algorithm, measured by the number of minutes required for the MSE or 90th-percentile squared error to fall below 10^{-8}. Panel (b) compares the time-to-solution of the DPI method and the Smolyak methods of orders 2, 3, and 4. The tolerance is set to 10^{-3}, the highest accuracy threshold reached by all Smolyak variants. The parameter values are $\rho = 0.04$, $\gamma = 1$, $\varrho = 0.0$, $\mu = 0.015$, and $\sigma = 0.1$. The HJB residuals are computed on a random sample of 2^{13} points from the state space.

Panel (b) of Figure 5.5 compares the time-to-solution of the DPI algorithm with the Smolyak methods of orders 2, 3, and 4. The Smolyak grids are solved using a conjugate-gradient method with a tolerance of 10^{-3}. The computational burden of the Smolyak approach rises sharply with both dimensionality and approximation order, and memory limits are reached for $N = 8$, 12, and 26 trees at orders 2, 3, and 4, respectively. In contrast, the DPI method maintains high accuracy and short solution times even for economies with more than 100 trees. This illustrates how DPI effectively overcomes the curse of dimensionality that constrains traditional sparse-grid methods.

5.4.3 Corporate Finance: Hennessy and Whited (2007)

We now apply the DPI algorithm to a corporate finance problem, a simplified version of the model in Hennessy and Whited (2007). This problem illustrates how the method can handle dynamic optimization with kinks in the value function and inaction regions in the optimal policy.

Model setup. Consider a firm with operating profits $\pi(k_t, z_t) = e^{z_t} k_t^{\alpha}$, where k_t denotes the capital stock and z_t the firm's log productivity. Log productivity follows an Ornstein–Uhlenbeck process:

$$dz_t = -\theta(z_t - \bar{z})\,dt + \sigma\,dB_t, \qquad \theta, \sigma > 0. \tag{5.30}$$

Given an investment rate i_t and depreciation rate δ, capital evolves as

$$dk_t = (i_t - \delta)\,k_t\,dt. \tag{5.31}$$

The state vector $\mathbf{s}_t = (k_t, z_t)^{\top}$ thus follows

$$d\mathbf{s}_t = \boldsymbol{\mu}_s(\mathbf{s}_t, i_t)\,dt + \boldsymbol{\sigma}_s(\mathbf{s}_t, i_t)\,dB_t, \tag{5.32}$$

with drift and diffusion

$$\boldsymbol{\mu}_s(\mathbf{s}_t, i_t) = \begin{bmatrix} (i_t - \delta)k_t \\ -\theta(z_t - \bar{z}) \end{bmatrix}, \qquad \boldsymbol{\sigma}_s(\mathbf{s}_t, i_t) = \begin{bmatrix} 0 \\ \sigma \end{bmatrix}.$$

The firm faces quadratic adjustment costs $\Lambda(k_t, i_t) = \frac{1}{2}\chi k_t i_t^2$ and linear equity issuance costs $\lambda > 0$. Operating profits net of adjustment costs are

$$D^*(k, z, i) = e^z k^{\alpha} - \left(i + \tfrac{1}{2}\chi i^2\right) k. \tag{5.33}$$

If net operating profits are negative, the firm issues equity to cover the shortfall, incurring the cost λ. Dividends are therefore given by

$$D(k, z, i) = \begin{cases} D^*(k, z, i), & D^*(k, z, i) \geq 0, \\ D^*(k, z, i)(1 + \lambda), & D^*(k, z, i) < 0. \end{cases} \tag{5.34}$$

The firm chooses the investment rate i_t to maximize the expected discounted value of future dividends:

$$v(\mathbf{s}_0) = \max_{\{i_t\}_{t \geq 0}} \mathbb{E}\left[\int_0^{\infty} e^{-\rho s} D(k_s, z_s, i_s)\,ds\right], \tag{5.35}$$

subject to (5.32).

The HJB equation. Equation (5.35) gives the sequential formulation of the firm's problem. In its recursive form, the value function $v(\mathbf{s})$ and policy function $i(\mathbf{s})$ satisfy

$$0 = \max_i \mathrm{HJB}(\mathbf{s}, i, v(\mathbf{s})), \tag{5.36}$$

where

$$\mathrm{HJB}(\mathbf{s}, i, v) = D(k, z, i) + \nabla v^\top \boldsymbol{\mu}_s(\mathbf{s}, i) + \tfrac{1}{2}\, \boldsymbol{\sigma}_s(\mathbf{s}, i)^\top \mathbf{H}v\, \boldsymbol{\sigma}_s(\mathbf{s}, i) - \rho v. \tag{5.37}$$

The first-order condition for the optimal investment rate is

$$\frac{\partial \mathrm{HJB}}{\partial i} = -\big(1 + \lambda\, \mathbf{1}_{D^*(k,z,i)<0}\big)\big[1 + \chi i\big] k + v_k(\mathbf{s}) k = 0. \tag{5.38}$$

When the shadow value of capital $v_k(\mathbf{s})$ equals the marginal adjustment cost, the firm is indifferent to adjusting capital. Because of the issuance cost λ, the value function exhibits a kink at this point, producing an inaction region where the firm finds it optimal neither to pay dividends nor to issue equity. In Appendix A.4, we provide an explicit characterization of the optimal investment policy for this version of the Hennessy and Whited (2007) model in terms of the derivatives of the value function. Below, we show how the DPI algorithm can recover the solution directly, even when solving for the policy variable analytically is infeasible or cumbersome.

A special case. To gain intuition about the model's behavior, it is useful to begin with a simple special case. We assume that productivity is constant, so $\theta = \sigma = 0$, and that investment adjustment costs are prohibitively high, so $\chi \to \infty$. This implies that it is optimal to keep the capital stock constant, $i(k, z) = \delta$. From the HJB equation, we obtain the value function:

$$v(k, z) = \frac{D(k, z, \delta)}{\rho} = \begin{cases} \frac{e^z k^\alpha - \delta k}{\rho}, & \text{if } k \leq k_{\max}(z), \\ \frac{e^z k^\alpha - \delta k}{\rho}(1 + \lambda), & \text{if } k > k_{\max}(z), \end{cases} \tag{5.39}$$

where $k_{\max}(z) = \big(\frac{e^z}{\delta}\big)^{\frac{1}{1-\alpha}}$.

At $k = k_{\max}(z)$, the firm switches from internal to external financing, so the equity issuance cost λ becomes binding. This creates a kink in the value function $v(k, z)$, corresponding to the boundary between the internal- and external-finance regions. Equation (5.39) therefore defines a smooth function of the state variables, except at $k = k_{\max}(z)$, where its derivative is discontinuous.

These properties are important when choosing the neural network architecture. A standard ReLU activation function can approximate the kink well but produces piecewise-linear derivatives, making it difficult to maintain smoothness elsewhere. Conversely, a GELU activation yields smooth derivatives but tends to blur sharp discontinuities. To capture both behaviors, we introduce a *shifted ReLU* activation function:

$$\mathrm{ShiftedReLU}(x; c) = \max(0,\, x - c), \tag{5.40}$$

where c is a trainable parameter that shifts the activation threshold. By learning the shift parameter c, the neural network can capture the discontinuity in the derivative at $k = k_{\max}(z)$ while remaining smooth elsewhere.

Listing 5.15 shows how to implement the shifted ReLU activation function in `Lux.jl`.

```julia
# Shifted ReLU activation function
struct ShiftedReLU <: Lux.AbstractLuxLayer end
Lux.initialparameters(::Random.AbstractRNG, ::ShiftedReLU) =
    (c = [0.0f0],)
function (m::ShiftedReLU)(x, ps, st)
    c = ps.c[1]
    return max.(0, x .- c), st
end

v_core = Chain(
    Dense(2, 24, Lux.gelu),
    Dense(24, 12, Lux.gelu),
    ShiftedReLU(),
    Dense(12, 1)
)

# Enforce boundary condition: V(0, z) = 0
v_net(s, θᵥ) =  s[1:1, :].^m.α .* v_core(s, θᵥ, stᵥ)[1]
```

Listing 5.15: Shifted ReLU activation function.

We first create a struct to store the layer parameters and define the forward pass of the activation. Having defined the layer, we can include it in the neural network as a standard `Lux` component. Listing 5.15 also shows how to enforce the boundary condition $v(0, z) = 0$ by defining a wrapper function that multiplies the network output by a term that vanishes at $k = 0$. A convenient choice is k^α, which not only enforces the boundary condition but also captures the asymptotic behavior of the value function as $k \to 0$, where $v_k(k, z)$ becomes unbounded.

Policy evaluation. In this special case, the policy function is known and constant, $i(k, z) = \delta$, so there is no policy improvement step. We therefore solve for the value function directly using the DPI algorithm.

We start by defining the model struct. It contains the model default parameters and the functions for the drift and diffusion of the state vector.

```julia
# Model parameters
@kwdef struct HennessyWhited
    α::Float64       = 0.55
    θ::Float64       = 0.26
    z̄ ::Float64       = -2.2976
    σz::Float64      = 0.123
    δ::Float64       = 0.1
    χ::Float64       = 20.0
    λ::Float64       = 0.059
    ρ::Float64        = 0.04
    kmax::Float64    = 2.5
    μₛ::Function = (s,i) -> vcat((i .-δ) .* s[1,:]',  # μk
                               -θ.*(s[2,:] .- z̄)')  # μz
    σₛ::Function = (s,i) -> vcat(zeros(1,size(s,2)),  # σk
                              σz*ones(1,size(s,2))) # σz
end;
```

Listing 5.16: Model struct.

