Udacity DRLND: Continuous Control

Research Report

Andrzej Wodecki

January 28th, 2018

Introduction

This report is a summary of results of the ***Continuous Control*** **project of Udacity Deep Reinforcement Learning Nanodegree.**

The goal of the project is to **train the Agent to operate an arm with two joints** in the Reacher environment provided by Unity Environments. After training it should be able to stay within the target green zones for a longer time. This **episodic** environment is considered **solved** when agents gets an average score of +30 over 100 consecutive episodes, where a reward of +0.1 is provided for each step that the agents's arm is in the goal location.

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

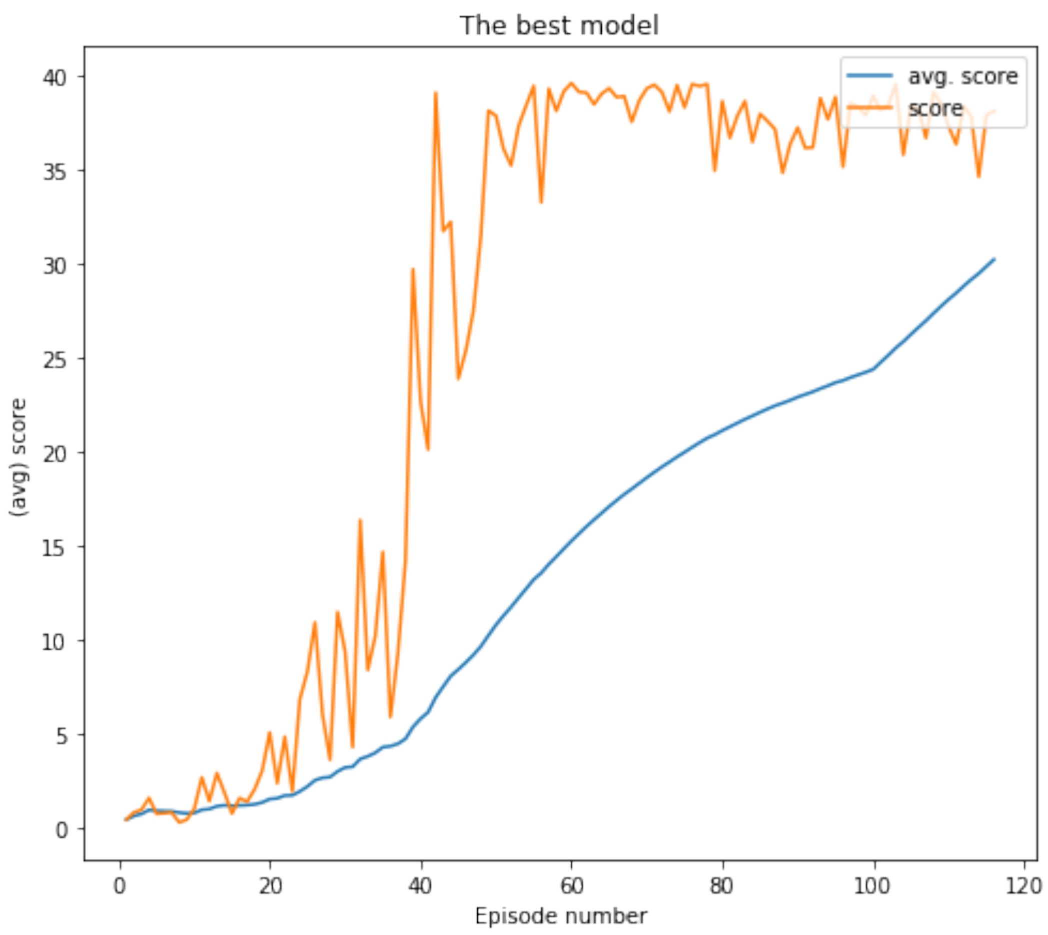
Implementation

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 with 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 16 episodes (see the figure below), and had the following hyperparameters values:



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

The goal of the project is to **train the Agent to operate an arm with two joints** in the Reacher environment provided by Unity Environments. After training it should be able to stay within the target green zones for a longer time.

