README.md

Challenge 2

Authors: Jannis Weil, Johannes Czech, Fabian Otto

Least-Squares Policy Iteration (LSPI)

In this section, we describe our experience with implementing LSPI based on the paper from Lagoudakis & Parr.

Our main focus for the implementation is the environment `CartpoleStabShort-v0` from the Quanser platform.

Implementation and observations

The implementation can be found in the python module `Challenge_2.LSPI`.

We implemented LSPI as offline algorithm and do not sample any new samples during the optimization. We found for the `CartpoleStabShort-v0` environment, random actions can cover the state space reasonably well and adding new samples do not increase the performance. Additionally, we always use all 25,000 samples during our update. Utilizing batches of smaller sizes (128, 512, 1024, 2048) always yielded worse results.

Discretization of actions

For the discretization of actions we decided to use [-5, 0, +5], this allows the cartpole to maintain its upright position by choosing action [0]. Removing [0] from the possible actions highly reduced the performance for us. We assume this happens due to the need to select going to the left and right, which are both equally good/bad at the fully upright position, i.e. the algorithm is harming its own performance as both action lead to worst states.

Feature functions

Finding a good feature function is the key challenge of LSPI. The algorithm itself does not require a lot of complex computation, the main part can be found in the LSTDQ-Model (see below). Therefor, finding a good feature function decides over success and failure. We implemented RBF features as well as Fourier Features (both can be found in `Challenge_2.LSPI.BasisFunctions`). During our tests, the Fourier features worked significantly better and we were not able to learn a consistent policy with RBF features. One reason for this is, in our opinion, the large hyperparameter space for RBFs. It is necessary to tune the RBF centers as well as the length scales. We tested two types similar types of Fourier features. The first implementation

$$f(\mathbf{x}) \equiv \sqrt{\frac{D}{2}}[\cos(\omega_1^T\mathbf{x} + b_1), \dots, \cos(\omega_D^T\mathbf{x} + b_D)]$$

performed in our experience worse than the second

$$f(\mathbf{x}) \equiv \left[\sin\left(\frac{\omega_1^T\mathbf{x}}{v} + \phi^{(1)}\right), \dots, \sin\left(\frac{\omega_D^T\mathbf{x}}{v} + \phi^{(D)}\right)\right]$$

with $\omega \sim \mathcal{N}(0, 1); b \sim U[0, 2\pi); \phi \sim U[-\pi, \pi)$

Fourier features have the advantage that they approximate the RBF kernel as described in the above papers while also limiting the need for a lot of hyperparameter tuning. Besides the amount of features D and the band width v in the second version, the hyperparameters are "fixed". Further, we found that combining the second fourier features with min-max normalization (`Challenge2.Common.MinMaxScaler`) was improving the results significantly from approximately 500 reward to 10,000 reward for the `CartpoleStabShort-v0` environment. In order to normalize $\dot x$ and $\dot \theta$, which have infinte state boundaries, we selected empirically choosen max and min values (based on samples), [-4,4] for $\dot x$ and [-20,20] for $\dot \theta$. Even though we observed slightly lower $\dot \theta$ in the samples, increasing the range helped, we assume some extreme cases were simply not covered by the random initial actions.

LSTDQ-Model $(D,\ k,\ \phi,\ \gamma,\ \pi,\ \mathcal{P},\ R)$ // Learns $\widehat{Q}^{\pi}$ from samples

// D : Source of samples (s, a)
// k : Number of basis functions
// ϕ : Basis functions
// γ : Discount factor
// π : Policy whose value function is sought
// $\mathcal{P}$: Transition model
// R : Reward function

$\widetilde{\mathbf{A}} \leftarrow 0$ // $(k \times k)$ matrix
$\widetilde{b} \leftarrow 0$ // $(k \times 1)$ vector

for each $(s, a) \in D$

$$\widetilde{\mathbf{A}} \leftarrow \widetilde{\mathbf{A}} + \phi(s,a)\Big(\phi(s,a) - \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s')\phi\big(s', \pi(s')\big)\Big)^{\mathsf{T}}$$

$$\widetilde{b} \leftarrow \widetilde{b} + \phi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') R(s,a,s')$$