Listing 5.17 uses the hyper-dual approach to Itô's lemma to compute the drift of the value function. It also defines the target value function and the loss function for this task.

```
# Hyper-dual approach to Ito's lemma
function drift_hyper(s::AbstractMatrix, m::HennessyWhited, θᵥ)
    ī       = m.δ * ones(1, size(s,2))
    μₛ, σₛ  = m.μₛ(s,ī ), m.σₛ(s,ī )
    F(ϵ)     = v_net(s .+ σₛ .* (ϵ / sqrt(2f0)) .+
                          μₛ .* (ϵ^2 / (2f0)), θᵥ)
  return ForwardDiff.derivative(ϵ ->
            ForwardDiff.derivative(F,ϵ),0.0f0)
end

function dividends(s, m::HennessyWhited)
    k = @view s[1,:]; z = @view s[2,:]
    π = (exp.(z) .* k.^m.α)'
    D_star = π .-  m.δ * k' # i = δ
    return D_star .* (1 .+ m.λ * (D_star .< 0))
end

function hjb_residual(s, m::HennessyWhited, θᵥ)
    D, drift = dividends(s, m), drift_hyper(s, m, θᵥ)
    return D .+ drift .- m.ρ .* v_net(s, θᵥ)
end

function target(s, m, θᵥ; Δtᵥ = 0.2f0)
    hjb      = hjb_residual(s, m, θᵥ)
    tgtᵥ     = v_net(s, θᵥ) + hjb * Δtᵥ
    return tgtᵥ, mean(abs2, hjb)
end

function loss_fn(dnn, θ, s, target)
    return mean(abs2, dnn(s, θ) - target)
end
```

Listing 5.17: Target value function for the special case.

We then train the neural network using the loop in Listing 5.18. The model parameters and neural network architecture follow the same setup as in the previous section. We first define the hyperparameters and specify the sampling of the state space. Each iteration of the training loop consists of sampling a mini-batch of states, computing the target value using the HJB equation, and updating the neural network parameters using the Adam optimizer.

```julia
# Initializaion:
rng         = Xoshiro(1234)
θᵥ, stᵥ     = Lux.setup(rng, v_core)
optᵥ        = Optimisers.Adam(1e-3, (0.9, 0.98))
osᵥ         = Optimisers.setup(optᵥ, θᵥ)

max_iter, Δtᵥ  = 200_000, 0.1
d_k, d_z       = Uniform(0.01, m.kmax), Normal(m.z̄ , m.σz/sqrt(2m.θ))

p = Progress(max_iter; desc = "Training...", dt = 1.0)
for it in 1:max_iter
    # Sample states
    k_batch = rand(rng, d_k, 100)'
    z_batch = rand(rng, d_z, 100)'
    s_batch = vcat(k_batch, z_batch)
    # Computing targets and policy evaluation step
    tgt, hjb_res = target(s_batch, m, θᵥ, Δtᵥ = Δtᵥ)
    loss, back = Zygote.pullback(p->loss_fn(v_net,p,s_batch,tgt),θᵥ)
    grad       = first(back(1.0))
    osᵥ, θᵥ     = Optimisers.update(osᵥ,θᵥ, grad)
    # Progress bar
    next!(p, showvalues = [(:iter, it),("Loss_v", lossᵥ),
                           ("HJB residual", hjb_res)])
end
```

Listing 5.18: Training the neural network for the special case.

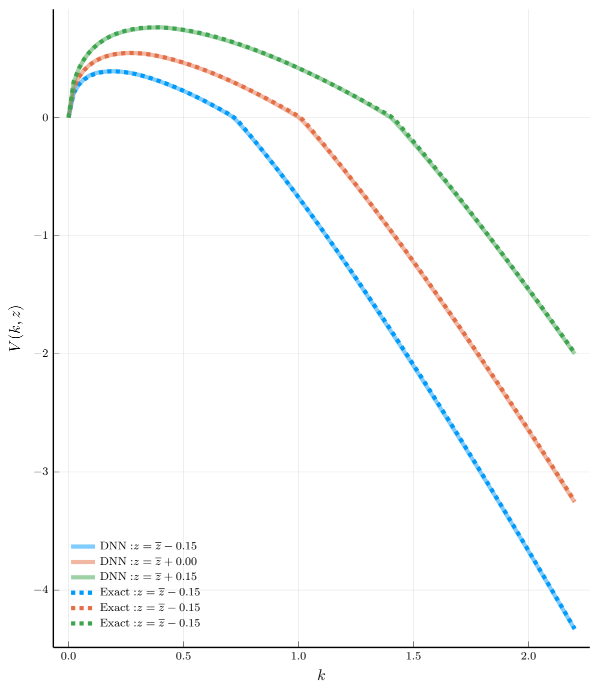
Tip

The Ornstein-Uhlenbeck process. When sampling for log productivity, we draw from the *unconditional distribution* for z_t, that is, its long-run distribution. To derive this distribution, notice we can write z_t in terms of the following stochastic integral:

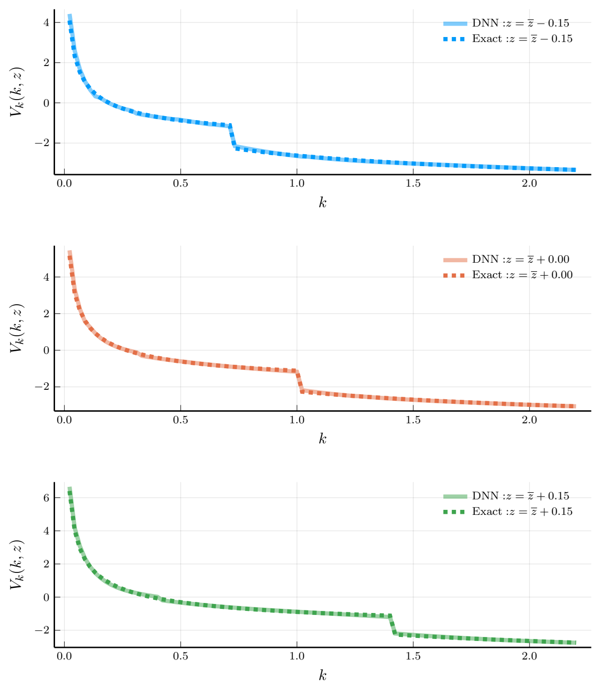
$$z_t = \bar{z} + (z_0 - \bar{z})e^{-\theta t} + \sigma \int_0^t e^{-\theta(t-s)} dB_s,$$

Using Itô's isometry, we can show that the distribution of z_t is

$$z_t \sim \mathcal{N}\left(\bar{z} + (z_0 - \bar{z})e^{-\theta t}, \frac{\sigma^2}{2\theta}(1 - e^{-2\theta t})\right).$$



(a) Value function.



(b) Value function derivative.

Figure 5.6: Value function and its derivative for the special case.

Taking the limit as $t \to \infty$, we get the unconditional distribution: $\mathcal{N}(\bar{z}, \frac{\sigma^2}{2\theta})$.

Figure 5.6 presents the results. Panel (a) shows the value function, while Panel (b) plots its derivative with respect to capital. The neural network's predictions are compared with the analytical benchmark from Equation (5.39). The value function is smooth except for a kink at $k = k_{\max}(z)$, where dividends switch from internal to external financing. The neural network reproduces this feature precisely, matching the analytical solution almost perfectly in both levels and derivatives. This confirms that the shifted ReLU activation provides an effective representation for problems involving kinks.

This exercise illustrates how the policy evaluation component of the DPI algorithm can recover the value function when the control is fixed. Next, we allow investment to be an endogenous choice variable and solve the full Hennessy–Whited model.

Solving the full model. We now solve the full Hennessy–Whited model using the DPI algorithm. We start by defining the neural network for the value function and for the investment policy function, as shown in Listing 5.19.

```
# Value function
v_core = Chain(
    Dense(2, 32, Lux.gelu),
    Dense(32, 32, Lux.gelu),
    Dense(32, 12, Lux.gelu),
    ShiftedReLU(),
    Dense(12, 1)
)
v_net(s, θᵥ) =  s[1:1, :].^m.α .* v_core(s, θᵥ, stᵥ)[1]

# Investment policy function
i_core = Chain(
    Dense(2, 32, Lux.gelu),
    Dense(32, 32, Lux.gelu),
    Dense(32, 12, Lux.gelu),
    ShiftedReLU(),
    Dense(12, 1)
)
i_net(s, θᵢ) = i_core(s, θᵢ, stᵢ)[1]
```

Listing 5.19: Neural network for the value function and the investment policy function.

Listing 5.20 shows the computation of the HJB residual and the corresponding loss function. These calculations are nearly identical to the ones for the special case, except that they depend on the parameters of the investment network $\boldsymbol{\theta}_i$. The dividends function computes both D and D^*, the latter of which is used to compute the FOC residual later on.

```
function drift_hyper(s::AbstractMatrix, m::HennessyWhited, θᵥ, θᵢ)
    ī       = i_net(s, θᵢ)
    μₛ, σₛ  = m.μₛ(s,ī ), m.σₛ(s,ī )
    F(ϵ)    = v_net(s .+ σₛ .* (ϵ / sqrt(2)) .+
                        μₛ .* (ϵ^2 / 2.0), θᵥ)
  return ForwardDiff.derivative(ϵ ->
            ForwardDiff.derivative(F, ϵ), 0.0)
end

function dividends(s, m::HennessyWhited, θᵢ)
    k = @view s[1,:]; z = @view s[2,:]
    ī  = i_net(s, θᵢ)
    π = (exp.(z) .* k.^m.α)'
    D_star = π .- (ī  + 0.5f0 * m.χ * (ī  .- m.δ).^2).* k'
    D      = D_star .* (1 .+ m.λ * (D_star .< 0))
    return D, D_star
end

function hjb_residual(s, m::HennessyWhited, θᵥ, θᵢ)
    D, drift = dividends(s, m, θᵢ)[1], drift_hyper(s, m, θᵥ, θᵢ)
    return D .+ drift .- m.ρ .* v_net(s, θᵥ)
end

```

```julia
function loss_fn(dnn, θ, s, target)
    return mean(abs2, dnn(s, θ) - target)
end
```

Listing 5.20: Computing the HJB residual and the loss function.

