Numerical Methods-Lecture IX: Reinforcement Learning

(See Powell Chapter 10)

Trevor S. Gallen

Motivation

- Previously, this lecture was devoted to quadrature, and specifically how to integrate sampling from Chebyshev nodes
- While valuable, it may be going out of style relative to modern methods
- Instead, we'll talk about Reinforcement Learning
- Could have had a whole course, but I want to give you the motivation and a crash course

The Three Curses of Dimensionality

- Recall the three curses of dimensionality
    1. As n_{states} proliferates, # problems to solve explodes for VFI
    2. As $n_{actions}$ proliferates, # choices to compare explodes
    3. As $n_{outcomes}$ proliferates (particularly stochastic), space to integrate over explodes

$$V(x) = \max_{y \in \Gamma(X)} \left\{ F(x, y) + \beta E(V(x'(y))) \right\}$$

- This is a problem if you're trying to model lifecycle behavior...age, permanent wage, transitory wage, marital status, age of kids, occupation, health, etc.

- How can we solve?

One way to approximately solve the problem

- There are many flavors of solution, but we'll focus on the "Actor-Critic" method
- Have two functions, parameterized by θ and ϕ:
  - Actor function $\overline{\pi}(y|x;\theta)$ takes in state and spits out an action (possibly probabilistically)
  - Critic function $\overline{V}(x|\phi)$ takes in state and spits out value (traditional value function)
- Good actor function embeds both reward and stochastic future (actions and integral!)
- *Given* θ and ϕ, can simulate an agent
- Need to find θ and ϕ

Actor-Critic Algorithm (Discussion)

- Start with a guess for θ and ϕ, and an initial state value
- Simulate the system many times (random draws & laws of motion for stochastic problems)
- Now we have a bunch of paths for a given θ and ϕ
- For every step t, compute the return G_t, sum of reward and discounted future reward, calculated with $\overline{V(x_{t+1})}$
- Calculate the "advantage function" $D_t = G_t - V(S_t|\phi)$, value of action vs value of what we think is best action embedded in V
- Calculate gradients for actor and critic networks:

$$d\theta = \sum_{i=1}^{N} \nabla)_\theta \log(\pi(A|x_t;\theta)D_t$$

$$d\phi = \sum_{t=1}^{N} \nabla_\phi (G_t - V(x_t;\phi))^2$$

- Idea: ∇_θ pushes us in direction of better choices/happiness, ∇_ϕ pushes us in direction of lower error in value function
- Update the parameters of the functions:

$$\theta = \theta + \alpha d\theta$$

$$\phi = \phi + \beta d\phi$$

- So we update the actor π using critic V, and update V using simulation

Explaining to Mum-I

- Start out with some surface that represents best action given state, and some surface that represents the value of being in that state
- Simulate
- Change value surface by comparing data with what you actually got (using initial surface for future, so change is slow), trying to match surface
- Change action surface by trying to increase received value vs value at best guess (small perturbations toward better actions)
- Repeat in tiny steps

Idea

- We try to toddle slowly to both how to evaluate our situation (V) and what to do (π)
- We learn about value function by exploring the space
- We learn about maximization (actor) by exploiting V and our actual actions
- We learn about expectation by simulation–enough simulations and we will explore the relevant space
- Additional cool (but dangerous) aspect: we only explore relevant functions of state space
- But how choose V and π?

Choosing V and π

- V and π could be any functional form (e.g. linear $V = \alpha + \beta \sum_{j=1}^{N} \phi_j x_j$)
- But we want an extremely functional flexible form
- Neural networks are (typically) just simple stacked functions interacted with one another–think very flexilble functions, with many parameters
- Advantage of NN-style flexibility is we can spend degrees of freedom in complicated areas and not in simple areas (as in sparse interpolation)
- I'm giving this short shrift (sorry), but we'll put together a neural network for V and π in Matlab

Problem to Solve

- We'll solve a simple finite-horizon NCG-style problem:

$$V(A,K,t) = \max_{K'} \left\{ \log(AK^{\alpha} - K') + \beta E(V(A',K',t+1)) \right\}$$

$$A' = (1-\rho) + \rho A + \epsilon, \quad \epsilon \sim \mathcal{N}(0, 0.05)$$

- How will we set this up?
  - Define a set of functions that:
    - A function that sets up actor and critic neural networks, defines observations, and trains (Main.m)
    - Initialize an agent (including random draws) (myResetFunction.m)
    - Step an agent forward in time, simulate draws (myStepFunction.m)

