Udacity DRLND: Collaboration and Competition

Research Report

Andrzej Wodecki

February 7th, 2019

Introduction

This report is a summary of results of the ***Collaboration and Competion*** **project of Udacity Deep Reinforcement Learning Nanodegree.**

The goal of the project is to **train two Agents to bounce a ball over a net** in the Tennis environment provided by Unity Environments. After training they should be able to play for a longer time without having the ball hit the ground or fall out of bounds.

The state space has 8 variables like the position and velocity of the ball and racket, and each agent receives it's own, local observations (in fact they form stacks of 3 consecutive timesteps observations resulting with state dim = 24) . **The action space** consists of 2 continuos actions: a movement (toward or away from the net) and jumping.

This is **episodic** environment. It is considered **solved** when agents get an average score of +0.5 over 100 consecutive episodes, with the score beeing the maximum of the scores of both agents.

Implementation strategy, the learning algorithm and a brief discussion of results are described in more details in the next sections of this report.

Implementation

The main steps of the procedure are the following:

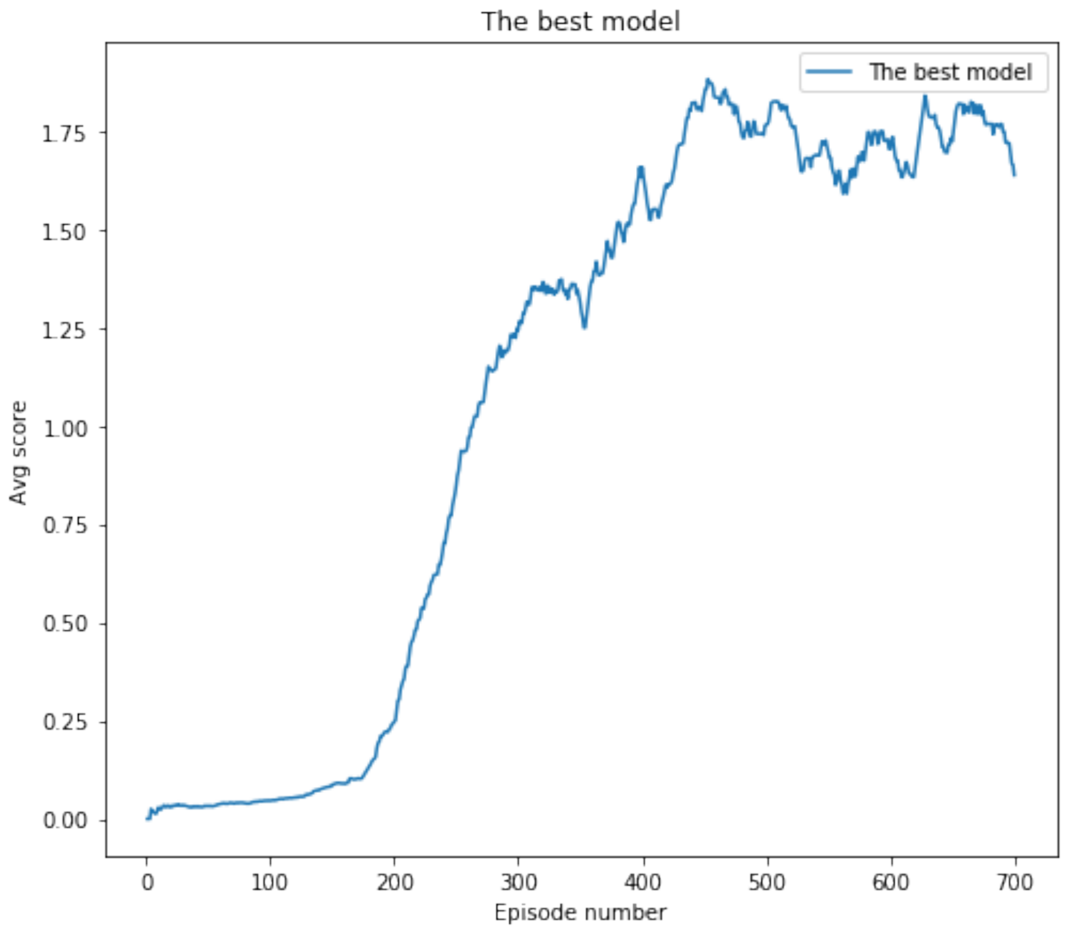
1. Initiate the environment
2. Initiate and train the agent
3. Check agents performace and store it's results.