In this report I address the Option 1 of the problem (only one agent acting in the environment).

The state space has 33 dimensions like the position, rotation, velocity and angular velocities of the arm.

The action space consists of 4 actions corresponding to torque applicable to two joints.

This is **episodic** environment. It is considered **solved** when agents gets an average score of +30 over 100 consecutive episodes.

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 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. Next, every time step:
    1. agent chooses the action
    2. this action is "presented" to the environment
    3. environment returns `next_state, reward` and `done` information.
    4. agents absorbs this parameters to update it's Q-network (`step` method)
    5. `state` and `score` parameters are updated, and appropriate average scores calculated.
3. Overall `ddpg` function returns the final `score` list.

A precise description of the DDPG algorithm is presented eg. in the paper by Lillicrap *at al* cited above.

Model architecture

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

1. Input layer with 33 input nodes (which is a state space size)
2. 2 hidden layers, with number of nodes beeing a parameters (in my research: 200, 300 or 400) and an activation function (I checked two options ReLU and Leaky ReLU provided by PyTorch)
3. Optionally: *batch normalization* (only for Critic) and *dropout* layers (both for Actor and Critic).
4. Output node with 4 nodes (action space size).

Hyper-parameter grid search and it's results

There are many different parameters influencing the final efficiency of the 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, gradient clipping 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
a_gradient_clipping	Whether to apply (TRUE) or not (FALSE) a gradient clipping to the Actor	boolean
c_gradient_clipping	Whether to apply (TRUE) or not (FALSE) a gradient clipping to the Critic	boolean
c_batchnorm	Whether to apply (TRUE) or not (FALSE) a batch normalization to the Critic	boolean
a_leaky	Whether to apply (TRUE) or not (FALSE) a leaky ReLU activation function to the Actor	boolean
c_leaky	Whether to apply (TRUE) or not (FALSE) a leaky ReLU activation function to the Critic	boolean
a_dropout	Whether to apply (TRUE) or not (FALSE) a dropout layer to the Actor	boolean
c_dropout	Whether to apply (TRUE) or not (FALSE) a dropout layer to the Critic	boolean

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

First experiments

After successful implementation of DDPG agent I started quite a random exploration of different combination of hyperparamater values.

In my first grid search:

1. I fixed the following hyper parameters:
    1. n_episodes = 100
    2. max_t = 1000
    3. GAMMA = 0.99
    4. TAU = 1e-3
    5. LR_ACTOR = 1e-4
    6. BATCH_SIZE = 128
2. And scanned the following ones:
    1. UPD = from 4 to 10, 4 points
    2. BUFFER_SIZE = from 1e5 to 12e5, 4 points
    3. LR_CRITIC = from 1e-4 to 1e-3, 3 points
    4. WEIGHT_DECAY = from 0 to 5e-3, 3, points

Altogether, it made 4x4x3x3 = 144 models to scan, what took few days on a moderate machine (with no GPU).

The results where not satisfactory: the best performing model achieved an avg. score (over 100 consecutive episodes) 17 points (after 100 episodes).

Thus, I decided to explore more possibilities, including:

1. Adding a gradient clipping
2. Modifying the neural network (adding an extra dropout layer and modifying the activation function)
3. Experimenting with Ornstein-Uhlenbeck noise process.

Interestingly, after many trials (but not a full grid search yet) it appeared that results significantly improved when I sampled Ornstein-Uhlenbeck noise from standard normal distribution rather than using a simple random. Keeping that in the agent model (DDPG.py), I run the 3 grid search experiments and finally solved the environment - see the description below.

