DRLND Continuous Control- Report

Matthias Schinacher matthias.schinacher@googlemail.com

2018-12-18

Contents

1 Intro

The project is a homework assignment for Udacity's **Deep Reinforcement Learning Nano Degree**. The project ist the second one called *Continuous Control*.
The project environment is a course provided version of the Unity "Reacher" environment of the ML- agents Unity has on the respective github- page. The environment has an "agent" - a robotic arm - which can be manipulated using 4 continuous-value actions with range -1.0 to 1.0 that must be provided at each timestep.
The environemt also contains a slowly moving ball- shaped target area, that the robotic arm must hit with it's end (it's hand?). The longer the robot keeps being in the target area with the hand, the higher the reward.
For this project I chose to implement DDPG with experience replay and a variant of priority replay in a python script to be invoked from command line.
I developed the script initially from the one I wrote for the Navigation- project, as there was a lot of overlap in how the assignments were structured (like a unity agent with a similar API that needs to maximize rewards over a number of episodes).
The DDPG needs 4 approximations (actor, critic and the 2 target- variants of them) that I modeled as neural networks with fully connected layers, ReLU layers, $Tanh$- layers and batch-norm ("BatchNorm1") layers using PyTorch.

1.1 Amendment 2018-12-18 (Version 3)

My attempt to solve the 1 agent environment were initially not conclusivly successful and the reviewer suggested to try the 20 agent version.
So I did that, but sadly not very successful. Thus I abandoned the 20 Agents approach and modified the single agent variant; the respective updates to the report are marked by **Amendment 2018-12-18 (Version 3)**.

2 Implementation

2.1 Python script

The script is named *ms_drlndcc_pr.py* and must be invoked from the command line with exactly one parameter, the name of the ".ini"- file (a.k.a. command file), that has all the parameters.
Amendment 2018-12-18 (Version 3): I changed the implementation of the script sligtly; the changes (plus one or two bug-fixes) were:

- Priority Replay: changed the implementation to a more traditional approach of using the "unexpectedness" of the transition as the basis instead of the the rewards.

- Additional flags and parameters: I added flags to be able to turn on/off priority-replay, batch-norm layers in the model and gradient-norm clipping for the critic optimization.

The δ of the prio-replay, upon which the priority of a transition is based, now uses the difference between the state-action value the critic computes and

the estimation of this value by the target critic using the reward and the target critics state-action value of the subsequent state.
The prio-replay is a partial implementation of the "PRIORITIZED EXPERIENCE REPLAY" paper by Tom Schaul, John Quan, Ioannis Antonoglou and David Silver

Parameters The parameters listed in the command file come in various sections. The basic format is similar to the Windows style INI files and is the one that pythons *configparser* module uses (as the module is used in the script).

Example :

```
[global]
runlog = test06.log
[mode]
train = True
[rand]
seed = 32177
[model]
h1 = 237
h2 = 237
c_h1 = 237
c_h2 = 237
load_file = test05
save_file = test06
[hyperparameters]
episodes = 50
warmup_episodes = 5
warmup_episodes_f = 0.5
replay_batchsize = 64
replay_steps = 5
optimizer_steps = 5
tau = 0.001
reward_gamma = 0.95
reward_offset = 0.01
no_reward_rm_prob = 0.05
epsilon_start = 0.3
epsilon_delta = 0.000001
epsilon_min = 0.0031
```

Amendment 2018-12-18 (Version 3)
Note that the parameters "reward_gamma", "reward_offset" and "no_reward_rm_prob" are now obsolete, since the prio-replay is not based on the transition-rewards as before.

Description :

