Collaboration & Competition

Project in Deep reinforcement learning as part of a Nano degree.

January 16, 2020
Jonas le Fevre Sejersen
Jonas.le.fevre@gmail.com

1 The scope of the project

For this project, we will train multiple agents to collaborate within the Tennis environment and play the game of tennis. Each agent can control the action of a tennis player, where the actions are the velocity of the player (1 forward, -1 backwards) and jump/shoot. The rules are quite simple, 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.

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 $+0.5$.

In this project, I chose to implement the multi agent Twin Delayed Deep Deterministic (TD3) Policy Gradient network meaning both player will be controlled by a TD3 model. These two TD3 agents are not disconnected, as they share a replaybuffer and the twin critic, leaving only the actor to be seperated. Since I already covered TD3 in the continuous control I will leave this report to describe the changes made to adapt the TD3 to a multi agent TD3. For more information about the baseline TD3 model visit my continous control github repository and read the report.pdf.

2 Multi agents

In this project we have two agents trying to play tennis together in a collaborative manner, so the more they pass to the ball to each other the higher the reward. If we were to change the reward function to reflect the score instead, then we have changed the environment to a competitive environment, where each agent is trying to have the opponent lose the ball.

In this project we are using a centralised trading with decentralised execution framework, where we share experiences among the critics during training but during execution we have trained separate actors / policies.

3 Hyper-parameters

This section will lead up to the result section by explaining some of the hyper-parameters used to produce the final result. We will go through each hyper-parameter and shortly describe how it affects the model.

3.1 General info

The general info is info about the game and what seed have been used:

```
game: Tennis.exe
seed: 0
state_size: 24
action_size: 2 # Continuous space between -1 and 1
slow_and_pretty=False # If it should display in real-time or training time
result_folder='TD3_multi_agent_tennis\\results' # Where to save result
```

Not much to note here since it is the basic game data and the only real changeable parameter is the seed.

3.2 Model info

The model info is the hyper-parameters for the model structure. We used two kind of models a **Actor** and a **Twin Critic** which is a single network but runs two separate streams, one for each approximated Q-function.

I have chosen to only make the seed a changeable parameter and let the structure of the model (the number of layers and neuron) be constant.

Twin Critics:
In the twin critic model i use two fully connected hidden layers for each stream. We take in the state and action so the input layer is of size state_size + action_size, then the first hidden layer contains 256 neurons and second hidden layer 156 neurons. We want the network to be small for faster training and less generalisation. For the output layer we want a single output which is the best action, so we have a single neuron.

An important observation is that using a batch normalisation layer in the critic model is not an improvement when learning. After removing the batch normalisation, the agents learned much faster. **Actor**:
is very similar as we have two hidden layers with same amount of neurons. The different here is the input layer takes in the state and the output layer gives the action values.

3.3 Agent info

The agent info is the hyper-parameters used to configure the TD3 agent.

```
discount: 0.99              # Discount value
TAU: 0.01                  # Amount we update the target model each update
lr_actor: 1e-4                # The learning rate of the actor
lr_critic: 1e-3               # The learning rate of the critic
weight_decay: 0               # The decaying on critic's learning rate
steps_before_train: 4         # The number of steps between model updates
train_delay: 2                # Policy delay
train_iterations: 2           # How many batches we train on each train call
policy_noise: 0.2             # noise to smoothing the policy
noise_clip: 0.5               # How big the noise is for smoothing
exploration_noise: 0.3        # Noise added for exploring new policies
noise_reduction_factor: 0.999 # Noise are decayed each timestep
noise_scalar_init: 2          # Noise scalar added to exloration
```

I found that using a high value of TAU (0.01) really improved the learning process. The reason might be because of the low amount of rewarding experiences, so whenever a important experience is found, we need the target network to update as well with the new found information. Another big factor is the initial exploration noise. We are using a noise_scalar to explore more at the start of each episode, which improves how fast the agents are picking up the relation between hitting the ball and given reward. We are reducing the noise as the episode goes on, so we do not lose the episode.

3.4 Replay buffer

The replay buffer info telling us something about how long we store experiences and how many experiences we use each training step.

```
BUFFER_SIZE: 1e5               # The space we use to store Experiences.
BATCH_SIZE: 256                # Amount of replays we train on each update.
```

I tend to use a large Buffer size, since this way we do not tend to forget less experienced replays, but found in this project that a lower buffer size and a lower batch size performed much better. It might be because of the high amount of filler actions, waiting for the ball to come closer, that having a lower batch size will prevent drowning the actions that actually matters. The lower buffer size might help us at the early stages of training since we are clearing out less performing actions much faster, increasing the probability of getting experiences that led to reward. The small buffer size, might be a bad decision if we had to train for longer. Since we are less resistance to bad behaviour, if we start exploring it will generate a lot of experience that leads to no rewards and hereby replacing experiences with high reward.

3.5 Training info

Lastly is the training info, which is simply how many episodes we want to train for, how long an episode is and if we want to load an old model or evaluate.

```
episodes: 4000          # Max 4000 episodes to complete the task.
load_model_path: ''     # If we want to train on a old model.
eval=False              # If we should just evaluate.
eval_load_best=False    # Load the solved folder model
```

In the next section we take a look at the result of both training phase of the model and the evaluation steps done through training.