Listing 5.21 shows the computation of the FOC residual and the targets used for training. We start by computing $v_k(\mathbf{s})$ using the forward-mode differentiation of the value function network. As the `ForwardDiff.jl` package works with scalar inputs, we first define a function that computes the derivative when taking a single value for k and z, and then define a batched version that takes a $2 \times B$ matrix of states. Given $v_k(\mathbf{s})$, the FOC residual is computed as the difference between the marginal adjustment cost and the value of capital. Finally, we define the targets for the value function and the investment policy function.

```julia
## Computing vk for the FOC residual
# Scalar version
function vk_scalar(k::T, z::T, ps_v) where {T}
    ForwardDiff.derivative(K -> begin
        x = @SVector [K, z]     # static vector
        v_net(x, ps_v)[1]
    end, k)
end

# Batched version
function vk(s::AbstractMatrix, ps_v)
    B = size(s, 2)
    vals = [vk_scalar(s[1,b], s[2,b], ps_v) for b in 1:B]
    return reshape(vals, 1, B)  # 1×B
end

function foc_residual(s, m::HennessyWhited, θᵥ, θᵢ)
    ī        = i_net(s, θᵢ)
    D_star  = dividends(s, m, θᵢ)[2]
    return @. vk(s, θᵥ) .- (1.0 + m.χ * (ī  .- m.δ)) * (1.0 +
                  m.λ * (D_star .< 0.0))
end

function targets(s, m, θᵥ, θᵢ; Δtᵥ = 0.2, Δtᵢ = 0.2)
    hjb      = hjb_residual(s, m, θᵥ, θᵢ)
    foc      = foc_residual(s, m, θᵥ, θᵢ)
    tgtᵥ     = v_net(s, θᵥ) + hjb * Δtᵥ
    tgtᵢ     = i_net(s, θᵢ) + foc * Δtᵢ
    return tgtᵥ, tgtᵢ, mean(abs2, hjb), mean(abs2, foc)
end
```

Listing 5.21: Computing the FOC residual and the loss function.

We then train the neural network using the loop in Listing **??**. The model parameters and neural network architecture follow the same setup as in the previous section. We first define the hyperparameters and specify the sampling of the state space. Each iteration of the training loop consists of sampling a mini-batch of states, computing the target value using the HJB equation, and updating the neural network parameters using the Adam optimizer.

5.4.4 Portfolio Choice: Asset Allocation with Realistic Dynamics

As our final application, we consider a portfolio-choice problem featuring a large number of state variables, shocks, and controls. This environment illustrates the strength of the DPI algorithm in settings where classical methods become computationally infeasible. Following the spirit of Campbell and Viceira (2002), we study optimal asset allocation when expected returns vary over time in a manner consistent with empirical estimates.

The investor's environment. The investor allocates wealth across a risk-free asset, with instantaneous rate $r(\mathbf{x})$, and J risky assets with state-dependent instantaneous risk premia $\xi_j(\mathbf{x})$, $j = 1, \dots, J$. Expected returns depend on a vector of state variables $\mathbf{x} \in \mathbb{R}^N$, which evolves according to a multivariate Ornstein–Uhlenbeck process:

$$d\mathbf{x}_t = -\Phi \mathbf{x}_t \, dt + \sigma_x \, d\mathbf{Z}_t, \tag{5.41}$$

where Φ is an $N \times N$ mean-reversion matrix, σ_x is an $N \times N$ volatility matrix, and $\mathbf{Z}_t$ is N-dimensional Brownian motion.

The investor chooses consumption C_t and portfolio shares $\boldsymbol{\alpha}_t = (\alpha_{1,t}, \dots, \alpha_{J,t})^\top$ subject to position limits

$$0 \le \alpha_{j,t} \le 1, \qquad j = 1, \dots, J,$$

which rule out leverage and short-selling.

Wealth dynamics. Let W_t denote the investor's wealth. Given portfolio weights $\boldsymbol{\alpha}_t$, wealth evolves as

$$dW_t = \Big[(r(\mathbf{x}_t) + \boldsymbol{\alpha}_t^\top \xi(\mathbf{x}_t)) W_t - C_t \Big] dt + W_t \, \boldsymbol{\alpha}_t^\top \sigma_{\mathbf{R}} \, d\mathbf{Z}_t, \tag{5.42}$$

where $\sigma_{\mathbf{R}}$ is the $J \times N$ matrix mapping risky-asset shocks into portfolio returns. For simplicity, we assume that the matrix of risk exposures is constant. Both the drift and diffusion of wealth depend on the state $\mathbf{x}_t$ and the control vector $\boldsymbol{\alpha}_t$, creating a high-dimensional HJB problem with multiple controls.

Preferences. The investor has continuous-time Epstein–Zin recursive preferences. Let $V(W, \mathbf{x})$ denote continuation utility. The intertemporal aggregator is

$$f(C, V) = \frac{\rho(1-\gamma)V}{1-\psi^{-1}} \left[\left(\frac{C}{\big((1-\gamma)V\big)^{1/(1-\gamma)}} \right)^{1-\psi^{-1}} - 1 \right], \tag{5.43}$$

where γ is relative risk aversion, ψ is the elasticity of intertemporal substitution, and ρ is the time-preference rate. The special case $\gamma = \psi^{-1}$ corresponds to standard expected-utility preferences.

The investor's problem. The value function satisfies

$$V(W, \mathbf{x}) = \max_{\{C_t, \boldsymbol{\alpha}_t\}_{t=0}^{\infty}} \mathbb{E}_0 \left[\int_0^{\infty} f(C_t, V_t)\, dt \right], \tag{5.44}$$

subject to the wealth dynamics, portfolio constraints, and the state dynamics (5.41).

This portfolio problem involves:

- state dimension: $1 + N$ (wealth plus N state variables),
- shock dimension: N Brownian motions,
- control dimension: $1 + J$ (consumption plus J portfolio weights),
- inequality constraints: $0 \leq \alpha_j \leq 1$.

It thus provides a natural testbed for the DPI algorithm in a realistic, high-dimensional continuous-time setting.

The asset structure. We assume that the investor has access to $J = 5$ risky assets and a risk-free asset. The risky assets consist of the aggregate stock market index and four bond indices, corresponding to medium- and long-term nominal and inflation-indexed bonds. These assets span the main components of the investable opportunity set relevant for long-term portfolio choice.

Estimation of the state dynamics. To capture the dynamics of expected returns across these asset classes, we require a sufficiently rich set of state variables that reflect the key drivers of stock and bond premia. Following Jiang et al. (2024), we assume that there are $N = 11$ state variables, comprised of the financial and macroeconomic predictors listed in Table 5.2. These variables include standard bond- and stock-market predictors, along with macroeconomic indicators that influence term premia, risk premia, and discount-rate variation.

Even though the state vector $\mathbf{x}_t$ follows the continuous-time process (5.41), the data are observed at discrete intervals. Time-aggregating the OU process implies that $\mathbf{x}_t$ follows the discrete-time VAR

$$\mathbf{x}_{t+\Delta t} = \exp(-\Phi \Delta t)\, \mathbf{x}_t + \mathbf{u}_{t+\Delta t}, \tag{5.45}$$

where the time-aggregated innovation is

$$\mathbf{u}_{t+\Delta t} = \int_t^{t+\Delta t} \exp(-\Phi(t + \Delta t - s))\, \sigma_x\, d\mathbf{Z}_s. \tag{5.46}$$

The discrete-time VAR can be estimated using standard methods. This yields (i) the autoregressive coefficient matrix

$$\Psi \equiv \exp(-\Phi \Delta t),$$

Table 5.2: List of State Variables Driving the Expected Returns of Assets

Variable	**Description**	**Mean**	**S.D.**(%)
π_t	Log Inflation	0.032	2.3
$y_t^{\$}(1)$	Log 1-Year Nominal Yield	0.043	3.1
$yspr_t^{\$}$	Log 5-Year Minus 1-Year Nominal Yield Spread	0.006	0.7
Δz_t	Log Real GDP Growth	0.030	2.4
Δd_t	Log Stock Dividend-to-GDP Growth	-0.002	6.3
d_t	Log Stock Dividend-to-GDP Level	-0.270	30.5
pd_t	Log Stock Price-to-Dividend Ratio	3.537	42.6
$\Delta\tau_t$	Log Tax Revenue-to-GDP Growth	0.000	5.0
τ_t	Log Tax Revenue-to-GDP Level	-1.739	6.5
Δg_t	Log Spending-to-GDP Growth	0.006	7.6
g_t	Log Spending-to-GDP Level	-1.749	12.9

Notes: The table reports the 11 state variables driving expected returns in our economy, together with their unconditional mean and standard deviation. The data are obtained from `https://www.publicdebtvaluation.com/data`.

and (ii) the covariance matrix of the innovations $\mathbf{u}_{t+\Delta t}$.

The appendix of Duarte et al. (2024) describes how to solve the associated *inverse problem*: given the estimated discrete-time parameters $(\Psi, \text{Var}(\mathbf{u}))$, recover the continuous-time parameters (Φ, σ_x) that rationalize the data.

The state-price density. It is important to ensure that the process for expected returns is consistent with the principle of *no arbitrage*. We therefore postulate a process for the state-price density (SPD) and recover expected excess returns from the SPD. Under absence of arbitrage opportunities, an SPD must exist.

The real SPD $\{M_t\}$ evolves according to

$$\frac{dM_t}{M_t} = -r(\mathbf{x}_t)\,dt - \boldsymbol{\eta}(\mathbf{x}_t)^\top d\mathbf{Z}_t, \tag{5.47}$$

where $\boldsymbol{\eta}(\mathbf{x}_t)$ is the vector of *market prices of risk* associated with the Brownian shocks.

