UDACITY

Machine Learning Engineer (MLE) Nanodegree
Capstone Report

Training Deep Reinforcement Learning (DRL) Agents to Play Pommerman

Kevin R. Chen

September 10, 2019

Definition

Project Overview

The goal of this project is to benchmark the three fundamental deep reinforcement learning (DRL) models by running and comparing their performance in an AI research environment called Pommerman[1], a variant of the game Bomberman. These three models are vanilla Deep Q-Network (DQN), vanilla Deep Deterministic Policy Gradient (DDPG), and Twin-Delayed DDPG (TD3).

Problem Statement

The challenge: find out which of the following AC model Agents can defeat a Random Agent in the game of Pommerman on the most efficient and consistent basis. In this Pommerman game, whichever Agent gets hit by a bomb radius first loses, regardless of whether the bomb is set by the opponent or the own Agent. The game format can be set as either 1v1 or 2v2 team of Agents, although this project will only focus on 1v1 as simplification. Agents in the environment will also encounter obstacles, including wooden and concrete walls, as well as upgrades.

The principal reason why Pommerman is created is to gamify the performance benchmarking and testing of various reinforcement learning algorithms. This is essential because RL is generally prone for being unstable and thus difficult to reproduce[2], especially more novel approaches.

Metrics

The primary metric is the average scoring derived from the rewards collected from the Pommerman env. For each episode (or game) the max reward is 1 (winning the game) while the min reward is -1 (losing the game). However, it is likely that the agent can get a score between [-1, 1] by earning positive rewards for subtasks including getting upgrades, planting bombs, destroying wooden walls, and surviving longer. This allows our agent to continue learning and improving even if fails to achieve its ultimate goal to taking down the opposing agent and winning the game. Additional details are found in the Benchmark and Implementation sections.

Analysis

Data Exploration Description

Pommerman's environment is a 11x11 board, or more specifically a flattened 121 integer vector. For each battle round, although the board is randomly initiated, all the agents (up to 4) will start at a corner.

There are 6 actions [0,5] that an Agent can make per timestep: move up, move down, move left, move right, plant bomb, and do nothing/pass. Each agent can only carry and plant one bomb at a time. Once an agent plants a bomb on a square, the agent cannot plant another one until the bomb has been

---

1 Pommerman <https://www.pommerman.com/>

2 Henderson, Peter et al. "Deep Reinforcement Learning that Matters" McGill University: arXiv:1709.06560v3 [cs.LG] (2017).

donated. The time from plant to detonation takes 10 “time steps”. The blast radius is 2 squares in all cardinal directions. If an agent (whether same or opposing team) is in the blast radius when denoted, then the agent is removed from the game. If all of the agent(s) from a side is gone, the other side wins and the battle round ends.

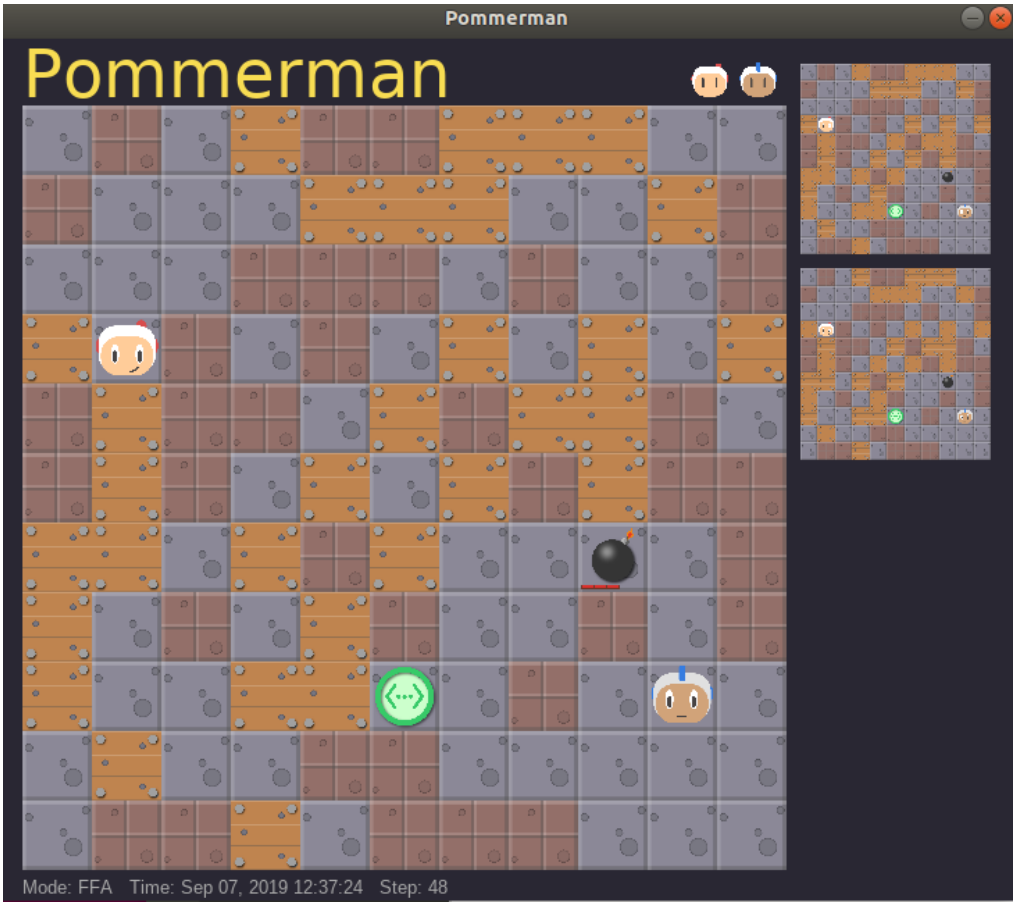
Randomly scattered around the board are wooden and rigid walls, which agents cannot step on. Wooden walls can be removed by a bomb while rigid ones cannot. There are also power-ups that agents can collect to enhances their strengths, from wider blast radius to shorter time-to-detonation.

The input of the agents is this 121 integer vector environment board, each grid populated by numbers presenting obstacles, upgrades, bombs, agents, or nothing at all. The output is 1 of the 6 action choices per agent to move in this environment board. An illustration will be given in the Exploratory Visualization below.

Although the Pommerman env uses the original OpenAI gym utility framework for spaces and seeding, it has its own characters and graphics.

Exploratory Visualization

Here is a rendered Pommerman env snapshot in the middle of a game:



Here is a condensed data output that detailed the env state.

```
{
        'alive': [10, 11],
        'board': array([[ 0, 1, 0, 2, 1, 1, 2, 2, 2, 0, 0],
                        [ 1, 0, 0, 0, 2, 2, 2, 0, 0, 2, 1],
                        [ 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1],
                        [ 2, 10, 1, 0, 1, 0, 2, 0, 2, 0, 2],
                        [ 1, 2, 1, 1, 0, 2, 1, 2, 2, 1, 0],
                        [ 1, 2, 1, 0, 2, 0, 2, 1, 2, 1, 1],
                        [ 2, 2, 0, 2, 1, 2, 0, 0, 3, 0, 1],
                        [ 2, 0, 1, 0, 2, 1, 0, 0, 1, 0, 1],
                        [ 2, 0, 0, 2, 2, 7, 0, 1, 0, 11, 0],
                        [ 0, 2, 0, 0, 1, 1, 0, 0, 0, 0, 0],
                        [ 0, 1, 1, 2, 0, 1, 1, 1, 0, 0, 0]]),
        'bomb_life': 6,
        'game_type': 1,
        'game_env': 'pommerman.envs.v0:Pomme',
        'position': (8, 9),
        'blast_strength': 2,
        'step_count': 47
}
```

As depicted in the visualized, there are two agents: Agent0 [10] and Agent1 [11]. Agent0 is located on (3, 1) on the board while Agent1 is located on (8, 9). Agent1 just had planted a Bomb [3] on (6, 8) that has 6 more timesteps before it explodes. There is also an Increase Range Power-Up [7] on (8, 5) that Agent1 can pick up to increase its bomb's blast radius. There are numerous Rigid Wall [1] and Wooden Wall [2] obstacles scattered across the board. The rest are Passages [0] or open space for agents to freely traverse through.

As further described in the Pommerman online doc[3], here are the summarized mappings between the visual rendering the data output vector:

Board: The 11x11 board is a numpy array where each value corresponds to one of the representations below:

- Passage = 0
- Rigid Wall = 1
- Wooden Wall = 2
- Bomb = 3
- Flames = 4 (radius of the bomb blast)
- Extra Bomb Power-Up = 6 (adds ammo)
- Increase Range Power-Up = 7 (increases the blast strength)
- Can Kick Power-Up = 8 (can kick bombs by touching them)
- Agent0 = 10
- Agent1 = 11

