Comparison of Sample Efficient Reinforcement Learning Algorithms

Hai To, Rohan Chauhan

Aalto University

Abstract. In recent years, application of reinforcement learning (RL) algorithm to learn dynamic systems have seen huge improvement with deep learning. Model based RL algorithms try to learn the model of the environment and maximize the reward with respect to the model whereas model free RL maximize the reward with respect to interactions from the environment. In this report, we compare model free RL algorithm Probabilistic Inference for Learning Control (PILCO) and it's deep learning variant called Deep PILCO to Soft Actor Critic (SAC) which is a model based RL algorithm. We present our findings with respect to the continuous control cart-pole swing-up task.

Keywords: PILCO · SAC · Deep PILCO

Rohan

1 Introduction

The field of artificial intelligence has been often inspired by the ways human learns. For example, deep learning is based on the functioning of neurons in human brain or genetic algorithm which is based on the process of natural selection of genes in human . Similarly, reinforcement learning (RL) algorithms can learn a task with trial and error just as humans do. Reinforcement learning is modeled as markov decision process (MDP) where there is an agent with a given set of actions and an environment which defines the transition probability and reward function for transition from one state to another. The goal of the agent is to maximize the reward in the environment for a given duration of timesteps. RL algorithms can be broadly classified into model based RL and model free RL. Model based RL algorithms try to learn a model of the environment and maximize their reward with respect to the learnt model whereas model free RL algorithms maximize the reward directly with respect to the environment. In recent years, reinforcement learning has seen great advances with the advent of deep learning like DDPG [6], TRPO [11] and A3C [8] .

However, applying reinforcement learning algorithms in real world is different in comparison to the simulated environment in which they are usually tested. Firstly, deploying RL algorithm in real world environment is difficult because we cannot account for all the factors. It is not possible to take into account all the variations offered by the real world in the simulated environment. So, RL algorithms need to generalize well to unseen situations. Secondly, RL algorithm usually require huge number of trials to learn a task which might not be possible in real world. We can be constrained by time, money or even due to wear and tear of the robotic system which is being trained. In such cases, we can use transfer learning where a model trained in simulated environment is deployed in real world. However, such models might not perform well due to poor generalization. The other solution is to lower the number of trials required to learn the task by improving data efficiency of the algorithm. Data efficient algorithms are able to learn more from lesser number of trials.

Data efficiency can be achieved in several ways. To increase data efficiency for task, we either need more knowledge from the expert or we need to extract more information from the data experienced during trials. Expert knowledge can be imparted by guiding the exploration in right direction. For example, in model based algorithm, we can model the dynamics of the environment using equations obtained from the expert. Experts can also guide the task in right direction by

demonstrating how to perform the task or by tuning the hyperparameters of the model suited for the task. Reward scaling is also very popular way to guide RL algorithms. However, in most cases, imparting expert knowledge can be tough due to task complexity or unknown dynamics. The second way, which is to extract more information from available data seems more ideal but has its own drawbacks. The process of improving the policy using the data collected from all the previous trials is known as experience replay. Experience replay can help in improving the convergence of an RL algorithm and better utilise the data. However, we need to carefully select the data which is used for policy improvement for models of finite capacity. Using only new trials means that the model will forget its bad experience and will try to re-explore those regions whereas using only older trials can lead to poor convergence.

In this project, we aim to examine Probabilistic Inference for Learning Control (PILCO) , its deep learning variant called Deep PILCO and Soft Actor Critic (SAC) from data efficiency perspective. We try to compare our results with the results from the research papers of respective authors. We present our findings and draw on conclusions based on data efficiency. The project report is organized as follows. In section 2, we describe the reinforcement learning algorithms used in the project followed by discussion of experiment in section 3. We present our results in section 4 and finally conclusions are drawn in section 5.

2 Reinforcement Learning Algorithms

Model based RL algorithms can suffer from model bias. Model bias occurs because of believing that the learnt model is the correct model of the environment. When no expert knowledge is present and there is not enough data to learn the right model, model based RL algorithms can perform poorly due to model bias. The case is worse when the model is deterministic. On the other hand model free RL algorithms suffer from low data efficiency and require careful tuning of hyperparameters for the task at hand. Model free RL algorithms require large number of trials even for simpler tasks. This often occurs due to on-policy learning because more data samples are required for gradient based learning of the right policy. Off-policy learning can re-use the past data but is more diffcult to tune for convergence.

2.1 PILCO

PILCO stands for Probabilistic Inference for Learning Control. PILCO consists of a probabilistic dynamics model and a deterministic policy. Initially, we collect some data using random trial. We use this data to train our probabilistic dynamics model. We then calculate the cost for current policy and improve our current policy using gradient based optimisation. We then set the optimized policy as our current policy and use it to gather more data. This data is again used to train the probabilistic dynamics model and the process continues until convergence. The complete algorithm is shown in Figure 1.