Parameters			
Section	Name	Description	Default
global			
	runlog	name of the logfile to use	`run.log`
mode			
	train	whether we're in training mode	`True`
	show	flag, whether to show the simulation in "human time"	`False`
rand			
	seed	seed for random number generation	no explicit random seeding performed
model			
	h1	first size- parameter for the actor- NN- model	`10`
	h2	second size- parameter for the actor-NN- model	`10`
	c_h1	first size- parameter for the critic- NN- model	`10`
	c_h2	second size- parameter for the critic-NN- model	`10`
	load_file	name- fragment for the files from which to load models (if any)	None
	save_file	name- fragment for the files to save the models to	"`DDPG-out`" if in training mode
hyperparameters			
	episodes	number of episodes to run	`200`
	warmup_episodes	epiosodes to run with pure random sampling	`10`
	warmup_episodes_f	scale factor for pure random sampling	`0.4`
	replay_buffersize	size of the replay memory	`99000`
	replay_batchsize	number of transitions to sample per optimizing step	`128`
	replay_steps	simulation-steps between each optimization run	`20`
	optimizer_steps	no. of batch optimization-steps per optimization run	`10`
	learning_rate	the learning rate	`0.001`
	gamma	γ	`0.99`
	tau	τ (soft target update)	`0.001`
		replay prioritization	
	reward_gamma	reward- γ discount factor	`0.99`
	reward_offset	importance offset	`0.01`
	no_reward_rm_prob	probability for transition with zero- reward to enter replay- memory	`0.25`
		sample action noise	
	epsilon_start	start value for ϵ	`0.5`
	epsilon_delta	value to subtract from ϵ for each optimization step	`0.5`
	epsilon_min	minimum/ final value for ϵ	`0.001`
	noise_theta	θ for noise process	`0.15`
	noise_sigma	σ for noise process	`0.02`

Note: the *show*- parameter determines, if the unity engine should show the

episodes slower, as to allow a human observer to follow the simulation; therefor it determines the value given to the unity- environment as the parameter *train_mode*.
The *train*- parameter of the script determines, if the algorithm will be learning from new transitions.

Amendment 2018-12-18 (Version 3)
"reward_gamma", "reward_offset" and "no_reward_rm_prob" are no longer used.

New and changed parameters:

Parameters			
Section	Name	Description	Default
model			
	batch_norm	flag, whether batch-norm layers are used	`False`
hyperparameters			
	learning_rate	the learning rate for the actor	`0.0001`
	learning_rate_c	the learning rate for the critic	`0.001`
	grad_norm_clip	threshold for grad-norm clipping; negative means no clipping	`-1.0`
replay prioritization			
	prio_replay	flag, whether to use prio-replay	`True`
sample action noise			
	noise_sigma	σ for noise process	`0.2`

2.2 Dependencies

The script has runtime dependencies; these are the ones as described in the project instructions; I will therefore refrain from detailing the dependencies *here*.

2.3 Details/ algorithm

The algorithm implemented is basically Deep Deterministic Policy Gradient (DDPG) with a few tweaks.
The replay memory-buffer is implemented as a simple list with a specified capacity (*replay_buffersize*), where the oldest entry is overwritten by the newest entry, if the capacity is already fully used.
The script uses a specified number (*replay_batchsize*) of transitions to perform the optimization step; how often the optimization step is performed is determined by *replay_steps*, that is once every *replay_steps* simulation-steps the optiization is performed, and *optimizer_steps* controls how many batches/ optimization steps are performed in sequence then.
Also the script implements an unusual non standard form of priority replay as well as a noise adjustment of the sampled actions.

Priority replay Different from the usual priority replay, where the sampling probability of the transition is governed by "how unexpected" the transition was (difference between actual reward and reward expected by the current state value function) I implemented a prioritization based on the actual reward only. This scheme has 3 elements:

- a transition with zero reward is only registered to replay memory with a certain probability (*no_reward_rm_prob*); transitions with reward greater zero are always registered.

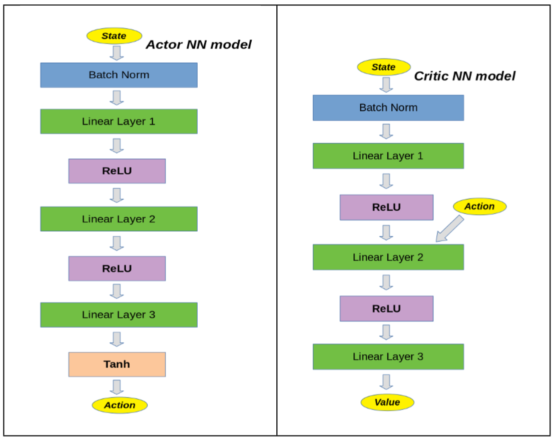
- sampling probability from the replay buffer for a batch is relative to the transition-reward plus a reward offset (*reward_offset*, to allow for sampling of zero reward transitions)
- the rewards used to compute the sampling probability are discounted by the reward- γ factor in each optimization run to favor newer transitions.