3 Pommerman Environment Doc <https://docs.pommerman.com/environment/>

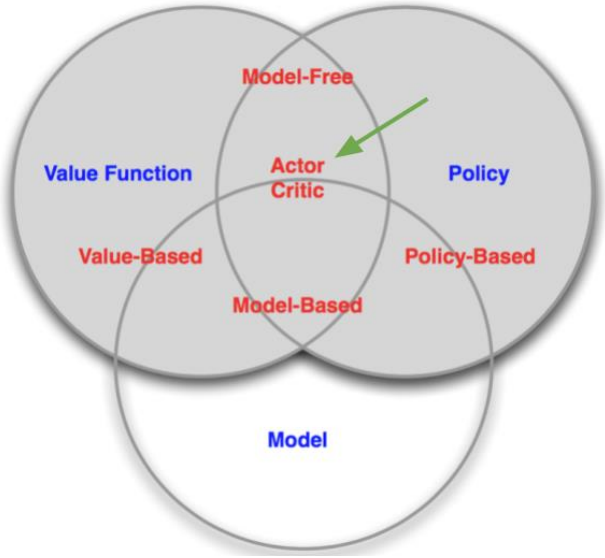
Algorithm and Techniques

In RL literature, there are two fundamental categories of methods: value-based and policy-based. Value-based methods (i.e. DQN) try to best estimate the value of each state or state-action pair, using this information to determine the best action policy for each state or scenario. Policy-based models (i.e. REINFORCE) directly maps states to their expected optimal actions without the need of calculating the values. Both value-based and policy-based methods have their pros and cons. While value-based methods are good for discrete action spaces, it is more difficult to scale and takes longer to train due to the extra value calculation steps. While policy-based methods are more scalable to continuous spaces and are more "lightweight", it tends to be more unstable due to the difficulty to calculating how optimal a policy is without values.

Actor-Critic (AC) methods is a hybrid value and policy-based approach and tries to get the best of both methods.

- The "Actor" acts as policy-based method to execute the expected optimal action for current state at every timestep
- The "Critic" critiques with a value-based method to determine the value of the current state-action pair, calculates the net gain/loss based on the actual reward(s) received from env, and then use that to update or "learn" both "Actor" and "Critic" models.

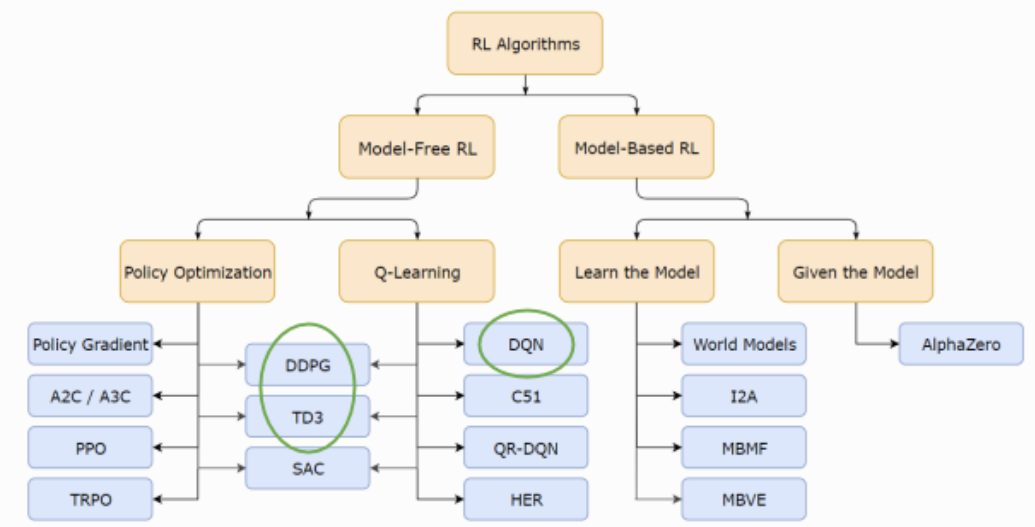
The benefit of AC models is how customizable they are. For example, while the Actor runs every step, the Critic can run more sporadically (similar to n-step bootstrapping). Therefore, ACs can maintain the "lightweightness" of policy-based methods while gaining the stability of value-based methods. As illustration, here are the holistic RL agent categories[4]:



4 SALLOUM, Ziad. "Top Down View at Reinforcement Learning" *Towards Data Science* Medium post (2019)

For this project, I will use one value-based Q-Learning method (as control) and two hybrid AC methods:

1. Value-based: Deep Q-Networks (DQN)[5]
2. AC-based: Deep Deterministic Policy Gradient (DDPG)[6]
3. AC-based: Twin-Delayed DDPG (TD3)[7]



DQN: Deep Q-Networks is known as the introductory model of combining reinforcement learning (RL) algorithms, such as Temporal Difference or TD Learning, with deep learning (DL) models. It uses deep neural networks to approximate the state-action values based on the observations seen in the environment. To further address instabilities, DQN is also known contain the following 2 features:

1. Experience Replay: It allows DQN to randomly sample a mini-batch of past observations (or "experiences") to learn from that. This is opposed to learning the experience tuples from the current episode or trajectory τ, which likely highly correlates with one another, resulting on more biased learning.

2. Fixed Q-Targets: This is duplicate target network of DQN, except its "stickies" the parameters weight w for several steps. This allows the actual (or "local") DQN network use this more stable target network weight, or w^-, to calculate to TD target to get the TD error, instead of using the current weight w. Using the current weight may cause correlations and thus bias.

---

5 Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning" Nature518.7540 (2015)

6 Lillicrap, Timothy P. et al. "Continuous control with deep reinforcement learning" Google DeepMind: arXiv:1509.02971v6 [cs.LG] (2015).

