DEEP REINFORCEMENT LEARNING

PROJECT 2. Continuous control.

Markus Buchholz

1. GOAL

In this project, the goal was to train the 20 Agents (each Agent controls a double – joint robot arm) to maintain its position at the target location for as many time steps as possible.
The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.
Project requirement is to get average score of +30 over 100 consecutive episodes.

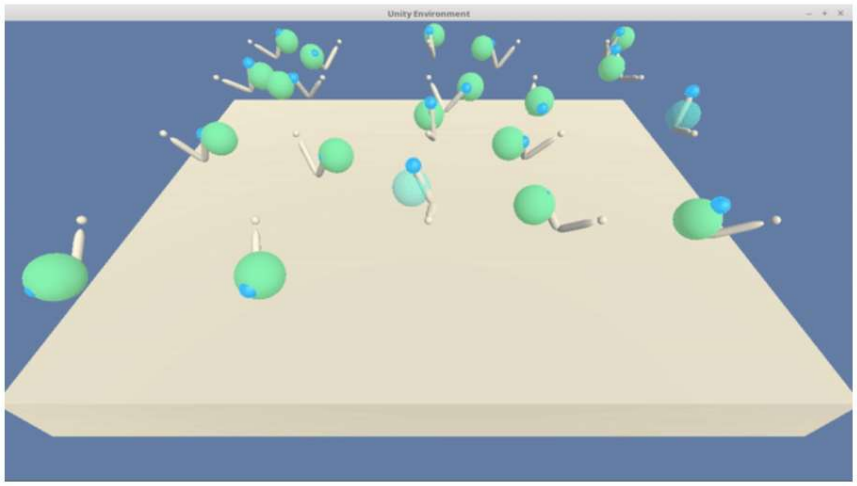

Figure 1. Continuous control Unity environment

2. INTRODUCTION

In previous project we use Q function to find the optimal policy (as the Q function) which approximates (in use of Q table and later deep neural network) selection of the best action to perform in every state. Formally, this could be formulated as,

$$\pi(s) = argmax_a Q(s,a)$$

which means that the result of our policy π at every state s is the action with the largest Q (value function).
However, in both cases, whether we used a table for small state spaces or a neural network for much larger state spaces, we had to first estimate the optimal action value function before we could tackle the optimal policy. Our Q-learning approach tried to answer the policy question indirectly via approximating the values of the states and trying to choose the best alternative. It seems that the portrayed process is rather cumbersome because as we will show later we find the optimal policy without worrying about a value function at all.

There are two main reasons why policy approach is more sufficient attitude to explore.

First of all, we care more about the total reward but not always about the highest values function (like DQN) at every state. To obtain these values (action and state), we have used the Bellman equation, which expresses the value on the current step via the values on the next step. Normally, when we must solve real, complex problems and the agent obtains the observation from the environment and needs to decide about what to do next, we need policy, not the value of the state or particular action. We need to know what to do next at each step (the action taken based on the highest value cannot be always optimal).
Another reason why policies may be more attractive than values is the environments with extreme number of actions or with a continuous action space.
To be able to decide on the best action, we need to solve an optimization problem finding action a, which maximizes Q(s, a). In the case of an Atari game with several discrete actions, such attitude wasn't a problem: we just approximated values of all actions and took the action with the largest *Q* (as argmax). If our action space is not a small discrete set, but has a scalar value attached to it, such as the steering wheel angle (continuous value which should be regulated), this optimization problem becomes hard, as *Q* is usually represented by a highly nonlinear neural network (NN), so finding the argument which maximizes the function's values can be treacherous. In such cases, it's much more feasible to avoid values function and pay more attention on policy approach.
The consecutive question comes straight forward. We need to answer how technically can we approach this idea of estimating an optimal policy?

3. POLICY GRADIENTS. Reinforce algorithm

For our learning reason we can consider cart pole environment. In this case, the Agent base on the feedback from the environment (car position, car velocity, pole angle and the pole speed at the tip) decides about the one of the possible action. The Agent can, at each time step push the cart either left or right. In order to approximate the policy, as we made that in DQN we construct a neural network, that accepts the state as input.