Grid Search experiment set-up

Fixed parameters

After the preliminary research (basing on both literature and own tests) I decided to fix the following parameters:

TAU = 1e-3

GAMMA = 0.99

n_episodes = 400

max_t = 2000

Grid search scope

All the grid search is controlled by parameters stored in *parameters.py*. All the variables in this file (with prefix *r_* followed by the name of the parameter explained in the table above) are lists specifing the values to be search over.

To simplify the code use the parameters are organized into the following groups (the numbers after the # sign are the values recommended for the grid search):

General agent parameters

r_UPD = [1] #[1, 4, 10] r_BUFFER_SIZE = [1e5] #[1e5, 5e5, 1e6] r_BATCH_SIZE = [128, 256] #[64, 128, 256, 512]

r_fc1_units = [300] #[200, 300, 400] r_fc2_units = [400] #[200, 300, 400]

Actor parameters

r_LR_ACTOR = [1e-3] #[1e-4, 5e-4, 1e-3] r_a_gradient_clipping = [False] #[True, False] r_a_leaky = [True] #[True, False] r_a_dropout = [False] #[True, False]

Critic parameters

r_LR_CRITIC = [1e-4] #[1e-4, 5e-4, 1e-3] r_c_gradient_clipping = [False] #[True, False] r_c_batch_norm = [True] #[True, False] r_c_leaky = [True] #[True, False] r_c_dropout = [False] #[True, False]

Experiment 1

In my first experiment I decided to check the influence of **gradient clipping, batch normalization, activation function and dropout layers of the Critic on the agent efficiency.** ** All the general agent and the Actor parameters remain frozen - see the table below.

Hyperparameter	**Value(s)**
UPD	1
BUFFER_SIZE	1e6
BATCH_SIZE	64
fc1_units	200
fc2_units	300
LR_ACTOR	1e-3
a_gradient_clipping	TRUE
a_leaky	TRUE
a_dropout	TRUE
LR_CRITIC	1e-4
c_gradient_clipping	**TRUE, FALSE**
c_batch_norm	**TRUE, FALSE**
c_leaky	**TRUE, FALSE**
c_dropout	**TRUE, FALSE**

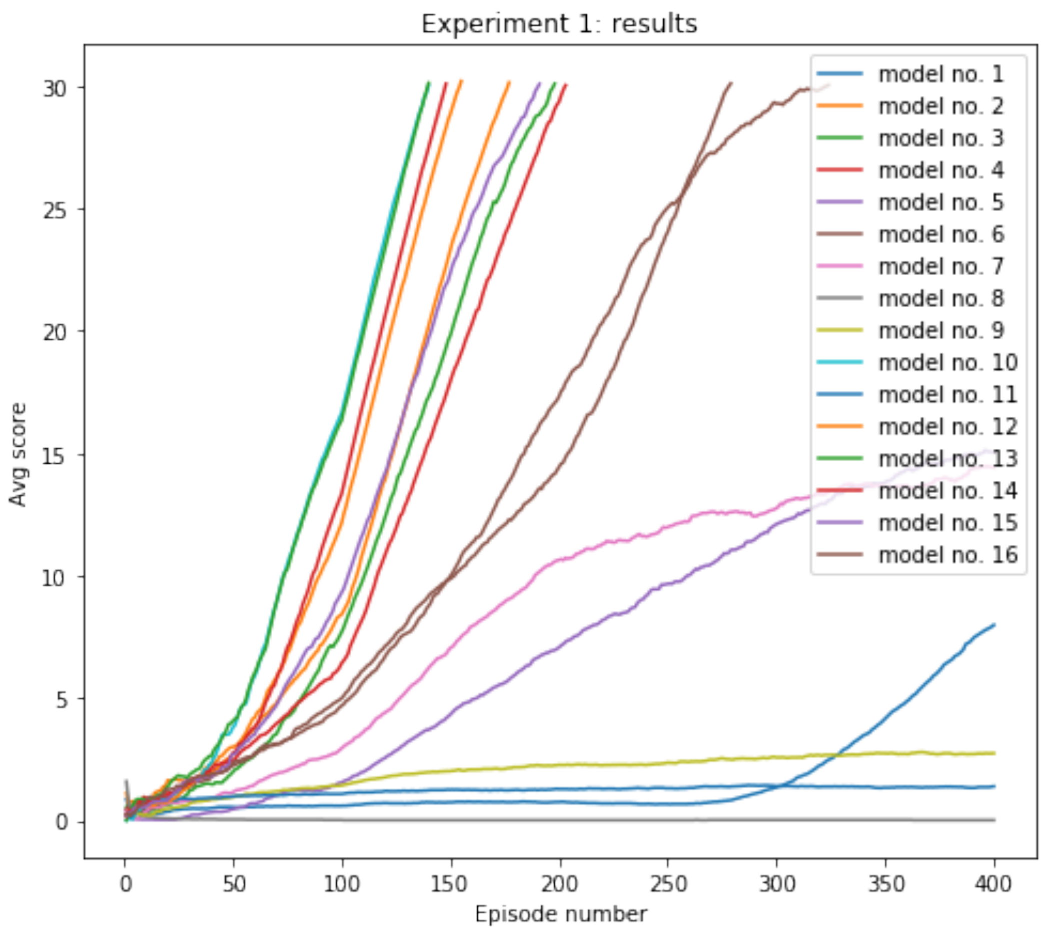
All the results are stored in the **exp1** subdirectory:

1. **Results for all the 16 models scanned: *results.txt***. In this file:
    1. first 3 columns contain:
        1. model number
        2. episode number
        3. avg. score
        4. score
    2. next columns contain all the hyperparameters in the following order: *UPD, BUFFOR_SIZE, BATCH_SIZE, fc1_units, fc2_units, LR_ACTOR, a_gradient_clip, a_leaky, a_dropout, LR_CRITIC, c_gradient_clip, c_BATCH_NORM, c_leaky, c_dropout.*
2. **The models which solved the environment**: *models* subdirectory:
    1. **successful actor and critic models** as *.pth files, where suffix corresponds to the model number
    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. **A jupyter notebook with supporting plots.**

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



The models which succesfully solved the model are:

Model number	The episode solved	c_gradient_clipping	c_batch_norm	c_leaky	c_dropout
2	77	TRUE	TRUE	TRUE	FALSE
3	98	TRUE	TRUE	FALSE	TRUE
4	103	TRUE	TRUE	FALSE	FALSE
6	179	TRUE	FALSE	TRUE	FALSE
10	40	FALSE	TRUE	TRUE	FALSE
12	55	FALSE	TRUE	FALSE	FALSE
13	40	FALSE	FALSE	TRUE	TRUE
14	48	FALSE	FALSE	TRUE	FALSE
15	91	FALSE	FALSE	FALSE	TRUE
16	224	FALSE	FALSE	FALSE	FALSE

As can be seen, the best performing model are models number 10 and 13 (where the environment was solved in 40 episodes) followed by models no. 14 and 12. Looking at their characteristics it looks like:

1. Leaky ReLU activation definitively helps (models 10, 13 and 14)
2. Batch normalization may be helpful (models 10 and 12).
3. Adding a gradient clipping to the actor doesn't improve it's efficiency (models 2, 3, 4 and 6)
4. Adding a dropout layer to the critic doesn't significantly improve it's efficiency (only 3 out of 10 winning models had this option turned on).

Experiment 2

In the second experiment I decided to study **the influence of the neural network layers dimensions on the agent efficiency** combined with **the different critic batch normalization and dropout layers combinations**. The grid search parameters (with the scanned ones in bold) are presented in the table below.

Hyperparameter	Value(s)
UPD	1
BUFFER_SIZE	1e6
BATCH_SIZE	64
fc1_units	**200, 300**
fc2_units	**300, 400**
LR_ACTOR	1e-3
a_gradient_clipping	FALSE
a_leaky	TRUE
a_dropout	FALSE
LR_CRITIC	1e-4
c_gradient_clipping	FALSE
c_batch_norm	**FALSE, TRUE**
c_leaky	TRUE
c_dropout	**TRUE, FALSE**

All the results are stored in the **exp2** subdirectory:

1. **Results for all the 16 models scanned: *results.txt***. In this file:
    1. first 3 columns contain:
        1. model number

      2. episode number
      3. avg. score
      4. score
   2. next columns contain all the hyperparameters in the following order: *UPD, BUFFOR_SIZE, BATCH_SIZE, fc1_units, fc2_units, LR_ACTOR, a_gradient_clip, a_leaky, a_dropout, LR_CRITIC, c_gradient_clip, c_BATCH_NORM, c_leaky, c_dropout.*

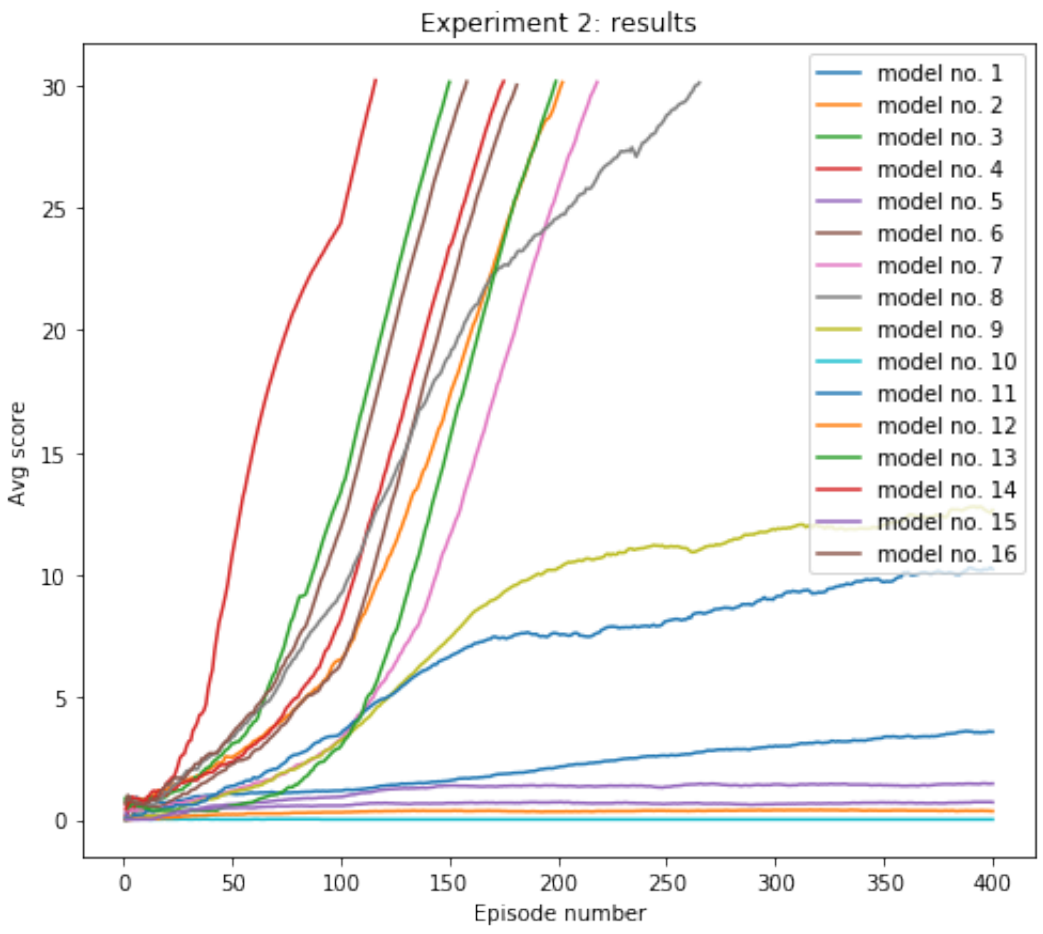
2. **The models which solved the environment**: *models* subdirectory:

   1. **successful actor and critic models** as *.pth files, where suffix corresponds to the model number

   2. **list of the models which solved the environmen**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. **A jupyter notebook with supporting plots.**

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



The models which succesfully solved the model are:

Model number	The episode solved	fc1_units	fc2_units	c_batch_norm	c_dropout
2	102	200	300	TRUE	FALSE
3	50	200	300	FALSE	TRUE
4	75	200	300	FALSE	FALSE
6	81	200	400	TRUE	FALSE
7	118	200	400	FALSE	TRUE
8	165	200	400	FALSE	FALSE
13	99	300	400	TRUE	TRUE
14	16	300	400	TRUE	FALSE
16	58	300	400	FALSE	FALSE

As can be seen, the best performing is model number 14 (where the environment was solved in 16 episodes) followed by models no. 3 and 16. Looking at their characteristics it looks like:

1. **Increasing the dimensions of internal layers in the agents model to 300 and 400 increases it's efficiency** assuming we don't apply batch normalization and an extra dropout layer (eg. model 15 with dropout and without batch normalization proved to be very poor)
2. **The added value of batch normalization and dropout layer is difficult to estimate**: there are different combination of these options in the winning hyper parameter set-up's.

Experiment 3

The goal of the third experiment was to study **the influence of the BUFFER_SIZE and the BATCH_SIZE parameters on the agent efficiency** **. The grid search parameters (with the scanned ones in bold) are presented in the table below.

Hyperparameter	Value(s)
UPD	1
BUFFER_SIZE	**1e5, 5e5, 1e6**
BATCH_SIZE	**64, 128, 256**
fc1_units	300
fc2_units	400
LR_ACTOR	1e-3
a_gradient_clipping	FALSE
a_leaky	TRUE
a_dropout	FALSE
LR_CRITIC	1e-4
c_gradient_clipping	FALSE
c_batch_norm	TRUE
c_leaky	TRUE
c_dropout	FALSE