Amendment 2018-12-18 (Version 3)
The described scheme for the prio-replay is no longer used; instead the prio-replay is now based on the difference between the state-action value the critic computes and the estimation of this value by the target critic using the reward and the target critics state-action value of the subsequent state. This is the approach were prioritization follows the "unexpectedness" of the transition.
The prio-replay is a partial implementation of the "PRIORITIZED EXPERIENCE REPLAY" paper by Tom Schaul, John Quan, Ioannis Antonoglou and David Silver. By using the *prio_replay* flag/ parameter, replay prioritization can be turned on/off per simulation run.

Warmup episodes The script/ program computes the action values to be used randomly for a number episodes (*warmup_episodes*), before the actual actor- model is used. The values are sampled from the standard normal distribution and mupltiplied by the factor *warmup_episodes_f* before they are clipped to the range -1 to 1.

Sampling noise When not in warmup, the action values are computed using the actor model, to which a noise (times ϵ) is added. The noise is computed by $noise = noise - \theta noise + \sigma random\_number$. This noise is initialized with zero and the *random_number* is taken from the usual uniform distribution between 0 and 1.
As the ϵ factor decreases the actual noise applied decreases also.

Neural Network layout The approximation functions for the actor and critic (the *normal* and the target variants) are implemented as neural networks using PyTorch.
Each uses 3 linear layers with ReLU- layers after the first and second linear layers and a batch-norm layer before the first linear layer. The actor model has a final $tanh()$- layer (the critic does not). The critic inputs the only the state to the first layers and mixes the action- input after the first ReLU by concatinating it to the ReLU layers output (before it goes intu the second linear layer).



Note: with 3 linear layers each and fixed sizes for the input (by the unchangable values for state size and action size) as well as output (action size and 1, cause the critic must yield a value), there are 2 choosable sizes for the actor and critic each (hence the parameters).

Amendment 2018-12-18 (Version 3)
By using the *batch_norm* flag/ parameter, the layout can be changed per simulation run to not include the batch-norm layers.

2.4 Usage

The script is invoked from the command line as a python script with one parameter, a file-name from which to read all the parameters governing the algorithm. Example:

```
python ms_drlndcc_pr.py test05.ini
```

2.5 Loading and saving models/ pretraining

The script is able to save the model- states of the neural networks as well as the transitions in the replay-memory to file and/or load these from file at the beginning of a script-run.
The parameters allow for a name- fragment to be specified, from which the actual filenames are derived. Each NN- model as well as the transitions- replay buffer (plus reward- buffer) gets it's seperate file.
The models are saved/ loaded using PyTorch's build in mechanism and the replay- buffer file is created using pythons *pickle*- module.

file names	
data	physical file name
actor model	`actor_{fragment}.model`
target actor model	`target_actor_{fragment}.model`
critic model	`critic_{fragment}.model`
target critic model	`target_critic_{fragment}.model`
replay buffer	`transitions_{fragment}.pickle`

The saved model/ transitions allow for a subsequent script/ program- run to pick up, where a previous run ended, effectivly using this as a pretraining. This also allows to continue a simulation-run with adjusted algorithm parameters; the neural net size parameters are ignored when loading aprevous model!

2.6 Outputs

The script prints some info to the standard output, but the actually important output is the run-log file; it prints a (non '#'-) textline per episode containing the episode, the score the episode reached, the average score of the last 100 episodes (or 0, if it's an episode before the 100th), the minimum of all non zero rewards for a single step, the maximum reward for a single step the replay buffer size at the end of the episode and ϵ for the episode separated by a blank. The logfile can thus be read by certain programs/ tools as some sort of time-series for the score and the average-100-last-episodes-score; one such tool is **gnuplot**, with which I created the graphics contained in the report.

2.7 Misc. algorithm details

The algorithm distinguishes between **optimization run** and **optimization step**. A optimization run occurs every *replay_steps* (when not in warmup) and contains *optimization_steps* steps. Each such step uses a freshly sampled batch from the replay buffer to feed it to the models as the DDPG algorithm prescribes; 2 instances of the PyTorch Adam- optimizer are used to make a step for actor and critic.
Note: the target networks are soft- updated per optimization step, and the ϵ for the action noise is also adjusted per step, while the rewards for the prioritization of the replay buffer are discounted per optimization run.

3 Results

3.1 Target score

To my big disappointment, the target score of 30 never even remotly materialized with my experiments/ simulations. I don't think a faster computer with maybe a real GPU (I used my laptop to run the simulations) would have gotten me there even though it would allow for longer runs and bigger or deeper networks.
But I do think, that the Reacher- environment I used **does not yield rewards of 0.1 per timestep, when the robot arm is in the target area!** I think, that it must either be a different environment compared to the one used to establish the score of 30, or there is some subtle dependency on the operating system I use (a variant of Ubuntu 18.04) or other software installed locally, that alters these rewards, that are returned.
I must stress, that I downloaded the environment more than twice from the location given in the instructions; this can not be the source of this problem (I mean a different Reacher environment).
I very purposely let my script print out the maximum and minimum reward (other than zero) returned by the environment per step for each episode, and the typical values are 0.039999999... and 0.009999999..., but never 0.1.
Amendment 2018-12-18 (Version 3):
as suggested by the reviewer, I tried the 20 agents environment instead of the 1 agent environment, but could not make that score the 30.
I then abandoned the 20 agents again and changed the prio-replay implementation and changed some other minor stuff, mostly to have the option of turning off

some features per simulation run. And **voila**, I came across a set of parameters that got me to the desired score of 30!

3.2 General remarks

I had some problems to get to the point where my implementation would yield a simulation-run, that actually clearly demonstrated learning. My DDPG implementation seems to behave quite fickle and is very sensitive regarding the *right* choice of parameters.
It seems to me that others had similar problems, therefor I guess this is mainly due to the DDPG algorithm as such in combination with the Reacher environment :-).
I was then very happy to have found some parameter combinations, where my script would show learning. I did try a variety of different things before I got to this point, not all of which are reflected in the final version of the python script since I deleted them after they did not yield the results I had hoped (plus one or two of these ideas were clearly bonkers to start with, I realize in hindsight).
In the end I did find the solution yielding score 30, but there was also some measure of luck involved.

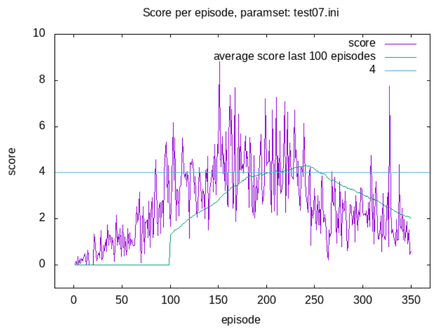
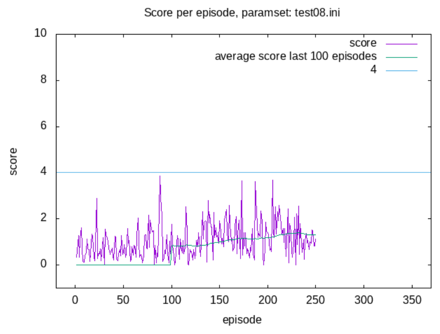
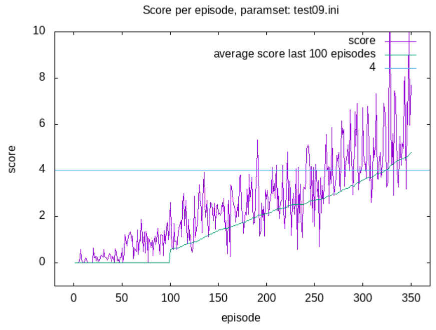
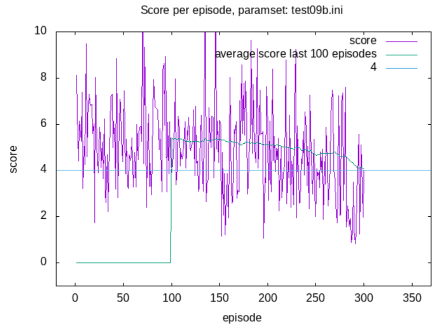
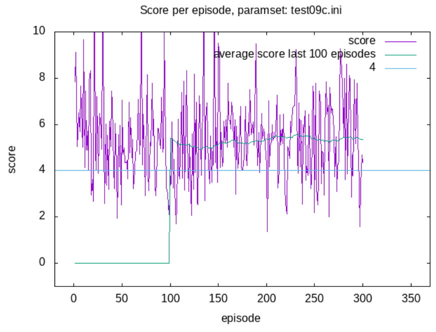
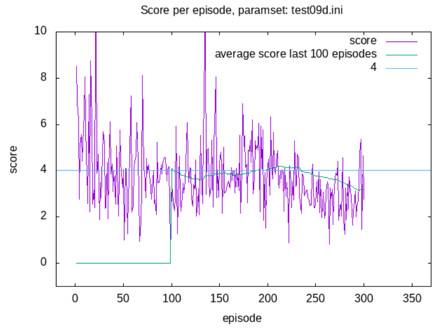
3.3 Results of selected simulation runs

