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

Neural Networks

- Neural networks, for our purposes, are just very very flexible nonlinear functional forms
- Have y^{data} and want to fit $f(x|\theta)$ to it
- One way of producing a flexible function with easy derivatives (for fitting) is to stack logit functions
- Idea: $X = [x_1, x_2, ...]$ gets fed into multiple logits $f_1(X)$, $f_2(X)$, $f_3(X)$, each with its own logit weight on X: β_1^f, β_2^f, β_3^f.
- The outputs of these three $(+)$ logits then (possibly) get fed into another set of logits, $g_1(f_1, f_2, f_3)$, $g_2(f_1, f_2, f_3)$, each with their own set of weights β_1^g...
- Eventually these are summed up or scaled to a single (or multiple!) output, such as y^{fit}
- For most uses, NN just find θ like we would in any fitting problem, but use gradient descent (chain rule!) rather than Newton's method (too many cross-derivatives $+$ many local minima)

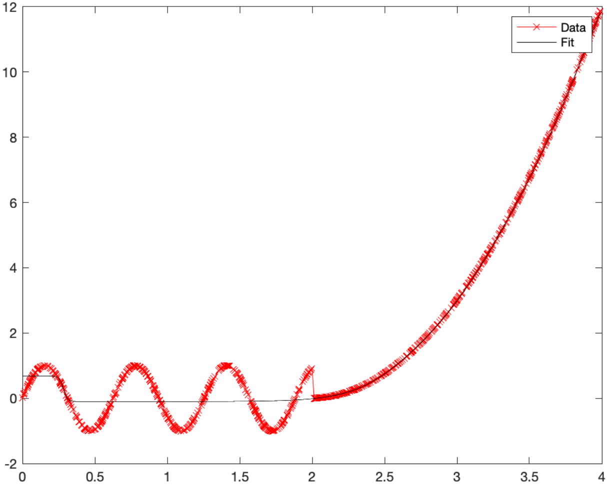
Neural Networks

- Concrete example: want to approximate $f(x)$
- Suggestion: one set of five logits, whose sum is then scaled:

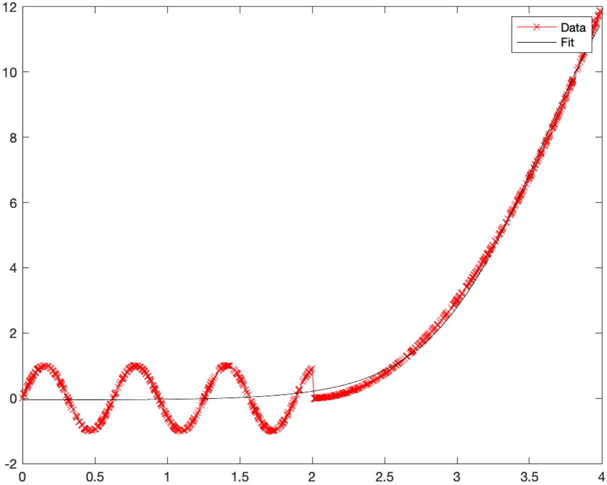
$$f(x) \approx \alpha_{L2} + \left(\sum_{i=1}^{5} \beta_{L2,i} \frac{1}{1 + e^{-(\alpha_{1,i} + \beta_{L1,i} x)}} \right)$$

- A little hard to read!
- See Simple_5_Main.m, and then Simple_N_Main.m for N logits

