"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

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.

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*[1] 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 $\theta$, 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: $\hat{A}_t = Q_\pi(s_t;\ \theta)\ - \hat{V}(s_t)$.

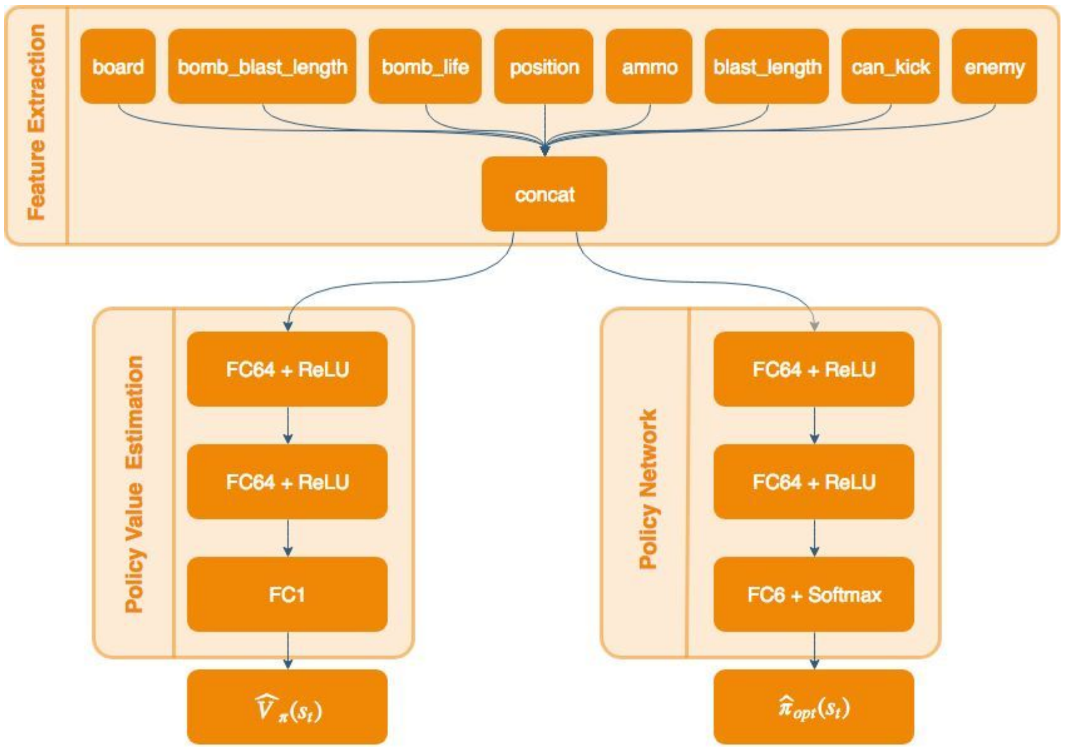
For each batch of episodes, we will try to optimize a **policy network** $\hat{\pi}_{opt}\,(s;\ \theta)$, which is modeled as a deep neural network to maximize the following objective:

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

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.

[1] More details and the rationale of choosing this algorithm is elaborated in the appendix.

Preliminary Mode Architecture



The diagram above illustrates our preliminary 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
Apply various feature extractors on the state s_t and form an embedding vector. Note that the embedding is shared between the Expected Reward and PPO model.

Policy Value Estimation
Estimate 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.

At the moment, the model is a 3-layer neural network. The first two layers are fully connected layers with 64 output units, followed by ReLU layers. The last layer is a linear combination of the outputs from the second layer.

In the future, we hope to experiment with different configuration of the neural network. For example, adding more depth and width to the model. We also hope to try different building blocks for the model, for example, applying convolutional layers on the board to extract spatial information more efficiently.

Policy Network
Compute the estimated optimal policy. The output of this module is a vector of length 6 representing the probabilistic distribution of the 6 actions.

The model we are experimenting with is a 3-layer neural network. The first two layers are fully connected layers with 64 output units, followed by ReLU layers. The last layer is a fully connected layers with 6 units followed by a softmax to get the normalized probability distribution of the 6 available actions. In the training stage, we will sample an action from the distribution, while in the inference stage, we choose the action with maximum probability.

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.

Baseline
simple_agent provided by the starter code. It is the official baseline model for the competition. The baseline agent's performance measured by intrinsic metric is **-0.067**.（It is negative because sometimes it kills itself quickly.)

Preliminary Model
At the time of submission, the model has been trained for over 50K iterations. In training stage, the model was trying to compete against 3 baseline agents. Below are some observations we found for the model:

In the first a few thousand iterations, the agent was barely able to survive until the end. Most often, the agent killed itself within 100 timestamps. Starting from the 8K steps, the agent started to learn some basics of dodging the bomb and keep it survive for a little longer. At around 40K iterations, the agent started to be able to beat the baseline agents in every a few games. The intrinsic metric measured so far is **+0.026**, and we expect the performance to continue to improve by a fair margin with more training steps and model architecture refinement.

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).

Appendix

Literature Review

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 four different two 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.

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

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.