Learning to Play Kendama: Project Final Report

Foster Collins FCOLLINS@SEAS.UPENN.EDU
Tighe Costa JOCOS@SEAS.UPENN.EDU
Daniel Orol DANOROL@SEAS.UPENN.EDU

Abstract

Our project focuses on using reinforcement learning (RL) to teach a simulated robotic system simple control algorithms. We learned a control policy to balance an inverted pendulum on a cart and then adapted our algorithms to successfully play a simple ball catching game called Kendama.

1. Overview and Motivation

We use a series of common Reinforcement Learning (RL) techniques and the Open AI Gym (Ope) to learn to swing-up and balance an inverted pendulum on a cart. Controllers are traditionally written by modeling the full dynamics of a system and characterizing the system response. However, this approach becomes impractical for systems with complex dynamics. Model-free RL techniques learn a controllers iteratively through trials rather than explicitly. While the cartpole problem is simple, the techniques used to learn it scale well. Ultimately, we apply the algorithms we used to solve the cartpole problem to learn to play Kendama.

2. Survey of Existing Algorithms

The main categories of RL techniques are value iteration, policy iteration, and actor-critic. Value iteration chooses a fixed policy and uses iterations of the learning event to update the $value(state, action)$. For example, Q-learning chooses the locally optimal action and updates the values each time step based on rewards. Policy gradient fixes the value function and updates the policy, usually via gradient ascent over a weight function to choose the best action at a given state. Actor-critic combines these two, first using a series of iterations to learn the value function and then another set of iterations to learn the policy function. This paper outlines our investigation and findings from implementing these in a simulation environment.

University of Pennsylvania, CIS 419/519 Course Project.

3. Approach

3.1. Simulation

Interfacing with the simulator is done through the `step` method of our CartPoleSwingUp and Kendama classes. The method takes an action and outputs the following state information: cart position, x, cart velocity, $\dot{x}$, pole angle, θ, and pole angular velocity $\dot{\theta}$. For kendama, we add r, ball distance from cart, and $\dot{r}$, radial velocity, to our state space.

The swing-up and balance/catch problem, however, is one of an agitator (swing-up) and a damper (balance/catch), and a linear combination of the current output state cannot act as both. Rather than expand the output state space to try and find one policy to solve the entire task, we chose a piecewise policy: one policy for agitating and one for damping.

3.2. Cartpole and Kendama

We first learn to swing-up and balance an inverted pendulum on a cart. This problem is very similar to the swing-up motion of Kendama, so the two problems are directly related. More importantly, we show that by implementing the RL algorithm properly, multiple problems can be solved with minimal tuning; these algorithms generalize well.

4. Finite Difference Policy Gradient

4.1. Description

This version of policy gradient randomly initializes a set of weight parameters associated with each part of the state. Each weight is disturbed by ϵ, and the gradient of each parameter is then estimated from the reward. Each weight parameter is updated by the learning rate, α times the gradient (Peters, 2010).

5. Implementation

5.0.1. Reward Function

For cartpole, to compute the reward for the swing-up section, we used a function that rewarded smaller θ values along with a goal reward for reaching an threshold on θ and $\dot{\theta}$. For balancing, the reward function was based on having the pole close to vertical. For the swing-up, by also punish-

