Pommerman Fight: A Detailed Analysis of Reinforcement Algorithms

Bin Chen **Jiarong Qiu** **Sairam Kamal Raj** **Shuwei Shi** **Wei Cheng**

University of Southern California

Abstract

Reinforcement learning is an area of machine learning that deals with software agents tending to take actions in the environment with aim to maximize the notion of cumulative reward by using state value and reward functions. It is vastly used in developing robots for manufacturing, inventory management, delivery management (to name a few). On the other hand, its abilities can also be vastly used in development of games like Pommerman, Atari, Go and Robocup.

Our research interest branches from the intention to understand the power of various Reinforcement Learning algorithms and its performances in multi-agent environment. Pommerman, a multi-agent environment based on the classic console game Bomberman provided the benchmark for our project.

Why Pommerman Fight

There are several remarkable achievements of agents learning to play games against humans and defeating them. Google's Agent57 can beat the average human at 57 classic Atari games, AlhaGo recorded a tremendous achievement by defeating Lee Sedol, a world champion in Go. Robocup is an international scientific initiative to advance the state of the art intelligent robots. However, these games are based on consensus benchmark involving two player zero sum game and there are sparsely any environment that utilizes general sum game settings with more than 2 players.

Pommerman is a multi-agent mode;-free game which typically involves 4 agents and contains both cooperative and competitive aspects and sets an epitome example for general sum game.

Few of the key notable differences between Robocup (a multi-agent two player zero sum game) and Pommerman are:

1. Pommerman includes an explicit communication channel. This changes the dynamics of the game and adds new research avenues.
2. Pommerman strips away the sensor input, which means that the game is less apt for robotics but more apt for studying other aspects of AI, games, and strategy.
3. Pommerman uses low dimensional, discrete control and input representations instead of continuous inputs. We believe this makes it easier to focus on the high-level strategic aspects rather than low level mechanics.
4. In team variants, the default Pommerman setup has only two agents per side, which makes it more amenable to burgeoning fields like emergent communication which encounter training difficulties with larger numbers of agents.
5. Pommerman promotes research that does not reduce to a 1v1 game, which means that a lot of the theory underlying such games (like RoboCup Soccer) is not applicable.

While on one hand there are a lot of researches focusing on the diverse set of tools and methods including planning, opponent/teammate modeling, game theory and communication aimed at advanced research on building state of the art technologies contributing to lasting results for years to come, our team focused on analyzing the performance of various RL algorithms in this model-free game.

There are attributes that are common to the best benchmarks beyond satisfying the community's research direction. These include having mechanics and gameplay that are intuitive for humans, being fun to play and watch, being easy to integrate into common research setups, and having a learning problem that is not too difficult for the current state of method development. Most games violate at least one of these. For example, the popular game Defense of the Ancients (OpenAI 2018) is intuitive and fun, but extremely difficult to integrate. On the other hand, the card game Bridge is easy to integrate, but it is not intuitive; the gameplay and mechanics are slow to learn and there is a steep learning curve to understanding strategy.

Multi-Agent Learning

Pommerman is very challenging model-free environment for reinforcement learning due to its sparse, delayed and deceptive rewards and it has shown that RL algorithms fail to achieve significant learning.

The game environment in pommerman changes by bombs placed by agents whose effects are observed when it explodes i.e; after 10 time steps. However, it has been noted that the action of placing bomb is highly correlated to losing and is presumably a major impediment for achieving good results using model-free reinforcement learning and could lead to catastrophic suicides when an agent intersects with an exploding bomb's flames, previously placed by itself. It has been seen that for each of the four agents, even when their configuration is different, their probabilities of ending up with suicide are ≈ 40% (0.39, 0.38, 0.46, 0.38,

Suicides can occur frequently in pommerman during learning phase because agents use intensive search to fully learn policies. Hence, it is essential to filter out actions leading to immediate negative terminal states so that model-free RL can imitate those safer actions to learn to safely explore.

According to the research paper Safer Deep RL with Shallow MCTS published by borealis AI, Pommerman is a challenging benchmark for multi-agent learning and model-free reinforcement learning, due to the following characteristics:

1. Multiagent component: the agent needs to best respond to any type of opponent, but agents' behaviors also change based on the collected power-ups, i.e., extra ammo, bomb blast radius, and bomb kick ability.
2. Delayed action effects: the only way to make a change to the environment (e.g., kill an agent) is by means of bomb placement, but the effect of such an action is only observed when the bombs explodes after 10 timesteps.
3. Sparse and deceptive rewards: the former refers to the fact that the only non-zero reward is obtained at the end of an episode. The latter refers to the fact that quite often a winning

reward is due to the opponents' involuntary suicide, which makes reinforcing an agent's action based on such a reward deceptive.