7 Fujimoto, Scott et al. "Addressing Function Approximation Error in Actor-Critic Methods" McGill University: arXiv:1802.09477v3 [cs.AI] (2018).

DDPG: It is an AC model that is known to emulate DQN. The DDPG Actor approximates the optimal policy in the deterministic fashion. Unlike other stochastic AC methods where Actors that can output probability distribution of actions, the DDPG Actor outputs only the expected max action of the current state, or $argmax_a Q(s, a)$. The DDPG Critic will then use this action to output the state-action value for the current policy parameters θ, or $Q(s, argmax_a Q(s, a); \theta_Q)$.

Like DQN, DDPG also has Experience Replay Buffer and this Fixed Q-Target networks, one of its Actor and one for its Critic. However, unlike DQN, DDPG uses soft updates, which gradually copies the local parameters to its targets instead of all at once in n-steps, like in DQN. This Pommerman project is a good env to try out DPPG, as it only has a single digit amount of actions, and do not need to worry about continuous action spaces, which suits better for A3C and A2C models.

TD3: Twin-Delayed DDPG that improves upon vanilla DDPG using 3 tricks[8]:

1. Clipped Double-Q Learning: TD3 learns two Q-functions instead of one and use the smaller Q-value for the error loss function. Hence the term "twin".

2. Delayed Policy Updates: TD3 updates policy and target networks less frequently than the Q-function. A recommend amount is one policy update for two Q-function updates.

3. Target Policy Smoothing: TD3 add random noise to target action to prevent policy from overfitting the Q-functions errors

Benchmark

Here are the benchmarks for the DRL agents both during training and during runtime.

Training

Each DRL Agent will be given 2000 episodes to train, with reward-per-# episode graphs shown to depict their efficiency of training. Each Agent's training will also be timed to determine the duration to get through the 2000 episodes training. During training, each DRL Agent will be trained via a SimpleAgent, which controls are dynamically programmed.

Runtime

Once the DRL Agents are trained by the dynamically-programmed SimpleAgent, they will then be run against a RandomAgent, which will be an agent that will randomly take steps. Each DRL Agent will be run 5 times with a different seed, each seed with 100 episodes. The episodes will then be averaged for each Agent to get a grand composite score to compare the performance of each Agent.

The fundamental DQN Agent can be considered as a comparison benchmark for the two Actor-Critic Agents, while the vanilla DDPG Agent can be considered as the comparison benchmark for the more refined TD3 Agent. For runtime, each DRL Agent will compete against several HybridAgents, which actions will be mostly driven by a SimpleAgent but will also periodically take random actions.

---

8 Open AI Spinning Up doc: TD3 <https://spinningup.openai.com/en/latest/algorithms/td3.html>

Methodology

Data Preprocessing

Because Pommerman I/O environment has already been clearly defined (as specified in the Data Exploration Description), no further data preprocessing step is needed. The only exception is the need to convert from Python lists provided by the env into Tensors in order to be fed into PyTorch-based NN models of the DRL agents.

Implementation

Implementation preparation include the following:

- Environment setup
- Structure of the DRL algos (DQN, DDPG, TD3)
- Structure of the underlying NN models that makeup the DRL algos
- Initial hyperparameter setups of the DRL algos

Environment

The environment will be "PommeFFA-v0", fast free-for-all Pommerman game.

DRL algorithms

DQN: vanilla DQN with Experience Replay Buffer and fixed Q-Target network with Tau of 0.1. It will update the local policy network once every two steps. To see the source code, please go to the /pommerman/agents/dqn/* directory.

DDPG: vanilla DDPG with an Actor and Critic, each has a local and target network. Unlike, it updates parameter weights using policy gradient method. Further modifications will be made in TD3. To see the source code, please go to the /pommerman/agents/ddpg/* directory.

TD3: Added the following from DDPG:

1. Clipped Double-Q Learning: clipped with [-1, 1] to constrain the action weights to within this boundary.
2. Delayed Policy Updates: Changed UPDATE_EVERY from 2 steps to 4, hence doubling the delay. Also lowered the Tau by a magnitude of 10, from 0.1 and 0.01. Therefore, a tenth less will be updated from the local network for each update.
3. Target Policy Smoothing: Added Ornstein-Uhlenbeck random noise (OU-Noise) to the agent action weights to prevent overfitting of the Critic's value estimate, due to likely error of the NN function approximation.

To see the source code, please go to the /pommerman/agents/td3/* directory.

Model

For consistency, all the DRL algorithm have the same NN network layer makeup. It will be a 4-layer NN with 1 input, 2 hidden, and 1 output layers. It uses ReLu activation functions in-between layers, softmax activation function for Actor's NN, and linear activation function for the Critic and Q-Network's NNs.