The most important part of the process is **agent training**. In this project I adopted **Deep Deterministic Policy Gradients (DDPG)** implementation provided by Udacity as the main agent training framework, and extended it with multi-agent and hyperparameters grid search capabilities.

After training of literally hundreds of models I have finally identified the best performing one. It was able to solve the environment in **219 episodes**** (see the figure below), after 700 episodes of training achieved the average (over 100 consecutive time steps) **score 1.88** , and had the following hyperparameters values (cf next section for their definitions):

1. n_episodes = 700
2. max_t = 1000
3. UPD = 1
4. GAMMA = 0.99
5. BUFFER_SIZE = 1e5
6. BATCH_SIZE = 128
7. WEIGHT_DECAY = 0.0
8. LR_ACTOR = 1e-3
9. LR_CRITIC = 1e-4
10. TAU = 1e-2
11. fc1_units = 300
12. fc2_units = 400.

All the actor weights, results and corresponding plots are stored in *The best model* subfolder. The avarege score of the winning agent over subsequent episodes is presented in the figure below.



Code structure

The structure of the code is the following:

1. *run.py* is the main code. Here all the parameters are read, training procedures called and the results written to the appropriate files and folders.
2. *parameters.py* stores all the hyper parameters - the structure of this file is presented in more details below in the *Hyperparameter grid search* section of this report.
3. all the results are stored in (see *Hyperparameter grid search* section below:
    1. *results.txt* file
    2. *models/* subdirectory.

To run the code:

1. Specify hyperparameters in the *parameters.py*. Be careful: too many parameters may results with a very long computation time!
2. run the code by typing: *python run.py*
3. ... and check results: both on the screen and in the output files/folders.

Learning algorithm, hyperparameters and model architecture

Learning algorithm

For Deep Q-Network algorithm I adopted the Udacity's Deep Deterministic Policy Gradient (DDPG) implementation available eg. here: https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum, with theoretical background described eg. in Lillicrap *at al*, *Continuous control with deep reinforcement learning*, https://arxiv.org/abs/1509.02971. The multi agent DDDPG (here it stands for Distributed Distributional Deterministic Policy Gradients) implementation is very well described eg in Barth-Maron *et al, Distributed Distributional Deterministic Policy Gradients*, https://arxiv.org/abs/1804.08617.

The most important part of the training is implemented in the `ddpg` function. For each episode:

1. Environment is reset (with `train_mode`parameter set to `True`), it's `state` is presented to the agent and `score` parameter set to `0`
2. Two agents are created, with a shared memory (reply) buffer and a critic network.
3. Next, every time step:
    1. both agents choose the action
    2. this action is "presented" to the environment
    3. environment returns (for each agent) `next_state, reward` and `done` information
    4. agents absorb this parameters to update the Q-network (`step` method)
    5. `state` and `score` parameters are updated, and appropriate average scores calculated.
4. Overall `ddpg` function returns the final `score` list.

Model architecture

Actor and Critic neural networks are implemented in `model.py`. They contain:

1. Input layer with 24 input nodes (which is an output state space size generated by the Unity env - each observation for each agent contains of 3 observations of 8 variables from consecutive timesteps)
2. 2 hidden layers, with number of nodes beeing a parameters (in my research: 200, 300, 400) and an activation function (I checked two options ReLU and Leaky ReLU provided by PyTorch)
3. *Batch normalization* (only for Critic)
4. Output node with 2 nodes (action space size).