We assume that both the short rate and the market prices of risk are affine functions of the state:

$$r(\mathbf{x}_t) = r_0 + \mathbf{r}_1^\top \mathbf{x}_t, \qquad \boldsymbol{\eta}(\mathbf{x}_t) = \boldsymbol{\eta}_0 + \boldsymbol{\eta}_1^\top \mathbf{x}_t. \tag{5.48}$$

Given the SPD, the vector of expected excess returns $\boldsymbol{\xi}(\mathbf{x}_t)$ is pinned down by the no-arbitrage condition

$$\boldsymbol{\xi}(\mathbf{x}_t) = \boldsymbol{\sigma}_{\mathbf{R}}\,\boldsymbol{\eta}(\mathbf{x}_t), \tag{5.49}$$

where $\boldsymbol{\sigma}_{\mathbf{R}}$ is the matrix of factor loadings of returns on the underlying shocks. Duarte et al. (2024) discuss in detail how to recover $\boldsymbol{\sigma}_{\mathbf{R}}$ in this setting.

Estimation of the SPD. Given the affine structure of $r(\mathbf{x})$ and $\boldsymbol{\eta}(\mathbf{x})$, the model implies closed-form expressions for bond prices, bond yields, and expected returns. We estimate the parameters of the SPD by minimizing the squared deviations between model-implied

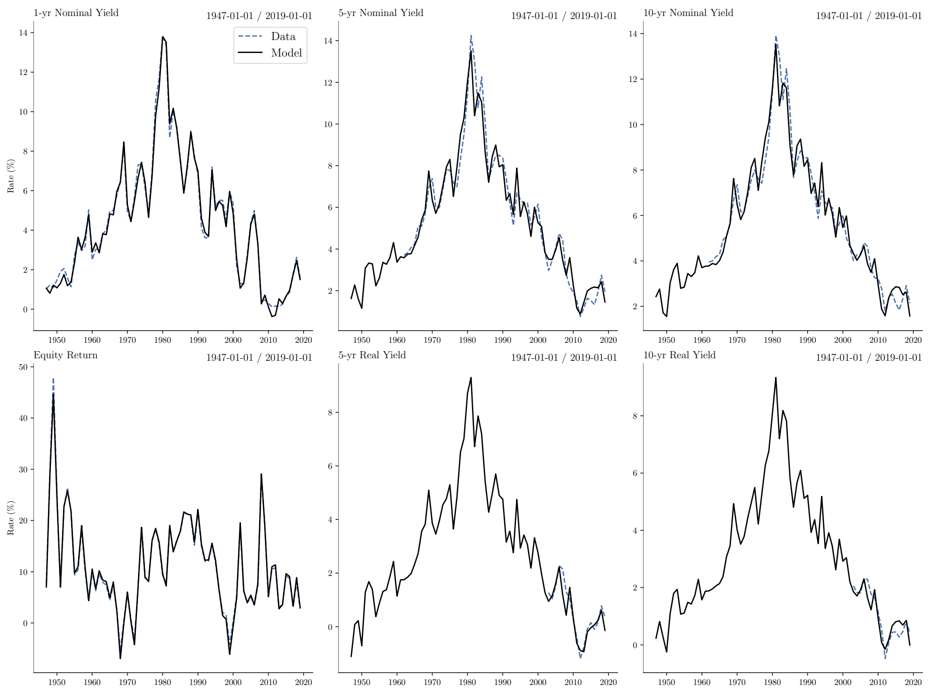


Figure 5.7: Bond yields and equity expected returns: model vs. data.

time-integrated values and their empirical counterparts across 12 time series: one-, two-, five-, ten-, 20-, and 30-year nominal yields; five-, seven-, ten-, 20-, and 30-year real yields; and expected stock returns.

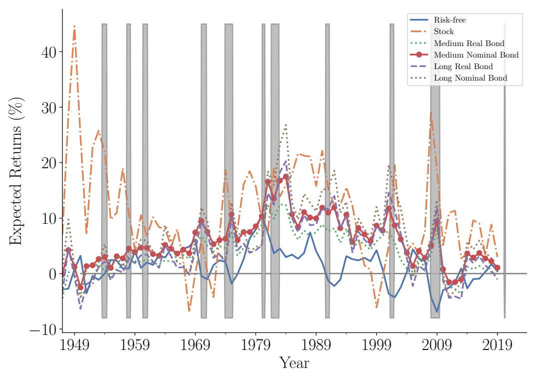
Figure 5.7 shows the fit for six representative series. The model successfully captures the dynamics of nominal and real bond yields across maturities. Moreover, the model-implied conditional equity premium closely matches the VAR-based estimates.

Optimal allocation. Having estimated the SPD parameters, we obtain $\boldsymbol{\xi}(\mathbf{x}_t)$ and $\boldsymbol{\sigma}_{\mathbf{R}}$, which fully characterize the return dynamics. Given these objects, we can solve the portfolio-choice problem using the DPI algorithm.

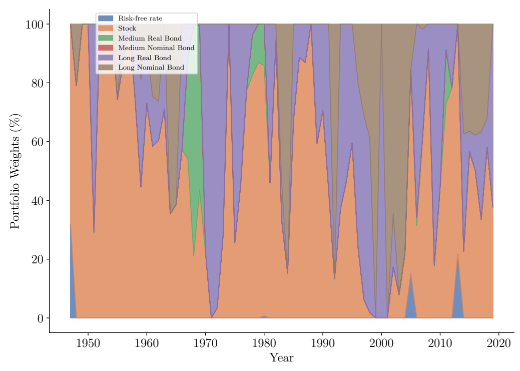
Panel (a) of Figure 5.8 shows substantial variation in expected returns over time, with a strong cyclical component and large spikes around recessions.

Panel (b) reports the optimal allocation derived from the DPI solution. The investor engages in substantial market timing: for instance, holding a large stock position during the 1950s–1960s, but nearly exiting equities in the early 1970s. Similarly, the optimal portfolio prescribes minimal equity exposure during the early 2000s (Dot-Com collapse).

The figure also reveals rich substitution patterns between stocks and bonds. During the early 1970s, declining equity exposure is accompanied primarily by increased holdings of long-term *real* bonds. In contrast, during the early 2000s downturn, the investor reallocates mostly toward nominal bonds. To better understand these substitution patterns, we next study how the policy functions vary with each state variable in isolation.



(a) Expected returns.



(b) Optimal asset allocation.

Figure 5.8: Time series of expected returns and optimal asset allocations.

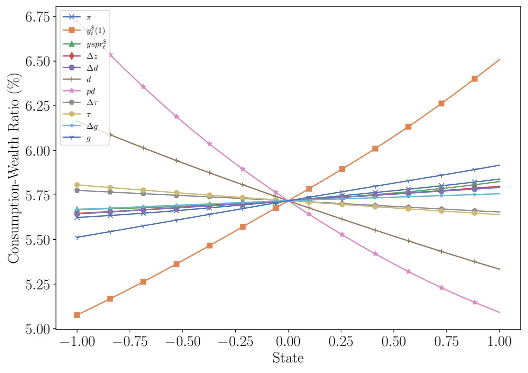
Notes. Panel (a) shows the expected returns implied by the model for five asset classes—equities; nominal and real long-term bonds (ten-year maturity); and nominal and real medium-term bonds (five-year maturity)—from 1949 to 2019. Shaded areas denote NBER recessions. Panel (b) displays the optimal asset allocation for an investor with recursive preferences solving the problem in Eq. (29), using the return dynamics in panel (a) and preference parameters $\rho = 0.04$, $\gamma = 20$, and $\psi = 0.5$.

Policy functions. Figure 5.9 shows how the consumption–wealth ratio and the portfolio shares of selected assets respond to changes in the state variables. Each line depicts the response of a given policy function as we vary one state variable by ± 1 unconditional standard deviation while holding all other variables fixed at their sample means.

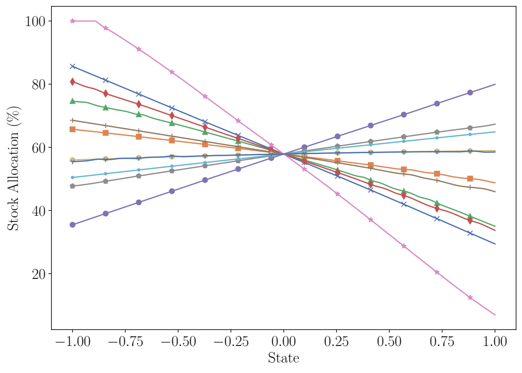
Panel (a) of Figure 5.9 shows how the consumption–wealth ratio responds to changes in the state vector. An increase in expected returns—captured, for instance, by higher short-term nominal yields or a lower price–dividend ratio—leads the investor to consume a larger fraction of her wealth. In other words, the consumption–wealth ratio rises when returns are high. This behavior reflects the dominance of the income effect over the substitution effect when the elasticity of intertemporal substitution is low (here, $\psi = 0.5$).

Turning to portfolio choice, Panel (b) shows that the optimal share invested in equities declines with the price–dividend ratio, consistent with the fact that a high price–dividend ratio forecasts lower future stock returns. More interestingly, equity demand also varies with variables traditionally associated with the bond market— such as the yield spread or inflation—reflecting substitution between stocks and the different fixed-income assets.

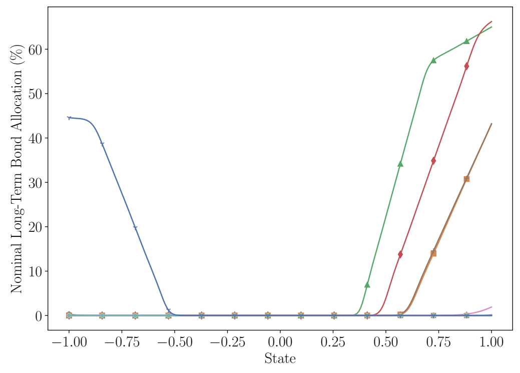
The demand for long-term real bonds is naturally increasing in the inflation rate, as these bonds provide inflation protection. For small deviations of inflation from its mean, the investor primarily relies on long-term bonds for inflation hedging; for larger deviations, she uses both medium- and long-term real bonds. We also find that demand for inflation-protected bonds is highly sensitive to movements in the price–dividend ratio. This is not a mechanical consequence of the model but reflects an active reallocation decision: when equities become less attractive due to a high price–dividend ratio, the investor could in principle allocate more to short-term nominal bonds or long-term nominal bonds. Instead, she reallocates disproportionately to inflation-protected assets.