- States input: 396 neurons (provided by the env)
- Fc1: 128 neurons

- Fc2: 128 neurons
- Actions output: 6 neurons (as required by the env)

Initial Hyperparameters

- BUFFER_SIZE = 100,000 (for all)
  - Desc: max capacity of experience replay buffer
- BATCH_SIZE = 128 (for all)
  - Desc: mini-batch sample size from experience replay
- GAMMA = 0.99 (for all)
  - Desc: reward discount factor
- TAU: 0.1 (DQN and DDPG), 0.01 (TD3 only)
  - Desc: the amount for local network to update the target network
- WEIGHT_DECAY = 0 (for all)
  - Desc: L2 regularization weight decay
- EPS_START = 1.0 (DQN only)
  - Desc: initial epsilon ε, which is 100% randomized action step
- EPS_END = 0.01 (DQN only)
  - Desc: the lower bound of ε, the prob of a randomized step is never < 1%
- EPS_DECAY = 0.998 (DQN only)
  - Desc: the amount of decrease in ε, %-wise
- LEARNING_RATE = 0.0005 (DQN only)
  - Desc: learning rate of the Q-Network
- LR_ACTOR = 0.0002 (DDPG and TD3)
  - Desc: learning rate of the Actor network
- LR_CRITIC = 0.0002 (DDPG and TD3)
  - Desc: learning rate of the Critic network
- UPDATE_EVERY = 2 (DQN and DDPG), 4 (TD3 only)
  - Desc: how often to update/learn the policy network, amount of *n*-steps
- UPDATE_AMOUNT = 1
  - Desc: amount of learning updates (times) to the policy network

The refined hyperparameter source code is in /pommerman/agents/utils/hyperparams.py.

Complications

There were two complications when implementing this project. The first one is how the Pommerman env is designed. It requires the list of agents input in order for the env to be created. Then the env will perform the act function to output values rather than the agents themselves. However, since the env does not have the step function, so I have the DRL agent then perform the step function in order to learn and populate its Replay Buffer. In top of that, the env requires all agents to inherit from its designated parent class BaseAgent. Therefore, when I developed the DRL agents, I had to be mindful about this inheritance.

The second complication was some confusion about what is deemed regular DDPG and what is deemed TD3. When I was taking the Udacity's Deep Reinforcement Learning (DRL), when it introduced the topic DDPG, it also mentioned the use of OU-Noise, clipping to smooth the policy, and slowing the target updates by decreasing the value tau τ. That is basically what TD3 is trying to accomplish. However, that Udacity made no mention about TD3 in its DRL course, so I am struggling how figure out how to best

differentiate between DDPG and TD3 for this project. My decision (and may not the most accurate one) is to completely "gut" out the DDPG algo, including removing the OU-Noise, and try to keep it close to the original definition as possible. However, as reflected in the Training and Results, that makes DDPG more difficult to train, as it does not have as much capacity the randomly explore the environment as DQN or TD3.

Refinements

Throughout the implementation process, I have made the following modifications:

1. Updates the NN models. It will now be a deeper 5-layer NN with 1 input, 3 hidden, and 1 output layers, with the hopes of better fitting the model.
    a. States input: 396 neurons
    b. Fc1: 256 neurons
    c. Fc2: 128 neurons
    d. Fc3: 64 neurons
    e. Actions output: 6 neurons
2. Replace all ReLu with Leaky ReLu.
3. For DQN, lowers the epsilon decay from 0.998 to 0.995. That way DQN can take greedy actions earlier in the training.
4. For TD3, decrease the clipped amount from [-1, 1] to [-0.9, 0.9]. That way, it further constrains the spikes of action decisions to prevent overestimation.
5. I decided to copy this Pommerman project workspace to the Udacity's Workbook environment to utilize the remaining GPU credits that I had. I then saved the model checkpoints to local drive.
6. When running, use 5 random seeds to initialize each agent instead of just 1. Picked the following five random seeds: 0, 2, 10, 25, 42, 64.
7. During runtime, test the DRL Agents on multiple HybridAgents with different SimpleAgent vs. random action step probabilities using the parameter epsilon E, which is different from DQN's ε.

Training

The following training source code can be found in /train_report.ipynb. It is a copy of the file /train.py. Here is the training result for DQN, throughout 2000 episodes:

```
# DQN
start_time = time.time()
scores = train(env, n_episodes=2000)
end_time = time.time()
```

