Intelligent Systems

Reinforcement Learning

REINFORCE & DDPG

Verfasser:
Ali Keysan [REINFORCE]
Anselm Paulus [DDPG]

Dozent:
G. Martius

1. September 2023

Inhaltsverzeichnis

1 Introduction

In this report we want to give an introduction and evaluation of two s.o.t.a. reinforcement learning algorithms, namely PPO and DDPG. Both algorithms have recently been able to solve difficult tasks such as playing DOTA [DOTA 5v5, 2018] and playing Atari games from raw pixel input [DDPG, 2016]. We will give a short description of the used methods and evaluate the algorithms on an easy gym environment as well as the more difficult laser-hockey environment. In this custom environment two players play the game of laser hockey against each other, trying to score a point by shooting the puck into the opponents goal while defending their own goal. The game is simulated to be played on an icy surface which means the players represented by filled-out half-circles are able to rotate which results in an increased complexity. The game can be played with continuous or discrete actions, while the discrete actions represent the max/min of the continuous action set. The state of the game is given by a set of coordinates and velocities. The first player to score a goal wins the game.

2 Algorithms

2.1 REINFORCE, PPO

Before I implemented a reinforcement algorithm, I started playing around with the laser-hockey environment. I got used to gym's functionalities and began to implement an own learning algorithm, where different approaches, like self-play, competitive self-play, playing against the game's own opponent were tried. But in the end the only thing that worked was imitation learning, where I let the basic opponent play against itself and fitted the corresponding data. In this way in merely one minute 1000 games could be played, because there is no neural network to be feeded forward to, and fitted in under a minute, which resulted an agent that played indistinguishable from the basic opponent. Because the network was only trained on the winning trajectories, the agent won two out of three games without counting the draws, resulting from early stopping. Thereupon the network was reduced as much as possible while still keeping the win ratio. At the end the smallest network consisted of one layer with nine nodes where the agent wasn't able anymore to mimic the basic opponent perfectly. It would be an interesting question if this result would be ever achievable by solely self-play of the small network, even with advanced match conditions like in OpenAI 5 or Alphastar.

When I realized it was time to began to implement a real reinforcement learning algorithm, I decided to try out PPO, since it achieved many outstanding results and best known because of DOTA 2 and it is said to strike a balance between ease of implementation and sample complexity [PPO, 2017]. After I glanced through a couple of PPO implementations, I was concerned of me understanding the code and decided on first implementing a typical policy gradient which supposedly needed minor changes to be a proximal policy optimization. I chose the REINFORCE variant because the CartPole-v0, LunarLander-v2 and laser-hockey environments are episodic and therefore well suited for Monte-Carlo learning and because PPO runs the policy for T timesteps and optimizes afterwards. The values for every timestep are calculated after each episode with length T and the discount factor γ.

$$V(s_t) = r_t + \gamma r_{t+1} + ... + \gamma^{T-t-1} r_{T-1} + \gamma^{T-t} r_T$$

To reduce the variance of the unbiased sample of $V(s_t)$ one can use a critic network to estimate the $V(s_t)$ and subtract it as a baseline $B(s_t)$ to get the advantage estimate.

$$\hat{A}_t = V(s_t) - B(s_t)$$

Finally one can calculate the objective function $L^{PG}(\theta)$ using the probability of our policy $\pi_\theta(a_t|s_t)$ and the advantage estimate.

$$L^{PG}(\theta) = \hat{E}_t[log\pi_\theta(a_t|s_t)\hat{A}_t]$$

Our goal is to maximize this objective which corresponds with maximizing the expected reward. PPO and the standard policy gradient have two main differences. One is that PPO doesn't update the policy every episode but rather collects data for N episodes of length T and performs multiple updates for K epochs. The other is the proposal of a novel objective function,

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

or the newer simpler version

$$L^{CLIP}(\theta) = \hat{E}_t\left[\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t, g(\epsilon, \hat{A}_t)\right)\right]$$

where ϵ is a hyperparameter and

$$g(\epsilon, \hat{A}_t) = \begin{cases} (1+\epsilon)A & \text{if } A \geq 0 \\ (1-\epsilon)A & \text{if } A < 0 \end{cases}$$

The main objective in this new objective function is to penalize large policy updates by putting an upper restriction with the ϵ, for which 0.2 turned out to be a good choice. The improvement compared to Trust Region Policy Optimization (TRPO), is that it needs less computing power for the constraint and is overall a more general approach. And well it did also outperform or was as good as most of the state-of-the-art algorithms like A2C, ACER and TRPO.

