DRL for Tennis

1. Introduction

This is a project that accompanies the last project assignment for the Udacity course on Deep Reinforcement Learning. This project is supposed to explore ways of training collaborative learning, and the problems that might be associated with it.

2. Environment

The picture of the Unity agent during play is shown in Fig. 1. The agent provided to us is similar (but not identical) to the Unity Tennis playing program available here.

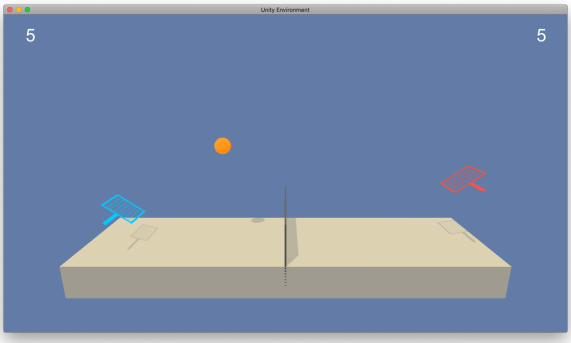


Fig. 1. Unity environment during play.

Rather than working directly on the images available from the environment, it has been converted into a state representation of size 24 for each agent at each time point. The action space has a size of 2 and should be a real number between -1 and 1. At the i^{th} time point, the agent provides an array for the state ($[s_i^0, s_i^1]^T$) and for an action ($[a_i^0, a_i^1]^T$), returns the next state ($[s_{i+1}^0, s_{i+1}^1]^T$) and corresponding reward ($[r_{i+1}^0, r_{i+1}^1]$) done ($[d_{i+1}^0, d_{i+1}^1]$). Notice that the state and actions are vectors, and the superscript T represents the matrix transpose operator. The superscript of 0 or 1 represent the agent that the particular action, state reward or done information is associated with.

Each reward is 0.1 for a successful hit, and -0.01 for a failure. The environment is solved when the agents are able to get a score (i.e., sum of all rewards within one episode) of at least 0.5 for consecutive 100 episodes. This means that at least one of the agents should be able to hit the ball across the net 5 times or more for 100 episodes in a row.

3. Dependency

The project depends upon the python environment and libraries provided with the course. This is also present within the Github environment in the `python/` folder. It is ideal to generate a virtual environment and load the libraries available within this folder before using the program. The only additional library that has been used is `tqdm` which can be downloaded from PyPI and installed by `pip`.

Furthermore, the Tennis environment must be added. This has to be manually downloaded and placed in a folder of choice. Note that the downloaded environment is specific to the operating system that is being used.

4. Implementation

The implementation details of the model will be described in the following subsections.

4.1. Model Description

A schematic representation of the model is shown in Fig. 2.

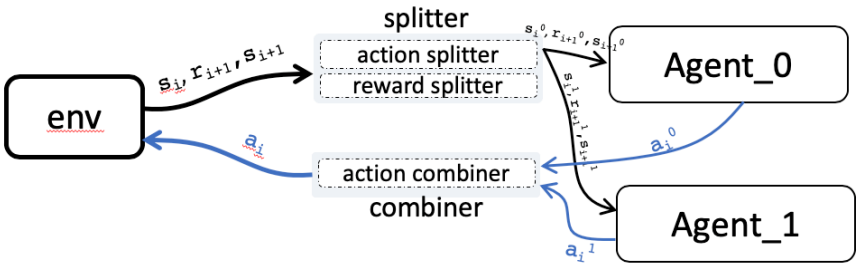


Fig. 2. A schematic representation of how the model is put together.

The model is not actually realized within a class. However, the astute observer will be able to recognize the rough structure within the training section in the file `src/utils/trainer.py`. It comprises of two agents that interact with the environment through a splitter/combiner. This splits the states (s_i), rewards (r_i), and *donnes* (not shown in the figure above), and combines the actions (a_i), from the agents, so that the agents are able to react to the environment.

Thus, the multi-agent problem is just broken down into two agents working independently. These will learn to react to the environment by themselves. Details of the agent and each of its components will be described in the following sections. The decision to use different agents, rather than using the same agent simply is because each agent is supposed to hit the ball in opposite directions. This means that the each agent is anti-symmetric and when one agent is supposed to hit the ball to the left, the other agent is supposed to hit the ball to the right under very similar conditions. It is not clear that this anti-symmetry is broken in some way when the states are generated.

4.1.1. The Agent

A schematic diagram of the agent is shown in Fig. 3.

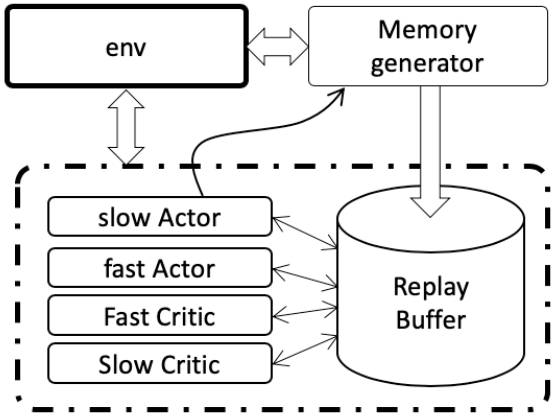


Fig. 3. A schematic diagram of the agent used by the model.

