safdark / **DRL-MultiAgent-Collaboration**

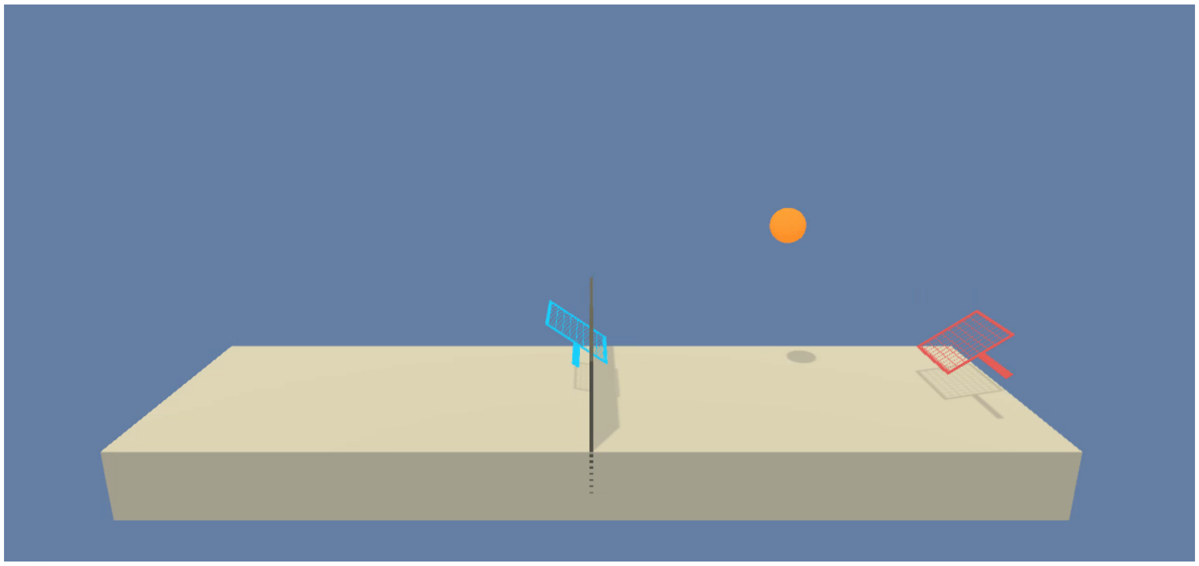
Branch: **master** **DRL-MultiAgent-Collaboration** / **DESIGN.md** Find file Copy path

safdark Update DESIGN.md 3e2a491 a minute ago

1 contributor

213 lines (143 sloc) 13.4 KB

Collaborative Multi-Agents -- Tennis



- Better GPU utilization
- Exploring model-free agents better suited for multi-agent environments

Overview

Multi-agent games involve 3 possible environments -- competitive, collaborative and mixed. The motivation for this project was to implement a reinforcement learning algorithm in a continuous state-space and continuous action-space wherein two agents are playing *collaboratively*.

Environment

As mentioned in README.md, the environment being worked in here is of Tennis.

In this environment, 2 agents are playing tennis *with* each other, and are collaborating because their individual rewards and punishments are ultimately scored as a team. For any player that hits the ball back into the other court, the reward given is a +0.1. For missing a ball, or hitting it out of bounds, the punishment is a -0.05. At the end of the game, the 2 agents are scored as a team, with the score being the maximum of their individual scores. The goal of the trained agents is to maximize their team score, the strategy for which involves keeping the ball in play for as many steps as possible.

Action Space

Each action is a vector with 2 numbers (both between -1 to +1): One corresponding to how far forward or backward the given agent is to move on its own court, and the second controlling how high to jump. For example:

```
[ 0.00212 -0.2234]
```

Observation Space

Each agent observes the system from its own perspective, and, as such, receives only a partial observation of the system. Each observation consists of 24 elements that comprise measurements for the positions and velocities for both the agent and the ball, in 3-D coordinate axes. For example:

```
Agent 1:
[ 0.          0.          0.          0.          0.          0.
   1.          0.          0.          0.          0.          0.
   2.          0.          0.          0.         -6.65278625 -1.5
  -0.          0.          6.83172083  6.         -0.          0.        ]

Agent 2:
 [ 0.          0.          0.          0.          0.          0.
   1.          0.          0.          0.          0.          0.
   2.          0.          0.          0.         -6.4669857  -1.5
   3.          0.         -6.83172083  6.          0.          0.        ]
```

As can be seen, each agent only partially observes the state of the system from its own perspective. Furthermore, the observation space is non-stationary, since the actions chosen by each agent are not the only actions that impact the observed states at each point -- it is also the action of the other agent that similarly influences each observation (and vice-versa).

