Appendix B: Examples of Deep RL in Ecological Decision Problems

A Preprint

Marcus Lapeyrolerie
Department of Environmental Science, Policy, and Management
University of California, Berkeley
Berkeley, California

Melissa Chapman
Department of Environmental Science, Policy, and Management
University of California, Berkeley
Berkeley, California

Kari Norman
Department of Environmental Science, Policy, and Management
University of California, Berkeley
Berkeley, California

Carl Boettiger
Department of Environmental Science, Policy, and Management
University of California, Berkeley
Berkeley, California
cboettig@berkeley.edu

May 27, 2021

In this appendix, we illustrate how to apply a deep reinfocement learning framework to train previously published RL algorithms on the novel decision environments as illustrated in the main text. Appendix C discusses the construction of such environments in code.

1 Deep Reinforcement Learning Frameworks

At the time of writing, several major frameworks exist which provide convenient access to baseline algorithms commonly used in deep reinforcement learning. While we do not seek to provide a comprehensive review of available frameworks, some familiarity with current frameworks can be helpful. Like many machine learning libraries, these frameworks are themselves each built around one of two popular machine learning libraries, PyTorch (**torch?**) or Tensorflow (**tensorflow?**), and all are based in Python. Existing frameworks we evaluated include Keras-RL (**keras-rl?**), Tensorflow Agents (**tensorflow-agents?**), OpenAI Spinning Up (**spinningup?**) OpenAI Baselines, Stable-Baselines2 (**sb2?**), and Stable-Baselines3 (Raffin et al. 2019). Keras-RL saw widespread early adoption, but is built on Tensorflow version 1.x. It is incompatible with

Tensorflow 2.x and not actively maintained. Tensorflow Agents is developed by the Tensorflow team, a recent and actively developed framework with support for both Tensorflow 1.x and 2.x and good support for low-level customization of RL algorithms. However, higher-level interfaces and high-level documentation are still relatively limited. OpenAI's SpinningUp is an education-targeted framework useful for developers wanting to become more familiar with the internal methods of RL algorithms.
OpenAI's Baselines is primarily Tensorflow 1.x-based implementation of many recently published RL algorithms. Researchers at Ensta Paris Tech first created Stable Baselines as a fork of the OpenAI implementation, rigorously addressing numerous issues in documentation, testing, and coding style that have helped make their fork see even greater adoption. Stable-Baselines3 is the most recent version (Feb 2021), a ground-up rewrite which switches to a PyTorch-based implementation and further strengthens internal checks such as static types. The examples here all use the Stable-Baselines3 framework, though researchers in this area should expect frameworks to continue to emerge and evolve. Grand challenge problems will likely require significant development beyond the current algorithms and capabilities available in existing frameworks.

2 Deep Reinforcement Learning in R

Although all the necessary tooling for RL is implemented in Python, the R language is more familiar to most ecologists. Fortunately, modern bindings such as the `reticulate` package (**reticulate?**) make it straightforward to use these tools without ever leaving the R interface. In this appendix, we detail this "pure R" approach, as well as a "pure Python" approach.

In the R based-approach, R functions take responsibility from the user for translating commands into Python code before it is executed, an approach commonly referred to as meta-programming. This still requires a local Python installation, which can be installed directly from R using the command `install_miniconda()` from the `reticulate` package in R. Alternately, users may prefer running the analysis inside a docker container. The Rocker Project (**rocker?**) provides pre-built docker containers which include the necessary R and Python environments, as well as CUDA libraries required to take advantage of GPU-based acceleration on suitable architectures. This can be particularly useful for users running ML algorithms on remote servers, as configuring R, Python, and CUDA environments appropriately can be challenging.

Clone the repository `https://github.com/boettiger-lab/rl-intro`, e.g. using the New Project->From Version Control->Git menu in RStudio. Note that the RMarkdown source-code for this file can be found in the `appendices` directory of the project repository. From the project directory, we can then install all the necessary dependencies using `renv`, which helps ensure a reproducible environment of fixed versions of R packages and Python modules.

```r
#install.packages("renv")
renv::restore()
```

```
## * The library is already synchronized with the lockfile.
## * The Python library is already up to date.
```

Once the packages have installed, we are ready to load the necessary libraries. Note that the `import` function from `reticulate` package acts much like the `library` command in R, though it does not attach the package function to the global namespace. To make it more convenient to access those functions, we can assign a shorthand name.

```r
# R dependencies
library(tidyverse)
library(patchwork)
library(reticulate)

## Python dependencies loaded via R
sb3         = import ("stable_baselines3")
gym         = import ("gym")
gym_fishing = import("gym_fishing")
gym_conservation = import("gym_conservation")

#
source("../R/plotting-helpers.R")
```

Numerical reproducibility can be challenging in machine learning problems, particularly when using GPU-based acceleration. In addition to setting a random seed in our Python environment, we can optionally disable GPU use to improve reproducibility by setting the `CUDA_VISIBLE_DEVICES` to a blank value.

```r
## reproducible settings
np = import("numpy")
seed = 24L # integer
np$random$seed(seed)

# Optionally set to "" to force CPU-evaluation if needing perfect reproducibility
Sys.setenv("CUDA_VISIBLE_DEVICES"="")
set.seed(seed)
```

The above code also illustrates a few conventions which may be helpful to bear in mind when using the `reticulate` interface to interact with Python from R: it is often necessary for integer values to be explicitly typed as integers by adding a trailing `L` (corresponding to the primitive C type `long` integer). Python is also more strongly object-oriented than many R packages, where "methods" of an "object" are accessed with the list-subset operator `$` in R (equivalent to the use of `.` in Python). Lastly, while R can use `<-` or `=` for assignment, Python uses only `=`. For simplicity we will stick with the latter.
With few exceptions, the R code shown here can be re-written as python code by dropping the `L` and replacing `$` with `.`, see stand-alone python code in the `python/` subdirectory of the repository.

3 Finding a known optimal solution using RL

3.1 Sustainable Harvest Quotas

We begin by selecting an environment from `gym_fishing` by name, passing the optional parameter `sigma` to initialize the environment with multiplicative Gaussian environmental noise of sd of `0.1`.

```r
## initialize the environment
env = gym$make("fishing-v1", sigma = 0.1)
```

3.2 An optimal solution

Recall that under the assumptions of the simple model used in the `fishing-v1` environment, we can determine the optimal harvest policy analytically if the model and parameters are known precisely (**Reed1979?**). The optimal strategy is a policy of 'constant escapement,' designed to keep the remaining stock size (the population that 'escapes' fishing harvest) at the biomass corresponding to a maximum growth rate, i.e. at $B_{MSY} = K/2$ in this model. `gym_fishing` defines a collection of non-RL agents in the `models` submodule, including the a human agent that merely asks to enter their desired quota manually. The `escapement` model implements the provably optimal constant escapement rule. A third model, `msy`, implements a policy based on "Maximum Sustainable Yield" policy (**Schaefer1954?**), which is actually more commonly used as a basis for management than constant escapement, despite only being optimal at the steady state under deterministic dynamics.