Algorithm 1 PILCO

1: *Define* policy's functional form: $\pi : z_t \times \psi \rightarrow u_t$.
2: *Initialise* policy parameters ψ randomly.
3: **repeat**
4: *Execute* system, record data.
5: *Learn* dynamics model.
6: *Predict* system trajectories from $p(X_0)$ to $p(X_T)$.
7: *Evaluate* policy:
$J(\psi) = \sum_{t=0}^{T} \gamma^t \mathbb{E}_X[\text{cost}(X_t)|\psi]$.
8: *Optimise* policy:
$\psi \leftarrow \arg\min_{\psi} J(\psi)$.
9: **until** policy parameters ψ converge

Fig. 1: PILCO Algorithm Improving PILCO with Bayesian Neural Network Dynamics Models [7]

Probabilistic Dynamics Model The probabilistic dynamics model is fed the current state and the action suggested by the policy as input and the target is the difference between the next state and current state. The dynamics model is implemented using a gaussian process (GP) with squared exponential kernel where the parameters to be learned by the model are lengthscale, output variance and the noise. Lengthscale determines how fast the function can vary whereas the output variance determines how much the function can vary away from its mean value. In case of high dimensional input, automatic relevance determination (ARD) is used. As longer lengthscales means that the function doesnt vary much with respect to that particular dimension, it can be removed from the modelling process. The parameters are learnt using evidence maximization which means to maximize the marginal likelihood of gaussian process with respect to the hyperparameters [9]. We can also obtain the posterior predictive distribution for the difference between the next final state and the given new state by conditioning on the posterior of gaussian process.

Policy Evaluation and Improvement The policy in PILCO is a RBF network using uses radial basis functions as activation function. Evaluating the policy requires the probability distribution for each state. One step prediction for the next state can be obtained by using the posterior predictive distribution. However, as the next state is also dependent on the selected action, we need the joint probability distribution for the current state and selected action at that timestep. The selected action is dependent on the current state and policy parameters. So, firstly the the predictive distribution for action is calculated out by marginalizing over the states. Then, the cross-covariance between the current state and suggested action is calculated. Cross covariance is the covariance of one stochastic process with another. We, then approximate the joint distribution over the current state and the selected action which will give us the probability distribution for the next state whose cost is to be calculated. The approximation is done using moment matching.

For calculating the cost over the entire episode, we use the law of iterated expectation. The expected total cost of the current policy depends on the sum of the expected cost of the states conditioned on the expected cost of previous states. The cost function is chosen such that the total cost can be computed analytically. For improving the policy, we calculate the gradient of the expected cost function with respect to the policy parameters. As, we know that the cost depend on states and states depend on chosen action, we can use chain rule to derive the final gradient equation with respect to policy parameters. Analytical gradient computation is chosen as it is much faster than sampling.

PILCO is highly data efficient. However, as mentioned by the authors, PILCO's data efficiency does not come from reward shaping. To quote the author, " Key to pilcos success is a principled way of reducing model bias in model learning, long-term planning, and policy learning." We believe that PILCOs data efficiency comes from the use of gaussian process. Whenever, the dynamics model is trained, it uses all of the previously collected data. Gaussian process is a non parametric

model and hence, it is able to account for all the previously seen data. When the policy is trained over this learnt dynamics models, it is able to learn a policy which is optimal for all previous trials. In a way, PILCO addresses the problem associated with experience replay and hence, is more data efficient compared to any other algorithm. However, PILCO suffers from its own set of problems which we will see in the next section.

2.2 Deep PILCO

Deep PILCO is the deep learning variant of PILCO which tries to improve over PILCOs weaknesses. Firstly, PILCO employs gaussian process which helps in bayesian modelling of uncertainty but do not scale well with the increase of data because inference in GP requires matrix inversion which is done using cholesky decomposition. As, the dimension of data increases, the inference becomes more and more difficult. Secondly, PILCO doesnt account for temporal correlation in model uncertainty between successive transitions by treating model uncertainty as noise. This means that the model errors are treated as independent during each timestep. This leads to PILCO underestimating the uncertainty at future timesteps because the error associated with the prediction adds up. [1]

Deep PILCO replaces gaussian processes used in PILCO with deep neural networks. In Deep PILCO, the probablistic dynamics model of PILCO is modeled using a bayesian neural network (BNN). The policy is a RBF network which is same as PILCO. Deep PILCO is able to imitate PILCO's behavior by propogating input uncertainty and output uncertainty using several modifications to adapt to neural networks. The overall algorithm remains same as that of PILCO which is shown in Figure 1. The step 6 of the Deep PILCO algorithm is shown in Figure 2.

Algorithm 2 Step 6 of Algorithm 1: *Predict* system trajectories from $p(X_0)$ to $p(X_T)$

Define time horizon T.
Initialise set of K particles $x_0^k \sim P(X_0)$.
for $k = 1$ to K **do**
  Sample BNN dynamics model weights W^k.
end for
for time $t = 1$ to T **do**
  for each particle x_t^1 to x_t^K **do**
    Evaluate BNN with weights W^k and input particle x_t^k, obtain output y_t^k.
  end for
  Calculate mean μ_t and standard deviation σ_t^2 of $\{y_t^1, ..., y_t^K\}$.
  Sample set of K particles $x_{t+1}^k \sim \mathcal{N}(\mu_t, \sigma_t^2)$.
end for

Fig. 2: Deep PILCO Algorithm Improving PILCO with Bayesian Neural Network Dynamics Models [7]

Dynamics Model To introduce output uncertainty in Deep PILCO's dynamics model, we need to use bayesian neural networks (BNN). In BNN, a posterior distribution is placed over the weights of the neural network. Calculation of this posterior distribution is intractable. Yarin et. al. presented that using dropouts in neural network can be interpreted as an approximation to the posterior distribution [3]. In neural networks, dropout is regularization technique in which some units of the deep neural network are dropped out or masked during training which helps other units to generalize well. In deep PILCO, output uncertainty is handled with the help of dropouts. To introduce input uncertainty in the dynamics model, particles method are used. A set of particles

are sampled from initial state distribution and then passed through the dynamics model. The output distribution of particles are fitted to Gaussian distribution by moment matching and re-fed into the dynamics model to obtain the state distribution for next state. This process is repeated until we obtain the state distribution of all states for the set time horizon. Repeatedly, applying dynamics model to obtain the distribution for final state from the initial state can be seen similar to a working of recurrent neural network (RNN). For approximate inference in RNN, the function weights are sampled once so similarly we keep the dropout masks fixed for the dynamics model. The policy is evaluated using a saturated cost function and optimized using ADAM [5].

Tuning Deep PILCO Improving the policy, using experience replay is more diffcult in deep PILCO than PILCO because neural networks are models of finite capacy compared to the non parametric gaussian processes. This means that as more and more data is accumulated the complexity of data increases but the model capacity remains same. We can either increase the capacity of model with the number of trials or downweigh older trials but both of them can lead to poor convergence. The most important part for deep PILCO is moment matching which forces the distribution of states to be unimodal leading to convergence of policy. Earlier works have faced problems in optimizing the policy with particle methods because of multiple local optimas. Deep PILCO solves this problem by resampling a set of new particles at every timestep. Lastly, the dropout in the dynamics model or in other words, modelling the output uncertainty is crucial to Deep PILCO. No dropout will result in model bias whereas with high dropout probability the model cannot learn anything.

Deep PILCO is definitely an improvement over PILCO considering that it can easily scale with data. Although, it is less sample efficient than PILCO, it has a lower total cost for learning a task compared to PILCO.

2.3 Soft Actor Critic

Soft Actor Critic (SAC) is model-free RL algorithm which replaces the original objective of maximizing rewards with maximizing rewards as well as the entropy of chosen actions by the agent. The objective of SAC has a temperature parameter which controls the importance of entropy in the policy. Setting it to 0 reduces the objective to standard RL objective of maximizing the expected reward. Maximum entropy objective helps by encouraging exploration as well as capture near optimal behaviour.It uses off-policy learning which helps to reduce its sample complexity and entropy maximization framework which makes it more robust. Actor critic architecture has actor which enacts the policy and critic which evaluates the policy. It alternates between improving the policy and evaluating the policy. The SAC algorithm is shown in Figure 3 and described below.

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$
  end for
end for

Fig. 3: Soft Actor-Critic Algorithm, taken from Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor [4]

Soft Policy Iteration There are two main steps in soft policy iteration. Soft policy evaluation computes the value of the policy as per maximum entropy objective. It requires soft Q value function and a soft state value function.

$$\mathcal{T}^\pi Q(s_t, a_t) \triangleq r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p}[V(s_{t+1})],$$

where

$$V(s_t) = \mathbb{E}_{a_t \sim \pi}[Q(s_t, a_t) - log\pi(a_t \mid s_t)]$$

For soft policy improvement, we reduce the KL divergence between chosen set of policy family and the improved policy.

$$\pi_{\text{new}} = \arg\min_{\pi' \in \Pi} \mathrm{D}_{\mathrm{KL}} \left(\pi'(\cdot|\mathbf{s}_t) \,\middle\|\, \frac{\exp\left(Q^{\pi_{\text{old}}}(\mathbf{s}_t, \cdot)\right)}{Z^{\pi_{\text{old}}}(\mathbf{s}_t)} \right)$$

In order to approximate the soft policy iteration algorithm, we use three neural networks. These 3 neural networks are soft state value function $V_\phi(s_t)$, soft q function $Q_\theta(s_t, a_t)$ and a policy network $\pi_\phi(a_t \mid s_t)$. The parameters of these networks are learnt by gradient descent.

SAC performs better than DDPG [6] in harder tasks. However, SAC is sensitive to reward scaling. For lower rewards, SAC is unable to learn whereas higher rewards can make the policy deterministic. Only right reward scaling can balance the trade-off between exploration and exploitation.

3 Experiments

This section presents the experiments conducted for each of the three methods. The hyperparameters are set as suggested in each original paper except some hyperparameters that are found to depend on the characteristic of the physic environment in used. Due to PILCO and DeepPILCO instability and lack of test from the original papers for high dimensional environments, we choose to test and compare all three methods on the popular entry level continuous control cart-pole swing-up task. In both PILCO and DeepPILCO papers, it is claimed that the method at hand

could learn the task successfully. Though no cart-pole domain benchmark was presented in SAC paper, it is one of the lower dimensional continuous control domains that SAC, a method targeted for high-dimensional complex tasks, is expected to perform well.

The cart-pole swing-up task from the Deepmind suite [12] was used. Similar to OpenAI Gym, Deepmind suite is a collection of benchmark tasks for reinforcement learning algorithms but only focuses on control tasks and provides more uniform interface across all the domains/tasks (especially the reward scale) which eases the comparison of methods across domains/tasks. Deepmind suite use Mujoco, a popular physics engine for studying reinforcement learning algorithms in control tasks. The cart-pole swing-up task requires the nonlinear controller to successfully swing-up and balance the pole. This hierarchical task consists of two easier subtasks on which the controller can be trained separately. The action is a continuous value in range [-1,1] as the torque applied to the cart to move it left or right on a rail limited between -1.8m and 1.8m. The pole has length of 1m and weight of 0.1kg. The mask of the cart is 1kg. The state is a vector of cart position, sine of pole angle (0 degree when the pole colinears with the right half of the rail), cosine of pole angle, cart linear velocity, pole angular velocity: $[x_c, \sin(\theta), \cos(\theta), \dot{x_c}, \dot{\theta}]^T$. The interaction with the environment is technically infinite in time but a time limit of 10 seconds is set for any rollout trajectory. The simulation timestep of Mujoco is set to 0.01 (10ms).

3.1 PILCO

The Tensorflow implementation of PILCO was taken from [10]. The implementation can be run out of the box. The only change is the environment: cart-pole swing-up task from Deepmind suite instead of the already deprecated 'inverted pendulum v2' in the latest OpenAI Gym version.

3.2 DeepPILCO

The original paper [7] did not come with any open source code. Our implementation of DeepPILCO was based on a Github repository [13]. However, the author failed to achieve any good policy. As in the original paper [7] saturating cost function is used: $1 - \exp(-0.05d^2/\sigma_c^2)$ where $\sigma_c = 0.25m$ and d^2 is the squared Euclidean distance between the pendulums end point (xp; yp) and its goal (0; l). This is the standard setting used with PILCO as well. The coefficient is modified from 0.5 to 0.05 in order to make the cost difference over change in distance larger hence clearer reward trace. The dropout probability for the BNN approximation of the dynamics model is set to 0.05 as the best value stated in [7]. Weight decay of the dynamics model is 10^{-4} as in the paper and no model's observation noise was used. The dynamics model network architecture was the same as in the paper. For the controller, we test both the same RBF network and a new MLP network with 2 fully connected layers 200 units each. The action output is `tanh_()-ed` to keep the action value in [-1,1]. The output is also augmented with zero mean, 0.1 standard deviation white noise. Both gradient clipping and exponentially decaying learning rate are tried.

As in line 3 to 5 of DeepPILCO algorithm, Figure 2 the training starts with collecting five episodes (K = 5) with a duration of 200 timesteps using a random linear policy (the policy is random at the beginning when the pole head is close to starting point but when the angle difference is above a threshold, a linear controller is applied). We keep all the sampled episodes in a non-discarding replay buffer from where transitions are sampled in batches to train the dynamics model. Then we fit the dynamics model and the policy and the current learned policy is used to sample one 200-timestep-length episode from the environment which also gets pushed to the replay buffer. The above sequence is repeated for many trials. To fit the model, 50 optimization epochs are used. Each epoch uses a batchsize of 100. This is much shorter than $5 * 10^3$ optimization steps in [7] since only one episode is collected each trial. To optimize the controller, in the forward step, 100 particles randomly initialized from the environment are propagated through the dynamics model (one fixed dropout mask for each particles) using actions from current learned policy. The rollout time horizon T experimented with are 100, 150 and 200. Note that the Deepmind suite environment used here has the Mujoco simulation timestep set to 10ms while in [7] it is 100ms. This justifies our use of long interaction horizon of 150 to 200 timesteps intead of 25 timesteps in [7].