Reset Function

```
function [InitialObservation, LoggedSignal] =
myResetFunction()
 % Initial values
 S.K = 250+(rand(1,1)-0.5)*200;
 S.A = 1+rand(1,1)*0.05;
 S.step = 1
 % Return initial environment state variables as
logged signals.
 LoggedSignal.State = [S.K;S.A;S.step];
 LoggedSignal.C=NaN;
 InitialObservation = LoggedSignal.State;
```

Step Function

- See myStepFunction.m
- This is the simulator, it takes in signals and an action and simulates the environment, returns the observations, rewards, and whether or not it is done
- Note: logged signals are known to Matlab, but not agent. Observations are known to agent. In this simulation, they are the same thing. (But could have had hidden state)

Main.m

- See main.M
- Idea: set up a flexible function for the actor and critic, and then send to trainer

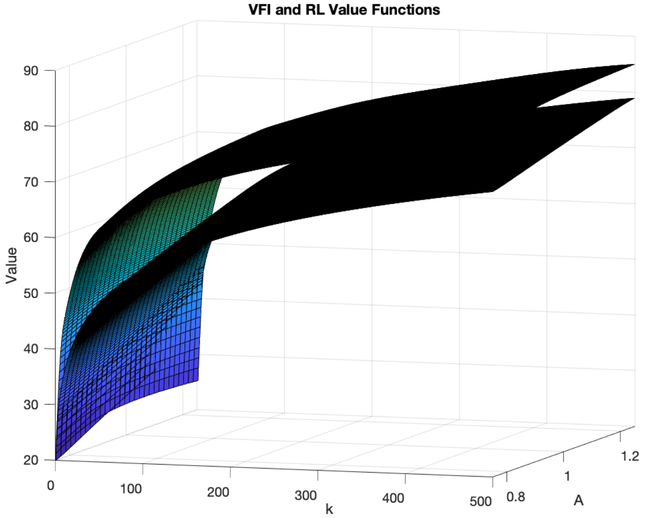
Comparing to Traditional VFI

- There are many options to maximize, can take more time, etc. but let's get a ballpark idea of how well this can do
- Hard to see, so let's look at GraphDiff.m

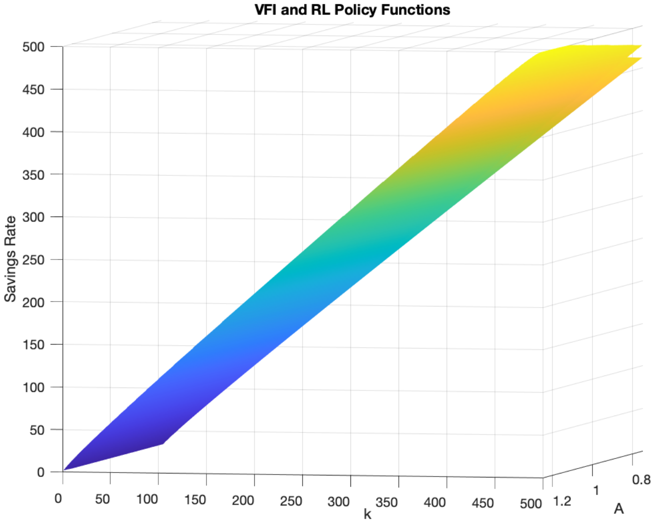
Value Functions are Similar

- There are many options to maximize, can take more time, etc. but let's get a ballpark idea of how well this can do after five hours
- Note: won't be perfect! Feel free to run longer and/or with more parameters
- Hard to see, so let's look at GraphDiff.m