The Agent is defined by the `Agent` class in the file `src/utils/NN.py`. The config file contains information that can be used for modifying the parameters of the `Agent`. The parameters of the Agent are shown below:

```
"Agent" : {
    "actorLR"  : 1e-4,
    "criticLR" : 1e-3,
    "Tau"      : 0.001,
    "Actor" : { /*configuration for the actor*/ },
```

```
    "Critic" : { /*configuration for the critic*/ },
    "ReplayBuffer" : { "maxDataTuples" : 5000} }
```

This part of the configuration file defines the configuration for the `Agent`. The learning rates for the actors and the critics are defined as the `actorLR` and `criticLR` respectively and is represented by α in the following equations. Although there has been a single α used in the equations below, it is important to know that each can be tuned individually. Tau represents the factor by which the slow-moving actor and critics are updated and is represented by the symbol τ in the equations below.

The `ReplayBuffer` has only one parameter `maxDataTuples`. This defined the maximum number of data tuples that will be saved by the memory buffer associated with this particular Actor.

The `Actors` and the `Critics` can be controlled to a significant degree, and this is described in Sections 4.1.1.1. and 4.1.1.2.

The agent comprises of two actors and two critics. It is intended to solve for the equation for RL:

$$q_i(s_i, a_i) = r_i + \sum_{j=i+1}^{\infty} \gamma^{j-i} r_j \tag{1}$$

In the equation above, s_i and a_i refer to the i^{th} state and action respectively, and γ represents the discount factor. Of course, we don't know the future reward. Hence it is typically approximated with what the future cumulative value *might have been*. This is typically rewritten in the following manner:

$$q_i(s_i, a_i) = r_i + \gamma q_{i+1}(s_{i+1}, a_{i+1}) \tag{2}$$

Note that not only do we not know the function q, we also have no idea how to come up with the action a. To solve this problem, we shall generate two *actors* and two *critics*. The actors take a state and predict an action, while a critic takes a state and an action and try to predict the Q value. We shall represent the actors by A^s and A^f for the slow- and fast-acting actors respectively. Similarly, we shall denote the critics by C^s and C^f. Using these, we shall rewrite eq. (2) by replacing the critics with the Q values.

$$C^f(s_i, a_i) = (1-\alpha)C^f(s_i, a_i) + \alpha[r_i + \gamma C^s(s_{i+1}, a_{i+1})] \tag{3}$$

Remember that this equation is not strictly a mathematical equation. It represents the update rule of the weights of the fast critic.

Now, let is replace the actors into eq. (3). This equation represents the learning equation for our fast critic. Notice that the slow critic is not changed. This is used for providing some stability to the critic for predicting the future rewards. This slow critic is periodically updated with the value of the fast critic with an τ that works in a manner similar to the learning rate α shown in eq. (3).

$$C^f\left(s_i, A^f(s_i)\right) = (1-\alpha)C^f\left(s_i, A^f(s_i)\right) + \alpha\left[r_i + \gamma C^f\left(s_{i+1}, A^s(s_{i+1})\right)\right] \tag{4}$$

Actors are just supposed to maximize the Q value. Hence the update equation is represented by the following equation:

$$A^f(s_i) = (1-\alpha)A^f(s_i) + \alpha C^s\left(s_i, A^s(s_i)\right) \tag{5}$$

Finally, the slow-moving actors and critics are periodically refreshed in a similar manner:

$$\begin{aligned} A^s &= (1-\tau)A^s + \tau A^s \\ C^s &= (1-\tau)C^s + \tau C^s \end{aligned} \tag{6}$$

The entire set of self-consistent equations are then recursively solved by going through interaction with the environment over an extended period of time until some solution criterion is reached. In this specific case, each actor has to be able to consistently hit the ball across 9 to 10 times.

4.1.1.1. The Actors

A schematic representation of the actor is shown in Fig. 4. (a).

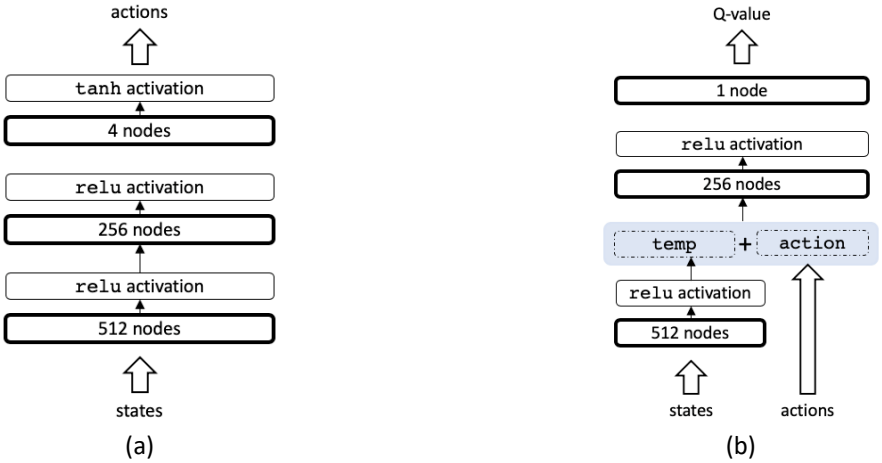


Fig. 4. A schematic representation of the actor (a) and the critic (b) is shown above.