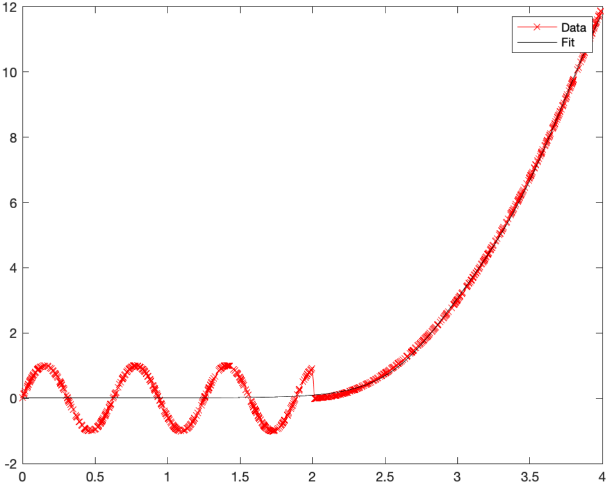
Neural Networks: 5 Logit+Linear Approx



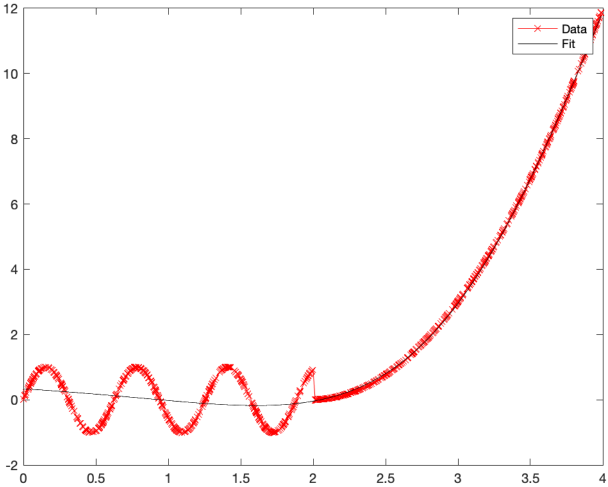
Neural Networks: 1 Logit+Linear Approx



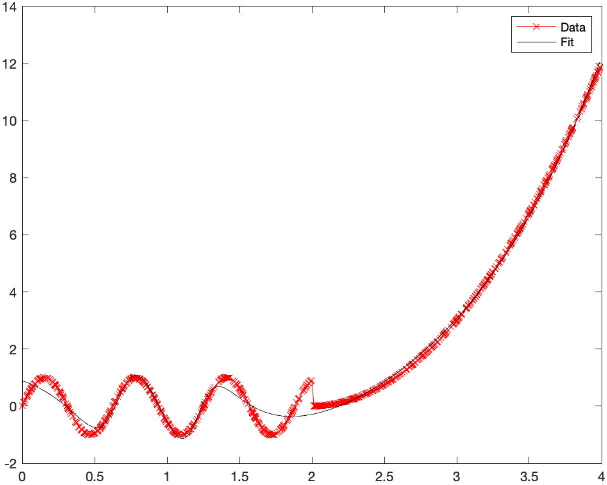
Neural Networks: 2 Logit+Linear Approx



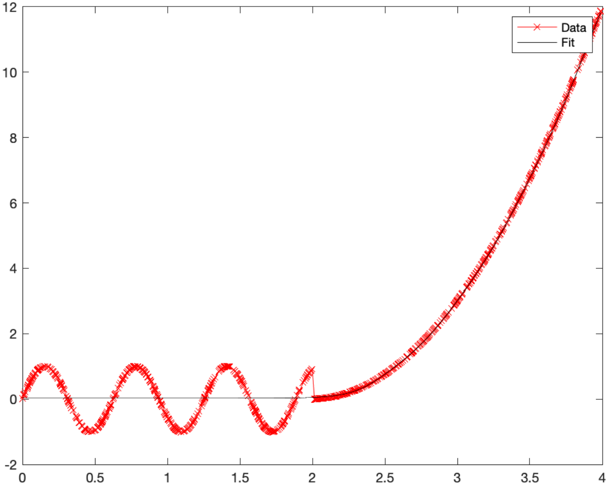
Neural Networks: 3 Logit+Linear Approx



