Project 3 Collaboration and Competition

Algorithm

Deep Deterministic Policy Gradient (DDPG) this algorithm continuously improves the policy by exploration of the environment. It applies gradient descent to the policy to a sampled data sampled from a replay pool. For this project I have created two agents and make the state information shared among them. Since states of both agents are merged replay pool for both agents is same.

I have used the modified DDPG implementation in my previous project. I have only changed the training procedure in the main notebook for calling the step function for each agent. Merged the states of both agents into one.

After this change I have seen that it is possible to reach average of 0.5 in last 100 episodes.

When I tested the agent I have seen that the weights produce higher scores.

Implementation details are as follows:

Model.py: two fully connected networks for Actor and Critic from bipedal implementation. (I have also tested with pendulum model but this one was reaching higher scores)

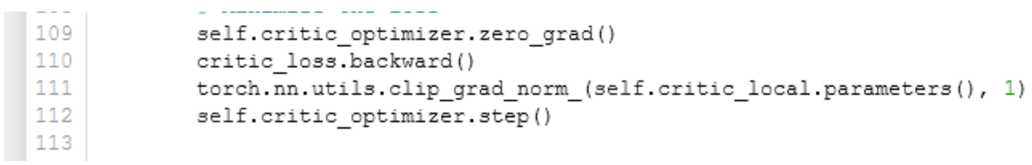
- Actor: two fully connected layers.
    - First layer: input size = state size, output is equal to 256. A 1d batch normalization followed by a RelU activation is used.
    - Second layer: input size 256, output size is equal to action size. A tanh activation function is applied to the output.
- Critic: Four fully connected layers are used. For the first connected layer a 1d batch normalized is utilized before a leaky RelU activation which is more balanced and has a faster learning rate.

Dpg_agent.py: A DDPG agent implementation. Following are the changes applied to the implementation provided during classes.

- Small changes to hyper parameters.

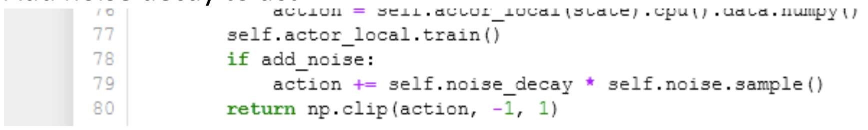
BUFFER_SIZE = int(1e5)	# replay buffer size
BATCH_SIZE = 128	# minibatch size
GAMMA = 0.99	# discount factor
TAU = 1e-3	# for soft update of target parameters
LR_ACTOR = 1e-3	# learning rate of the actor
LR_CRITIC = 1e-3	# learning rate of the critic
WEIGHT_DECAY = 0	# L2 weight decay
TRAIN_EVERY= 1	# Apply training after every step
TRAIN_NUM = 1	# number of training after every TRAIN_EVERY
NOISE_DECAY = 0.99	# decay parameter for NOISE

- Suggested change in benchmark implementation for clipping to the learning



```
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        torch.nn.utils.clip_grad_norm_(self.critic_local.parameters(), 1)
        self.critic_optimizer.step()
```

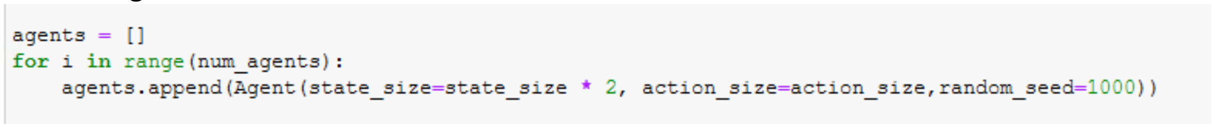
- Agent trains in very step. Since the step size of the starting episodes are very small agents are not trained otherwise.
- Add noise decay to act



```
            action = self.actor_local(state).cpu().data.numpy()
        self.actor_local.train()
        if add_noise:
            action += self.noise_decay * self.noise.sample()
        return np.clip(action, -1, 1)
```

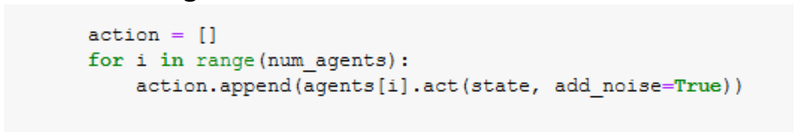
Tennis.ipynb: Main loop for test and training.

- Create two agents



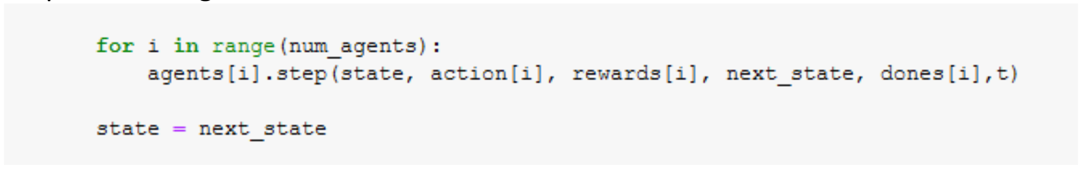
```
agents = []
for i in range(num_agents):
    agents.append(Agent(state_size=state_size * 2, action_size=action_size,random_seed=1000))
```

