"Game of Booms": Proximal Policy Optimization for Pommerman

Binbin Xiong, Minfa Wang
{bxiong,minfa}@stanford.edu
Stanford University

Abstraction

In this paper, we propose a policy optimization based algorithm to train an agent to play the game of Pommerman. Pommerman is a play which is stylistically similar to Bomberman, the famous game from Nintendo. Every battle starts on a randomly drawn symmetric 11x11 grid with four agents, one in each corner. Our task is to develop a single agent under FFA mode (Free For All, where four agents enter and one leaves and the board is fully observable) that can beat all other players. After thorough literature review, we choose to develop our agent based on the current state-of-the-art policy gradient methods called Proximal Policy Optimization Algorithm.

Game Setup And Task Definition

The **state** of the game for each agent is represented as a dictionary of following fields:

- **Board**: 121 Ints. The flattened board.
- **Position**: 2 Ints, each in [0, 10]. The agent's (x, y) position in the grid.
- **Ammo**: 1 Int. The agent's current ammo.
- **Blast Strength**: 1 Int. The agent's current blast strength.
- **Can Kick**: 1 Int, 0 or 1. Whether the agent can kick or not.
- **Enemies**: 3 Ints, each in [-1, 3]. Which agents are this agent's enemies.
- **Bombs**: List of Ints. The bombs in the agent's purview, specified by (X int, Y int, BlastStrength int).

For each play, the agents will be put to the corner of a randomly generated board, the game ends when maximum number of time steps are met or only one player leaves.
In each time step, an agent can choose from one of six **actions**:

$$Actions(state) = \{ Stop, Up, Left, Down, Right, Bomb \}$$

Our task is to build a model which given state outputs an action.

Background

Recent years reinforcement learning has been proven effective in many game setup environment where one needs to learn to control agents directly from high-dimensional sensory inputs to achieve certain goals without the necessity of any supervised training labels.

In general, modern solutions to reinforcement learning could be categorized into two different types, search based, and function approximation based. For search based models, the problem could be treated as a Markov Decision Process where traditional solutions includes exact methods such as value iteration and policy iteration. For zero sum games, one could also use purely search based method with search space pruning skills, such as alpha-beta pruning. However, these solutions either require heavy feature engineering or heavy computation resources as they don't generalize well.

Another category of solution is function approximation, where one could use machine learning to model statistics inside the game state, for example, Q-Value, Value, or Reward. Recently, as deep learning advances and showed power in general function approximation, many breakthroughs have been made in areas such as computer vision and speech recognition. This also inspired a series of Deep Learning based Reinforcement Learning methods, while the most common ones are Q-Learning based, which is off-policy learning and Policy Gradient Based, which is on-policy learning.

Q-Learning is to learn the Qvalue given each state and action that minimizes the some loss based on Bellman equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

While in Deep Q-Learning, instead of keeping a table of Q values for each state and action, one could use deep neural nets on top of state and action related features to approximate all Q values. However, in practice Q-Learning could be very complicated since it's complexity is strictly related to number of actions.

Another leading contenders are policy based gradient methods, which includes "vanilla" policy gradient methods, trust region policy gradient methods, and proximal policy optimization methods. The vanilla policy gradient method is based on likelihood ratio policy gradient, where

one compute the expected utility of a policy with definition, i.e., the sum of rewards of trajectory weighted by likelihood of the trajectory:

$$\nabla U(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)}; \theta) R(\tau^{(i)})$$

While vanilla policy gradient method only use empirical based baseline value in the Advantage function, A3C push this one step forward by also apply function estimation not only only state value, but also on the advantage function.

However, even with estimation approximation, A3C still suffers from high variance issues which make the network unstable and difficult to train. Trust Region method is another policy gradient method which is based on importance sampling. The extra KL divergence based penalty is applied to keep the update localized, thus to decrease learning variance.

$$\underset{\theta}{\text{maximize}}\, \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t - \beta \, \text{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)] \right]$$

Proximal policy gradient method is developed on top of trust region method, which has the benefits of trust region policy optimization but is much simpler to implement, more general, and have better sample complexity.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \Big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \Big]$$

As we can see from their defined objective above, instead of using KL divergence penalty, they use a much simply clipped penalty to keep update localized. With this schema, it only ignores the change in probability ratio when it would make the objective improve, and include it when it makes the objective worse.
We will give more modeling details in the following sections as how we build our deep network for function estimation on top of this method.

