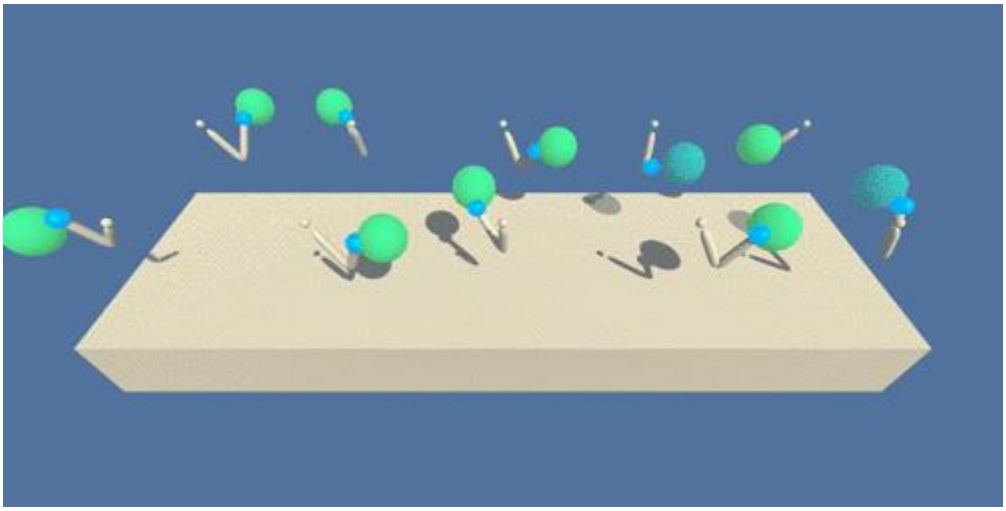
Deep Reinforcement Learning Nanodegree

Project 2- Continuous Control – Reacher.

REPORT



Introduction

This report provides a description of the implementation for the Deep Reinforcement Learning Nanodegree Project 2 to address the Continuous Control problem. In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible. Please refer to the README.md on this repository for more information regarding this environment and its installation.

Learning Algorithm

1. The Agent

Following the suggestions on section 7 of the Project lecture ("Not sure where to start?") I decided to master the details of the Deep Deterministic Policy Gradients (DDPG) algorithm and use Udacity's DDPG implementations (for the OpenAI Gym Pendulum and for the OpenAI gym BipedalWalker environment problems)

I adapted the code to solve the Reacher problem with a single agent. An endless series of questions arose while I was learning and adapting the code. Let me address this questions since are relevant in my learning process:

The Model Architecture Problem

I started with the architecture presented in Udacity's DDPG implementation with 3 hidden layers for both the Actor and the Critic. Soon after a few long training sessions I knew that there are many hyper-parameters that may be causing the learning to be extremely "slow". First I realized that I needed to formally define some performance parameters: What is "slow"?, i.e. I had no performance baseline to compare to. I knew that I needed to pass 30 points rewards average over 100 consecutive episodes…but should I expect to reach this level at 200 episodes, 2,000 episodes, 20,000 episodes? Besides that, how long are the episodes supposed to be? Should we use a "Max steps" per episode parameter or should we finish the episode when the environment returns "done"? All these questions where piling up in my mind and I decided to press "pause" and start with the very basic question (usually associated to "Beginners"): How deep should my network be? i.e. How many Hidden layers and how many Neurons per layer should I have to solve this problem? After dedicating a good amount of time reading on this subject, I decided to simply test 2 architectures:

1. Model 1: with **3 hidden** layers and test 3 distinctive number of neurons per hidden layer: 48, 96, 256 (Why powers of 2? – Not really any especial reason, but I if I were to use Convolutions, I know is simply computationally a better choice)

