Switched Soft Actor Critic For The Acrobot Swing-up And Balance Task

Sean Gillen, Marco Molnar, and Katie Byl

***Abstract*— In this work we present switched soft actor critic (SSAC), a novel extension of SAC designed to control an acrobot system. Our method allows us to combine traditional controllers with learned policies. This combination allows us to leverage both our own domain knowledge and some of the advantages of model free reinforcement learning. Unlike prior work, our method simultaneously learns neural network policies and the transitions between these policies. We demonstrate that our technique outperforms other state of the art reinforcement learning algorithms in this setting.**

I. INTRODUCTION

Advances in machine learning have allowed researchers to leverage the massive amount of compute available today in order to better control robotic systems. The result is that modern model-free reinforcement learning has been used to solve some very difficult problems. Recent examples include controlling a 47 DOF humanoid to navigate a variety of obstacles [1], dexterously manipulating objects with a 24 DOF robotic hand [2], and allowing a physical quadruped robot to run [3], and recover from falls [4].

Despite this, these algorithms can struggle on certain low dimensional problems from the nonlinear control literature. Namely the acrobot [5] and the classic cart pole pendulum. These are both under-actuated mechanical systems that have unstable fixed points in their unforced dynamics (see section II-B for specifics). Typically, the goal is to bring the system to this fixed point and keep it there. In this paper we focus on the acrobot as we found less examples of model free reinforcement learning performing well on this task.

It is not uncommon to see some variation of these systems tackled in various reinforcement benchmarks, but we have found these problems have usually been modified in a way to make them much easier. For example the very popular OpenAI Gym benchmarks [6] includes an acrobot task. But the objective is only to get the system in the rough area of the unstable fixed point, and the dynamics are integrated with a fixed time-step of .2 seconds, which makes the problem much easier and unrepresentative of a physical system. We have modified this environment to include a higher order integrator with a smaller timestep, and changed the objective such that the system needs to be held at the unstable fixed point. We have found that almost universally, modern model free reinforcement learning algorithms fail to solve the task. Indeed we have found most bench marking suites exclude the full acrobot problem ([7], [8], [6]), and we suspect this is at least partially because most algorithms perform poorly on this problem. Tellingly, the Deep Mind control suite ([9]) did include the full acrobot problem, and all but one algorithm that that they tested (the exception being [10] learned nothing, the average return after training was the same as before training.

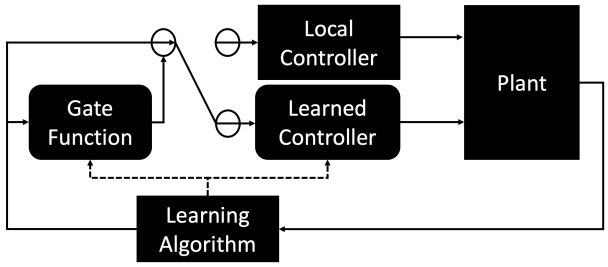

Fig. 1: System diagram for our technique. Rounded boxes represent learned neural networks, squared boxes represent static functions

Despite this, there are many traditional model based solutions ([5], [11]) that can solve this problem well. If this is a solved problem, why should we care about solving it with new tools? We argue that the tools used to modify model free methods to solve the acrobot problem better can be used to expand the class of problems that reinforcement can solve well.

Why does reinforcement learning struggle so much? Even with generous torque limits, the region of state space that can be brought to the unstable fixed point is very small. Furthermore reaching this region by taking random actions (which is exactly what the untrained agent will do) is fairly rare, even with carefully chosen values for the update frequency we found that random actions will reach the basin of attraction for a well designed LQR in about 1% of trials. This is not so bad by itself, but just reaching this special region of state space is not enough, the agent must already know how to balance the acrobot, otherwise it will not get a strong reward signal for being in this region. The result is that the controller we need is a needle in a haystack as far as the reinforcement learning is concerned.