The agent is a simple sequential neural network model that takes the state as input and generates the actions as output. It comprises of three layers with 512, 256 and 4 nodes each, with the first two layers having `relu` activation, while the final layer having the `tanh` activation. The `tanh` activation of the last layer allows the output to be bonded to somewhere between -1 and 1, as is the requirement of the four actions that the actor is supposed to return. Although the actor has the current specifications, the size of the nodes can be easily changed using the config file `src/config.json` within the `Actor` section. This section is shown below:

```
"Actor" : {
    "state_size"  : 24,
    "action_size" : 2,
    "seed"        : 1234,
    "fc1_units"   : 512,
    "fc2_units"   : 256 }
```

The size of the state and action is fixed by the problem. Hence, it is best not to change these. However, the sizes of the first and the second layers can be changed by changing the numbers in this section.

4.1.1.2. The Critics

A schematic representation of the critic is shown in Fig. 4. (b). The critic is also a fairly simple sequential neural network model. The state goes through a layer that has 512 nodes and is then activated by `relu` activation. This is then combined with the actor vector, and the result is then passed through the next layer containing 256 nodes with `relu` activation again. This is then passed onto the next layer that has a single node. This is the Q value. Note that the Q value is unbounded and linear, and thus is not activated.

The configuration file also has information about the critic, and this can be changed as well. Information about the critic is shown below:

```
"Critic" : {
        "state_size"  : 24,
        "action_size" : 2,
        "seed"        : 1234,
        "fcs1_units"  : 512,
        "fc2_units"   : 256}
```

4.1.1.3. The Replay Buffers

The replay buffers have been generated using a simple `deque`. The replay buffer is a class that has been defined within the file `src/utils/memory.py` by a class called `ReplayBuffer`. This exposes several methods that allow the replay buffer to be populated (`append`, and `appendMany`), sampled (`sample`) and also has methods that allow the `ReplayBuffer` to be saved (`save`) and reloaded (`load`) to and from disk at any point. This is especially useful because it is possible that we have already had a great set of memories that we can use. It is typically a total waste to throw away the entire buffer at one go. Hence, these buffers may be saved, so that they may be used in the future if needed.

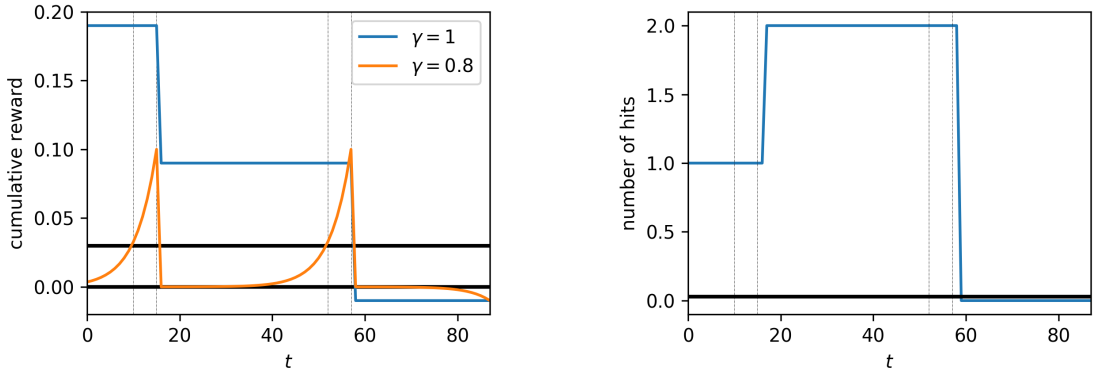
There are several characteristics of the specific part of the game that makes the design of the memory buffer efficient.

1. The game is episodic.
2. Points are awarded every time the racquet of an agent is able to hit the ball across the net.
3. Each hit is fairly independent of the other hits. Hence the probability that a particular agent is going to be able to hit the ball across the net depends mostly upon the set of actions that the agent takes a few time-points (let's say 5 time points) before actually hitting the ball, at which point the racquet actually lobs the ball over the fence. Hence, it would be most ideal if the agent learns around this segment, rather than during all segments that the agent does not actually make a dent at the learning experiment.
4. Note that it is quite easily possible to calculate the actual cumulative reward for this problem for a fairly large number of episodes.