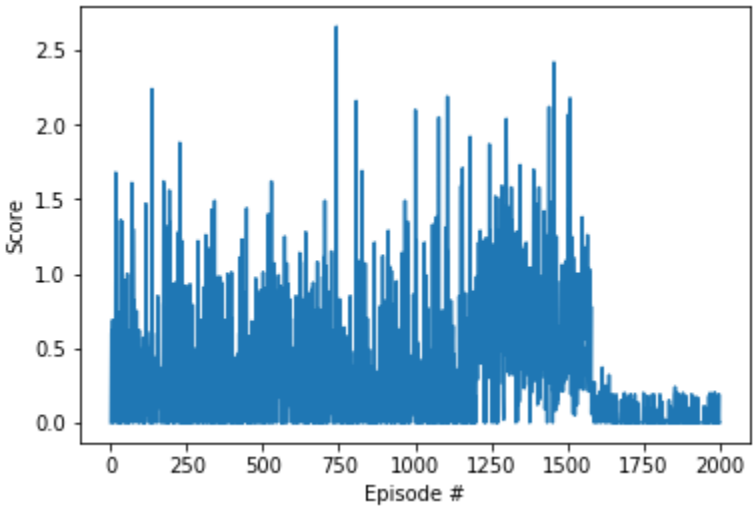
2. Model 2: With **4 hidden** layers and again testing it with 3 distinctive number of neurons per hidden layer: 48, 96, 256

From these initial tests I noticed that the model with 3 hidden layers (originally found on Udacity's implementations) was learning faster and keeping a higher average of rewards but still **none of its variants (48, 96, 256 neurons per hidden layer) solved the problem**. Therefore, the only answer to this must be on the problem parameters; either the Net hyper-parameters or the other setup parameters.

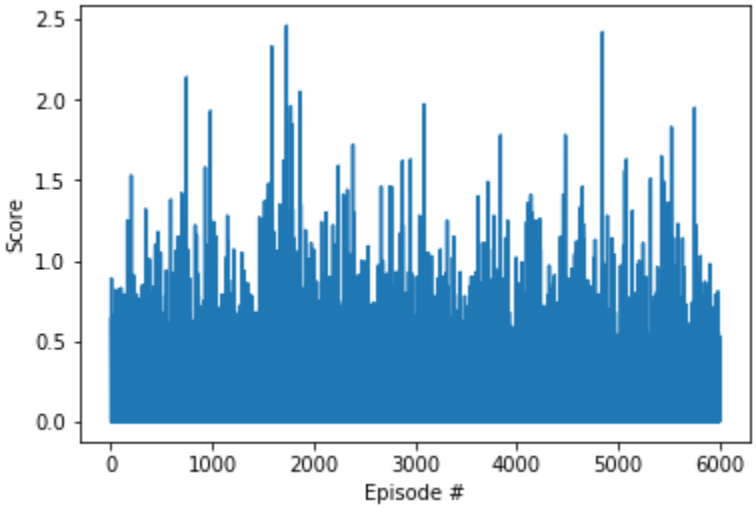
After so many unsuccessful trials using 1 single agent I run out of ideas and decide to study other implementations of the same problem to learn about my shortcomings. I downloaded some implementations that reported passing in less than 200 episodes. I tried, unsuccessfully to replicate the results. I contacted the authors and I learned that they trained using the multi-agent approach. Quickly after replicating the results I realized that using a single agent, a "good" implementation will take almost 3000 episodes. Most importantly I realized that my implementation had all the ingredients for success and only

had a small but “powerful” difference: Adams Optimizer epsilon! My implementation was using a value different that the default value (10e-8) and never really learned.

Below is the clear evidence of these unsuccessful tries. The first one is using a single agent approach, running on a 3-layer network with 256 neurons per layer, running over 2000 episodes and reaching an average (last 100 episodes) of 0.04 on the last episode. (simply terrible!)



The next one, also terrible, was using the same network and changing the length of the memory buffer required to start learning and letting it run for 6,000 episodes. At that point, I was trying to get a feeling of whether the problem needed more time to be solved or if the architecture was not appropriate.



After using the Adam's optimizer default epsilon value, things started to look much better and now I just needed to see how to improve performance.

I re-trained with the models shown above (3 & 4 hidden layers) to re-evaluate and make sure that 4 layers were not needed; and decided to stick with a 3 layers Neural Net.

I also implemented a Prioritized Experience Replay Memory used on my project 1 ("Rainbow" implementation for DQN) and didn't manage to significantly improve the performance (number of episodes required to keep a 30 points reward on 100 consecutive episodes). For more details refer to the paragraph below that addresses this issue.