Value Functions

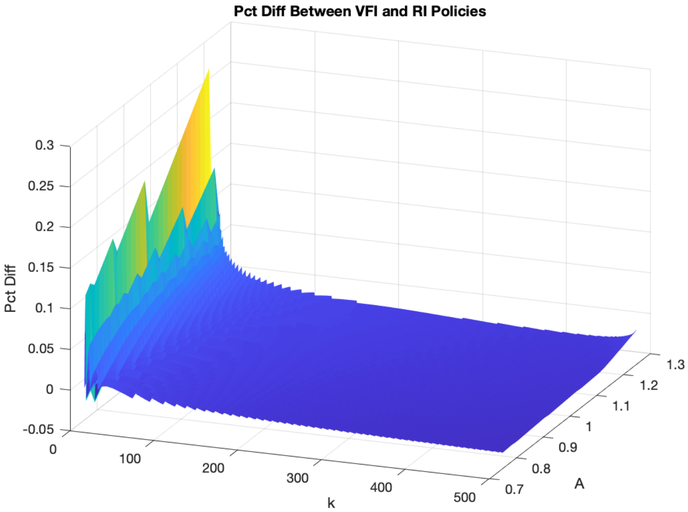


Policy Functions



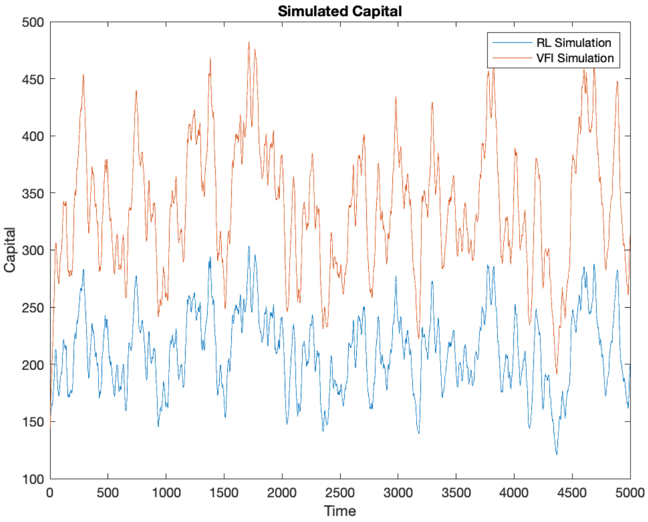
Hard to see, but right on top of one another

Policy Functions Difference



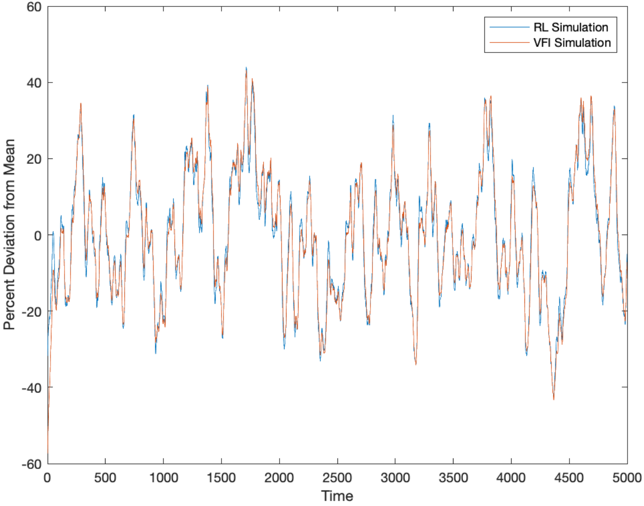
Mostly very small differences...except at k near zero (why?)

Simulation



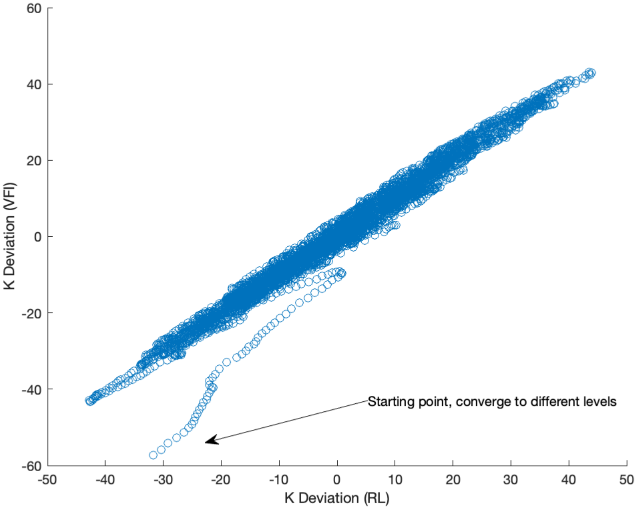
We get towards the right answer in percent deviations (let's check!)

