Deep Reinforcement Learning Agent for Tennis Environment

Daniel Szemerey

Udacity

Author Note

This project is an obligatory delivery for Project III in

Udacity's Deep Reinforcement Learning Nanodegree

Abstract

This report summarizes the work done on Project III of the Deep Reinforcement Learning Nanodegree.

An agent utilizing Deep Deterministic Policy Gradient (DDPG) was developed and tuned to solve a multi agent Tennis environment, which runs on the unity engine and was provided as a task by Udacity

The DDPG agent was able to solve the environment under 2000 episodes and reached a maximum overall score averaged over 100 episode of +2.05 by episode 2655.

To speed up the hyperparameter tuning phase, an abstract training loop was utilized, that could calculate permutations of hyperparameters during evening hours when electricity cost are lower.

Keywords: Artificial Intelligence, Reinforcement Learning, DDPG, Actor-Critic, Multi-Agent

Table of Contents

Environment

Goal and Reward: Hit the tennis ball over the fence. Interaction is continuous, but training loop was constrained to 1000 steps per episode to keep need training resources constrained.

state space:	**8** (position and velocity of ball and racket)
action space:	**2** (continuous, both between -1 and 1)
agents (brains):	**2**
considered solved:	**> +0.5** avg. over 100 episodes
termination criteria:	**1000** time steps (set by user)
reward:	**+0.1** if agent hits the ball over the net, **- 0.01** if ball falls on agent's side

Source of the environment: Udacity - Deep Reinforcement Learning engine: unityagents

Agent with Deep Deterministic Policy Gradient

To solve the environment an Agent leveraging Deep Deterministic Policy Gradients has been chosen. This network is considered by many to be an Actor-Critic method. In the learning step the agent selects a next step to calculate the Temporal Difference (which is biased in regards of the actual value) with a Policy network (which has large variance in regards of the actual value). This way both bias and variance is decreased.

Below you can find the final network parameters that showed reliable convergence:

Actor Network Architecture

Input Layer size	8
Hidden Layer size 1	140
Hidden Layer size 2	140
Output Layer size	2 (action-size)
Activation function	ReLU
Output function	tanh

Critic Network Architecture

Input Layer size	8
Hidden Layer size 1	140
Action size (appended to hidden layer 1)	2
Hidden Layer size 2	140
Output Layer size	1
Activation function	ReLU
Output function	–

Training Hyperparameters

Input Layer size	8
TAU *(soft update of target network parameter)*	1e−4
Learning Rate *(for both Actor and Critic)*	1e−4
Target network is updated every {} episode:	1
MU *(Noise)*	0. 0
Theta *(Noise)*	0. 15
Sigma *(Noise)*	0. 05
Random seed	1

Tuning Hyperparameters

Abstracting Experiments

To optimize the available time to experiment with hyperparameters, an abstract training loop was implemented and modified. The abstraction layer was first developed by the author of the report, in collaboration with Mark Szulyovszky (another student at Deep Reinforcement Learning Udacity Nanodegree) for a side project where a custom environment was developed. This implementation needed further work to function with unity environments.

The abstract training loop consists of a configuration file, an Experiment object, the training loop generalized for different types of agents, helper and monitoring functions that save the result.

According to the configuration files the training loop generates Experiments with different Agents and Environments. These were then run during evening hours, when electricity costs are the lowest and when experimenters aren't productive.

Running Experiments

To successful train the algorithm three batches of experiment configurations were submitted. After running each batch, the experiments were analysed and new configuration files devised.

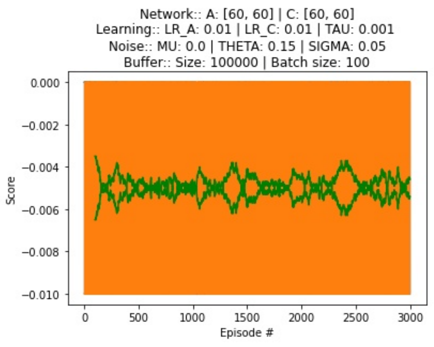
Experiment Batch 1

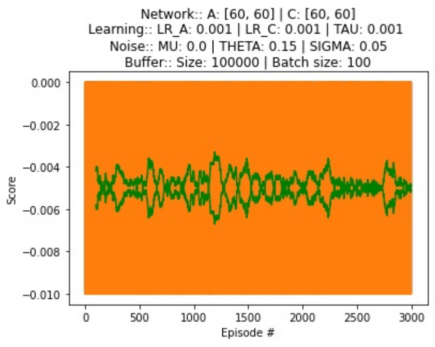
The following parameters in all permutations ($N_{Experiments} = 15$) were submitted as experiments:

