Implementation and Hyperparameter optimization of the Soft-Actor-Critic Algorithm

David A. Parham
DTU Electrical Engineering
Technical University of Denmark
Kgs. Lyngby, Denmark
s202385@student.dtu.dk

Lasse Starklit
DTU Compute
Technical University of Denmark
Kgs. Lyngby, Denmark
s165498@student.dtu.dk

Markus Böbel
DTU Compute
Technical University of Denmark
Kgs. Lyngby, Denmark
s201844@student.dtu.dk

Link to the Github Repository

https://github.com/davelbit/DTU-RL-Sample-Efficiency

I. Introduction

Deep learning represents a vital importance in many areas of today's technological progress. The concept of deep learning is "learning by doing" which is typically related to how the human progresses in life. The algorithms are inspired by the human brain, why typically referred to as artificial neural networks, perform tasks repeatedly, each time adjusting the decision making factors to improve the performance.

An important field within deep learning is reinforcement learning, which is the training of machine learning models to make a sequence of decisions in a specific environment. The model is trained by exploring the environment by trying actions in a given state and evaluating the outcome of such. Reinforcement learning is an important study within robotics, where the ultimate goal is to equip the robots with learning, improving, and adapting abilities to simulate the human behavior and even solve challenges that are beyond the ability of humans.

In reinforcement learning, there are 2 main kind of policies that algorithms learn from. 'On-policy' learning algorithms, which is seen as some of the most successful reinforcement learning algorithms in recent years (TRPO, PPO, A3C) [1], suffer from a sample efficiency, due to the fact, that they need new samples after each update. This issue is tackled introducing a so-called replay buffer, that contains previous experienced samples. In this way the 'off-policy' learning algorithms, such as the Deep Deterministic Policy Gradient (DDPG) or Twin delayed Deep Deterministic Policy Gradient (TD3PG) algorithms, are able to learn efficiently from the past, more related to the human behavior. However, this kind of learning makes these algorithms suffer from high hyperparameter sensitivity. Thus it requires a lot hyperparameter tuning in a new environment, to make the algorithms' performance stable and converging.

A state-of-art 'off-learning' algorithm, Soft Actor Critic (SAC), tackles the issue of convergence. However, this algorithm is typically hard to implement.

In this project, we implemented a SAC algorithm, which is found on Github[1], performed hyperparameter tuning, and tested the performance in various environments. In general we ended up with a descent performing algorithm able to learn and even succeed a number of tasks. The rest of the paper is structured as follows. In Section II we introduce related work "off-policy" learning algorithms. In Section III we describe the theory behind the SAC and our implementation of such. In Section IV we present our experiments and results. Section V and VI present the discussion and conclusion.

II. Related Work

A. Sample efficiency

The area of sample efficiency is an active research topic in the field of reinforcement learning, especially in continuous action space environments. It addresses the problem of training machine learning models for complex control tasks more efficiently.

In 2015, Lillicrap et al. [2] presented their model-free approach to this problem by introducing Deep Deterministic Policy Gradient (DDPG). Which is based on the Deterministic Policy Gradient (DPG) [3] and Deep Q-Network (DQN) [4] algorithms. DDPG is an actor-critic technique and consists of two models - actor and critic. The actor is the policy that takes the state as input and outputs the exact action instead of a probability distribution over actions. The critic is a Q-value network that takes the states and actions concatenated together as input and returns Q-values as output. DDPG differs from DPG essentially in two ways. Firstly, it uses two target networks to stabilise the training process and secondly, it uses a replay buffer to not only learn from recent experiences but also past experiences and therefore making it more sample efficient. Another approach that utilises a replay buffer is the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm developed by researchers at McGill University in Montreal, Canada [5]. This method is an extension of DDPG.

The number 3 in the abbreviation indicates the number of improvements the authors of the paper propose to address the major flaw that DDPG has, namely that the Q-function tends to dramatically overestimate the Q-values, known as

[1] https://github.com/davelbit/DTU-RL-Sample-Efficiency

function approximation error, which then leads to the policy breaking because it exploits the errors in the Q-function. TD3 uses two value networks and reduces the likelihood of the policy agent exploiting the critic by dismissing the clipping of expected values and instead taking the smallest of the target values. Also, updating the policy less frequently, thus using the concept of "delayed policy update", allows for better value estimation and helps to prevent actor fooling. And finally, adding noise to the target action makes it more difficult for the policy to exploit Q-function errors by smoothing out Q along with any action changes.

