Project Report: Cart-Pole Swing-up with Deep Reinforcement Learning

Huawei Sun, Kang Lin, Tobias Sukianto

***Abstract*— In this report we present our reinforcement learning methods that we have used for the cart-pole swing-up. Here, we focus on model-free deep-learning based methods. We present one Q-Learning approach, namely the *Deep Q-Network*, and two Actor-Critic approaches. One is a *Deep Actor-Critic* which uses deep neural networks for both, the actor and the critic. The other is a *Random-Feature Actor-Critic* which uses a linear random-feature model as critic and a deep neural network as actor. We compare the methods for the cart-pole swing-up and discuss our results.**

I. INTRODUCTION

In recent years, deep-learning based reinforcement learning (RL) methods have shown impressive result (e.g. Deepmind's Alpha-Zero [5]). We take this project as an opportunity to gather experiences with deep reinforcement learning. The goal is to use deep reinforcement learning in order to accomplish the cart-pole swing up.

We start by introducing and explaining the methods used in Sec. II. These methods will be evaluated on the cart-pole in Sec. III in which we also discuss the results and methods in more detail. Lastly, we conclude our findings in Sec. IV.

II. METHODS

In the following, let $\mathcal{S}$ denote the state-space and $\mathcal{A}$ the action-space of an environment.

A. Deep Q-Network

In this Q-learning approach, we approximate the action-value function $Q : (\mathcal{S} \times \mathcal{A}) \to \mathbb{R}$ by a deep neural-network, which we refer to as Deep Q-network (DQN), denoted $Q(s, a; \theta)$, where θ are the trainable network parameters [2]. Furthermore, we assume a discrete action space. As an off-policy method, we decide to use the *ϵ-greedy* algorithm to trade off exploration and exploitation. Hence, at any time-step and state $s \in \mathcal{S}$, we choose an action $a \in \mathcal{A}$ according to the ϵ-greedy strategy to move to the next state $s' \in \mathcal{S}$ and obtaining the reward r. These (s, a, r, s') samples are saved in a *memory* which we use to perform so-called *Experience-replay*.

Experience-replay allows to use past experience to learn the optimal action-value function. After each time-step, we randomly select a mini-batch of transitions (s, a, r, s') from the memory, compute the expected state-action values

$$\hat{q} = r + \gamma \max_{a'} Q\left(s', a'; \theta\right), \tag{1}$$

where γ is the discount-factor, and then perform gradient descent step on the parameters θ by computing the gradient of the mean-square error (MSE) between $\hat{q}$ and $Q\left(s, a; \theta\right)$ for all samples in the mini-batch.

Experience-replay is used for two main reasons:

- Deep neural networks, as supervised learning models, require that the data collections satisfies the independent identical distribution (i.i.d.) assumption in order for stochastic-gradient descent to work properly. However, due to biased sampling in Q-Learning and the non-stationarity of the problem, this assumption is not satisfied. By using Experience-replay this problem can be mitigated.
- Without Experience-replay, the gradient descent algorithm is only applied for the current sample at each time. However, with experience replay, we can select historical data for training. This can improve the efficiency and stability of the algorithm.

The above approach still suffers from stability issues during training. Recall that in (1), the target values $\hat{q}$ are calculated using the current Q-network. In this way, the targets are highly non-stationary and reveal strong correlations between them. This can lead to so-called *catastrophic forgetting* in neural networks, and thus, to drastic convergence issues of the algorithm.

Hence, we use Deep Q-learning [3]. It uses two structurally identical networks. One is simply the DQN $Q\left(s, a; \theta\right)$ which is used to select actions and whose parameters are being updated. The other network is the *target*-network $\hat{Q}(s, a; \hat{\theta})$ and is only used to calculate the target value $\hat{q}$. That is, $Q\left(s, a; \theta\right)$ in (1) is substituted by $\hat{Q}(s, a; \hat{\theta})$ yielding

$$\hat{q} = r + \gamma \max_{a'} \hat{Q}(s', a'; \hat{\theta}). \tag{2}$$

The parameters of the target network are not being optimized using gradient descent; rather, they are copied in from the network $Q\left(s, a; \theta\right)$ after certain time intervals. Now, at each time step, Experience-replay is performed using (2) to update the parameters of the DQN $Q\left(s, a; \theta\right)$.

In our approach, we design the DQN such that it takes as input a state s and outputs K values, where K is the total number of actions. Each output value then corresponds to one specific state-action pair. The network has 5 hidden layers with 1000 units each. The activation functions of the hidden layer units are *Rectified Linear Units* (ReLU). The outputs-layer has no activation function. In the following, "DQN" shall always refers to the DQN approach.