Hence, in this specific case, the memory buffer contains a tuple of the following form:

```
(state, action, reward, next_state, done, cumRewards, numHits)
```

The cumulative rewards (represented by `cumRewards`) are calculated with a γ of 1. However, the choice of the value of γ that should be used will be discussed in the next section. `numHits` represents the number of hits that a particular agent is having for a particular hit. This is easier to visualize with the following graphs



(a) (b)

Fig. 5. Different characteristics of an agent is shown for an entire episode. In this episode, the agent starts, and successfully lobs the ball over the net (approximately at a time point of 10 units), the other agent hits the ball back, and this agent is able to hit it one more time approximately around 60 time units. The cumulative rewards for the first agent is shown in (a) for two different values of γ, while the hit number is represented by the figure in (b).

Generating memories: Refreshing a memory buffer is a needed very often. Hence a function has been presented that will allow a set of two list of tuples to be generated. The function called `memories` is present in present in the file `src/utils/generateMemories.py`. This function and the ideas that went into designing this function will be briefly discussed in the following paragraphs.

Since we are able to calculate the cumulative reward for every point in an episode, this is exactly what has been done. An episode in played using a given `policy`[1] that will last at most N_{st} steps (or when a game stops).

Now, note that there is very little to be learnt from regions where the agent is unable to get a score. Hence, it would be great if we were able to learn from the region close to where the agent actually gets a point. For finding this reason, the cumulative rewards with a cumulative rewards is generated with a γ of 0.8 as shown in the Figure above. This shown that the cumulative rewards decreases exponentially, as we get farther away from the location of a successful hit. Here, we have found regions within the episode wherein this value of the cumulative reward remains above the value of 0.03. These are narrow regions demarcated by the thin gray lines in the Figures above. At least in the beginning, it is possible that it might be best to learn from within this region.

Sampling from memory: Although it might be possible to sample randomly form the memory buffer, a slightly smarter sampling methodology has been used. Sampling is performed based upon the cumulative rewards captured for each episode. The probability that a particular tuple i is sampled form the memory buffer is based upon the equation:

$$p_i = \frac{x_i + \epsilon}{\sum_i [x_i + \epsilon]} \tag{7}$$

Here, x_i represents the i^{th} parameter and ϵ is a small number that determines how strict we want the input to be used for learning. Several different things have been experimented upon to get parameters that are most relevant for this process. We shall discuss the different attempts in the experimentation section.

Note that there are two reasons for adding the value ϵ. First, this will prevent any possibility of division by zero. Next, it will be possible to also include values for which the cumulative reward is very small. Tuning this parameter down (say to 0) will p_i be the same as that represented by the bare c_i. On the other hand, using a large value of say 1000 results in a all tuples being approximately equally likely to be sampled.

4.2. Training the Model

The model is trained using a fairly standard algorithm. The only difference is that each of the agents learns individually. Practically all aspects of the training process can be controlled using the config files.

Training is performed in the file `src/utils/trainer.py`. As mentioned before, the configuration information present in `src/config.json` may be used to fine-tune practically all aspects of the training process. The parameters for the training process are shown below:

```
"training" : {
    "comment"           : "some comment about the current run",
    "initExplore"       : 0.3,
    "exploreDec"        : 0.9999,
    "exploreDecEvery"   : 100,
    "printEvery"        : 10,
    "totalIterations"   : 5000,
    "nSteps"            : 50,
    "memorySize"        : 5000,
    "sampleSize"        : 100,
    "fillReplayBuffer"  : 10000,
    "minScoreAdd"       : 0.11,
    "filterVal"         : 0.03,
    "hotStart"          : "path\to\saved\folder or `null`"}
```

[1] Generation of policies for both generating memories and training is discussed in Section 4.2.2.

4.2.1. Hot Start

It is possible that the model can start training from a previous point where the model was saved. This allows a previously trained model which was saved with different hyper-parameters aimed at training in one aspect well, to be loaded and trained again with another set of hyper parameters, which concentrates on another aspect of training that the model has not yet learnt properly. For this, make sure you set the `hotStart` parameter to the location of the folder wherein information for the previous data is saved.

4.2.2. Policy Generation

A policy can be generated either by only considering the results of the `Actors`, or by considering a random action for exploration. This is especially important when we are generating memories for the `ReplayBuffer` for each `Actor`. A special function has been created (called the `explorePolicy`) that takes a parameter called `explore` and returns a function that is either a random policy with a probability of `explore` and returns the result of an actor with a probability of `1 - explore`.

4.2.3. Fill the Replay Buffer

Before beginning the program, a long sequence of episodes is played that will fill the `ReplayBuffer` of each `Agent`. If `hotStart` is not used, it is recommended that this filling be done using exploration set to 1.