$\widetilde{w}^{\pi} \leftarrow \widetilde{\mathbf{A}}^{-1}\widetilde{b}$

return $\widetilde{w}^{\pi}$

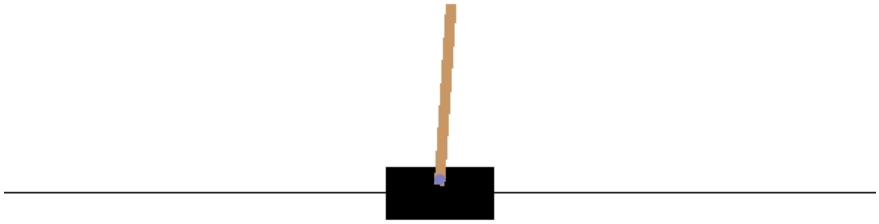
Issues

As mentioned above finding an appropriate feature function was the hardest part. The final result we found was, honestly, slightly lucky. Implementing the LSPI itself was straightforward and as long as we used the normal LSTDQ-model, matrix computations were possible. However, the optimized LSTDQ version was not fast. Even though the optimized version avoids computing the inverse of A, it depends on the approximate inverse of A, which is computed iteratively from the previous sample and therefore makes it necessary to use loops in the computation. Consequently, the higher performance matrix computations in C cannot be used. Additionally, as before mentioned, normalization played a key role for good results, without it we often experienced that LSPI is not converging. On big remaining issue is that our policy cannot be exactly reproduced with a different seed, as a change of the seed does not only change the samples but also the ω and ϕ parameters of the fourier features. However, we get more stable results over multiple seeds when we are using more training samples.

Results

Using the above setting we achieve a reward of 19,999.95 over 10,000 steps and 25 different seeds for the test run.

The following animation visualizes the learning process (episodes are shortened). One can see, that the agent is able to stabilize the pole better for each policy update and can balance it without moving in the end.



Deep Q-Learning (DQN)

In this section, we describe our experience with implementing DQN based on Deepmind's paper.

DQN was tested out on the `CartpoleSwingShort-v0` environment form the Quanser platform. Additionally, we created a model for the classic `Pendulum-v0` environment.

Implementation and observations

The implementation can be found in the python module `Challenge_2.DQN`.

Model type (experiments with different architectures)

As suggested in the paper, we use neural networks for our models. We tried using deep networks, but according to early experiments it seems like shallow network architectures work better in our case. We experimented with different networks using 1 to 3 hidden layers with a small amount of hidden notes between 15 and 128. We also tried different activation functions but setteled down with ReLU, as we were able to achieve very good results on the pendulum with it. Classical techniques applied in supervised such a Batch-Normalization-Layers lead to worse results.

Replay Memory and Exploration

Our replay memory stores each observed sample up to the specified capacity, then it starts to overwrite old samples.

We choose the actions during the online-learning process based on an epsilon-greedy policy. For the training process, we implemented an exponetially decreasing epsilon starting from 100% random actions and ending with 1% random actions. This encourages exploration at the beginning and sticks to the learned policy at the end. For the hyperparameters it appeared to be useful to use a rather large memory size (e.g. 1 million) and a high `minibatch_size` (e.g. 1024). This might be to avoid a high correlation between the training samples.

Stability

One major problem we encountered with DQN is the stability of the learned policy. We often saw quite good policies being directly followed by policies where the cart just drives to one of the borders of the track as fast as it can.

We came up with the following strategies to improve the stability of the learned model:

- Use more steps before updating the target Q model.
- Use actions with lower values. E.g. `[-5, +5]` instead of `[-24, +24]`. When using high values, the agent often learns a sucidal policy where it crashes into the wall very quickly.

- Use reward shaping (e.g. punishing when the agent comes close to the border). **As this is not allowed for the challenge, we disabled this featue for the submission** and did no further investigations. However, early experiments suggest that it is much easier to learn a good policy with reward shaping. This indicates that the environment `CartpoleSwingShort-v0` is designed suboptimally by enabling suicidal policies as a local optimum.
- We make use of gradient clipping = 1 in order to avoid numerical instabilities
- **Learning Rate Schedule**

  We tried different learning rate schedules like `StepLR` which reduces the learning rate by a given factor at each timestep, as well as `CosineAnnealingLR` which smoothly lowers the learning rate. Although these learning schedules are often beneficial in the supervised case we didn't notice any improvements when using them in this RL-problem.

Using all these techniques, we are still not able to achieve a totally stable policy for `CartpoleSwingShort-v0`, meaning that it does not change much in further training episodes (except for directly running into the wall, this policy is quite stable). However, we can still extract the policies from the learning process which performed well.

Notes

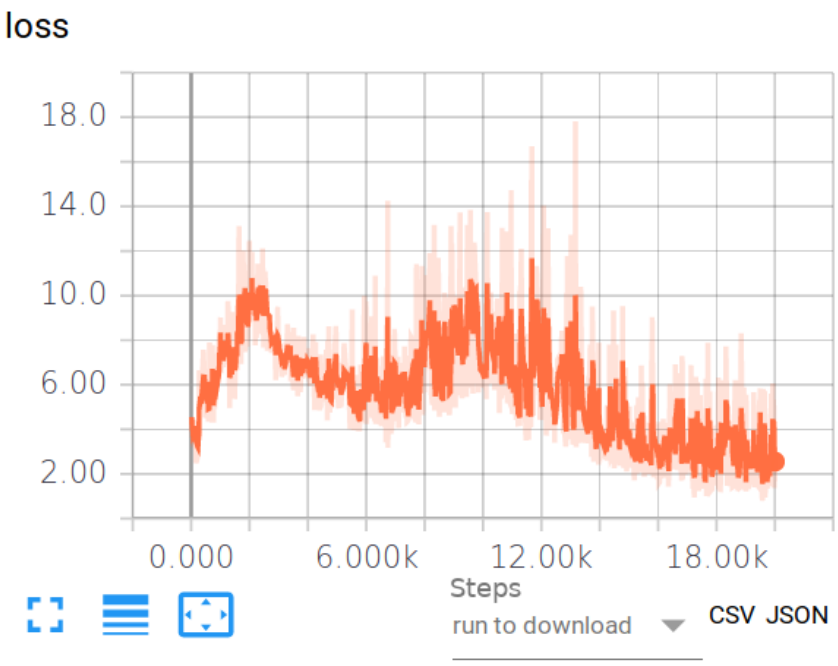
We used tensorboard logging for our metrics during training, but because `tensorboardX` is not part of the defined python environment we deactivated it in the final submission.

Results

Pendulum-v0

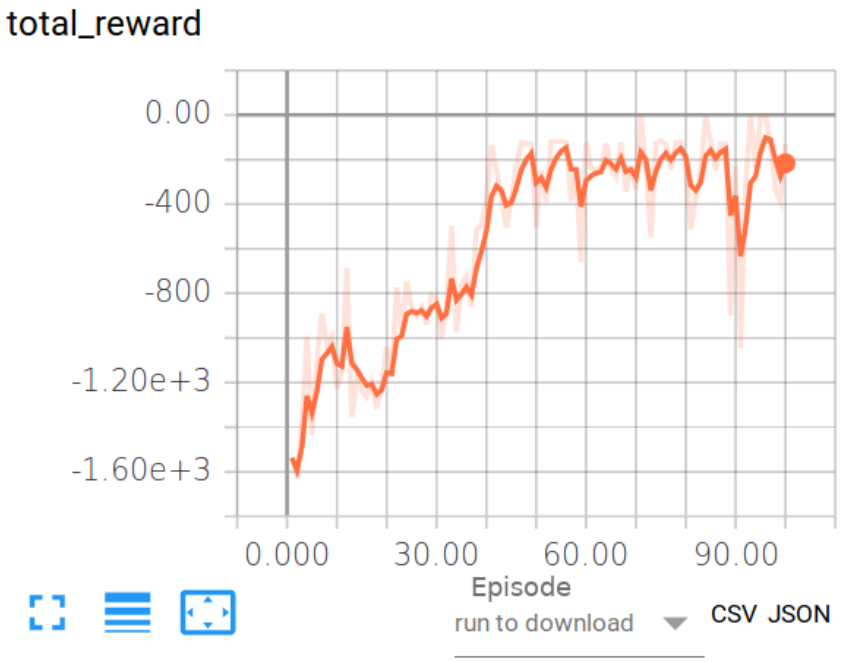
We tested the `Pendulum-v0` environment first to make sure that our implementation of DQN itself works. We were able to achieve a very good policy with an average reward of `-133.6028 +/- 71.4909` over 100 episodes after training for 40 episodes:

Loss development

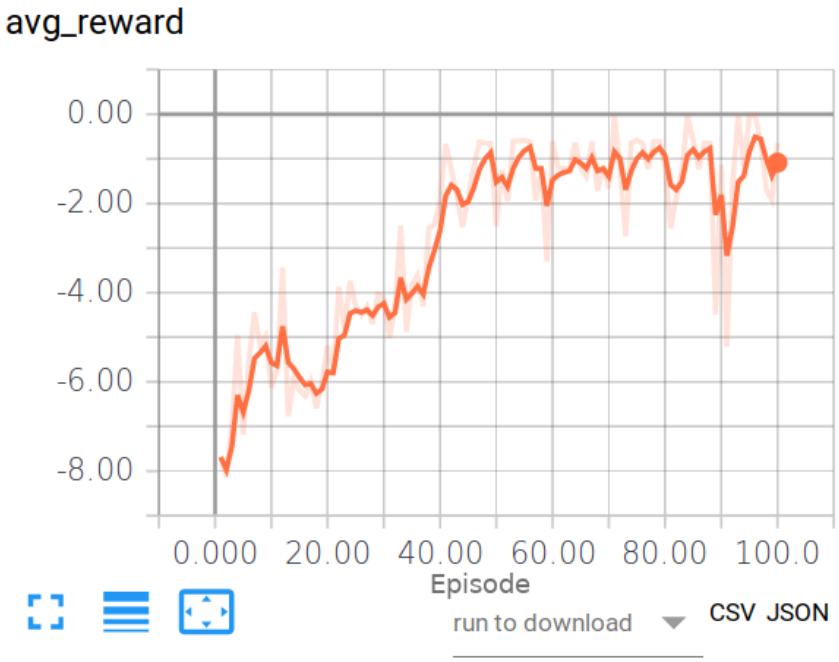


The loss development shows a general downward trend.

Total episode reward development

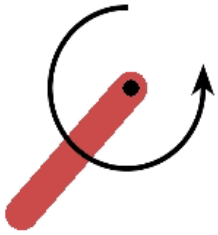


Average reward per step development



The development of the reward looks quite smooth and the shape of the plot is identical between average and total reward.

The following animation shows the final policy on some episodes.