3.3 SAC

The SAC paper [4] came out with publicly available source code containing full tests for all benchmark tasks with pre-tuned hyperparameters. Although there is no test and hyperparameters available for cart-pole swing-up task, thanks to SAC robustness and hyperparameter insensitivity, by reusing the same hyperparameter set from the simplest available benchmark 'hopper', we get good result almost straight out of the box. As stated in [4], SAC is sensitive to the scaling of the reward signal, we experimented with different scales starting from 5 similar to 'hopper' task. The episode length is 1000 unchanged from the default one.

4 Results

4.1 PILCO

We fail to produce any successful result from the test. The rollout time horizon T of 40 steps as in [2] and [10] does not allow the agent enough time to see any useful transition. Simply increasing T to 100 or higher would result in runtime error due to out-to-memory. This is expected since PILCO is known for having the computational resource expense increased along with simulation time and environment dimension. To get over this, a frame-skipping mechanism with well-tuned number of skipped frames should be implemented. We were late to discover this. By that time, we already moved on to DeepPILCO.

4.2 DeepPILCO

In general, from our observation, while the dynamics model is relative easy to train, the policy fitting is quite hyperparameter sensitive especially with the learning rate and the initial random trajectory samples. The policy fitting in the very first trials most often would decide the convergence of the training. This policy fitting again depends on the amount of useful data the dynamics model is able to encapsulate after fitting against the samples from the random linear policy. Hence, it is important that rich and diverse state transitions are observed during the random initialization step (especially those transitions that lead to higher rewards). Keeping all sampled trajectories in the replay buffer rather than discarding the oldest one when a capacity limit is reached slows down the training gradually but helps reduce the fitting variance. The MLP network controller performs much better than the RBF network in many tests with different hyperparameter combinations.

We discover that time horizon T is a very important hyperparameter to tune right. Initially, with T = 100, regardless of how other hyperparameters are tuned, the best policy learned was always to keep a maximum constant action output of either 1 or -1 throughout the interaction. This results in the cart accelerating to one end of the rail and get the pole swung up upon collision. This is a sub-optimal policy and impractical. Then, with higher T = 150 and 200 and appropriate tuning of other hyperparameters, more intuitively correct controller is learned Figure 4 and Figure 5. With T = 150, the learned policy though better than that of T = 100, it requires multiple attempts to swing the pole to the upright position. However, the motion is quite aggressive, resulting in a high angular velocity of the pole at the upright position, thus making balance task after that more difficult. T = 200 resulted in a controller that was able to get the pole into a stable upright position for a short time after successfully swinging the pole up during first attempt. This seems to be because higher T, despite making training more unstable, allows the policy to observe and learn from transitions that contain the result of earlier moves (timeteps before 100). In fact due to the physical constraint of the simulation, it seems impossible to achieve swing-up before timestep 100 Figure 6.

Decaying learning rate stabilizes the training with higher T. The 0.8 decaying coefficient after every 5 trials turned out to work best. Experiments show that with decaying learning rate, gradient clipping is unnecessary.

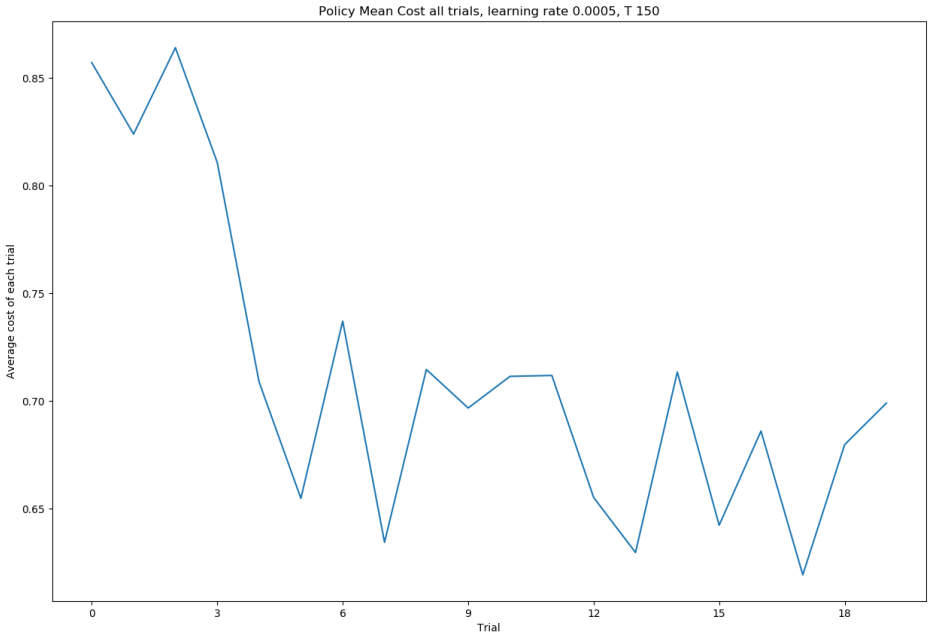

Fig. 4: DeepPILCO learning curve for T = 150. Here the learning rate is fixed at 0.0005.

