Collaboration and Competition Project

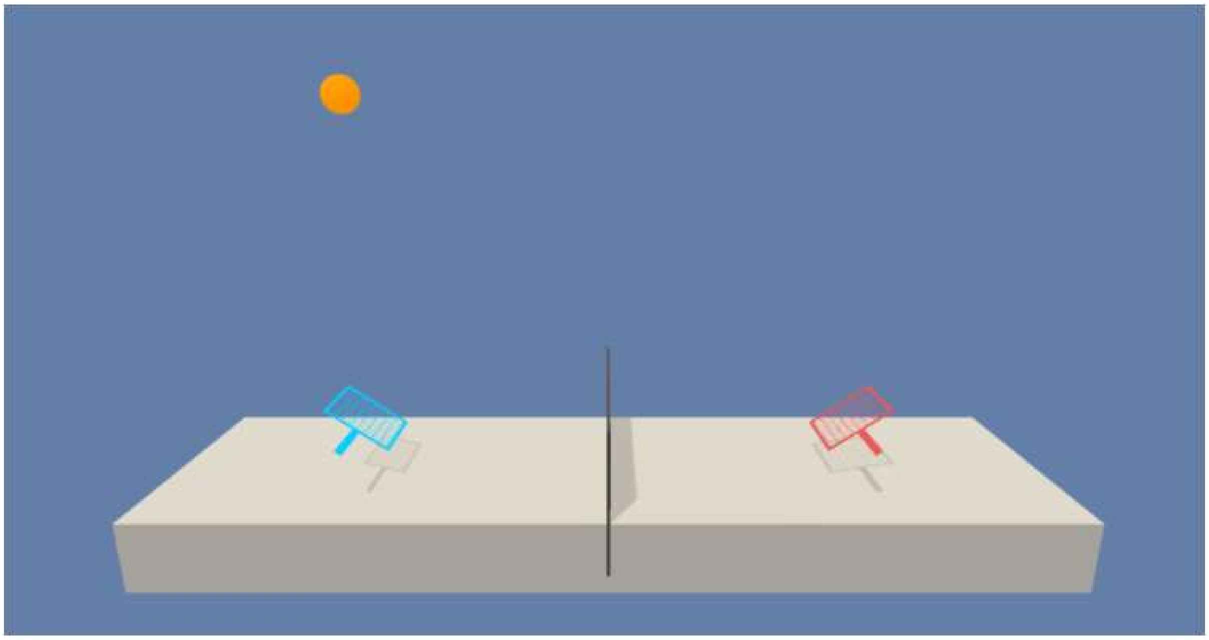
Udacity Deep Reinforcement Learning

Dhar Rawal (dharrawal@gmail.com)

2/24/2021

Introduction

In this environment, two agents control rackets to bounce a ball over a net. They cooperate with each other to keep the ball in play as long as possible.



In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. The observation space consists of 8x3 (24) variables corresponding to the position and velocity of the ball and racket for 3 consecutive time-steps. **Note that getting the data for 3 consecutive time-steps is a huge benefit to training the agent. It gives some of the benefits of n-step (3-step in this case) learning even though I have not implemented true n-step learning here**.

Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping. ***In other words, the action space is continuous***.

The task is episodic, and in order to solve the environment, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically, after each episode, we:

1. Add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores.

2. We then take the maximum of these 2 scores.

This yields a single score for each episode. The environment is considered solved, when the average (over 100 episodes) of those scores is at least +0.5..

Learning Algorithm

We will use the MADDPG algorithm, which is a modification of the DDPG algorithm (Multi Agent Actor Critic for Mixed Cooperative Competitive environments) that incorporates multiple agents into DDPG. The idea is to include the states and actions of other agents when training an agent's critic network. Note however, that the actor network for an agent is trained only using its own state/action information. After training, the agents rely only on their own states to determine their actions. **This idea turns what is essentially a non-Markov decision process from an agent's perspective (state depends on other agent's actions which are unpredictable and unknown) into a Markov decision process (by including the states/actions of other agents into the state for this agent)**.

Note that multiple design alternatives exist even within MADDPG. Here are some of the few:

- Using a single actor network and a single critic network for all agents. A single critic network is possible if the agents are homogeneous (identical behavior, state and action spaces).
- Using multiple actor networks and a single critic network.
- Using multiple actor networks and multiple critic networks. In this case, each agent has its own actor/critic network.

