DRLND Collaboration and Competition- Report

Matthias Schinacher matthias.schinacher@googlemail.com

2018-12-31

Contents

1 Intro

The project is a homework assignment for Udacity's **Deep Reinforcement Learning Nano Degree**. The project ist the third one called *Collaboration and Competition*.
The project environment is a course provided version of the Unity "Tennis" environment of the ML- agents Unity has on the respective github- page. The environment has two "agents" that play a game resembling tennis.
Each agent, represented as a crude form of tennis racket, has 2 continuous-value actions; move towards/ away from the net and move up/ down. If an agent lets the ball reach it's side of the court or shoots the ball outside the court, a negative "reward" of -0.01 is earned and if the agent manages to play the ball across the net, it gets 0.1 as a reward.
The goal is to get as high a reward as possible, and as there is no reward for an agent, when the other agent fails, this means to keep the ball in play as long as possible.
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 derived the script from the one I wrote for the "Continuous Control"- project, which itself was based partially on my script for the "Navigation"- project.
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 using PyTorch.

2 Implementation

2.1 Python script

The script is named *ms_drlnd_collab_comp.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.
The δ of the prio-replay, upon which the priority of a transition is based, 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 implementation partially follows 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 = test11.log
[mode]
train = True
show = False
[rand]
seed = 14111
[model]
save_file = test11
model_h1 = 411
```