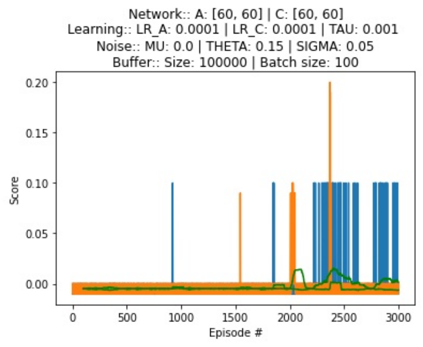
- 3 different types of learning rates: 0.01, 0.001, 0.0001

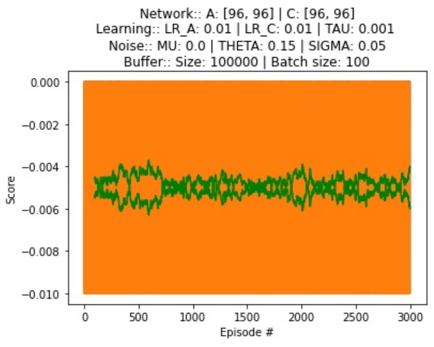
- 5 different layer sizes ([Actor, Critic]): [60, 60], [96, 96], [120, 120], [160, 160], [240, 240]

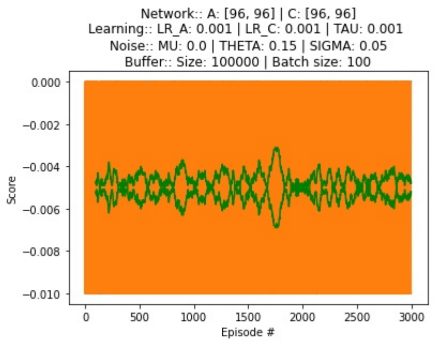
All agents were unique, meaning had separate networks and learning processes. Below you can see the results of the experiments.

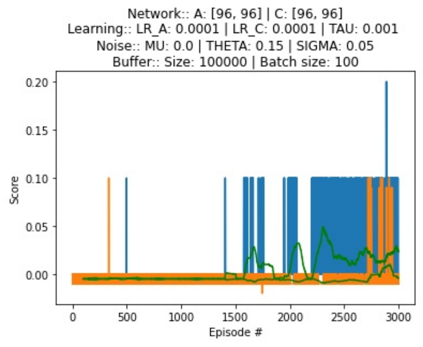

Network:: A: [60, 60] | C: [60, 60]
Learning:: LR_A: 0.01 | LR_C: 0.01 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

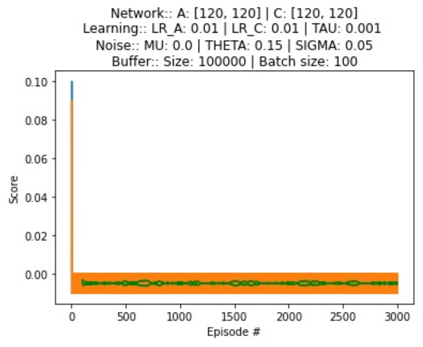

Network:: A: [60, 60] | C: [60, 60]
Learning:: LR_A: 0.001 | LR_C: 0.001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

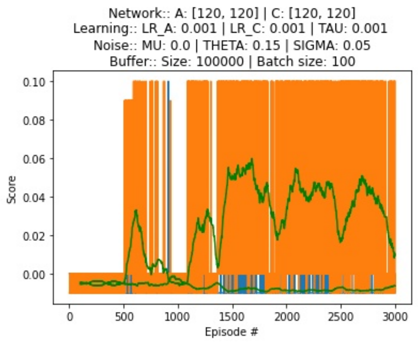

Network:: A: [60, 60] | C: [60, 60]
Learning:: LR_A: 0.0001 | LR_C: 0.0001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

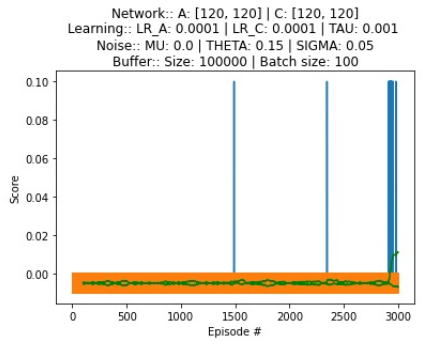

Network:: A: [96, 96] | C: [96, 96]
Learning:: LR_A: 0.01 | LR_C: 0.01 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