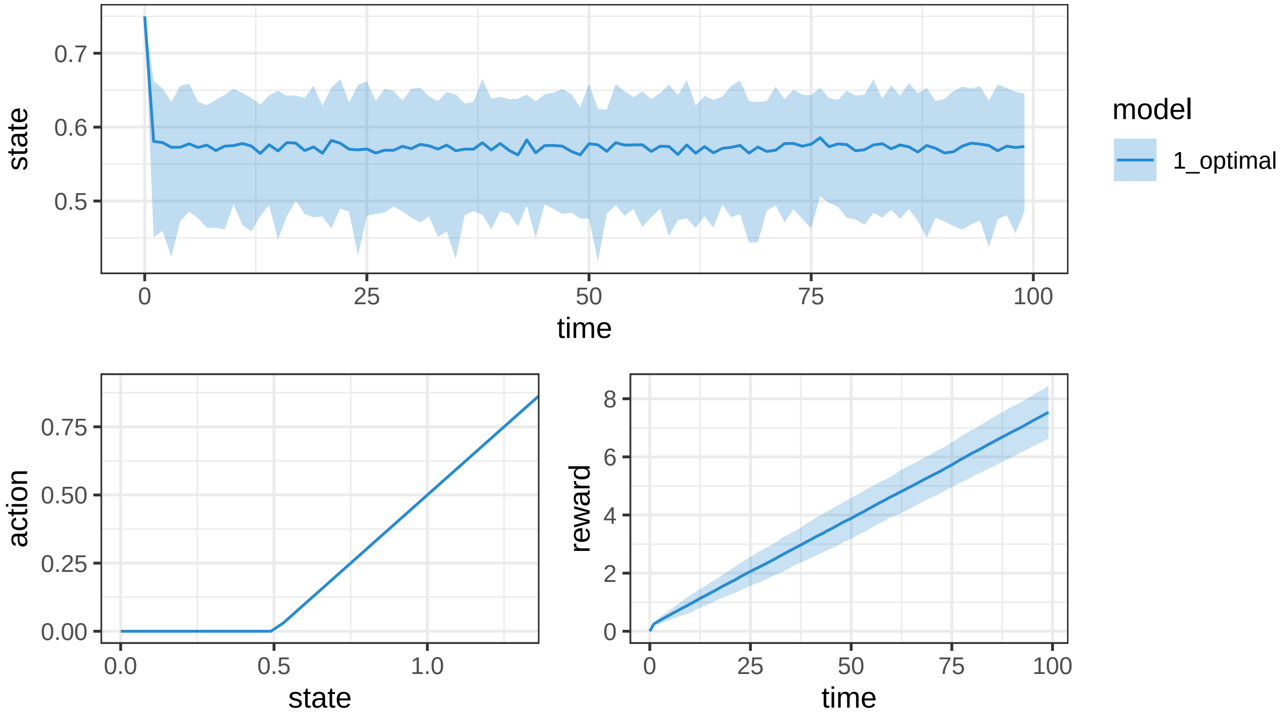
```r
# Simulate under the optimal solution (given the model)
opt = gym_fishing$models$escapement(env)
```

Note that the `escapement` function makes very specific assumptions about the environment that it is given. In particular, it assumes we can compute B_{MSY} directly from the internal model of stock recruitment contained in the environment. In contrast, the RL methods make no such assumption of being able to access that internal model directly.

```r
opt_sims = env$simulate(opt, reps = 100L)
opt_policy = env$policyfn(opt)
```

We can plot the resulting data.frames using standard `ggplot2` methods shown in the `R/plotting-helpers.R`, or in python using either the `plot` methods defined in `gym_fishing` or standard plotting libraries.

```
sims_df <- opt_sims %>% mutate(model = "1_optimal")
policy_df <- opt_policy  %>% mutate(model = "1_optimal")

plot_sims(sims_df) / ( plot_policy(policy_df) + plot_reward(sims_df))
```



3.3 RL-based solutions

We will compare this optimal solution with results from the several different RL algorithms. Unlike the optimal solution, the RL methods never know the underlying growth function used by the simulation (or even if such a function actually exists). These methods merely seek to learn a strategy for setting harvest quotas which maximizes the cumulative reward the receive from the environment. At the heart of each deep RL algorithm is a neural network or a collection of neural networks. The RL agent uses these networks to approximate a policy function and/or a value function. The policy function is the agent's mapping of states to actions – i.e. the agent's strategy. Value functions attempt to evaluate the goodness of a state or state-action pair towards achieving the RL objective of maximizing cumulative rewards. Confusingly, the parameters of a RL algorithm that detail the overall learning process are called *hyper-parameters* while the parameters of a RL algorithm that are learned during training are called parameters. For example, the number of layers in a neural network is a hyper-parameter but a weight in the neural net is referred to as a parameter. Before training an agent, we must first specify the hyper-parameters; the algorithm's parameters are often randomly initialized. Each algorithm may have different hyper-parameters. `stable-baselines3` provides default values for all hyper-parameters based on the original papers that introduced the corresponding algorithms, for instance, the A2C algorithm originally described in Mnih et al. (2016). Here, we train an agent using the A2C algorithm using default hyper-parameter settings with a multi-layer perceptron policy and value network composed of two 64-neuron layers.

```
a2c = sb3$A2C('MlpPolicy', env, verbose=0L, seed = seed) # L indicates a (Long) integer, not floating p
```

Training is the main computationally intensive process, which can take anywhere from a few minutes to many days, depending on the complexity of the environment, the neural network architecture and the number of training iterations budgeted. Therefore, we save the trained agent to disk, and only execute the training method (`learn`) if no previously saved agent is found:

```r
if(!file.exists("cache/a2c.zip")) {

  # Train the agent for a fixed number of timesteps
  a2c$learn(total_timesteps=300000L)

  # Save our trained agent for future use
  a2c$save("cache/a2c")
}
```

Note that while default hyper-parameters provide a useful starting place (particularly when the environment has been suitably normalized, a best-practice we discuss above), better performance can almost always be achieved by *tuning* the hyper-parameter selection. This is discussed further below. Having saved our trained agent to disk (`cache/a2c.zip`), we can then re-load this agent for evaluation or to continue training. Note that a copy of the trained agents are included in the corresponding GitHub repository.

```r
# Simulate management under the trained agent
a2c = sb3$A2C$load("cache/a2c")
```

We can supply an observation to our trained agent and it will return it's proposed action using the `predict` method. This is all we need to evaluate or employ the agent on a decision-making task. Recall that state space and action space in the fishing gym have been re-scaled to a (-1, 1) interval. Note that this is equivalent to a choice of appropriate units – we can re-scale the interval without loss of generality. This is often an important step in the design of an RL environment to facilitate successful training. Following the `gym` standard, the core methods such as `predict` and `step` operate on the re-scaled units, so it is necessary to first transform the original units into this re-scaled state space. For example, if we wish to start a simulation with a stock size of 0.75, we can use the helper method `get_state()`, to determine the corresponding value in the re-scaled state space.
Unlike `predict` and `step`, `get_state()` is not a standard method of all gym environments – typically a user must first inspect the state space of an environment and choose themselves how to re-scale their problem into that state space.

```r
## represent the initial state size in the 'rescaled' state space.
state = env$get_state( 0.75 )
state
```

```
## [1] -0.25
```

With an initial state in hand, we are ready to simulate management using our agent. The iteration is simple: we use the agent to predict what action we should take given the current state. Then, we take said action and examine the result to determine the future state. Because these methods return additional information as well, a little extra sub-setting is required in R:

```r
for(i in 1:10){

  out = a2c$predict(state)
  action = out[[1]]
  result = env$step(action)
  state = result[[1]]

}
```

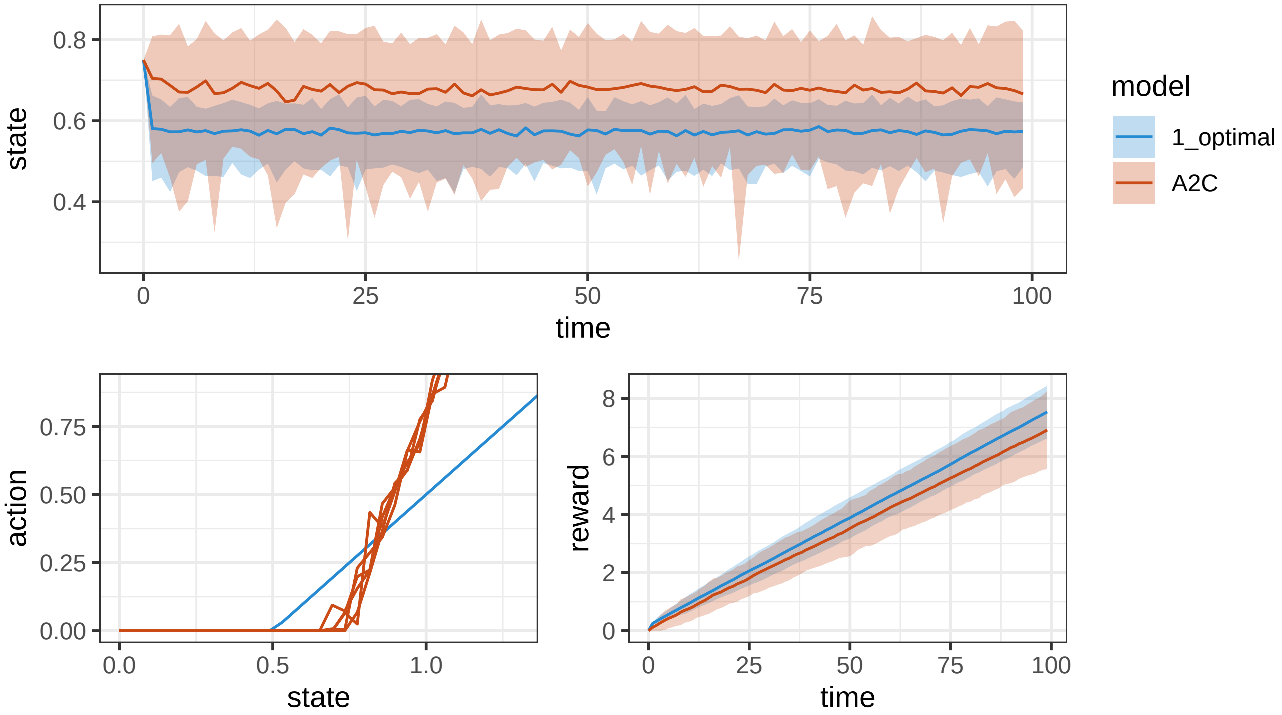
For convenience, `gym_fishing` defines the helper routine `simulate` to perform the above iteration `reps` number of times. The `simulate` method returns the state, action, and reward resulting from each time step of each replicate:

```r
a2c_sims = env$simulate(a2c, reps = 100L)
```

Because the agent typically attempts to retain the fish population near a specific value, simulations with well-trained agents will usually not explore the full range of possible states. To get a better idea of how the agent behaves across the full state space, it is common to examine the policy function: to do this, we'll plot the action taken at every possible state. Some agents are non-deterministic, so we may want to use replicate draws at each state to get a better picture of the agent's behavior. Note that the `policyfn` method is another custom method and not part of the `gym` standard. The use of `policyfn` will not make sense for agents that consider the history of many previous states in selecting their action, such as agents which utilize recurrent neural network architectures like LSTMs (**LSTM?**).

```
a2c_policy = env$policyfn(a2c, reps = 5L)
```

```
sims_df <- a2c_sims %>% mutate(model = "A2C") %>% bind_rows(sims_df)
policy_df <- a2c_policy  %>% mutate(model = "A2C")  %>% bind_rows(policy_df)

plot_sims(sims_df) / ( plot_policy(policy_df) + plot_reward(sims_df))
```



3.4 TD3

We train a second agent based on the TD3 algorithm. Recall that the Twin Delayed Deep Deterministic Policy Gradient, commonly known as TD3 (Fujimoto, Hoof, and Meger 2018), is a generalization of Deep Deterministic Policy Gradient, or DDPG, which extends the DQN algorithm Mnih et al. (2015) to a continuous action space using the deterministic policy gradient. DDPG and DQN are also available in the stable baselines framework (Raffin et al. 2019). One of the most salient differences here is that A2C uses an Actor-Critic strategy that combines policy gradient and value iteration approaches, while TD3 is a policy gradient approach.

We will start with the default hyperparameters, which correspond to those used in Fujimoto, Hoof, and Meger (2018), where the algorithm was first introduced.

```
td3_untuned = sb3$TD3('MlpPolicy', env, seed = seed)
```

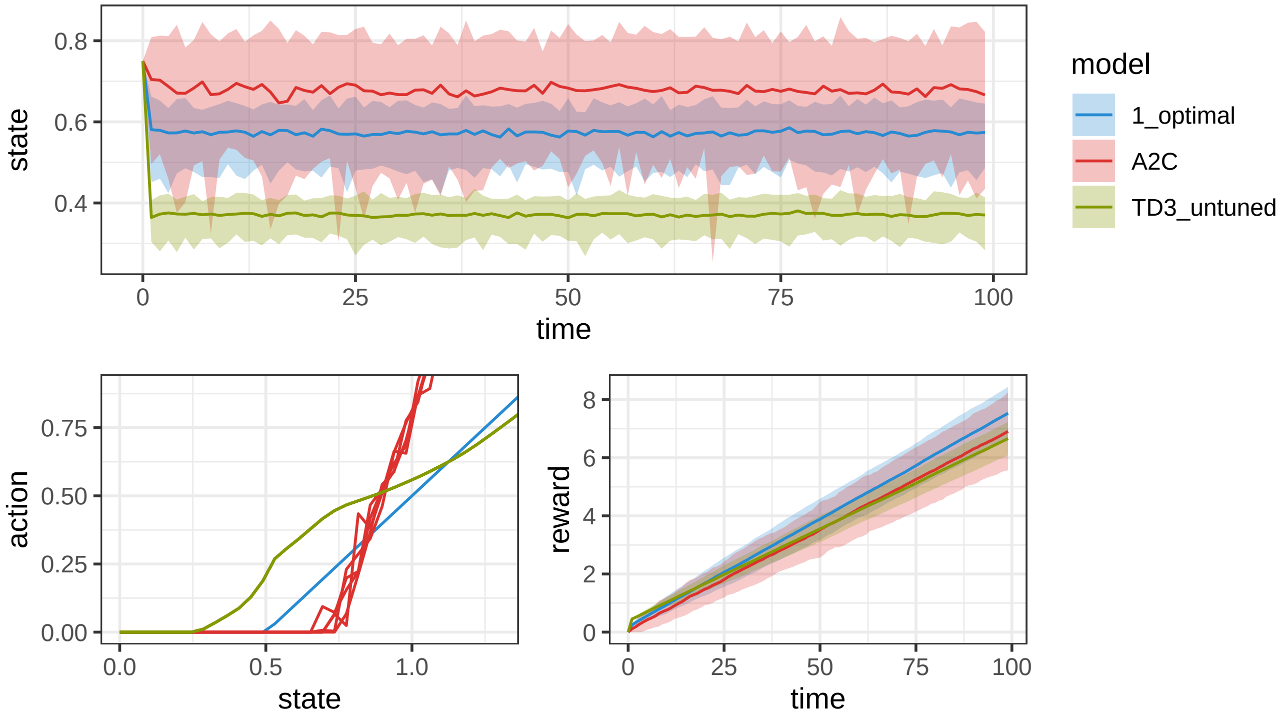
Once again, after our agent is defined we are ready to train it:

```
if(!file.exists("../python/cache/td3_untuned.zip")){

  td3_untuned$learn(total_timesteps=300000L)
  td3_untuned$save("../python/cache/td3_untuned")

}
```

With our trained agent, we can simulate replicate scenarios or scan across states to determine the implicit policy function:

```
# Simulate management under the trained agent
td3_untuned = sb3$TD3$load("../python/cache/td3_untuned")
td3_sims_untuned = env$simulate(td3_untuned, reps = 100L)
td3_policy_untuned = env$policyfn(td3_untuned, reps = 5L)
```

```
sims_df <- td3_sims_untuned %>% mutate(model = "TD3_untuned") %>% bind_rows(sims_df)
policy_df <- td3_policy_untuned  %>% mutate(model = "TD3_untuned")  %>% bind_rows(policy_df)

plot_sims(sims_df) / ( plot_policy(policy_df) + plot_reward(sims_df))
```



The untuned TD3 policy tends to overfish relative to the optimal solution, but is less variable than the A2C solution. While it is possible that with much more training, this TD3 algorithm could further improve, subsequent learning may be difficult.

Note that the reward under TD3 is nevertheless often within the margin of error of the optimal policy, which may limit future learning since further policy exploration is more likely to reduce rewards. Instead, we may be able to improve performance of TD3 by choosing different hyperparameters, which may allow it to imagine more possible policies or learn more effectively.

3.5 Tuned TD3

Some of these hyper-parameters of the TD3 algorithm, such as the `learning_rate` and discount rate, `gamma`, are also hyper-parameters of the A2C algorithm. Others, like the `action_noise` describing how new actions are proposed, are not used in A2C. Good choices for hyper-parameters can be found through a process known as "hyper-parameter tuning," which uses standard function optimization approaches to vary the choice of hyper-parameters to determine which hyper-parameters maximize the agent's performance. For simplicity, we show only training with specified hyper-parameters here, found by various tuning experimentation (see `https://github.com/boettiger-lab/conservation-agents`) and compare to the tuning utilities found in `stable-baselines3-zoo` (**zoo?**).

```
# train an agent (model) on one of the environments:

policy_kwargs = list(net_arch=c(400L, 300L)) # "big"
# non-episodic action noise:
noise_std = 0.6656948079225263
n_actions = env$action_space$shape[0]
action_noise = sb3$common$noise$NormalActionNoise(
        mean=np$zeros(n_actions),
        sigma= noise_std * np$ones(n_actions)
        )

td3 = sb3$TD3( 'MlpPolicy',
               env,
               verbose=0L,
               seed = seed,
               policy_kwargs = policy_kwargs,
               learning_rate = 0.0001355522450968401,
               gamma= 0.995,
               batch_size = 128L,
               buffer_size = 10000L,
               train_freq = 128L,
               gradient_steps = 128L,
               action_noise = action_noise
              )
```

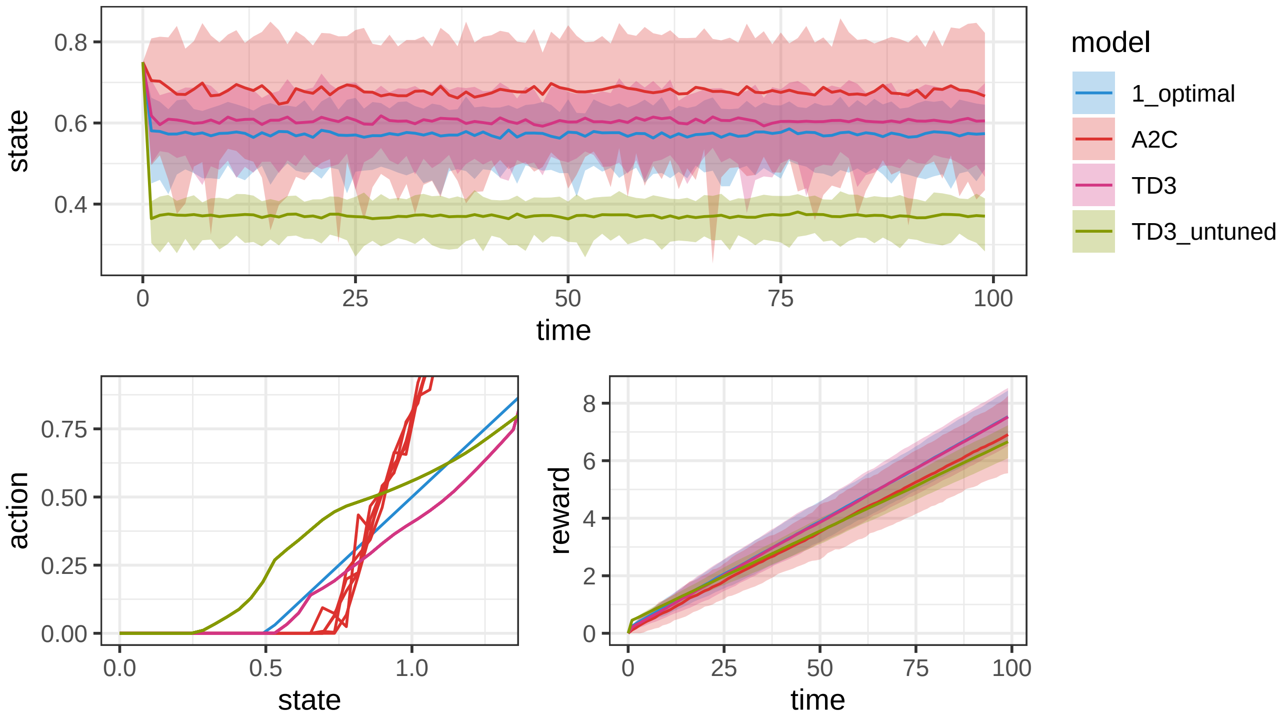
We train this new TD3 agent:

```
if(!file.exists("cache/td3.zip")){

  td3$learn(total_timesteps=300000L)
  td3$save("cache/td3")

}
```

