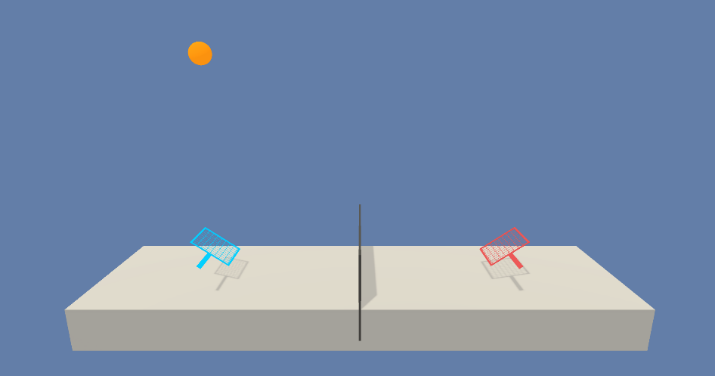
# Project 3 – Collaboration and competition

## Introduction:

The purpose of this project is to train two agents which control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

For this project, we will work with the Unity Tennis environment.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically:

* After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
* This yields a single **score** for each episode.

The environment is considered solved, when the average (over 100 episodes) of those **scores** is at least +0.5.

## Learning Algorithm:

We may consider that both agents are learning to play a game of trying to get the ball staying in the air, so it may be considered as a unique game played by two agents, instead of a sum-zero (competitive) game where both agents want the other to fail.

Based on the above premise, we will start testing with the DDPG algorithm used in the previous project (Continuous Control). Both agents will use the same neural network architecture and the **same replay buffer**.

**Poner más explicación del foro:**

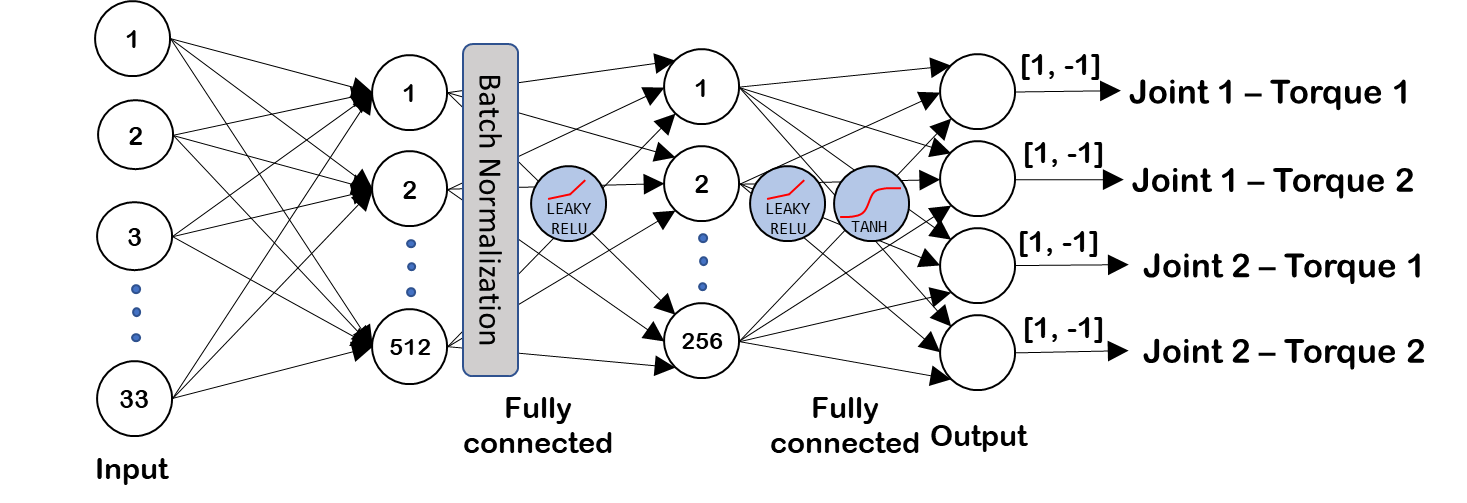
<https://knowledge.udacity.com/questions/366827>