2.2 DDPG

To solve the given tasks I used a version of Deep Deterministic Policy Gradient (DDPG) [DDPG, 2016]. This is an off-policy actor-critic algorithm for continuous action-spaces using neural networks for both the actor and the critic to estimate a deterministic policy. The critic is a network estimating the q-value for a given state and action, it is simply updated by minimizing the TD(0)-error:

$$\begin{aligned} y &= r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\Theta^{Q'})) \\ L &= (y - Q(s_t, a_t|\Theta^Q))^2 \end{aligned}$$

I use batches for training but leave them out here to make the notation simpler. Q and μ describe the critic and actor networks, when they are followed by a ' we are referring to the target networks introduced later on. The actor, given a state, estimates the action to be taken. It is updated based on the Deterministic Policy Gradient Theorem derived by Silver et al., which states that

$$\nabla_{\Theta^\mu}\mu \approx \mathbb{E}_{\mu'}[\nabla_a Q(s, a|\Theta_k^Q)|_{s=s_t, a=\mu(s_t)}\nabla_{\Theta^\mu}\mu(s|\Theta^\mu)|_{s=s_t}]$$

holds for a deterministic target policy. This means that we can update the weights of our actor-network by simply multiplying the gradient of our critic network output w.r.t. the action with the gradient of our actor network output w.r.t. to its parameters.

Stochasticity is introduced by adding action noise to the deterministically calculated actions when acting. Following the original paper I use Ornstein-Uhlenbeck-Noise (OU) which describes the velocity of a massive brownian particle under friction. This is a particularly good choice because it favors exploration by generating a time-correlated noise. I multiply the action noise with an exponentially decaying parameter ϵ to have control over the exploration schedule.

Implementing the algorithm, we make use of several tricks to increase performance.

I use prioritized experience replay [PER, 2016], which is a version of experience replay [], in which the stored transitions are sampled for training based on a priotity system instead of uniform sampling. When stored, each new transition is assigned the highest priority, which is updated later on when the transition has been used for training. The priority update is performed based on the absolute TD-error generated in by the critic. I do this because we want to assign high priority to surprising transitions (both positive and negative), I don't really care about transitions that we can already predict with high accuracy. However, the non-uniform sampling leads to a difference between the sampled distribution of transitions and the real one, introducing a bias I get rid off by using importance-sampling (IS) weights, which compensate for the non-uniform probabilities. Therefore weighting transitions that are less likely to be sampled higher than transitions with high priority. We slowly increase the effect of IS over time because the closer we get to convergence, the more we need unbiased updates.

It turns out using the same network that we want to update to calculate our TD-target introduces a huge amount of variance, causing our networks to not learn or even diverge. We can tackle this problem by using target networks, slowly following our actual networks by mixing up the weights of the actual network and the target network with a mixing parameter (another approach would be to get the target network by copying the actual network every other step). We use target networks for both actor and critic.

3 Evaluation

3.1 REINFORCE

The REINFORCE algorithm was able to solve the CartPole-v0, Pendulum-v0 and LunarLander-v2 environments, whereas the result for the Pendulum case were a bit unexpected. The CartPole environment was expectedly solved in under 200 episodes, where in figure (1) it was trained without a critic and in figure (2) with a critic. As expected it learned faster with a critic. I also tried out performing multiple steps K of optimization for every N episodes and using the last $M \geq N$ episodes of data to train, like suggested in the PPO paper. As long as one didn't optimize as often as N the agent keeps stable and doesn't unlearn. But since this is a REINFORCE algorithm, the fastest learning was still achieved with one update every episode and $K = 3$ for the CartPole environment. For the other two environments I wasn't yet able to fine-tune the hyperparameters because it also takes quite a time to train the agent.

This is where the unexpected behaviour comes in. At first I thought my agent wasn't able to learn the Pendulum environment because most of the implementations on the internet were able to learn it with around $100 - 200$ episodes, which I compared with my results on the CartPole environment. But one has to say I didn't find an implementation on the internet which solved Pendulum using REINFORCE. When I finally let the agent train for a longer period of time it was able to learn it after 3000 episodes (figure (3)). One of my suggestions is that Pendulum is not an episodic environment and one therefore has to stop after a timestep T and because I discretized the one dimensional continuous action space into five equidistant actions I first lost the ability to perform arbitrary actions and also added more actions in comparison to the CartPole environment.