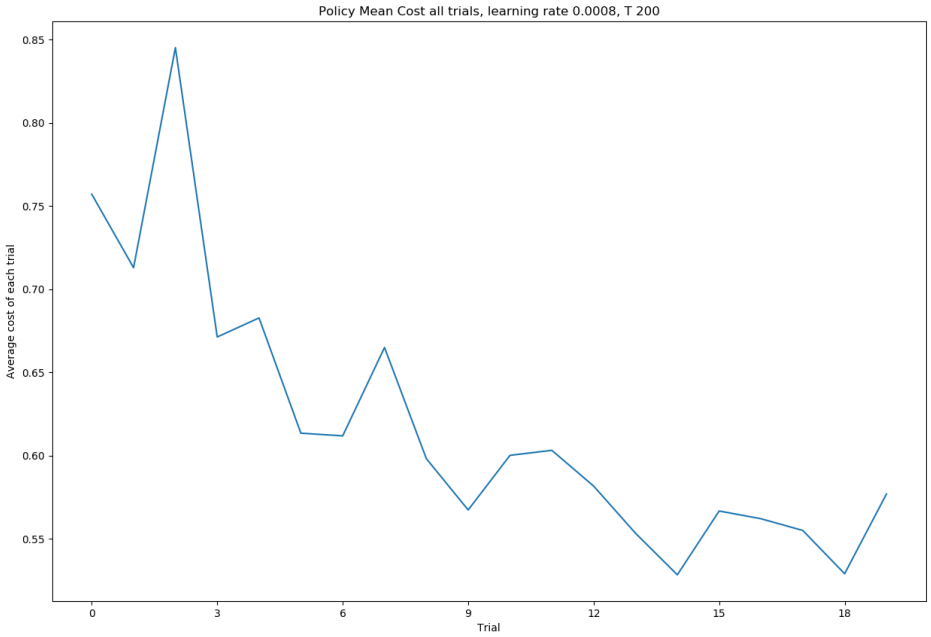

Fig. 5: DeepPILCO learning curve for T = 200. Here the learning rate is exponentially decaying from 0.0008

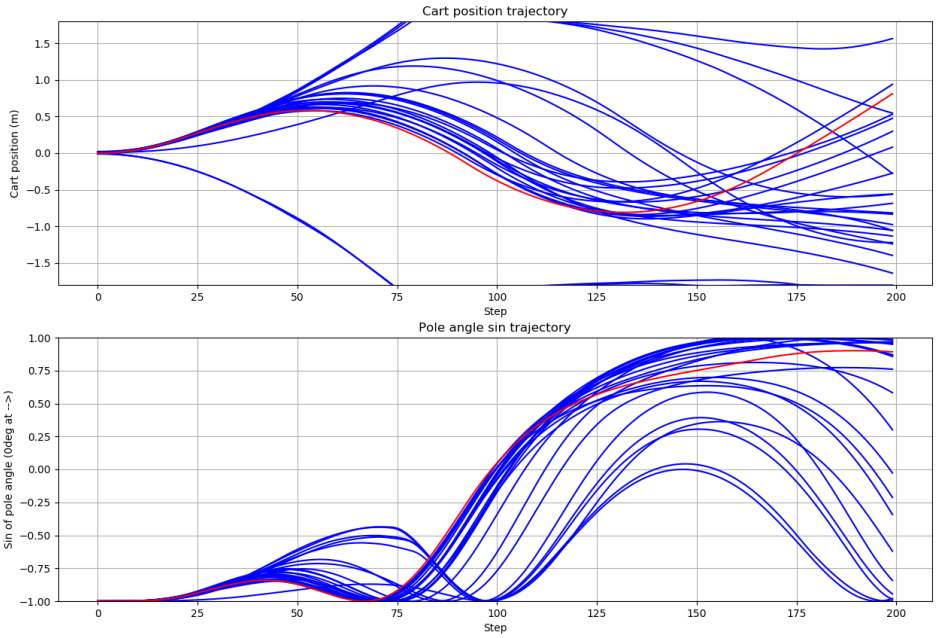

Fig. 6: DeepPILCO all cart position and sin of pole angle trajectories for 20 trials. Pre-balancing happens when sin of pole angle starts to converse to 1.0

4.3 SAC

The best result was achieved when we train the SAC agent for 200 episodes (200,000 steps) using a reward scale of 5. The learning curve is shown in figure Figure 7. Note that the learning curve is the return of each episode. In Deepmind suite environment, reward for each timestep ranges from 0 to 1. Therefore, with the episode length of 1000 timesteps, the maximum return at the end of each episode is also 1000. The learned policy is able to achieve the cart-pole swing-up task.