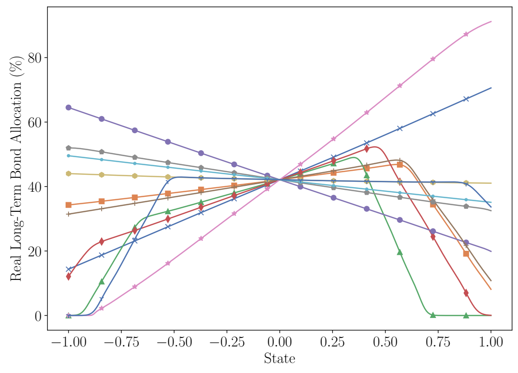
(a) Consumption–wealth ratio.



(b) Stock share.



(c) Nominal long-term bond.



(d) Real long-term bond.

Figure 5.9: Policy functions for consumption and selected asset positions.

The investor's response to changes in the yield spread is more intricate. An increase in the yield spread initially reduces stock holdings and raises positions in long-term real bonds. For larger deviations, however, the investor substitutes away from real bonds toward long-term nominal bonds. These nonlinear and state-dependent substitution patterns produce highly nonmonotonic policy functions—features that standard log-linear approximations used in portfolio problems (e.g., Campbell–Viceira style approximations) would fail to capture.

5.5 Conclusion

This chapter discussed a novel numerical method that alleviates the three curses of dimensionality. The method rests on three pillars. First, it uses deep learning to represent value and policy functions. Second, it combines Ito's lemma and automatic differentiation to compute exact expectations with negligible additional computational cost. Third, it uses a gradient-based version of policy iteration that dispenses root- finding methods to find the optimal control for a given state. We show that the DPI method has broad applicability in several areas of Finance, such as asset pricing, corporate finance, and portfolio choice, and that it can solve complex large-dimensional problems with highly nonlinear dynamical systems.

The ability to solve rich high-dimensional problems can be an invaluable tool in economic analysis. We oftentimes are forced to make assumptions that have no clear economic interest but are necessary for the model solution to be feasible. This often makes it hard to determine whether results are due to these auxiliary assumptions or to the economically motivated ones. By significantly expanding the set of models that researchers can solve, or even potentially estimate, our methods enable researchers to focus on models that better capture the rich phenomena that we observe in modern economies, instead of focusing on models that current numerical methods can solve.

Appendix A

Appendix

A.1 The multivariate version of Itô's lemma

In Chapter 5, we used the multivariate version of Itô's lemma to derive the HJB equation. In this section, we provide a derivation of the multivariate version of Itô's lemma, which builds on the multivariate Taylor expansion and the rules of Itô's calculus. For a comprehensive treatment of matrix differential calculus, see Magnus and Neudecker (1999).

A.1.1 The multivariate Taylor expansion

First-order approximation. Let $f(\mathbf{s}) \in \mathbb{R}$ be a scalar-valued function of the state vector $\mathbf{s} \in \mathbb{R}^n$. The derivative of $f(\mathbf{s})$ is a linear operator that maps a small perturbation $\mathbf{h} \in \mathbb{R}^n$ in $\mathbf{s}$ to a small perturbation in $f(\mathbf{s})$, that is, the *differential* of $f(\mathbf{s})$ is given by:

$$df(\mathbf{s};\mathbf{h}) = Df(\mathbf{s})\mathbf{h}. \tag{A.1}$$

Equivalently, this gives the first-order approximation of $f(\mathbf{s})$ around a given point $\mathbf{s}_0 \in \mathbb{R}^n$:

$$f(\mathbf{s}+\mathbf{h}) = f(\mathbf{s}) + Df(\mathbf{s})\mathbf{h} + o(\|\mathbf{h}\|). \tag{A.2}$$

As $\mathbf{s}$ is a column vector, $Df(\mathbf{s}_0)$ must be a row vector, such that the product $Df(\mathbf{s}_0)(\mathbf{s}-\mathbf{s}_0)$ is a scalar. Formally, the derivative is given by:

$$Df(\mathbf{s}_0) = \left.\frac{\partial f}{\partial \mathbf{s}^\top}\right|_{\mathbf{s}=\mathbf{s}_0} = \left(\frac{\partial f}{\partial \mathbf{s}_1}, \frac{\partial f}{\partial \mathbf{s}_2}, \dots, \frac{\partial f}{\partial \mathbf{s}_n}\right). \tag{A.3}$$

Notice that we denote the derivative as $\frac{\partial f}{\partial \mathbf{s}^\top}$, where the transpose in $\mathbf{s}^\top$ acts as a reminder that the derivative is a row vector.

It is often useful to work with the gradient of $f(\mathbf{s})$ instead of the derivative. In our setting, the gradient is a column vector, given by the transpose of the derivative:

$$\nabla f(\mathbf{s}) = Df(\mathbf{s})^\top = \left(\frac{\partial f}{\partial \mathbf{s}_1}, \frac{\partial f}{\partial \mathbf{s}_2}, \dots, \frac{\partial f}{\partial \mathbf{s}_n}\right)^\top. \tag{A.4}$$

We can then write the first-order approximation of $f(\mathbf{s})$ around a given point $\mathbf{s}_0 \in \mathbb{R}^n$ as:

$$f(\mathbf{s}+\mathbf{h}) = f(\mathbf{s}) + \nabla f(\mathbf{s})^\top\mathbf{h} + o(\|\mathbf{h}\|). \tag{A.5}$$

Second-order approximation. We can define the second differential of $f(\mathbf{s})$, which is simply the differential of the first differential:

$$d^2 f(\mathbf{s};\mathbf{h}) = d(df(\mathbf{s};\mathbf{h});\mathbf{h}). \tag{A.6}$$

Using the expression for the first differential, we can write the second differential as:

$$d^2 f(\mathbf{s};\mathbf{h}) = d(\nabla f(\mathbf{s})^\top \mathbf{h}) = d(\nabla f(\mathbf{s}))^\top \mathbf{h} = \left(D\nabla f(\mathbf{s})\mathbf{h}\right)^\top \mathbf{h} = \mathbf{h}^\top (D\nabla f(\mathbf{s}))^\top \mathbf{h}, \tag{A.7}$$

where $D\nabla f(\mathbf{s}) \in \mathbb{R}^{n\times n}$ is *Jacobian* matrix of $\nabla f(\mathbf{s})$.

The matrix inside the quadratic form is the *Hessian* matrix of $f(\mathbf{s})$, given by:

$$\mathbf{H}f(\mathbf{s}) = D\nabla f(\mathbf{s})^\top = \frac{\partial^2 f}{\partial \mathbf{s}\partial \mathbf{s}^\top} = \begin{pmatrix} \frac{\partial^2 f}{\partial s_1 \partial s_1} & \frac{\partial^2 f}{\partial s_1 \partial s_2} & \cdots & \frac{\partial^2 f}{\partial s_1 \partial s_n} \\ \frac{\partial^2 f}{\partial s_2 \partial s_1} & \frac{\partial^2 f}{\partial s_2 \partial s_2} & \cdots & \frac{\partial^2 f}{\partial s_2 \partial s_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial s_n \partial s_1} & \frac{\partial^2 f}{\partial s_n \partial s_2} & \cdots & \frac{\partial^2 f}{\partial s_n \partial s_n} \end{pmatrix}. \tag{A.8}$$

Given the second differential, we can obtain a second-order approximation of $f(\mathbf{s})$:

$$f(\mathbf{s}+\mathbf{h}) = f(\mathbf{s}) + df(\mathbf{s};\mathbf{h}) + \frac{1}{2}d^2 f(\mathbf{s};\mathbf{h}) + o(\|\mathbf{h}\|^2). \tag{A.9}$$

Using the expression for the first and second differential, we obtain the second-order Taylor expansion of $f(\mathbf{s})$ around a given point $\mathbf{s}_0 \in \mathbb{R}^n$ as:

$$f(\mathbf{s}+\mathbf{h}) = f(\mathbf{s}) + \nabla f(\mathbf{s})^\top \mathbf{h} + \frac{1}{2}\mathbf{h}^\top \mathbf{H}f(\mathbf{s})\mathbf{h} + o(\|\mathbf{h}\|^2). \tag{A.10}$$

A.1.2 Itô's lemma in higher dimensions

Suppose now that $\mathbf{s}$ follows a diffusion process given by:

$$d\mathbf{s} = \mathbf{f}(\mathbf{s},\mathbf{c})dt + \mathbf{g}(\mathbf{s},\mathbf{c})d\mathbf{B}, \tag{A.11}$$

where $\mathbf{f}(\mathbf{s},\mathbf{c}) \in \mathbb{R}^n$ is the drift and $\mathbf{g}(\mathbf{s},\mathbf{c}) \in \mathbb{R}^{n\times m}$ is the diffusion matrix. We can think of $d\mathbf{s}$ as a small perturbation in $\mathbf{s}$, so using the second-order Taylor expansion of $f(\mathbf{s})$ around $\mathbf{s}$, we obtain:

$$df(\mathbf{s}) = \nabla f(\mathbf{s})^\top d\mathbf{s} + \frac{1}{2}d\mathbf{s}^\top \mathbf{H}f(\mathbf{s})d\mathbf{s} + o(\|d\mathbf{s}\|^2). \tag{A.12}$$

Now, using the Itô's product rules, $d\mathbf{B}d\mathbf{B}^\top = I_m dt$ and $d\mathbf{B}dt = dt^2 = 0$, and taking expectations, we obtain:

$$df(\mathbf{s}) = \nabla f(\mathbf{s})^\top \mathbf{f}(\mathbf{s},\mathbf{c})dt + \frac{1}{2}\mathbb{E}\left[d\mathbf{B}^\top \mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}f(\mathbf{s})\mathbf{g}(\mathbf{s},\mathbf{c})d\mathbf{B}\right] + o(dt^2), \tag{A.13}$$

using the fact that $\mathbb{E}[\|d\mathbf{s}\|^2] = \mathcal{O}(dt^2)$.

