Project 2 Continuous Control

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.

I have used the DDPG implementation provided for bipedal environment. I have only changed the training procedure in the main notebook for calling the step function for each agent.

I have first implemented the suggested implementation in 'Benchmark Implementation' but I tried many different parameters I could not achieved the performance. I have added a batch normalization to both actor and critic networks it achieved scores close to 30 but it could not keep it for 100 episodes. Basically, I have seen that even though algorithm reach the target It could not keep it and quickly collapses. What I understand is that added noise makes increases the exploration even though algorithm reaches the expected performance. I have implemented a decay in the noise, and I was able to achieve the required performance.

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= 20 # Means a training performed after every 20 steps
- 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

```
109         self.critic_optimizer.zero_grad()
110         critic_loss.backward()
111         torch.nn.utils.clip_grad_norm_(self.critic_local.parameters(), 1)
112         self.critic_optimizer.step()
113
```

- Change in the step action to avoid training in every step

```
59     def step(self, state, action, reward, next_state, done,t):
60         """Save experience in replay memory, and use random sample from buffer to learn."""
61         # Save experience / reward
62
63         self.memory.add(state, action, reward, next_state, done)
64
65         # Learn, if enough samples are available in memory
66         if len(self.memory) > BATCH_SIZE  and t % TRAIN_EVERY == 0:
67             for _ in range(TRAIN_NUM):
68                 experiences = self.memory.sample()
69                 self.learn(experiences, GAMMA)
70
```

- Add noise decay to act

```
77         self.actor_local.train()
78         if add_noise:
79             action += self.noise_decay * self.noise.sample()
80         return np.clip(action, -1, 1)
```

Continous_control_play.ipynb: Main loop for test and training

Results

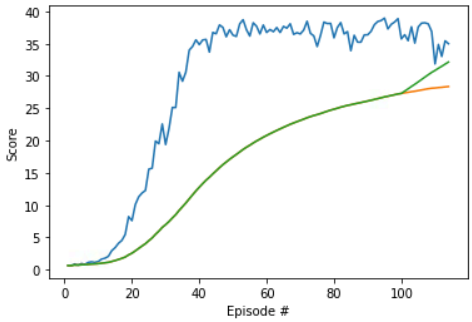
An avaregae of 32 achieved after 114 episodes.

```
Episode 111     Average Score: 34.88
[38.56999914 29.45999934 32.38999928 29.75999933 38.00999915 36.26999919
 36.17999919 33.62999925 28.15999937 35.29999921 30.00999933 38.71999913
 27.54999938 24.52999945 30.95999931 26.52999941 28.88999935 37.85999915
 39.51999912 38.19999915]
Episode 112     Average Score: 33.02
[29.15999935 39.67999911 35.7699992  36.75999918 33.59999925 35.31999921
 39.49999912 39.00999913 38.04999915 38.77999913 35.6099992  28.24999937
 35.48999921 38.21999915 32.33999928 32.39999928 29.50999934 35.32999921
 37.52999916 38.55999914]
Episode 113     Average Score: 35.44
[36.93999917 32.23999928 34.60999923 35.25999921 32.98999926 36.35999919
 37.73999916 36.28999919 33.28999926 36.98999917 36.31999919 34.58999923
 34.59999923 35.01999922 32.47999927 34.72999922 36.39999919 36.39999919
 32.41999928 34.28999923]
Episode 114     Average Score: 35.00
```

In my previous trials I have seen that algorithm was collapsing after reaching 30+ to be able to see more episodes I have increased the target to 32.

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.

```
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(1, len(scores)+1), scores)
plt.plot(np.arange(1, len(scores)+1), avg_scores)
plt.plot(np.arange(1, len(scores)+1), last100_avg)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```



I have also tried the trained agent and I have seen that it can play the game very well. Following is an example episode. All he scores are above 30. I have also captured a video of this episode and recorded as demo3.wmv.

```
Number of agents: 20
Size of each action: 4
There are 20 agents. Each observes a state with length: 33
The state for the first agent looks like: [ 0.00000000e+00 -4.00000000e+00  0.00000000e+00  1.00000000e+00
 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00 -1.00000000e+01  0.00000000e+00
  1.00000000e+00 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  7.90150833e+00 -1.00000000e+00
  1.25147629e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
 -5.22214413e-01]
[34.56999923 31.00999931 37.52999916 39.59999911 36.49999918 39.52999912
 39.21999912 33.46999925 32.12999928 39.48999912 39.49999912 37.87999915
 37.17999917 34.97999922 32.33999928 33.06999926 34.40999923 39.10999913
 39.46999912 36.63999918]
```

Future and Extra Work

I have tried the crawler environment and try to understand it is possible to solve it with a DDPG I could not produce more than 30 average even though I tried with different network models and parameters.
I have tried PPO implementation since it seems more successful with higher dimensional continuous action spaces. With PPO initial implementation I was able to reach 300+ average scores. I did not try to optimize it, but I will try later on.
As a future work I am planning to test the Reacher environment with PPO and compare the results.