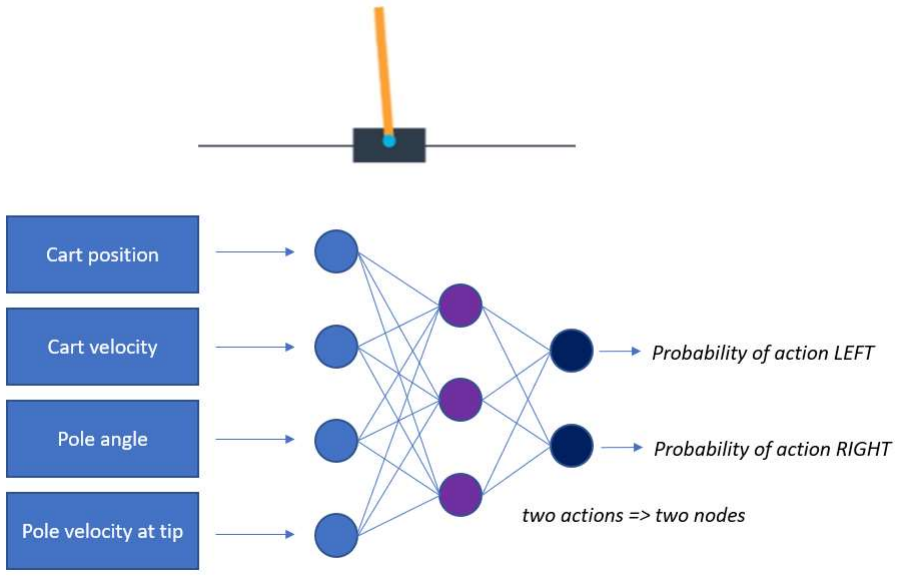

Figure 2. Neural network approximates the policy

This time however, as output, the neural network **returns the probability** that the agent selects each possible action. The agent follows this policy and interacts with the environment by just passing the most recent state to the network. Network generates the action probabilities (probability distribution) and then the Agent samples from those probabilities to select an action as response (left or right).

We need to remember that when we worked with DQN, the output of the network were Q-values, so if one of the action was converged to value of 0.3 and another action converged to value of 0.5, so due to our "strategy" argmax the action with higher value was preferred 100% of the time. Actually, we consider the **probability distribution**, so if the first action has a probability of 0.3 and the second 0.5, so the Agent can take the first action with 30% chance and the second with 50% chance.
Our objective then is to establish appropriate values for the network weights so that for each state that we pass into the neural network it returns action probabilities where the optimal action is most likely to be selected (has the highest probability). As the Agent interacts with the environment and learns more about which action is best for maximizing reward, it changes neural network weights. Changes of neural network weights happen in the rhythm of gradient update in a such a way that actions yielding high reward in a state will have a high probability and actions yielding low reward will have a low probability (transition of weights will increase the probability for the actions where the episode finishes with success or decrease the action probability where the episode finish with lost).
Approaching our solution in finding the optimal policy we need to subsequently define **trajectory** (τ), which can be expressed as a sequence of the length H (as Horizon) of the states and actions. As R(τ) we can define the reward for the trajectories we are considering.

$$\tau = (s_0, a_0, s_1, a_1, s_2, a_2, s_3, \ldots, s_H, a_H, s_{H+1})$$

$$R(\tau) = r_1 + r_2 + r_3 + \ldots + r_H + r_{H+1}$$

The goal here is still the same, we need to find the weights θ of the neural network that maximize expected return – U. As we discussed, we adjust the network weights.
We can express U as following function,

$$U(\theta) = \sum_{\tau} \mathbb{P}(\tau; \theta) R(\tau)$$

where R(τ) is a just the return corresponding to an arbitrary Trajectory τ. The other component in above formula P(τ**;** θ) is the probability of each possible Trajectory.