Another popular algorithm and also the one used for this report is SAC. This is explained in more detail in the next chapter along with the advantages over the methods already mentioned.

III. Soft Actor Critic Algorithm

As briefly described in I, "off-policy" learners have the advantages of sample efficiency. However, they also suffer from convergence brittleness due to the ability to reuse old samples. This allow the model to get stuck in a local optimum by repeatedly selecting actions that leads to short-term highest reward. The new and complete model free state-of-the-art reinforcement algorithm, soft actor critic (SAC),developed by Haarnoja et al. [6] tackles the issue of convergence brittleness by introducing a trade-off between the algorithm's ability to explore and exploit. In general reinforcement algorithms seeks to maximize the expected reward. However, as the SAC wants to take advantage of randomness to increase the exploration by adding the entropy of the action space to the reward. Thus the SAC does not only seek to maximize the expected return but also the expected entropy- meaning it seeks to find the best solution in terms of reward with the highest randomness.

The SAC's objectives can be written down as:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha H(\pi(\cdot|s_t))] \quad (1)$$

It is clear that the objective function, does not only depend on the reward $r(s_t, a_t)$ but also the entropy $H(\pi(\cdot|s_t)$ which is regularized by some factor α, also known as the trade-off coefficient. A higher value of α, obviously encourage to more exploration, while a lower value encourage to more exploitation. This coefficient α is treated differently depending on the implementation of the algorithm. Some suggests to have a fixed value of α, while others treats α as a constraint by varying it over the during to make the policy continuously explore states where the optimal action is uncertain.

In reinforcement learning, the algorithm generally consists of 3 functions:

- The state-value function V, which is the expected return from starting from a state, following a policy. The value function is typically given as:

$$V^\pi(s) = \mathbb{E}[\sum_{t=0}^{T} \gamma^t (R(s_t, a_t, s_{t+1}))|s_0 = s] \quad (2)$$

- The action-value function Q, computes the maximum expected rewards for an action in a given state. It has it names from the Q learning algorithm, which is the used to learn the quality of an action, guiding the agent to perform what action in the given circumstance. The Q function, in a regular reinforcement learning algorithm, is typically given as:

$$Q^\pi(s,a) = \mathbb{E}[\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1})|s_0 = s, a_0 = a] \quad (3)$$

- The policy function, that defines the action related to a state. It seeks to get the maximum accumulated reward.

As the SAC algorithm also considers the entropy in the output, equation 2 and 3 are both expanded with the second term of entropy:

$$V^\pi(s) = \mathbb{E}[\sum_{t=0}^{T} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t)))|s_0 = s] \quad (4)$$

and

$$\begin{aligned} Q^\pi(s,a) = \mathbb{E}[&\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1}) \\ &+ \alpha \sum_{t=1}^{T} \gamma^t H(\pi(\cdot|s_t))|s_0 = s, a_0 = a] \end{aligned} \quad (5)$$

It is seen from these equations, that they are connected. Older versions of the SAC suggests to have all 3 functions separately, new discoveries tend to the neglect the value function, V, due to the Q- and V functions' relation to the policy. In this project, we decided to follow the new discoveries, and the Q function is implemented using the Bellman equation including the value function:

$$Q^\pi(s,a) = \mathbb{E}[R(s_t, a_t, s') + \gamma^t V^\pi(s')] \quad (6)$$

.