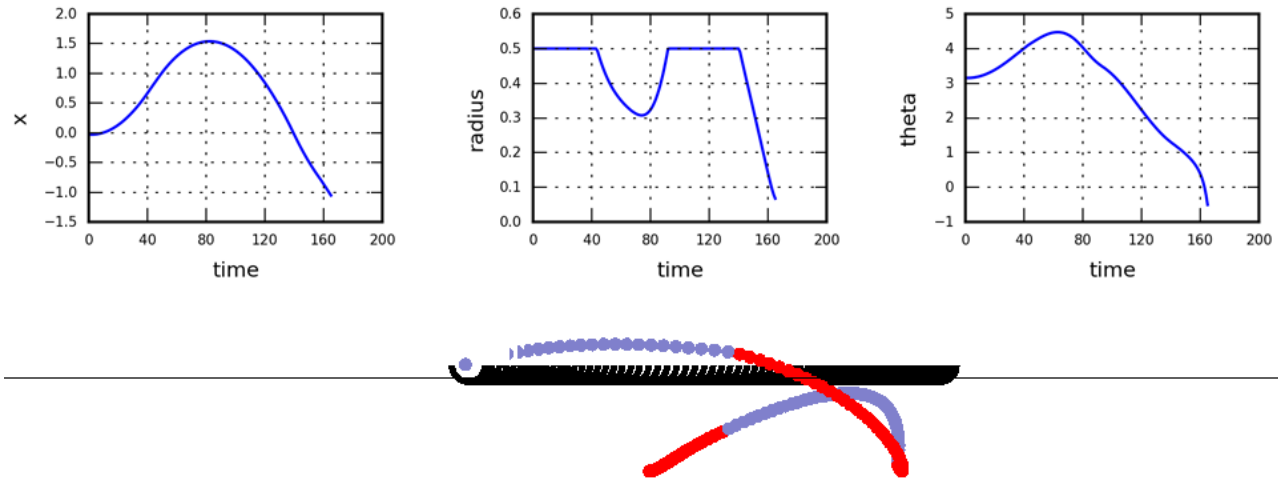


Figure 1. Plots of x, r, and θ over time for a successful Kendama catch (top) and a blurred image of successful catch (bottom).

Algorithm 1 Finite Difference Policy Gradient

Input: params
n = num params
originalParams=params
$\Delta J \Leftarrow [0]_{nX1}$
for $i = 1$ **to** n **do**
  params[i] = originalParams[i] + ϵ
  J_i = Total Reward of New Params
  params[i] = originalParams[i] - ϵ
  J_{ref} = Total Reward of New Params
  $\Delta J[i] = J_i - J_{ref}$
end for
$\Delta\Theta = I * \epsilon$
return: gradient = $(\Delta\Theta^T \Delta\Theta)^{-1} \Delta\Theta^T \Delta J$

ing x and $\dot{x}$ of the cart, we could generate a slow, gradual swing-up that was much more stable. This illustrates the power of the reward function and its tuning.

5.0.2. Learning Parameters

The gradient was computed based on the reward over 200 episodes with the disturbed parameters from Algorithm 1. We chose a constant ϵ = 2 and α = 0.005. While this ϵ was higher than we expected, it must be this large to avoid local maxima. With a small step size, valleys cannot be avoided.

5.0.3. Adapting to Kendama

To adapt the algorithm to Kendama, we adjusted the reward function and the parameter vector size (adding weights for r and $\dot{r}$). The swing-up reward function was similar, but large r was punished instead of $\dot{\theta}$ in order to avoid missing the cup. The catching reward function was based on having a small r at low θ, plus a goal reward for success.

5.1. Results

This algorithm was very successful. For cartpole, using 1200 learning iterations for swing-up and 300 iterations for balancing, we observed success in 86/100 trials. For Kendama, using 500 learning iterations for both swing-up and catching, we observed success in 89/100 trials.

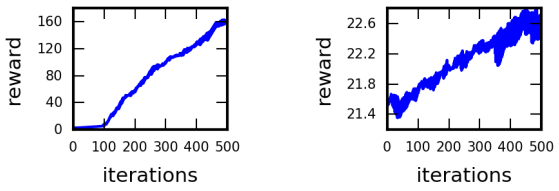


Figure 2. Learning curves for (a) swing-up and (b) catch.

5.2. Discussion

The gradient ascent of this algorithm can be clearly seen in the learning curves. Although they are noisy, the reward function steadily increases over the iterations and then starts to plateau at the end. This plateau effect indicates that a local maximum has been reached.

The primary strength of this algorithm is that it moves from an initial point towards a local maximum. Therefore, it is guaranteed to converge to a local maximum. Although we cannot show that we converged to the global maximum, by adjusting our ϵ, we were able to improve the success rate from about 50% to 86%. This high success rate demonstrates that finite difference policy gradient methods are robust and repeatable.

The biggest drawback of this algorithm is the sensitivity of the tuning parameters. The importance of tuning ϵ appropriately can be seen by computing the success rate for

$\epsilon = 1.5, 2, 2.5$ with a fixed α. These values were 19%, 86%, and 77% respectively, showing the steep drop-off around the optimal learning parameters.

With regards to Kendama, we left our ϵ and α the same for both cartpole and Kendama (swing-up, balance, and catch). The reward function was adjusted as explained above, and this immediately allowed a controller to be learned.