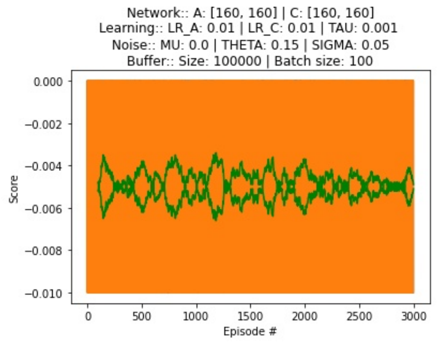

Network:: A: [96, 96] | C: [96, 96]
Learning:: LR_A: 0.001 | LR_C: 0.001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

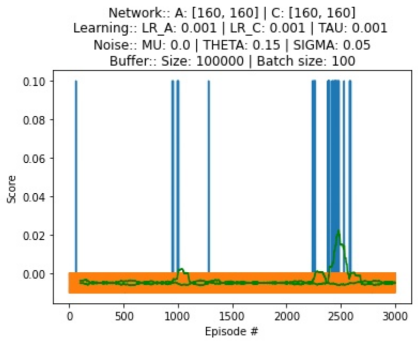

Network:: A: [96, 96] | C: [96, 96]
Learning:: LR_A: 0.0001 | LR_C: 0.0001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

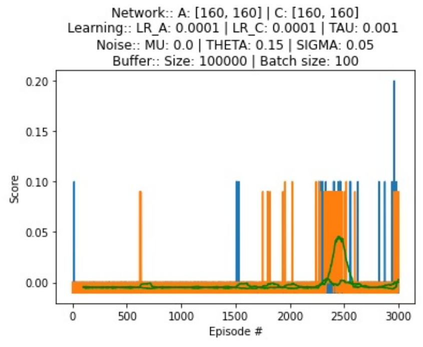

Network:: A: [120, 120] | C: [120, 120]
Learning:: LR_A: 0.01 | LR_C: 0.01 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

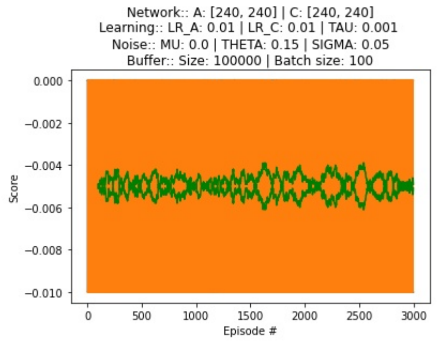

Network:: A: [120, 120] | C: [120, 120]
Learning:: LR_A: 0.001 | LR_C: 0.001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

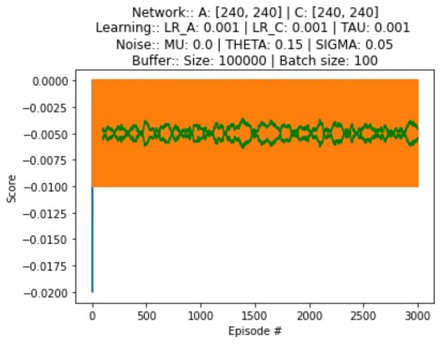

Network:: A: [120, 120] | C: [120, 120]
Learning:: LR_A: 0.0001 | LR_C: 0.0001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

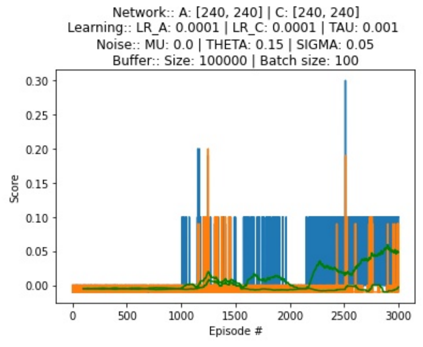

Network:: A: [160, 160] | C: [160, 160]
Learning:: LR_A: 0.01 | LR_C: 0.01 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #


Network:: A: [160, 160] | C: [160, 160]
Learning:: LR_A: 0.001 | LR_C: 0.001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #


Network:: A: [160, 160] | C: [160, 160]
Learning:: LR_A: 0.0001 | LR_C: 0.0001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #


Network:: A: [240, 240] | C: [240, 240]
Learning:: LR_A: 0.01 | LR_C: 0.01 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #


Network:: A: [240, 240] | C: [240, 240]
Learning:: LR_A: 0.001 | LR_C: 0.001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #


Network:: A: [240, 240] | C: [240, 240]
Learning:: LR_A: 0.0001 | LR_C: 0.0001 | TAU: 0.001
Noise:: MU: 0.0 | THETA: 0.15 | SIGMA: 0.05
Buffer:: Size: 100000 | Batch size: 100
Score
Episode #

The first batch of experiments showed that too large learning rates introduced too much variance and agents general performed very poorly (<0.00), with the exception of Agents with layers [160, 160] and 0.001 learning rate.