SAC learns from the policy and two Q functions. More specifically the Q functions of the algorithm are learned using the following Q-loss function:

$$L(\phi_i, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathbb{E}}[(Q_{\phi_i}(s,a) - y(r,s',d))^2] \quad (7)$$

. In this equation, the Q function which was derived in equation 6 is used. However, using the definition of entropy and the fact that the right hand side is an expectation and thus can use samples, it can be rewritten as:

$$Q^\pi(s,a) \approx r + \gamma(Q^\pi(s', \widetilde{a}') - \alpha \; log \; \pi(\widetilde{a}'|s')) \quad (8)$$

. where $\widetilde{a}'$ is the action from a fresh sample and r and s' should come from the replay buffer. The second term of the loss function 7 is the target, which is given as:

$$y(r,s',d) = r + \gamma(1-d)(Q_{targ}(s', \widetilde{a}') - \alpha log \pi_\theta(a'|s')) \quad (9)$$

where the Q_targ refers to the minimum results of the two Q-functions. and is a combination of a number of key features:

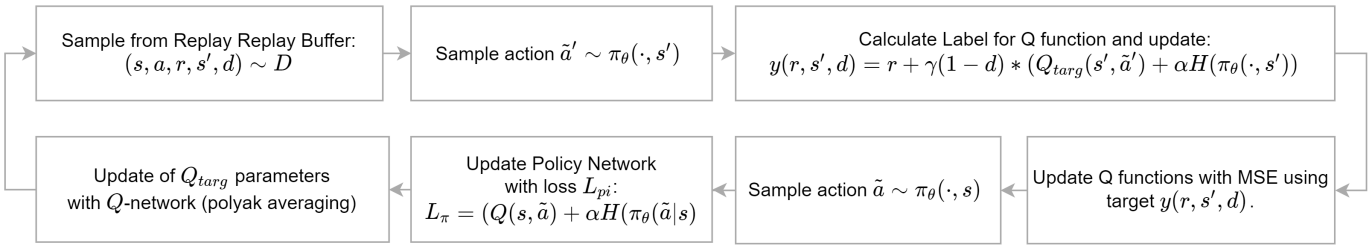

Fig. 1. Training process of the SAC

- Each Q-function learns by regressing to a single target using MSBE minimization. The target is computed by target Q-networks, that are obtained by a moving average between Q and the respective target which is also called polyak averages during training. The use of Q-targets is to avoid cyclic dependencies.
- To find the shared target for the algorithm, equation 9 uses the clipped double-Q trick, which is taking the minimum of the 2 estimates.
- The last term of the function includes the entropy regularization.

The SAC trains a stochastic policy, which uses entropy regularization to explore in an "on-policy" manner. The policy is optimized by using the so called reparameterization trick, which is used to make sure sampling from the policy is differentiable.

Having introduced both the Q-and policy-learning, we can present a depiction of how the SAC algorithm runs in our implementation as in figure 1.

The figure shows an update process of the SAC. For each update step we sample a batch from the replay buffer D and use these samples to calculate the target for the Q-functions given in equation 9. This target represents the expected value (reward + entropy) of the new state. In the next step, the Q functions from equation 8 are updated using the calculated target. In the second last step, we do a gradient step on the policy network using the loss function shown in equation 7. As of the last step in the process we update the Q_{target} network with the use of polyak averaging.

IV. Experiments & Results

A. Gamma

Gamma, the discount factor, that quantivies the relative importance of the future, is a very important hyperparameter in the algorithm. The closer gamma is to zero the more short-termed the agent tends to be. In general, this means setting the gamma value to 1, the agent sees the importance as a future reward as high as the current reward. In contrary, if gamma is zero, the agent will prioritize immediate rewards, meaning it does not care about future rewards.

The experiments we did for this hyperparameter were conducted in 2 different environments. The Cartpole Balance and the Cartpole Swingup. The biggest difference of these environments is the future actions, since a high immediate

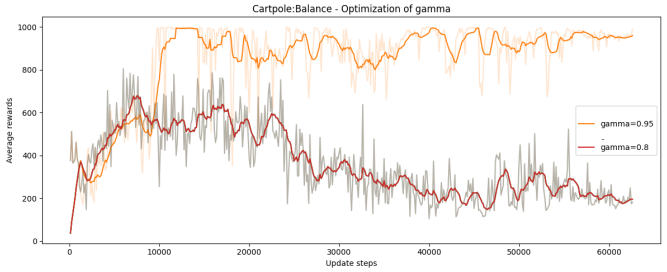

Fig. 2. Reward from edge cases of gamma values in the *Cartpole Balance* environment

reward in the Balance means the pole is stabilized, while a high immediate reward in the Swingup environment can be a result of a snapshot of the pole being at the top as the pole swings 360 degrees, why it is important to consider future rewards too. Our hypothesis is thus:

- Low gamma values will eventually result in a drop of performance, as the training does not go very long in the future.
- The drop of performance will be visible much faster in the Swingup environment, as the changes in the states/rewards for given states are much more critical.
- As the agent becomes more complex, the further it has to look into the future, we expect, the training time to increase as gamma increases

For the balance environment, it was interesting to look at edge cases, since such simple environment, cannot contribute a lot to the final decision on the optimal gamma. Looking at the graph in figure 2, it is clear that there is a large drop in the rewards after around 20.000 episodes for the low gamma value (0.8), while the high gamma value (0.95) have been able to foresee a lot more in the future, making the performance more stable.

As seen from graphs from the Swing-up environment in figure 3, a low gamma value (0.86 brown line) initially performs better short termed, but is very unstable having major drops in rewards (especially the drop at 20.000 steps like for balance environment, seen in figure 2. The higher gamma value, which seems to be the best performing value (0.9 red line) increases slower, but does not suffer major performance drops. It is also see, at the end of the cycle, that higher gamma value eventually outperforms the lower gamma value, why we

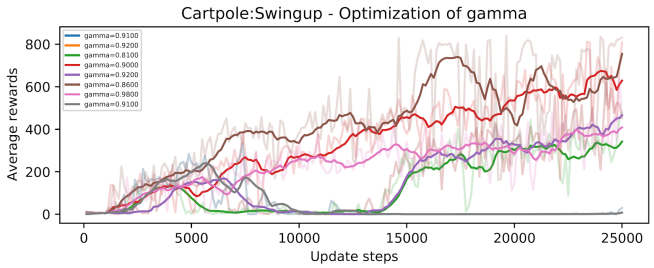

Fig. 3. Reward from varying gamma values in Swingup environment

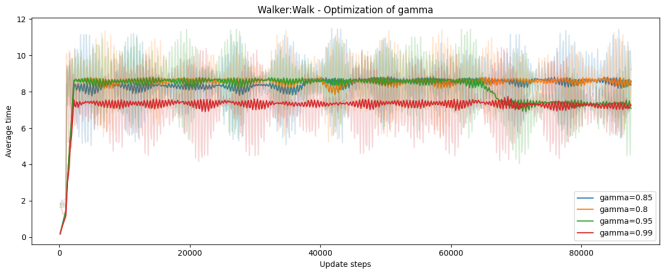

Fig. 4. Training time on varying values of gamma

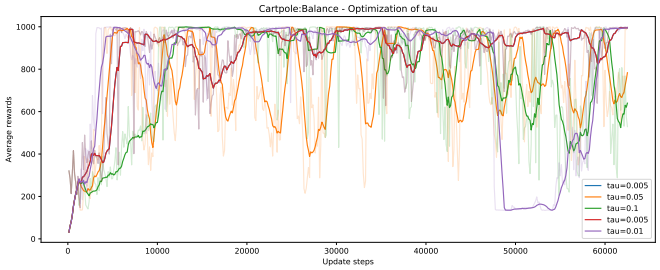

Fig. 5. Rewards within Cartpole Balance with different values for τ

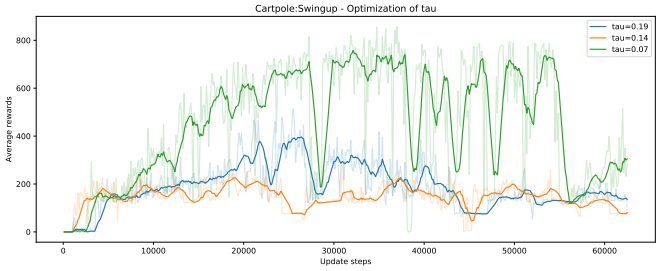

Fig. 6. Rewards within *Cartpole Swingup* with different Values for τ

conclude the most optimal gamma value in this environment.

For our third hypothesis, we expected the training time pr step to increase when increasing Gamma, however as seen in figure 4, we could not proof this, as it seems like the training time reduces as the gamma is increased.

B. Factor for Polyak Averaging τ

As described in section III our implementation of the SAC algorithm makes use of two target and two Q-networks to overcome cyclic dependency between the networks. We update the weights of the target network by applying polyak averaging. Hence, in each update state the weights of the target network $\theta_t arg$ equals:

$$\theta_{targ,i} = \tau * \phi_i + (1 - \tau)\phi_{targ,i} \tag{10}$$

based on the weighting τ which is in the interval $[0, 1]$.

In other words, the smaller the parameter τ is set, the faster the target network is updated. Hence, with higher higher values we get a lack between the Q network and the Target network, hence the rewards will be more unstable over time. Figure 5 shows the training of the *Cartpole Balance* environment. In the graph we can see that the values of 0.01 and 0.05 for τ make the agent to learn fastest. Also the smallest value, 0.005, shows a high convergence. Our hypothesis, that higher values will fluctuate more, seems to be shown in the data. So the value of 0.01 and 0.05 converge fast, however they also show high peaks within the average reward over training. The value of 0.005 seems to be more stable.

In the next heavy task, the *Cartpole Swingup*, we further can see that a high correlation between the τ value and the maximum reward is. Figure 6 shows that the two higher chosen values for τ hinder the agent to learn the ability to catch the ball. In this environment however, we see an opposite effect than to the *Cartpole Balance*: A lower value for the parameter, results in higher training fluctuations. However, this can be caused by the low overall reward, hence the variances are as well smaller.

All in all, the parameter τ is shown to be sensible to tune. However, smaller values are to be prefered, because the faster update through the agents networks allow a more stable training of the agent as seen in the *Cartboard Balance* experiment

C. Alpha and Alpha decay

As discussed in section III, the alpha value is what decides the trade-off between exploration and exploitation. The bigger alpha, the more the model tends to explore, while a smaller alpha value makes the model ”less curious” and thus choosing exploited actions.

The first experiment we conducted for the hyper-parameter alpha, is testing whether our theory given in III holds, that treating alpha as a constraint rather than a fixed value is preferable.

From the experiments with varying alpha seen in the first graph, we see that, introducing the automatic adjusting alpha, which are seen as $init\_alpha$, the reward is not as good in the beginning as for the fixed alpha, since the entropy (exploration) is dominant. Especially the higher value of the varying alpha (orange line, 0.074), increases at a very slow pace. But as the alpha for the orange alpha (init_alpha=0.074) reaches its peak, seen after 16.000 steps in figure 8 it starts converging towards 0, meaning there is now a clear distinction between a good and bad action, the exploitation factor becomes dominant

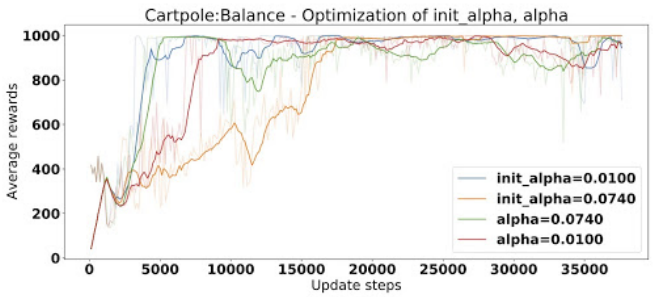

Fig. 7. Performance of different values of alpha

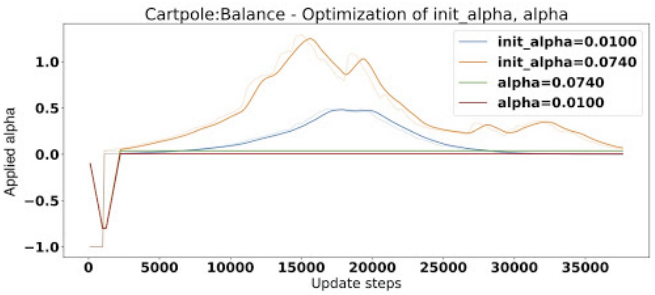

Fig. 8. Comparison of alpha values as a fixed value and a constraint

and we see a rapid rise for that model. Future performance is much stable performance for the well-explored model.