6. Q-Learning

6.1. Description

Q-learning is a value iteration algorithm that finds an optimal action-selection policy for a finite Markov Decision Process (MDP). It works by learning an action-value (Q-value) function that provides the expected reward of taking an action given a certain state, and, once that Q-value function has been built, takes the locally optimal action.

Because Q-learning learns a discrete policy, the state space for both the cartpole and Kendama problems must be discretized. The challenge of discretizing a continuous space is to build the smallest state/action space that can provide a solution. If the state/action space has too few bins, then the policy will not be able to differentiate between similar states that require different actions. The larger the state/action space (i.e. more bins), the more iterations Q-learning requires to learn the optimal policy.

6.2. Implementation

6.2.1. Discretization

We combined linear and logarithmic discretization to balance policy performance and convergence time for the cartpole simulation. x and $\dot{x}$ are sorted into fixed-size bins (linear discretization) while θ and $\dot{\theta}$ are sorted into bins of logarithmically increasing size. This method is motivated by the fine resolution in θ and $\dot{\theta}$ required for pole balancing and relatively coarse resolution required for swing-up and results in a 4,603-bin state/action space.

While likely neither swing-up nor catch in Kendama would require fine radial resolution, even the coarsest linear discretization of r and $\dot{r}$ (two bins each) would increase the state/action space by four-fold. The vulnerability of Q-learning to problems with many features is apparent; we chose not to pursue a Q-learning solution to Kendama, as sufficient exploration of its state/action space is not efficient compared to our other explored methods.

6.2.2. Reward Function

The reward function for cartpole swing-up is comprised of living cost, objective failure cost, and objective success reward. The living cost is small and inversely proportional to θ and x to encourage early agitation centered around $x = 0$. Failure cost is a large constant to deter out-of-bounds movements, and the success reward is large and inversely proportional to $\dot{\theta}$ and cart velocity at the transition angle to encourage gentle swing-ups. The reward function for cartpole balancing does not have a success reward.

6.2.3. Learning Parameters

Viable convergence to a swing-up solution given the size of the state/action space for the cartpole simulation required high initial exploration and learning rates to get a large preliminary sampling of utility estimates. Swing up uses initially high learning and exploration rates, and which are stepped down after 500 and 1000 iterations, respectively. Q-learning converged to a solution for balancing with constant, relatively low learning and exploration rates.

6.3. Results

Q-learning converged to a solution within 2000 and 1000 iterations for swing-up and balance respectively. 20% of the learned policies did not drop the pole. Figure 3 shows the learning curves for cartpole swing-up and balance.

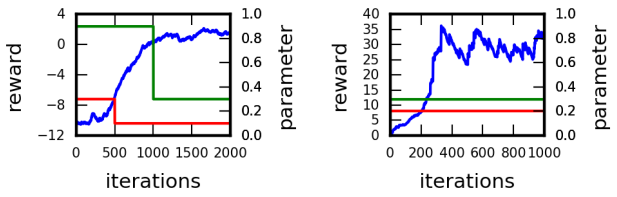


Figure 3. Learning curves with learning rate (red) and exploration rate (green) for (a) swing-up and (b) balance.

6.4. Discussion

Q-learning is not well suited for the cartpole problem because it requires discretization of the state/action space. Finer resolution yields a better controller, but the size of the state/action space prevents Q-learning from being a viable solution.

7. Random

7.1. Description

Random search randomly populates the weight vector associated with each element of the state. The set of weights with the highest reward over a specific number of iterations is returned. (Frans, 2016)

7.2. Implementation

For cartpole, randomly searching across the policy space by regenerating both the swing-up and balance policies simultaneously did not converge in 100,000 iterations. We found

that training each policy separately with random search, however, typically converged within 20,000 iterations.

Swing-up is trained first, followed by balance using the learned swing-up. Each policy is randomly initialized with values in $[-1, 1]$ and simulated for five episodes (to reduce variance). A total reward is determined from those episodes. This procedure is repeated n times and the policy with the highest total reward is returned.

For Kendama, we only needed to randomly iterate over 1000 trials each for swing-up and catching.