We can see that probability depends on the weights θ in the neural network (θ defines the policy which is used to select the actions in the Trajectory). As we discussed before, our goal is to find the value of θ that maximizes expected return. To do so we can compute the Gradient Ascent (opposite to gradient descent which we use to find the minimum of the function).
Here, we need to perceive that in computation of the real value of gradient ascent stipulates that we need to evaluate all possible trajectories. It seems that this process can be very computationally expensive, so our approach here will be to just sample a few trajectories (m) using the policy and then use those m – trajectories only to estimate the gradient.
Finally, the gradient can be express as follows,

$$\nabla_\theta U(\theta) \approx \hat{g} := \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) R(\tau^{(i)})$$

Once we have an estimate for the gradient, we can use it to update the weights of the policy. Then, we repeatedly loop over these steps to converge to the weights of the optimal policy.
Here we need to understand/recall one more thing. The output from the neural network is just the number (when 2 nodes like in the our Cart Pole example, we have two numbers). Computation of probability distribution involves the need of softmax function exertion (function that takes as input a vector of *K* real numbers, and normalizes it into a probability distribution consisting of *K* probabilities).

Other thing is the log of probability π, which refers to the policy, which is parameterized by theta. This mathematical operation can be understood as the standard cross entropy operation, which is used to to quantify the difference between two probability distributions. Finally, we can estimate how we should change the weights of the policy theta, if we want to change the log probability (remember what we specified before. The agent changes the weights to increase the probability for the actions where the episode finishes with success or decrease the probability of the actions, where the episode finish with lost).

The pseudo code of **Reinforce algorithm** can be summarize as follows.

1. Use the policy $\pi\vartheta$ to collect M – trajectories with horizon H.

$$\tau^{(i)} = (s_0^{(i)}, a_0^{(i)}, \ldots, s_H^{(i)}, a_H^{(i)}, s_{H+1}^{(i)})$$

2. Use the trajectories to estimate the policy gradient:

$$\nabla_\theta U(\theta) \approx \hat{g} := \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) R(\tau^{(i)})$$

3. Update the neural network weights of the policy (α – is a hyper parameter of neural network, step – learning rate):

$$\theta \leftarrow \theta + \alpha \hat{g}$$

4. Loop over steps 1 – 3.

4. ACTOR – CRITIC METODS

Our goal is to apply DDPG algorithm but due to complexity we should first approach our design throughout understanding new (discussed below) concepts.

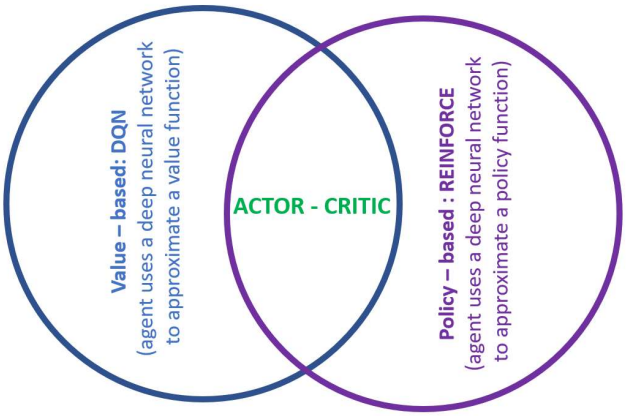

Figure 3. Actor – Critic methods

Above picture depicts the main concept of Actor -Critic methods. Here Actor – Critic methods combine the value-based methods such as DQN and policy-based methods such as Reinforce.

Previously we defined DQN Agent (in project 1) which learns to approximate the optimal action value function. If the Agent learns sufficiently well so deriving a good policy for the Agent it is straightforward.
On the other side the Reinforce Agent parameterizes the policy and learns to optimize it directly. Here, the policy is usually stochastic, as we receive the distribution probability.
Right now, we will investigate **deterministic policies**, which take a state and return the single action (no stochasticity, the policy will be deterministic).

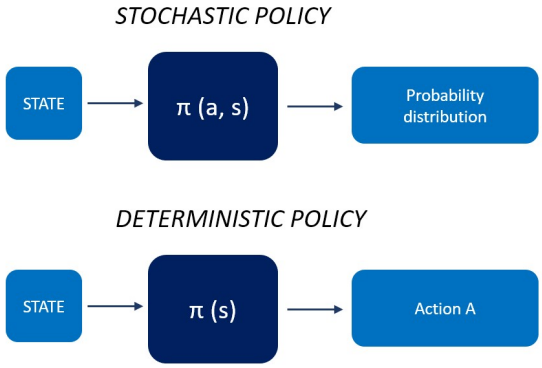

Figure 4. Stochastic and deterministic policy

In previous section we presented the Reinforce algorithm, which has to complete the episode before we can start training. For the environments, where every episode can last hundreds or even thousands of frames (like Atari games). It can be wasteful for the training perspective, where we have to interact with the environment in long perspective, only just to perform a single training step (in order to estimate Q as accurate as possible). It this case, the training batch becomes very large. For the DQN however, it is possible to replace the exact value for a discounted reward with our estimation using the one-step Bellman equation:

$$Q(s,a) = r + \gamma V(s')$$

When we consider the Policy Gradient method (as we discussed above), we contemplated that the values V(s) or Q(s, a) exist anymore. In this case we apply the Actor – Critic method instead, where we use neural network to estimate V(s) and use this estimation to obtain Q.
Estimated gradient in Policy Gradient method is proportional to the discounted reward from the given state. However, the range of this reward is highly environment – dependent (it can happen that the Agent plays only short game – the Agent lose very quick (low value of reward) or the Agent is smart enough and plays the game for the longer time, while collecting the rewards). Large difference between rewards collection can seriously affect the training dynamics, as one lucky episode will dominate in the final gradient. In such occurrences, the policy gradient method has high **variance**, which can influence the training process can become unstable).

In reinforcement learning we look for the bias – variance trade – off (consider below figure), when the Agent tries to estimate value functions or policies from returns (we need to remember that the Agent samples only environment, so we are able to only estimate these expectation). Generally, the main effort in our domain (reinforcement learning) is an attempt to reduce the variance of algorithms while keeping bias to a minimum. The task is hard to achieve, but we will approach to some techniques that are designed to accomplish this.

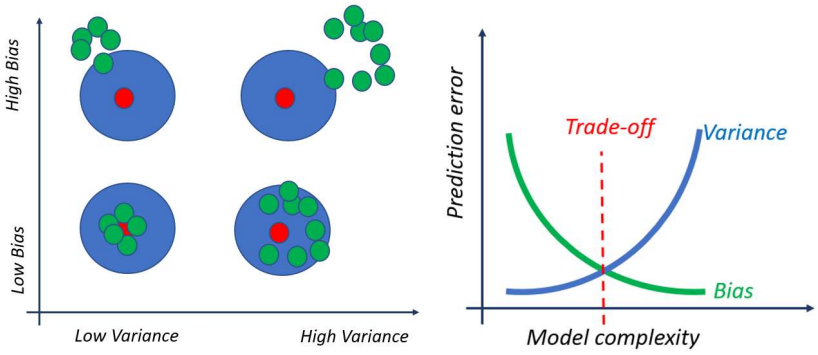

Figure 5. Trade – off for variance and bias

5. DEEP DETERMINISTIC POLICY GRADIENT (DDPG) algorithm

As it was discussed in Udacity Deep Reinforcement Learning nanoprogram there exist two complimentary ways for estimating expected returns.

First is the **Monte – Carlo estimate**, which roles out an episode in calculating the discounter total reward from the rewards sequence.

In Dynamic Programming, the **Markov Decision Process** (**MDP**) is solved by using value iteration and policy iteration. Both techniques require transition and reward probabilities to find the optimal policy. When the transition and reward probabilities are unknown, we use the Monte Carlo method to solve MDP. The Monte Carlo method requires only sample sequences of states, actions, and rewards. Monte Carlo methods are applied only to the episodic tasks.

We can approach the Monte – Carlo estimate by considering that the Agent play in episode A. We start in state S_t and take action A_t. Based on the process the Agent transits to state S_{t+1}. From environment, the Agent receives the reward R_{t+1}. This process can be continued until the Agent reaches the end of the episode. The Agent can take part also in other episodes like B, C, and D. Some of those episodes will have trajectories that go through the same states, which influences that the value function is computed as average of estimates. Estimates for a state can vary across episodes so the Monte – Carlo estimates will have high variance.

On the other side, we can apply the **Temporal Difference estimate**. Here, the TD approximates the current estimate based on the previously learned estimate, which is also called **bootstrapping** (we try to predict the state values).

$$V(s) = V(s) + \alpha(\underbrace{r + \gamma V(s')}_{\text{Actual reward}} - \overbrace{V(s)}^{\text{Expected reward}})$$

Above equation is the difference (**TD error**) between the actual reward and the expected reward multiplied by the learning rate alpha (the learning rate, also called step size, used for convergence reason).

