Appendix for: Deep Reinforcement Learning for Conservation Decisions

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

April 21, 2021

Abstract

Keywords blah · blee · bloo · these are optional and can be removed

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.

2 Deep Reinforcement Learning in R

All though 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 straight forward 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. 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.

```
#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 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")
```

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.

```
## 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)
```

3 Finding a known optimal solution using RL

3.1 Sustainable Harvest Quotas

We begin by select 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`.

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

At the heart of each algorithm is a neural network. The type, size and depth of network, choice of activation functions and so forth, are considered "hyper-parameters" of algorithm: we must first make a selection for

each of these hyper-parameters to define our (untrained) agent.
Other hyper-parameters determine other aspects of the training regime: such as how the agent chooses to explore possible actions, the learning rate, or how future rewards are discounted (gamma). Each algorithm may have different hyper-parameters. `stable-baselines3` provides default values for all hyperparameters based on the original papers that introduced the corresponding algorithms, for instance, the A2C algorithm originally described in (**A2C?**). Here, we train an agent using the A2C algorithm using default hyperparameter settings, utilizing a multi-layer perceptron neural network composed of two 64-neuron layers.

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

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 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:

```
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 hyperparameters 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 hyperparameter 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.

```
# 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 a critical 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.

```
## 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, and 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:

```
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:

```
a2c_sims <- env$simulate(a2c, reps = 500L)
```

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 common to consider a policy function: a plot of the action taken at each 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, and the use of a policy function 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 LSTM (**LSTM?**).

```
a2c_policy <- env$policyfn(a2c, reps = 50L)
```

3.2 TD3

We train a second agent based on the TD3 algorithm. This illustrates some of the salient differences between the different algorithms used in reinfocement learning. Bear in mind that the best algorithm for any given environment will depend on the details of the environment, as well as the length of training and the selection of hyper-parameters, including the agent's underlying neural networks which provide the 'brains' of the agent's decision-making ability. This time, we will declare our choices for the hyper-parameters explicitly. Note that some of these hyper-parameters, such as the `learning_rate` and discount rate, `gamma`, are also hyper-parameters of the A2C algorithm, while 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 simply uses standard function optimization approaches to vary the choice of hyper-parameters used in training to determine which maximizes the agent's performance. For simplicity, we show only training with specified hyper-parameters here, example scripts for tuning the algorithm to discover these values can be found in `https://github.com/boettiger-lab/rl-toolkit`, 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,
                n_episodes_rollout = -1,
                gradient_steps = 128L,
                action_noise = action_noise
               )
```

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

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

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

}
```

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 <- sb3$TD3$load("cache/td3")
td3_sims <- env$simulate(td3, reps = 500L)
td3_policy <- env$policyfn(td3, reps = 50L)
```

We compare the performance of these two RL agents trained using the algorithms A2C and TD3 respectively to the known optimal policy for this particular environment.
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 $B = K/2$ 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.

```
# Simulate under the optimal solution (given the model)
opt <- gym_fishing$models$escapement(env)
opt_sims <- env$simulate(opt, reps = 500L)
opt_policy <- env$policyfn(opt)
```

Having performed simulations under each method, we gather the results under each management strategy together into a single data frame for simulations (`sims_df`), the policy function (`policy_df`) and the cumulative reward, used to make the plots in the corresponding panels.
We preserve a copy of each data frames in local `.csv` text files for maximum cross-platform compatibility.

```
sims_df <- bind_rows(td3_sims, a2c_sims, opt_sims, .id = "model") %>%
  mutate(model = c("TD3", "A2C", "optimal")[as.integer(model)])

policy_df <- bind_rows(td3_policy, a2c_policy, opt_policy, .id = "model") %>%
  mutate(model = c("TD3", "A2C", "optimal")[as.integer(model)])

gamma <- 1 #discount
reward_df <- sims_df %>%
  group_by(rep, model) %>%
  mutate(cum_reward = cumsum(reward * gamma^time)) %>%
  group_by(time, model) %>%
  summarise(mean_reward = mean(cum_reward),
```

```
            sd = sd(cum_reward), .groups = "drop")


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

Standard R methods are used to plot the results summarized across the replicates:

```
ymin <- function(x) last(x[(ntile(x, 20)==1)])
ymax <- function(x) last(x[(ntile(x, 20)==19)])

fig_sims <-
sims_df %>%
  group_by(time, model) %>%
  summarise(ymin = ymin(state),
            ymax = ymax(state),
            state = mean(state), .groups = "drop") %>%
  ggplot(aes(time, state, ymin = ymin, ymax = ymax, fill=model)) +
  geom_ribbon(alpha= 0.3) + geom_line(aes(col = model))
```

```
fig_policy <-
  policy_df %>% ggplot(aes(state, action,
                           group=interaction(rep, model),
                           col = model)) +
  geom_line(show.legend = FALSE) +
  coord_cartesian(xlim = c(0, 1.3), ylim=c(0,0.9))
```

```
fig_reward <- reward_df %>%
  ggplot(aes(time, mean_reward)) +
  geom_ribbon(aes(ymin = mean_reward - 2*sd,
                  ymax = mean_reward + 2*sd, fill = model),
              alpha=0.25, show.legend = FALSE) +
  geom_line(aes(col = model), show.legend = FALSE) +
  ylab("reward")
```

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?**).