Additional Model Changes

ReLu Vs LeakyReLu: Leaky ReLU has a small slope for negative values, instead of altogether zero as ReLu does. The reason I wanted to try LeakyReLu was because it has two benefits over ReLu:

- It fixes the "dying ReLU" problem, as it doesn't have zero-slope parts.
- It speeds up training. There is evidence that having the "mean activation" be close to 0 makes training faster. (It helps keep off-diagonal entries of the Fisher information matrix small, but you can safely ignore this.) Unlike ReLU, leaky ReLU is more "balanced," and may therefore learn faster.

In this case, LeakyReLu did NOT show evidences of faster learning, although is possible to think that a different combination of my hyper-parameters could actually bring up LeakyReLu as a better choice.

Batch-normalization: The use of Batch normalization is a fascinating topic that started with me reading the paper that introduced it (https://arxiv.org/abs/1502.03167) Its benefits made clear sense after reading it but questions like: Should I apply Batch-Norm to just the input layer or to every input of every layer? Would it really make a difference? Researching on this topic was interesting and sometimes frustrating since I had no other choice but to perform a few tries…hoping that the differences observed were not going to be simply associated to the initial randomness of the weights. Batch-Normalization applied to every layer's input showed the best performance. For details, please refer to model.py and model2.py files.

Gradient Clipping on the Critic Network: As it was clearly explained on the lectures, during 'Training' of a Deep Learning Model, we back-propagate our Gradients through the Network's layers. Sometimes these gradients (tangent of the slopes) might be very large causing an overflow. Gradient clipping will 'clip' the gradients or cap them to a

threshold value to prevent the gradients from getting too large. On the DDPG case and since the Critic Network uses the Actor's best action approximation for training the Value/Critic Network, Clipping is only necessary on the Critic Net.

2. Experience Replay Vs Prioritized Experience Replay

The replay memory is a critical element on the DDPG implementation. Whether using 1 or 20 agents, the replay memory was implemented "outside" the agent's class allowing 1 or 20 agent's experiences to be "memorized" together and later shared in the "Learning" process with the other agents (in the case of more than 1). I wanted to test how Prioritized Experience Replay (PER) could improve my performance. It took me by surprise to see no improvement and even damaging the baseline performance. A quick look at this issue lead me to think that PER (https://arxiv.org/abs/1511.05952) uses the probability of a transition to occur, i.e. the number of times a transition occurs with respect the total number of transitions to calculate their "priority". In a continuous action space, transitions can mathematically be infinite leading to priorities being totally normalized. At this point, when all memories (recorded transitions) have the same probability, sampling becomes exactly the same as in the basic Experience Replay, where we randomly select K samples. It seems only reasonable that prioritizing these memories using a different criteria of "importance" (more applicable to our case) might speed up the learning and even improve final performance. It make sense to think that using the "immediate" reward from every transaction might lead us to use and repeat the actions that lead us to a better reward. This is something I'd like to try in the future since time constraints to finish this assignment are not allowing me to do so now.

3. Other parameters of the DDPG algorithm

Looking at "Class **args"** (defined on Continuous_Control.ipynb) we can now focus and discuss how these arguments/parameters are used and how they affect the algorithm performance. Let me start by simply listing them below:

- seed = 777 Random seed
- disable_cuda = False Disable CUDA
- device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
- T_max = int(1e3) Number of training steps
- max_num_episodes = int(500) Max number of episodes
- hidden_1_size = 96 Network hidden layer 1 size
- hidden_2_size = 96 Network hidden layer 2 size
- hidden_3_size = 96 Network hidden layer 3 size
- noise_std = 0.05 Initial standard deviation of noise added to weights

- memory_capacity = int(1e6) Experience replay memory capacity
- batch_size = 256*1 Batch size: Number of memories will be sampled to learn
- learning_starts_ratio = 1/75 Number of steps before starting training => mem_capacity * ratio
- learning_frequency = 2 Steps before we sample from the replay Memory again
- priority_exponent = 0.5 Prioritized experience replay exponent (originally denoted α)
- priority_weight = 0.4 Initial prioritized experience replay importance sampling weight
- discount = 0.99 Discount factor
- target_update = int(30) # of steps after which to update target network
- tau = 1e-3 Soft Update interpolation parameter
- reward_clip = 1 Reward clipping (0 to disable)
- lr_Actor = 1e-3 Learning rate - Actor
- lr_Critic = 1e-3 Learning rate - Critic
- adam_eps = 1e-08 Adam epsilon (Used for both Networks)
- weight_decay = 0 Critic Optimizer Weight Decay

