DRL Collaboration and Competition Project Report

Madison Estabrook

MS; Facebook PyTorch Phase 3 Nanodegree Scholar

This document includes a description of the learning algorithm, a plot of rewards, and ideas for future work. The description of the learning algorithm not only provides the code, but also provides an explanation. The plot of rewards is included, along with the number of episodes needed to solve the (Tennis) environment. The ideas for future work contains concrete ideas for future work.

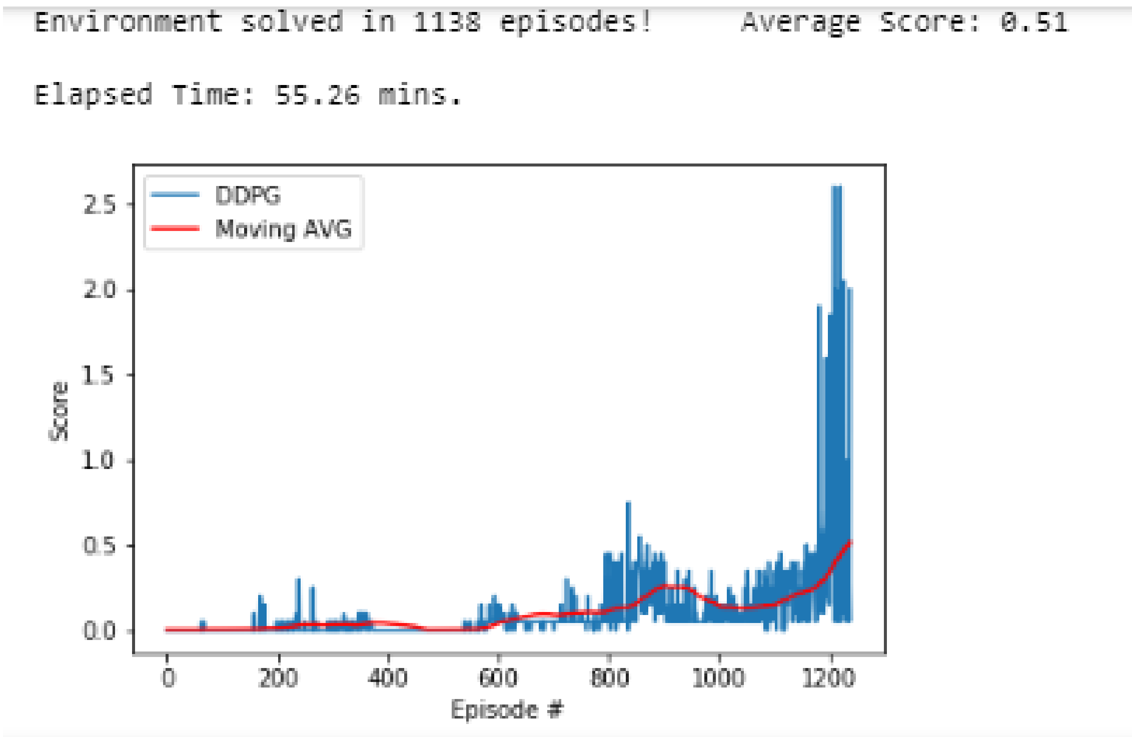

Figure 1: A plot of episodes versus scores

Learning Algorithm

DDPGN The learning algorithm chosen is Deep Deterministic Policy Gradients (*DDPG*). This algorithm was chosen for its success in other projects. This algorithm is as follows:

```
def ddpg(n_episodes=1500, max_t=1000, print_every=10):
    scores_deque = deque(maxlen=100)
    mean_list = []
    moving_avg_list = []
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]
```

```
7           states = env_info.vector_observations
8           scores = np.zeros(num_agents)
9           agent.reset()
10          start_time = time.time()
11          for t in range(max_t):
12              actions = agent.act(states,  add_noise=True)
13              env_info = env.step(actions)[brain_name]
14              next_states = env_info.vector_observations
15              rewards = env_info.rewards
16              dones = env_info.local_done
17              for state, action, reward, next_state, done in zip(states, actions,
    ↪   rewards, next_states, dones):
18                  agent.step(state, action, reward, next_state, done, t)
19              states = next_states
20              scores += rewards
21              if np.any(dones):
22                  break
23          duration = time.time() - start_time
24          scores_deque.append(np.amax(scores))
25          mean_list.append(np.mean(scores))
26          moving_avg_list.append(np.mean(scores_deque))
27          print('\rEpisode {} ({}s)\tAverage Score:
    ↪   {:.2f}'.format(i_episode,round(duration),mean_list[-1]), end="")
28          if i_episode % print_every == 0:
29              torch.save(agent.actor_local.state_dict(), 'checkpoint_actor.pth')
30              torch.save(agent.critic_local.state_dict(), 'checkpoint_critic.pth')
31              print('\rEpisode {} ({}s)\tMean: {:.2f}\tMoving Avg: {:.2f}'.format(
32                    i_episode, round(duration), mean_list[-1],
    ↪   moving_avg_list[-1]))
33