Modeling & Algorithm

Overview

In this section we describe in detail about our model and show how it works with concrete examples. Our proposed modeling is based on two parts, one is the *Proximal Policy Optimization* based algorithm, while the other is for policy value estimation.

During training, our proximal policy optimization based learning runs as follow:

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

We now define how we compute the **advantage estimates**. Intuitively, a good agent for Pommerman should be aware of at least the two group of information: information of enemies and current environment. While all of this information is already provided in the given state input (Note that each state feature is a list of length 4, representing states for all agents). With it, we could compute an estimated value function $\hat{V}(s_t)$. This function is modeled by a deep neural network and is elaborated in next section.

For each finished episode, we could also collect the actual discounted sum of rewards $Q_\pi(s_t;\ \theta)$ obtained by the learning agent. Then the advantage estimate is just: $\hat{A}_t = Q_\pi(s_t;\ \theta) - \hat{V}(s_t)$.

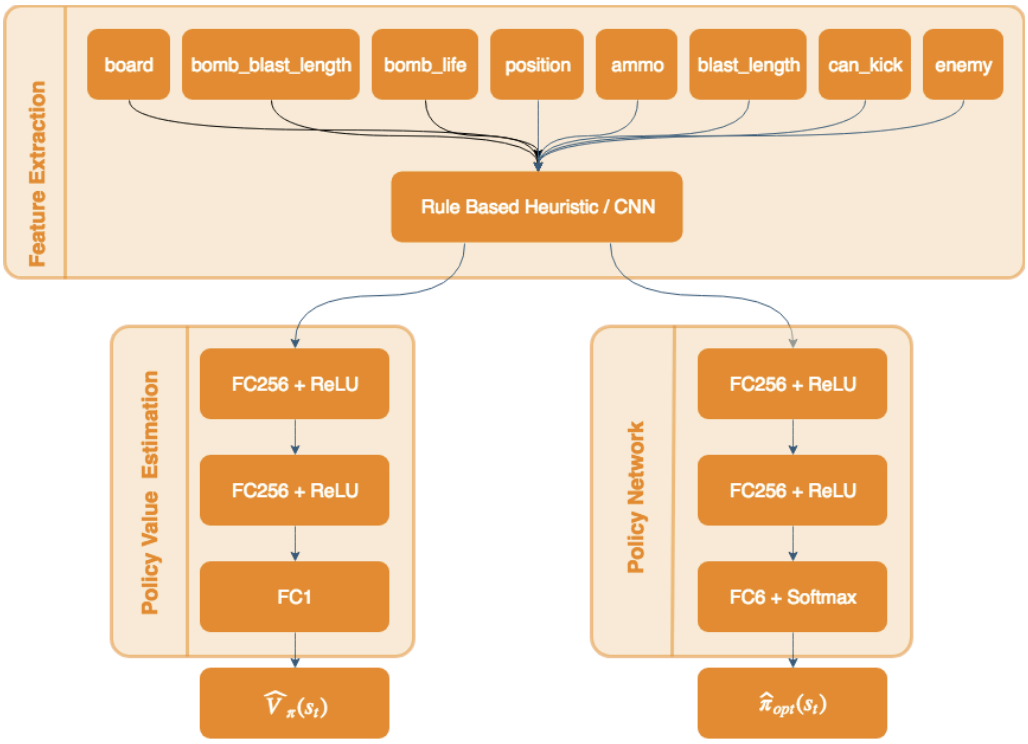
For each batch of episodes, we will try to optimize a **policy network** $\hat{\pi}_{opt}(s;\ \theta)$, which is modeled as a deep neural network to maximize the following objective:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

At serving time, given an observation/state, we would run the feature extractions and feed the feature embedding into a policy network $\hat{\pi}_{opt}(s;\ \theta)$ to derive the optimal action.