Notice that the second term in the expression above is a scalar, so it is equal to its trace. Using the cyclic property of the trace ($\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$), we obtain:

$$\mathbb{E}\left[d\mathbf{B}^\top \mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}f(\mathbf{s})\mathbf{g}(\mathbf{s},\mathbf{c})d\mathbf{B}\right] = \mathrm{Tr}\left[\mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}f(\mathbf{s})\mathbf{g}(\mathbf{s},\mathbf{c})\mathbb{E}\left[d\mathbf{B}d\mathbf{B}^\top\right]\right]. \tag{A.14}$$

Using the fact that $\mathbb{E}[d\mathbf{B}d\mathbf{B}^\top] = I_m dt$, we obtain the multivariate version of Itô's lemma:

$$df(\mathbf{s}) = \mathcal{D}f(\mathbf{s})dt, \tag{A.15}$$

where

$$\mathcal{D}f(\mathbf{s}) = \nabla f(\mathbf{s})^\top \mathbf{f}(\mathbf{s},\mathbf{c}) + \frac{1}{2}\operatorname{Tr}\left[\mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}f(\mathbf{s})\mathbf{g}(\mathbf{s},\mathbf{c})\right]. \tag{A.16}$$

A.2 The hyper-dual approach to Itô's lemma

A.2.1 Proof of Proposition 5.1

In this section, we provide a proof of Proposition 5.1 in Chapter 5. We start from the multivariate version of Itô's lemma for a function $V(\mathbf{s})$:

$$dV(\mathbf{s}) = \mathcal{D}V(\mathbf{s})dt + \nabla V(\mathbf{s})^\top \mathbf{g}(\mathbf{s},\mathbf{c})d\mathbf{B}, \tag{A.17}$$

where

$$\mathcal{D}V(\mathbf{s}) = \nabla V(\mathbf{s})^\top \mathbf{f}(\mathbf{s},\mathbf{c}) + \frac{1}{2}\operatorname{Tr}\left[\mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}V(\mathbf{s})\mathbf{g}(\mathbf{s},\mathbf{c})\right], \tag{A.18}$$

and we drop the dependence on the control $\mathbf{c}$ for simplicity.

The matrix inside the trace is the Hessian matrix of $V(\mathbf{s})$, given by:

$$\mathbf{H}V(\mathbf{s}) = \frac{\partial^2 V}{\partial \mathbf{s}\partial \mathbf{s}^\top} = \begin{pmatrix} \frac{\partial^2 V}{\partial \mathrm{s}_1 \partial \mathrm{s}_1} & \frac{\partial^2 V}{\partial \mathrm{s}_1 \partial \mathrm{s}_2} & \cdots & \frac{\partial^2 V}{\partial \mathrm{s}_1 \partial \mathrm{s}_n} \\ \frac{\partial^2 V}{\partial \mathrm{s}_2 \partial \mathrm{s}_1} & \frac{\partial^2 V}{\partial \mathrm{s}_2 \partial \mathrm{s}_2} & \cdots & \frac{\partial^2 V}{\partial \mathrm{s}_2 \partial \mathrm{s}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 V}{\partial \mathrm{s}_n \partial \mathrm{s}_1} & \frac{\partial^2 V}{\partial \mathrm{s}_n \partial \mathrm{s}_2} & \cdots & \frac{\partial^2 V}{\partial \mathrm{s}_n \partial \mathrm{s}_n} \end{pmatrix}. \tag{A.19}$$

Therefore, the trace of the matrix above is given by:

$$\operatorname{Tr}\left[\mathbf{g}(\mathbf{s})^\top \mathbf{H}V(\mathbf{s})\mathbf{g}(\mathbf{s})\right] = \sum_{i=1}^{m} \mathbf{g}_i(\mathbf{s})^\top \mathbf{H}V(\mathbf{s})\mathbf{g}_i(\mathbf{s}), \tag{A.20}$$

and we can write the drift of $V(\mathbf{s})$ as:

$$\mathcal{D}V(\mathbf{s}) = \nabla V(\mathbf{s})^\top \mathbf{f}(\mathbf{s}) + \frac{1}{2}\sum_{i=1}^{m} \mathbf{g}_i(\mathbf{s})^\top \mathbf{H}V(\mathbf{s})\mathbf{g}_i(\mathbf{s}). \tag{A.21}$$

Auxiliary functions. Next, define the auxiliary functions $F_i : \mathbb{R} \to \mathbb{R}$ as

$$F_i(\epsilon) = V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right). \tag{A.22}$$

The derivative of $F_i(\epsilon)$ is given by:

$$F_i'(\epsilon) = \nabla V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right)^\top \left(\frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}} + \frac{\mathbf{f}(\mathbf{s})}{m}\epsilon\right). \tag{A.23}$$

Evaluating at $\epsilon = 0$, we obtain:

$$F_i'(0) = \nabla V(\mathbf{s})^\top \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}. \tag{A.24}$$

This implies that the diffusion matrix of $V(\mathbf{s})$ is given by:

$$\nabla V(\mathbf{s})^\top \mathbf{g}(\mathbf{s}) = \sqrt{2}[F_1'(0), F_2'(0), \cdots, F_m'(0)]. \tag{A.25}$$

Drift. The second derivative of $F_i(\epsilon)$ is given by:

$$F_i''(\epsilon) = \left[\frac{d}{d\epsilon}\nabla V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right)\right]^\top \left(\frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}} + \frac{\mathbf{f}(\mathbf{s})}{m}\epsilon\right) + \nabla V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right)^\top \frac{\mathbf{f}(\mathbf{s})}{m}. \tag{A.26}$$

Notice that the term inside the brackets is given by:

$$\frac{d}{d\epsilon}\nabla V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right) = \frac{\partial \nabla V}{\partial \mathbf{s}^\top}\left(\frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}} + \frac{\mathbf{f}(\mathbf{s})}{m}\epsilon\right). \tag{A.27}$$

Evaluating at $\epsilon = 0$, we obtain:

$$F_i''(0) = \frac{1}{2}\mathbf{g}_i(\mathbf{s})^\top \left(\frac{\partial \nabla V(\mathbf{s})}{\partial \mathbf{s}}\right)^\top \mathbf{g}_i(\mathbf{s}) + \nabla V(\mathbf{s})^\top \frac{\mathbf{f}(\mathbf{s})}{m}. \tag{A.28}$$

Defining the function $F(\epsilon) = \sum_{i=1}^m F_i(\epsilon)$, we obtain:

$$F''(0) = \sum_{i=1}^m F_i''(0) = \frac{1}{2}\sum_{i=1}^m \mathbf{g}_i(\mathbf{s})^\top \mathbf{H}V(\mathbf{s})\mathbf{g}_i(\mathbf{s}) + \nabla V(\mathbf{s})^\top \mathbf{f}(\mathbf{s}), \tag{A.29}$$

using the fact that $\mathbf{H}V(\mathbf{s}) = \frac{\partial^2 V(\mathbf{s})}{\partial \mathbf{s}\partial \mathbf{s}^\top} = \left[\frac{\partial \nabla V(\mathbf{s})}{\partial \mathbf{s}^\top}\right]^\top$. Hence, $F''(0) = \mathcal{D}V(\mathbf{s})$. This concludes the proof.

A.2.2 A hyper-dual implementation

We can implement the computation of the drift of $V(\mathbf{s})$ using a hyper-dual number. Let's start by defining hyper-dual number for a scalar-valued function as a triplet containing the value of the function, the drift vector, and the matrix of risk exposures:

$$s = s_0 + \mathbf{s}_\sigma \mathbf{i} + s_\mu \mathbf{j}, \tag{A.30}$$

where $s_0 \in \mathbb{R}$ corresponds to the primal value, $\mathbf{s}_\sigma \in \mathbb{R}^{1\times m}$ corresponds to the diffusion matrix, and $s_\mu \in \mathbb{R}$ corresponds to the drift.

For a pair of hyper-dual numbers, s and t, we can define the following operations:

1. Multiplication by scalar: $\alpha \cdot s \equiv (\alpha \cdot s_0) + (\alpha \cdot \mathbf{s}_\sigma)\,\mathbf{i} + (\alpha \cdot s_\mu)\,\mathbf{j}$.
2. Addition: $s + t \equiv (s_0 + t_0) + (\mathbf{s}_\sigma + \mathbf{t}_\sigma)\,\mathbf{i} + (s_\mu + t_\mu)\,\mathbf{j}$.

3. Multiplication: $s \cdot t \equiv (s_0 \cdot t_0) + (s_0 \cdot \mathbf{t}_\sigma + \mathbf{s}_\sigma \cdot t_0)\,\mathbf{i} + (s_0 \cdot t_\mu + s_\mu \cdot t_0 + \frac{1}{2}\mathbf{s}_\sigma^\top \mathbf{t}_\sigma)\,\mathbf{j}$.