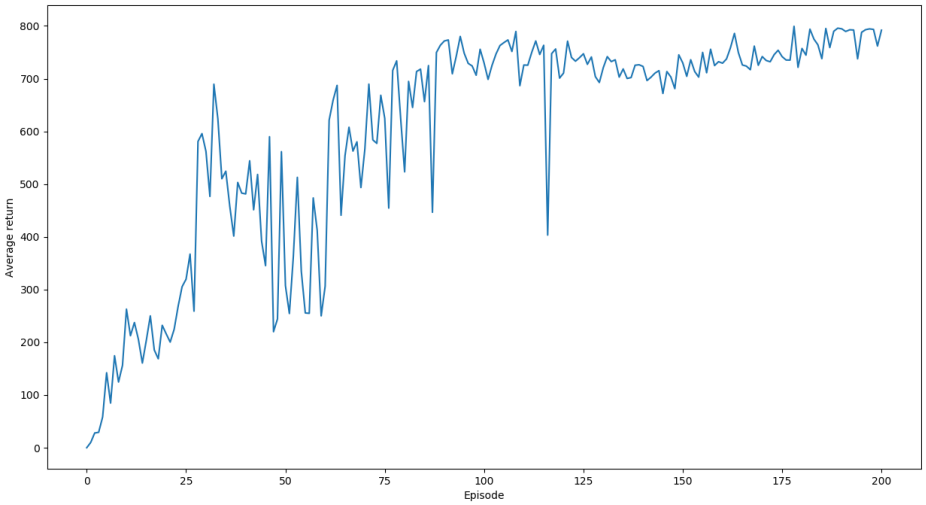

Fig. 7: SAC learning curve. Average episode return for 200 episodes

4.4 Comparison

Since we do not have any useful result from PILCO, the comparison is only between SAC and DeepPILCO on the cart-pole swing-up task. We tested the best policy of SAC and DeepPILCO in a 10-second-long session using Deepmind suite viewer functionary to render the environment. The average return for 300 steps and full 1000 steps are in Table 1. The DeepPILCO policy was faster to get the pole upright at around 2 seconds into the episode thus having higher average reward during 300 steps. However, since the DeepPILCO policy was unable to balance the pole after that, its average reward for the whole 10 seconds was only about half of SAC's.

Method	300 steps (3 seconds)	1000 steps (10 seconds)
SAC	0.309	0.739
DeepPILCO	0.556	0.385

Table 1: Average reward for 10 seconds test

We also plot the action traces for the above test to show that both SAC and DeepPILCO policy actually do generate the same 'good action sequence' pattern required for swing-up task at

the beginning. Figure 8 However, after that only SAC policy was able to generate a sequence of balancing actions.Figure 9

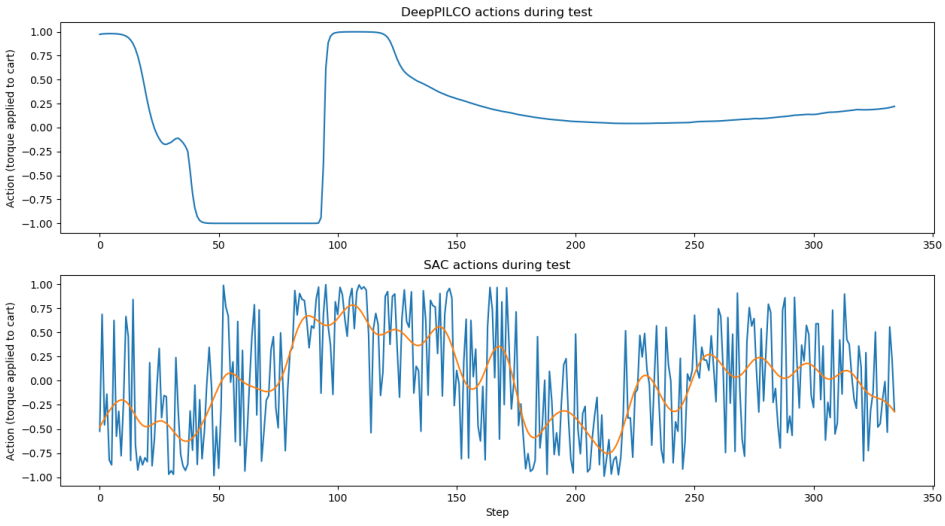

Fig. 8: Action sequence from Deep PILCO and SAC best policy. The smoothed out action sequence of SAC is in orange. The same pattern can be seen here. First the cart moves to one end a bit then switches to the other end for a longer time, then abruptly reverses for a short distance. Finally the action sequence tends to concentrate around zero for balancing.