Our solution to this is to give the algorithm some help

*This work was funded in part by NSF NRI award 1526424.

Sean Gillen and Katie Byl are with the Electrical and Computer Engineering Department at the University of California, Santa Barbara CA 93106 sgillen@ucsb.edu, katiebyl@ucsb.edu. Marco Molnar is with TU Berlin marco.molnar@posteo.de.

**Source code for this paper can be found here: https://github.com/sgillen/ssac

in the form of a balancing controller, this is comparatively easy to design, and can be done with a linear controller. Our contribution is a novel way to combine this balancing controller with an algorithm that is learning the swing up behavior. We simultaneously learn the swingup controller, and a function that switches between the two controllers.

In this paper we apply this technique to the acrobot, but we believe it can be generalized in a straightforward way to other problems. Our technique allows engineers to use their insights and domain knowledge in their design, as well as leverage well understood controllers from the existing controls literature.

A. RELATED WORK

Work done by Randolov et. al. [12] is closely related to our own. In that work they construct a local controller, an LQR, and combine it with a learned controller to swing-up and balance an inverted double pendulum (similar to the acrobot we study but with actuators at both joints). The primary differences between our work and theirs is that they hard code the transition between their two controllers. In contrast we learn our transition function online and in parallel with our swing-up controller.

Work done by Yoshimoto et. al. [13], like ours, learns the transition function between controllers in order to swing-up and balance an acrobot. However, unlike our work they limit the controllers they switch between to pre-computed linear functions. In contrast our work simultaneously learns a nonlinear swing-up controller and the transition between a learned and pre-computed balance controller.

Wikledt et. al [14] too swing up and balance an acrobot using a combined neural network and LQR. However they only learn to swingup from a single initial condition, whereas our method learns to solve the task from any initial position.

Doya [15] also learns many controllers using reinforcement learning, and adaptively switches between them. However unlike our work, the switching function is not learned using reinforcement learning, but is instead selected according to which of the controllers currently makes best prediction of the state at the current point in state space. We believe our model free updates will avoid the model bias that can be associated with such approaches. Furthermore our work allows for combining learned controllers with hand designed controllers, such as the LQR.

II. BACKGROUND

A. Nomenclature

We formulate our problem as a Markov decision process, $M = (S, A, R)$. At each time step t, Our agent receives the current state $s_t \in S$ and chooses an action $a_t \in A$. It then receives a reward according to the reward function $r_t = R(s_t, a_t, s_{t+1})$. The goal is to find a policy $\pi : S \rightarrow p(A = a|s)$ that satisfies:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})\right] \tag{1}$$

B. Acrobot

The acrobot is described in Figure 2. It is a double inverted pendulum with a motor only at the elbow. We use the parameters from Sponge [5]:

TABLE I: Mass and inertial parameters used in simulation

Parameter	Value	Units
m_1, m_2	1	Kg
l_1, l_2	1	m
l_{c1}, l_{c2}	.5	m
I_1	.2	Kg*m^2
I_2	1.0	Kg*m^2

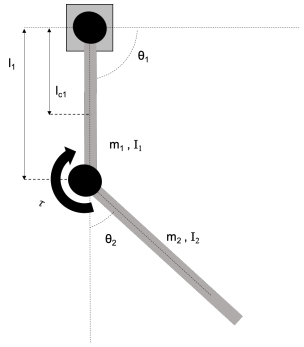

Fig. 2: Diagram for the acrobot system, l_{c*} denote the center of mass for each link. l_2, l_{c2} follow as l_1, l_{c1}

The state of this system is $s = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]$. The action $a_t = \tau$, is the torque at the elbow joint. The goal we wish to achieve is to take this system from any random initial state, to the upright state $gs = [\pi/2, 0, 0, 0]$, which we will refer to as the goal state. To achieve this goal, we seek the maximum of the following reward function:

$$r_t = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) \tag{2}$$

This was motivated by the popular Acrobot-v1 environment [16], We found empirically that for our algorithm this reward signal led to the same solutions as the more obvious $\|s_t - gs\|$, $\|s_t - gs\|^2$ etc. However we found that some of the other algorithms we compared to perform better with the sinusoidal reward function.