There are also design choices on how the critic network state may be composed:

- The states and actions of all agents could be concatenated and passed to the critic network as the state of the agent.
- All states and actions of all the ***other*** agents could be concatenated and passed to the critic network as the state of the agent. The action of the agent whose data is being used would be introduced in the first hidden layer of the critic network. This design borrowed from D4PG seems to give good results.

My Final MADDPG Network Architecture

After trying all permutations and combinations (Recorded in the appendix), I settled on the following:

1. A single actor network and a single critic network for all agents. In this Tennis environment, the agents are homogeneous
2. The state of the critic consists of the states of all agents and the actions of all other agents but the agent whose data the critic is training on. The action of the agent whose data is being trained on are concatenated with the output from the 1st hidden layer of the critic network.
3. A replay buffer, target/local networks, and soft updates (ideas from DDPG) to reduce variance and stabilize learning
4. Use of noise to perturb agent actions to promote exploration

Actor Network Architecture

Pytorch is used to implement the below deep neural network in a class called *Actor*:

1. nn.Linear (24 => 256)
2. nn.Linear (256 => 128)
3. nn.Linear (128 => 2)

The first two layers have their weights initialized using a uniform distribution. The last layer weights are initialized as follows based on the reference Udacity implementation of DDPG. ***This step is critical to getting good results***.

self.dense3.weight.data.uniform_(-3e-3, 3e-3)

The Adam optimizer is used for its momentum and automatic learning rate adjustment capabilities. Relu is used as the activation function for the 1st two layers, and tanh is used for the output layer because the output must be in the range [-1, 1].

Critic Network Architecture

Pytorch is used to implement the below deep neural network in a class called *Critic*:

1. nn.Linear (2*24+(2-1)*2=> 256)
    a. Note: For this layer, the states of both agents are concatenated (2*24) with the action of the other agent (2)
2. nn.Linear (256+2 => 128)
    a. Note: For this layer, the previous layer output (256) is concatenated with the actions of the agent whose data is being trained on (2)
3. nn.Linear (128 => 1)

Layer weights as initialized same as the Actor network. Note that the output here consists of a single Q (action) value.

The Adam optimizer is used for its momentum and automatic learning rate adjustment capabilities. Relu is used as the activation function for the 1st two layers, but the output layer is linear and no activation function is applied. A dropout layer (dropout=0.2) is used before the output layer to prevent overfitting and for regularization.

The RL Agent

The RL agent is implemented in a class called *Agent*. It maintains a couple of instances of the *Actor* class and a couple of instances of the *Critic* class. The “local” actor and critic classes are actively used for selecting action values, and the “target” actor and critic classes are used to maintain a copy of weights from the immediately prior batch training call.

```
    # Actor-Network

    self.actorNetwork_local = Actor(state_size, action_size, seed).to(device)

    self.actorNetwork_target = Actor(state_size, action_size, seed).to(device)

    self.actorNetwork_optimizer = optim.Adam(self.actorNetwork_local.parameters(),
lr=LR_ACTOR)

    # Critic-Network unmodified
```

*self.criticNetwork_local = Critic(num_agents*state_size+(num_agents-1)*action_size, action_size, seed).to(device)*

*self.criticNetwork_target = Critic(num_agents*state_size+(num_agents-1)*action_size, action_size, seed).to(device)*

self.criticNetwork_optimizer = optim.Adam(self.criticNetwork_local.parameters(), lr=LR_CRITIC, weight_decay=WEIGHT_DECAY)

ReplayBuffer

The agent also maintains the replay buffer which is as follows:

```
self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
```

Here we use a BUFFER_SIZE of 10^6 to give us enough space to store lots of episodes worth of data and a BATCH_SIZE of 256, so the data from prior 256 steps would be used for training the agent.

This replay buffer is populated with data from both agents at every time step as follows:

```
for i, (a_states, a_actions, a_rewards, a_next_states, a_dones) in enumerate(zip(states, actions, rewards, next_states, dones)):
    all_states = a_states
    all_next_states = a_next_states
    for j in range(self.num_agents):
        if j == i:
            continue
        all_states = np.concatenate((all_states, states[j]), axis=None)
        all_next_states = np.concatenate((all_next_states, next_states[j]), axis=None)
```

#Note BELOW how actions of all other agents are concatenated together AFTER all the states (Instead of state1/action1/state2/action2…) have been concatenated ABOVE. This is because next states are all concatenated here, and next_actions of other agents will only be available in the learning function. HAVE TO KEEP IT CONSISTENT!!

```
    for j in range(self.num_agents):
        if j == i:
            continue
        all_states = np.concatenate((all_states, actions[j]), axis=None)
```

```
            self.replay_buffer.add(all_states, a_states, a_actions, a_rewards, all_next_states,
        a_next_states, a_dones)
```