34          if moving_avg_list[-1] >= 0.5 and i_episode >= 100:
35              print('\nEnvironment solved in {:d} episodes!\t Average Score:
    ↪   {:.2f}'.format(i_episode-100, moving_avg_list[-1]))
36              break
37
38      return mean_list, moving_avg_list
```

On line 1, there are 2 hyperparameters and 1 option:

1. `n_episodes` - this determines how many attempts the agent has at solving the environment. This is initialized to be equal to 1500 because the code is self-stopping and no additional attempts were needed.
2. `max_t` - this determines how many timesteps the agent has when attempting to solve the environment. This value was not overwritten because success was achieved with the initial value, which was based on previous research.
3. `print_every` - this determines how often the score is printed and saved on the user's screen. This value to chosen to balance avoid overwhelming the user and underwhelming the user.

Line 2 sets the variable `scores_deque` to be equal to an instance of the deque data type with a maximum length of 100. Lines 3-4 set the variables `mean_list` and `moving_avg_list` to be equal to an empty list. For each individual episode (`i_episode`) in the range of 1 and the hyperparameter `n_episodes` + 1:

1. Resets the environment

2. Retrieves the state
3. Sets `score` equal to a numpy array of length `num_agents`
4. Resets the agent
5. For each timestep in `m_tax`:
    (a) Sets the variable `actions` to be equal what the agent did, which depends on the state. Noise was added to the state.
    (b) Retrieves the next state
    (c) Retrieves the agent's reward
    (d) Determines if the agent is done
    (e) Causes the agent to step (learn from its previous actions)
    (f) Retrieves the next state
    (g) Adds the next reward to `scores`
    (h) If there are any elements in the numpy array `dones`:
        i. Breaks (stops the loop)
6. Adds the mean of `scores` to the list of means
7. Adds the mean of `scores_deque`, which is defined to be equal to the max of `scores`, to the moving average list
8. Prints training statistics
9. Saves the models
10. If the remainder (modulo) of the current episode number and `print_every` is 0:
    (a) Saves the printed message
11. If the latest average of the list of scores is greater than or equal to +0.5 and the episode number is at least 100:
    (a) Prints a success message
    (b) Breaks (stops) the loop
12. Return `mean_list` and `moving_avg_list`

The Models There are 2 models - the Actor and the Critic.