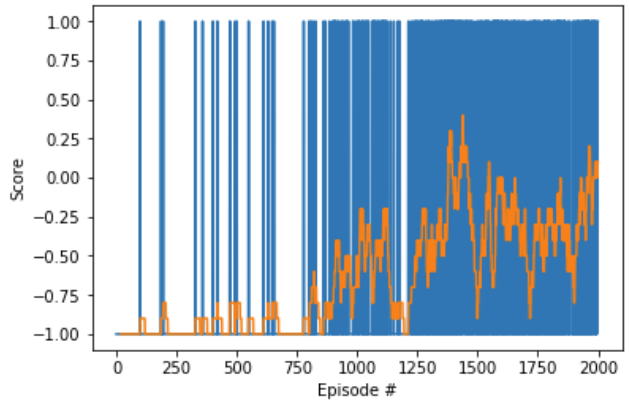
```
Episode 200     Avg Score: -0.96
Episode 400     Avg Score: -0.96
Episode 600     Avg Score: -0.96
Episode 800     Avg Score: -0.98
Episode 1000    Avg Score: -0.60
Episode 1200    Avg Score: -0.74
Episode 1400    Avg Score: -0.24
Episode 1600    Avg Score: -0.26
Episode 1800    Avg Score: -0.44
Episode 2000    Avg Score: -0.08
```

```
# GPU time
total_time = end_time - start_time
print(f"{round(total_time/60, 2)} min")
env.close()
```

```
63.5 min
```

Here is the graphical version. As you see, there is a lot of volatility of each episode's score (in blue) because the DQN Agent can get lucky and beat the SimpleAgent, resulting in the max score of 1. Nevertheless, you can see the score rolling average (in orange) trending upward, signifying the DRL Agent is gradually learning overall. It started from -1 and increased upward to 0. The final saved Q-Network model checkpoint for DQN can be found at /pommerman/agents/dqn/models/checkpoint.pth.

```
# DQN Plot the scores ################################################
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.plot(np.arange(len(scores)), pd.DataFrame(scores).rolling(20).mean(
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```



Here is the training result for DDPG, throughout 2000 episodes:

```
# DDPG
start_time = time.time()
scores = train(env, n_episodes=2000)
end_time = time.time()
```

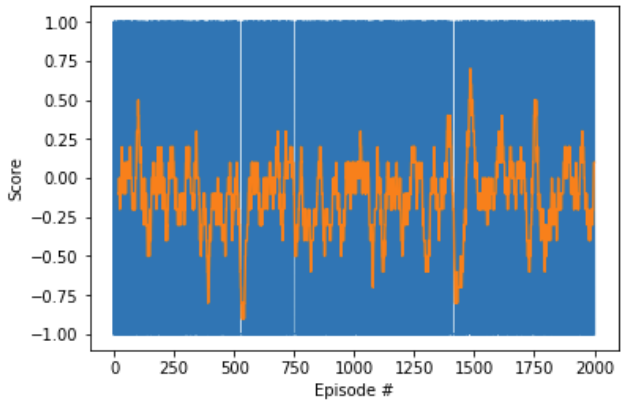
```
Episode 200     Avg Score: -0.10
Episode 400     Avg Score: -0.18
Episode 600     Avg Score: -0.24
Episode 800     Avg Score: -0.08
Episode 1000    Avg Score: -0.10
Episode 1200    Avg Score: -0.22
Episode 1400    Avg Score: 0.002
Episode 1600    Avg Score: -0.08
Episode 1800    Avg Score: -0.18
Episode 2000    Avg Score: -0.08
```

```
# GPU time
total_time = end_time - start_time
print(f"{round(total_time/60, 2)} min")
env.close()
```

```
192.85 min
```

Here is the graphical version. Unlike in DQN, the orange line has oscillated around the score 0, signifying that it has not been learning too much. That is because a mentioned in Complications, I removed the OU-Noise from DDPG, thus the Agent remains stationary and waits until the SimpleAgent of come of it and defeats it. The DDPG does not explore action including planting the bomb and killing itself in the process prematurely, thus resulting with an even lower score. The final saved Actor and Critic model checkpoints for DDPG can be found at /pommerman/agents/ddpg/models/*.pth.

```
# DDPG Plot the scores ##################################################
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.plot(np.arange(len(scores)), pd.DataFrame(scores).rolling(20).mean(
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```



Here is the training result for TD3, throughout 2000 episodes:

```
# TD3
start_time = time.time()
scores = train(env, n_episodes=2000)
end_time = time.time()
```

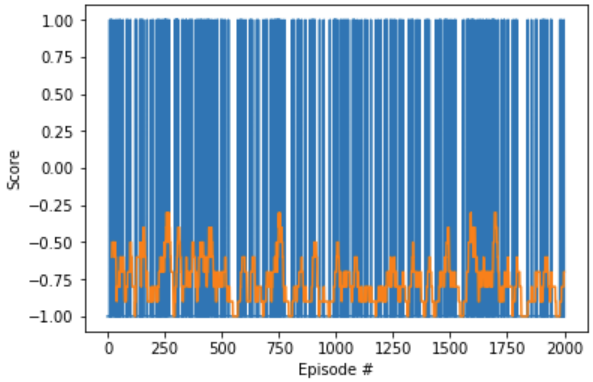
```
Episode 200     Avg Score: -0.74
Episode 400     Avg Score: -0.70
Episode 600     Avg Score: -0.84
Episode 800     Avg Score: -0.74
Episode 1000    Avg Score: -0.84
Episode 1200    Avg Score: -0.86
Episode 1400    Avg Score: -0.80
Episode 1600    Avg Score: -0.72
Episode 1800    Avg Score: -0.80
Episode 2000    Avg Score: -0.82
```

```
# GPU time
total_time = end_time - start_time
print(f"{round(total_time/60, 2)} min")
env.close()
```

```
37.84 min
```