As shown in these experiments, the tuning of the alpha parameter is highly sensitive to the environment. Especially in environments in which the needed exploration differs through time. One example of this is the *Cartpole Swingup* experiment. While in the *Cartpole Balance* environment the needed actions are similar throughout the experiment, the *Cartpole Swingup* combines the balancing act with the initial swingup at the beginning. To overcome this issue [7] propose an automatic adjustment of the used entropy. To do so the authors use the dual optimization problem with the minimum entropy as tight restriction. As a result we get a minimization problem on the function:

$$\alpha_t^* = arg\ min\ \mathbb{E}_{a_t^* \sim \pi_t^*} \left[-\alpha_t log \pi_t^*(a_t|s_t;\alpha_t) - \alpha_t \mathcal{H} \right] \quad (11)$$

In every update step of the networks, the authors then recursively update the applied alpha value using gradient descent. With the automatic update of the alpha values, we get a new, yet not as strict hyper-parameter, the initial value of the alpha, which will be referred to as `init_alpha`. Figure 7 shows the comparison between two fixed alphas as well as two start alpha values for the automatic optimisation. We see that, as expected that the choice of the initial alpha is crucial for the time of convergence. However, on the *Cartpole Balance* environment, we can see similar performance between the `init_alpha` value of 0.01 and a fixed alpha of 0.074. However, the fixed alpha seems to be more stable. Figure 8 shows the development of the alpha values. Here, we see that in the middle of the training period, the alpha values are rising, the agent is exploring more. In the initial alpha value of 0.074, we observe a higher applied alpha value in the middle of the training, this causes the model to explore more and therefore also except reduced reward through this episode. When we tried to use a static alpha on more complex problems like the *Cartpole Swingup* we could not find suitable parameters which allowed the agent to solve the experiment.

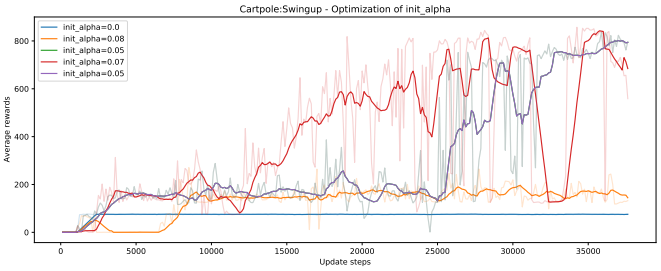

Fig. 9. Average episode rewards for the Swingup environment for different $init\_values$.

To further check our hypothesis we will try different values for the initial alpha to see its dependency on the environment. We expect the reward to converge with different times. Figure 9 shows the average awards for different values of the initial alpha. We see that the values for 0.0 do not converge at all. This makes sense, because the model is fully exploiting the

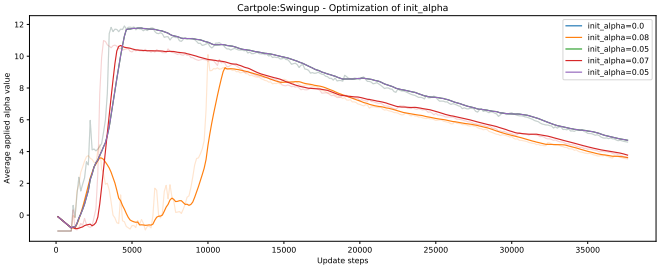

Fig. 10. Development of the applied values for α based on the initial value.

seen episodes from the begin on. We can see the performance of this run as a baseline to see if the models learn. A value of 0.08 for the `initial_alpha`, the maximal value we chose, seems to not learn either. However, the close value of 0.07 shows the fastest learning of the problem, however, with fluctuations in the learning. A value of 0.05 increases the time of the learning. Nonetheless, here the agent also are able to solve the problem. Figure 10 shows the development of the applied alpha based on the initial value. We see a clear trend that the applied alpha is reaching for its maximum and then slowly decrease. This means the agent tries to explore first and slowly start using its experience. Only the 0.08 value shows an different development, which can be the reason for its lack of performance: The agent directly used its experience before starting to the exploration.