Hyper-parameter grid search and it's results

There are many different parameters influencing the final efficiency of the MA(Multi Agent)DDPG agents (for a short description see the table below):

1. **training procedure parameters**, like n_episodes or max_t
2. **agent parameters**, like BUFFER_SIZE, BATCH_SIZE, TAU, GAMMA, WEIGHT_DECAY, learning rates or Ornstein-Uhlenbeck process.
3. **neural network (model) parameters** like network architecture (including layer types and dimensions or existence of dropout layers) or activation function type.

First (unsuccessful) experiments showed the need for much more organized research. After checking many different options, my final hyperparametr grid search procedure allows to scan the following parameters:

Hyperparameter	Description	Type
n_episodes	max number of the episodes in the training	int
max_t	max number of time steps in the training episode	int
BUFFER_SIZE	replay memory size	int
BATCH_SIZE	minibatch replay buffer size	int
TAU	soft update of the target Q-network parameter	float
GAMMA	discount factor (for future rewards)	float
WEIGHT_DECAY	regularization parameter	float
UPD	the frequency of updating the online Q-network	float
fc1_units	number of nodes of the first layer of the neural network (both for Actor and Critic)	int
fc1_units	number of nodes of the second layer of the neural network (both for Actor and Critic)	int
LR_ACTOR	Learning rate for the Actor network	float
LR_CRITIC	Learning rate for the Critic network	float
noise	Initial level of noise amplitude in the Ornstein-Uhlenbeck noise process	float
noise_reduction	noise amplitude reduction factor in the Ornstein-Uhlenbeck noise process	float

Below I present the results of my experiments and the final, winning model.

Grid search set-up

Based on my experiences from the Continuous Control project I decided to:

1. use normal ReLU activation function for the Actor neural network
2. apply batch normalization and LeakyReLU activation for the Critic network.
3. fix the following hyper parameters:
    1. n_episodes = 700
    2. max_t = 1000
    3. UPD = 1
    4. GAMMA = 0.99
    5. BUFFER_SIZE = 1e5
    6. BATCH_SIZE = 128
    7. WEIGHT_DECAY = 0.0
    8. LR_ACTOR = 1e-3

After first initial scans I've found that:

1. **There exist a kind of a break even point (episode) in the agents learning curve**: until that point there is almost no learning at all, but after the learning "explodes".
2. **Results are very "unstable"**: the "break even episode" mentioned above can be very different for few runs on the same machine (I was working on 3 different servers in parallel) with the same set of parameters.
3. **Computation speed drastically decreases with episodes**, especially after the agents achieve the "break even episode" in their learning curve. This is probably due to the fact that more skillful agents play for much longer without pitfalls.
4. A bit surprisingly compared to my experiences from the *Continuous control* project, **small values (like 1e-3) of TAU parameter decrease the model efficiency**.

Following this assumptions, I decided to divide my research into 2 steps:

1. **Identify the best performing models**: those, which solve the environment in the lowest number of episodes. Stop learning at that moment and go start training a new model
2. **Train the best models for 700 episodes -** identify the best possible average score.

Experiment 1: Identify the best performing models

As mentioned above, the goal fo this experiment was to identify the models which solve the environment in the lowest number of episodes.

I scanned the following set of model hyper parameters:

1. TAU = 5e-3, 1e-2 (2 values)
2. LR_CRITIC = 1e-4, 1e-3 (2 values)
3. Three different sets of fc1_units and fc2_units (just to check the influence of the size of hidden neural network layers on agents learning efficiency):
    1. 100, 200
    2. 200, 300
    3. 300, 400

Altogether, it made 2x2x3 = 12 models to scan, what took app. 12 hours a quite fast, compute optimized AWS instance (c4.4xlarge).

All the results are stored in the *Exp1. The best performing models* folder.