At first I didn't thought of testing the LunarLander environment, but since my algorithm was only able to solve CartPole and I saw an online case solving LunarLander with policy gradient, I wanted to give it a try. And since LunarLander is an episodic environment with dense reward

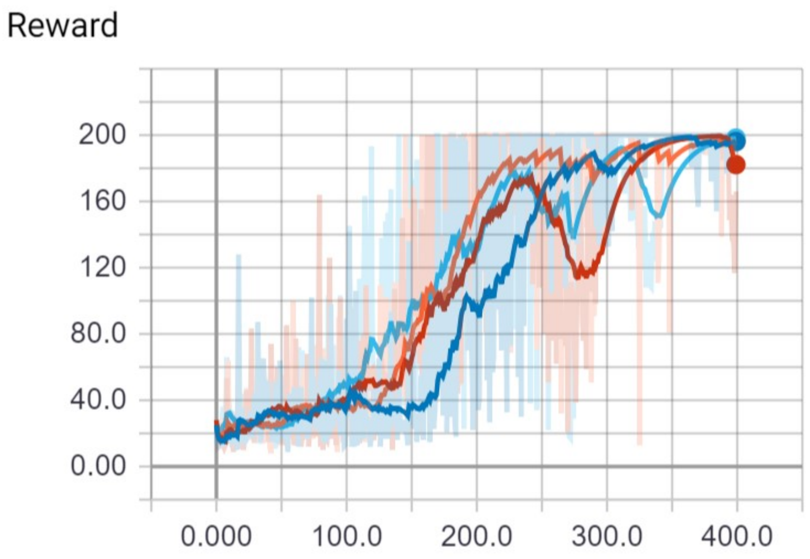

Abbildung 1: REINFORCE agent on CartPole-v0 environment, average reward over episodes without a critic baseline

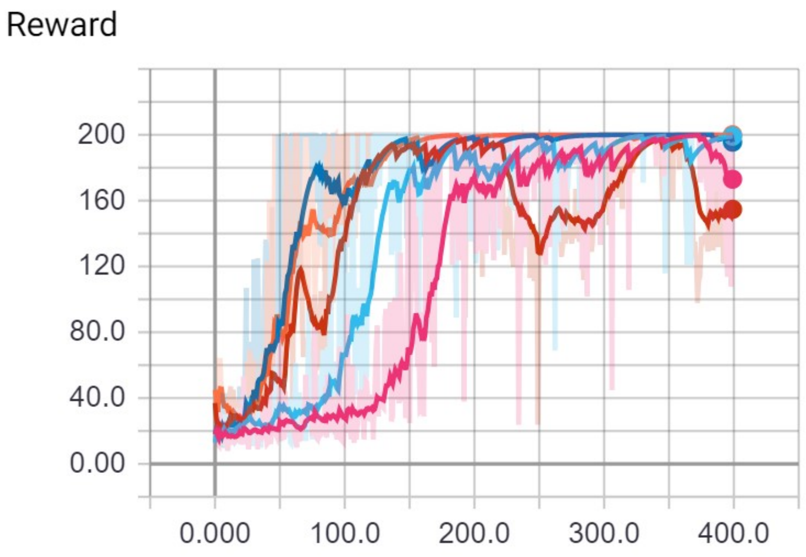

Abbildung 2: REINFORCE agent on CartPole-v0 environment, average reward over episodes with critic baseline

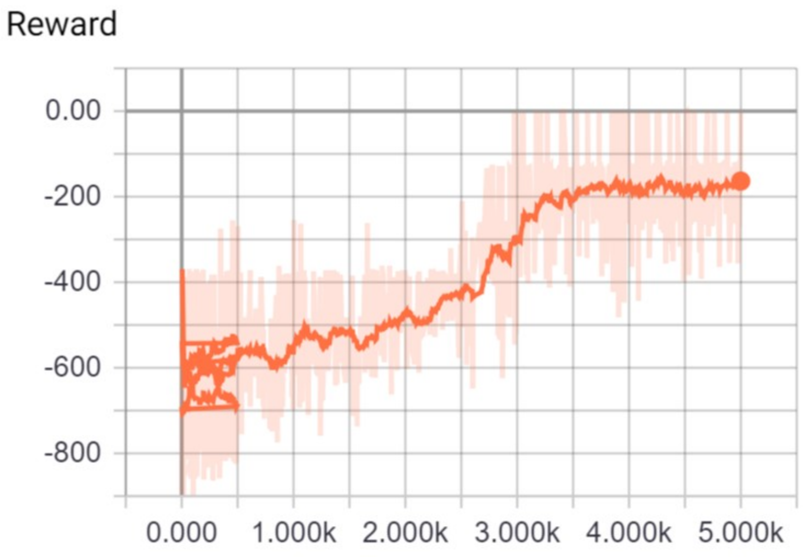

Abbildung 3: REINFORCE agent on Pendulum-v0 environment, average reward over episodes

and a big reward for success at the end, it might have a chance of solving it, which I already spoiled it, was actually the case after 3000 episodes as you can see in figure (4). After training and watching it play, one can clearly see that it learned to land quite fast, which encourages more reward.

Unfortunately the PPO implementation did not yield good results. It couldn't learn on the Pendulum and LunarLander environment and needed 5000 epochs to finally solve CartPole. I am currently not sure where the problem is, since there are only minor changes to a policy gradient implementation, which therefore the PPO implementation is also included in the same code, but I am looking forward to improve my implementation over the holidays and especially trying to solve the laser-hockey environment.

For every environment the hyperparameters stayed almost the same. The actor and critic learning rates were both 0.001 where the Adam optimizer was used and had both two hidden layers with 64 nodes. The last layer of the critic was linear and a softmax for the agents action probabilities. γ was 0.97 and Tensorflow and Keras was used to implement the models. The episode length for the CartPole environment was 200, 100 for the Pendulum and 500 for LunarLander. A hyperparameter tuning would be feasible but the algorithm would also lose its generality.

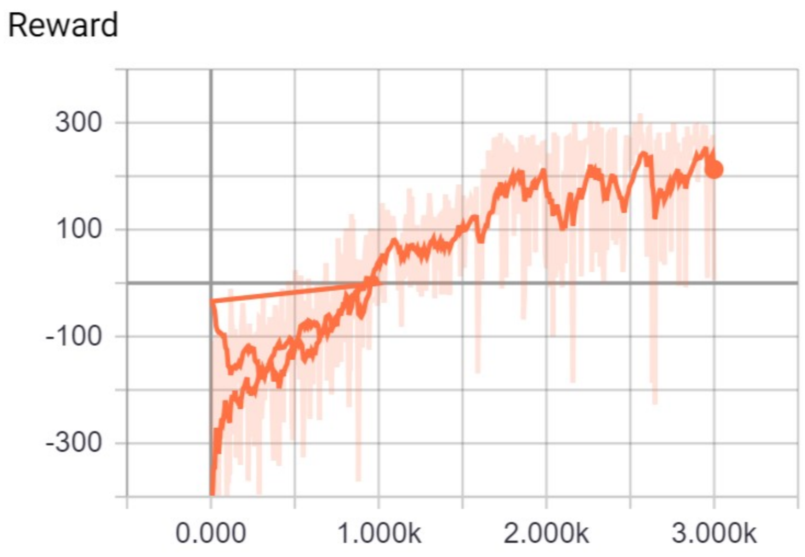

Abbildung 4: REINFORCE agent on LunarLander-v2 environment, average reward over episodes

3.2 DDPG

3.2.1 Pendulum

We evaluate the DDPG algorithm on the pendulum swing-up-task environment which uses a continuous action-space.

Implementation details: We use almost the same hyperparameters as the ones used in the original paper ($\gamma = 0.99$, $\tau = 0.001$, $\epsilon_{start} = 1$, $\epsilon_{decay} = 0.95$, $\epsilon_{end} = 0.001$, learning rate of 10^{-4} nd 10^{-3} for actor and critic respectively, two hidden layers with 400 and 300 units for the networks, $\alpha_{PER} = 0.3$, $\beta_{start,PER} = 0.4$, $\beta_{frames,PER} = 15000$, mini-batch size 64, replay buffer size 10^6, $\mu_{OU} = 0$, $\sigma_{OU} = 0.3$, $\theta_{OU} = 0.15$, AdamOptimizer). We do not use any weight decay (suggested in the original paper) in our network because we noticed a decrease in performance when using it with this environment. We perform online updates after every action taken.

The results of the pendulum environment with and without PER can be seen in figure (5) and (6). The agent consistently learns to perform the swing-up task. The agent with PER seemms to learn it a little faster which is indicated by the decreased amout of reward drops after episode 100. I also tried to immediately calculate the current td-error when storing a transition and using it (its absolute value) as the initial priority instead of the maximum over all priorities. However, this decreases the number of steps per second and causes the agent to perform worse than before.

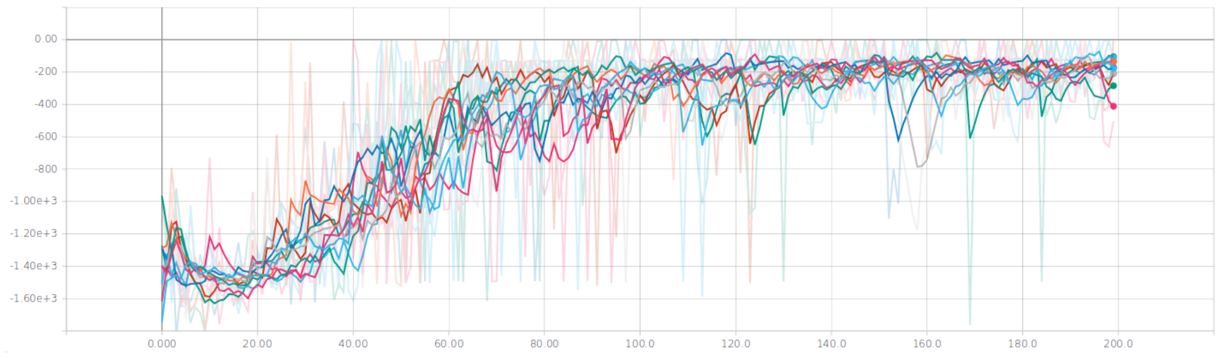

Abbildung 5: DDPG agent on pendulum environment, average reward over episodes

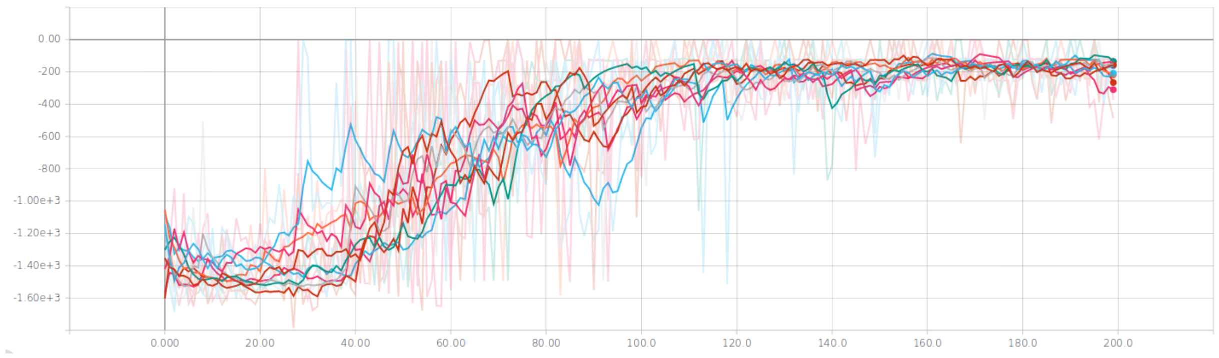

Abbildung 6: DDPG agent with PER on pendulum environment, average reward over episodes

3.2.2 Laser-Hockey

For this environment it is important to notice that we are dealing with very sparse rewards. The only reward signal we ever get is when the puck reaches one of the goals. This makes learning much harder compared to the pendulum environment.

The plan was to first learn shooting and defending in the two provided environments "train shooting"(vs. static agent, static ball initially on our side) and "train defending"(vs. static opponent and ball initially moving towards our goal) and then combine the two agents by maybe training another network by self-play to decide which proposed action to apply.

We start by training how to shoot the puck. Exploring is very important here because of the sparse rewards, therefore we decay our ϵ very slow, otherwise the agent starts to follow its own policy too soon which mostly results in the agent getting trapped in a corner every game. We use the same settings as before except $\epsilon_{decay} = 0.99$ and $\beta_{frames,PER} = 1e5$ (because the training lasts much longer). Learning progress is very slow here, I tried experimenting with different training methods (e.g. discarding most of the games with no goal scored or playing many completely random games in the beginning to have at least some goal transitions stored in the buffer) but none of them seemed to really make the agent a good shooter. On some training runs the agent learned to shoot the initially non-moving ball (not with good accuracy though) but if the ball wouldn't hit the goal and bounce back, which was the case most of the time, the agent didn't know how to deal with the moving ball. One training run can be seen in figure (7), where I discarded all the games without a goal scored, this made the agent aim for the ball initially but after that it was clueless, resulting in mostly random goals scored throughout the run.

I noticed that the main issue preventing the agent from learning was that the reward signal

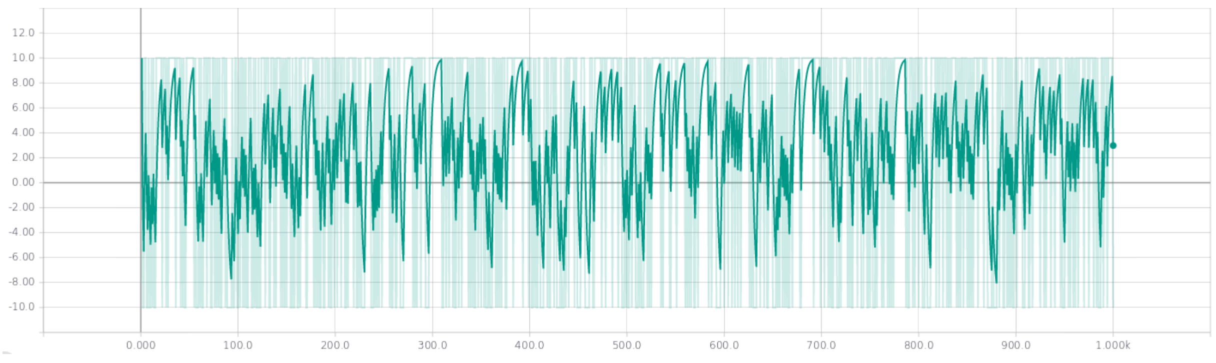

Abbildung 7: DDPG agent on train shooting environment ignoring draws, average reward over episodes

comes in delayed in time (actions responsible for scoring the goal are not the ones right before the reward comes in), which basically makes it impossible to learn shooting for the agent in its current version. First I tried to implement eligibility traces, but those are not easily combined with the PER (eligibility traces require time-correlated sequences for training, PER does the exact opposite by decorrelating the transitions). Later I decided to do offline updates instead of online updates, which didn't really harm the agent because we don't need the online updates for very fast convergence, but it allowed me to calculate the lambda reward before storing a transition to the replay buffer. Specifically, I save all transitions of one episode in an extra buffer, when the episode is done I update the rewards according to

$$R_t^{(n)} = \sum_{k=0}^{n} \gamma^k R_{t+k}$$

$$R_t^* = R_t^\lambda = (1-\lambda) \sum_{n=1}^{N} \lambda^{n-1} G_t^{(n)}$$

which is the lambda return. Unfortunately the offline computation is quite costly, I will refer to this in the discussion. For this specific environment however we only need to calculate it in case the final reward is non-zero, or even easier, we can just use the discounted future reward instead:

$$R_t^* = \gamma^{N-t} R_N$$

After updating the rewards I move all transitions to a transition-queue. From this queue I add one transition (with priority) to the real replay buffer before every training step. The alternative would be to immediately add all the transitions from the last episode to the buffer, but I think this would not be the best option because all new transitions in the buffer get assigned a high priority and therefore we would massively sample transitions from the last episode at once, which might cause our agent to overfit to this episode. Instead we slowly add transitions from the queue to the buffer by uniformly sampling one at a time.

This whole process makes the reward signal backpropagate in time which makes it possible for the agent to learn which actions really caused a goal.

A training run for shooting can be seen in figure (8). Although it might not look like it, it performed much better than before, which is mostly indicated by the amount of goals being

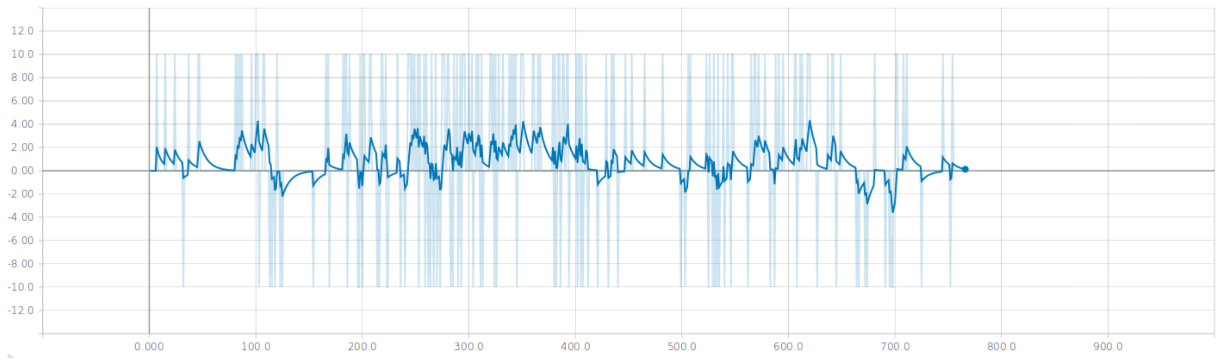

Abbildung 8: DDPG agent with on train shooting environment, average reward over episodes

scored. The agent with online updates had trouble even hitting the puck (except when discarding draws, which we didn't do in this run). Now the agent at least learned to aim for the puck.

Unfortunately I didn't have enough time to further explore the new version. I'm confident that with some tuning this version is capable of learning a good policy.

4 Discussion

The DDPG algorithm is able to solve the pendulum environment, this shows that the algorithm itself is working quite good. Unfortunately this implementation was not yet able to solve the laser-hockey task, the main reason being that only in the end I noticed that I could get around the eligibility traces by just doing offline-updates. I think that with the offline updates and some hyperparameter tuning the algorithm would able to beat the basic opponent. Maybe it would also be worth to experiment with different noises, Ornstein-Uhlenbeck might not be the best noise to apply for this problem. It would also be interesting to try to implement the methods used in a recent paper on efficient eligibility traces in reinforcement learning [ETRL, 2018] which would allow the usage of experience replay while using eligibility traces, which is much more efficient than the offline lambda-return calculation I'm using at the current state.

The REINFORCE algorithm was also able to solve the simple environments. Whereas it uses a stochastic policy, DDPG is designed for a deterministic policy with a continuous action space. On one hand DDPG can perform any available action and is therefore able to find a better minimum and has to learn from a lot less action dimension space, especially for increasing dimensions, where a discrete policy would end up with a d^k dimensional action space. On the other hand a stochastic policy generalizes over the action and is also able to learn stochastical behaviours of the environment, like taking half the time action A and the other times B. In a recent paper they were able to overcame the increasing dimensionality problem and also got state-of-the-art performance with significant performance gains [DAS, 2019].

So in conclusion traditional policy gradient and DDPG are valuable algorithms that pioneered our improvements in reinforcement learning. They are well suited for simple tasks and can sometimes even accomplish great results given enough compute. It is great to see that with increasing complexity of the problems we as humans want solve, that there are still simple and general algorithms that can tackle these tasks like PPO, having a much bigger potential than first assumed. But in those cases it is really for the best working at a place which has access to the great amount of compute needed train those models.

Literature

[PPO, 2017]	Schulman, Wolski, Dhariwal, Radford, Klimov, OpenAI, 'Proximal Policy Optimization Algorithms', arXiv:1707.06347
[DDPG, 2016]	Silver et al., 'Continuous Control With Deep Reinforcement Learning', arXiv:1509.02971
[DOTA 5v5, 2018]	OpenAI, 'OpenAI DOTA 5v5', URL: https://blog.openai.com/openai-five/
[PER, 2016]	Schaul, Quan, Antonoglou, Silver, 'Prioritized Experience Replay', arXiv:1511.05952
[ETRL, 2018]	Daley, Amato, 'Efficient Eligibility Traces For Deep Reinforcement Learning', arXiv:1810.09967
[DAS, 2019]	Tang, Agrawal, 'Discretizing Continuous Action Space for On-Policy Optimization', arXiv:1901.10500