TD estimates are low variance because you're only compounding a single time step of randomness instead of a full rollout like in Monte – Carlo estimate. However, due to applying a bootstrapping (dynamic programming) the next state is only estimated. Estimated values introduce bias into our calculations. The agent will learn faster, but the converging problems can occure.

Deriving the Actor – Critic concept requires to consider first the **policy – based approach** (performed by **AGENT**). As we discussed before the Agent playing the game increases the probability of actions that lead to a win, and decrease the probability of actions that lead to losses. However, such process is cumbersome due to lot of data to approach the optimal policy.
On the other hand, we can evaluate the **value – based approach** (performed by **CRITIC**), where the guesses are performed on-the-fly, throughout all the episode. At the beginning our guesses will be misaligned (not correct). But over time, when we capture more experience, we will be able to make solid guesses. Though, this is not a perfect approach either, guesses introduce a bias because they'll sometimes be wrong, particularly because of a lack of experience.

Based on this short analysis we can summarize that the Agent using policy – based approach is learning to act, while in a value-based approach, the agent is learning to estimate states and actions. Merging these two approaches often yields better results.
Actor – critic agents learn by interacting with environment and adjusting the probabilities of good and bad actions just as with the actor alone. In parallel we use a Critic, which is to be able to evaluate the quality of actions more quickly (proper action or not) and speed up learning. Actor-critic method is more stable than value – based agents, while requiring fewer training samples than policy-based agents.

As a result of merge Actor – Critic we utilize two separate neural networks. The
role of the Actor network is to determine the best actions (from probability distribution) in the state by tuning the parameter θ (weights). The Critic by computing the temporal difference error TD (estimating expected returns), evaluates the action generated by the Actor.

In previous project (DQN) we discussed about discrete environments where the number of action which could be performed by the Agent was limited. However, very often we operate with continuous environment, securing continuous motion. The number of action then can be unlimited (huge). This is one of the problems DDPG solves. DDPG algorithm uses Agent – Critic concept, where we use two deep neural networks.
In DDPG, the Actor is used to approximate the optimal policy deterministically. That means we want always to generate the best believed action for any given state.
The Actor follows the policy-based approach, and learns how to act by directly estimating the optimal policy and maximizing reward through gradient ascent. The Critic however, utilizes the value-based approach and learns how to estimate the value of different state – action pairs.

The critic learns to evaluate the optimal action value function by using the actors best believed action. Afterwards, we use this actor, which is an approximate maximizer, to calculate a new target value for training the action value function.

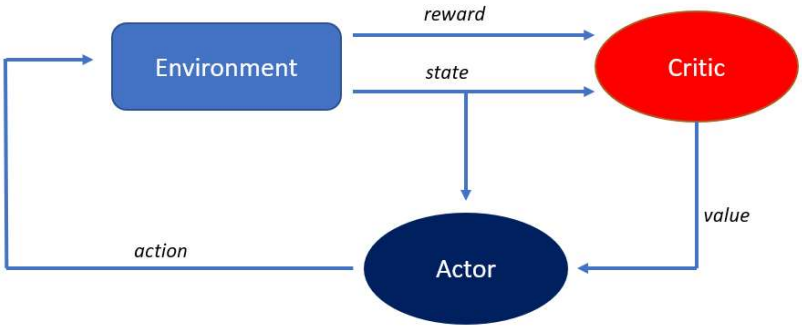

Figure 6. Reinforcement learning architecture

The DDPG algorithm can be presented as follows (Reinforcement Learning with Python by Sudharsan Ravichandiran)

1. The Actor and the Critic use separate neural network.
2. The Actor network with: $a = \mu(s; \theta^{\mu})$ which takes input as a **state s** and results in the **action a** where $\theta^{\mu}$is the Actor network learning weights. The actor here is used to approximate the optimal policy deterministically. That means that the output is the best believed action for any given state. This is unlike a stochastic policy (probability distribution) in which we want the policy to learn a probability distribution over the actions. In DDPG, we want the believed best action every single time we query the actor network. The actor is basically learning the argmax a Q(S, a), which is the best action.
3. The Critic network $Q(s; a, \theta^{Q})$ => $Q(s; \mu(s; \theta^{\mu}), \theta^{Q})$ which takes an input as a **state s** and **action a** and returns the *Q* value where is the Critic network θ^{Q} weights. The critic learns to evaluate the optimal action value function by using the actors best believed action.
4. We define a target network for both the Actor network $\mu(s; \theta^{\mu`})$ and Critic network $Q(s; a, \theta^{Q`})$ respectively, where $\theta^{\mu`}$and $\theta^{Q`}$ are the weights of the target Actor and Critic network.
5. Next, we perform the update of Actor network weights with policy gradients and the Critic network weight with the gradients calculated from the TD error.
6. In order, to select correct action, first we have to add an exploration **noise *N*** to the action produced by Actor: $\mu(s; \theta^{\mu}) + N$ (add noise to encourage exploration since policy μ is deterministic).
7. Selected action in a state, *s*, receive a reward, *r* and move to a new state, *s'*.
8. We store this transition information in an experience replay buffer.
9. As it is performed while we use DQN algorithm, we sample transitions from the replay buffer and train the network, and then we calculate the target *Q* value:
$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$
10. Then, we can compute the TD error as:

$$L = \frac{1}{M}\sum_i (y_i - Q(s_i, a_i|\theta^{Q})^2)$$

Where *M* is the number of samples from the replay buffer that are used for training.

11. Subsequently, we perform the update of the Critic networks weights with gradients calculated from this loss *L*.
12. Then, we update our policy network weights using a policy gradient.
13. Next, we update the weights of Actor and Critic network in the target network. In DDPG algorithm topology consist of two copies of network weights for each network, (Actor: regular and target) and (Critic: regular and target). In DDPG, the target networks are updated using a **soft updates strategy**. A soft update strategy consists of slowly blending regular network weights with target network weights. In means that every time step we make our target network be 99.99 percent of target network weights and only a 0.01 percent of regular network weights (slowly mix of regular network weights into target network weights)
14. We update the weights of the target networks (Agent, Critic) slowly, which promotes greater stability (soft updates strategy):

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

DDPG algorithm can be expressed in conscience shape as a pseudocode:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 7. Pseudocode of DDGP (Continuous Control with Deep Reinforcement Learning paper)