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

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

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

model = sb3$TD3('MlpPolicy',
            env,
            verbose = 0,
            seed = seed,
            "gamma"= 0.995,
            "learning_rate"=  8.315382409902049e-05,
            "batch_size"= 512L,
            "buffer_size"= 10000L,
            "train_freq"= 1000L,
            "gradient_steps"= 1000L,
            "n_episodes_rollout"= -1L,
            "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)])

gamma <- 1 #discount
reward_df <- sims_df %>%
  group_by(rep, model) %>%
  mutate(cum_reward = cumsum(reward * gamma^time)) %>%
  group_by(time, model) %>%
  summarise(mean_reward = mean(cum_reward),
            sd = sd(cum_reward), .groups = "drop")
```

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, "figs/tipping_sims_df.csv")
write_csv(policy_df, "figs/tipping_policy_df.csv")
write_csv(reward_df, "figs/tipping_reward_df.csv")
```

```
# ensemble statistics
ymin <- function(x) last(x[(ntile(x, 20)==1)])
ymax <- function(x) last(x[(ntile(x, 20)==19)])

fig_sims <-
  sims_df %>%
   group_by(time, model) %>%
   summarise(ymin = ymin(state),
             ymax = ymax(state),
             state = mean(state), .groups = "drop") %>%
  ggplot(aes(time, state, ymin = ymin, ymax = ymax, fill=model)) +
   geom_ribbon(alpha= 0.3) + geom_line(aes(col = model))
```

```
fig_policy <-
  policy_df %>% ggplot(aes(state, action,
                           group=interaction(rep, model),
                           col = model)) +
  geom_line(show.legend = FALSE)
```

```
fig_reward <- reward_df %>%
  ggplot(aes(time, mean_reward)) +
  geom_ribbon(aes(ymin = mean_reward - 2*sd,
```

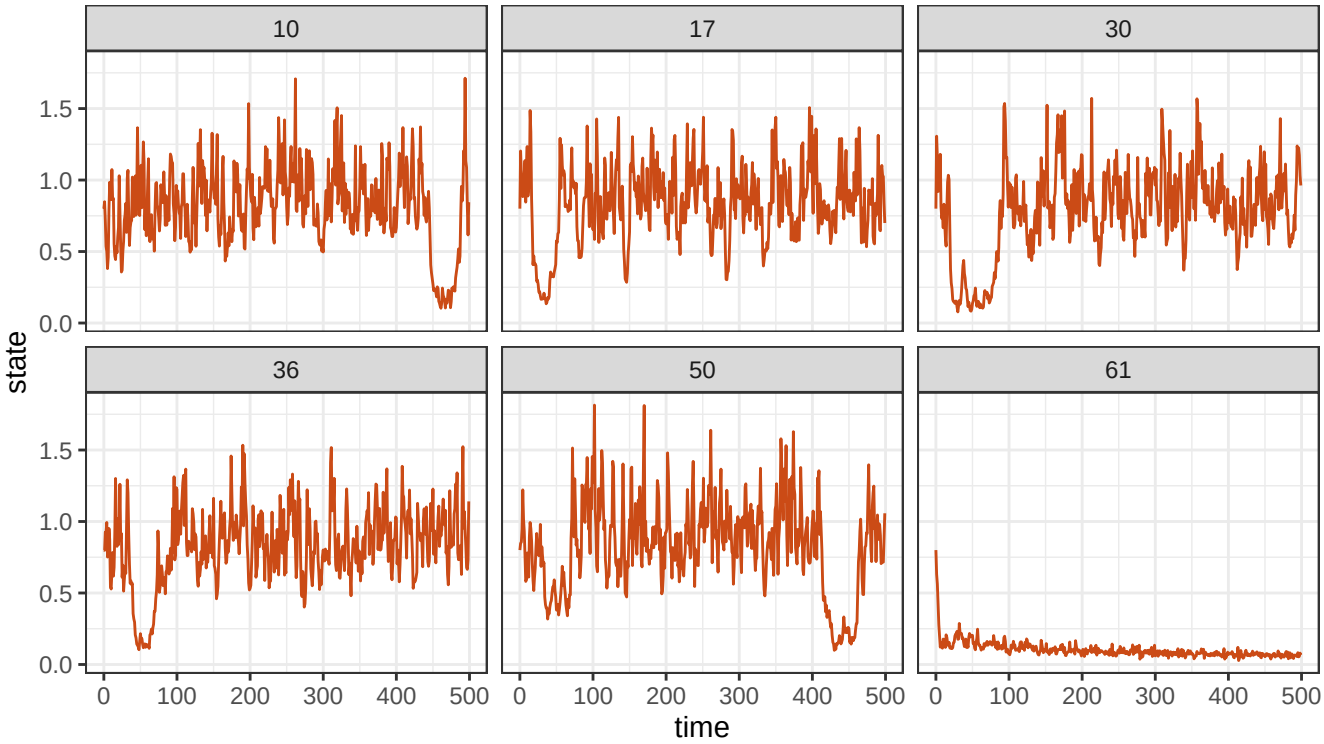
```
                  ymax = mean_reward + 2*sd, fill = model),
              alpha=0.25, show.legend = FALSE) +
  geom_line(aes(col = model), show.legend = FALSE) +
  ylab("reward")
```