To keep the discussion short, and even though all parameters are relevant, I will only comment on those ones that I believe have a deeper impact on the final performance.

```
T_max = int(1e3)                    # Number of training steps\n"
```

This parameter is definitely not that important, but I wanted to comment that not only is a good practice to limit the number of training steps taken on each episode to basically prevent it from getting into an infinite loop, but also it is interesting to see that in many "Toy Challenges" we need to use the "done" bit to be able to "reset" our environment in the case that "done" meant that we failed the task (died, fell, drove outside the road, etc..). In this case, I'm not sure what the "done" bit really means. I'm not sure if is simply making sure the episodes have a pre-defined length or is considered done when we receive a +0.1 reward and we are successfully touching the ball. So, for this reason I moved from: "while not np.any(dones):" to "while timestep<=T_Max:" in the Notebook code to make sure all episodes are consistently containing the same number of time steps/transitions. In fact, using the "dones" lead to incredible similar results than using a fixed number of time steps per episode. See results bellow.

First, every episode finishes when we receive the a "done" bit from any agent

Episode 100 last 100 avg: 5.43 avg score: 16.41

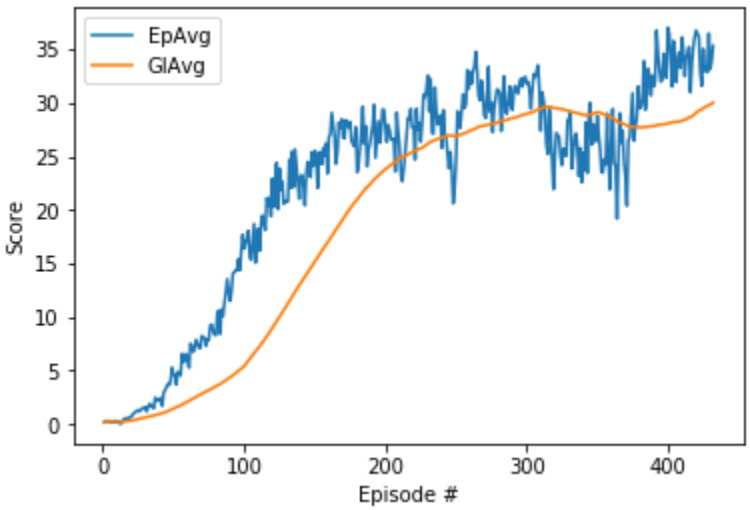
Episode 200 last 100 avg: 23.83 avg score: 27.82

Episode 300 last 100 avg: 28.97 avg score: 31.82

Episode 400 last 100 avg: 28.08 avg score: 37.04

Episode 432 last 100 avg: 30.03 avg score: 35.34

Solved in 432 episodes! last100scores_average: 30.03, time taken(min): 47.66170169512431



And in this second one, the episodes are finished when we precisely take 1,000 steps

Episode 100 last 100 avg: 5.43 avg score: 16.41

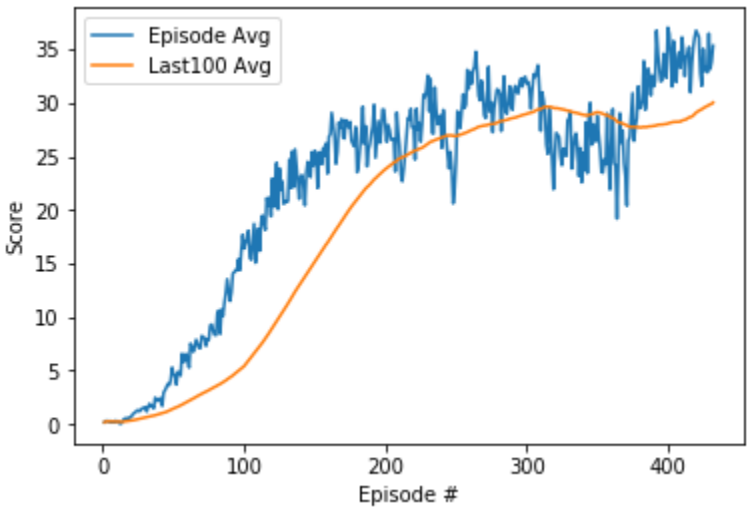
Episode 200 last 100 avg: 23.83 avg score: 27.82

Episode 300 last 100 avg: 28.97 avg score: 31.82

Episode 400 last 100 avg: 28.08 avg score: 37.04

Episode 432 last 100 avg: 30.03 avg score: 35.34

Solved in 432 episodes!last100scores_average: 30.03, time taken(min): 42.56376190980276




```
hidden_1_size = 96                # Network hidden layer 1 size\n",
```

```
hidden_2_size = 96              # Network hidden layer 2 size\n",
hidden_3_size = 96              # Network hidden layer 3 size\n",
```

As mentioned before, these define the number of Neurons used in each of the 3 layers I finally used. 96 Neuron showed to be enough while 48 neurons were not enough to solve the task.

```
memory_capacity = int(1e6)      # Experience replay memory capacity\n",
batch_size = 256*1              # Batch size: Number of memories will be
sampled to learn"
learning_starts_ratio = 1/5     # Number of steps before starting training =
memory capacity * this ratio\n",
learning_frequency = 20         # Steps before we sample from the replay

Memory again    \n",
```

These are all Replay Memory/buffer related parameters. The "`learning_starts_ratio`" defines the number of transitions/steps that needed to be stored in memory before we start "learning". The "`learning_frequency`" defines the number of steps/transitions that we will have to go through (add to memory) after the condition to start sampling (learning) has been satisfied. In this way, we avoid sampling every timestep after this condition has been satisfied.

These parameters and their "proper" combination proved to be an important factor from the performance point of view. An art that took some time and testing to adjust.

For instance, with:

memory_capacity = int(1e6)

batch_size = 256

learning_starts_ratio = 1/950

learning_frequency = 256

the results are terrible!!

Episode 100 last 100 avg: 0.52 avg score: 0.92

Episode 200 last 100 avg: 0.31 avg score: 0.08

Episode 300 last 100 avg: 0.05 avg score: 0.02

Episode 400 last 100 avg: 0.04 avg score: 0.02

Episode 419 last 100 avg: 0.04 avg score: 0.03

And with

memory_capacity = int(1e6)

batch_size = 256*2

learning_starts_ratio = 1/3

learning_frequency = 1024*10

The results are not any better as you can see below

Episode 100 last 100 avg: 0.25 avg score: 0.10

Episode 200 last 100 avg: 0.22 avg score: 0.32

Episode 300 last 100 avg: 0.24 avg score: 0.31

Episode 400 last 100 avg: 0.24 avg score: 0.16

Episode 486 last 100 avg: 0.24 avg score: 0.11

But why?

Well, as soon as the memory has about twice the batch size (512 transitions or steps) we start to sample and train our networks...and update them. We wait until we introduce another 256 memories and we do sample again 256 memories. This means we are starting to train our networks too soon with very low probabilities that any "good" transitions had happen. The vicious circle closes when the ill updated networks make decisions that are as good as a random motion and we keep sampling and updating for this aweful behavior that the agent never recovers from.

Let's thing about what would be ideal. We don't want to wait too long, but at least long enough that experiences collected have a enough successful rewards so when we sample from them we are likely to learn what moves drove us to success and how these moves might have been related to the status of the environment (where we were and where the ball was when taking that move lead us to a good reward)

We also have to take into account that if we fix the length of each episode to be 1,000 steps, then if we set the learning_starts_ratio = 1 / 4 that means that we will NOT start learning until we have accumulated 1 / 4 of the total memory capacity which is about 250k steps. That ultimately means that the learning will not start until we are gone through 250 episodes!

To see this, let's try with the following values:

memory_capacity = int(1e6)

batch_size = 256

learning_starts_ratio = 1/6 (It will take 166K time steps i.e. 166 episodes before we start learning)

learning_frequency = 10

And the results are much better as we expected

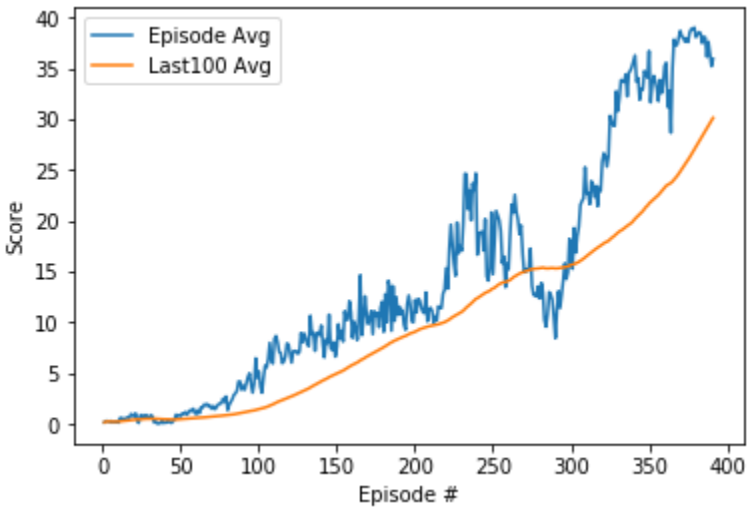
Episode 100 last 100 avg: 1.48 avg score: 5.18

Episode 200 last 100 avg: 9.09 avg score: 12.21

Episode 300 last 100 avg: 15.65 avg score: 16.92

Episode 391 last 100 avg: 30.14 avg score: 35.94

Solved in 391 episodes!last100scores_average: 30.14, time taken(min): 29.698861376444498



One more case shows that

memory_capacity = int(1e6)

batch_size = 512

learning_starts_ratio = 1/10 (It will take 100K time steps i.e. 100 episodes before we start learning)

learning_frequency = 20

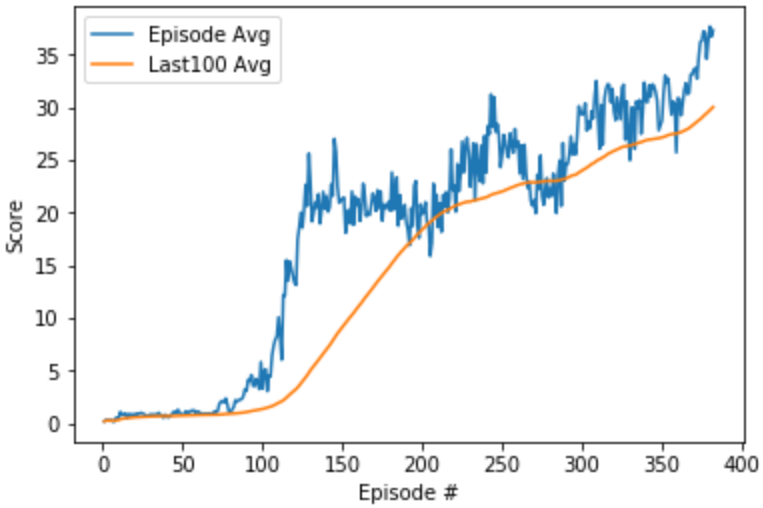
Episode 100 last 100 avg: 1.36 avg score: 3.30

Episode 200 last 100 avg: 18.38 avg score: 19.86

Episode 300 last 100 avg: 23.99 avg score: 29.27

Episode 382 last 100 avg: 30.01 avg score: 37.29

Solved in 382 episodes!last100scores_average: 30.01, time taken(min): 27.682628881931304



It looks like starting to learn earlier by using "learning_starts_ratio" helps to improves the performance of the solution, but it does have a limit as we saw before.