```
model_h2 = 277
model_c_h1 = 409
model_c_h2 = 279
batch_norm = False
[hyperparameters]
episodes = 1500
warmup_episodes = 50
warmup_episodes_f = 0.4
replay_buffersize = 10000
replay_batchsize = 384
replay_steps = 1
gamma = 0.99
learning_rate = 0.0001
learning_rate_c = 0.001
optimizer_steps = 1
tau = 0.01
max_steps = 850
epsilon_start = 3.0
epsilon_delta = 0.0035
epsilon_min = 0.01
noise_theta = 0.15
noise_sigma = 0.2
prio_replay = True
prio_offset = 0.2
grad_norm_clip = 5.0
```

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	`311`
	h2	second size- parameter for the actor-NN- model	`177`
	c_h1	first size- parameter for the critic- NN- model	`309`
	c_h2	second size- parameter for the critic-NN- model	`179`
	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
	batch_norm	flag, whether batch-norm layers are used (currently broken)	`False`
hyperparameters			
	episodes	number of episodes to run	`1000`
	max_steps	maximum number of steps in episode	`500`
	warmup_episodes	epiosodes to run with pure random sampling	`50`
	warmup_episodes_f	scale factor for pure random sampling	`0.4`
	replay_buffersize	size of the replay memory	`10000`
	replay_batchsize	number of transitions to sample per optimizing step	`512`
	replay_steps	simulation-steps between each optimization run	`1`
	optimizer_steps	no. of batch optimization-steps per optimization run	`1`
	learning_rate	the learning rate for the actor	`0.0001`
	learning_rate_c	the learning rate for the critic	`0.001`
	gamma	γ	`0.99`
	tau	τ (soft target update)	`0.01`
	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`
	replay_offset	used to calculate priorities (see details for more info)	`0.2`
		sample action noise	
	epsilon_start	start value for ϵ	`2.5`
	epsilon_delta	value to subtract from ϵ for each optimization step	`0.001`
	epsilon_min	minimum/ final value for ϵ	`0.02`
	noise_theta	θ for noise process	`0.15`
	noise_sigma	σ for noise process	`0.2`

- the *train*- parameter of the script determines, if the algorithm will be learning from new transitions.
- though the script will try to honor *batch_norm*, the current implementation contains a bug, so that this feature is not usable currently!

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

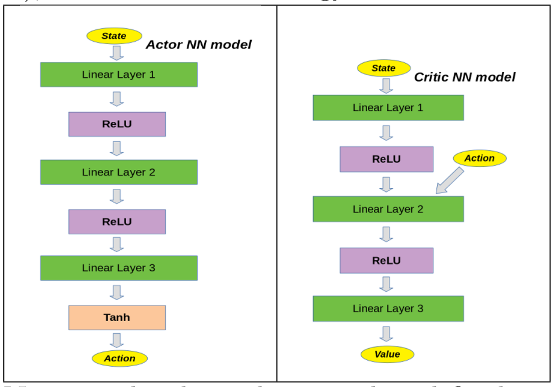
The algorithm implemented is basically Deep Deterministic Policy Gradient (DDPG).
The replay memory-buffer is implemented as a simple list with a specified capacity (*replay_buffersize*), where the oldest entry is overwritten by the newest entry, if the capacity is already fully used.
The script uses 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 optimization is performed, and *optimizer_steps* controls how many batches/ optimization steps are performed in sequence then. Also a noise- adjustment of the sampled actions is used.

Prio- replay Also the script implements priority replay partially following "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.
The priorities computed for the actually sampled transitions to update the replay- priorities in the replay buffer are not simply the δ's for the transitions, but are computed using the `replay_offset`- parameter to be $(replay\_offset + \delta)^2$.

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. 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: the script actually has an option/ a flag that would allow for an optional batch-norm layer before the first linear layer (actor and critic), but the implementation seems to have a bug currently (I'm planning to fix this at some point), so the batch-norm thingy is not usable in the moment.



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).

Neural Network use by the algorithm Though we have 2 agents in the simulation, the 4 networks (actor, critic, target-actor and target critic) represent **one set of DDPG- networks** which **both agents** use; but the state and action vectors fed to the networks are technically **not** joined vectors but the *local* state and action vectors per agent.

Replay buffer/ memory Also the replay buffer is only one replay buffer filled by both agents, thus the algorithm registers 2 transitions per time-step, one for each agent.

Learning Consequently the learning/ optimization of the networks uses the combined collected transitions of the 2 agents.

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_drlnd_collab_comp.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 priority- 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 number of steps in the episode, 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 run, and the ϵ for the action noise is also adjusted per episode.

3 Results

I did need quite some time experimenting with different hyper- parameters to find a combination, that would meet the project target-score.
But before I found a *winning combination*, I experimented with a different

DDPG- variant. Instead of shared actor/ critic networks I used a seperate setup per agent, where each agent had it's own set of actor/ critic- networks (normal and target) and it's own replay buffer.
Unfortunatly, I could not find a parameter combination that would result in meaningful learning (maybe my code still has a bug I missed?) and thus I abandoned this approach (see the script `ms_drlnd_collab_comp_sep.py`)

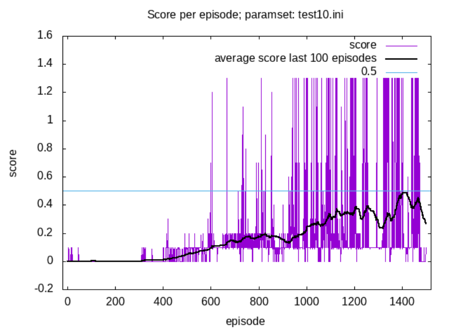
3.1 Results of selected simulation runs