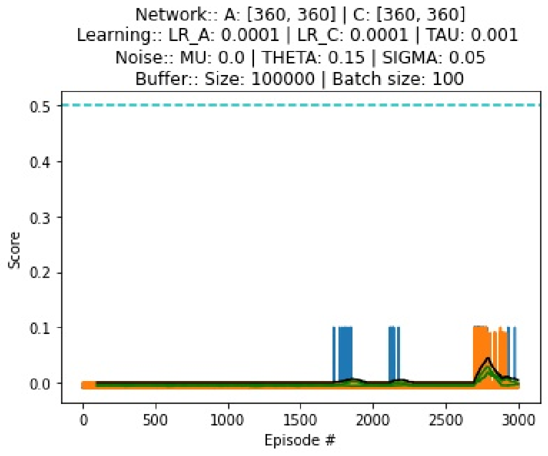
Another important aspect from the first batch of experiments was the fact that larger networks (above [100, 100]) were performing more reliably. Based on these observations a new set of experiment configurations were devised and executed.

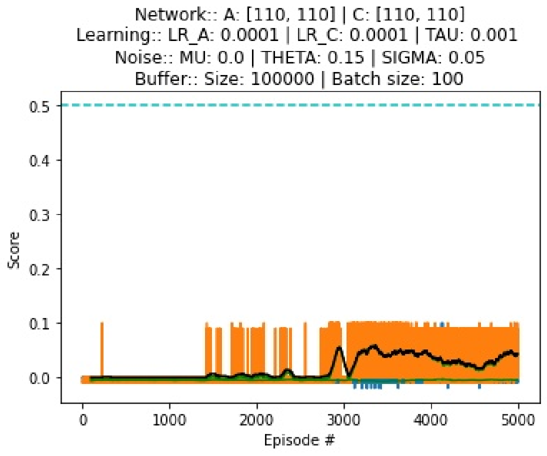
Experiment Batch 2

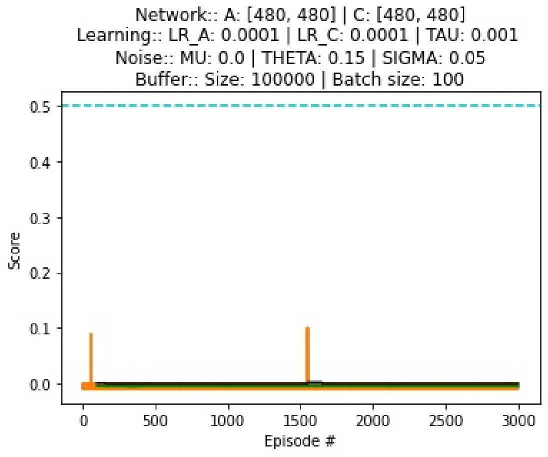
As larger networks performed better and a small learning rate converged the most reliably, the following parameters in all permutations ($N_{Experiments} = 6$) were submitted as experiments:

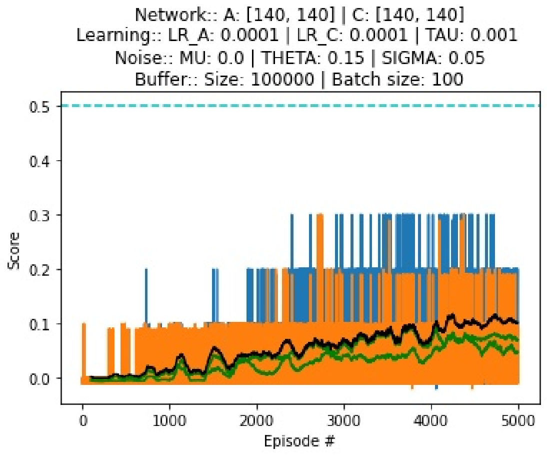
- 1 different types of learning rates: 0.0001
- 6 different layer sizes (both Actor and Critic): [110, 110], [140, 140], [196, 196], [240, 240], [360, 360], [480, 480]

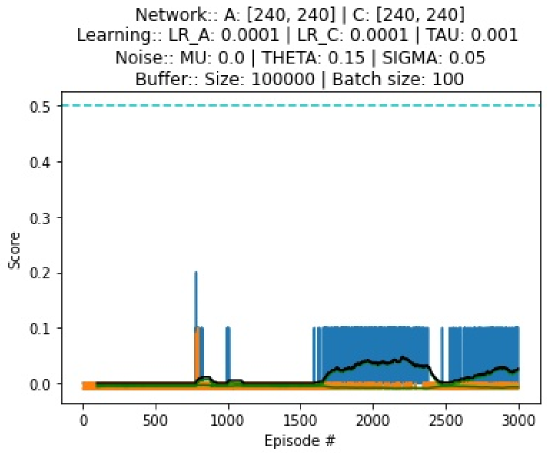
The best performance was by Agent with Actor and Critic sized as [140, 140]. All agents were unique, meaning had separate networks and learning processes. Below you can see the results of the experiments.