Command files/ INI- files Each command file (a.k.a. INI- file) represents one *experiment* (program/ script- run). See the list below for an overview of these experiments/ command files (refer to the actual files for details of all hyperparameters).

Filename	Description
*test*07.*ini*	first successful simulation with actual learning with size- parameters $h1$ and $h2$ each 117 (actor and critic)
*test*08.*ini*	continuation of *test*07.*ini*, uses model from latter as pre-training different hyper-parameters shows almost no learning!
*test*09.*ini*	second successful simulation actor and critic with different linear layer sizes
*test*09*b*.*ini*	continuation of *test*09.*ini*, uses model from latter as pre-training different hyper-parameters
*test*09*c*.*ini*	continuation of *test*09.*ini*, uses model from latter as pre-training different hyper-parameters
*test*09*d*.*ini*	continuation of *test*09.*ini*, uses model from latter as pre-training different hyper-parameters

Graph- plots *(All plots are score per episode plus average score of the last 100 episodes per episode. All plots are plotted with the same x/y- max/min-*

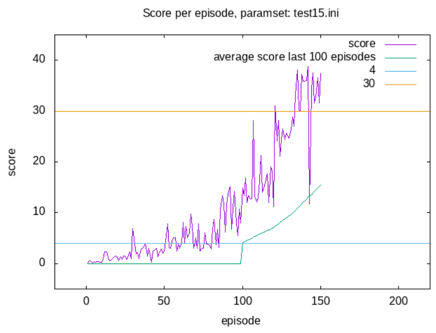
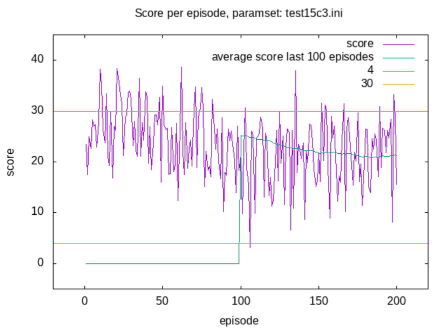
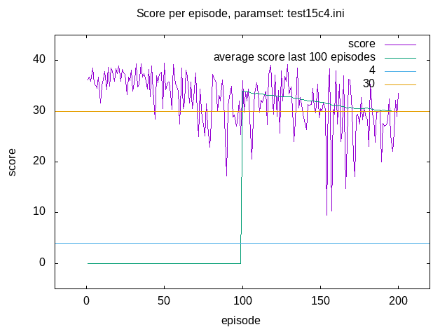
values for comparability. Constants of 4 and 30 are plotted as reference points.)

test07.ini	test08.ini
	
test09.ini	test09b.ini
	
test09c.ini	test09d.ini
	

Command files/ INI- files - 2018-12-18 (Version 3) :

Filename	Description
test15.ini	first simulation with episodes that actually had score above 30
test15c3.ini	continuation of *test15.ini*, uses model from latter as pre-training not with sustained score 30
test15c4.ini	continuation of *test15.ini*, uses model from latter as pre-training with sustained score 30! **The actual solution!**

Graph- plots - 2018-12-18 (Version 3) *(All plots are score per episode plus average score of the last 100 episodes per episode. All plots are with comparable x/y- max/min- values for comparability. Constants of 4 and 30 are plotted as reference points.)*

`test15.ini`	`test15c3.ini`
 	
`test15c4.ini [actual solution!]`	
 	

3.4 Remarks/ discussion

The `test07.ini` simulation shows clear learning and reaches a sustained score of about 4 before mildly "crashing" at about episode 250. I'm frankly not able to say why it goes down like this.