4. For an arbitrary unary function $f : \mathbb{R} \to \mathbb{R}$, we have

$$f(s) \equiv f(s_0) + (\mathbf{s}_\sigma \cdot f'(s_0))\,\mathbf{i} + (s_\mu \cdot f^{'}(s_0) + \frac{1}{2} f''(s_0)\mathbf{s}_\sigma^\top \mathbf{s}_\sigma)\,\mathbf{j}. \tag{A.31}$$

Given the rules of propagation for a scalar-valued hyper-dual number, it is straightforward to extend to the case of a vector-valued hyper-dual number: $\mathbf{s} = (s_1, \ldots, s_n)^\top$, where $s_i = s_{i0} + \mathbf{s}_{i\sigma}\mathbf{i} + s_{i\mu}\mathbf{j}$ is a scalar-valued hyper-dual number.

A.3 Derivations for the Lucas orchard model

Consider a Lucas orchard model with $N+1$ trees. Dividends for the i-th tree are given by:

$$\frac{dD_{i,t}}{D_{i,t}} = \mu_i dt + \sigma_i dB_{i,t}, \qquad i = 1, 1, \ldots, N, \tag{A.32}$$

where $B_{i,t}$ is a Brownian motion

As in the two-trees model, it is useful to work with the dividend share for the i-th tree, $s_i = D_{i,t}/C_t$, where aggregate consumption is given by $C_t = \sum_{i=1}^N D_{i,t}$.

The process for consumption is given by:

$$\frac{dC_t}{C_t} = \sum_{i=1}^N \frac{D_{i,t}}{C_t}\frac{dD_{i,t}}{D_{i,t}} = \mu_c(\mathbf{s}_t)dt + \sigma_c(\mathbf{s}_t)^\top d\mathbf{B}, \tag{A.33}$$

where $\mu_c(\mathbf{s}_t) = \sum_{i=1}^N s_i\mu_i$ and $\sigma_c(\mathbf{s}_t) = [s_1\sigma_1, s_2\sigma_2, \ldots, s_N\sigma_N]^\top$.

The dividend share process is given by:

$$\frac{ds_i}{s_i} = \frac{dD_{i,t}}{D_{i,t}} - \frac{dC_t}{C_t} - \frac{dD_{i,t}}{D_{i,t}}\frac{dC_t}{C_t} + \left(\frac{dC_t}{C_t}\right)^2 \Rightarrow ds_{i,t} = \mu_{s_i}(\mathbf{s}_t)dt + \sigma_{s_i}(\mathbf{s}_t)^\top d\mathbf{B}, \tag{A.34}$$

where

$$\mu_{s_i}(\mathbf{s}_t) = s_i\left[\mu_i - \mu_c(\mathbf{s}_t) - s_i\sigma_i^2 + \sigma_c(\mathbf{s}_t)^\top\sigma_c(\mathbf{s}_t)\right], \tag{A.35}$$

$$\sigma_{s_i}(\mathbf{s}_t) = s_i\left[\sigma_i e_i - \sigma_c(\mathbf{s}_t)\right], \tag{A.36}$$

and e_i is the i-th unit vector. Notice that $\mu_{s_i}(\mathbf{s}) = 0$ and $\sigma_{s_i}(\mathbf{s}) = \mathbf{0}_{N\times 1}$ for $s_i = 0$ and $s_i = 1$. We can then write the law of motion of the vector of dividend shares as:

$$d\mathbf{s}_t = \mu_s(\mathbf{s}_t)dt + \sigma_s(\mathbf{s}_t)d\mathbf{B}, \tag{A.37}$$

where $\mu_s(\mathbf{s}_t) = [\mu_{s_1}(\mathbf{s}_t), \mu_{s_2}(\mathbf{s}_t), \ldots, \mu_{s_N}(\mathbf{s}_t)]^\top$ and $\sigma_s(\mathbf{s}_t) = [\sigma_{s_1}(\mathbf{s}_t), \sigma_{s_2}(\mathbf{s}_t), \ldots, \sigma_{s_N}(\mathbf{s}_t)]^\top$. Notice that the $n \times n$ diffusion matrix is singular, as $\mathbf{1}_N^\top\sigma_s(\mathbf{s}_t) = \mathbf{0}_{1\times N}$.

The price-consumption ratio $v_{i,t} \equiv P_{i,t}/C_t$ for the i-th tree is given by:

$$v_{i,t} = \mathbb{E}_t\left[\int_0^\infty e^{-\rho s} s_{i,t+s} ds\right]. \tag{A.38}$$

The stationary HJB equation for $v_{i,t} = v_i(\mathbf{s}_t)$ is given by:

$$\rho v_i(\mathbf{s}) = s_i + \nabla v_i^\top \mu_s(\mathbf{s}) + \frac{1}{2}\operatorname{Tr}\left[\sigma_s(\mathbf{s})^\top \mathbf{H}v_i(\mathbf{s})\sigma_s(\mathbf{s})\right]. \tag{A.39}$$

with boundary conditions $v_{i,t}(0) = 0$ and $v_{i,t}(1) = 1/\rho$.

A.4 Derivations for the Hennessy and Whited (2007) model

In this section, we provide a chracterization of the optimal investment policy for our version of the Hennessy and Whited (2007) model.

Optimal investment policy. The firm's investment behavior depends on whether the firm is incurring the equity issuance costs or not. We have three possible cases. First, if the firm pays positive dividends, then the optimal investment policy is given by:

$$i_+(k,z) = \frac{v_k(\mathbf{s}) - 1}{\chi}, \tag{A.40}$$

Second, if firm decides to issue equity, so $D_t < 0$, then the optimal investment policy is given by:

$$i_-(k,z) = \frac{v_k(\mathbf{s})/(1+\lambda) - 1}{\chi}, \tag{A.41}$$

There is a region of the state space where the firm chooses not to issue equity and not to pay dividends. We refer to this region as the *inaction region*. When the firm is in the inaction region, dividends are zero, so the optimal investment policy is given by:

$$e^z k^{\alpha-1} = i + 0.5\chi i^2 \Rightarrow \chi i^2 + 2i - 2e^z k^{\alpha-1} = 0. \tag{A.42}$$

Dividends will be negative if $i > i_{\max}$ or $i < i_{\min}$, where

$$i_{\max}(k,z) = \frac{\sqrt{1+2\chi e^z k^{\alpha-1}} - 1}{\chi}, \quad i_{\min}(k,z) = \frac{-\sqrt{1+2\chi e^z k^{\alpha-1}} - 1}{\chi}. \tag{A.43}$$

Therefore, the optimal investment policy is given by:

$$i(k,z) = \begin{cases} i_+(k,z) & \text{if } |v_k(\mathbf{s})| < \sqrt{1+2\chi e^z k^{\alpha-1}}, \\ i_-(k,z) & \text{if } |v_k(\mathbf{s})| > (1+\lambda)\sqrt{1+2\chi e^z k^{\alpha-1}}, \\ i_0(k,z) & \text{if } \sqrt{1+2\chi e^z k^{\alpha-1}} \leq |v_k(\mathbf{s})| \leq (1+\lambda)\sqrt{1+2\chi e^z k^{\alpha-1}}. \end{cases} \tag{A.44}$$

where $i_0(k,z) = i_{\max}(k,z)$ if $v_k(x,z) > 0$ and $i_0(k,z) = i_{\min}(k,z)$ if $v_k(x,z) < 0$.

A.4.1 Finite-differences solution

For simplicity, consider first the case where $\theta = \sigma_z = 0$, so z is constant, and $\lambda = 0$. In this case, we can treat z as a parameter and the problem reduces to a single state variable. Fix a grid for k, $\{k_1, \ldots, k_N\}$, where $k_1 = 0$, and let $v_j^n \equiv v^n(k_j)$ denote the value function evaluated at k_j.

Explicit method. Let's start by solving the HJB equation using an explicit method. We can write the discretized HJB equation as:

$$\frac{v_j^{n+1} - v_j^n}{\Delta t} + \rho v_j^n = e^z k_j^\alpha - (i_j^n + 0.5\chi(i_j^n)^2)k_j \tag{A.45}$$

$$+ \frac{v_{j+1}^n - v_j^n}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n > \delta} + \frac{v_j^n - v_{j-1}^n}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n < \delta}, \tag{A.46}$$

Define the forward and backward differences:

$$v_{j,F}^n = \frac{v_{j+1}^n - v_j^n}{\Delta k}, \quad v_{j,B}^n = \frac{v_j^n - v_{j-1}^n}{\Delta k}, \tag{A.47}$$

and the corresponding drifts:

$$\mu_{j,F}^n = (i_{j,F}^n - \delta)k_j, \quad \mu_{j,B}^n = (i_{j,B}^n - \delta)k_j \mathbf{1}_{i_{j,B}^n < \delta}, \tag{A.48}$$

where $i_{j,F}^n = \frac{v_{j,F}^n - 1}{\chi}$ and $i_{j,B}^n = \frac{v_{j,B}^n - 1}{\chi}$.

Then, the investment policy is given by:

$$i_j^n = i_{j,F}^n \mathbf{1}_{i_{j,F}^n > \delta} + i_{j,B}^n \mathbf{1}_{i_{j,B}^n < \delta} + \delta \mathbf{1}_{i_{j,F}^n < \delta < i_{j,B}^n}, \tag{A.49}$$

We can rearrange the expression above to obtain:

$$v_j^{n+1} = u_j^n + \ell_j^n v_{j-1}^n + s_j^n v_j^n + r_j^n v_{j+1}^n, \tag{A.50}$$

where $u_j^n = e^z k_j^\alpha - (i_j^n + 0.5\chi(i_j^n)^2)k_j$, and

$$r_j^b \equiv \frac{\Delta t}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n > \delta} \tag{A.51}$$

$$\ell_j^n \equiv -\frac{\Delta t}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n < \delta} \tag{A.52}$$

$$s_j^n \equiv 1 - \rho\Delta t - \frac{\Delta t}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n > \delta} + \frac{\Delta t}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n < \delta} \tag{A.53}$$

In matrix form, we can write:

$$\mathbf{v}^{n+1} = \mathbf{A}^n \mathbf{v}^n + \mathbf{u}^n, \tag{A.54}$$

where $\mathbf{A}^n$ is a tridiagonal matrix

$$\mathbf{A}^n = \begin{pmatrix} s_1^n & r_1^n & 0 & \dots & 0 \\ \ell_2^n & s_2^n & r_2^n & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \ell_{N-1}^n & s_{N-1}^n & r_{N-1}^n \\ 0 & \dots & 0 & \ell_N^n & s_N^n \end{pmatrix}, \tag{A.55}$$

Notice that $\ell_1^n = 1^n = 0$ and, if $i_{j,N}^n < \delta$, then $r_N^n = 0$, so we don't need to impose a boundary condition at the upper boundary.