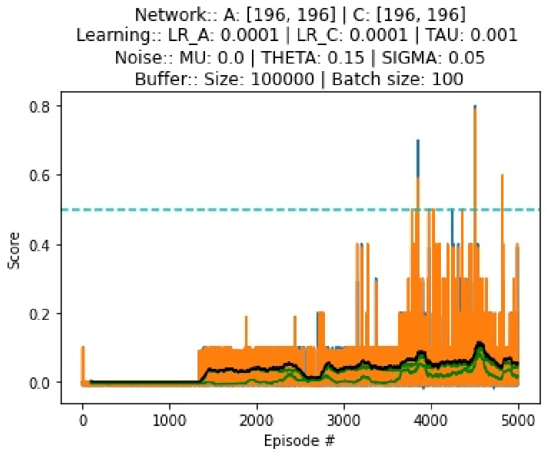












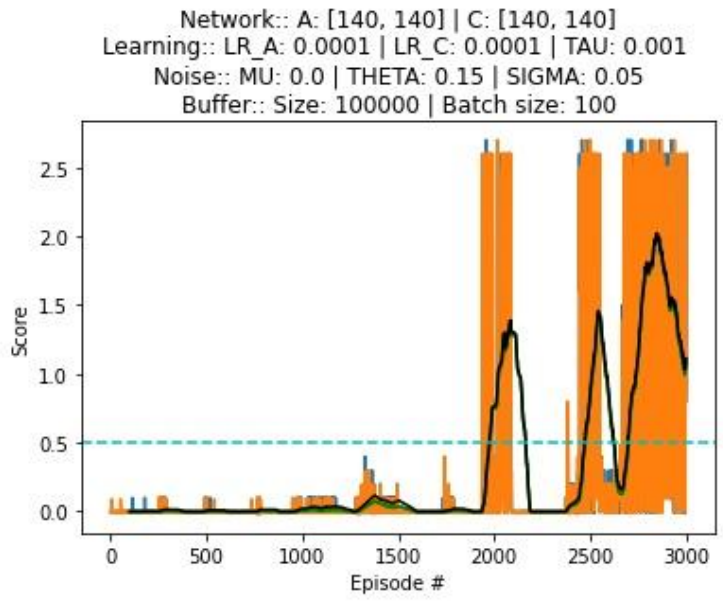
Final Agent Results

Experiment Batch 3

Taking the observations and analysis of the first two batch of experiments, it was concluded that an agent with Actor and Critic both sized [140, 140] with learning rate 0.001 could achieve better results. As a last step the experiment setup was changed, so that both agents were the same objects. This effectively resulted in the Agent playing against itself. This tweak looked reasonable as playing against a opponent learning from different experiences introduced a lot of noise. The agent also learns twice as fast as both agents experiences and learning steps are executed at each episode.

Result

Below are the final results of the Agent:



The DDPG agent was able to solve the environment under 2000 episodes and reached a maximum overall score averaged over 100 episode of +2.05 by episode 2655. Agents could achieve higher rewards per episode, if maximum allowed time steps were increased from 1000.

Future Ideas

There are several ways to further improve the algorithm:

- Modify the buffer of each agent to sample according to *Prioritized Replay*. This would add further complexity to the hyperparameters, which would need to be tuned.
- Add additional computational steps to the model. Two layers that the project could be expanded with are *batch normalization layers*, so that sample experiences are normalized to a fixed range between layers. Although the literature is not clear on this, but a dropout layer could be introduced in the beginning of the training, helping the agent generalize across the solution space when in a local optimum. The *dropout layer* can later be switched off, as overfitting is in general desirable in Deep Reinforcement Learning problems.
- Create more elaborate model architectures. Currently the *Critic* network receives the actions, by appending it to the second hidden layer. It could be interesting to feed the actions in different locations.

References

Grokking, Deep Reinforcement Learning, Miguel Morales, 2020

Dataset

Entire Project Folder with code can be found here (folder): deep-reinforcement-learning/p3_collab-compet/experiments/saved at master · szemyd/deep-reinforcement-learning (github.com)