Here is the graphical version, showing the TD3 oscillating around the score of -0.75. That shows that although the Agent is exploring the environment, such as planting the bombing and killing itself, it is not learning effectively from this process. The final saved Actor and Critic model checkpoints for TD3 can be found at /pommerman/agents/td3/models/*.pth.

```
# TD3 Plot the scores ###################################################
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.plot(np.arange(len(scores)), pd.DataFrame(scores).rolling(20).mean(
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```



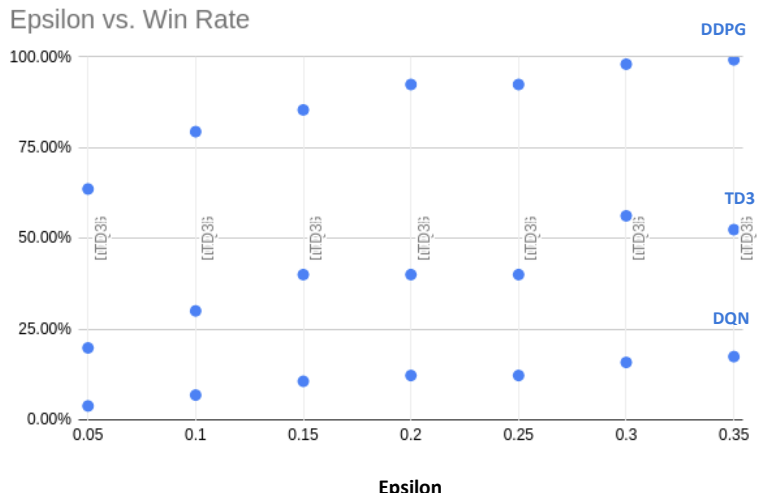
Results

Model Evaluation and Validation

The following data can be found at /run_data.xlsx. It can be reproduced by running /run.py.
After training the DRL Agents. I ran them against a HybridAgent set as various epsilon E. E is the probability the the HybridAgent will take a random step. For example, an Agent with E = 0.25 will take a random step 25% of the time and act as SimpleAgent the remainder of time. For each epsilon, each DRL Agent will run 500 episodes, with the win rate against their respective HybridAgent averaged.

Epsilon	Agent	Average - Win Rate
0.05	DDPG	63.60%
	DQN	3.80%
	TD3	19.80%
0.1	DDPG	79.40%
	DQN	6.80%
	TD3	30.00%
0.15	DDPG	85.40%
	DQN	10.60%
	TD3	40.00%
0.2	DDPG	92.40%
	DQN	12.20%
	TD3	40.00%
0.25	DDPG	92.40%
	DQN	12.20%
	TD3	40.00%
0.3	DDPG	98.00%
	DQN	15.80%
	TD3	56.20%
0.35	DDPG	99.20%
	DQN	17.40%
	TD3	52.40%

Here is the resulting data in graph form. It shows DDPG consistently performing the best across all epsilons, followed by TD3, then finally DQN performing the worst. The reason why all DRL Agent perform better the higher the E is because the HybridAgent is taking more random steps, which increases its likelihood in making mistakes and losing the game.



The source code for SimpleAgent, RandomAgent, HybridAgent, and all other non-DRL Agents can be found in /pommerman/agents/*.py.

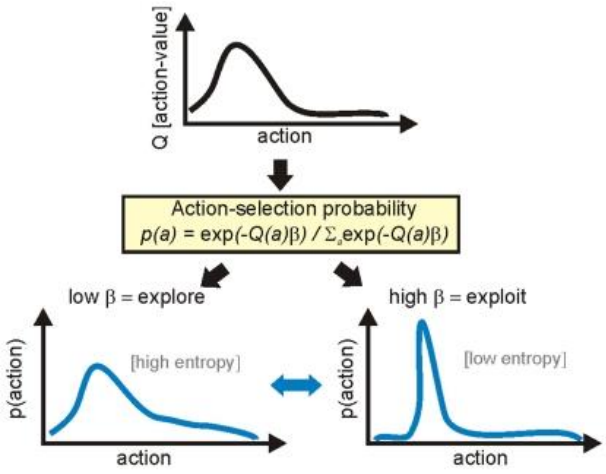
Justification

I think that TD3 performing better in the runtime than DQN is justified because even though its training result is not as good compared to DQN's, TD3 has shown that it is more generalizable than the DQN. This makes sense as TD3 is supposed to be an improvement compared to DQN. As discussed in the Algorithm and Techniques section, Actor-Critic methods is designed to get best traits from both value-based (such as DQN) and policy-based method.

However, I do not think DDPG performing better than TD3 here is justified because DDPG is slow to react, as explained in the Complication section. Therefore, it is basically playing possum and "waiting" for the HybridAgent to make a mistake. As shown in the graph above, this is more effective when the HybridAgent has to take more random steps. However, I do not think the Agent being idle is a viable strategy to winning the Pommerman game, especially against Agents that have matured from random exploration and can exploit approximately every action-state.

Conclusion

Free-Form Visualization



An important quality of this project is how can the models best balance the exploration-exploitation trade-off[9], as depicted in the picture above. This project provides a good example regarding the sensitivity of this balance. Vanilla DDPG is too much exploitation, which gives little to no more to explore. On the other side of the spectrum, DQN is too much exploration, especially if its epsilon ε decays too slowly. That means it will be using more of its early episodes to take random steps in order to further explore the environment states and its actions. Finally, TD3 is somehow in-between, as it is designed to me. Its features such as clipping and delayed updates in order to ultimately smooth the action curve above, in order to prevent overfitting or premature "favoritism" of the actions at hand. I believe that current research of DRL, including further extensions DDPG or more generally Actor-Critic methods, involve finding more dynamic and robust balancing between exploration and exploitation.

Reflection