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).

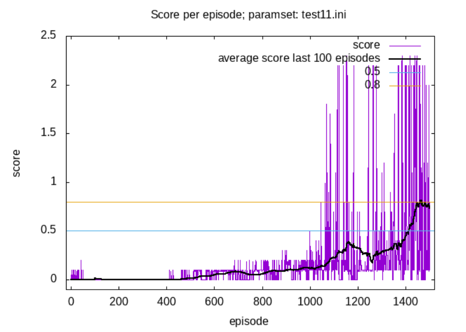
Filename	Description
*test*10.*ini*	simulation with actual learning almost reaching the 0.5 sustained score
*test*11.*ini*	first simulation solving the task
*test*12.*ini*	similar parameters as *test*11.*ini*, shows very little learning
*test*13.*ini*	another parameter combination, again, little learning

Graph- plots *(All plots are score per episode plus average score of the last 100 episodes per episode. Constants (e.g. 0.1 or 0.5 for target score) are plotted as reference points.)*

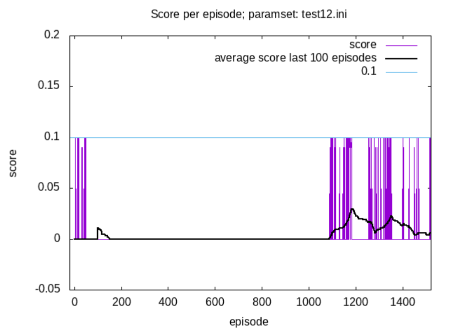
`test10.ini`



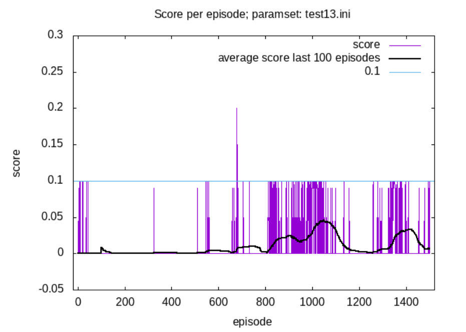
`test11.ini`



`test12.ini`



`test13.ini`



3.2 Remarks/ discussion

The `test11.ini` simulation did reach score 0.5 over the last 100 episodes at episode 1413 and actually reached 0.8 (avrg. last 100).
`test10.ini` almost reached the sought sustained score of 0.5 at episode 1412 with an average score for the last 100 episodes of 0.4875, but this decreases the following episodes. This *failure* might in part be due to the fact, that it uses a maximum number of steps of 500, thus cutting off the maximum score possible per episode (`test11.ini` uses up to 850 time-steps).
`test12.ini` and `test13.ini` use parameters not that vastly different from the solving parameter-set, but show no real learning at all; I conclude that the algorithm is quite fickle/ sensitive to the specific values of at least some of the hyperparameters (but I did not have the time to methodically research which parameters in which way). This is in line with the similar experience from the other project (especially *Continuous Control*).
As mentioned earlier I also experimented with a setup, where each agent had it's own set of networks and seperate replay-memory but could not get that to *learn*. I find this somehow counter- intuitive as the states and action spaces for the 2 agents are symmetric but not identical. Using one set of networks and a joined replay buffer I was expecting to learn at least slower and with more difficulty or maybe requiring larger networks. This seems not to be the case.

3.3 Result models and data

The final neural network models for the simulations (at the end of the simulation run) 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.
For the *solving* simulation runs, there is an additional set of files containing not the networks states at the end of the simulation run, but the networks at the end of the highest scoring episode **after** reaching the sustained 0.5 score critieria (these are the "_max.model"- files).
As only `test11.ini` actually solved the task, the is only one set of these max-model files (written after episode 1417 with score 2.3)
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 :-)).

3.4 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. Since the setup is the same as for the previous project (*Continuous Control*), the same possible modifications apply here:

- applying noise to the state (input) instead of the computed actions (output)
- currently the 2 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 both actions; a variety of schemes are possible here.

Tweaking ϵ- decay Again, as with the previous projects, the ϵ- decay for the noise was rather simple, a start value, a minimum value and a delta to subtract per episode, resulting in a linear decrease.
Different to the last project, my intuition here is, that the final ϵ value is important, but it seems to me the actual decay- scheme not so much (but I did not really research this).
Nevertheless one could try other forms of ϵ decay (e.g. non linear).

Non DDPG Of course, other algorithms could be tried out, e.g. PPO.

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 would not be that promising, but maybe even larger networks or using a different activation function for the actor (instead of $Tanh()$) or adding convolutional layers might smooth the erratic score yield? This again seems to me actually pretty similar to the way my **Navigation-** project output behaved; coincidence?

Batch-norm layer The NN- models implemented in my script *try* to implement batch-norm as an option for the input layer, but when I use it, it currently yields a runtime-error; this is clearly some bug I could fix to experiment with this feature (I could not find the time yet).

4.3 Combined states and action spaces

Though I did not try this yet, my intuition is, that a DDPG- setup with combined state and action spaces to be fed to the neural networks should be most promising.
This would be as if to view the setup not as 2 agents playing together, but one agent with two rackets palying by itself. The combined state-space would be 48 values long (instead of two states with 24 values each) and the number of actions would be 4 (instead of 2 per agent). The replay memory would be filled with one transition per time-step (instead of 2 in the current implementation, where we collect one transition per agent per step).
This would be on the other side of the spectrum compared to the *each player by itself*- approach I **did** experiment with, but since the task does **not** pit the 2 agents against each other, but is a more collaborative task, where each agent only get's a high score if the other agent does also, I would think it would be promising.