It is possible that the `ReplayBuffer` be replenished with memory that comes from a judicious mix of exploration and exploitation. For this, the parameters `initExplore`, `exploreDec` and `exploreDecEvery` are used. In the beginning, of training, the replay buffer is started with an exploration probability of `initExplore`. However, after every `exploreDecEvery` iterations, this parameter is multiplied by `exploreDec` to decrease the overall exploration rate. This way, the replay buffers are updated with more and more memories that make for better `Agents`.

As mentioned in Section 4.1.1.3., there are a lot of parameters that go into generating memories. This is because, while taking random actions, there is a great chance that random exploration will result in a miss. If we are to allow every single tuple to be inside the buffer, the `ReplayBuffer` is going to be filled with a vast collection of failed attempts. That is why, it would be wise to only allow parts of an episode that helps in the learning process, to accelerate learning. There are several parameters in the config file that govern this. The `minScoreAdd` parameter will prevent any episode that scored below a certain point to be added. Hence, if you want tuples to be added from episodes that had at least 2 hits, you would want this parameter to be 0.12. This will make sure that only episodes where a particular agent was able to hit the ball twice will be inserted. Furthermore, it might be important to only add tuples close to the hit. This can be accomplished by setting the `filterVal` parameter. This is the value that a cumulative sum value with a γ of 0.8 must cross before the tuple is added to the replay buffer. For generating values farther from the actual hit, lower this number.

4.2.4. Learning Algorithm

The learning algorithm can be described as follows:

1. If `hotStart` is available, load parameters for both agents
2. Set `explore` to `initExplore`
3. Replenish `ReplayBuffer` with data from `fillReplayBuffer` episodes
4. For every iteration in `totalIterations`
    a. Replenish replay buffer with memories from 10 episodes
    b. For every iteration in `nSteps`
        i. Sample `sampleSize` memories
        ii. Update weights of the `fastCritic`
        iii. Update weights of the `fastActor`
    c. Update the `slowActor`
    d. Update the `slowCritic`
    e. Using the `slowActors` calculate the score of one episode

f. If the score is ≥ all previous scores, save this state in a `tmp\` folder.

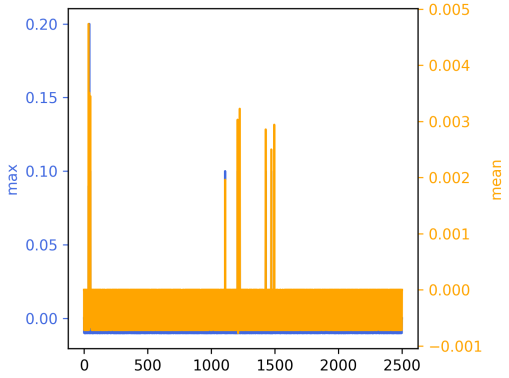
5. Plot all scores, `aLoss`, `qLoss`
6. Save the final model, the config file and the plots in the same folder

4.3. Experiments

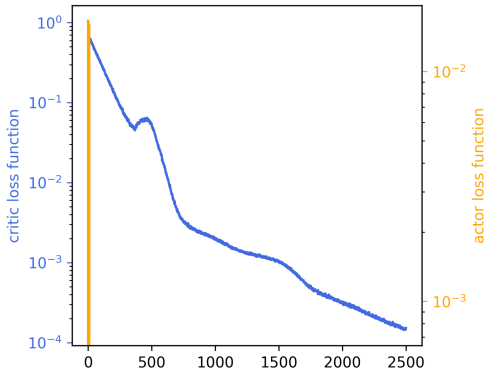
Given the versatility of the learning algorithm, it is possible to do short experiments with different types of inputs. Here are some experiments and what they looked like …

4.3.1. Simple run

This is a fresh run and went through 2500 episodes. The `initialExplore` was set to 1.



Scores over every iteration

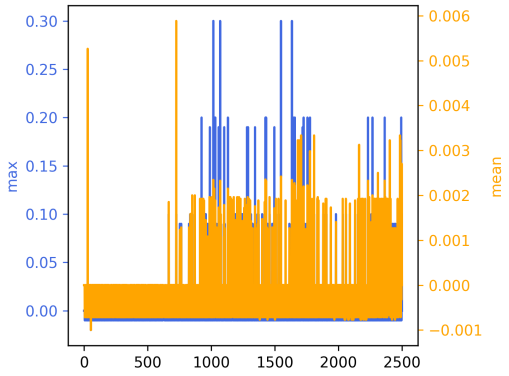


Loss function

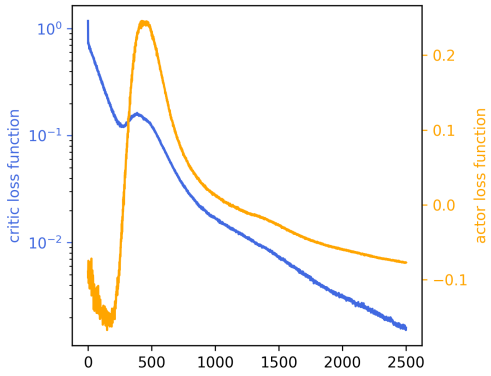
This attempt resulted in a fairly simple structure. The cumulative reward was calculated with a γ of 0.8. This is not the real cumulative reward, since the reward should be that due to 1. At this time, we were using cumulative sum for sampling. This is the x_i in eq. (8). Hence, it was mainly concentrating on tuples which were really close to the actual hit.