Mode Architecture

The diagram above illustrates our overall model architecture. At a high level, the architecture contains three modules: **Feature Extraction**, **Policy Value Estimation**, and **Policy Network**. We intentionally design the architecture to be modularized and hence easy to make incremental improvements. Feature extraction module is mainly responsible for processing observation data and produce input embeddings. Policy value estimation and policy network are two individual network on top of the feature extraction module. The former is approximating the expected reward by following the existing policy π from the current state s_t and onward. The output of this module is a single float number, while for policy network, it computes the estimated optimal policy as the baseline. It's worth mentioning that using a policy network as baseline is crucial in helping reduce learning variance and help our model to learn faster overall.



In this project, we experimented with two different types of feature extraction techniques, one with feature engineering by hand where we extract high order features based on our own experience, while the other is to directly apply convolutional neural networks on top of raw input feature maps. We will give more detail about each of the approach in the following section.

Feature Engineering

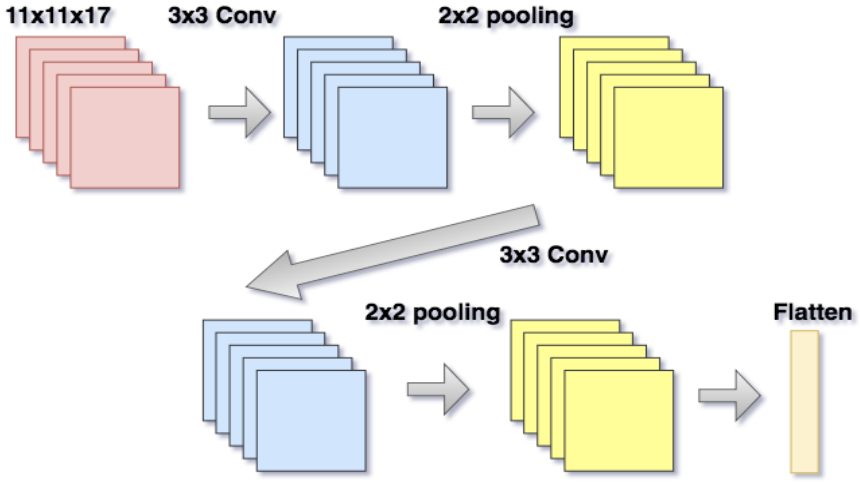
For value and baseline model estimation, we first experimented fully-connect network with features engineered by hand. Inspired by our own play, we extracted the following features from the given observations state:

- Distance from agent to enemies
- Number of enemies within certain range
- For each bomb on the graph, we obtain the following information:
    - Bomb have player in range
    - Bomb have wood in range
    - Bomb strength
    - Bomb life
- Number of valid directions that agent can move
- Distance to ammo power up
- Distance to can kick power up
- Distance to bomb strength power up

We then concatenate the features and feed them to a fully connected neural network to generate embeddings which map to the six possible actions at the very end. It's worth mentioning that for distance measure, we tried both Manhattan distance, and true distance (since some of the cells are not passable due to being blocked by walls). For each of the distance measure, we also included their invert value. Using invert value has a few advantages, for example, we could easily set not available values as negative, which aligns well with the fact that the higher the invert value is, the closer the distance is.

Convolutional Neural Network

Recent years, convolutional neural network has been proven effective in various of vision tasks, such as object identification, scene understanding and image classification. In some dataset such as ImageNet, deep, and big neural networks built with convolution layers has already outperform human performance. One intuitive understanding for the success of convolutional neural nets are in that they are effective at extracting spatial information which mimic the biological structure of eyes. In the above handcrafted feature engineering network, we use various of distance measures, which aim at capture the spatial information. With convolutional networks, we could allow the model to learn automatically how to extracted spatial patterns from the graph based on its play experience.



However, what's given in the observation data structure is not well suited as input to the convolutional layer directly, in that the numbers in the input map such as board is categorical. For example, value 11 and 12 doesn't mean they are bigger than 0, but rather mean enemy 11, enemy 12. Inspired by work done in Backplay, we designed the following feature 17 maps each of size (11,11,1)

- Binary maps contain the integer values of each bomb's blast strength, remaining life at the location of that bomb.
- Binary maps containing the position of all agents, 1 at agent's position, 0 otherewise.