- Act for both agents



```
    action = []
    for i in range(num_agents):
        action.append(agents[i].act(state, add_noise=True))
```

- Step for both agents



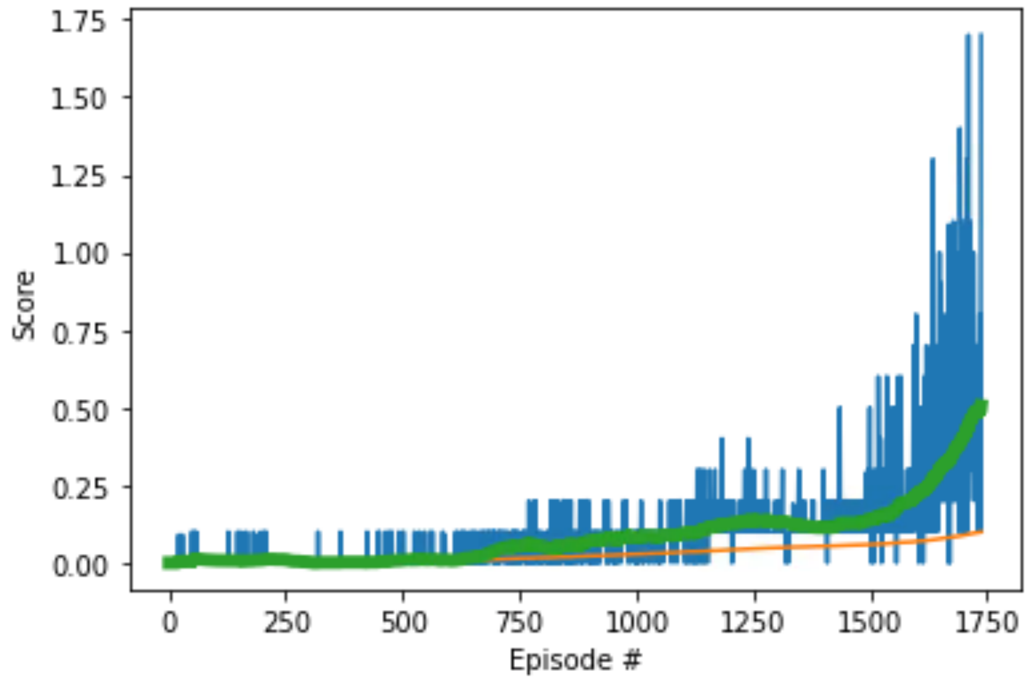
```
    for i in range(num_agents):
        agents[i].step(state, action[i], rewards[i], next_state, dones[i],t)

    state = next_state
```

Results

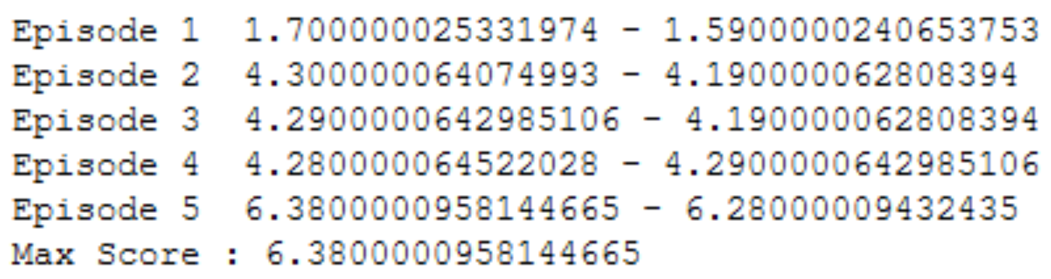
An average of 0.51 for last 100 episodes is achieved after 1737 episodes.

```
Episode 1727	last 100 Average Score: 0.48
Episode 1728	last 100 Average Score: 0.49
Episode 1729	last 100 Average Score: 0.48
Episode 1730	last 100 Average Score: 0.49
Episode 1731	last 100 Average Score: 0.49
Episode 1732	last 100 Average Score: 0.49
Episode 1733	last 100 Average Score: 0.49
Episode 1734	last 100 Average Score: 0.49
Episode 1735	last 100 Average Score: 0.49
Episode 1736	last 100 Average Score: 0.49
Episode 1737	last 100 Average Score: 0.51
```

Following is the score graph where blue is the score of the episode, green is the average of last 100 episodes and orange is general average.



I have also tried the trained agent and I have seen that it can play the game very well. Following is an example episode. Agents play very long rallies. I have also captured a video of this episode and recorded as demo2.wmv.



```
Episode 1  1.700000025331974 - 1.5900000240653753
Episode 2  4.300000064074993 - 4.190000062808394
Episode 3  4.2900000642985106 - 4.190000062808394
Episode 4  4.280000064522028 - 4.2900000642985106
Episode 5  6.3800000958144665 - 6.28000009432435
Max Score :  6.3800000958144665
```

Future and Extra Work

Instead of using two agents another class can be created to merge replay buffer of both agents. This class can use only both agents' models.

I have tried this implementation, but it was not training very well and possibly state conversion was not correct. I will continue to test that also.

I am planning to try Soccer environment as well.