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 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.

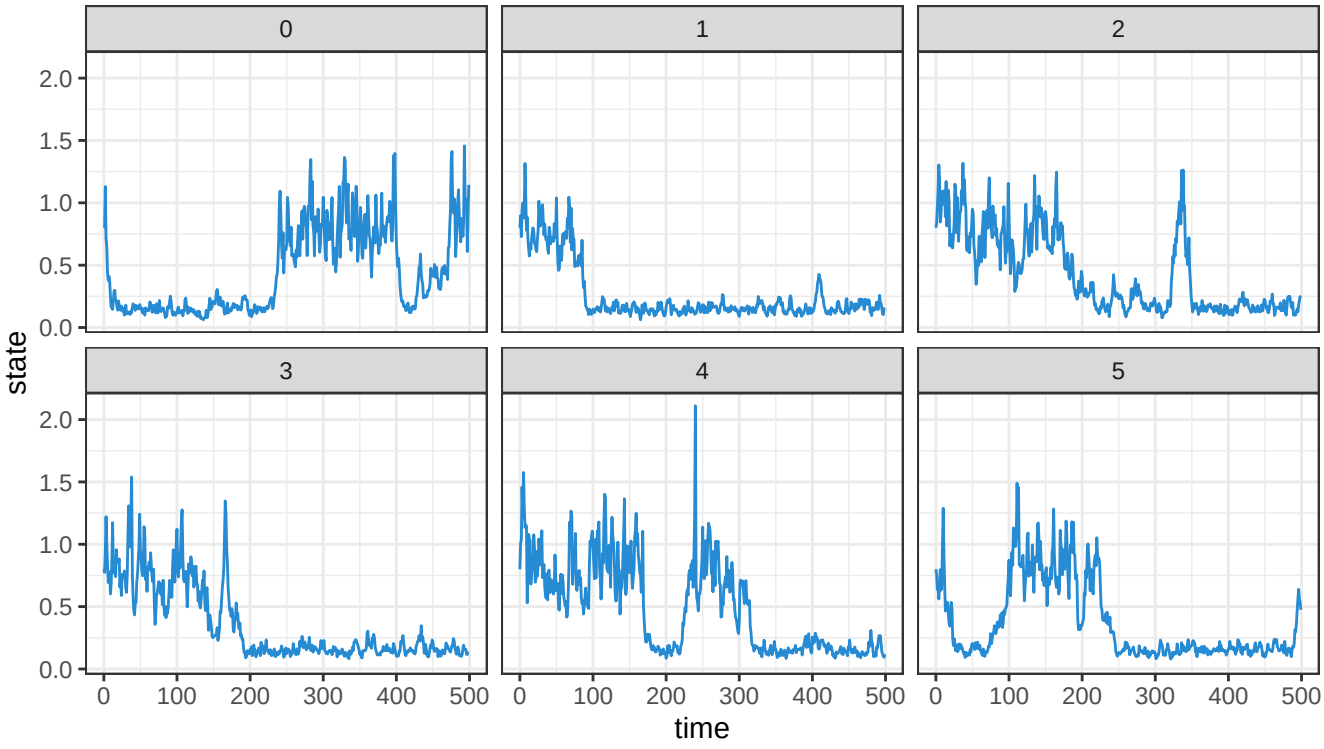
```
# Some individual replicates, for comparison

## 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)
```


0
1
2
3
4
5
state
time
0.0
0.5
1.0
1.5
2.0
0
100
200
300
400
500