Neural Networks: 4 Logit+Linear Approx

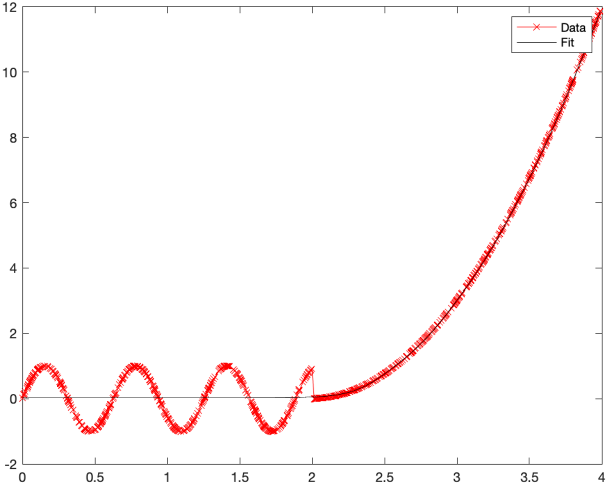


Neural Networks: 6 Logit+Linear Approx



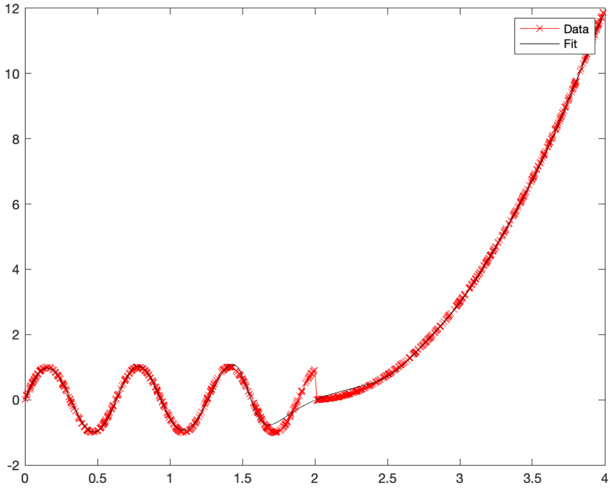
Got stuck at local minima! (Bad!)

Neural Networks: 7 Logit+Linear Approx

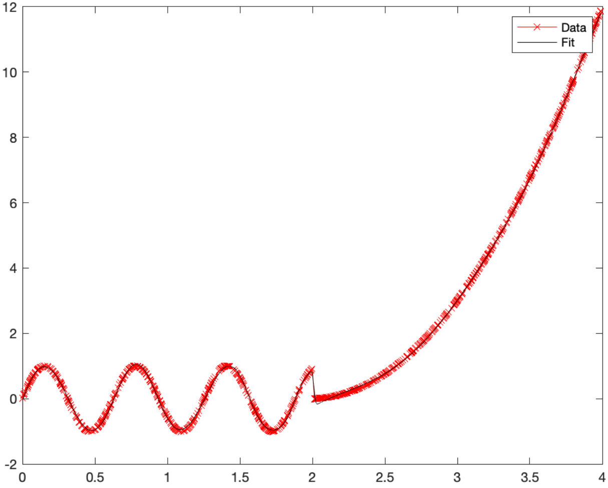


Got stuck at local minima! (Bad!)