<https://knowledge.udacity.com/questions/341942>

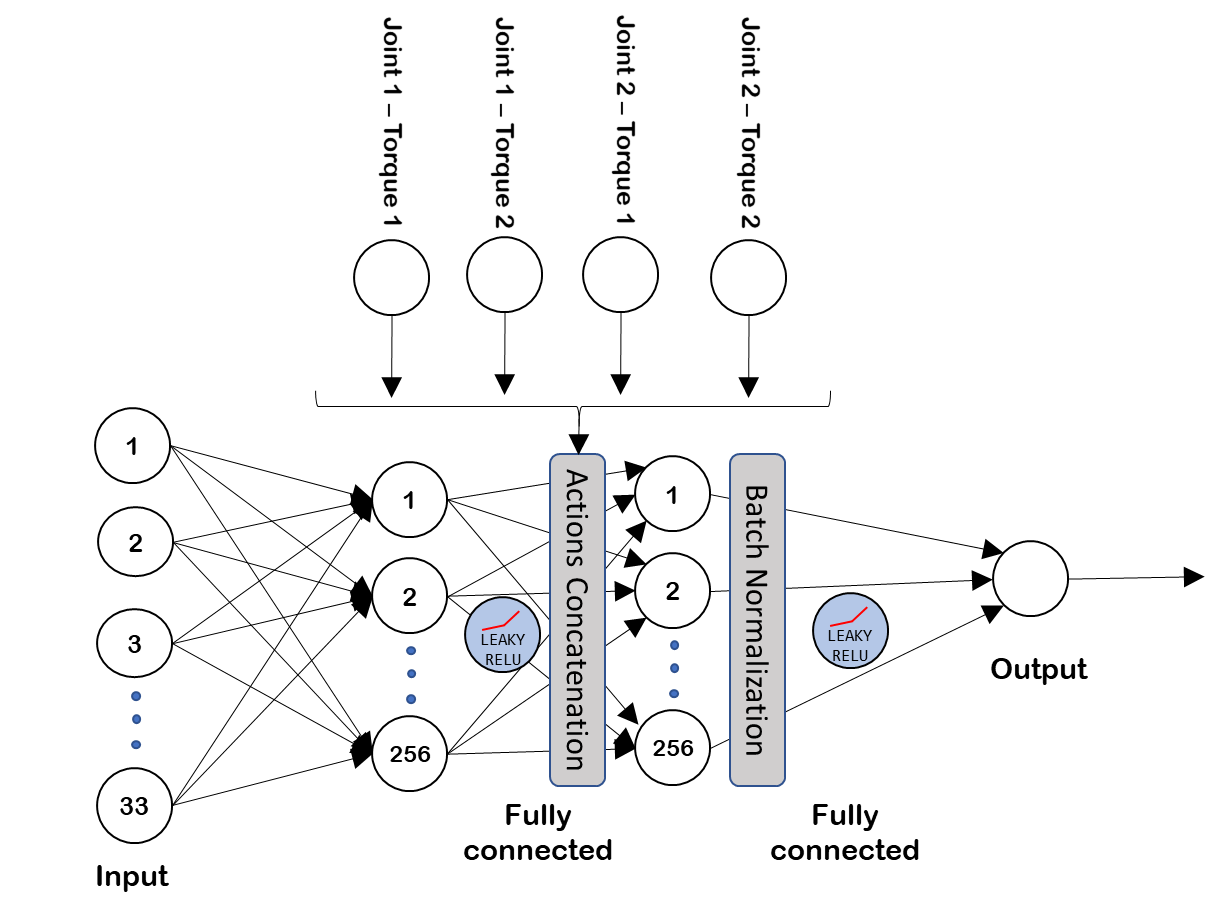
<https://knowledge.udacity.com/questions/303326>

## Actor Neural Network:

The actor NN used is the following one, the ending function used is a Tanh to be able to output continuous variables between 1 and -1:



## Critic Neural Network:



Thanks to the implementation of **experience replay**, the algorithm stores states which are rare and actions

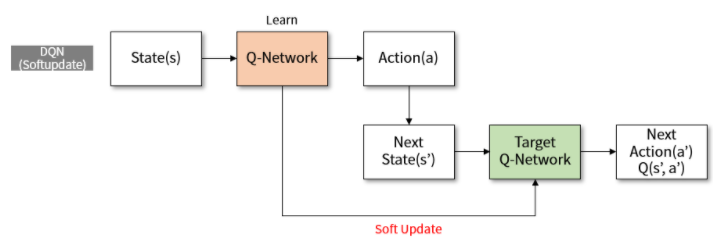
that are costly to be able to recall them, each experience (Which is formed by State, Action, Reward and Next

State) is stored in a buffer as the agent is interacting with the environment, afterwards the agent randomly samples a small batch of these experiences in order to learn from them. Thanks to this:

* it learns from individual state-actions multiple times, so it recalls rare occurrences and make a better use of the experience obtained.
* Because of the random sampling, it helps breaking the correlation and prevent actions from oscillating or diverging in wrong ways.

Another critic functionality of the DDPG algorithm is a **soft update** to the target network.

The implementation of this soft update means that the algorithm has two neural networks. The called “Regular” neural network and the “Target” neural network.



In our algorithm, this soft update is done every step, so the networks are blended with a percentage of merging considering the TAU variable, so considering the TAU variable is set to 0,05%, every step a 0,05% of the regular network will be merged with the target network.