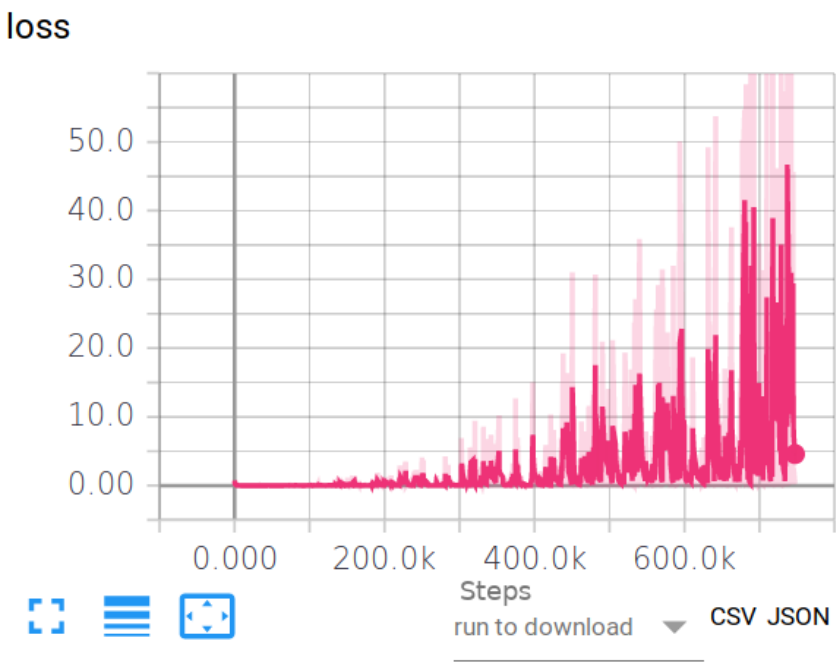


CartpoleSwingShort-v0

For the cartpole swingup, we achieve a "propeller policy" for which the cart stays inside the boundaries of the track and spins the pole in circles.

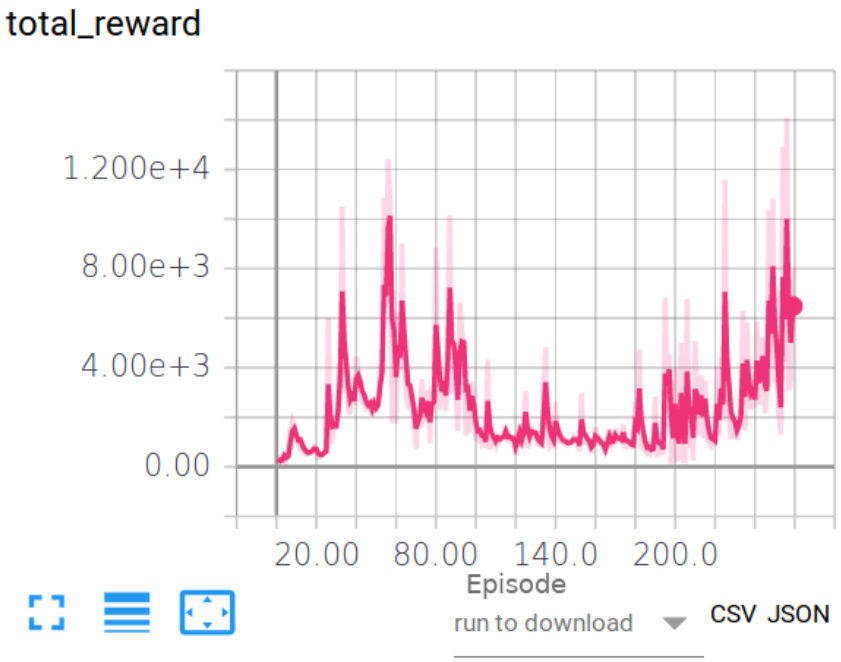
We achieve a total reward of `10385.86 +/- 264.80` evaluated over 100 episodes with a maximum episode length of 10k steps. The policy was obtained in `episode 56` during training.

Loss development



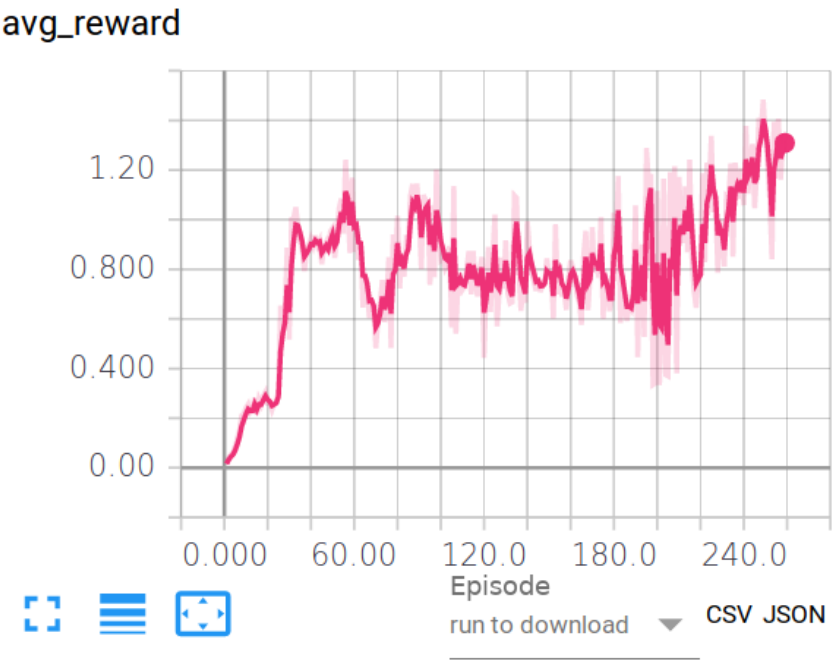
This loss shows an increasing variance towards the end.

Total episode reward development



As can be seen in the plot of the total reward, the learning is quite unstable mostly because the agent ends the episode too early. This is in our opinion a problem of the environment and a suboptimal formulation of the reward.

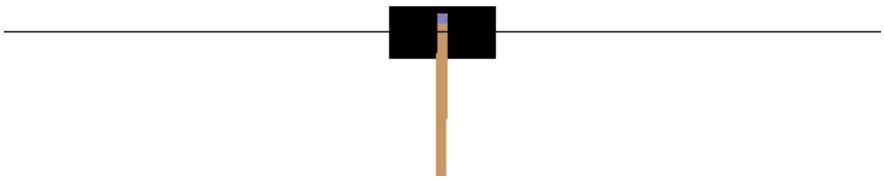
Average reward per step development



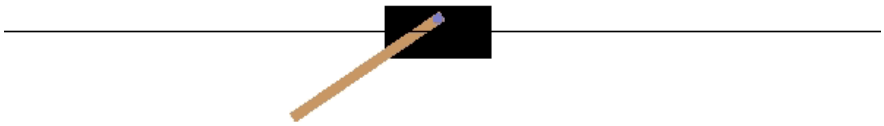
You can notice a general trend in increase of the average reward per step.

Visualization of progress during training

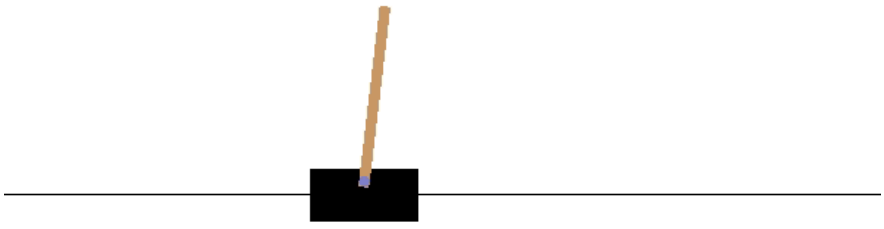
The following description refers to one episode where the agent acts according to the extracted policy. In the beginning, the agent tries to swing up the pole quite slowly:



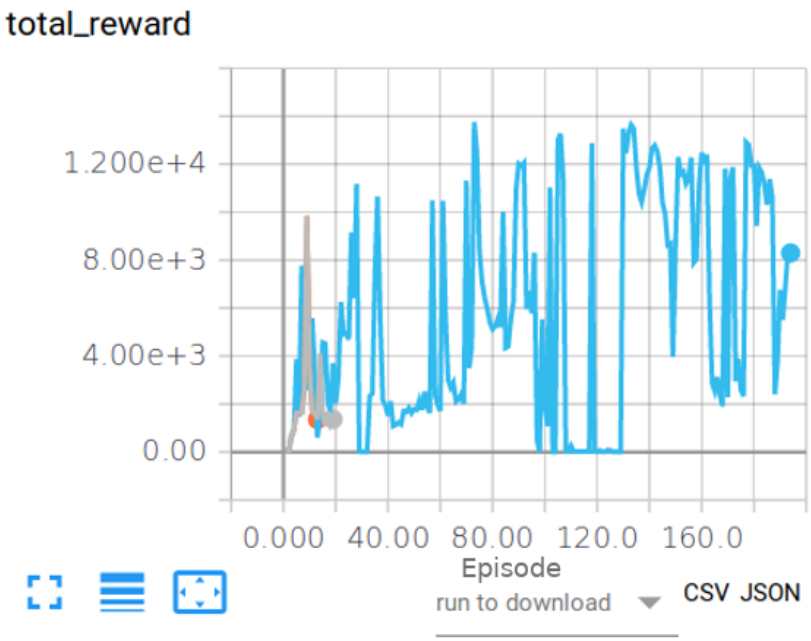
After about 3000 steps the agent tries to stabilize the pole:



But starts doing a propeller quickly after it failed:

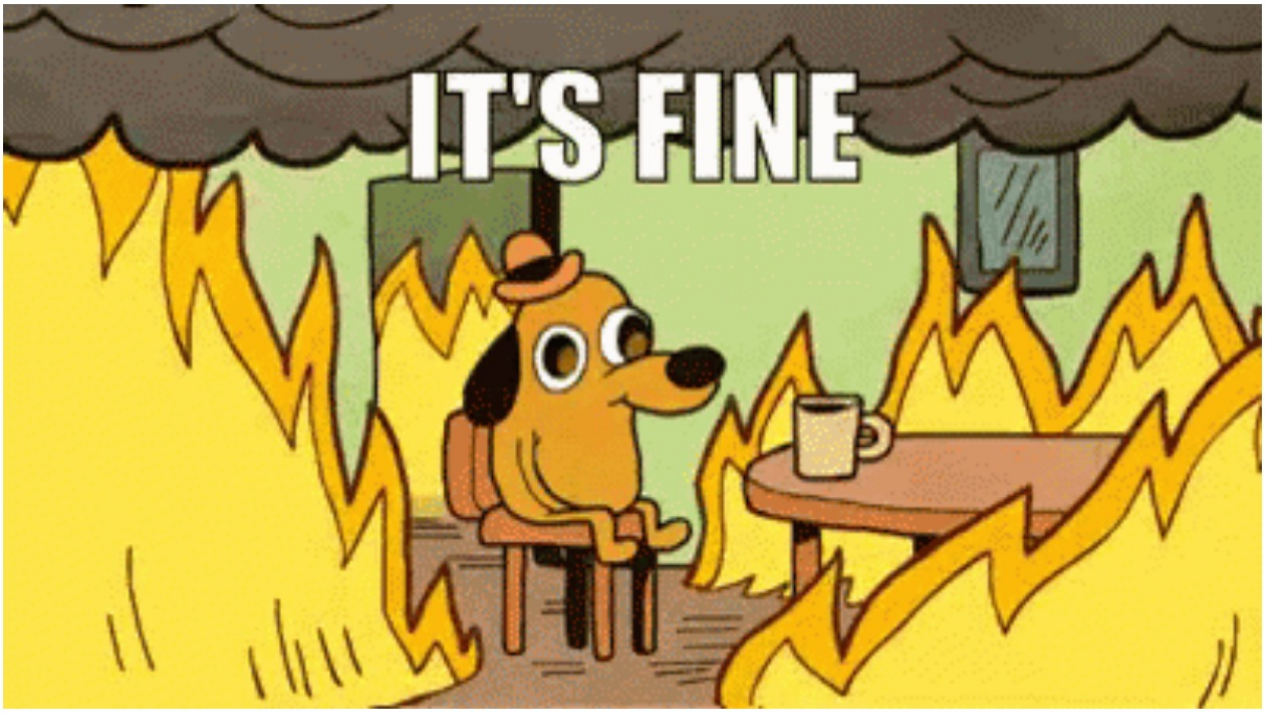


Total episode reward development by using reward shaping



With the use of our proposed reward shaping the agent is able to overcome bad policies that quickly crashes the wall and maintains a general total reward > 10k. These results suggest that one might be able to solve `CartpoleSwingShort-v0` with vanilla DQN easier by improving the reward formulation of the environment.

> I've taken to imagining deep RL as a demon that's deliberately misinterpreting your reward and actively searching for the laziest possible local optima. It's a bit ridiculous, but I've found it's actually a productive mindset to have.
>
> -- Irpan, Alex (https://www.alexirpan.com/2018/02/14/rl-hard.html)



RL researchers all the time