4.3.2. Updated γ

Now, we try the same thing. However, we realize that the value of γ for any calculation should not be 0.8, but rather 1. This will allow hits much earlier in the process to be used for sampling. The calculation was amended such that the tuple that is returned uses a γ of 1. The calculation of the filtering value still uses the old value of γ.



Scores for every iteration

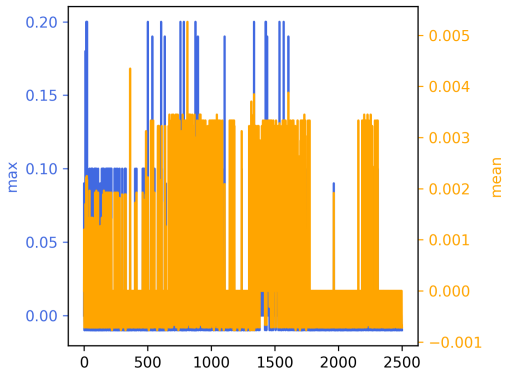


Loss functions

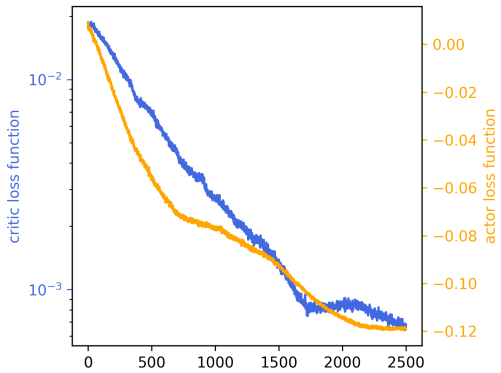
As can be seen, the scores immediately improved. There were times when the paddle was able to get in 3 hits, and was getting 2 hits consistently, and definitely a lot of single hits. This was very encouraging. It is interesting

to see the actor loss function. This is basically the negative of the Q value. It starts negative, quickly goes to positive, and becomes negative again, approximately around iteration number 1000. It appears that even though the initial loss function was negative, it was actually very far from the truth, as the actor was terrible. As the actor improved, the critic was able to start representing the actual Q value. This is encouraging. Although the system is thought of as not solved, this looks very promising indeed.

4.3.3. Start from the previous solution and retrain for a longer time



Scores for every iteration



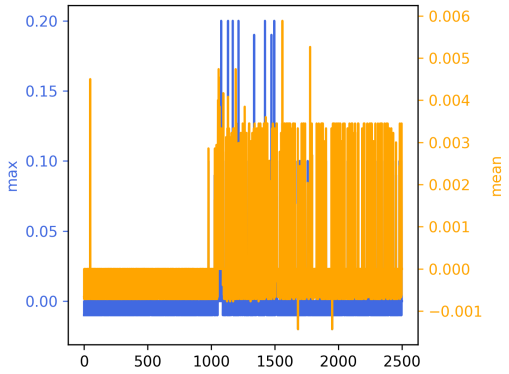
Loss functions

So although we see a fairly good decrease in the actor and critic loss functions, the results from the loss functions do not seem to be meaningful. We do get a lot of two hits. However, this is not working. Something is not right. It is able to hit the first ball but not subsequent balls. So, when the ball drops vertically, it seems ok. But, when it comes from the front, it cannot handle that?

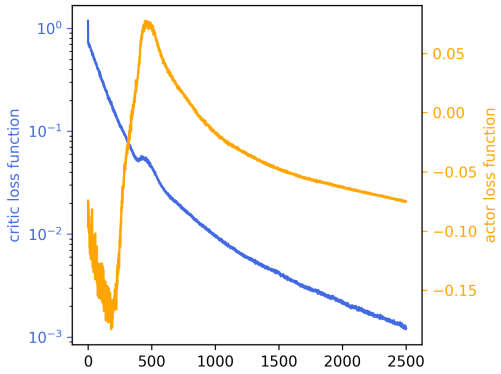
4.3.3. Say Hello to `numHits`

If the paddles are training more on the ball drops, would it help if they trained more on tuples were second and third hits? For this, we introduce the concept of `numHits`. This represents the hit number that a particular tuple belongs to within the episode. Look at Fig. 5. (b) for an intuitive explanation.

Given this, the sampling algorithm was modified such that the x_i in eq. (7) now represents `cumRewards + numHits`. Will this bring us luck?

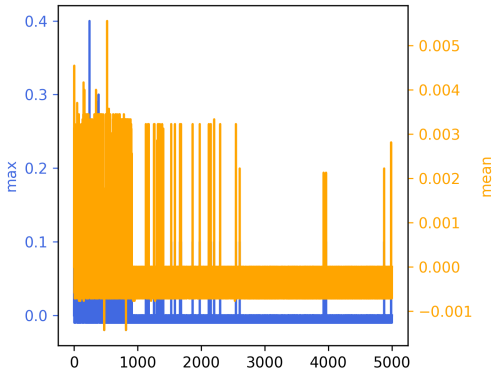


Scores for every iteration

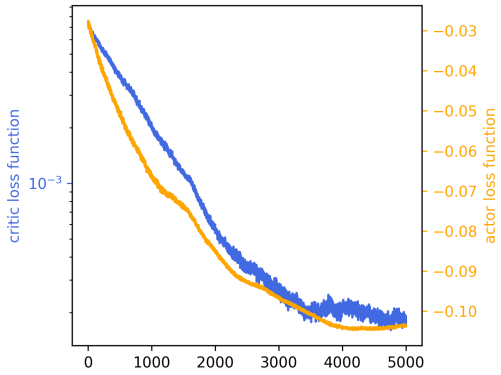


Loss functions

Not much of an improvement, but it does appear to be much more stable. Let's try a longer run for this:



Scores for every iteration

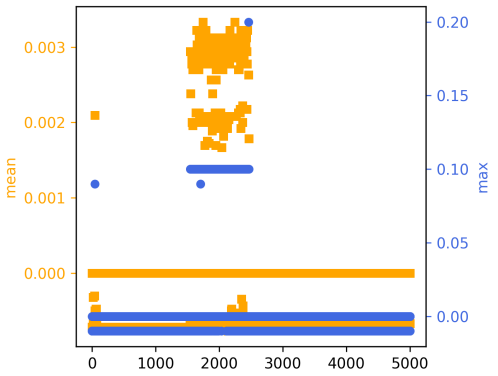


Loss functions

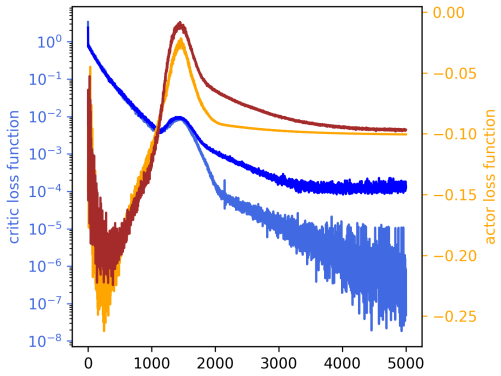
Well, we have our first 4. However, still not very good or stable. This is starting to get depressing.

4.3.4. Incorporate some failures

After watching some of the videos, we realized that it is possible, we are not incorporating the failures within our calculations. Hence, we incorporate a little bit of failures within the mix. Approximately at the 2000 mark, we see that the actor functions basically flatten out.



Scores for every iteration



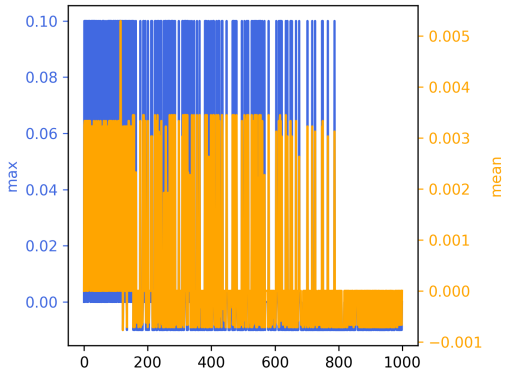
Loss functions

At this point, it is obvious that there is a difference between `Agent_0` and `Agent_1`. It turns out that `Agent_0` can hit the initial ball every time. However, `Agent_1` sucks at it. If we initialize both agents with values of Agent_0, then we at least start them off with some interesting results …

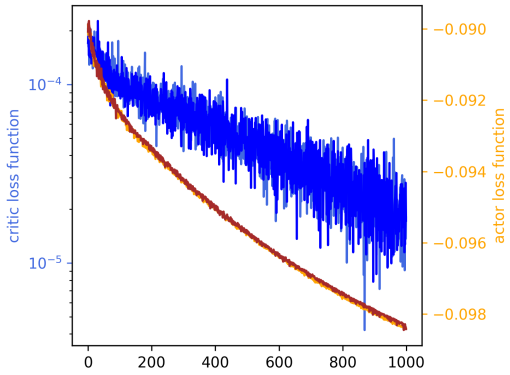
- With separate agents, they look like the following: https://youtu.be/ryhIdbBlKz8
- With Agent_0 for both, they look like the following: https://youtu.be/ZkZjs1Gq0yU

Hence, for the next run, we start at the 2000 time unit with the same agent: `Agent_0.` We want to check whether we are able to generate better result. Note that basically both actor losses are saturating to -0.1. This means that the critic is mostly predicting a single hit. Furthermore, the critic loss seems to have significant instability. A sign that the critic has perhaps a significant learning rate.