If we try with "learning_starts_ratio = 1 /100" and we increase the batch size as in:

memory_capacity = int(1e6)

batch_size = 256 * 3

learning_starts_ratio = **1/100** (It will take 10K time steps i.e. 10 episodes before we start learning)

learning_frequency = 20

We see a result that already is worst

Episode 100 last 100 avg: 1.41 avg score: 4.61

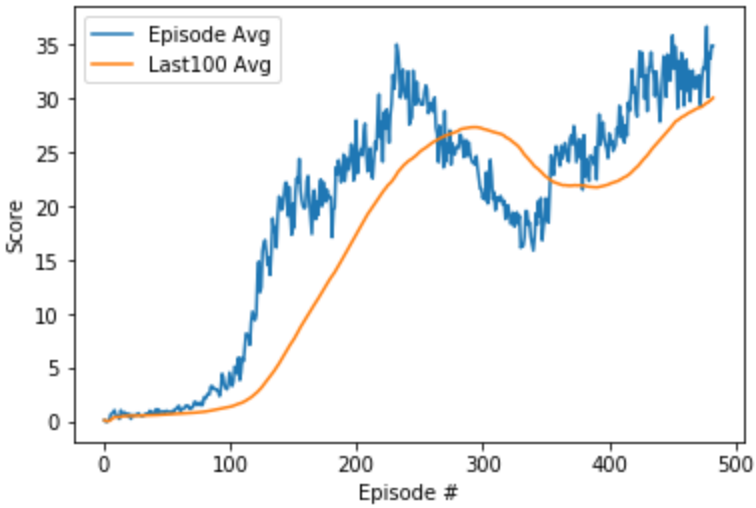
Episode 200 last 100 avg: 17.43 avg score: 28.00

Episode 300 last 100 avg: 27.21 avg score: 20.77

Episode 400 last 100 avg: 22.10 avg score: 26.78

Episode 482 last 100 avg: 30.09 avg score: 34.90

Solved in 482 episodes!last100scores_average: 30.09, time taken(min): 57.02300565640132



After some more testing we get close to an optimal combination of parameters such:

memory_capacity = int(1e6)

batch_size = 256

learning_starts_ratio = **1/50** (It will take 20K time steps i.e. 20 episodes before we start learning)

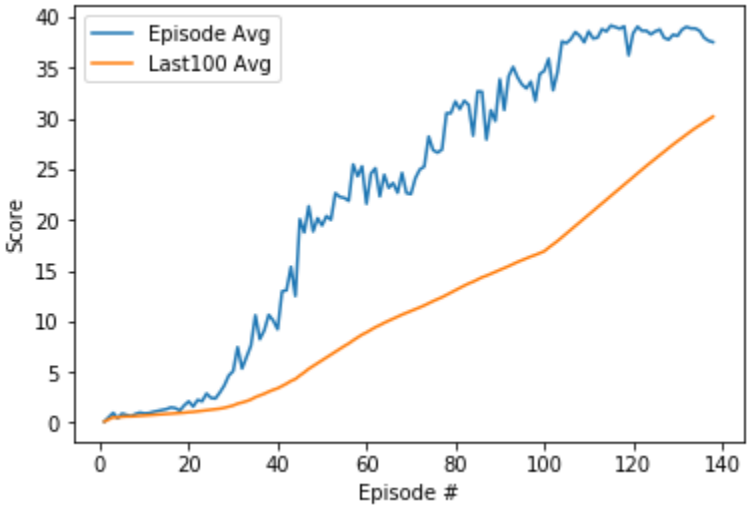
learning_frequency = 2

The result is Good enough to stop the testing as you can see below

Episode 100 last 100 avg: 16.92 avg score: 34.71

Episode 138 last 100 avg: 30.20 avg score: 37.55

Solved in 138 episodes!last100scores_average: 30.20, time taken(min): 29.39831927617391



```
target_update = int(1)          # Number of steps after which to update
target network (Soft update for Actor & Critic)\n",
```

```
tau = 1e-3                      # Soft Update interpolation parameter\n",
```

These 2 parameters are related to when (as in how often) and how do we update the 4 Networks: The current and target networks for each the Actor and the Critic. In my last modifications I decided to make a soft update every time we passed all the conditions to “learn”

Ideas for Future Work

- Implement other algorithms to be able to compare performances.
- Implement “Priority Replay Memory” with a “priority” criteria that fits this example and compare performances with normal (random sample) replay memory
- Investigate further the effects of batch size, learning_starts_ratio and learning_frequency parameters described above.
-