We implement the system in python (all source code is provided, see footnote on page one), the dynamics are implemented using Euler integration with a time-step of .01 seconds, and the control is updated every .2 seconds. We experimented with smaller timesteps and higher order integrators, generally we found these made the balancing task easier, but made the wall clock time for the learning much slower.

C. Soft Actor Critic

Soft actor critic (SAC) is a recent off policy deep reinforcement learning algorithm shown to do well on control tasks with continuous actions spaces [17]. To aid in exploration, rather than directly optimize the discounted sum of

future rewards, SAC attempt to find a policy that optimizes a surrogate objective:

$$J^{soft} = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \Big(R_t + \alpha H(\pi(\cdot|s_t)) \Big)\right] \tag{3}$$

Where H is the entropy of the policy.

SAC introduces several neural networks for the training. We define a soft value function $V_\phi(s_t)$, a neural network defined by weights ϕ, which approximate J^{soft} given the current state. Next we define two soft Q functions $Q_{\rho_1}(s_t, a_t)$ and $Q_{\rho_2}(s_t, a_t)$ which approximate J^{soft} given both the current state and the current action. Using two Q networks is a trick that aids the training by avoiding overestimating the Q function. We must also define a target soft value function $V_{\bar{\phi}}(s_t)$, which follows the value function via polyak averaging:

$$V_{\bar{\phi}^+}(s_t) = c_{py} V_{\bar{\phi}}(s_t) + (1 - c_{py}) V_\phi \tag{4}$$

With c_{py} a fixed hyper parameter. We need this because the target value function shows up in the loss for the Q functions, using the true value function in the loss causes instability in the learning. We also define Π_θ, a neural network that outputs $\mu_\theta(s_t)$ and $\log(\sigma_\theta(s_t))$ which define the probability distribution of our policy π_θ. The action is given by:

$$a_t = \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t)\epsilon_t) \tag{5}$$

where ϵ_t is drawn from $N(0, 1)$.

SAC also make use of a replay buffer D which stores the tuple (s_t, a_t, r_t) after policy rollouts. When it is time to update we sample randomly from this buffer, and use those samples to compute our losses and update our weights.

With this we can define the losses for each of these networks (see [17] if you would like to understand where these come from.)

The loss for our two Q functions is:

$$L^Q = \mathbb{E}_{s_t, a_t \sim D}\left[\frac{1}{2}\Big(Q_\rho(s_t, a_t) - \hat{Q}(s_t, a_t)\Big)^2\right] \tag{6}$$

where

$$\hat{Q}(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1}}\Big[V_{\bar{\phi}}(s_{t+1})\Big] \tag{7}$$

Our policy seeks to minimize:

$$L^\pi = \mathbb{E}_{s_t \sim D, \epsilon_t \sim N(0,1)}\left[\log \pi_\theta(f_\theta(\epsilon_t, s_t)|s_t) - Q_{\rho_1}(s_t, f_\theta(\epsilon_t, s_t))\right] \tag{8}$$

And our value function:

$$L^{\mathrm{V}} = \mathop{\mathbb{E}}_{s_t \sim D}\left[\frac{1}{2}\Big(V_\phi(s_t) - \hat{V}_\phi(s_t)\Big)^2\right] \tag{9}$$

Where

$$\hat{V}_\phi = \mathbb{E}_{a_t \sim \pi_\theta}\left[Q^{min}(s_t, a_t) - log\pi_\theta(a_t|s_t)\right] \tag{10}$$

And $Q^{min} = \min(Q_{\rho_1}(s_t, a_t), Q_{\rho_2}(s_t, a_t))$

SAC starts by doing policy roll outs, recording the state, action, reward, and the active controller at each time step. It stores these experiences in the replay buffer. After enough trials have been run, we run our update step. We sample from the replay buffer, and use these sampled states to compute the losses above. We then run one step of Adam [18] to update our network weights. We repeat this update n_u times with different samples. Finally we copy our weights to our target network and repeat until convergence (or some other stopping metric).