Simulation



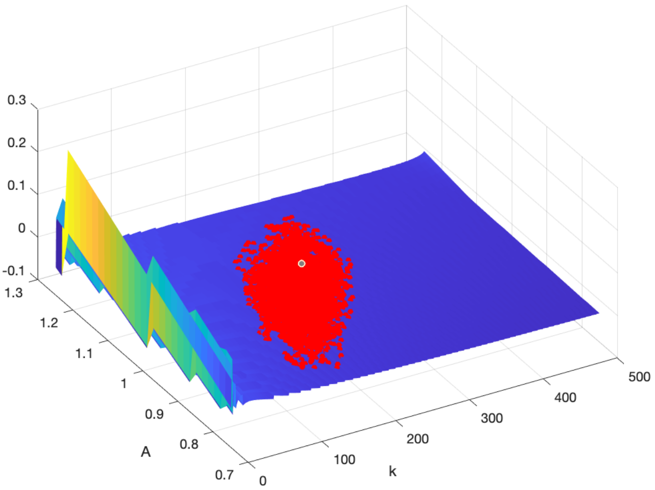
Mostly spot-on in differences

Simulation

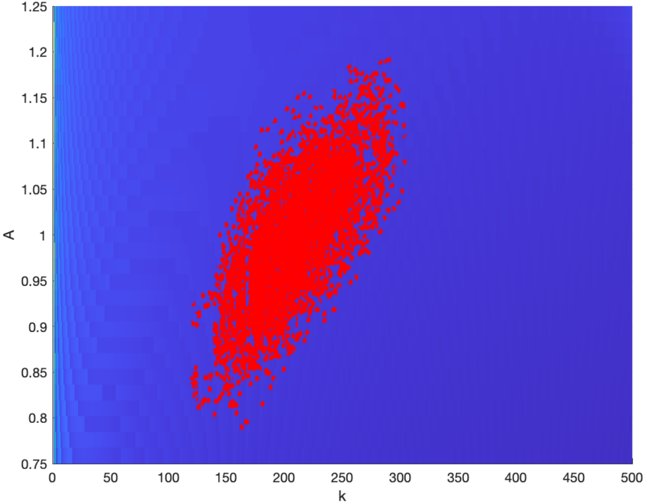


Not terrible, but could use more running & debugging to get level

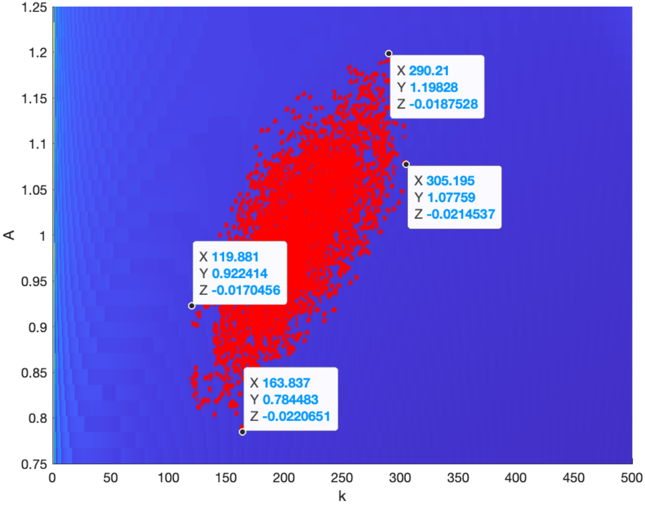
We mostly sample points in the red area, so most accurate there



We mostly sample points in the red area, so most accurate there



Some useful bounds



Summing Up

- Reinforcement learning incredibly useful
- Nothing spectacularly clever, just a combination of:
  - Flexible functional forms (so many state variables can be accomodated efficiently)
  - *Forward*-looking (average over many simulations is estimation)
  - Simple maximization (use gradient and flexible function, so don't have to solve perfectly and solutions potentially informative across state space)
- Could have gone much farther. Could have had parameters α, ρ, etc. be draws too (solve not only over k and A but over parameterization, so can solve for heterogeneous agents or estimate parameters easily! (VFI would require solving for each parameter set).
- In practical terms, you just have a setup function, and then a function that steps through time (simulates) and pass it off to solver

Last warning

- There's a lifetime of details in terms of efficiency, problem setup, solvers, etc., and the devil is in the details
  - Discrete, continuous
  - Shallow RL, deep RL (how setup?)
  - On/off policy
  - Delayed learning
- We went through one algorithm (actor-critic) and one example (continuous state & action space).
- There are a cornucopia of flavors & algorithms–I haven't yet found one that isn't intuitive & obvious in retrospect (once you grok the core idea behind approximate dynamic programming idea)
- However, economist notation and ML notation diverge a bit, so there's an investment in learning