7.3. Discussion

Random search learned these problems successfully but not robustly. The motions learned resulted in relatively unstable balancing; more importantly, it was inconsistent and depended heavily on the iteration. Random search worked more rapidly for Kendama because the swing-up section is more important than the catch section in that problem.

8. Policy Iteration

8.1. Description

We experimented with different methods involving a continuous Gaussian policy. Unfortunately, we were unsuccessful in getting these types of policies to work on the Kendama problem. In this section we will give a brief overview of the methods tried, and some of the issues encountered. We implemented a well know policy iteration algorithm called REINFORCE. It uses a fixed policy to complete an episode. It then uses information about the total reward from the episode to update the policy.

8.2. Implementation

The stochastic policy $\Pi(state)$ was parametrized by a Gaussian random variable whose mean is determined by a linear combination of the state vector of the system and whose variance is $\sigma \tilde{} N(f(state), \sigma(episode))$. To promote exploration, σ was high in early episodes and then decreased so that it would converge on the optimal policy.

8.3. Results

To test this implementation on a simple case, we tested the controller on the cart pole balancing problem. For this problem, this algorithm worked out of the box. After tuning, we saw convergence on a stable solution for balancing after 4000 episodes. Because this algorithm requires knowledge of the final reward of an episode before starting to update the policy, it is very slow to converge. The figure below shows the typical convergence of this algorithm. (Sutton & Barto, 2017)

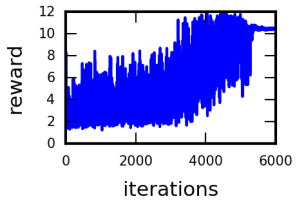


Figure 4. This shows the learning rate of REINFORCE on the cart pole balancing

8.4. Discussion

Converting this over to the Kendama problem created some issues. Because of the non-linear nature of the policy, we again used a piece-wise policy based on θ. In addition, the current policy update step is not suited for a sequence task like the swing up or the catching. Instead of looking at whether each action was an improvement over the last, the algorithm needs to look ahead to see if the actions it is taking now will lead to high rewards later. This is achieved with the discount parameter. After the sequence of actions in an episode is determined, the final reward is propagated back through with a discount depending on how far you want to look ahead. This method is very prone to local minimums in learned policy, and we were never able to force it out of these local minimums.

9. Actor-Critic

9.1. Description

In hopes of avoiding local minima, we investigated the Actor-Critic method. The Critic maintains a parametrized approximation of the action-value function Q(a,s) while executing some parametrized policy. The Critic uses estimates of Q to update the Actor at each time step using the Temporal Difference (TD) error.

$$TDerror = \delta = R_t + \gamma Q_w(S_{t+1}, A_{t+1}) - Q_w(S_t, A_t)$$

Where $Q_w(S_t, A_t)$ is the dot product of the $f(s, a)^T w$ This allows for online learning where the policy is updated at each time step instead of at the end of the episode.

9.2. Implementation

The major implementation difference between Actor-Critic and REINFORCE is that instead of using the actual return from the full episode to calculate the value of each state, the value $Q(s, a)$ and policy are updated at each step. The value function is updated with a gradient decent based on the TD error (Silver, 2015),

$$w_{new} \leftarrow w + \alpha \delta_t S_t$$

and the policy is updated as follows.

$$\Theta_{new} \leftarrow \Theta + \alpha \frac{(A_t - f(S_t))}{\sigma} Q(s, a) S_t$$

9.3. Results

Again we ran into problems with the basic linear combinations of features on the cart-pole problem. We had good performance if we restricted the initialized position to very small areas (+/- 0.1 radians) where it could consistently learn a solution to the pole balancing with 100 exploration episodes and 100 actor training episodes! We performed feature expansion on the state space to form other features from the basic states. We tried the following expansion:

$$State = (1, x, x^2, x\dot{x}, x\theta, x\dot{\theta}, \dot{x}, \dot{x}^2, ...\dot{\theta}^2)$$

9.4. Discussion

While feature expansion improved the TD error values we saw training, it struggled to capture the discrete rewards and penalties in the value function. In order to better approximate this we would need a more powerful function approximation like a Neural Network or a Gaussian function approximator. We experienced large spikes in the TD error, which continued to grow the weight vector even when the algorithm had converged. We had problems with overflow if too many episodes were run. With the problems discovered with the cart pole, we did not attempt to move the Actor-Critic framework over to the Kendama Problem. While this algorithm originally seemed like the most promising because it can learn and optimal value and action simultaneously, it ultimately was unsuccessful, largely due to the nature of the problem we are solving.