III. Switched Soft Actor Critic

Our primary contribution is to extend SAC in two key ways.

The first is to modify the structure of the learned controller in order to inject our domain knowledge into the learning. Our controller consists of three distinct components. The gate function, the balancing controller, and the swing-up controller. The gate, $G_\gamma : S \to [0, 1]$, is a neural network parameterized by weights γ which takes the observations at each time step and outputs a number g_t representing which controller it thinks should be active. $g_t \approx 1$ implies high confidence that the balancing controller should be active, and $g_t \approx 0$ implies the swing-up controller is active. This output is fed through a standard switching hysteris function (think a schmitt trigger), to avoid rapidly switching on the class boundary, parameters given in the appendix. The swing-up controller can be seen as the policy network from vanilla SAC, the action then is determined by 5. The parameters for these networks are given in the appendix. The balancing controller is a linear quadratic regulator $C : S \to A$ about the acrobot's unstable equilibrium. We use the LQR designed by sponge [5]:

Using

$$Q = \begin{pmatrix} 1000 & -500 & 0 & 0 \\ -500 & 1000 & 0 & 0 \\ 0 & 0 & 1000 & -500 \\ 0 & 0 & -500 & 1000 \end{pmatrix}, R = \begin{pmatrix} .5 \end{pmatrix}$$

The resulting control law is:

$$u = -Ks$$

with

$$K = [-1649.8, -460.2, -716.1, -278.2]$$

These three functions together form our policy, π_θ. Algorithm 1 demonstrates how the action is computed at each timestep.

We learn the basin of attraction for the regulator by framing it as a classification problem, our neural network takes as input the current state, and outputs a class prediction between 0-1. A one implying that the LQR is able to stabilize the system, and a zero implying that it cannot. We then define a threshold function $T(s)$, as a criteria for what we consider a successful trial:

$$T(s) \ = \|s_t - gs\| < \epsilon_{thr} \quad \forall t \in \{N_e - b, ..., N_e\} \tag{11}$$

Here s is understood to be an entire trajectory of states, N_e is the length of each episode, e_{thr} and b hyper parameters with values given in the appendix. We are following the

convention of a programming language here, (11) returns one when the inequality holds, and zero otherwise. To gather data, we sample a random initial condition, do a policy roll out using the LQR, and record the value of 11 as the class label.

To train the gating network we minimize the binary cross entropy loss:

$$L^{\mathrm{G}} = \mathop{\mathbb{E}}_{\gamma} - [c_w y_i \log(G_\gamma(s_i)) + (1 - y_i) \log(1 - G_\gamma(s_i))] \tag{12}$$

Where c_w is a class weight for positive examples, we set $c_w = \frac{n_t}{n_s} w$ where n_t is the total number of samples, n_p is the number of positive examples, and w is a manually chosen weighting parameter to encourage learning a conservative basin of attraction. We found that the learned basin was very sensitive to this parameter, a value of .01 empirically works well. Note that unlike the other losses above, the data here is not computed over a sample from the replay buffer, but is instead computed over the entire buffer (we store the state, gate. We found the gate was prone to "forgetting" the basin of attraction early in the training otherwise. This sounds inefficient, but it allows us to update the gate infrequently compared to the other networks, and so the actual wall clock hit is modest.

The second extension is a modification of the replay buffer D. We do this by constructing D from two separate buffers, D_n and D_r. Only roll outs that ended in a successful balance ((11) returns one) are stored in D_r. The other buffer stores all trials, the same as the unmodified replay buffer. Whenever we draw experience from D, with probability p_d we sample from D_n, and with probability $(1-p_d)$ we sample from D_r. We found this to speed up learning dramatically, as even with the LQR and a decent gating function in place, the swingup controller finds the basin of attraction only in a tiny minority of trials.