All in all, is seen that usage of the automated alpha value allows the agent to faster learn the ability to learn harder

experiments. However the choice of the initial value for the alpha is still a challenge of the hyperparameter tuning.

D. RL-Networks

Furthermore, we also conducted some network-specific experiments and worked with the network sizes, different initialisers and learning rates. In the beginning, we had the hypothesis that the performance of our agent rises when we increase the size of our networks. However, what we discovered was that the performance decreases proportionally to the number of linear layers added. We suspect that poor performance is not strictly correlated with the network size, but with the increasing number of parameters that need to be adjusted accordingly to achieve similar or even better results. Besides, we found that performance also strongly depends on the complexity of the task. For simpler control tasks such as ”balance”, a shallower network size, such as the original SAC network depth, was far superior to deeper networks, whereas the performance gap between shallow and deep networks was much smaller for a complex task such as ”swingup”, as shown in Fig. 11

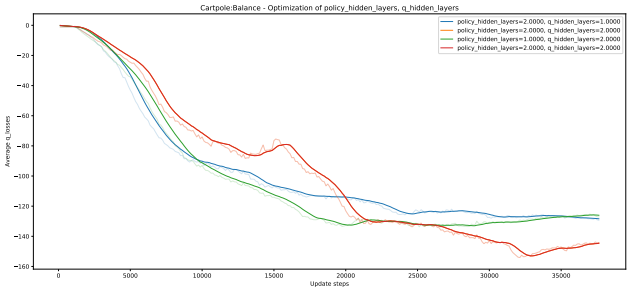

Fig. 11. The performance gap between shallow and deep networks is smaller for complex control tasks, such as swingup.

Changing the weight and bias initialisation from a normal to an orthogonal initialisation increased the performance dramatically. The problem with normal initialisation is that the variance increases with the number of inputs, which is impractical in an environment with potentially infinite inputs [8]. Therefore, we used orthogonal initialisation to combat the vanishing gradient behaviour. Due to its property that all eigenvalues of an orthogonal matrix have the absolute value of 1, regardless of how many inputs we feed into our network [9].

Moreover, we have tried to optimise the time it takes our models to find local minima and thus converge faster by experimenting with the learning rates, but unfortunately our plots do not show that this is the case. However, we were also interested in seeing to what extent the learning rate affects the performance of the network.

Fig. 13 shows that subtle changes in the value of the learning rates can have a high impact on the performance of the RL agent. We discovered that the learning rate values:

- lr_actor: 0.002
- lr_critic: 0.006

achieve the best results.

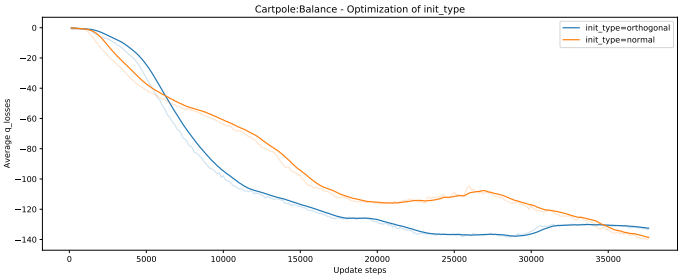

Fig. 12. The losses of the two initialisations show that the orthogonal init (blue) performs better.

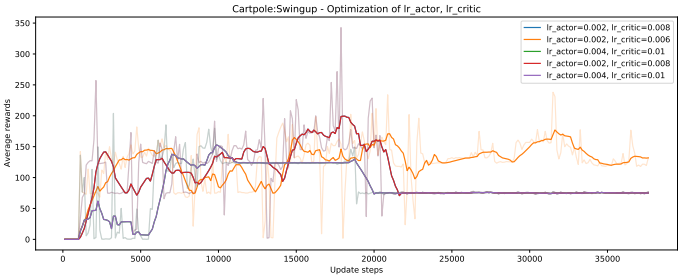

Fig. 13. Small changes in the value of the learning rate have a

V. Discussion

The main objective of this project task was the implementation of the Soft-Actor-Critic algorithm. We saw through the conducted experiments above, that the networks of the implementation are learning in the expected way. We further observe that tasks like the *Cartpole Balance*, *Swingup* or *Catch* task of the *Ball in Cup* with sparse rewards, can be achieved by the resulting RL agent. However, due to the complexity of the underlying algorithm, it cannot be ruled out that the implementation has minor bugs, which causes slower learning.