With our trained agent, we can now simulate replicate scenarios or scan across states to determine the implicit policy function:

```
# Simulate management under the trained agent
td3 = sb3$TD3$load("cache/td3")
td3_sims = env$simulate(td3, reps = 100L)
td3_policy = env$policyfn(td3, reps = 5L)
```

```
sims_df <- td3_sims %>% mutate(model = "TD3") %>% bind_rows(sims_df)
policy_df <- td3_policy  %>% mutate(model = "TD3")  %>% bind_rows(policy_df)

plot_sims(sims_df) / ( plot_policy(policy_df) + plot_reward(sims_df))
```



The tuned TD3 algorithm, though trained for an identical 300,000 timesteps as the untined one, settles on a policy much closer to the optimal policy. The tuned TD3 agent tends to underfish on average, but only slightly, and avoids the occassional sharp drops in stock size seen when A2C occassionally sets too high a havest quota. The tuned solution is still not optimal, but it is very close. Moreover, the RL algorithm has been able to learn this nearly-optimal policy with no prior knowledge about the shape or nature of the population growth function used in the environment.

We preserve a copy of each data frames in local `.csv` text files for maximum cross-platform compatibility.

```
write_csv(sims_df, "../manuscript/figs/sims_df.csv")
write_csv(policy_df, "../manuscript/figs/policy_df.csv")
```

3.6 Applying an RL agent to real data

We use historical data from Argentine Hake to illustrate how an RL agent might be applied in practice.

Historical biomass and catch data for Argentine Hake can be found in the R.A. Myers Legacy Stock Assessment Database (**ramlegacy?**). We will use the agent we have just trained using the `gym_fishing` simulator above,

```
# Simulate management under the trained agent
env = gym$make("fishing-v1", r = 1.0379274, K = 1.197693, sigma = 0.1121662)
td3 = sb3$TD3$load("cache/td3")
```

```
hake = read_csv("../data/hake.csv")
x0 = hake %>%
 filter(year == min(year)) %>%
  pull(scaled_biomass)
Tmax = 15
years =1986:2000

td3_state = td3_action = numeric(Tmax)
td3_state[1] = x0
td3_action[1] = NA
```

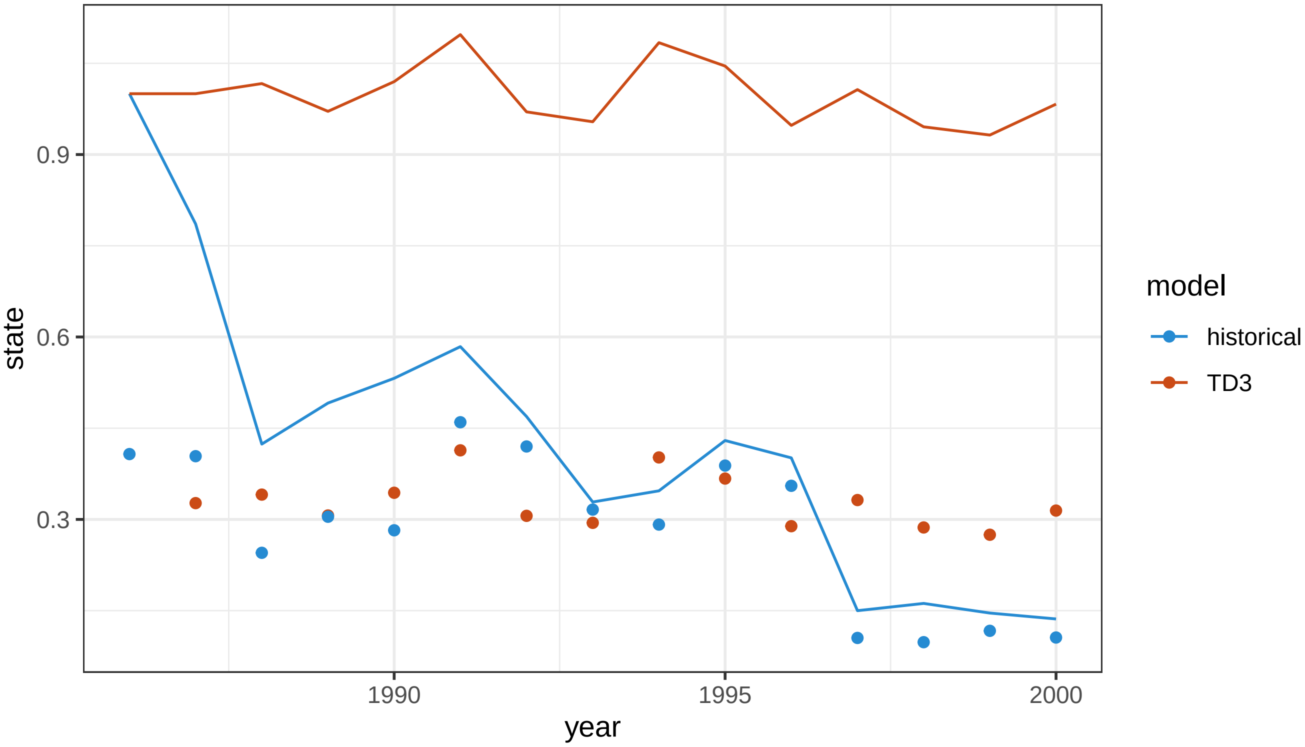
Our hindcast considers the scenario of managing catch quotas using TD3, beginning in 1986. We compare both the anticipated harvest and resulting stock biomass to the historically recorded harvest and biomass.

This shows that quotas set by TD3 would have been lower than historical values throughout the late 80s and early 90s, which saw a sharp decline in estimated biomass of the Argentine Hake. For the same or higher estimated stock sizes from the historical record, we see that TD3 would recommend a lower quota. We also see that under TD3 the stock recovers sufficiently to predict higher quotas by the second half of the 90s than were being set in the now-depleted stock.
Whether the TD3 quotas would have truly been sufficient to permit recovery in stock sizes shown in the simulation clearly depends on the accuracy of the underlying simulation. However, the fact that TD3 would set a lower quota than the historical management given the same or even higher estimate of stock size is independent of the Hake simulation. Combined with the observed historical declines, this provides some evidence that the RL agent could have decreased or avoided the over-harvesting in this case.

```
env$reset()
```

```
## [1] -0.3737961
```

```
## represent the initial state size in the 'rescaled' state space.
state = env$get_state( x0 )

for(i in 2:Tmax){

  # RL-recommended harvest action:
  out = td3$predict(state)
  action = out[[1]]

  # Record state and resulting action
  td3_state[i] = env$get_fish_population(state)
  td3_action[i] = env$get_quota(action)

  # Implement RL-recommended harvest.
  result = env$step(action)
  state = result[[1]]

  #state = env$get_state(hake$scaled_biomass[i+1])
}

df = bind_rows(
  tibble(year = years, state = td3_state, action = td3_action, model = "TD3"),
  tibble(year = years, state = hake$scaled_biomass, action = hake$scaled_catch, model = "historical"))

df %>% ggplot(aes(year, state, col=model)) + geom_line() + geom_point(aes(year, action, col=model))
```



4 Ecological tipping points

Our second example utilizes our `gym_conservation` to provide the necessary environment to simulate an ecosystem approaching a tipping point.