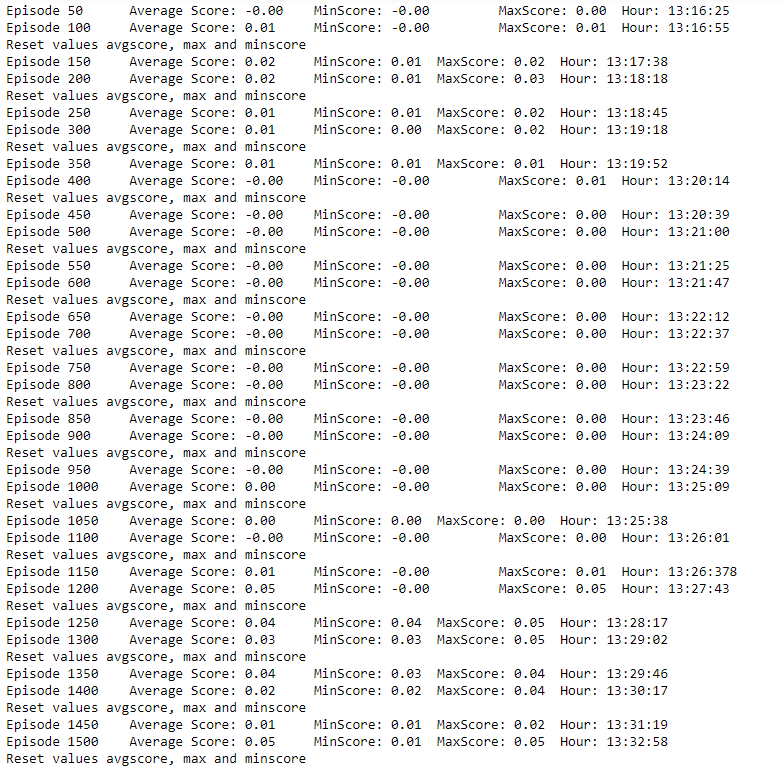
Thanks to soft update, the DDPG algorithm gets faster convergence. Soft update logic could be implemented in other off-policy algorithms that use target networks, such as DQN.

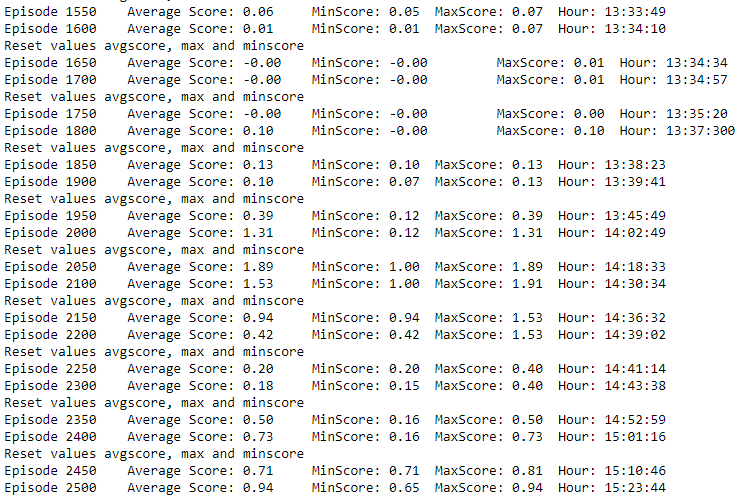
To encourage exploration in the training process, we added **noise** using the [Ornstein-Uhlenbeck](http://entsphere.com/pub/pdf/1930%20Uhlenbeck,%20on%20the%20theory%20of%20the%20Brownian%20motion.pdf) noise process, the noise is reduced through time, because when the agent is more experienced, the need for exploration decreases.

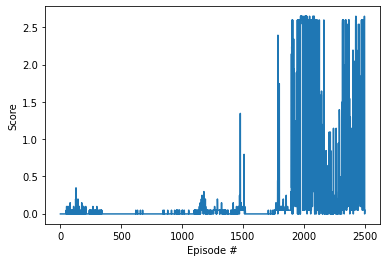
We used [**batch normalization**](https://arxiv.org/pdf/1502.03167.pdf) in both networks for stabilizing the learning process and reduce the training time for deep neural networks, batch normalization is especially useful to limit covariate shift in environments where the input distribution changes with each step, like this.

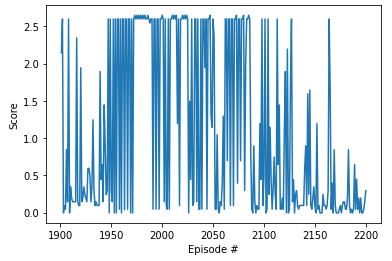
On the critic side, we have **used leaky relu** activation functions for dealing better with the possible negative values coming from the learning (*Dying ReLU problem***)** and improve training performance.

## Algorithm performance:









## Future improvements: