Udacity Deep Reinforcement Learning Nanodegree Project

Continuous control

Author: Piotr Bazan

1. Introduction

In order to solve the Continuous control project I came up with an idea to split different parts of the problem into separate pieces (concerns) and implemented it in the object-orientated way so that each component could be either exchanged or re-implemented. This allowed experimenting and being open for future improvements.

2. Components

Experiment

`Experiment` is a central class for conducting training and evaluation. In order to create it we need an agent and an environment.

- agent - there is only one implementation - `DDPGAgent`
- environment - there is a wrapper around unity environment - `UnityEnv`

Environment

`UnityEnv` is an environment with API similar to openAi gym. This allows an abstraction over Unity environment.

Agent

`DDPGAgent` class implements DDPG reinforcement learning algorithm and requires several components:

- actor - a Pytorch model for an actor – currently there is one model `FCDP` (fully connected deterministic policy)
- critic – a Pytorch model for a critic - currently there is one model `FCQV` (fully connected Q-value)
- memory - a reply buffer – currently only there is one buffer `ReplyBuffer`
- train_strategy – a strategy that supports continuous actions i.e. child of `ContinuousActionBaseStrategy` for example `NormalNoiseDecayStrategy`

The agent works in two modes: train and evaluation. During training is uses online actor to pick action which is passed to the environment. The resulting next state, rewards, terminal information and state are stored in reply buffer.
Once the reply buffer is filled with enough data (batch size * warm up batches) there is a sampling of data and both actor and critic are trained. During training online critic is used to evaluate the next state Q-value. After each training target actor and critic adjust their weights according to Polyak averaging of online actor and critic.
In evaluation mode the online actor is used to pick actions.

The FCDP model – the actor

Model consists of:

- input layer
- hidden layers
- rescale layer

The input layer accepts environment state as an input, then it applies RELU activation and passes result to hidden layers. Each of the hidden layer except the last one applies RELU. Last layer calculates TANH activation so that actions are distributed within <-1; 1> range. The rescale layer shifts action values to range required by the environment.

The FCQV model – the critic

Model consists of:

- input layer
- hidden layers
- output layer

The input layer accepts environment state. Than it calculates RELU activation. The results is concatenated with action tensor and put through hidden layers, each completed with RELU activation. At the end result is passed to output layer which does not have any activation function so that the Q-value is unlimited.

ReplyBuffer

Memory is a reply buffer (cyclic buffer) with limited size - I have chosen limit of 50_000 items.

Action picking strategy

Agent requires training and evaluation strategy.

- for training the `NormalNoiseDecayStrategy` – picks actions using actor and adds noise (with normal distribution, mean=0, standard deviation=.1). The class support also decaying mechanism so that noise (which can be seen as exploration) is decreased with each episode
- for evaluation the `ContinuousGreedyStrategy` – picks actions using actor. This can be seen as greedy strategy since there is no noise added

3. Hyper parameters for the agent

The agent has the following components:

- `actor = FCDP(nS, action_bounds, (256, 256)) – two hidden layers with 256 units`
- `critic = FCQV(nS, nA, (256, 256)) – two hidden layers with 256 units`
- `memory = ReplayBuffer(50000) – buffer size 50000`

where:

- `nS – state space size`
- `nA – action space size`
- `action_bounds – mininum and maximum values for each action`

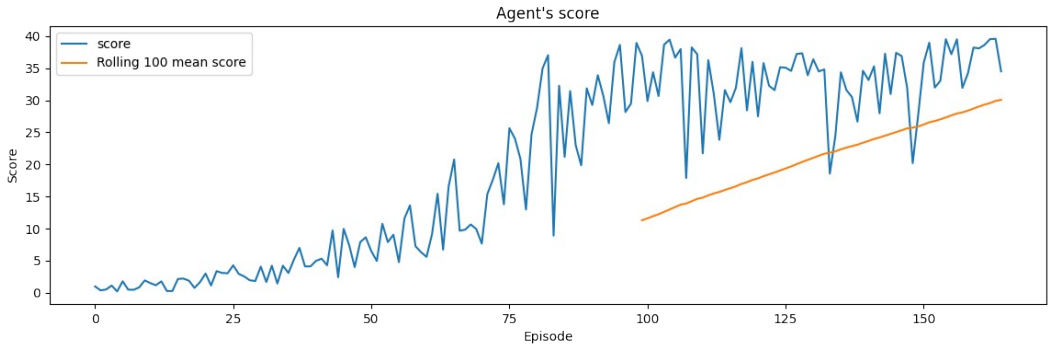
The agent has the following parameters:

- `gamma=.99 – discounted rate`
- `batch_size=128 – number of samples for training`
- `warm_up_batches=5 – required number of batches before training happens`
- `actor_lr=1e-4 – actor learning rate`
- `critic_lr=3e-4 – critic learning rate`
- `actor_max_grad_norm=float('inf') – actor maximum gradient during back propagation`
- `critic_max_grad_norm=float('inf') – similar to the above`
- `train_every_steps=1 – number of steps before training occurs`
- `update_target_every_steps=1 – number of steps before update occurs`
- `tau=.005 – weight for updating target actor/critic with online actor/critic parameters`

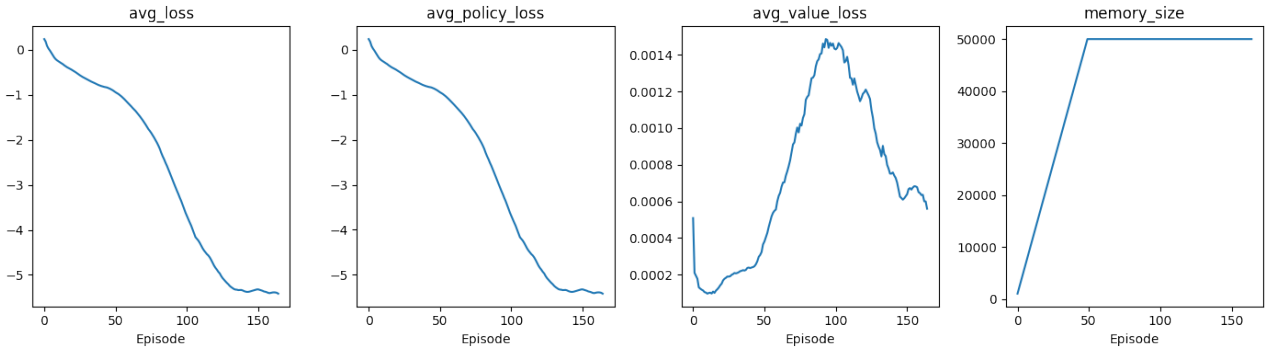
4. Experiment results

The experiment had target of mean of 30 points in 100 consecutive trials.

Agent passed grading achieving min score: 7.67, mean score: 30.07. Number of required episodes is around 160.

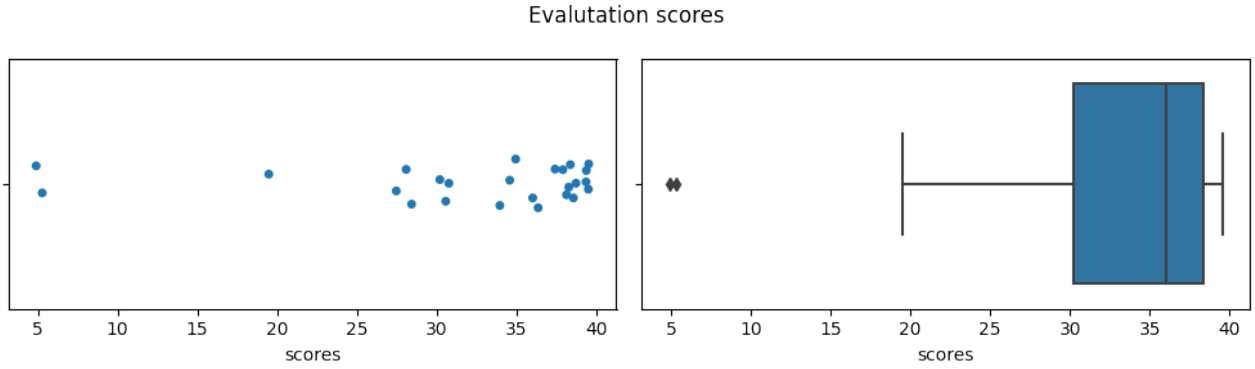


The agent had the following statistics:



5. Evaluation

I evaluated agent on 25 episodes and obtained the following results.



The mean is above 35 points, the lowest score is around 5.

6. Ideas for future work

- Hyper parameters tuning
    - learning rate – currently actor and critic have different learning rates. I would check same rates and increase them to speed up learning
    - train/update every episodes – I would choose higher value to check how models behave, whether is can train quicker
    - tau – other values to consider .01 and .1 – see which gives better results
    - maximum gradient clipping – I used unlimited (infinity), also I checked value of 1 but the model learned really slowly. Perhaps some values like 10 would result in quicker and more stable learning
- model architecture
    - currently actor and critic are similar in terms of hidden layers size. I would check different sizes for each model