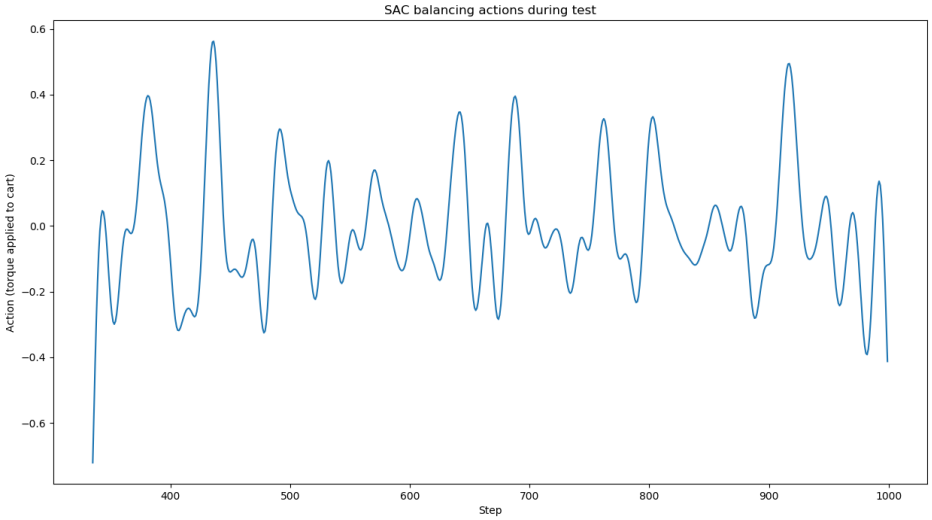

Fig. 9: SAC balancing actions. The action sequence is already smoothed out with a low-pass filter. Intuitively, the torque fluctuates around 0 resulting in the balance of the pole

For the training sample efficiency, metric to compare is the number of environment interaction steps. For SAC, it's clearly 200,000 steps. For DeepPILCO, it's five random policy initial sampling episodes x 200 steps plus one episode x 200 steps every trial after that with the learned policy. It requires about 20 trials to get the best policy so the total number of steps is 5000. Increasing the episode length (both number of interaction steps with the environment and time horizon T with the dynamics model) is a promising direction to get the DeepPILCO agent to learn the balancing skill. Even with that, an increase from 200 to about 300 or at most 400 steps still results in a much better sample efficiency than SAC.

5 Conclusion

In this experiment, we try to reproduce and compare the result of SAC, PILCO and DeepPILCO on a simple continuous control task of cart-pole swing-up. We could only confirm the performance of SAC and partially DeepPILCO. Despite very minimal requirement for hyperparameter tuning, SAC as an off-policy model-free reinforcement learning algorithm could successfully learn a robust policy for this task. DeepPILCO, on the other hand, is very sensitive to hyperparameters tuning and initial sampling of experiences from random policy. Though it fails to learn the balancing skill, the learned swing-up action sequence is very optimal in term of time and the pole is set upright in a very stationary pose ready for a balancing control sequence to be applied with minimal effort. We believe that with higher T and appropriate hyperparameters fine-tuning, optimal policy for both swing-up and balance subtasks could be learned. In comparison to SAC, DeepPILCO as a model-based method is more sample efficient though SAC can be robustly applied to higher dimensional tasks. We fail to reproduce the result of PILCO due to the computational complication of Gaussian Process to learn from big amount of data. To keep the data fed to the dynamics model minimal without throwing away too much information, an appropriate number of frames should be skipped when collecting experience from the simulated environment. The failure of PILCO from learning from the same amount of data confirms the improvement of DeepPILCO over PILCO using BNN approximated dynamics model but still keeping a good sample efficiency.

References

1. Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *IEEE transactions on pattern analysis and machine intelligence*, 37(2):408–423, 2015.
2. Marc Peter Deisenroth and Carl E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *ICML*, 2011.
3. Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.
4. Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. 2017.
5. Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
6. Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.
7. Rowan McAllister and Carl E. Rasmussen. Improving pilco with bayesian neural network dynamics models. 2016.
8. Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.
9. Carl Edward Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.
10. Nikitas Rontsis and Kyriakos Polymenakos. A modern and clean implementation of the pilco algorithm in tensorflow. https://github.com/nrontsis/PILCO, 2018.
11. John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.
12. Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy P. Lillicrap, and Martin A. Riedmiller. Deepmind control suite. *CoRR*, abs/1801.00690, 2018.
13. Zuoxingdong. reproducing the results of the deep pilco in pytorch. https://github.com/zuoxingdong/DeepPILCO, 2018.