- Binary maps containing the status of agent's current bomb count, blast radius, can kick or not.
- Binary maps containing information about passages, ridig walls, wooden walls, extra-bomb power-ups, increase balse strength power-ups and kicking-ability power-ups.
- Finally, a map with float ratio of the current steps to the total number of steps. This is helpful to let the agent know about game timeout threshold.

We further concatenate the feature maps into a 17 channel, 11 by 11 map, and then feed it to following convolutional layers. We then use three convolution layers interleaved with three max pooling layers before feeding them to the following policy value estimation network and policy network.

Rewards

The original game rule has the following reward system:

- $if\ isJustDead(agent):\ reward\ =\ -1$
- $if\ isLastSurvivor(agent):\ reward\ =\ +1$
- $otherwise:\ reward\ =\ 0$

Basically, all the meaningful rewards are only granted at the end of the game. This is a very sparse learning system. Given that a regular game could easily span over 200 steps, this learning environment is quite harsh for our agent to learn any meaningful behaviors. Therefore, we supplement the following rewards/feedbacks to the existing system:

- $if\ isAnyOpponentJustDead():\ reward\ = 0.1$
- $if\ eatPowerUp():\ reward\ =\ 0.1$

These additional rewards are designed to facilitate the exploring behavior of the agent. The first condition encourages the agent to lay more bombs to damage opponents, and the second condition encourages the agent to run into the power-ups.

Backplay

In our experimentation of several different algorithms to help the agent master Pommerman, we observed that there are two major issues with the learning process:

1. In the first a few thousand episodes, the agent could barely win any game. So despite of our attempt of more dense reward settings, the agent could not learn an effective strategy to beat other opponents.
2. In the cases where the agent wins, it is due to suicides of other opponents. So the agent just learns to dodge the bombs at its best effort. A side effect is that the agent gradually learns an extremely conservative strategy that barely lay any bombs.

With these two issues in mind, we started to explore methodologies to alleviate them, and we discovered a technique called Backplay, which is introduced by Resnick et al. and is designed to increase sampling efficiency in reinforcement learning settings. The main idea is that instead of learning from the beginning of the episodes, we first record a full episode, and then let the agent to learn from the later states of the episode and gradually trace back to earlier states.

In Backplay, there are higher likelihood that the agent is starting learning from a strongly advantageous state in the game, and hence brings more frequent cases that the agent survives till the end, and it alleviates issue 1). For the similar reason, in the advantageous state, it encourages the agent to explore and lay bombs, and hence alleviates issue 2). We adapted the idea and transformed it into the following algorithm:

$demoGym := \ pommerman\ environment\ with\ 4\ heuristic\ agents$
$gym := \ pommerman\ environment\ with\ 1\ learning\ agents\ and\ 3\ heuristic\ agents$
$for\ episode\ in\ 1,\ 2,\ 3,\ ... :$
    $reset\ demoGym,\ gym$
    $states\ = demoGym.recordFullGameStates()$
    $states\ = downSampleEvenlyWithMaxCount(states,\ maxCount)$
    $if\ didAgentLoseOrTie() :$
        $states\ =\ states[: 1]\ with\ probability\ dropProb$
    $while\ not\ states.empty() :$
        $state\ =\ uniformSampleAndPop(states[-\ sampleSize\ :])$
        $gym.LearnWithInitState(state)$

There are several hyper-parameters in this algorithm:

- The $maxCount$ is used in the down sampling step limiting max number of different states to keep from a demo game. It ensures that the model is not training excessively in very long episodes, which will cause sampling inefficiency and is prone to overfitting. Our experiment uses $maxCount\ =\ 100$.
- The $sampleSize$ is used in the uniform sampling operation. Instead of backplay the game frame by frame deterministically, each time we sample one state from the last a few states. This treatment adds more uncertainties to the game and helps generalization. Our experiment uses $sampleSize\ =\ 10$.
- The $dropProb$ is a mechanism to reset the states space to only the initial state when the agent is losing. Because in the $demoGym$ the 4 heuristic agents are playing again each other, and there is only ~18% of the chance that the learning agent will be on the winning track. It would result in a data imbalance without this resetting step. Our experiment uses $dropProb\ =\ 0.75$.