B. Deep Actor-Critic

For this actor-critic approach we assume a continuous action-space and a policy denoted by the random variable $\pi(a|s)$. We optimize the policy over a class of distributions to achieve the given task. As actor-critic methods are on-policy temporal-difference methods [6, p. 331] we have to

strictly follow the policy when moving in the environment to collect samples.

The presented Deep Actor-Critic (DAC) method is a memory-based method and inherits the key ideas of the DQN. That is, we utilize Experience-replay and target-networks. The method uses two deep neural networks, one as the critic and one for the actor. However, for the training itself we use another two networks, one as *target-critic* and one as *target-actor*. These target-networks share the same purpose as the target-network from the DQN approach, i.e., increasing the stability during training. The critic-network takes as input a state $s \in \mathcal{S}$ and outputs a scalar, as it represents the state-value function $V_C : \mathcal{S} \to \mathbb{R}$.

We decide to model $\pi(a|s)$ as a Gaussian distribution. Therefore, the actor-network takes as input a state $s \in \mathcal{S}$ and outputs a two-dimensional real-valued vector. One output is interpreted as the mean $\mu_A(s)$ of the Gaussian, the other output is interpreted as its standard deviation $\sigma_A(s)$. This yields the policy

$$\pi_A(a|s;\theta_A) \sim \mathcal{N}(\mu_A(s;\theta_A), \sigma_A^2(s;\theta_A)), \tag{3}$$

where θ_A are the trainable parameters of the actor-network. Especially, for the given cart-pole task we restrict the outputs to be $\mu_A(s;\theta_A) \in (-10, 10)$ and $\sigma_A(s;\theta_A) \in (0, 5)$ by utilizing tanh-activation function and sigmoid-activation function in the output layer, respectively. Other than that, both networks share the same hidden-layer structure as the DQN from Sec. II-A. Though, these are only design choices and need not to be followed. We denote the target networks by a hat symbol, i.e., target-critic $\hat{V}_C$ and target-actor $\hat{\pi}_A$.

The training of the critic-network is similar to the training of the DQN. We sample a random mini-batch from the memory and compute the expected state values

$$\hat{v} = r + \gamma \hat{V}_C(s'; \hat{\theta}_C), \tag{4}$$

where $\hat{\theta}_C$ are the non-trainable parameters of the target-critic network. Then, we perform one gradient-descent step on the trainable parameters θ_C of the critic-network $V_C(s;\theta_C)$ by computing the gradient of the MSE between $V_C(s;\theta_C)$ and $\hat{v}$, for all (s, a, r, s') in the mini-batch.

The actor-network is also updated in a similar fashion; however, we use the mini-batch to first compute the temporal differences $\delta = \hat{v} - V_C(s;\theta_C)$, which allows to compute the *action-targets*

$$\hat{z} = \log \hat{\pi}_A(a|s;\hat{\theta}_A) + \beta\delta, \tag{5}$$

where $\pi(\cdot|s)$ in this context means to compute the probability density value at the respective argument, and $\beta > 0$ is a hyperparameter determining the size of the differences between the target log-probability-density values and the log-probability-density values for the actions yielded by the current policy. Then, we perform one gradient-descent step on the actor-parameters θ_A by computing the gradient of MSE between $\log \pi_A(a|s;\theta_A)$ and the action-targets $\hat{z}$.

Note that in this approach, the updates are executed in a slightly different manner, as compared to the lecture notes.

C. Random-Feature Actor-Critic

Motivated by the *double-descent* phenomenon [4] and implicit regularization [1], we present as third method an over-parameterized random-feature model for the Actor-Critic approach. Furthermore, to introduce even more differences to the previously introduced DAC method, we discretize the action-space, thus, making the policy $\pi(a|s)$ a discrete probability distribution.

We again utilize Experience-replay and target-networks for critic and actor. The actor is a neural-network denoted by $f : \mathcal{S} \to \mathbb{R}^K$, where K is the total number of actions. The hidden-layer structure is the same as for DQN. Note that this time, the actor-network has as many outputs as there are actions. They correspond to scores that are fed into the soft-max function, whose outputs are interpreted as probabilities. Hence, the policy implied by the actor is expressed as

$$\pi_A(a_i|s;\theta_A) = \frac{e^{f_i(s;\theta_A)}}{\sum_{k=1}^{K} e^{f_k(s;\theta_A)}}, \tag{6}$$

where $f_i(s)$ is the i-th output of the actor-network.