Algorithm 1 Do-Rollout(G_γ, Π_θ, K)

$s = r = a = g = r = \{\}$
Reset environment, collect s_0
for $t \in \{0, ..., T\}$ **do**
  $g_t = hyst(G_\gamma(s_t))$
  if $(g_t) == 1$ **then**
    $a_t = -K s_t$
  else
    Sample ϵ_t from $N(0, 1)$
    $a_t = \beta \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t) * \epsilon_t)$
  Take one step using a_t, collect $\{s_{t+1}, r_t\}$
  $s = s \bigcup s_t$, $r = r \bigcup r_t$
  $a = a \bigcup a_t$, $g = g \bigcup g_t$
return s, a, r, g

IV. Results

A. Training

To train SSAC we first start by training the gate exclusively, using the supervised learning procedure outlined in section III This allows us to get a decent estimate of the basin of attraction before we try to learn to reach it. We trained the gate for 1e6 timesteps, and then trained both in parallel using algorithm 2 for another 1e6 timesteps. The policy, value, and Q functions are updated every 10 episodes, and the gate every 1000. The disparity is because, as mentioned earlier, the gate is updated using the entire replay buffer, while all the other losses are updated with one sample batch from the buffer. Hyperparameters were selected by picking the best performing values from a manual search, which are reported in the appendix.

Algorithm 2 Switched Soft Actor Critic

Initialize network weights $\theta, \phi, \gamma, \rho_1, \rho_2$ randomly
set $\overline{\phi} = \phi$
for $n \in \{0, ..., N_e\}$ **do**
  $s, r, a, g =$ Do-Rollout$(G_\gamma, \Pi_\theta, K)$
  if $T(s)$ **then**
    Store s, r, a in D_n
  Store s, r, a in D_r
  Store $s, g, T(s)$ in D_g
  if Time to update policy **then**
    sample s^r, a^r, r^r from D
    $\hat{Q} \approx R + \gamma V_{\overline{\phi}}(S)$
    $Q^{min} = \min(Q_{\rho_1}(s^r, a^r), Q_{\rho_2}(s^r, a^r))$
    $\hat{V} \approx Q^{min} - \alpha H(\pi_\theta(A|S))$
    Run one step of Adam on $L^Q(s^r, q^r, r^r)$
    Run one step of Adam on $L^\pi(s^r)$
    Run one step of Adam on $L^V(s^r)$
    $\overline{\phi} = q\overline{\phi} + (1 - q)\phi$
  if Time to update gate **then**
    Run one step of Adam on L^G using all samples in D_g

In addition to training on our own version of SAC and Switched SAC we also examined the performance of several algorithms written by OpenAI and cleaned up by the community [19]. We examine PPO and TRPO, two popular trust region methods. A2C was included to compare to a non trust region, modern policy gradient algorithm. We also include TD3, which has been shown in the literature to do well on the acrobot and cartpole problems [20]

Stable baselines includes hyperparameters that were algorithmically tuned for each environment. For algorithms where parameters for Acrobot-v1 were available we chose those, some algorithms were missing tuned Acrobot-v1 examples, and for those we used parameters for Pendulum-v0, simply because it is another continuous, low dimensional task. Note we don't expect the hyper-parameters to make or break the performance, only effect how fast learning occurs. Reported rewards are averaged over 4 random seeds. Every algorithm makes 2e6 interactions with the environment. Also note that this project was largely inspired by spending a large amount of time manually tuning these parameters to work on this task (with no success better than what we see here). Figure 3 shows the reward curve for our algorithm and the

algorithms from stable baselines. Table II shows the mean and standard deviation for the final rewards obtained by all algorithms.