It turns out that the instability in the learning rate is significant and cannot be used for the problem. This is even worse …



Scores for every iteration

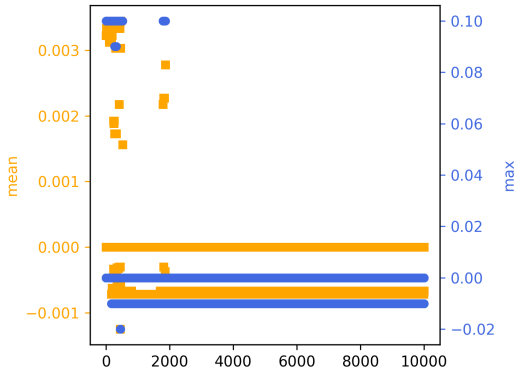


Loss functions

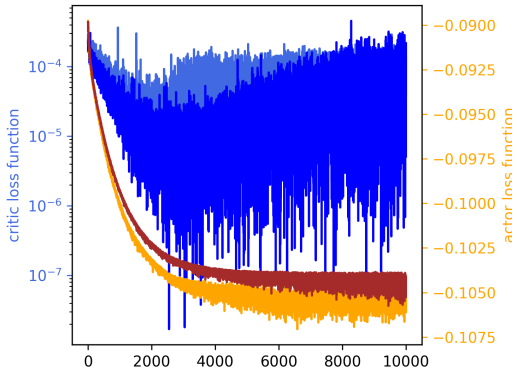
Losses are decreasing. However, we are not learning the right thing.

4.3.5. Start with the same agent, but do a very long run

We have changed the program so that, after the first hit, the program switches to a 50% exploration function. Even this doesn't seem to be working …



Scores for every iteration

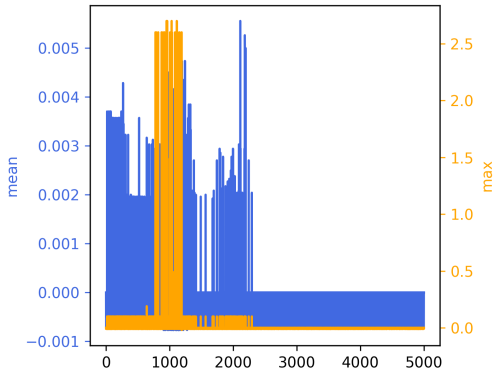


Loss functions

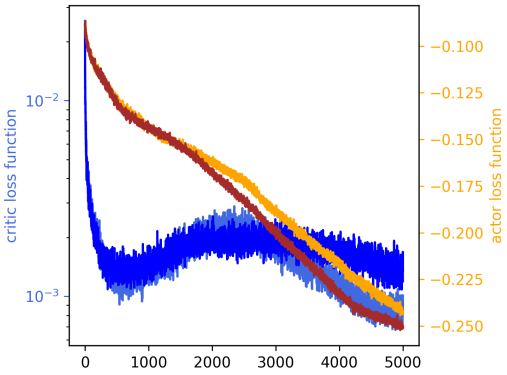
Looks like there is very little improvement for the actor loss function. We need some other method of improvement …

4.3.1. Change the NN Architecture

Now that everything failed, I tried another set of NN architecture. Now, the architecture had 3 layers, and each layer is now activated with `tanh` activation (rather than the earlier `ReLU`) to avoid the exploding gradients problem. Since there are only 3 layers, I didn't expect significant problems with vanishing gradients. This looks more promising, as shown below:



Scores for every iteration



Loss function for every iteration

It appears that the scores hit a maximum of 2.6 and 2.7 around iteration 1000 on a fairly consistent manner. They however quickly deteriorate because of the inherent instability of the problem.

5. Results

Finally, this model appears to have been trained. Although this appears to be a project about collaboration, it was fairly simple to encode this in a form that looked like an independent problem. The results are not perfect. However, it appears to be trained for a fairly good set of initial conditions.

6. Future Work

A lot needs to be done to improve this model. Especially with the initial conditions. Some things that are worthwhile trying are as follows:

1. I shall try to play with the different parameters, and also run the model for much longer to see whether the model magically learns something, as in the case of the course instructors.
2. I shall also try to see whether it is possible to combine the last few views to generate something meaningful as input, rather than just using the latest observation. This means that the agent will have knowledge of not only the current observation, but also the last few observations. This should allow the agent to learn a much better (as described in https://blog.openai.com/better-exploration-with-parameter-noise/).
3. I also want to change the exploration by adding noise to the weights, rather than obtaining a random vector. There are some papers that look at adding Gaussian noise to the action, and others that add noise to the weights. Apparently adding noise to the weights are much better. We can explore those options.
4. I have used tanh activation for avoiding the exploding gradients problem. I would like to try gradient clipping along with `ReLU` units.

7. Acknowledgements

`https://github.com/jknthn/unity-tennis-rl`