The critic, which acts as the state-value function $V_C : \mathcal{S} \to \mathbb{R}$, however, is modeled as linear random-feature model, i.e.,

$$V_C(s) = \theta_C^\top \phi(s), \tag{7}$$

where $\theta_C \in \mathbb{R}^q$ are the trainable parameters, and $\phi : \mathcal{S} \to \mathbb{R}^q$ is a q-dimensional, random but fixed (i.e. non-trainable) feature-map. In the over-parameterized regime, q is assumed to be much larger than the number of training samples n, i.e., $q >> n$. Without prior knowledge, there are no restrictions on how to choose ϕ, thus, for the sake of simplicity we define

$$\phi(s) = \mathrm{ReLU}(Ws), \tag{8}$$

where $W \in \mathbb{R}^{q \times d}$ is random matrix whose entries are i.i.d. sampled from a uniform distribution, and ReLU is the *rectified linear unit* typically used in neural-networks as activation function.

The critic, as well as the actor, are updated accordingly as described for the DAC method in Sec. II-B. However, only for the actor we compute the action-targets in a slightly different manner. That is,

$$\hat{z} = \hat{f}(s;\hat{\theta}_A) + (1 - \pi_A(a|s;\theta_A))\beta\delta. \tag{9}$$

Note, that in this approach we assume to have a fixed number of training examples n. However, in reinforcement we may collect samples for an indefinite amount of time. Nevertheless, we may assume that the number of training examples are defined by the memory size of the Experience-replay.

III. EVALUATION

A. Experiment Set-up

We evaluate our approaches on the cart-pole swing-up. For training, we utilize a discount-factor $\gamma = 0.9$. We train on episodes containing 5 (simulated) seconds each. The total training time for the DQN is 2000s, for DAC 2500s,

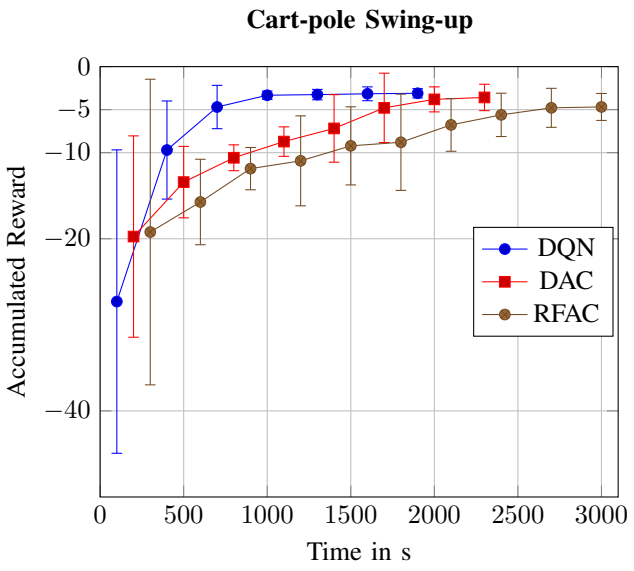

Fig. 1. Performance of Deep Q-Network, Deep Actor-Critic and Random-Feature Actor-Critic on the cart-pole swing-up. Depicted are the average accumulated rewards with 95% confidence intervals.

and for RFAC 3000s. Every 100 seconds we run a test-episode containing also 5 seconds. During each test-episode the (undiscounted) reward is accumulated and saved. The described experiment is run 10 times independently for each method. Lastly, we compute the average accumulated rewards of all 10 runs, which we use to assess the performance of our methods.

DQN and DAC use a replay-memory of size 2000. RFAC uses a replay-memory of 250, which is motivated by trying maximize the ratio between model-complexity q and number of samples n. The mini-batch size is 128 for all methods. The target-network updates happen every 10 seconds and the Adam-optimizer with a learning-rate of 10^{-4} is utilized for the network updates.

For the DQN approach, the action-space is discretized to contain 11 actions that are evenly spaced within the interval $[-10, 10]$. The ϵ of the ϵ-greedy algorithm changes according to $\epsilon(t) = \epsilon_\infty + (\epsilon_0 - \epsilon_\infty)\exp(-t/10^4)$, where $\epsilon_0 = 0.99$, $\epsilon_\infty = 0.01$ and t is the current number of time-steps.

We increase the hyperparameter β of the DAC approach with increasing training time. We start with $\beta = 0.3$, after 300 seconds we increase it to $\beta = 0.5$, after 500 seconds to $\beta = 1$ and after 1000 seconds it is increased one last time to $\beta = 1.5$.

The model-complexity for the RFAC is set to $q = 10^5$ and $\beta = 0.6$.

Generally, all these hyperparameter choices are found by empirical testing. More detailed explanations and motivations for the choices are found in the Sec. III-C.

B. Results

The results are illustrated in Fig. 1. The graphs depict the average accumulated rewards and the 95% confidence intervals from the 10 runs. The blue graph shows the result of DQN, the red graph shows the result for DAC, and the brown graph shows the result for RFAC.

We notice that DQN starts at a value of -28 and converges to a value above -5 after 700s, while DAC starts at a value of -32 and reaches the same performance after 1700s. RFAC starts at a value of -26 and converges after 2500s.

From empirical observations we have noticed that a value above -5 implies a quick and successful swing-up and balancing of the cart-pole around the center of the track. Hence, the results imply that DQN successfully accomplishes the task the fastest, while DAC takes around 2.5 times that time, and RFAC about 3.5 times that time.

C. Discussion

1) DQN: It is not surprising that the fastest convergence among the presented methods is achieved by the DQN. As an off-policy method, it can freely explore all actions without any restrictions, which allows to determine the optimal trade-off between exploration and exploration more easily. Therefore, finding the optimal policy is expected to be faster than on-policy approaches such as Actor-Critic methods.

As we use the ϵ-greedy strategy to trade of exploration and exploitation, we decide to start with a high ϵ-value and then decrease it with increasing training-time. This allows for more exploration early in the training but for more exploitation later. Furthermore, the action-space has been discretized to contain 11 actions, which possibly contributes to an even faster search.

However, if we were to test on a real robot, then DQN may not be as safe as on-policy methods, like DAC or RFAC, due to its off-policy nature which could damage the robot or contribute to a larger wear-out. Nonetheless, it is impressive to see that the DQN can achieve the task in such a short time.

2) DAC: Although the DAC takes about twice as along as the DQN to achieve the task, its performance, as we find, is still impressive, since not only does it consider an on-policy search for the optimal policy but it also searches within the entire continuous action-space.

However, one could possibly reach faster (or slower) convergence by loosening (or tightening) the restrictions on the hyperparameter β or on the actor-outputs. On the hand, choosing β small allows for more exploration but reaching convergence may be time-consuming since the policy-gradients would decrease. On the other hand, choosing β large results in stronger policy changes which may be in favor of obtaining good performance more quickly, but at the same time risk to not converge at all or risk falling in a local optimum of the given task. This behavior has been observed in simulations of the cart-pole swing up. Hence, for this task we decide to set β small at the beginning of training and later increase it. This should allow for more exploration early in the training, while focusing on exploitation in later stages of training. Now, by allowing a larger range of standard deviation $\sigma_A(s)$, one could trade-off exploitation for more exploration. However, on a real robot, this strategy may introduce higher wear-out or damaging risks.

If such risks are of big concern, then choosing DAC over DQN could be more beneficial even though DAC takes longer to learn the task. Another advantage is that DAC can exploit the entire continuous action-space which allows for more general settings, while DQN needs to find a proper discretization of the action-space. However, if damaging risks are not of concern and one can afford to discretize the action-space, then DQN may be a better choice.

3) RFAC: RFAC takes the longest time to achieve the task, but this may not be surprising, since, first, we assume an over-parameterized random feature model for which convergence is suspected to be reached after many gradient-descent steps [4].

Secondly, in each run an entirely new class of models is sampled as the the feature map $\phi(s)$ is random. Therefore, we do not only take the average over the stochastic policy and trainable parameter initialization, but also over the random initialization of the model class. This is not the case for the DQN and DAC approach since the model class there is fixed. Hence, the average also takes into account models that are just inherently better or worse.

Nevertheless, we can also see advantages from the RFAC curve. Notice that the slope is smaller than for DQN or DAC. This may implies that we can learn the task more safely by trading off time for smaller damaging-risks since the policy changes more smoothly. Another, but rather obvious, advantage is that the critic of the RFAC is only a simple linear model, which for example exhibits possible computational benefits over the DQN or DAC.

A strange behavior that has been noticed while performing the experiments is that RFAC often manages to balance the cart-pole fairly early on but not right at the center of the track, thus, accumulating fewer rewards. This behavior has not been observed in DQN or DAC. Those methods seem to value learning to stay at the center first, rather than first balancing the pole.

IV. CONCLUSIONS

For this project we have presented three different reinforcement learning methods that utilizes deep-learning to achieve the cart-pole swing up: The Deep Q-network (DQN), Deep Actor-Critic (DAC), and Random-Feature Actor-Critic (RFAC).

We have evaluated the methods and found that all approaches successfully accomplish the task, though, the rate of convergence is different for each method.

Furthermore, we compared the methods and discussed hyperparameter choices, as well as advantages and disadvantages that were mainly concerned with the rate of convergence or wear-out/damaging risks implied by the methods. There, we noticed that DQN learns the fastest but safer approaches may be provided by DAC or RFAC.

REFERENCES

[1] A. Jacot, B. imek, F. Spadaro, C. Hongler, and F. Gabriel. Implicit regularization of random feature models, 2020.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015.

[4] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever. Deep double descent: Where bigger models and more data hurt, 2019.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[6] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.