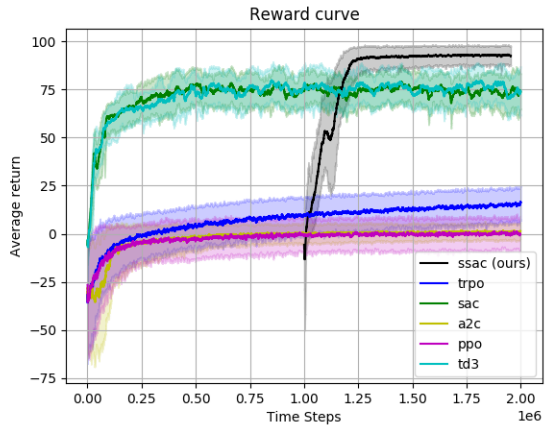

Fig. 3: Reward curve for SSAC and the other algorithms we compare it to. the solid line is the smoothed average of episode reward, averaged over four random seeds. The shaded area indicates the best and worst rewards at each epoch across the four seeds. A reward of 100 is the maximum possible reward, although it does take some amount of time to swing up the system. We excluded the rewards from SSAC during the training of the gating function, because they are all over the place and obscure the reward curve of the other functions.

Algorithm (implementation)	Mean Reward $\pm$ Standard Deviation
SSAC (Ours)	**92.12** $\pm$ **2.35**
SAC	73.01 $\pm$ 11.41
PPO	0.43 $\pm$ 8.89
TD3	78.67 $\pm$ 61.85
TRPO	17.63 $\pm$ 3.39
A2C	2.57 $\pm$ 3.63

TABLE II: Rewards after training for across learning algorithms. This table shows results after 2 million environment interactions

As we can see, for this environment, with the number of steps we have allotted, our approach outperforms the algorithms we compared to, with TD3 making it the closest to our performance. This is a necessarily flawed comparison. These algorithms are meant to be general purpose, so it is unfair to compare them to something designed for a particular problem. But that is part of the point we are making, that adding just a small amount of domain knowledge can improve performance dramatically.

B. Analyzing performance

To qualitatively evaluate the performance of our learned agent we examine the behavior during individual episodes. SSAC gives us a deterministic controller (we can set ϵ_t from 5 to zero). We chose the initial condition $s_0 = (-\pi/2, 0, 0, 0)$ and record a rollout. The actions are displayed in figure 4, and the positions in 5.

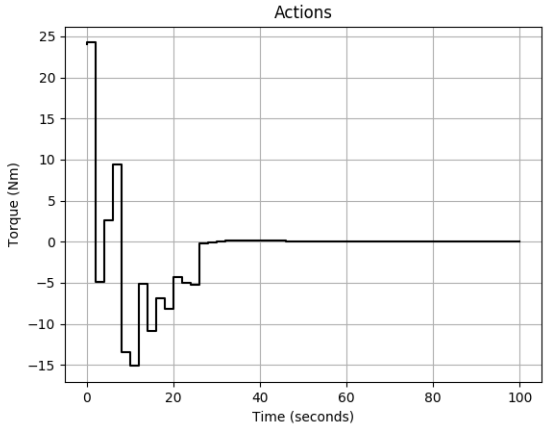

Fig. 4: Torque exerted during the sampled episode

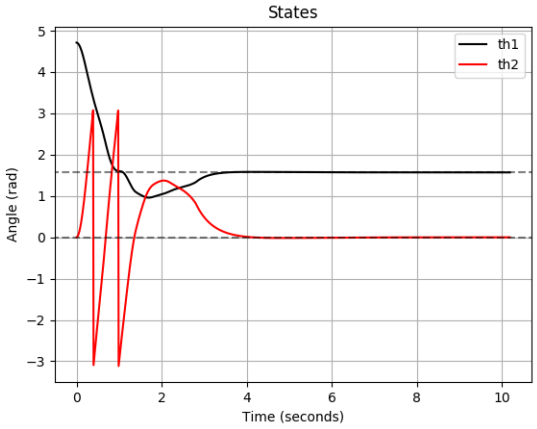

Fig. 5: Observations during the sampled episode

We have also found that despite achieving relatively high rewards, the other algorithms we compare to often fail to meet the balance criteria T(s). We often see solutions where the first link is constantly rotating, with the second link constantly vertical. To demonstrate this we run roll outs with the trained agents across a grid of initial conditions, recording if the trajectory satisfies T(s) or not. We compare our method with TD3, which was the best performing model free method we could find on this task.Figures 6 and 7 show the results. Note that we did conduct this experiment with the other algorithms and found that there performance here was worse.