4 Results

Every model trained is saved to the folder called *results/* and then a sub-folder is created called test and a number generated, where a **model_test.md** is placed containing all the hyper-parameters and the result of that run. A tensorboard file is also generated in to the logging folder, which contains a scalar plot over average_100_score and the average max score for each episode Lastly the folder contains the trained model weights, this is the file you would load to use the trained model. In this section we are going through the result of the folder *solved* that will be located in the results folder.

Training

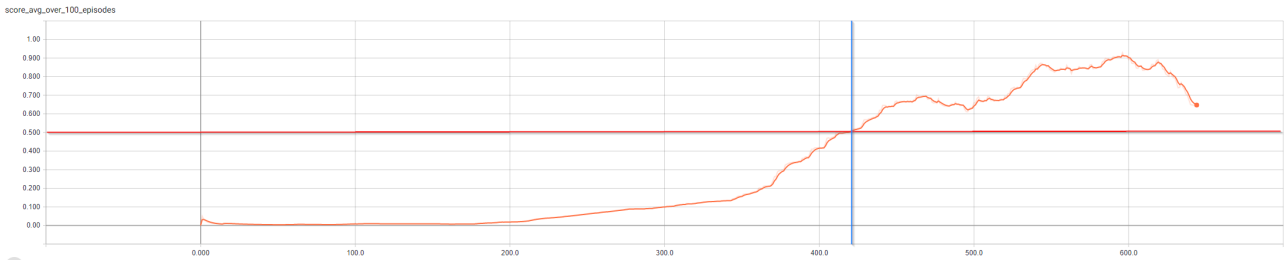

Fig. 1: The first plot is an average score over the last 100 episodes (or all episodes if total of episodes is lower than 100). The second plot is showing the average score of all 20 agents for each episode

In figure 1 a plot over the training period representing the average max scores on the y-axis and the episodes on the x-asis, we can make the observation that during the first 200 episodes the agent is exploring the state space and here is the scalar_noise really important for a faster exploration. After the first 200 episodes the agents start to understand that they have to hit the ball, and a lot of episode goes on by just having one player hit the ball. At around episode 350 they have the ability to pass the ball to eachother, and here it starts to learn really fast.

So by the definition of solved given by udacity, the environment was solved at episode 412, since the average of the average scores from episode 312 to 412 was greater than +0.5. The full training values are shown in my git repository and the results are here: model_test.md.

5 Future improvement

Some future improvement that I didn't get to implement (yet), is the use of N-step learning. A behaviour we might see from using N-step is the agent learns from the experiences leading up to an reward, hopefully shorten the distance between the player. Another improvement is using a Priority experienced replay buffer. The paper: Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards have already tried to come up with a priority function for DDPG, we might be able to transfer the priority function to work with TD3.

Appendix: A

Hyper-parameters used:

```
batch_size=256
buffer_size=100000
discount=0.99
episodes=4000
eval=False
eval_load_best=False
exploration_noise=0.3
load_model_path=''
lr_actor=0.0001
lr_critic=0.001
max_timesteps=2000
noise_clip=0.5
noise_reduction_factor=0.999
noise_scalar_init=2
policy_noise=0.2
result_folder='D:\\dev\\learning\\TD3_multi_agent_tennis\\results'
seed=0
slow_and_pretty=False
steps_before_train=4
tau=0.01
train_delay=2
train_iterations=2
warmup_rounds=0
weight_decay=0
```

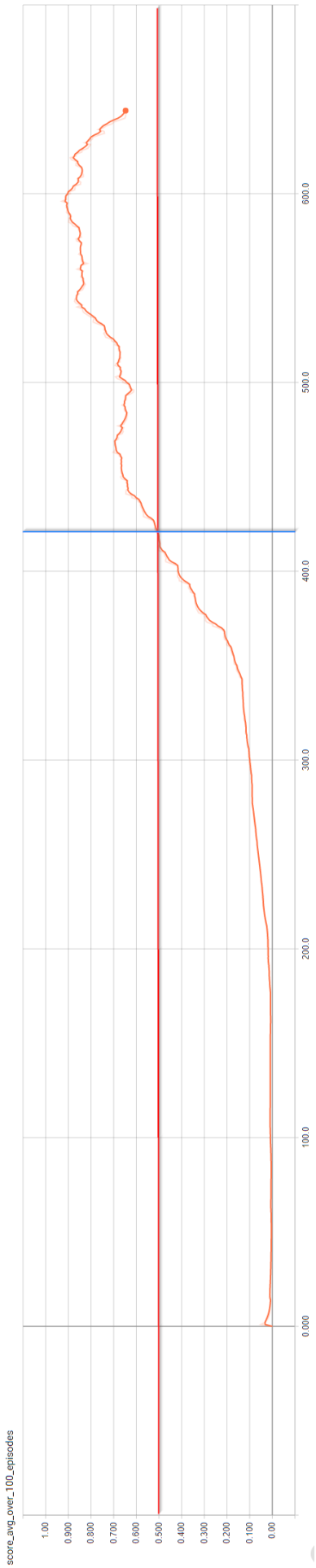

Fig. 2: Bigger plot but same as fig 1