As this is my first genuine end-to-end RL project, the biggest takeaway I got is the amount of preparation required before even running the first model. I read that > 75% of data scientists' work involves data engineering and wrangling while the rest amount of time is spent on actual ML development, training, tuning, and deployment. That ratio of time spent is similar to me with this project, except instead of data preparation (which there is little here), I spent the following time implementing the coding infrastructure required for me to run my DRL models in top of this Pommerman environment.

Even though I have enrolled in both Udacity's Machine Learning Engineer (MLE) and Deep Reinforcement Learning (DRL) courses, I think I have learned the most via hands-on project. The reason is two-fold. First, I am working with a completely new environment Pommerman, which has not been spoon-fed to me compared to other Udacity projects. Second, I am writing this report, which really help synthesize and digest what I have learned from ML and DRL so far.

---

9 ResearchGate <https://www.researchgate.net/figure/Formalization-of-the-exploration-exploitation-trade-off-In-formal-models-of_fig1_221844558>

Even though I have learned so much, I realized that the road to ML (especially DRL) is still a long way to go. Throughout this challenging process, I have gained more respect and admiration to those ML pioneers who are making breakthroughs in this space, as I have more sense about the trials and tribulations they have to overcome in order make it this far.

Lastly, I am starting to understand my role and where I can best contributor. While, I don't think I can ever become an expert pure-AI researcher, who can develop these cutting edge model and write whitepapers about them, I can become more of an AI practitioner, who can take these models and incorporate them into the next-gen apps. Therefore, my focus moving forward should be less on creating my own basic models from scratch, just like what I did for this project, but more on how I can best utilize existing models out there that were created and improved upon by more seasoned experts.

Improvements

Here is a list of idea I can improve on this project:

1. Prioritized Experience Replays: Using prioritized replay buffers allows for the model to learn from higher priority experiences more frequently, thus enabling more efficient learning.
2. Alternative DRL models: There are other popular models that are worth experimenting on and comparing the result, from pure policy-based methods (i.e. Trust Region or TRPO[10]) to other Actor-Critic methods (i.e. Distributed Distributional DDPG or D4PG[11]). They have their own respective approaches to curtail overestimation bias while enabling more efficient exploration or searching.
3. Hyperparameter tuning: One major parameter I did not have a chance to tune of this project is the learning rates of the different models. Another parameter is the amount of impact the random OU-Noise has in the TD3 action steps, including using some form of regularization. It will be nice for a way to automate and find the best hyperparameters for these models, for example using AutoML[12].
4. Multi-Agent: Since Pommerman allows for
5. Distributed computing: I can try to reproduce this project in Spark[13] on a cloud environment, such as AWS or Google Cloud, to utilize parallel computing in order to achieve much faster training times.
6. Transfer Learning: As mentioned in Reflection, perhaps the most efficient way to improve is to adopt the best models and algos that others in the Pommerman community have already developed and build on top of them. This is where getting comfortable with Transfer Learning methods comes into play.

---

10 Schulman, John et al. "Trust Region Policy Optimization" Department of EECS, University of California, Berkeley: arXiv:1502.05477v5 [cs.LG] (2015).

11 Barth-Maron, Gabriel et al. "Distributed Distributional Deterministic Policy Gradients" Google DeepMind: arXiv:1804.08617v1 [cs.LG] (2018).

12 AutoML – Google Cloud <https://cloud.google.com/automl/>

13 Apache Spark <https://spark.apache.org/>