Non-stationary environments do not hold any convergence characteristics, which means that there is no guarantee that the agents will settle on an optimal strategy by training longer using single-agent approaches.

Goal

The average score of the learned policy (averaged over 100 episodes) should be 0.5 or higher.

Design

I have structured the solution using the below components (classes).

Driver (driver.py)

This is the command-line entry-point to launch the training system. The goal was to mimic the IPynb Notebook environment without having to use the notebook, which greatly facilitated development and testing.

Trainer (trainer.py)

Encapsulates the train() and play() routines.

The train() routine runs the simulation in train mode, and coordinates the iteration through episodes, and the interplay between the Environment and the Agents.

The play() routine runs the simulation in play mode, for just one episode, querying the (presumably trained) agents for the next action to take for every environment state observed.

Tracker (tracker.py)

Encapsulates, collects and tracks various metrics through the training process, and can be used to generate various graphs about the training performance.

Agent (agent.py)

Encapsulates the Deep Deterministic Policy Gradient algorithm in reinforcement learning, with the following key components:

- Experience replay buffer: A deque that stores experience tuples (state, action, reward, state', done)
- Noise generator: To add stochasticity to each chosen action, a noise vector of the same size is generated using an ornstein-uhlenbeck noise process. And, to ensure that each agent does not utilize the same noise, I generate a different noise vector for each of the two action vectors generated on every step.
- Learning: This is the meat of the ddpg implementation, utilizing 2 neural networks -- one for the Actor and another for the Critic (discussed next).

Actor & Critic Networks (model.py)

The Actor is a neural network used to directly learn the optimal policy (low variance, low bias) for the environment.

The Critic is a neural network that learns the value-based function (low variance, low bias) of the environment, and is used as a stabilizing supervisor for the policy learned by the Actor. The agent also uses the Actor's selected actions and corresponding rewards to update the Critic's network (to fine tune its approximation of the environment's value function).

Regularization

I have added 2 regularization techniques to help overcome the limitations of a single-agent learning algorithm -- in this case, the DDPG algorithm.

Data Augmentation - Observation Disambiguation (feature_extractor.py)

Instead of using a purely multi-agent algorithm (such as MADDPG), I chose instead to tweak the DDPG agorithm itself, to suit this particular multi-agent environment. This solution may not, however, work forevery multi-agent scenario.

Whereas the MADDPG algorithm uses the concept of a joint action space, my approach uses the concept of a joint *state-space*, which essentially serves like a complete observation of the environment -- that is, instead of getting partial observations for a seemingly non-stationary environment, my approach augments the observation with parameters that allow the network to learn to associate any "confusion" (due to non-starionarity) to those additional parameters, thereby learning a sufficiently robust policy.

More specifically, since state observations received by the system at each step are from 2 different perspectives, the system will merely confuse itself trying to learn contradictory behaviors to seemingly similar states. The workaround was to disambiguate between observations for each agent by adding an agent-identifier to the start of each observation vector. This increases the length of each observation vector from 24 to 25 elements -- the first element being 0 for the first agent, and 1 for the second. For example, the sample observation from above, after being augmented, would look like below:

```
Player 1:
[ *0.*          <-------------- First Player
   1.          0.          0.          0.          0.          0.
   2.          0.          0.          0.          0.          0.
   3.          0.          0.          0.         -6.65278625 -1.5
  -0.          0.          6.83172083  6.         -0.          0.         ]
```

```
Player 2:
 [ *1.*         <-------------- Second Player
   1.          0.          0.          0.          0.          0.
   2.          0.          0.          0.          0.          0.
   3.          0.          0.          0.         -6.4669857  -1.5
   4.          0.         -6.83172083  6.          0.          0.        ]
```

This data augmentation serves as a regularization that allows the system to correlate the conflicting learning examples to that first element, allowing the network to learn 2 different behaviors for seemingly similar observations, based on the value of that first element. This was a crucial choice for this learning algorithm.

Network Dropouts

An important element in both the Actor and Critic networks was a Dropout (with probability of 0.2), after the hidden layer, to help the networks generalize better. This regularization technique was a necessary addition to the data augmentation done above, because, intuitively, it allowed the agent to better utilize that disambiguating element above, with learning that was generalized over the remaining 24 observation elements, since those 24 elements would have to serve 2 different purposes, using the first element as the "switch", so-to-speak.