Though `test08.ini` uses the identical model sizes and only sligthly different other hyperparameters, it is not able to learn nearly in the same way. I suspect the main difference to be the changed minimum ϵ noise parameter. Anyhow, I think it shows the sensivity of the algorithm towards changes in these parameters.

`test09.ini` on the other hand seems to learn more slowly than `test07.ini`, but also more "sustainable" (does not go down). The main difference being a much larger batch size and different sizes for the NN- layers, where the 2 size parameters is different (first layer decidedly bigger than the second one) for the actor and critic models; also the size parameters differ between actor and critic. At first look, one would think, the `test09.ini` run would yield even higher scores, if it had been longer, since the average score is still climbing at the end, but since `test09b.ini`, `test09c.ini` and `test09d.ini` are continuations of it (same NN model structure), and they seem to linger on the level reached or even sligthly go down, the score reached seems to be about the best value I can get with my algorithm; Thus about a sustained 5.3- score is what I get.

Amendment 2018-12-18 (Version 3): I was somewhat surprised, that the actual successful simulations would not need batch-normalization, as this was strongly hinted at by some.
Even more surprising was, that the actual configuration, that would maintain the above 30 score for long enough did not use prio-replay or grad-clipping, though the simulation that created the pre-trained models actually did use them.
I would suspect, that the whole solution is very sensitive to **some** of the hyper-parameters, and that especially the sizes of the network and the magnitude of the sampling noise are important.
I would suggest, that the latter was the decisive difference between `test15c3.ini`, which could not deliver a sustained score of 30, and `test15c4.ini`, which not only could, but had an **average score of the last 100 episodes** of above 30 for almost 100 episodes. Thus the criteria is clearly fulfilled.
Though the `test15c4.ini` simulation run clearly met the goal, the score deteriorates sligthly over time; I guess this configuration does not have the perfect parameter-set/ configuration just yet.

3.5 Result models and data

The neural network models for the simulations can be found in the respective `*.model`- files, where they are saved in the PyTorch specific manner; note that you need to use the `MSA` (actor) and `MSC` (critic) classes within the script in addition to PyTorch.
You can also kind of "print out" the models with the script `print_model.py`, but this will not give you all parameter values out of the box (modify the script accordingly, if you want :-)).

3.6 ZIP files

I zipped the resulting log-files, model files, transitions (replay buffer)- files and INI- files in various `*.zip` archives.

4 Possible improvements

4.1 Algorithm

Random sampling/ noise The implementation uses a random-noise source to tweak the actions, that the actor model computes for a state. I could think of 2 straightforward things one might try here:

- applying noise to the state (input) instead of the computed actions (output)
- currently all 4 action dimensions are supplied with noise at each step; this could be altered by randomly choosing not to apply noise per step or applying noise not to all 4 actions; a variety of schemes are possible here.

Tweaking ϵ- decay Additionally, the ϵ- decay for the noise was rather simple, a start value, a minimum value and a delta to subtract per optimization step, resulting in a linear decrease.
The actual simulation results suggest, that the ϵ- parameter does have a serious impact on the algorithms performance; thus other forms of ϵ decay (e.g. non linear) could be explored.

Priority replay The priority replay used seems a bit unusal. The idea was, that a step, where we do get a reward is more important to further examine than one with zero reward.
Of course it would be interesting to see, whether a more traditional approach to priority replay would yield different results.

Non DDPG Of course, because DDPG did not yield the results I wanted, maybe I should try something else, maybe Q-Prop.

4.2 NN- Models

The models for the neural networks have considerable influence on the performance of the simulation as a matter of course.
My gut feeling is, that a much deeper network or bigger layers would not be that promising, but maybe using a different activation function for the actor (instead of $Tanh()$) or adding convolutional layers might smooth the erratic score yield?
This seems to me actually pretty similar to the way my **Navigation-** project output behaved; coincidence?