Adrian Chow

About

Multi-Agent Control using Deep Reinforcement Learning

Adrian Chow 2 days ago · 4 min read

One difficult task in the Deep Reinforcement Learning space is having multiple agents interact and learn. The reality is that the world is a multi-agent environment in which intelligence is developed by interacting with multiple agents. A major breakthrough in this regard was when researchers at DeepMind developed an AI engine, alphago zero, which learned to play Go by training in a multi-agent environment. The agent developed such a high level of competency it was able to beat LeeSedol, a professional Go player. Extensions of this finding led to OpenAI Five which was able to conquer a complex game such as Dota 2. Clearly, DRL in a multi-agent environment has expanded the capabilities of Markov learning.

Source: OpenAI

Bill Gates @BillGates

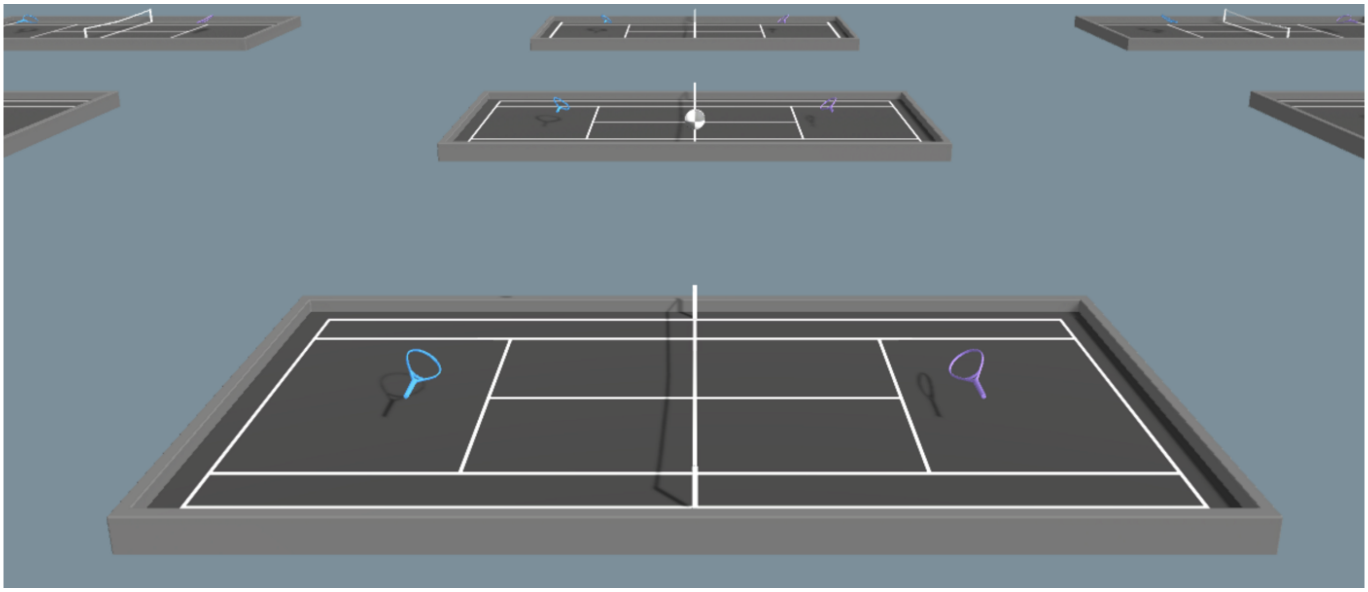
#AI bots just beat humans at the video game Dota 2. That’s a big deal, because their victory required teamwork and collaboration – a huge milestone in advancing artificial intelligence.

OpenAI Five
Our team of five neural networks, OpenAI Five, has started to defeat amateur human teams at Dota 2. While today w…
openai.com

7:25 PM · Jun 26, 2018

I have taken the challenge to myself to implement a multi-environment agent using my pre-existing knowledge of RL to solve the Tennis environment on the Unity ML-Agents. The goal of this project is to teach two agents to play a low-level version of tennis. By applying deep reinforcement learning to this environment, two separate agents compete in order to define their individual policies. This method allows competition to essential become collaboration as agents compete to find the optimal policy. The following project has been an implementation based on OpenAi’s **Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments** research paper.

The Environment



Source: Unity-Technologies

RL Problem Specifications

- Goal of Agent: keep the ball in play
- Rewards : `+0.1` every time agent hits the ball over the net, `-0.1` every time agent lets a ball hit the ground or hits the ball out of bounds
- Action Space — Continuous, 4-Dimension Vector Space from [-1, +1]
- State Space — Continuous, 8-Dimension, 3 Stacked Observations
- Solving Condition: Average score of +0.5 over 100 consecutive episodes.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to the movement toward (or away from) the net, and jumping.

Learning Algorithm

The algorithm is a multi-agent variation of the standard DDPG algorithm that I have implemented in the past. If you are unfamiliar with the Deep Deterministic Policy Gradent algorithm, you can check out my other medium post. Essentially this article is an extension of the previous DDPG I have implemented. Here is the basis of the algorithm:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

DDPG Algorithm (Source: Here)

Some constraints/assumptions to make when implementing our multi-agent policy include:

- The learned policies can only use local information (i.e. their own observations) at execution time
- Do not assume a differentiable model of the environment dynamics
- Do not assume any particular structure on the communication method between agents

Multi-Agent DDPG Initialization:

```
1  class MADDPG:
2      """ Class for training multiple agents in the multi-agent environment"""
3  
4      def __init__(self, state_size, action_size, num_agents, seed):
5  
6          super(MADDPG, self).__init__()
7          self.state_size = state_size
8          self.action_size = action_size
9          self.num_agents = num_agents
10         self.seed = random.seed(seed)
11 
12         # Initialize agents in the multi-agent environment
13         self.DDPGs = [DDPG(state_size, action_size, num_agents, seed) for i in range(num
14 
15         # Replay Buffer (shared by all agents)
16         self.memory = ReplayBuffer(action_size, num_agents, BUFFER_SIZE, BATCH_SIZE, see
```


- Notice the number of agents is passed in as a parameter for the MADDPG. In the case of the Tennis Environment, there are two agents.
- Individual agents with their own set of actors and critics are created based on the number of agents, see line 13.
- The replay buffer will be the same as before, except instead this time an experience will contain the states and actions for both agents in a given timestep.
- Note the DDPG follows the same class implemented in the previous project. See below:

```
class DDPG:
    """ Base Class for an Agent in the multi-agent environment"""

    def __init__(self,
                 state_size,
                 action_size,
                 num_agents,
                 seed,
                 hidden_in_actor=200,
                 hidden_out_actor=150,
                 hidden_in_critic=200,
```

```
                 hidden_out_critic=150,
                 lr_actor=LR_ACTOR,
                 lr_critic=LR_CRITIC):

        super(DDPG, self).__init__()

        self.state_size = state_size
        self.action_size = action_size
        self.num_agents = num_agents
        self.seed = random.seed(seed)

        self.actor = Actor(state_size,
                           action_size,
                           seed,
                           hidden_in_actor,
                           hidden_out_actor).to(device)

        self.target_actor = Actor(state_size,
                                  action_size,
                                  seed,
                                  hidden_in_actor,
                                  hidden_out_actor).to(device)

        self.critic = Critic(state_size,
                             action_size,
                             num_agents,
                             seed,
                             hidden_in_critic,
                             hidden_out_critic).to(device)

        self.target_critic = Critic(state_size,
                                    action_size,
                                    num_agents,
                                    seed,
                                    hidden_in_critic,
                                    hidden_out_critic).to(device)

        # Ornstein Uhlenbeck Noise for Action Space Exploration
        self.noise = OrnsteinUhlenbeckNoise(action_size, seed)

        # Initialize targets same as original networks
        self.copy_weights(self.critic, self.target_critic)
        self.copy_weights(self.actor, self.target_actor)

        # Actor and Critic Adam Optimizers
```

```
        self.actor_optimizer = optim.Adam(self.actor.parameters(), lr=lr_actor)
        self.critic_optimizer = optim.Adam(self.critic.parameters(), lr=lr_critic)


    def act(self, state, add_noise=True):
        """Returns actions for given state as per current policy."""

        state = torch.from_numpy(state).float().to(device)
        self.actor.eval()
        with torch.no_grad():
            action = self.actor(state).cpu().data.numpy()
        self.actor.train()
        if add_noise:
            action += self.noise.noise()
        return np.clip(action, -1, 1)


    def step(self, experiences, gamma):
        self.learn(experiences, gamma)


    def learn(self, experiences, gamma):
        """Update policy and value parameters using given batch of experience tuples.
        Q_targets = r + γ * target_critic(next_state, target_actor(next_state))
        where:
            target_actor(state) -> action
            target_critic(state, action) -> Q-value
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', a2, s2' done) tup
            gamma (float): discount factor
        """

        states, actions, rewards, next_states, dones = experiences

        next_states_tensor = torch.cat(next_states, dim=1).to(device)
        states_tensor = torch.cat(states, dim=1).to(device)
        actions_tensor = torch.cat(actions, dim=1).to(device)

        # ---------------------------- update critic ---------------------------- #
        # Get predicted next-state actions and Q values from target models

        next_actions = [self.actor(state) for state in states]
        next_actions_tensor = torch.cat(next_actions, dim=1).to(device)
```

```
        Q_targets_next = self.target_critic(next_states_tensor, next_actions_tensor)
        # Compute Q targets for current states (y_i)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        # Compute critic loss
        Q_expected = self.critic(states_tensor, actions_tensor)
        critic_loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # ---------------------------- update actor ---------------------------- #
        # Compute actor loss
        actions_pred = [self.actor(state) for state in states]
        actions_pred_tensor = torch.cat(actions_pred, dim=1).to(device)
        actor_loss = -self.critic(states_tensor, actions_pred_tensor).mean()
        # Minimize the loss, thererby maximizing the reward
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        # ----------------------- update target networks ----------------------- #
        self.soft_update(self.critic, self.target_critic, TAU)
        self.soft_update(self.actor, self.target_actor, TAU)


    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        θ_target = τ*θ_local + (1 - τ)*θ_target
        Params
        ======
            local_model: PyTorch model (weights will be copied from)
            target_model: PyTorch model (weights will be copied to)
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(), local_model.par
            target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)


    def reset(self):
        self.noise.reset()


    def copy_weights(self, source, target):
        """Copies the weights from the source to the target"""
```

```
146         for target_param, source_param in zip(target.parameters(), source.parameters())
147             target_param.data.copy_(source_param.data)
```

ddpg.py hosted with ❤ by GitHub view raw

DDPG, Source

- DDPG is still using then Ornstein Uhlenbeck Noise for action space exploration.
- The replay buffer is shared between agents, that is why it is stored in the MADDPG class.
- Since there are multiple states, actions per experience they are concatenated into single tensors, see lines 100 and 115.

Multi-Agent DDPG Methods:

```
class MADDPG:
    """ Class for training multiple agents in the multi-agent environment"""

    def __init__(self, state_size, action_size, num_agents, seed):

        super(MADDPG, self).__init__()
        self.state_size = state_size
        self.action_size = action_size
        self.num_agents = num_agents
        self.seed = random.seed(seed)

        # Initialize agents in the multi-agent environment
        self.DDPGs = [DDPG(state_size, action_size, num_agents, seed) for i in range(num

        # Replay Buffer (shared by all agents)
        self.memory = ReplayBuffer(action_size, num_agents, BUFFER_SIZE, BATCH_SIZE, see


    def act(self, states, add_noise=True):
        """Returns actions for each agent."""

        return [agent.act(state, add_noise) for agent, state in zip(self.DDPGs, states)]


    def step(self, states, actions, rewards, next_states, dones):

```

```
        self.memory.add(states, actions, rewards, next_states, dones)

        for agent in self.DDPGs:
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                agent.step(experiences, GAMMA)
```

maddpg_methods.py hosted with ❤ by **GitHub** **view raw**

MADDPG, Source

- Per the time step for MADDPG, each DDPG is updated normally (randomly sampling from replay buffer) to allow each agent to learn independently from each other.

Multi-Agent DDPG Training:

- Before we get into the training aspect here are some of the hyperparameters that have worked for me in this project. Notice that I have increased the batch_size, due to the increase in learning capacity required.

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 250        # minibatch size
GAMMA = 0.99           # discount factor
TAU = 1e-3              # for soft update of target parameters
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 1e-3        # learning rate of the critic
```

hyperparameters.py hosted with ❤ by **GitHub** **view raw**

Hyperparameters

- The critic and actor networks have stayed the same for the most part. One minor change is the fact that the critic now accounts for the states of both agents when calculating the return. This minor change allows for the critic to adapt to a broader state space and converge better.

```
def hidden_init(layer):
    fan_in = layer.weight.data.size()[0]
    lim = 1. / np.sqrt(fan_in)
    return (-lim, lim)

class Actor(nn.Module):
```

```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=200, fc2_units=150):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return F.torch.tanh(self.fc3(x))


class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, num_agents, seed, fcs1_units=200, fc2_un
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
```

```
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.num_agents = num_agents
        self.fcs1 = nn.Linear((state_size+action_size) * num_agents, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values.""
        xs = torch.cat((state, action), dim=1)
        x = F.relu(self.fcs1(xs))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

networks_maddpg.py hosted with ❤ by GitHub view raw

- The training loop is the same as usual

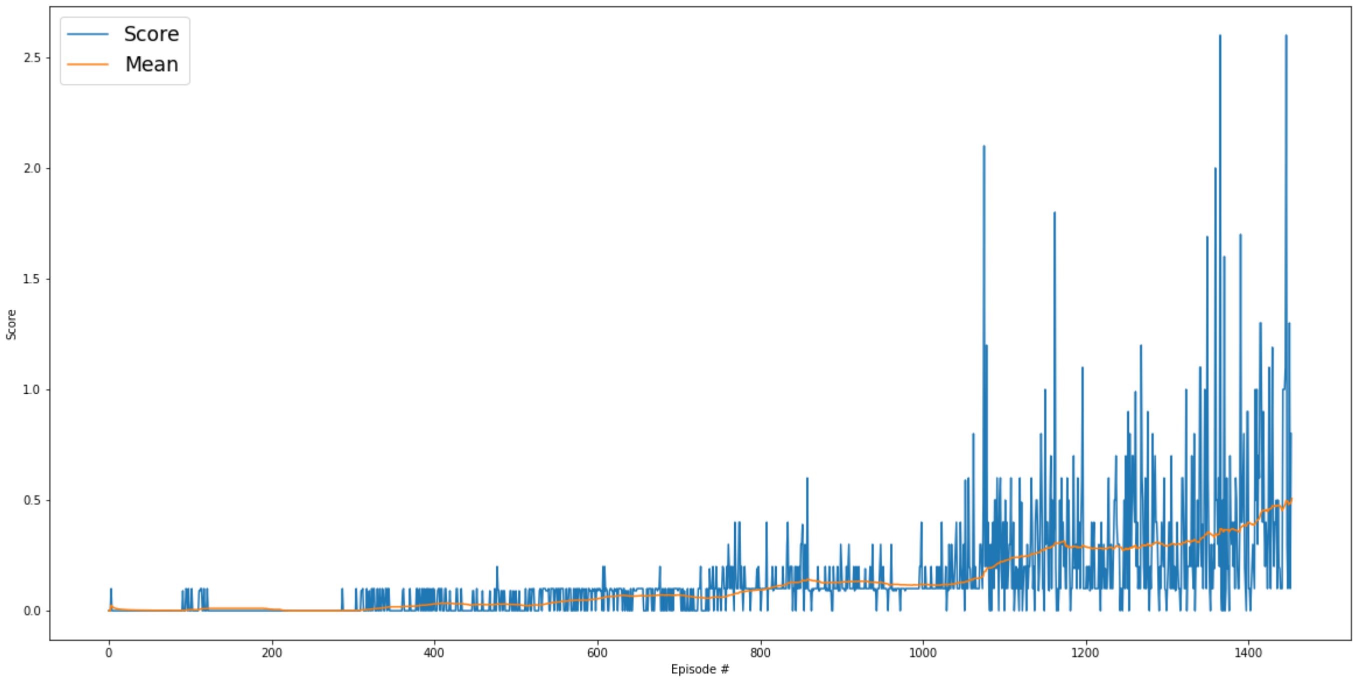
```

def maddpg_train(agent,
                 env,
                 brain_name,
                 n_agents,
                 n_episodes=10000,
                 max_t=10000,
                 print_every=100,
                 win_condition=0.5):

    """

    Tennis using MADDPG.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        print_every (int): how many episodes before printing scores
```

```
        n_agents (int): how many agents are in the environment

    """

    scores_deque = deque(maxlen=100)
    scores = []
    average_scores_list = []

    for i_episode in tqdm(range(1, n_episodes+1)):

        env_info = env.reset(train_mode=True)[brain_name] # reset the environment
        states = env_info.vector_observations             # get the current state (for e
        score = np.zeros(n_agents)                       # initialize the score (for each

        for t in range(max_t):
            actions = agent.act(states)                   # consult agent for actions
            env_info = env.step(actions)[brain_name]      # take a step in the env
            next_states = env_info.vector_observations    # get next state (for each age
            rewards = env_info.rewards                    # get reward (for each agent)
            dones = env_info.local_done                   # see if episode finished
            agent.step(states, actions, rewards, next_states, dones)  # take a learning
            score += rewards                              # update the score (for each a
            states = next_states                          # roll over states to next tim
            if np.any(dones):                             # exit loop if episode finishe
                break

        score_max = np.max(score)
        scores.append(score_max)
        scores_deque.append(score_max)
        average_score = np.mean(scores_deque)
        average_scores_list.append(average_score)

        print('\rEpisode {}\tAverage Score: {:.3f}'.format(i_episode, np.mean(scores_deq

        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage score: {:.3f}'.format(i_episode , average_score

        # Winning condition + save model parameters
        if average_score >= win_condition:
            print("\rSolved in episode: {} \tAverage score: {:.3f}".format(i_episode , a
            break

    execution_info = {'last_score': scores.pop(),
                      'solved_in': i_episode,
                      'last_100_avg': average_score}
```

```
    return scores, scores_deque, execution_info
```

Training_loop.py hosted with ❤ by GitHub view raw

- The results from training were approximately on-point with the baseline training requirements presented for this environment. I was able to solve the environment in 1454 episodes.



Training Score-Episodes Plot. Source

Possible Improvements

- Modify the critic and actor to account for the agent's belief of the opponent's next move. If the agent can understand the opponent's policy it may be able to perform better?
- Update the agents selectively or stochastically.
- Decrease action exploration as the training episodes increase.