V. CONCLUSIONS

We have presented a novel control design methodology that allows engineers to leverage their domain knowledge, while also reaping many of the benefits from recent advances in deep reinforcement learning. In our case study we constructed a policy to swing up and balance an acrobot

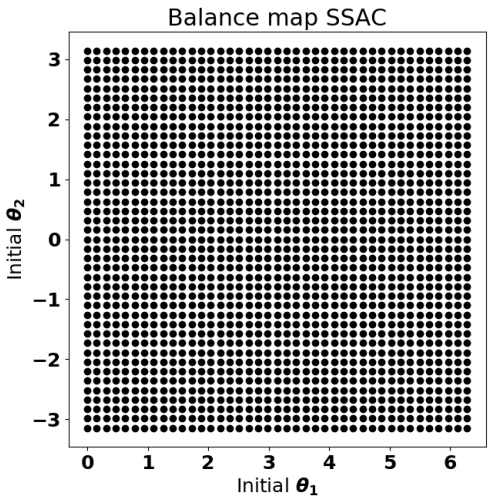

Fig. 6: Balance map for SSAC, X and Y indicate the initial position for the trial, a black dot indicates that the trial started from that point satisfies (11), and red indicates the converse

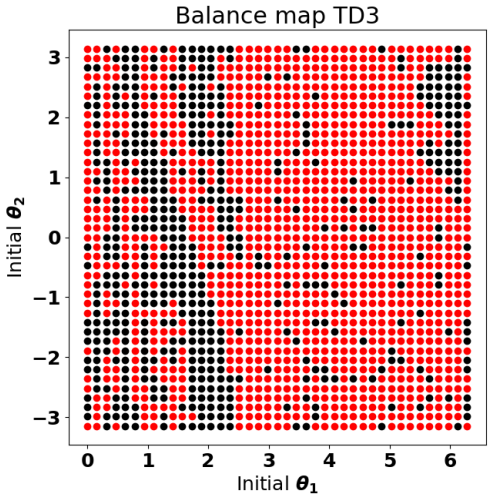

Fig. 7: Balance map for TD3, X and Y indicate the initial position for the trial, a black dot indicates that the trial started from that point satisfies equation (11), and red indicates the converse

while only needing to manually design a linear controller for the balancing task. We believe this method of control will be straightforward to apply to the double or triple cartpole problems, which to our knowledge no model free algorithm is reported as solving. We also think that this general methodology can be extended to more complex problems, such as legged locomotion. In that case the linear controller here could be a nominal walking controller obtained via trajectory optimization, and the learned controller could be a recovery controller to return to the basin of attraction of this nominal controller.

References

[1] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver, "Emergence of Locomotion Behaviours in Rich Environments," *arXiv:1707.02286 [cs]*, July 2017, arXiv: 1707.02286. [Online]. Available: http://arxiv.org/abs/1707.02286

[2] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning Dexterous In-Hand Manipulation," *arXiv:1808.00177 [cs, stat]*, Aug. 2018f, arXiv: 1808.00177. [Online]. Available: http://arxiv.org/abs/1808.00177

[3] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *Science Robotics*, vol. 4, no. 26, p. eaau5872, Jan. 2019. [Online]. Available: http://robotics.sciencemag.org/lookup/doi/10.1126/scirobotics.aau5872

[4] J. Lee, J. Hwangbo, and M. Hutter, "Robust Recovery Controller for a Quadrupedal Robot using Deep Reinforcement Learning," *arXiv:1901.07517 [cs]*, Jan. 2019, arXiv: 1901.07517. [Online]. Available: http://arxiv.org/abs/1901.07517

[5] M. W. Spong, "Swing up control of the acrobot using partial feedback linearization *," *IFAC Proceedings Volumes*, vol. 27, no. 14, pp. 833–838, Sept. 1994. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1474667017474040

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[7] T. garage contributors, "Garage: A toolkit for reproducible reinforcement learning research," https://github.com/rlworkgroup/garage, 2019.