Results for all the 12 models scanned: *results.txt*. In this file:

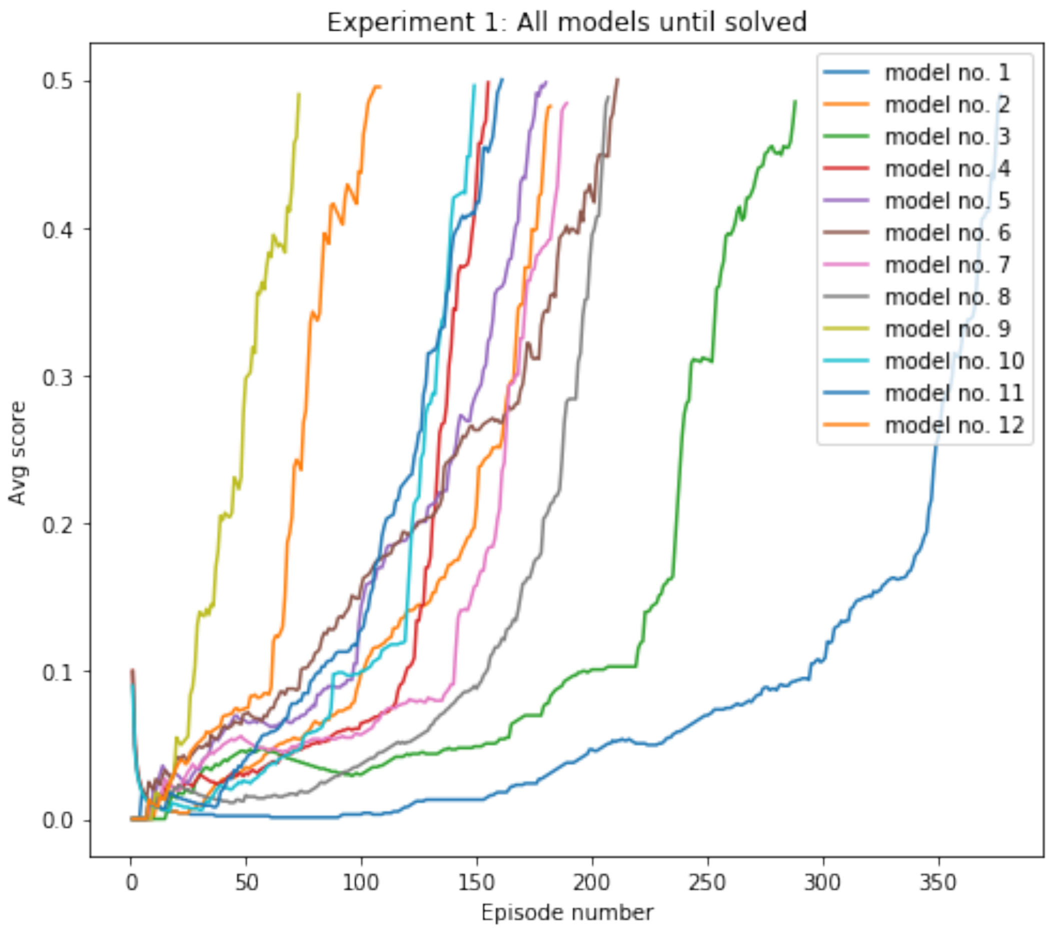
1. first 4 columns contain:
    1. model number
    2. episode number (all 700 episodes, even if the environment has been solved earlier)
    3. score
    4. average score (over 100 consecutive timesteps)

2. next columns contain all the hyperparameters in the following order: *TAU, LR_ACTOR, LR_CRITIC, [fc1_units, fc2_units]*

The information on the episode at which model solved the environment is stored in the *solved.txt* file, where:

1. the first column denotes the model number
2. the second column denotes the episode when the agent solved the environment. **Important notice:** the *episode number* in this file and the table below corresponds to the episode, after which avarage over 100 consecutive episodes is >= 0.5 (see *run.py*). According to UDACITY definition the environment was solved 100 episodes earlier.
3. the average score at the end of the episode which solved the problem.

The plot of the average (over 100 consecutive time steps) score is presented in the table below (cf Exp1.ipynb notebook to generate it).



The TOP5 best performing models are outlined in the table below:

Model number	The episode solved	TAU	LR_Critic	fc1_units	fc2_units
9	74	1e-2	1e-4	200	300
12	109	1e-2	1e-3	300	400
10	150	1e-2	1e-3	200	300
4	156	5e-3	1e-3	200	300
11	162	1e-2	1e-4	300	400

As mentioned above, when interpreting these results we must take into account the "instability" of the algorithm: next runs with the same parameters and on the same machine may generate a bit different outcomes.

Main conclusions:

1. Optimal value of TAU is 1e-2
2. LR_Critic and hidden layers dimensions doesn't seem to be crucial.

Experiment 2: Fine-tune the best performing model

The goal of this experiment was to fine tune the most promising models by training them for a longer number of episodes.

Due to the increasing (with the number of episodes) computation time I decided to limit this to 700 episodes. With this set-up, training 4 models on a compute optimized AWS instance (c4.4xlarge) took app. 24 hours.

For this experiment I choose 4 models with the following parameters:

Model number	Max. avg score achieved	TAU	LR_Critic	fc1_units	fc2_units
1	1,537	5e-3	1e-4	200	300
2	1,634	5e-3	1e-4	300	400
3	1,749	1e-2	1e-4	200	300
4	**1,881**	**1e-2**	**1e-4**	**300**	**400**

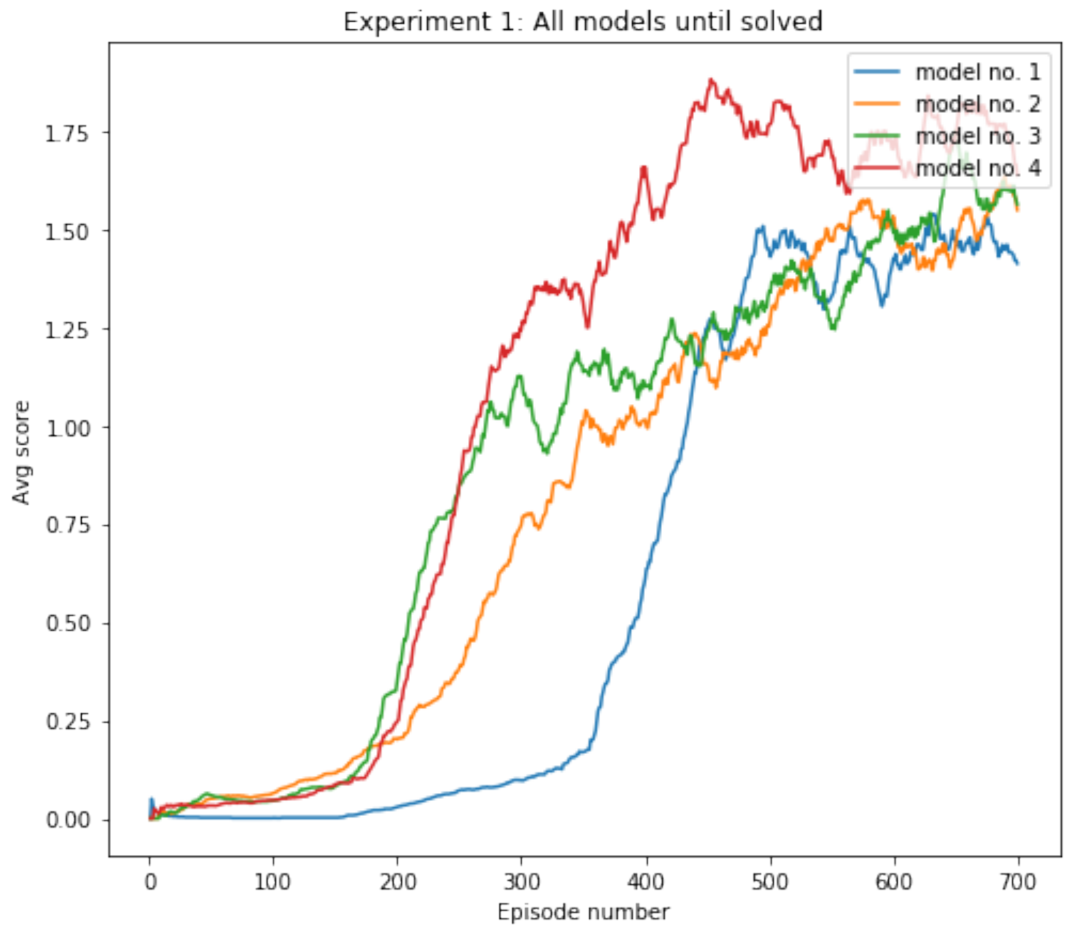
All the results are stored in the *Exp2. Fine tuning* folder.