4.1 Tipping point model

The tipping point model is based on the consumer-resource model of (**May1977?**), which creates alternative stable states, which we subject to log-normal environmental noise:

$$\mu_t = X_t + X_t r \left(1 - \frac{X_t}{K}\right) - \frac{a_t X_t^q}{X_t^q + b^q}$$
$$X_{t+1} \sim \text{lognormal}(\mu_t, \sigma)$$

where we take $r = 0.7$, $K = 1.2$, $q = 3$, $b = 0.15$, $a_0 = 0.19$ and $\sigma = 0.2$ Slow change over time in the parameter a_t represents a process of environmental degradation, modeled as a constant increment $a_{t+1} = a_t + \alpha$, where we will take $\alpha = 0.001$. This model supports the dynamics of a fold bifurcation, widely used to model critical transitions in both theory and empirical manipulation in systems from microbes (**Dai2012?**) to lakes (**Carptenter2011?**) to the planet biosphere (**Barnosky2014?**).

```
bifur_df = read_csv("../manuscript/figs/bifur.csv", col_types = "dccd")
bifur_df %>%
  ggplot(aes(parameter, state, lty=equilibrium, group = group)) +
  geom_line()
```

We assume the benefit provided by the ecosystem state is assumed to be directly proportional to the state itself, bx_t.
We further assume that each year the manager has the option to slow or reverse the environmental degradation by taking action A_t, such that under management, the resulting environment in the next time step is given by

$$a_{t+1} = a_t + \alpha - A_t$$

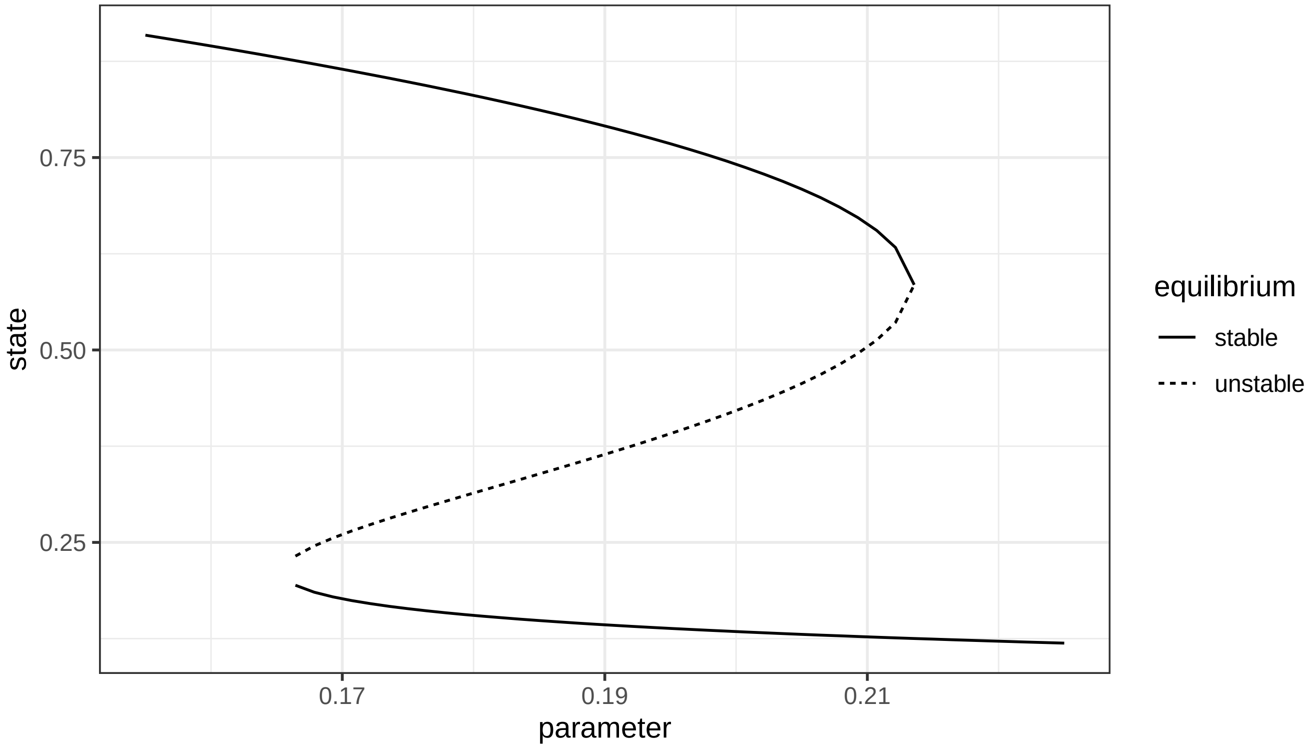


Figure 1: Bifurcation diagram for tipping point scenario. The ecosystem begins in the desirable 'high' state under an evironmental parameter (e.g. global mean temperature, arbitrary units) of 0.19. In the absence of conservation action, the environment worsens (e.g. rising mean temperature) as the parameter increases. This results in only a slow degredation of the stable state, until the parameter crosses the tipping point threshold at about 0.215, where the upper stable branch is anihilated in a fold bifurcation and the system rapidly transitions to lower stable branch, around state of 0.1. Recovery to the upper branch requires a much greater conservation investment, reducing the parameter all the way to 0.165 where the reverse bifurcation will carry it back to the upper stable branch.

We assume the cost associated with that action to be proportional to the square of the action, such that large actions are proportionally more costly than small ones. Consequently, the utility at time t is given by the sum of costs and benefits:

$$U(X_t, A_t) = bX_t - cA_t^2$$

where we will take $b = 1$ and $c = 1$.
Our implementation in `gym_conservation` allows the user to consider alternative parameter choices, alternative models for the ecological dynamics, and alternate types of actions, such as manipulating the ecosystem state directly rather than manipulating the environmental parameter. For some such scenarios the optimal solution is known or can be determined by stochastic dynamic programming, while for others, including the scenario of focus here, the optimal solution is unknown.