Experimentation

Evaluation Metrics

The intrinsic metric may evolve as the project goes along. For now, we choose random walking agent as our enemy to compete against to. Each timestamp, each agent will be given an reward = { +1 if it's the only survivor and the game is end, 0 if it's still alive, -x: if it's dead }. We normalize the rewards to make it a zero-sum game.

Model Performance

We have experimented with multiple different models, with the difference of model architecture (CNN), dense reward function (DR), and backplay technique (BP). We evaluated the model on their average reward across thousands of episodes. The Heuristic agent is our baseline, which is provided by the starter code.

Model	**Average reward**	**Average timestamp**
Heuristic agent	-0.46	-
PPO (NN) + DR	-0.60	192
PPO (NN) + BP + DR	-0.49	196
PPO (CNN + NN) + BP + DR	-0.45	212

The plots above compare different models based on average reward (left plot) and average timestamp (right plot) respectively. From the reward plot, we could see that the CNN with backplay achieves the best performance, which is about the same as heuristic agent. The different models overall learn to converge to similar number of timestamps.

Agent move patterns analysis

We further looked into the game replay to try to understand the moving patterns of agents under different models. In general, what we observed for models to have in common is that all agents tends to stay in corners, or behind the blocks where the bombs are unreachable. Since in many cases, the heuristic agent could kill themselves, which means our agent could simply win by staying still in some cases. This is especially true for models trained with Backplay. In the early stage, the agent could only win if other enemies kill themselves. With biased sampling in backplay, the agent tends to be more static. For model trained with hand crafted features, although we put the number of movable actions in the input feature, sometimes it would still go into a dead end and stay there, which make it very vulnerable from enemies in that enemies could simply lay a bomb to the only opening of the dead end and kill our agent. This issue was alleviated by Convolution Networks. It seems the added convolution layers successfully helped our agent to recognize the dead end location pattern, which enables the agent to avoid it in a smarter way.

Another pattern that we observed was that the agents tends to get more active when meeting with enemies. This strategy could in general help agents to dodge bombs laid by enemies. They also sometimes lay bombs when enemies are around. Among all the models, agent learnt with (CNN + NN) + BP + DR did best in dodging bombs and laying bombs in such situation, although the performance is very unstable, and our agent could still sometimes kill itself.

Besides, none of the agents learnt to explore the map for power-ups, nor learned how to use kick power-ups, even if they capture it. We think this is caused by not enough training iterations. If given more time to train the model, and let agent to explore more game scenarios, it might be able to capture strategies with power-ups.

Conclusion

In this project, we explored ways to tackle the Pommerman game using proximal policy optimization framework. We experimented with different techniques to facilitate the learning process, including convolutional neural network architecture, feature engineering, dense reward functions and backplay, etc. We evaluated the effectiveness of techniques by observing the change of expected rewards over episodes. With all these techniques, the agent manages to achieve a basic level of intelligence. However, we also observed that reinforcement learning is a delicate process that is harder to manage than the traditional supervised learning environment. There are still many behaviors that we hoped the agent to to learn but is not achieved yet so far. We look forward to keep iterating on our algorithms to build the invincible Pommerman agent.

Reference

- N-Person Minimax and Alpha-Beta Pruning: extend minimax to multi-agents scenario, and use alpha-beta pruning for efficient searching.
- Depth-limited search: reduce search space with evaluation function.
- Proximal Policy Optimization Algorithms (PPO): Efficient policy gradient updates.
- Opponent Modeling in Deep Reinforcement Learning: Learn strategy patterns of opponents through encoding observations of opponents into a deep Q-Network (DQN).
- Backplay: "Man muss immer umkehren": Backplay for efficient sampling.
- Pommerman: A Multi-Agent Playground: Introduction of the Pommerman competition.
- A Hybrid Search Agent in Pommerman: A sophisticated heuristic-based agent.