[8] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2019.

[9] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. de Las Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, T. Lillicrap, and M. Riedmiller, "DeepMind control suite," https://arxiv.org/abs/1801.00690, DeepMind, Tech. Rep., Jan. 2018. [Online]. Available: https://arxiv.org/abs/1801.00690

[10] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap, "Distributed Distributional Deterministic Policy Gradients," *arXiv:1804.08617 [cs, stat]*, Apr. 2018, arXiv: 1804.08617. [Online]. Available: http://arxiv.org/abs/1804.08617

[11] M. W. Spong, "Energy Based Control of a Class of Underactuated Mechanical Systems," *IFAC Proceedings Volumes*, vol. 29, no. 1, pp. 2828–2832, June 1996. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1474667017581057

[12] J. RandlÃžv, A. G. Barto, and M. T. Rosenstein, "Combining Reinforcement Learning with a Local Control Algorithm," in *Proceedings of the Seventeenth International Conference on Machine Learning*, ser. ICML '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 775–782. [Online]. Available: http://dl.acm.org/citation.cfm?id=645529.657804

[13] J. Yoshimoto, M. Nishimura, Y. Tokita, and S. Ishii, "Acrobot control by learning the switching of multiple controllers," *Artificial Life and Robotics*, vol. 9, no. 2, pp. 67–71, May 2005. [Online]. Available: http://link.springer.com/10.1007/s10015-004-0340-6

[14] L. Wiklendt, S. Chalup, and R. Middleton, "A small spiking neural network with LQR control applied to the acrobot," *Neural Computing and Applications*, vol. 18, no. 4, pp. 369–375, May 2009. [Online]. Available: http://link.springer.com/10.1007/s00521-008-0187-1

[15] K. Doya, K. Samejima, K.-i. Katagiri, and M. Kawato, "Multiple Model-Based Reinforcement Learning," *Neural Computation*, vol. 14, no. 6, pp. 1347–1369, June 2002. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/089976602753712972

[16] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," https://github.com/openai/baselines, 2017.

[17] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *arXiv:1801.01290 [cs, stat]*, Aug. 2018, arXiv: 1801.01290. [Online]. Available: http://arxiv.org/abs/1801.01290

[18] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Dec. 2014, arXiv: 1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980

[19] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.

[20] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv:1509.02971 [cs, stat]*, Sept. 2015, arXiv: 1509.02971. [Online]. Available: http://arxiv.org/abs/1509.02971

APPENDIX

Hyperparameters

Hyperparameter	Value
Episode length (N_e)	50
Exploration steps	5e4
Initial policy/value learning rate	1e-3
Steps per update	500
Replay batch size	4096
Policy/value minibatch size	128
Initial gate learning rate	1e-5
Win criteria lookback (b)	10
Win criteria threshold (ϵ_{thr})	.1
Discount (γ)	.95
Policy/value updates per epoch	4
Gate update frequency	5e4
Needle lookup probability p_n	.5
Entropy coefficient (α)	.05
Polyak constant (c_{py})	.995
Hysteresis on threshold	.9
Hysteresis off threshold	.5

Network Architecture

The policy, value, and Q networks are all made of four fully connected layers, with 32 hidden nodes and Relu activations. The gate network is composed of two hidden layers with 32 nodes each, also with Relu activations, the last output is fed through a sigmoid to keep the result between 0-1.