When BATCH_SIZE worth of data is gathered, training begins. Training is then done every UPDATE_EVERY steps (set at 1)

```
        replay_buffer_len = len(self.replay_buffer)

        if episode > BEGIN_LEARNING_AT_EPISODE and replay_buffer_len > BATCH_SIZE and
    timestep % UPDATE_EVERY == 0:

            # If enough samples are available in memory, get random subset and learn

            for _ in range(UPDATE_TIMES):

                experiences = self.replay_buffer.sample()

                self.learn(experiences, GAMMA)
```

The ReplayBuffer class provides a sample function that returns a batch of experience data sampled according to a uniform random distribution as follows:

```
        experiences = random.sample(self.memory, k=self.batch_size)


        all_states = torch.from_numpy(np.vstack([e.all_state for e in experiences if e is not
    None])).float().to(device)
        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not
    None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not
    None])).float().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not
    None])).float().to(device)
        all_next_states = torch.from_numpy(np.vstack([e.all_next_state for e in experiences if e is
    not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not
    None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not
    None]).astype(np.uint8)).float().to(device)

    return (all_states, states, actions, rewards, all_next_states, next_states, dones)
```

The learn function

This is the guts of the Agent and how it learns! It basically uses the Actor Critic RL algorithm to determine a TD loss that is then used to train the *Critic*.

First, I do some preprocessing to build the next_states by concatenating next_states with the next_actions predicted by the target actor network.

```
# 2D tensors batch_size x ...

all_states, states, actions, rewards, all_next_states, next_states, dones = experiences

#print("actions: ", actions.data.size())


# start at 1 because we only want to concatenate actions corresponding to other agents

# and all_next_states[:,0 to self.state_size-1] is this agent's next_state (same as next_states)

for i in range(1, self.num_agents):

   next_states_oa = all_next_states[:, i*self.state_size: (i+1)*self.state_size]

   next_actions_oa = self.actorNetwork_target(next_states_oa)

   all_next_states = torch.cat((all_next_states, next_actions_oa), dim=1)
```

The guts of this algorithm calculates a delayed (“fixed”) target using a prior snapshot of the *Critic*.

```
# Update the critic network

Qs = self.criticNetwork_local(all_states, actions)


next_actions = self.actorNetwork_target(next_states)

Qns = self.criticNetwork_target(all_next_states, next_actions)


TD_Target = rewards + gamma*Qns*(1-dones)   # multiply by (1-dones): Qmax_ns = 0 for
terminal state


critic_loss = F.mse_loss(Qs, TD_Target)


self.criticNetwork_optimizer.zero_grad()

critic_loss.backward()

#torch.nn.utils.clip_grad_norm_(self.criticNetwork_local.parameters(), 1)

self.criticNetwork_optimizer.step()
```

After updating the critic weights, the algorithm then updates the *Actor* network. First, the *Actor* network is used to predict the actions using the current state. The *Critic* is then used to predict the Q-value of the actions predicted by the actor. This Q-Value is directly used as the loss and this is what the actor tries to maximize.

```
# Update the actor network

pred_actions = self.actorNetwork_local(states)

actor_loss = -self.criticNetwork_local(all_states, pred_actions).mean()   #negative sign because we want to maximize


self.actorNetwork_optimizer.zero_grad()

actor_loss.backward()

#torch.nn.utils.clip_grad_norm_(self.actorNetwork_local.parameters(), 1)

self.actorNetwork_optimizer.step()
```

As the last step, it calls the *soft_update* function to update the prior snapshots of the *Actor/Critic* using the latest iteration of the networks

```
self.soft_update(self.actorNetwork_local, self.actorNetwork_target, TAU)

self.soft_update(self.criticNetwork_local, self.criticNetwork_target, TAU)
```

A value of 0.001 is used for TAU. This value assigns little importance to the latest iteration of the QNetwork and greater importance to the prior snapshot of the QNetwork, thus ensuring that the weights update very gradually and the learning improvement is stable.

The act function

The act function uses the Actor (local) network to predict the actions. Given that setting up an exponentially decaying exploration vs. exploitation parameter is not straightforward. SO…, we inject noise to the action prediction to simulate exploration. Note also, that for the first *BEGIN_LEARNING_AT_EPISODE* episodes (Set at BATCH_SIZE*2, so 512 in this case), the agent takes completely random actions, again to promote exploration in the beginning.

```
if episode <= BEGIN_LEARNING_AT_EPISODE:

    actions = np.random.randn(self.num_agents, self.action_size) # select an action (for each agent)

else:

    states = torch.from_numpy(states).float().to(device)
    with torch.no_grad():
      self.actorNetwork_local.eval()    # switch to eval mode
      actions = self.actorNetwork_local(states).cpu().data.numpy()
    self.actorNetwork_local.train()    # switch back to training mode

    actions += self.noise.sample()

  actions = np.clip(actions, -1, 1)
  return actions
```

The noise function is as follows (Borrowed from the reference Udacity implementation of DDPG):

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    #def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
    def __init__(self, size_tuple, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""
        #self.mu = mu * np.ones(size)
        self.mu = mu * np.ones(size_tuple)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.size_tuple = size_tuple
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)
    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        #dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in range(len(x))])
        #dx = self.theta * (self.mu - x) + self.sigma * np.random.random(self.size_tuple)
        dx = self.theta * (self.mu - x) + self.sigma * np.random.standard_normal(self.size_tuple)
        self.state = x + dx
        return self.state
```

The training function

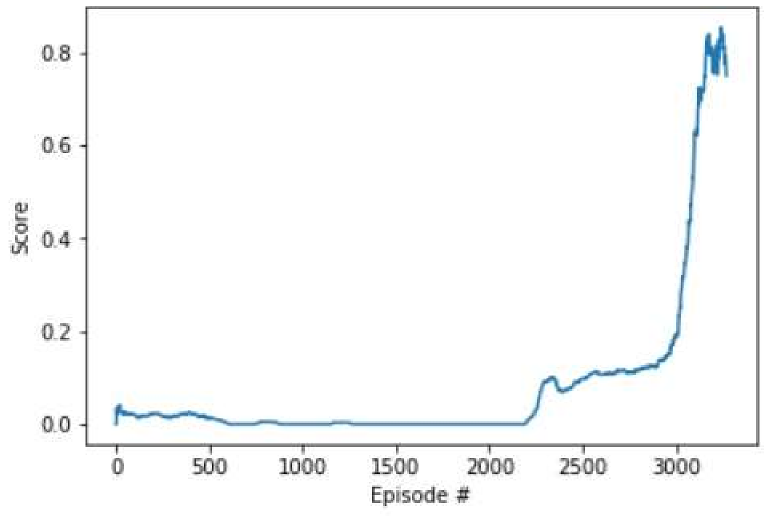
Given the episodic nature of the game, we will not be training for a fixed number of time steps. Instead, the training continues until the end of the episode.

```
    for i_episode in range(1, n_episodes+1):
…
        while True:
…
            if np.any(dones):
…
                break                    # uncomment if you reset training at end of each episode
```

Plot of Rewards (Performance)

With the final selection of MADDPG design and hyperparameters, the agent performs well, learning to score 0.69 per episode, averaged over 100 consecutive time steps within 3271 episodes. Below is a snapshot of learning progress from one such run.

```
Episode 3100    Average Score: 0.61
Environment solved in 3100 episodes!    Average Score: 0.61
Episode 3271    Average Score: 0.75
```



Sample result from playing the game with this trained agent:

Step: 784\Max Reward: 2.09

Below is the complete agent training code:

```
def train_d4pgAgent(env, brain_name, agent, n_episodes=100, max_t=1000):
    """D4PG for continuous control.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action selection
        eps_end (float): minimum value of epsilon
        eps_decrement (float): subtractive factor (per episode) for decreasing epsilon
    """
    max_scores_window = deque(maxlen=100)  # last 100 average scores over all agents
    envSolved = False
    envSolvedInEpisodes = 0

    maxAvgScore = -99
    avg_max_scores = []
    for i_episode in range(1, n_episodes+1):
        agent.reset_noise()
        scores = np.zeros(num_agents)                       # initialize the score (for each agent)
        env_info = env.reset(train_mode=True)[brain_name]      # reset the environment

        t = -1
        states = env_info.vector_observations                # get the current state (for each agent)
```

```
while True:
   t += 1
#for t in range(max_t):
   actions = agent.act(states, i_episode)    # num_agents x action_size

   env_info = env.step(actions)[brain_name]       # send the action to the environment
   next_states = env_info.vector_observations     # get the next states
   rewards = env_info.rewards                     # get the rewards
   dones = env_info.local_done                    # see if episodes have finished

   agent.step(states, actions, rewards, next_states, dones, i_episode, t)

   states = next_states
   scores += rewards
   if np.any(dones):
      maxScore = np.max(scores)                   # save max score at end of episode
      #scores = np.zeros(num_agents)              # uncomment if you are continuing training
      break                                       # uncomment if you reset training at end of each episode

max_scores_window.append(maxScore)       # save most recent max score over both agents

avgMaxScoreOver100Episodes = np.mean(max_scores_window)
avg_max_scores.append(avgMaxScoreOver100Episodes)
if avgMaxScoreOver100Episodes > maxAvgScore:
```

```
            maxAvgScore = avgMaxScoreOver100Episodes


        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, avgMaxScoreOver100Episodes),
end="")


        #if i_episode % 100 == 0:


        if envSolved and maxAvgScore - avgMaxScoreOver100Episodes >= 0.1:   #stop if envSolved
and score reducing

            torch.save(agent.actorNetwork_local.state_dict(), 'actorNetwork.pth')

            torch.save(agent.criticNetwork_local.state_dict(), 'criticNetwork.pth')

            break


        if not envSolved and avgMaxScoreOver100Episodes > 0.6:

            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode,
avgMaxScoreOver100Episodes))