Training

Without using any regularization, the agent's learning was rather unstable, with the 100-episode average swinging up and down as training progressed, without ever hitting the 0.5 mark. Using the regularization techniques above, allowed the agents to learn to play sufficiently to reach a 100-episode average score above 0.5.

I utilized a GPU (NVidia Titan Xp) for the training, which significantly helped the speed of training, and was able to get to 35% peak GPU utilization.

Hyperparameters

The following hyperparameters were used:

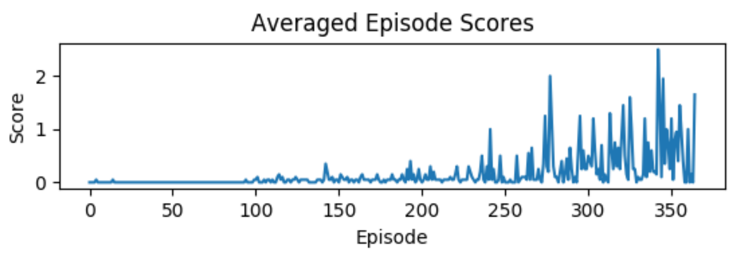
```
BATCH_SIZE = 514        # minibatch size
BUFFER_SIZE = int(1e5)  # replay buffer size
GAMMA = 0.99            # discount factor
TAU = 0.15              # for soft update of target parameters
LR_ACTOR = 1e-3         # learning rate of the actor
LR_CRITIC = 1e-3        # learning rate of the critic
WEIGHT_DECAY = 0.00     # L2 weight decay
```

Results

As the graphs below illustrate, the agent learned the problem space pretty well, achieving the goal score in a little over 100 episodes. See here for the training log.

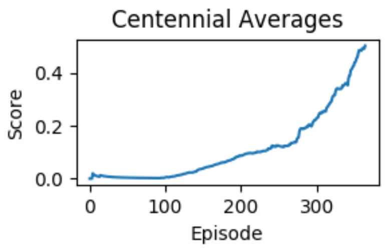
Average Agent Scores

The average scores for the 2-person team, at the end of each episode, are plotted here. Note how the learning is slow until episode 235, after which the agents begin to learn faster, attaining a max score of ~3.0 -- that's keeping the ball in play for 30 to-fro returns!



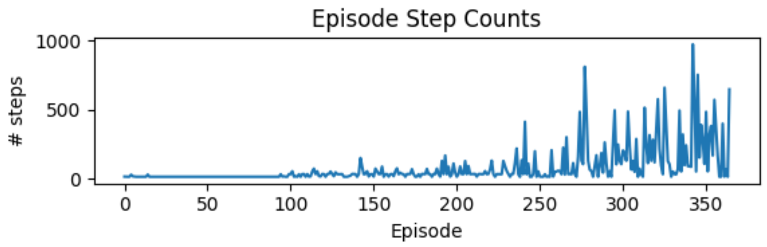
Running 100-episode averages

This is a clean and smooth curve that agrees with the Average scores graph above.



Episode step counts

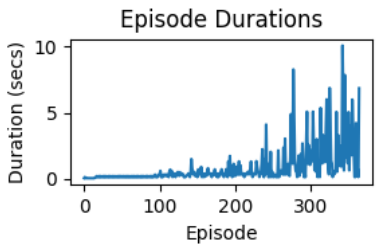
Given the specifics of game play here, this makes perfect sense, since higher score corresponds directly to more steps where teh ball is kept in play.



Episode duration

There are 2 factors at play here, when we see the episode durations:

1. The number of steps that the agents keep the ball in play (whose average is increasing), AND
2. The consumption of the replay buffer, which is sampled from, since sampling time is linearly correlated with the length of the sample pool ((O(n) runtime).



Future Enhancements

Prioritized experience replay

The learning rate could be further improved using this, especially since it would help the learning algorthim remain more stable by focusing on experiences that achieved a more desirable outcome.

Better Tooling

This is not a very user-friendly application at the moment, since the focus has been on exploring the learning process via iPython Notebooks. However, adding command-line parameters to launch the training and simulations from the shell would signficantly improve its usability.

Better GPU utilization

The GPU capped at ~35% utilization with my present setup. I would like to explore ways to increase that utilization to favor faster learning.

Exploring model-free agents better suited for multi-agent environments

Such as MADDPG.