1. **Results for all the models scanned: *results.txt***. In this file:
    1. first 4 columns contain:
        1. model number
        2. episode number (all 700 episodes, even if the environment has been solved earlier)
        3. score
        4. average score (over 100 consecutive timesteps)

2. next columns contain all the hyperparameters in the following order: *TAU, LR_ACTOR, LR_CRITIC, [fc1_units, fc2_units]*

2. **The models which solved the environment** are stored in the *models* subdirectory:
    1. **successful actor and critic models** as *.pth files, where suffix corresponds to the model number. **I stored *.pth files for the best performing episode of a given model**. For example, if the avg. score at the 700th episode was 1.81, but on the episode 775 it was 1.87, I stored the latter one. That enables identification of the best overall "brains" of our agents.
    2. **list of the models which solved the environment**t: *models_solved.txt*, where:
        1. the first column denotes the model number
        2. the second column denotes the episode when the agent solved the environment. **Important notice:** the *episode number* in this file corresponds to the episode, after which avarage over 100 consecutive episodes is >= 30 (see *run.py*). According to UDACITY definition the environment was solved 100 episodes earlier. Thus, in this report I subtract 100 from the number printed to *models_solved.txt*
        3. the average score at the end of the episode which solved the problem.
3. **A jupyter notebook with supporting plots.**

The avg score for all the models searched are presented in the plot below:



The winning model 4 achieved the avg. score at the level of 1.881 on 453rd episode of training.

As can be seen, a quite small learning rate for the Critic (LR_CRITIC = 1e-3), TAU = 1e-2 and a bit larger hidden layers (300 and 400) may be helpful in achieving better results.

Ideas for the future improvements

For the future improvements I recommend to explore:

1. other RL algorithms, like:
    1. Asynchronous Actor-Critic Agents (A3C),
    2. Distributed Distributional Deterministic Policy Gradients (D4PG),
    3. Trust Region Policy Optimization (TRPO)
    4. or Proximal Policy Optimization (PPO)
2. implementation of Prioritized Experience Replay.
3. other ideas from DDDPG paper: Barth-Maron *et al, Distributed Distributional Deterministic Policy Gradients*, https://arxiv.org/abs/1804.08617.

Conclusions

The most important conclusions and my personal take-aways from this project:

1. Even hundreds of first episodes may result with no learning at all, which may be extremely frustrating. But as some moment it may explode - so be patient, let server work overnight, and check in the morning...
2. For problems where multiple agents are rewarded for efficient collaboration the training time may significantly increase with the average score (skills acquired)
3. Sharing a memory buffer is very important to agents efficiency
4. Quite simple model architectures (in terms of number of layers and their dimension) seem to work pretty well
5. Neural network activation function significantly improved my agent performance
6. Ornstein-Uhlenbeck noise process, especially it's decreasing amplitude, proves to be very important for my agents learning curve.