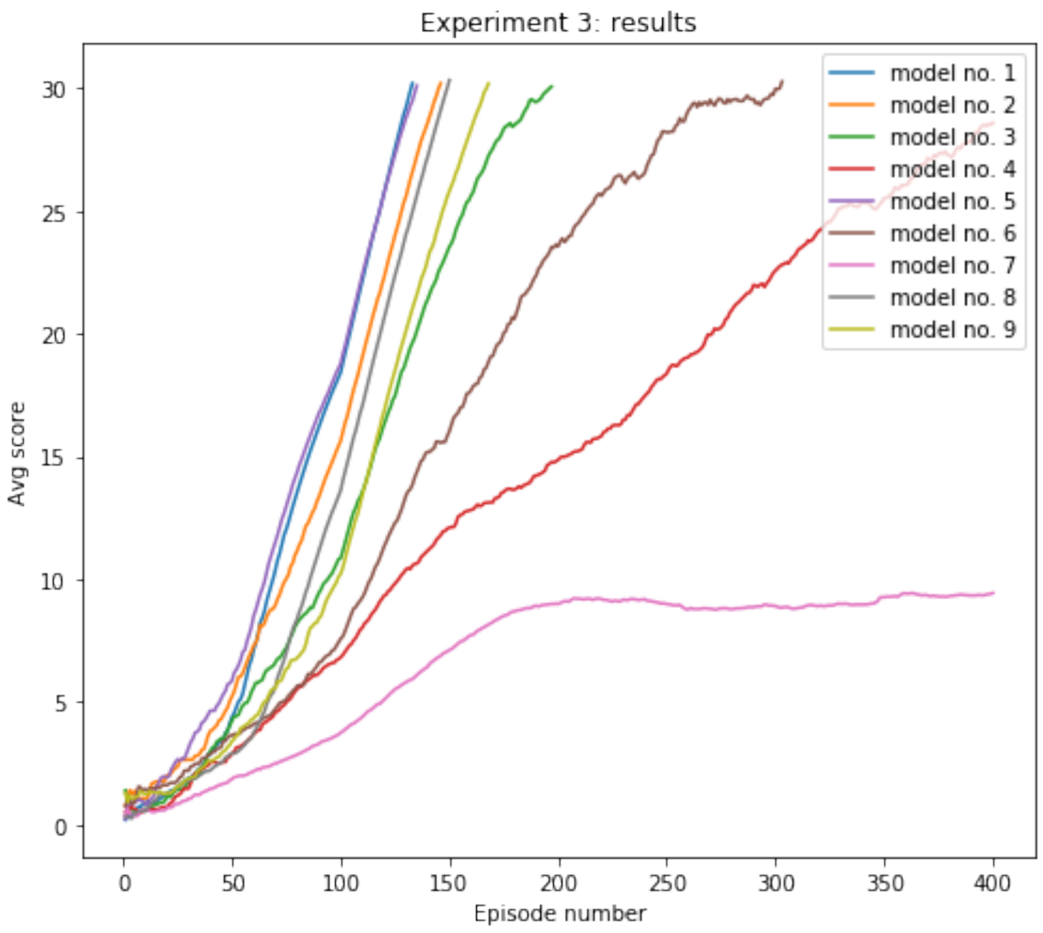
All the results are stored in the **exp3** subdirectory:

1. **Results for all the 9 models scanned: *results.txt***. In this file:
    1. first 3 columns contain:
        1. model number
        2. episode number
        3. avg. score
        4. score
    2. next columns contain all the hyperparameters in the following order: *UPD, BUFFOR_SIZE, BATCH_SIZE, fc1_units, fc2_units, LR_ACTOR, a_gradient_clip, a_leaky, a_dropout, LR_CRITIC, c_gradient_clip, c_BATCH_NORM, c_leaky, c_dropout.*
2. **The models which solved the environment**: *models* subdirectory:
    1. **successful actor and critic models** as *.pth files, where suffix corresponds to the model number
    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. **A jupyter notebook with supporting plots.**

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



The models which succesfully solved the model are:

Model number	The episode solved	BUFFER_SIZE	BATCH_SIZE
1	33	1e5	64
2	46	1e5	128
3	97	1e5	256
5	35	5e5	128
6	203	5e5	256
8	50	1e6	128
9	68	1e6	256

As can be seen, the best performing is model number 1 (where the environment was solved in 33 episodes) followed by models no. 5, 2 and 8. Looking at their characteristics it looks like **the BUFFER_SIZE and BATCH_SIZE parameters within the ranges explored don't significantly influence the agents efficiency.**

Other experiments

In my research I also investigated the influence of UPD parameter on the agents efficiency (I didn't identify any significant impact) and LR_ACTOR (the same).

Ideas for the future improvements

For the future improvements I recommend to explore:

1. other RL algorithms, eg. Distributed Distributional Deterministic Policy Gradients (D4PG) or Proximal Policy Optimization (PPO)
2. other architectures of Actor and Critic models, including:
    1. new layers
    2. different layers dimensions
    3. different activation functions
3. although exotic, for me a very temptating is also a closer look at the impact of Ornstein-Uhlenbeck noise process, eg. choosing it's different parameters or noise distributions.

Conclusions

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

1. Quite simple model architectures (in terms of number of layers and their dimension) seem to work pretty well
2. Neural network activation function significantly improved my agent performance
3. Ornstein-Uhlenbeck noise process proved to be quite important for my agents learning curve.