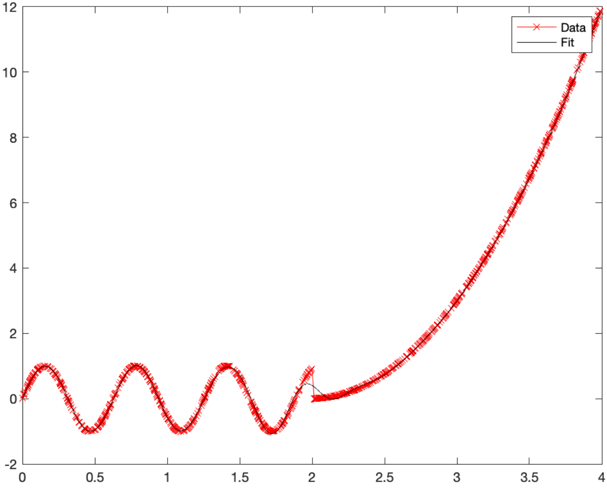
Neural Networks: 8 Logit+Linear Approx



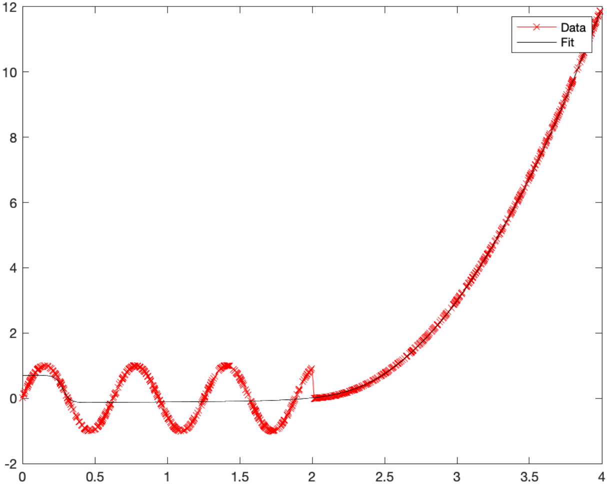
Neural Networks: 9 Logit+Linear Approx



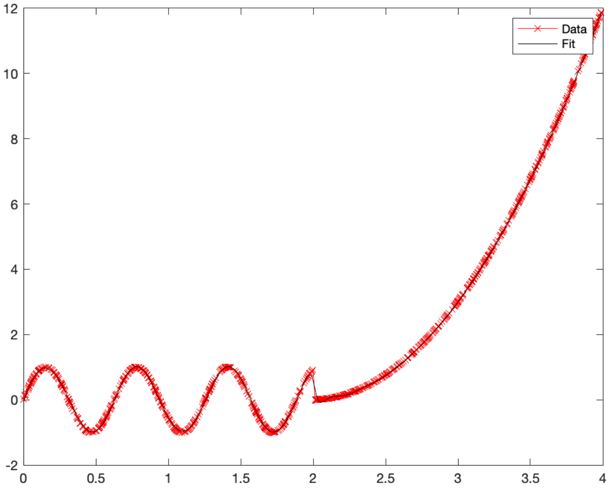
Neural Networks: 10 Logit+Linear Approx



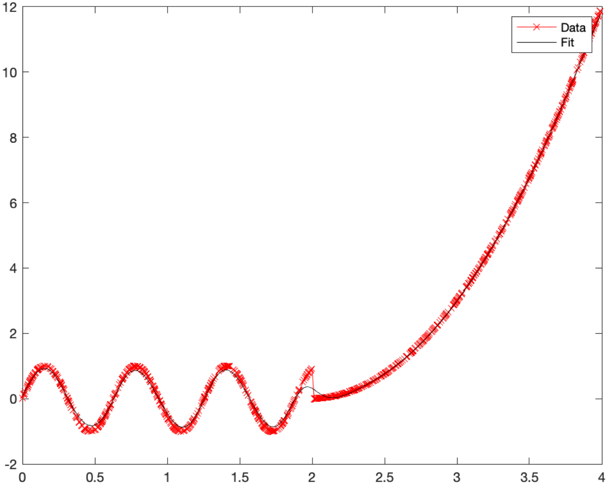
Neural Networks: 11 Logit+Linear Approx



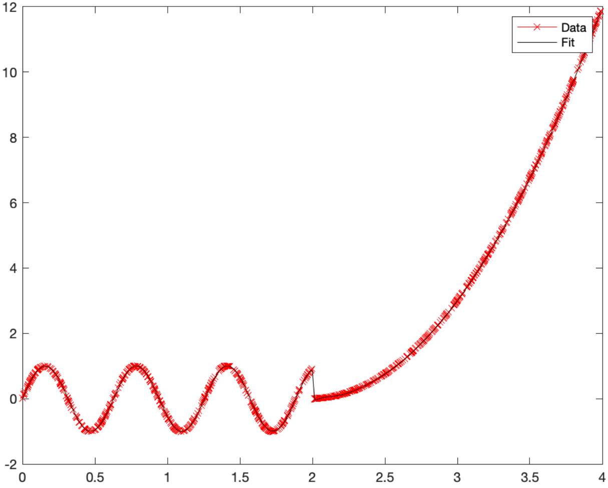
Neural Networks: 12 Logit+Linear Approx



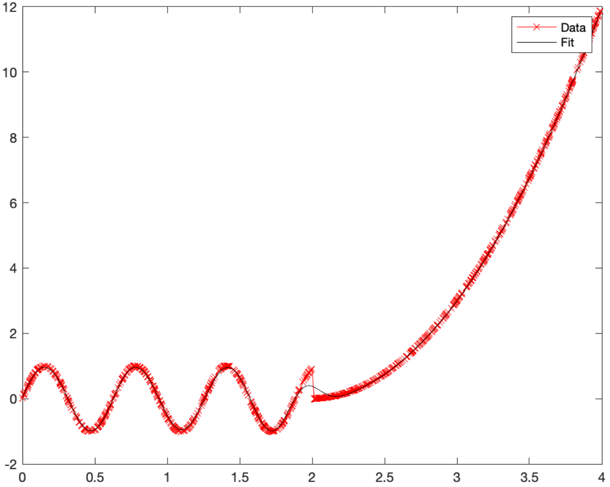
Neural Networks: 13 Logit+Linear Approx



Neural Networks: 14 Logit+Linear Approx



Neural Networks: 15 Logit+Linear Approx



Deep Learning

- I think of Neural Networks as really flexible sets of if statements combined with linear functions.

- Logits act as "if's"

- But could have a doubly-stacked if statement:
  - First layer:

$$f_{a,1}(x_1, x_2) = 1 \iff x_1 > 3$$

$$f_{a,2}(x_1, x_2) = 1 \iff x_2 < 1$$

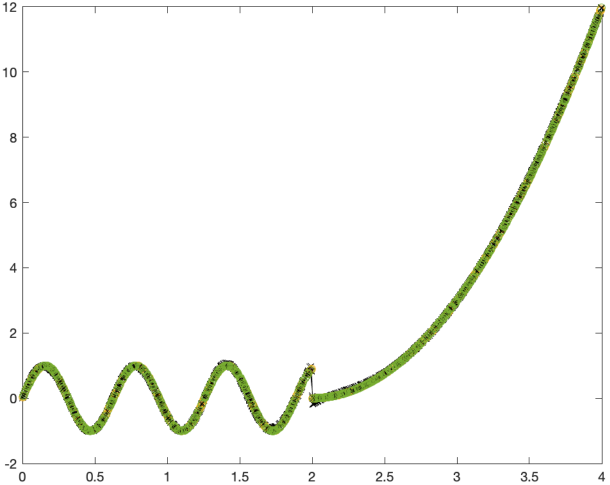
  - Second layer:

$$f_b(f_{a,1}, f_{a,2}) = 1 \iff f_{a,1} == 1 \ \& \ f_{a,2} == 1$$

- Idea: multiple layers give ability to encode complex “if” statements “easily”

- Let's see my stacked version! Deep_N_Main.m

Neural Networks: 5x5 Logit+Linear Approx



Neural Networks

- Can generalize this, add more layers of logits or other functions
- Because minimized with gradient descent, we want things that we can use chain rule on
- Because multiple minima, want things that are bounded
- Some common ones:
  - "Rectifier" $f(x) = max(0, x)$ (helps for piecewise)
  - Hyperbolic tangent $tanh(x)$ (similar to logit/"sigmoid", but -1 to 1)
  - Scaling layer: $\alpha + \beta X$
  - Softmax/"categorical distribution": vector output, takes vec x and spits out $\frac{exp(x_i)}{\sum_j exp(x_j)}$
  - Less relevant in most econ: convolution, pool "nearby" observations
  - LSTM: let value today affect fits for tomorrow (so $f(X_2)$ (X matrix at $t = 2$) is a function of X_2 and X_1), all the way back
  - Dropout layer: randomly drop inputs (train drunk!)s

Matlab: build in shallow NN

- Matlab has a lot of built-in neural network stuff
- Unfortunately(?) this is an evolving field, and so commands are changing, becoming obsolete, etc.
- Can be frustrating, but equivalent to Theano dying, Google transitioning from TensorFlow to JAX(?)
- But can be as simple as:

  ```
  net = fitnet(netsize);
  net = train(net,Xdata,Ydata,'UseParallel','yes');
  ```
- Just like we write before, a bunch of logits that are netsize, give it X and Y data, minimizes.
- Great for predicting, obviously **not solution for causality** (sorry micro ppl)

Example to fit: $x.^2 + y.^2 - 2sin(x. * y)$

- See Fitnet Example/Main.m for details!

The Three Curses of Dimensionality

- 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 $\overline{Q}(x, y|\phi)$ takes in state and spits out value (Q-function, value function would be $\overline{V}(x|\phi) = \overline{Q}(x, \overline{\pi}(y|x;\theta)|\phi))$

- We can represent Actor & Critic as flexible neural networks parameterized by θ and ϕ but how get values to fit?

- Good actor function embeds both reward and stochastic future (actions and integral!)

- *Given* θ and ϕ, can simulate an agent, get data to fit

- Need to find θ and ϕ

ACTOR-CRITIC ALGORITHM (**sketch(!)**)

- Start with a guess for θ and ϕ, and an initial state value
- Simulate the system, get data
- Now we have a bunch of paths for a given θ and ϕ
- Define "disappointment" as difference between what we thought we would get and what we actually got, using our critic to fill in for the future:

$$D_t = reward_t + \overline{V}(x'|\phi) - \overline{Q}(x, y|\phi)$$

- Update the critic to minimize the sum of squared errors between expectation and experience

$$d\phi = \sum_{t=1}^{N} \nabla_{\phi}(D_t(\phi))^2$$

- Update the actor using the chain rule: θ affects A, A affects Q.

$$d\theta = \sum_{i=1}^{N} \nabla_y Q(x, y; \phi) \nabla_{\theta} \overline{\pi}(y|x; \theta)$$

- Idea: $d\theta$ 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:

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

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

Detail on the Gradient

- Let's say that $Q(x, y) = abs(y - x)$, and we know the true Q
- So optimal action function is $y^* = x\ \phi^{true} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
- But we start with a bad action function: $y^* = 0.5 + 2 * x$.
  $\phi = \begin{bmatrix} 0.5 \\ 2 \end{bmatrix}$
- We sample the point $x = 1$, and choose $y = 2.5$
- Then $\nabla_y Q(x, y; \phi) = -1$, b/c linear penalty to missing target
- And $\nabla_\theta = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- Multiplying together, we get:

$$d\phi = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

- Then, if we put a weight of 0.01 on the update we get:

$$\phi = 0.99 \begin{bmatrix} 0.5 \\ 2 \end{bmatrix} + 0.01 \begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.485 \\ 1.97 \end{bmatrix}$$

- Closer to the truth!

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

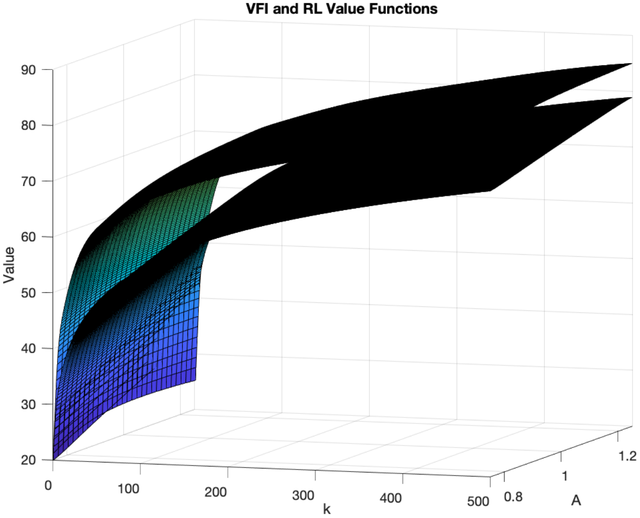
Comparing to Traditional VFI

- 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

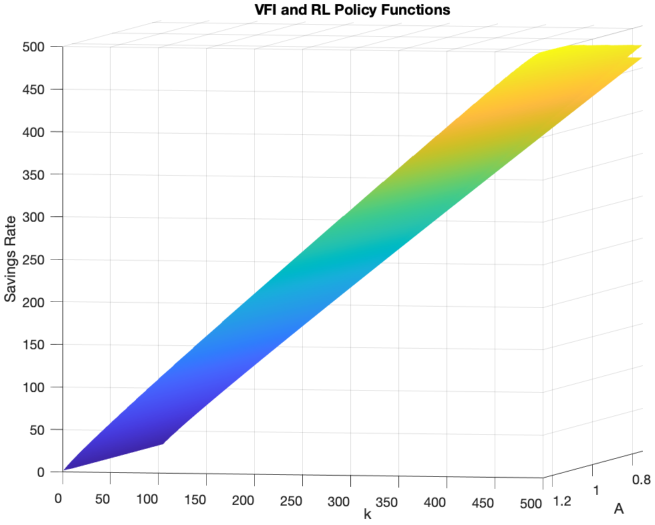
Value Functions are Similar

- 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

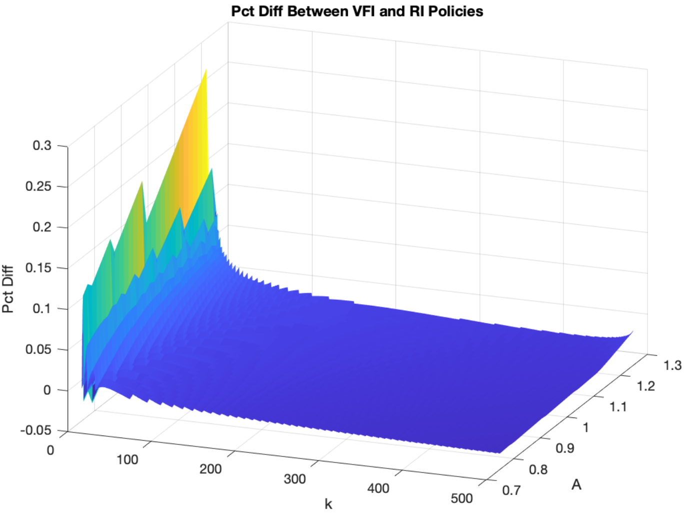


Policy Functions



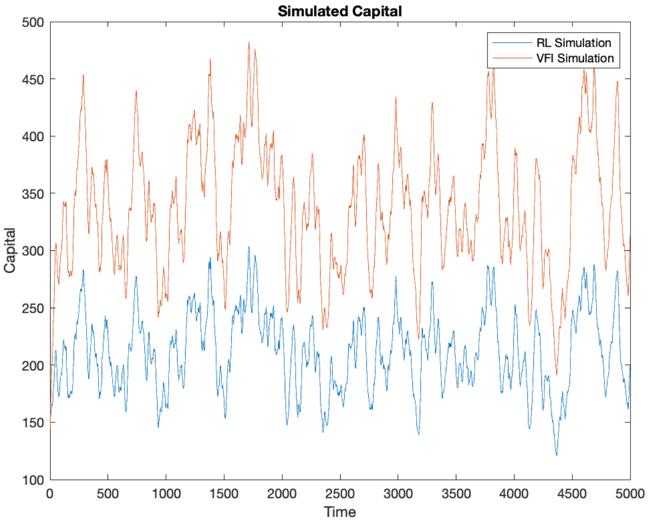
Hard to see, but right on top of one another

Policy Functions Difference



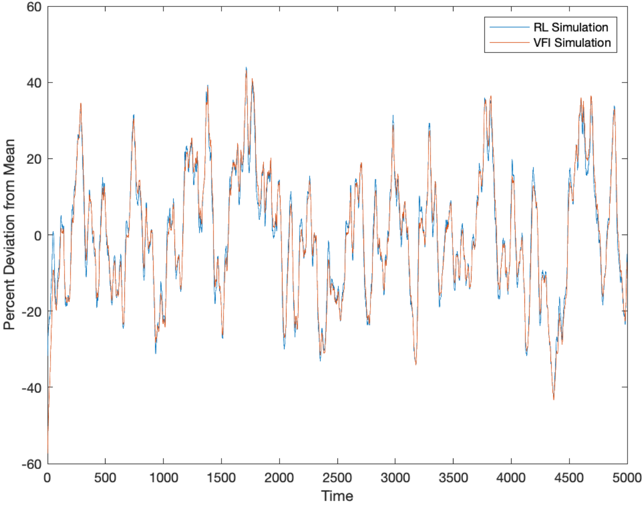
Mostly very small differences...except at k near zero (why?)

Simulation



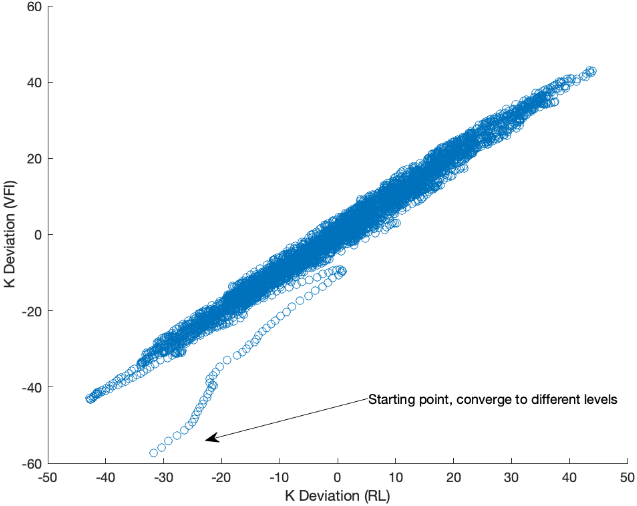
We get towards the right answer in percent deviations (let's check!)