            envSolved = True

            envSolvedInEpisodes = i_episode

            #break


    return avg_max_scores


from session_utils import active_session

import matplotlib.pyplot as plt

%matplotlib inline
```

agent = Agent(num_agents=num_agents, state_size=state_size, action_size=action_size, seed=2)

with active_session():

scores = train_d4pgAgent(env, brain_name, agent, 7000, 1000)

Ideas for Future Work (Recommendations)

The next logical challenges in the project are to implement:

1. Implement true n-step learning.
2. Tackle the Soccer optional project
3. Implement networks to predict the states and actions of other agents, so critic for each agent would not have to rely on knowing the states/actions of other agents
    a. Implement an ensemble of networks instead of a single network to predict the states and actions of other agents. This would make the predictions stabler with less variance.

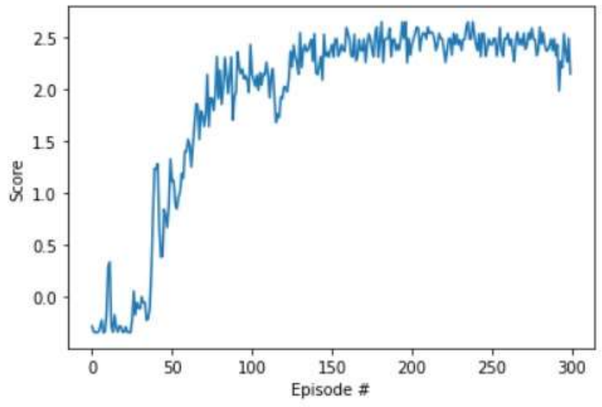
Design iterations

I tested a number of designs to arrive at the final MADDPG design presented above. Below are performance data from some of those design iterations that did not make the final cut:

Design 1:

Tennis with D4PG from the Reacher project unmodified. This is with 1000 time-steps per episode. Episode dones are ignored. **Note how this performance is actually superior to MADDPG!** This is probably because of the simple homogeneous nature of the Tennis game in this case BUT we are actually more interested in implementing MADDPG so we will continue.

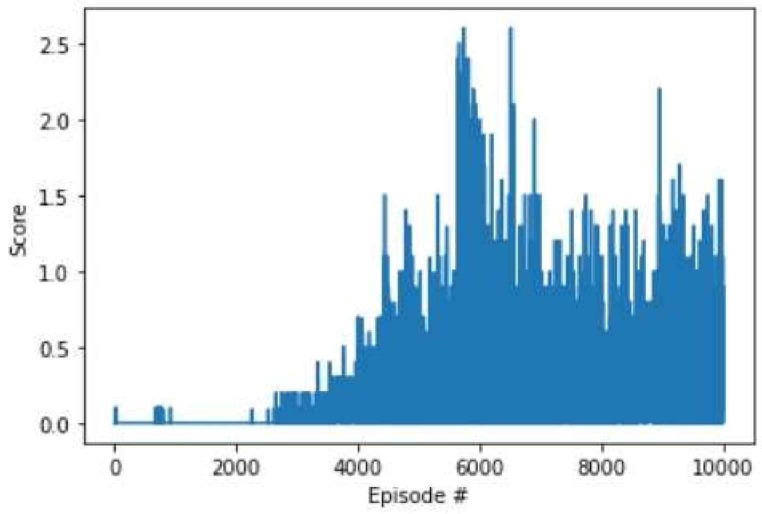
```
Episode 100     Average Score: 0.91
Episode 200     Average Score: 2.31
Episode 300     Average Score: 2.45
```



Design 2:

Tennis with D4PG from Reacher unmodified but now episode ends when any agent is "done". It's no longer for fixed time-step of 1000.

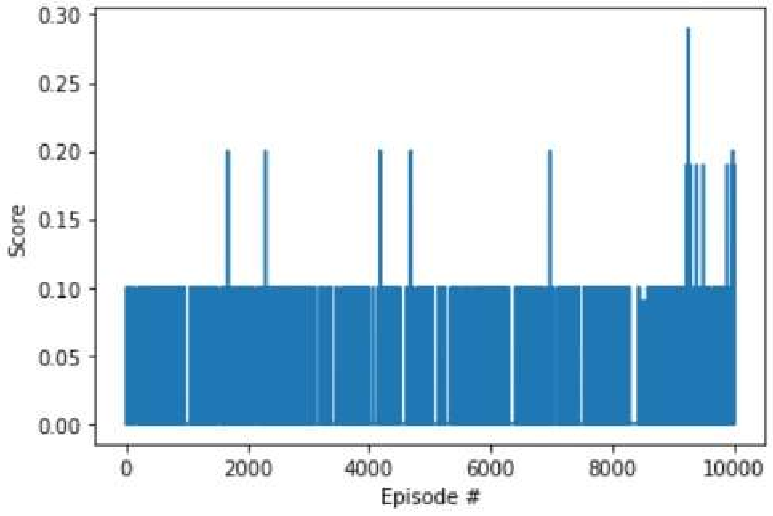
```
Episode 5703    Average Score: 0.50
Environment solved in 5703 episodes!    Average Score: 0.50
Episode 10000   Average Score: 0.37
```



Design 3:

Multiple actor networks/one critic network. Note the poor performance in this case, very likely from some bug in the code. Also, each agent network is only receiving it's own training data at each step, so there's less data to train.

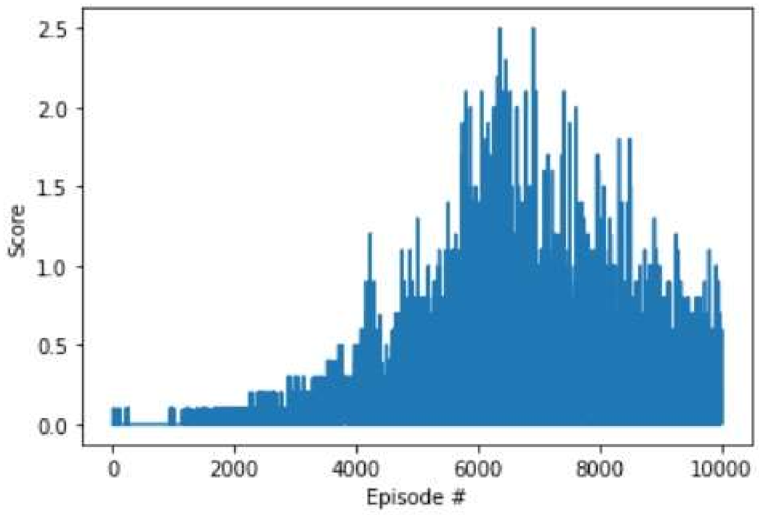
```
Episode 10000   Average Score: 0.04
```



Design 4:

Multiple actor networks/multiple critic networks – Critic state is own state + other agents actions

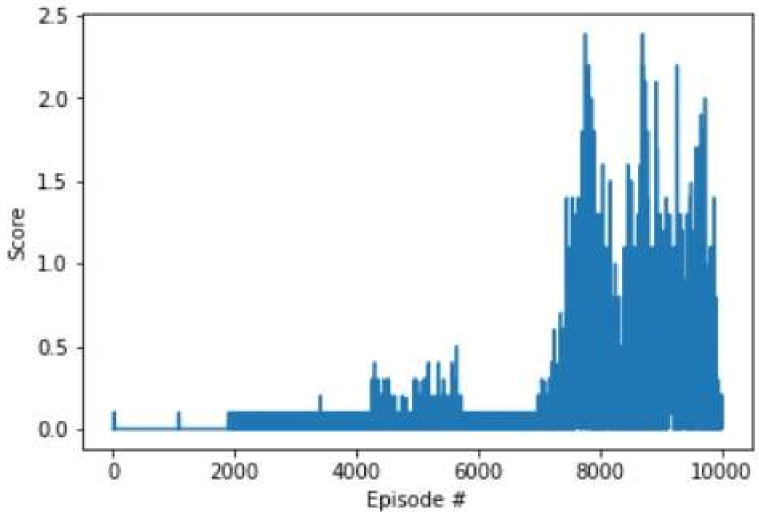
```
Episode 10000	Average Score: 0.20
```



Design 5:

Multiple actor and critic networks. Critic state is composed of all states and all actions, including the agent whose data is being trained on