General overview of DDPG algorithm flow was portrayed on underneath chart

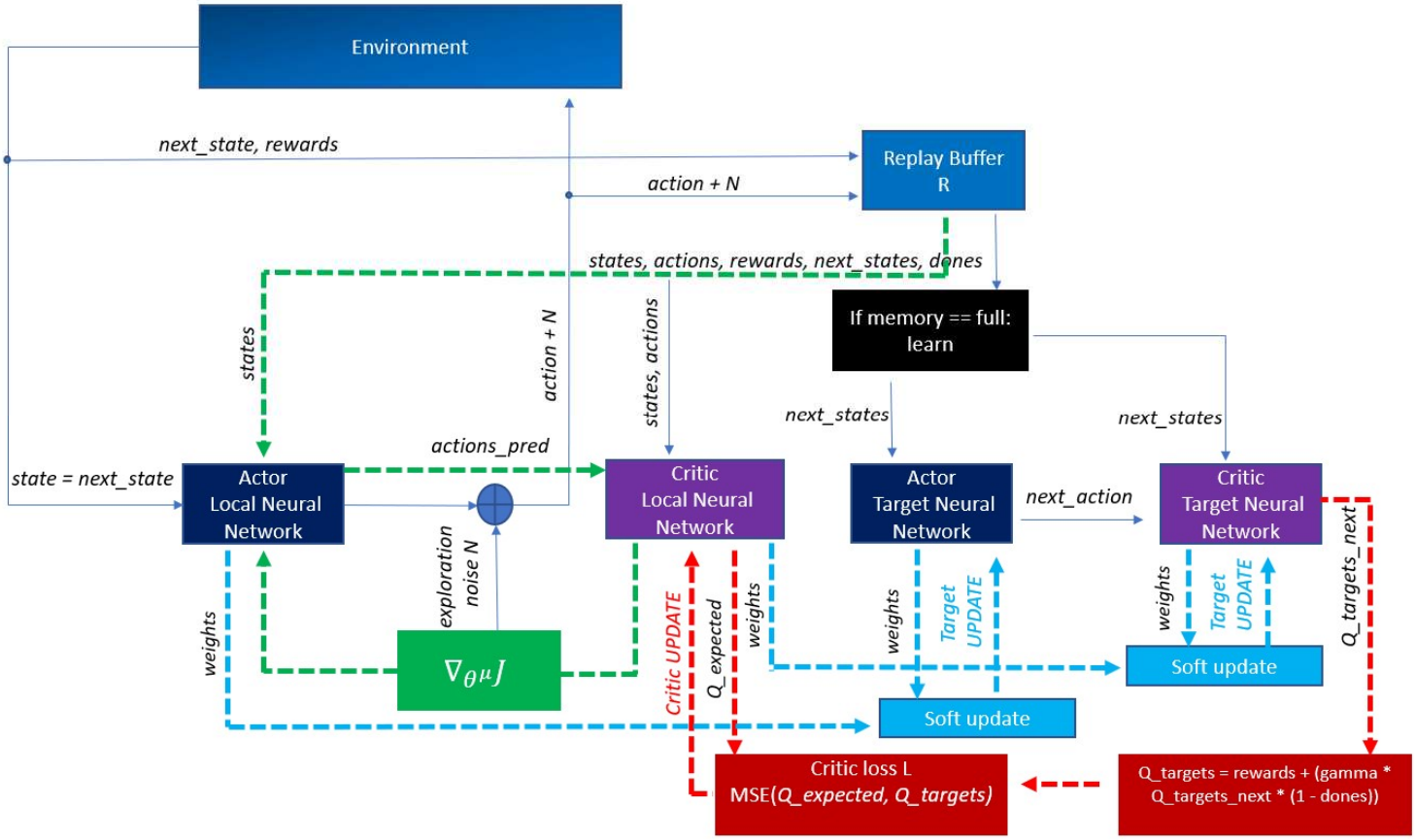

Figure 7. DDPG algorithm flow

6. PROJECT SETUP. RESULTS.

In Continuous Control project following setup of neural networks (for Agent and Critic) were involved.

Depicted below plot of rewards illustrates that the agent is able to receive an average reward (over 100 episodes, and over all 20 Agents) while playing 206 episodes.