For simplicity of illustration, we will train only a TD3-based agent on this scenario.

```
env = gym$make("conservation-v6")
```

```
noise_std = 0.4805935357322933

OU = sb3$common$noise$OrnsteinUhlenbeckActionNoise
action_noise = OU(mean = np$zeros(1L),  sigma = noise_std * np$ones(1L))

model = sb3$TD3('MlpPolicy',
            env,
            verbose = 0,
            seed = 42L,
            "gamma"= 0.995,
            "learning_rate"=  8.315382409902049e-05,
            "batch_size"= 512L,
            "buffer_size"= 10000L,
            "train_freq"= 1000L,
            "gradient_steps"= 1000L,
            "action_noise"= action_noise,
            "policy_kwargs"= list("net_arch"= c(64L,64L)))
```

```
if(!file.exists("cache/td3-conservation.zip")){

  model$learn(total_timesteps=3000000L)
  model$save("cache/td3-conservation")

}
```

```
# Simulate management under the trained agent
# See https://github.com/boettiger-lab/conservation-agents/blob//conservation/TD3-v6.py

model = sb3$TD3$load("cache/td3-conservation")
TD3_sims = env$simulate(model, reps = 100L)
TD3_policy = env$policyfn(model, reps = 10L)
```

In general, the optimal solution depends on the ecological dynamics, the benefit of the ecosystem services and the costs associated with a management response. Because the tipping point problem is non-autonomous, we cannot solve for the optimal policy even given the model and objective (utility) function using Markov Decision Process methods. However, a simple heuristic solution provides a reasonable starting point for comparison.

As discussed in the main text, we have no existing optimal solution to the tipping point problem, and so rely on a common heuristic strategy instead: select a fixed level of conservation investment that is sufficient to counter-balance any further side towards the tipping point, preserving it in it's current state. This is implemented using the `fixed_action` method provided in our `gym_conservation` module, which also implements other heursitic models, including a human agent which requires interactive input to select the action each year.

```
# Simulate under the steady-state solution (given the model)
K = 1.5
alpha = 0.001
opt = gym_conservation$models$fixed_action(env, fixed_action = alpha * 100 * 2 * K )
opt_sims = env$simulate(opt, reps = 100L)
opt_policy = env$policyfn(opt)
```

We gather together the results under the RL agent and steady-state policy as before,

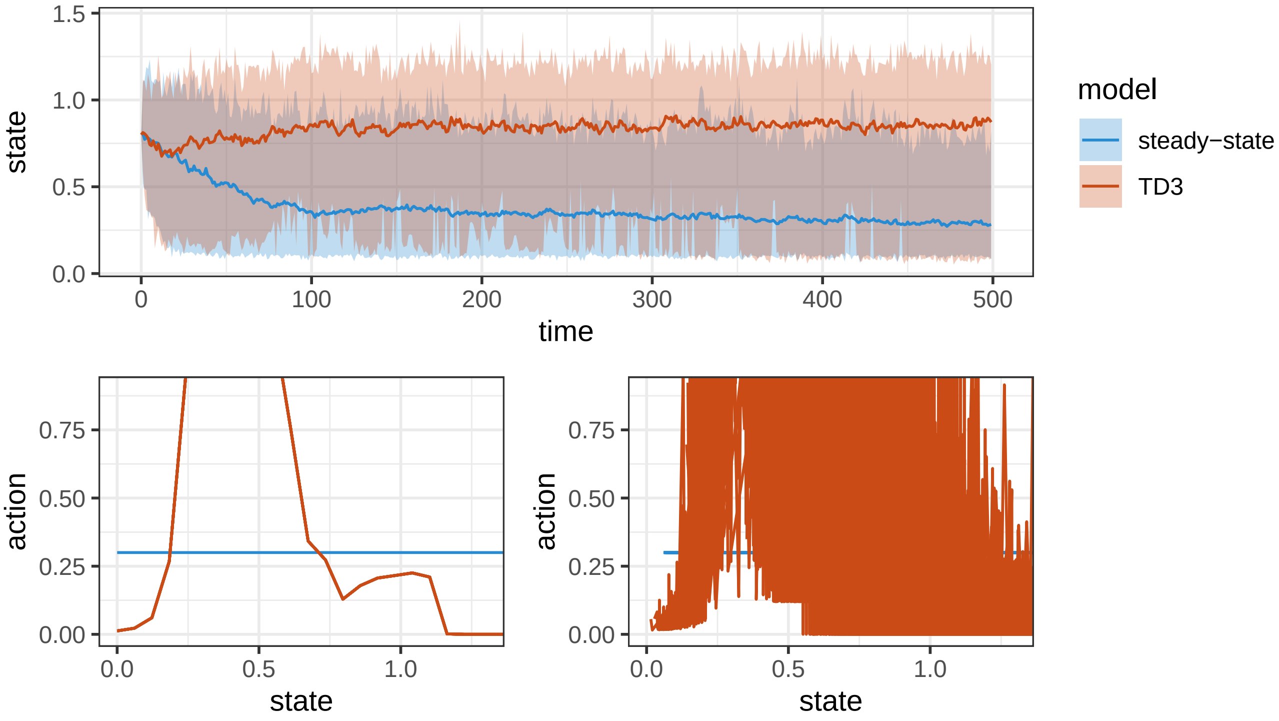
```
sims_df = bind_rows(TD3_sims, opt_sims, .id = "model") %>%
  mutate(model = c("TD3", "steady-state")[as.integer(model)])

policy_df = bind_rows(TD3_policy, opt_policy, .id = "model") %>%
  mutate(model = c("TD3", "steady-state")[as.integer(model)])
```

The resulting three data frames contain the necessary data for each of the subplots in figure 3 of the main text.

```
write_csv(sims_df, "../manuscript/figs/tipping_sims_df.csv")
write_csv(policy_df, "../manuscript/figs/tipping_policy_df.csv")
```