To test our implementation on the environment task of the *Walker Walk*, we observed an exploding of the actor as well as the critic losses. In our analysis of the problem, it seems that the dynamic generated α values are starting to increase exponential, which causes the loss explosions. However, we were not able to find the root cause of this effect and cannot rule out possible bugs. False hyper-parameters is another cause for this behaviour.

As described before, we are facing a vast variety of sensible hyper-parameters. We used [6], [7], [10], [11], and [12] as well as inputs of our tutor for the specification of our initial hyper-parameters. When we did the tests on the different parameter combinations, we only changed one parameter at a time. Hence, there might be more combinations which can lead to higher results.

To make the results reproducible, we used a unified seed to train the model. While this gives deterministic behaviour of the networks' training, it can show effects which are not necessarily generalize-able. From this follows, that the shown rewards in the tests might differ when using different seeds.

VI. Conclusion and Future Work

Within this project we implemented the SAC algorithm from scratch. The resulting realisation of the agent was able to solve the problems for the *Cartboard Balance*, *Swingup* and the *Catch* task of the *Ball in Cup* domain. However, the implementation was not able to achieve great rewards in the *Walker Walk* task.

As a second task, we did advanced hyper-parameter tuning on the algorithm. Here, we found out that the higher hyper-parameter γ, the weight for future rewards, the more stable the algorithm is following. We further could show that the automatic alpha value outperforms, static temperature in difficult experiments. However, the initial value for the automated alpha still requires sensitive tuning. The weight for updating the target networks τ also showed high sensibility. However, smaller values which causes a faster update of the target value seem to be preferred.

In terms of the architecture of the agent's networks, we could show that the size of the hidden layers correlate with the difficulty of the task. We further showed that orthogonal initialization outperforms a uniform one. At last, we observed that the learning rates of the networks are highly sensitive parameter to tune.

Subject of following project can be the interdependence of certain hyper-parameter. Furthermore, advances experiments based on the results of this project can be done to provide specific guidelines and methodologies on how to tune the SAC algorithm more efficient. At last, comparisons to other RL algorithms can be done to see the change in hyper-parameter sensitivity between the algorithms.

References

[1] V. V.Kumar. soft actor critic demystified. [Online]. Available: https://towardsdatascience.com/soft-actor-critic-demystified-b8427df61665

[2] T. P. Lillicrap, J. J. Hunt, A. P. N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015. [Online]. Available: https://arxiv.org/abs/1509.02971

[3] Silver, David, Lever, Guy, Heess, Nicolas, Degris, Thomas, Wierstra, Daan, Riedmiller, and Martin, "Deterministic policy gradient algorithms," 2014. [Online]. Available: http://proceedings.mlr.press/v32/silver14.pdf

[4] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, A. A, Veness, Joel, Bellemare, M. G, Graves, Alex, Riedmiller, Martin, Fidjeland, A. K, Ostrovski, and Georg, "Human-level control through deep reinforcement learning," 2015. [Online]. Available: https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf

[5] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2019. [Online]. Available: https://arxiv.org/pdf/1802.09477.pdf

[6] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.

[7] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," 2019.

[8] Weight initializations activation functions. [Online]. Available: https://www.deeplearningwizard.com/deep_learning/boosting_models_pytorch/weight_initialization_activation_functions/#why-do-we-need-weight-initializations-or-new-activation-functions

[9] S. Merity. (2016) Explaining and illustrating orthogonal initialization for recurrent neural networks. [Online]. Available: https://smerity.com/articles/2016/orthogonal_init.html

[10] D. Yarats. Github repository for pytorch_sac_ae. [Online]. Available: https://github.com/denisyarats/pytorch_sac_ae

[11] seungeunrho. Github repository for minimalrl. [Online]. Available: https://github.com/seungeunrho/minimalRL

[12] OpenAi. Github repository for spinning up in rl. [Online]. Available: https://github.com/openai/spinningup