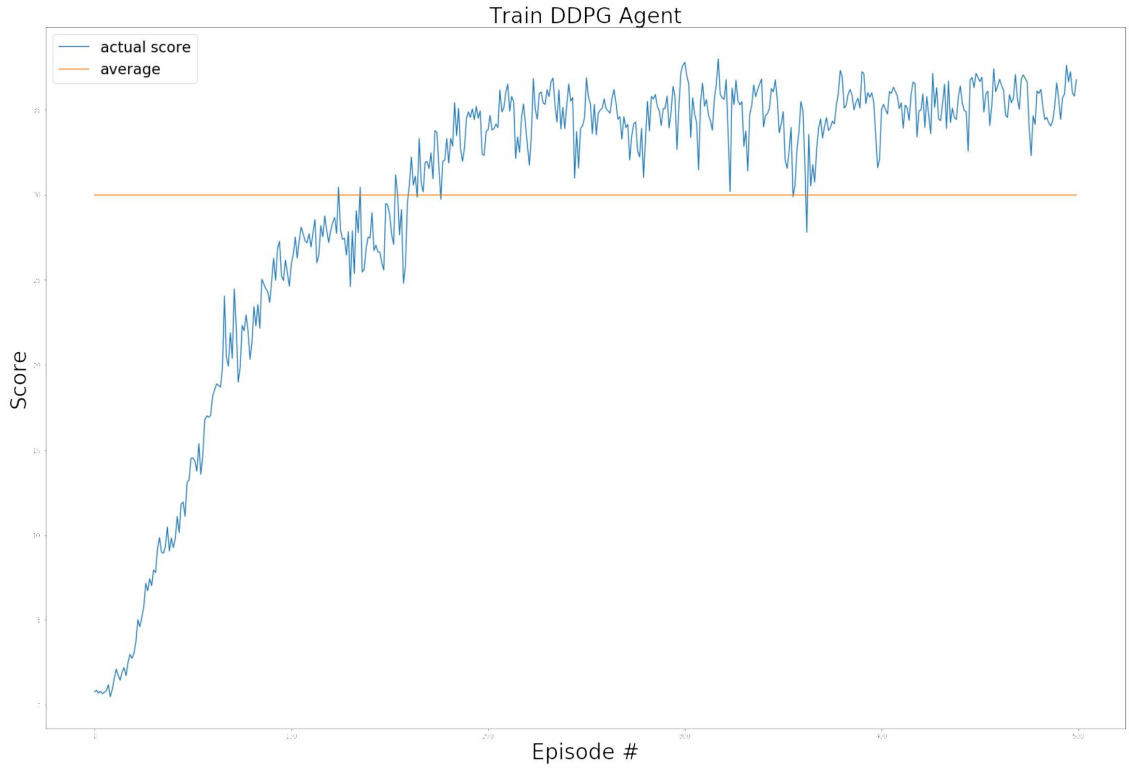

Figure 8.Training process. Results

```
Episode 100     Average Score: 13.53
Episode 200     Average Score: 29.62
Episode 206     Average Score: 30.05
Environment solved in 106 episodes!     Average Score: 30.05
Episode 207     Average Score: 30.13
Environment solved in 107 episodes!     Average Score: 30.13
Episode 208     Average Score: 30.21
Environment solved in 108 episodes!     Average Score: 30.21
Episode 209     Average Score: 30.29
Environment solved in 109 episodes!     Average Score: 30.29
Episode 210     Average Score: 30.37
Environment solved in 110 episodes!     Average Score: 30.37
Episode 211     Average Score: 30.47
Environment solved in 111 episodes!     Average Score: 30.47
Episode 212     Average Score: 30.54
```

Applied Deep Neural Network architecture (Agent):

Input layer FC1: 33 nodes in, 200 nodes out
Hidden layer FC2: 200 nodes in, 150 nodes out
Output layer FC3: 150 nodes in, 4 out – action size

Applied Deep Neural Network architecture (Critic):

Input layer FC1: 33 nodes in, 400 nodes out
Hidden layer FC2: 400 nodes in, 300 nodes out
Output layer FC3: 300 nodes in, 1 out – action size

Applied hyperparameters:

```
BUFFER_SIZE = int(1e6   )  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99          # discount factor
TAU = 1e-3            # for soft update of target parameters
LR_ACTOR = 1e-3             # learning rate of the actor
LR_CRITIC = 1e-3            # learning rate of the critic
WEIGHT_DECAY = 0       # L2 weight decay
Epsilon start = 1.0
Epsilon start = 0.01
Epsilon decay = 1e-6
```

7. IDEAS OF FUTURE WORK

The discussed work compounds contemporary advances in deep learning and reinforcement learning. Modern combination yields exceptional results in solving challenging AI issues across a variety of domains. However, in this project a few limitations to presented approach remain. Future work should concentrate mainly on following tasks. Most notably, which was treated as a minor case was tuning of the **hyper parameters**. Future work can include more parameter choice and checks in order to limit number of episodes exhausting the project goal. This can also comprise the deep neural **network architecture design**. The other thing which will be reasonable to verify is the choice of applied algorithm. Here, it is suggest to deploy the A3C (**Asynchronous Advantage Actor – Critic**) algorithm which exploits multiple agents. Each Agent has its own set of network parameters. Additionally, the Agent interacts (in parallel to the other Agents) with its own copy of the environment. Agents (workers) follows a different exploration strategy to learn an optimal policy. Following this, the Agents compute value and policy loss so the derived gradient can be applied the global network. Proceeding process is continued and the advantage function is estimated. Finally, the global value loss and policy loss are computed. The policy loss function includes the entropy component, which reveals the spread of action probabilities. Low entropy secures that one of the actions has a higher probability then the other actions, so the Agent has a good opportunity for convenient action selection. Although, the high entropy displays that every action's probability is the same, so the Agent will be unsure as to which action to perform. We use the entropy to improve exploration, by encouraging the Agent to be reliable of excavation of perfect (optimal) action. A3C algorithm can sufficiently speed up the training process.