10. Conclusion

The continuous nature of our state space lends itself to policy gradient methods over value iteration methods.

For methods such as Q-learning, researchers compensate for this difficulty by implementing approximate Q-learning. As we learned from our results, the actor-critic method is very sensitive to its evaluation of the value function. Similarly, our Q-learning implementation learned the state space, but it required more than 4500 bins. This quickly becomes intractable as the state space becomes even larger (a 12-degree of freedom robot for example).

The key point here is the difference between random exploration of states and random exploration from a given state. There are very few random actions that will result in an object swinging up on a moving cart. However, there are multiple ways to swing-up from a state. In other words, once we determine that the policy gradient is learning a linear combination of features, it can take coherent sequential actions that will result in a successful swing-up; the value iteration cannot frequently achieve successful swing-ups through completely random, unrelated actions, however, so it does not receive the required rewards to properly learn.

Comparing the random search policy iteration to the finite difference policy gradient, the latter learned to balance the pole closer to the center of the screen and remained more stable over time. Additionally, by using the gradual swing-up reward function, the finite difference function could be tuned to remain even more stable. Policy gradient intelligently updates the weights to repeatedly improve on the current policy. Additionally, in a space with significantly more weight parameters, the random search performance would start to decay because every parameter would need to be randomly chosen simultaneously.

Although it appears that Kendama is a slightly easier problem to succeed at because it does not require a balancing section, the overall problem of Kendama is still more complex because it has two sets of dynamics that describe it (tension/not tension). The policy iteration methods applied to Kendama were easily able to solve both problems without knowing about the difference in dynamics; we have demonstrated the generalizability of RL algorithms to analogous problems.

Acknowledgments

We would like to acknowledge Dr. Eric Eaton for his advice on learning approaches for our project.

11. Citations and References

References

Frans, Kevin. Simple reinforcement learning methods to learn cartpole. 2016. URL http://kvfrans.com/simple-algoritms-for-solving-cartpole/.

OpenAI Gym. OpenAI. URL https://gym.openai.com/.

Peters, Jan. Policy gradient methods. *Scholarpedia*, 2010. URL http://www.scholarpedia.org/article/Policy_gradient_methods.

Silver, David. Policy gradient. 2015. URL http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching_files/pg.pdf.

Sutton, Richard S. and Barto, Andrew G. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Massachusetts, 2017.

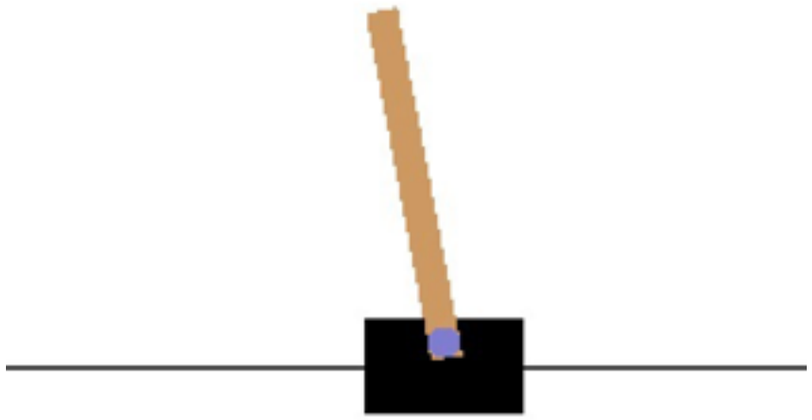
Learning Kendama

Problem Statement:

We used a series of algorithms to learn to swing-up and balance of an inverted pendulum on a cart using the OpenAI Gym, and then extended those algorithms to learn on the more complex game of Kendama (a cup-in-ball game).

Motivation:

Traditional controllers require models of system dynamics and expertise in control tuning. RL can be used to learn a model-free controller of a simple system and then easily expand it to a complex system.



Modified OpenAI simulator used to test swing-up and balance of inverted pendulum.

Kendama, a cup-in-ball game

Explored Algorithms:

Random Search: Choose random parameters that give the highest reward.

Policy Gradient: Given the value of a state, pick the best policy. Update policy by computing its gradient.

Actor-Critic: First, learn a value function given a series of states. Use that to learn an optimal policy.

Value Iteration (Q-learning): Given a policy, update the value of each state to maximize the reward over time.

Results:

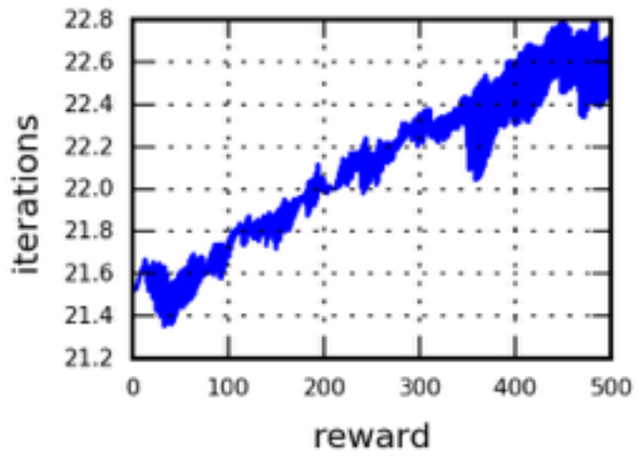
We found our **policy gradient** algorithm yielded both the fastest learning and the most robust policy.

Success Rates:

86/100 trials cartpole

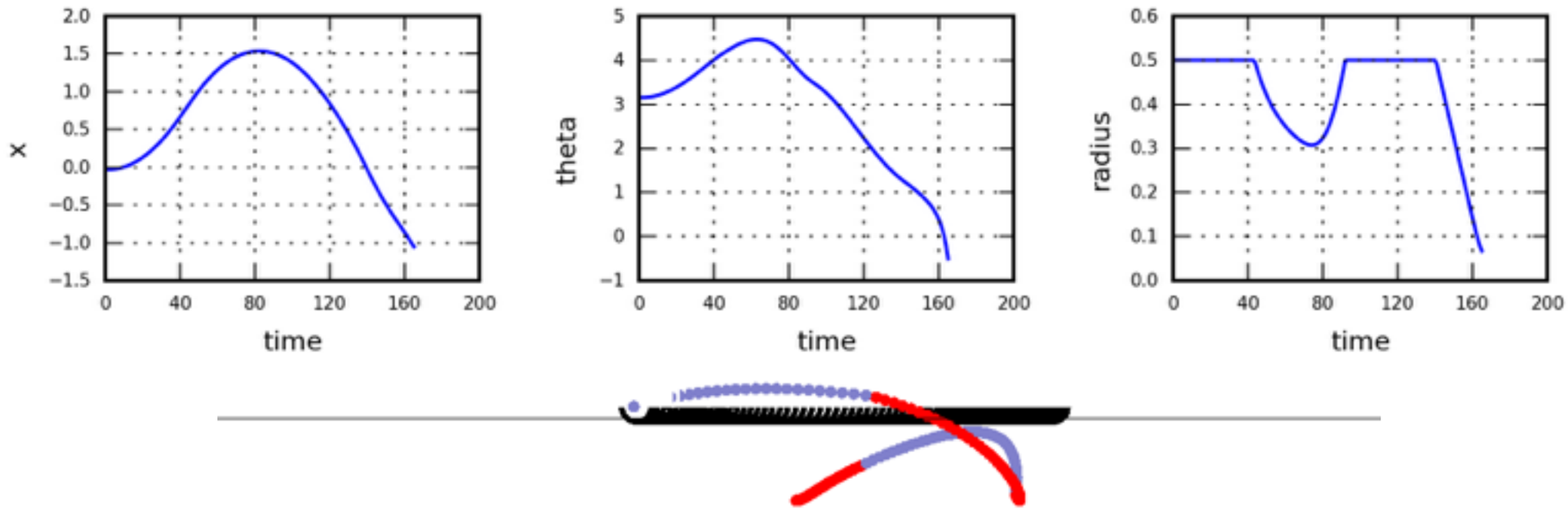
89/100 trials Kendama

Policy gradient learning curve for Kendama swing-up.



Policy gradient learning curve for Kendama catch.

Simulated Policy:



Plots of x, r, and θ over time for a successful Kendama catch (top) and a blurred image of that catch (bottom).