```
Episode 10000	Average Score: 0.10
```



Design 6:

MADDPG – One actor/one critic. This exhibited relatively stable performance and clued me to the final design

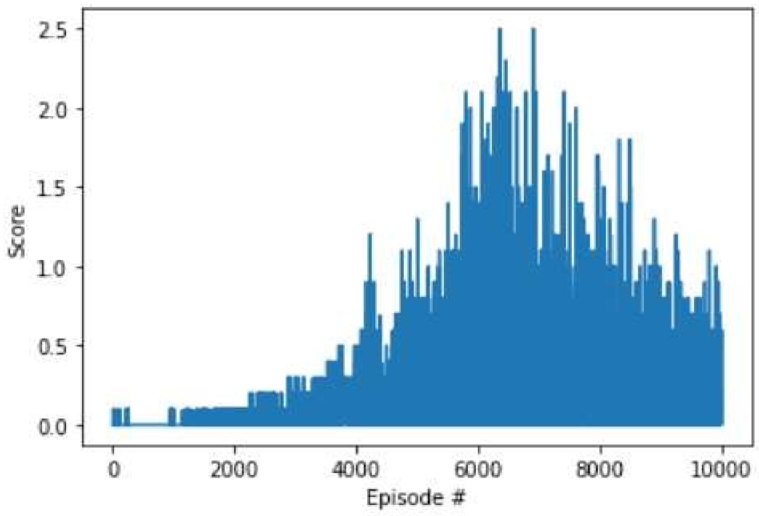
```
Episode 7225	Average Score: 0.50
```