Semi-implicit method. We can also solve the HJB equation using a semi-implicit finite-difference method. We can write the discretized HJB equation as:

$$\frac{v_j^{n+1} - v_j^n}{\Delta t} + \rho v_j^{n+1} = e^z k_j^\alpha - (i_j^n + 0.5\chi(i_j^n)^2)k_j \tag{A.56}$$

$$+ \frac{v_{j+1}^{n+1} - v_j^{n+1}}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n > \delta} + \frac{v_j^{n+1} - v_{j-1}^{n+1}}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n < \delta}, \tag{A.57}$$

where i_j^n is computed using an upwind scheme, as before.

Rearranging the expression above, and taking the limit as $\Delta t \to 0$, we obtain:

$$-\ell_j^{n+1} v_{j-1}^{n+1} + s_j^{n+1} v_j^{n+1} - r_j^{n+1} v_{j+1}^{n+1} = u_j^n, \tag{A.58}$$

where $u_j^n = e^z k_j^\alpha - (i_j^n + 0.5\chi(i_j^n)^2)k_j$, and

$$r_j^b \equiv \frac{1}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n > \delta} \tag{A.59}$$

$$\ell_j^{n+1} \equiv -\frac{1}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n < \delta} \tag{A.60}$$

$$s_j^n \equiv \ell_j^{n+1} + r_j^{n+1} + \rho \tag{A.61}$$

In matrix form, we can write:

$$A^n \mathbf{v}^{n+1} = \mathbf{u}^n, \tag{A.62}$$

where $\mathbf{A}^{n+1}$ is a tridiagonal matrix

$$\mathbf{A}^{n+1} = \begin{pmatrix} s_1^{n+1} & -r_1^{n+1} & 0 & \dots & 0 \\ -\ell_2^{n+1} & s_2^{n+1} & -r_2^{n+1} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & -\ell_{N-1}^{n+1} & s_{N-1}^{n+1} & -r_{N-1}^{n+1} \\ 0 & \dots & 0 & -\ell_N^{n+1} & s_N^{n+1} \end{pmatrix}, \tag{A.63}$$

Bibliography

Achdou, Yves, Jiequn Han, Jean-Michel Lasry, Pierre-Louis Lions, and Benjamin Moll, "Income and wealth distribution in macroeconomics: A continuous-time approach," *The review of economic studies*, 2022, *89* (1), 45–86.

Barles, Guy and Panagiotis E. Souganidis, "Convergence of Approximation Schemes for Fully Nonlinear Second Order Equations," *Asymptotic Analysis*, 1991, *4*, 271–283.

Black, Fischer and Myron Scholes, "The Pricing of Options and Corporate Liabilities," *Journal of Political Economy*, 1973, *81* (3), 637–654.

Bonnans, J. Frédéric, Élisabeth Ottenwaelter, and Housnaa Zidani, "A fast algorithm for the two dimensional HJB equation of stochastic control," *ESAIM: Modélisation mathématique et analyse numérique*, 2004, *38* (4), 723–735.

Boyd, John P., *Chebyshev and Fourier Spectral Methods*, Mineola, NY: Dover Publications, 2001.

Brenner, Susanne C. and L. Ridgway Scott, *The Mathematical Theory of Finite Element Methods*, 3 ed., Vol. 15 of *Texts in Applied Mathematics*, New York: Springer, 2008.

Brumm, Johannes and Simon Scheidegger, "Using Adaptive Sparse Grids to Solve High-Dimensional Dynamic Models," *Econometrica*, 2017, *85* (5), 1575–1612.

Campbell, John Y and Luis M Viceira, *Strategic asset allocation: portfolio choice for long-term investors*, Clarendon Lectures in Economic, 2002.

Carroll, Christopher D., "The Method of Endogenous Gridpoints for Solving Dynamic Stochastic Optimization Problems," *Economics Letters*, 2006, *91* (3), 312–320.

__ **and Miles S. Kimball**, "On the Concavity of the Consumption Function," *Econometrica*, 1996, *64* (4), 981–992.

Cochrane, John H., Francis A. Longstaff, and Pedro Santa-Clara, "Two Trees," *The Review of Financial Studies*, 2008, *21* (1), 347–385.

Cybenko, G, "Approximation by superposition of sigmoidal functions," *Mathematics of Control, Signals and Systems*, 1989, *2* (4), 303–314.

d'Avernas, Adrien, Damon Petersen, and Quentin Vandeweyer, "A Solution Method for Continuous-Time Models," 2024. Working paper. Available at https://faculty.chicagobooth.edu/quentin-vandeweyer/research.

Duarte, Victor, Diogo Duarte, and Dejanir H. Silva, "Machine Learning for Continuous-Time Finance," *The Review of Financial Studies*, 2024, *37* (11), 3217–3271.

Duchi, John, Elad Hazan, and Yoram Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, 2011, *12*, 2121–2159.

Fella, Giulio, "Endogenous grid method," 2025.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press, 2016.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2 ed., New York, NY: Springer, 2009.

Hendrycks, Dan and Kevin Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

Hennessy, Christopher A and Toni M Whited, "How costly is external financing? Evidence from a structural estimation," *Journal of Finance*, 2007, *62* (4), 1705–1745.

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

Jiang, Zhengyang, Hanno Lustig, Stijn Van Nieuwerburgh, and Mindy Z. Xiaolan, "The U.S. Public Debt Valuation Puzzle," *Econometrica*, 2024, *92* (4), 1309–1347.

Judd, Kenneth L., Lilia Maliar, Serguei Maliar, and Rafael Valero, "Smolyak Method for Solving Dynamic Economic Models: Lagrange Interpolation, Anisotropic Grid and Adaptive Domain," *Journal of Economic Dynamics and Control*, 2014, *44*, 92–123.

Kingma, Diederik P. and Jimmy Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds., *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., 2012, pp. 1097–1105.

Kushner, Harold J. and Paul G. Dupuis, *Numerical Methods for Stochastic Control Problems in Continuous Time* number 24. In 'Applications of Mathematics.', 2 ed., New York: Springer, 2001.

Leshno, Moshe, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural networks*, 1993, *6* (6), 861–867.

Loshchilov, Ilya and Frank Hutter, "Decoupled Weight Decay Regularization," in "International Conference on Learning Representations" 2019.

Lu, Zhou, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang, "The Expressive Power of Neural Networks: A View from the Width," in "Advances in Neural Information Processing Systems," Vol. 30 2017, pp. 6232–6240.

Ludwig, Alexander and Matthias Schoen, "Endogenous Grids in Higher Dimensions: Delaunay Interpolation and Hybrid Methods," *Computational Economics*, 2018, *51* (3), 607–636.

Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in "Proceedings of the 30th International Conference on Machine Learning (ICML 2013), Workshop on Deep Learning for Audio, Speech and Language Processing" 2013.

Magnus, Jan R. and Heinz Neudecker, *Matrix Differential Calculus with Applications in Statistics and Econometrics*, revised ed., Chichester, UK: John Wiley & Sons, 1999.

Martin, Ian, "The Lucas Orchard," *Econometrica*, 2013, *81* (1), 55–111.

McGrattan, Ellen R., "Solving the Stochastic Growth Model with a Finite Element Method," *Journal of Economic Dynamics and Control*, 1996, *20* (1-3), 19–42.

Merton, Robert C., "Theory of Rational Option Pricing," *The Bell Journal of Economics and Management Science*, 1973, *4* (1), 141–183.

Phelan, Thomas and Keyvan Eslami, "Applications of Markov Chain Approximation Methods to Optimal Control Problems in Economics," *Journal of Economic Dynamics and Control*, 2022, *143*, 104437.

Prince, Simon J. D., *Understanding Deep Learning*, Cambridge, UK: Cambridge University Press, 2023.

Ross, Stephen A, "Options and efficiency," *Quarterly Journal of Economics*, 1976, *90* (1), 75–89.

Rouwenhorst, K. Geert, "Asset Pricing Implications of Equilibrium Business Cycle Models," in Thomas F. Cooley, ed., *Frontiers of Business Cycle Research*, Princeton, NJ: Princeton University Press, 1995, chapter 10, pp. 294–330.

Schaab, Andreas and Allen Zhang, "Dynamic Programming in Continuous Time with Adaptive Sparse Grids," *SSRN Electronic Journal*, May 2022, pp. 1–57.

Smolyak, Sergei Abramovich, "Quadrature and interpolation formulas for tensor products of certain classes of functions," in "Doklady Akademii Nauk," Vol. 148 Russian Academy of Sciences 1963, pp. 1042–1045.

Sutton, Richard S. and Andrew G. Barto, *Reinforcement Learning: An Introduction*, 2 ed., Cambridge, MA: MIT Press, 2018.

Tauchen, George, “Finite State Markov-Chain Approximations to Univariate and Vector Autoregressions,” *Economics Letters*, 1986, *20* (2), 177–181.

_ **and Robert Hussey**, “Quadrature-Based Methods for Obtaining Approximate Solutions to Nonlinear Asset Pricing Models,” *Econometrica*, 1991, *59* (2), 371–396.

White, Matthew N., “The Method of Endogenous Gridpoints in Theory and Practice,” *Journal of Economic Dynamics and Control*, 2015, *60*, 26–41.