Simulation



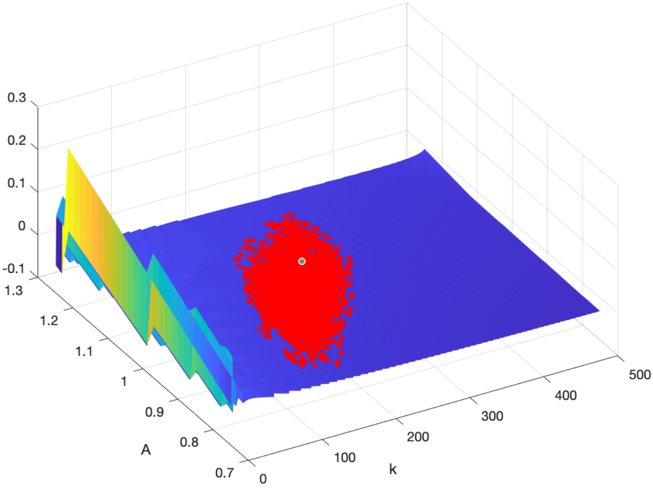
Mostly spot-on in differences

Simulation

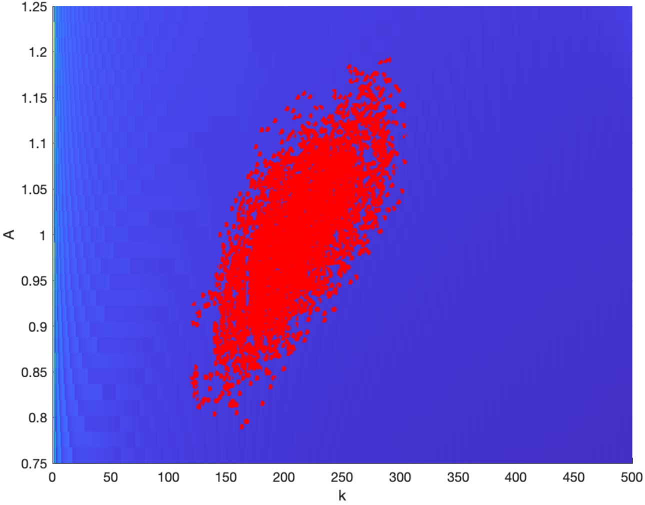


Not terrible, but could use more running & debugging to get level

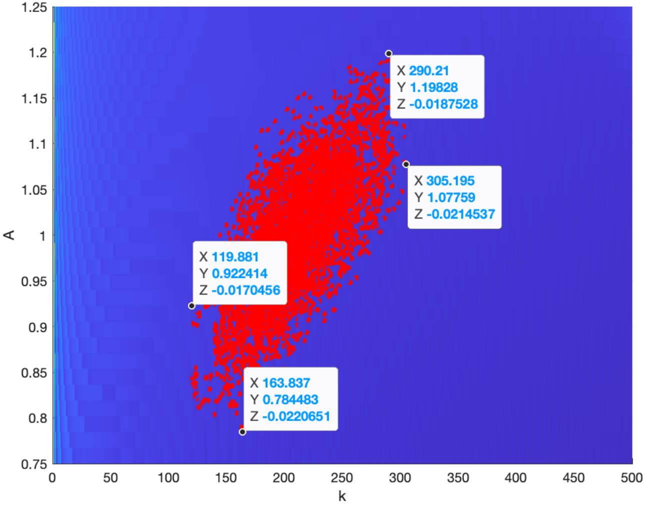
We mostly sample points in the red area, so most accurate there



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

Other ML Topics

- We talked about reinforcement learning: have a problem, throw it at computer over and over, have computer solve it
- There are other things you might care about: highly nonlinear functions more generally (Kriging/gaussian process regression "fitrgp")
- Classification learning (such as support vector machines "fitcsvm")
- Regression learner
- We won't talk about these, instead we'll start applying!