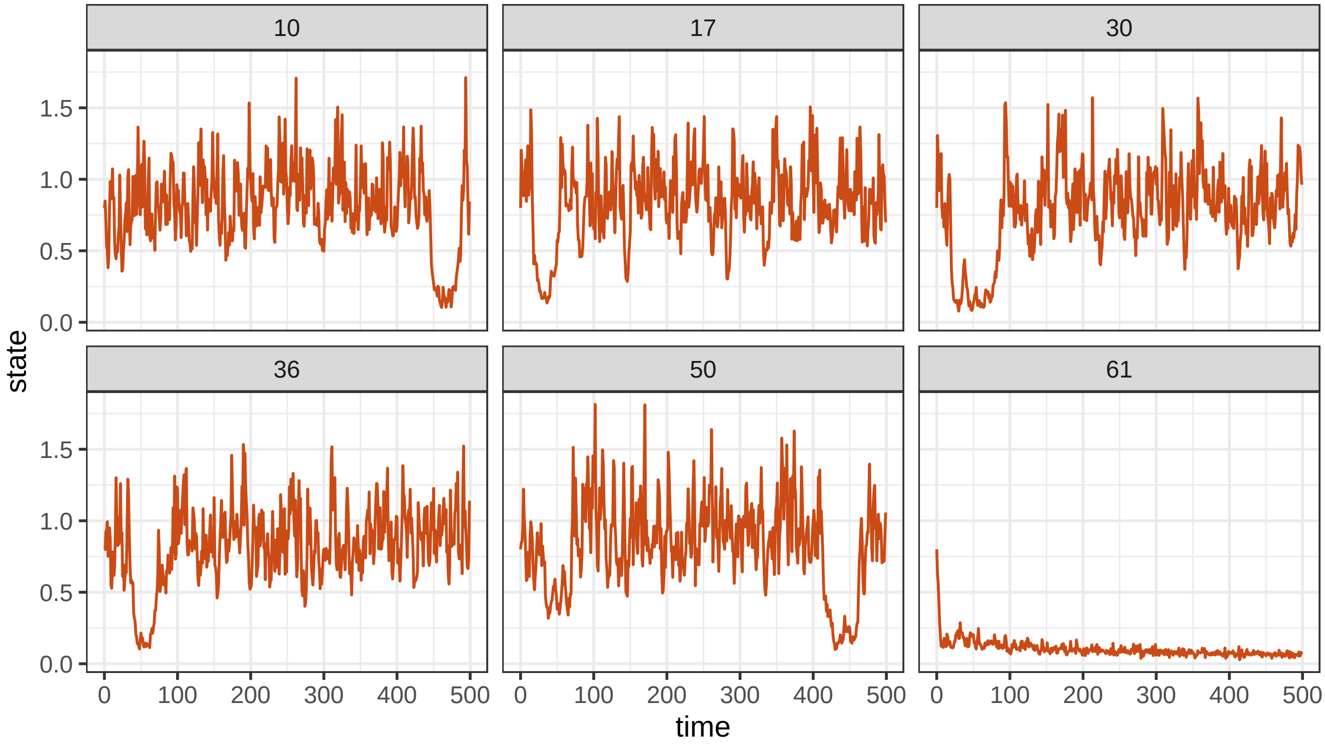
4.1.1 Manuscript Figure 3

```
plot_sims(sims_df) / ( plot_policy(policy_df) + plot_policy(sims_df))
```

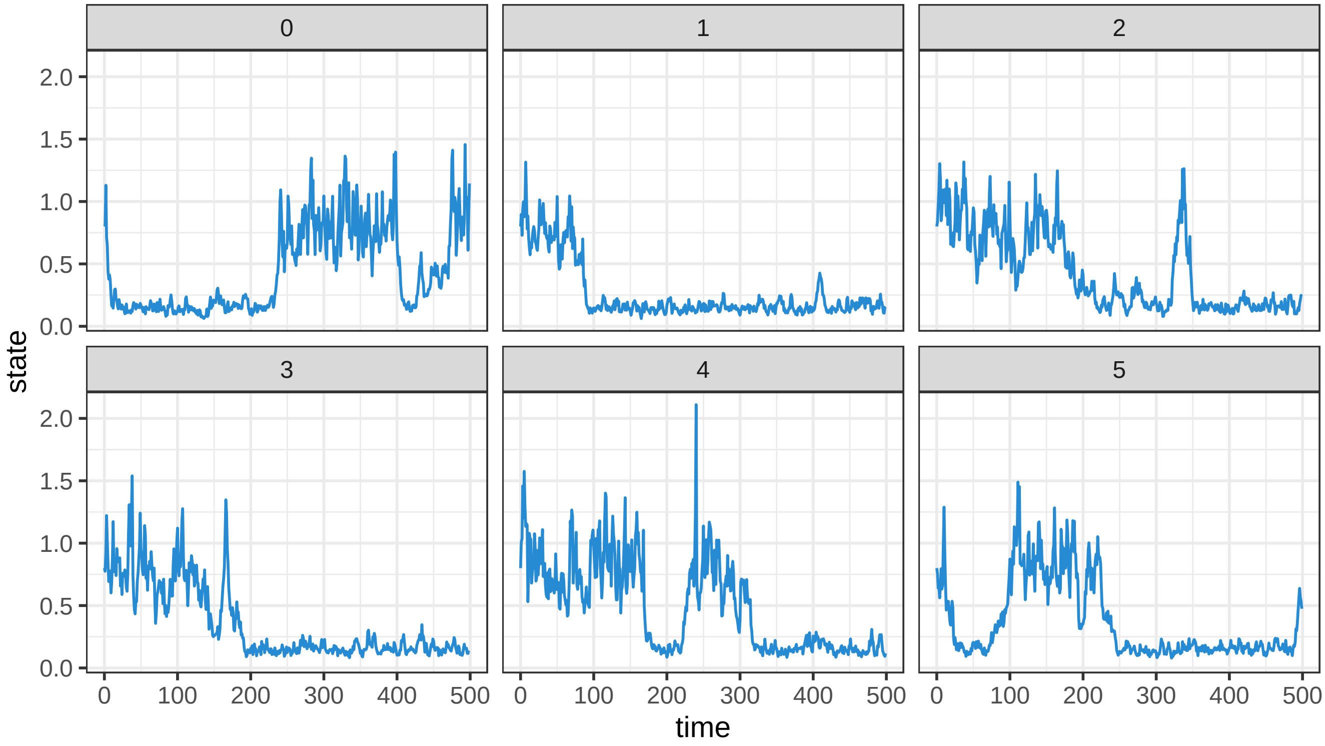


Because it is difficult to get a feel for the dynamics of individual replicate simulations from ensemble statistics, we select a few example trajectories to examine directly. Of the 100 replicate simulations, we pick 4 examples that dip below a state value of 0.2 for over 15 consecutive timesteps, indicating a transition into the lower basin of attraction. Comparing the dynamics under the rule of thumb steady-state strategy to that of the RL-trained agent, it is clear that the RL agent does a better job at both avoiding tipping points and promoting the recovery of those selected trajectories that cross into the lower attractor.

```
# Some individual replicates, for comparison

## First 4 of the TD3 reps falling below .2 for more than 15 steps
is_low = sims_df %>%
  filter(model == "TD3") %>%
  group_by(rep, model) %>%
  summarize(low = sum(state < .2) > 10) %>%
  filter(low) %>% head(6)

## First 4 such cases:
sims_df %>% inner_join(is_low) %>%
  ggplot(aes(time, state,  group=interaction(model, rep))) +
  geom_line(color = pal[3], show.legend = FALSE) + facet_wrap(~rep)
```



```
# Some individual replicates, for comparison
is_low = sims_df %>% filter(model == "steady-state") %>%
  group_by(rep, model) %>% summarize(low = sum(state < .2) > 10) %>%
  filter(low) %>% head(6)
sims_df %>% inner_join(is_low) %>%
  ggplot(aes(time, state, group=interaction(model, rep))) +
  geom_line(color = pal[1], show.legend = FALSE) + facet_wrap(~rep)
```



Fujimoto, Scott, Herke van Hoof, and David Meger. 2018. "Addressing Function Approximation Error in Actor-Critic Methods." *arXiv:1802.09477 [Cs, Stat]*, October. http://arxiv.org/abs/1802.09477.

Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. "Asynchronous Methods for Deep Reinforcement Learning." *arXiv:1602.01783 [Cs]*, June. http://arxiv.org/abs/1602.01783.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518 (7540): 529–33. https://doi.org/10.1038/nature14236.

Raffin, Antonin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. 2019. "Stable Baselines3." *GitHub Repository*. https://github.com/DLR-RM/stable-baselines3; GitHub.