Actor

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

    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):
        """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.bn1 = nn.BatchNorm1d(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.bn1(self.fc1(state)))
        x = F.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))
```

This model is a relatively simple neural network. This model has 3 methods - `__init__`, `reset_parameters`, and `forward`. The `__init__` method defines all the proprieties that this class has:

1. `seed` - this is a random value that will be used later. This is another hyperparetmer.
2. `fc1` - this is the first of 3 fully-connected (*fc*) layers of the neural network. This layer is a linear layer from `state_size` and `fc1_units`
3. `bn1` - this is the batch-normalising layers. This layer can help the network perform faster, better, and be more stable.
4. `fc2` - this is the second of 3 fc layers of the neural network. This layer is a liner layer from `fc1_units` and `gc2_units`
5. `fc3` - this is the third of 3 fc layers of the neural network. This layer is a linear layer from `fc2_units` to `action_size`
6. `reset_parameters` - this layer calls the next method

The `reset_parameters` method smooths the weights of the fc layers.

The `forward` method is the heart of the neural network. This method builds a network that maps `state` into action values using the proprieties that were previously defined. This method uses the ReLU function as an activation function. The ReLU function is defined to be:

$$ReLU(x) = max(0, x)$$

This means for any value of x, return the greater of x or 0. The value of x is the returned value from a fc layer. This method also uses the hyperbolic tangent as an activation function. The hyperbolic tangent function is defined to be:

$$\text{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The value of x is the returned value from a fc layer.

Critic

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

    def __init__(self, state_size, action_size, seed, fcs1_units=400, fc2_units=300):
        """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.fcs1 = nn.Linear(state_size, fcs1_units)
        self.bn1 = nn.BatchNorm1d(fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, 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 = F.relu(self.bn1(self.fcs1(state)))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

This model is a relatively simple neural network. This model has 3 methods - `__init__`, `reset_parameters`, and `forward`. The `__init__` method defines all the proprieties that this class has:

1. `state_size` - this is a random value that will be used later. This is another hyperparameter.
2. `fcs1` - this is the first of 3 fully-connected (*fc*) layers of the neural network. This layer is a linear layer from `state_size` and `fcs1_units`
3. `bn1` - this is the batch-normalising layers. This layer can help the network perform faster, better, and be more stable.
4. `fc2` - this is the second of 3 fc layers of the neural network. This layer is a liner layer from `fc1_units` plus `action_size` and `fc2_units`
5. `fc3` - this is the third of 3 fc layers of the neural network. This layer is a linear layer from `fc2_units` to `1`
6. `reset_parameters` - this layer calls the next method

The `reset_parameters` method smooths the weights of the fc layers.

The `forward` method is the heart of the neural network. This method builds a network that maps `state` into action values using the proprieties that were previously defined. This method uses the ReLU function as an activation function. The ReLU function is defined to be:

$$ReLU(x) = max(0, x)$$

This means for any value of x, return the greater of x or 0. The value of x is the returned value from a fc layer.

Other Hyperparameters Other hyperparameters include the following:

- `BUFFER_SIZE` - this is the replay buffer size and was initially set to `int(1e6)`
- `BATCH_SIZE` - this is the minibatch size and was initially set to `128`
- `GAMMA` - this is the discount factor and was initially set to `0.99`
- `TAU` - this is used in the soft update of the target parameters and was initially set to `1e-3`
- `LR_ACTOR` - this is the learning rate of the Actor and was initially set to `1e-3`
- `LR_CRITIC` - this is the learning rate of the Critic and was initially set to `1e-3`
- `WEIGHT_DECAY` - this is the L2 weight decay and was initially set to `0`
- `LEARN_EVERY` - this is the learning timestep interval and was initially set to `20`
- `LEARN_NUM` - this is the number of learning passes and was initially set to `10`
- `GRAD_CLIPPING` - this is the gradient clipping factor and was initially set to `1.0`
- `OU_SIGMA` - this is the first of two Ornstein-Uhlenbeck noise parameters and was initially set to `0.15`
- `OU_THETA` - this is the second of two Ornstein-Uhlenbeck noise parameters and was initially set to `0.05`
- `EPSILON` - this is helps determine how much noise is added to the state and was initially set to `1.0`
- `EPSILON_DECAY` - this is also helps determine how much noise is added to the state was initially set to `1e-6`
- `seed` - this helps determine the degree of randomness and was seed to `9*5` after experimentation

Ideas for Future Work

Although success was achieved in the present project, there are methods through which the project could be improved. These include:

- Achieving success in the same environment in less than 1138 episodes and/or less than 55.26 minutes
- Comparing the following learning algorithms:
  - Prioritized Experience Replay (*PER*)
  - DDPG
- Further documenting the agent's experience, such as through a .GIF or an online video