```
Environment solved in 7225 episodes!    Average Score: 0.50
Episode 11197    Average Score: 0.66
```

Design 7:

Multiple actor and critic networks. Critic state composed of all states and all actions of all agents. TAU increased to 1e-2 from 1e-3 to give more importance to local network instead of target network. Also bigger networks (512 x 256) and greater noise (theta 0.17 and sigma 0.24)

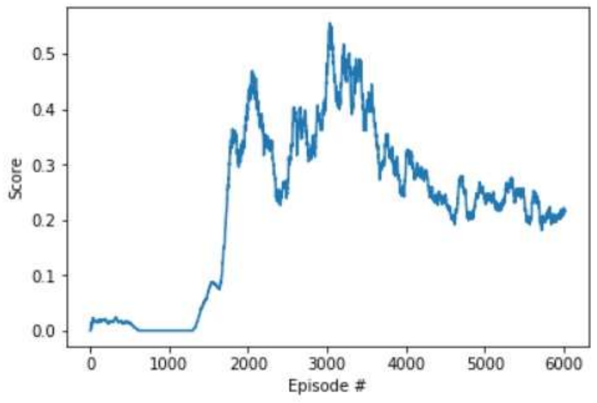
```
Episode 10000    Average Score: 0.20
```



Design 8:

One network for actor and one for critic. Network size (512,256). TAU of 1e-2. Noise theta 0.17, sigma 0.24. Random actions until episode 512. **Confirms advantage of single network for actor/critic but design 6 was better!**

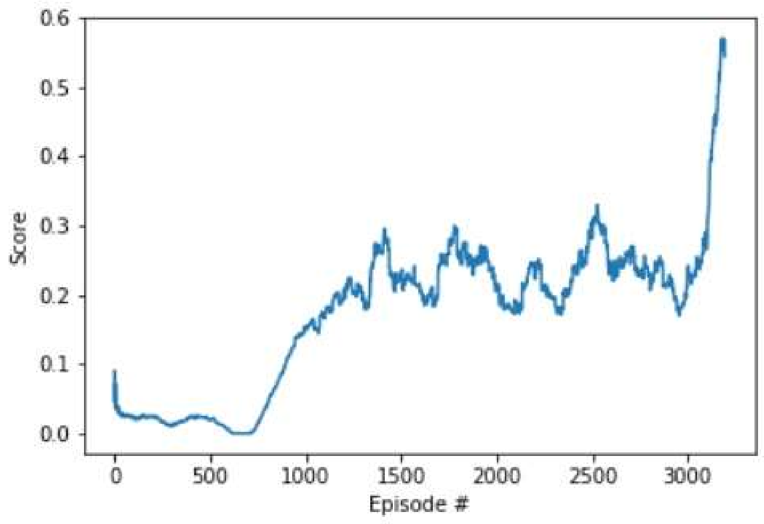
```
Episode 3016    Average Score: 0.51
Environment solved in 3016 episodes!    Average Score: 0.51
Episode 6017    Average Score: 0.22
```



Design 9:

One network for actor/critic. Network size (256, 128). Added dropout (0.15). **But… why does performance drop off after reaching peak??**

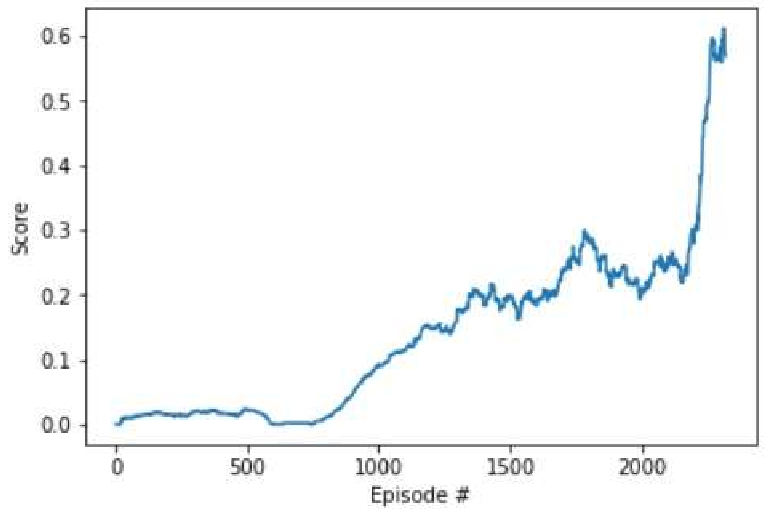
```
Episode 3163    Average Score: 0.51
Environment solved in 3163 episodes!    Average Score: 0.51
Episode 3194    Average Score: 0.54
```



Design 10:

Fixed a critical bug!!! For the critic, current state was being incorrectly composed as STATE1+ACTION1+STATE2+ACTION2+… while next state was composed as STATE1+ STATE2+…+ACTION1+ACTION2+…!! After fixing…

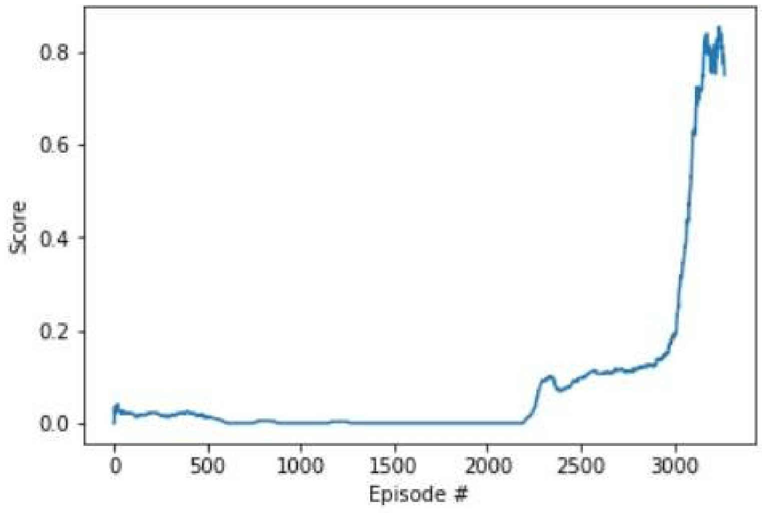
```
Episode 2256    Average Score: 0.50
Environment solved in 2256 episodes!    Average Score: 0.50
Episode 2321    Average Score: 0.57
```



Design 11:

Same as design 10, but reduced TAU back to 1e-3 to increase stability of learning and prevent the score from backsliding after hitting ~0.6. **This is our final design!!**

```
Episode 3100    Average Score: 0.61
Environment solved in 3100 episodes!    Average Score: 0.61
Episode 3271    Average Score: 0.75
```


Score
Episode #