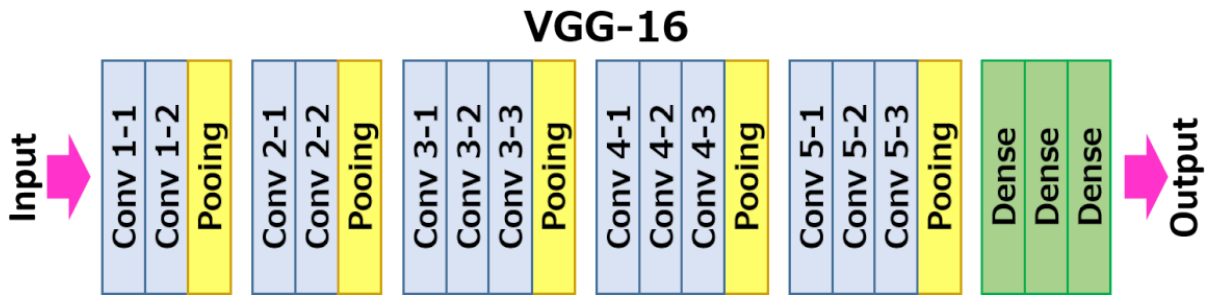
For these reasons, this game is considered to be challenging for many standard RL algorithms and a local optimum is commonly learned, i.e., not placing bombs or stay put.

Indeed, the problem of suicide stems from acting randomly without considering constraints of the environment and in extreme cases, in each time step, the agent may have only one survival action.

Prior work in this area

Prior to working on our project we researched several Github profiles and determined the tools required to efficiently train our agents. The following tools were the important amongst all available tools:

VGG



VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition ". The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. It was one of the famous models submitted to ILSVRC-2014. It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using NVIDIA Titan Black GPU's.

Tensorflow

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library and is also used for machine learning applications such as neural networks. It is used for both research and production at Google.

TensorFlow was developed by the Google Brain team for their internal use. It was released under the Apache License 2.0 on November 9, 2015.

Pytorch

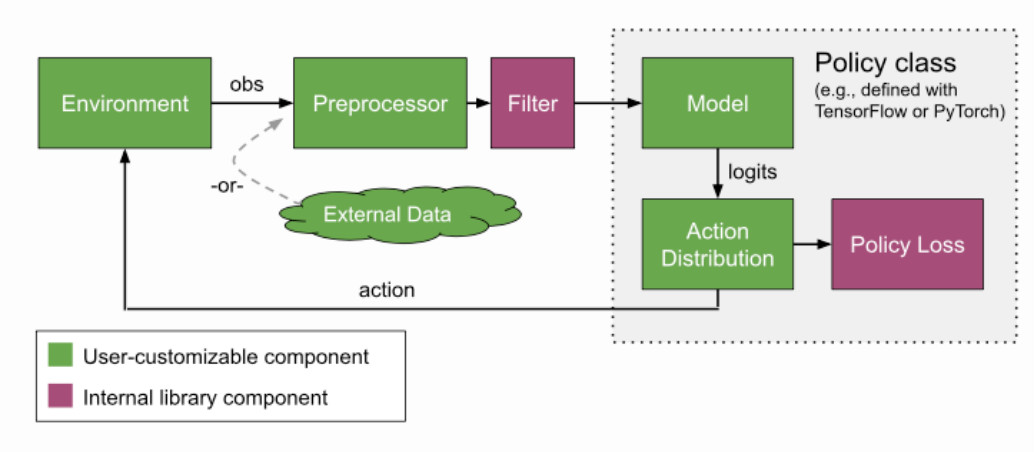
PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing. It was primarily developed by Facebook's AI Research lab (FAIR).It is a free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.

A number of pieces of Deep Learning software are built on top of PyTorch, including Uber's Pyro, HuggingFace's Transformers, and Catalyst.

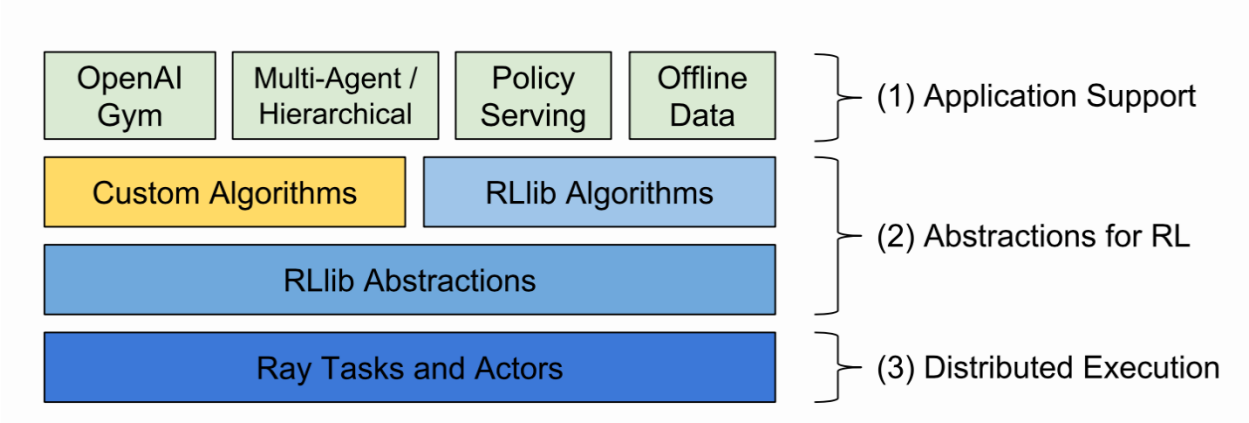
PyTorch provides two high-level features:

- Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU)
- Deep neural networks built on a tape-based autodiff system

RLlib



Basically, RLlib builds up a keras like api that only need us to provide a custom environment and model without worrying about the details in the RL optimization process. Also, they adopt the setting of OpenAI Gym, a widely used interface for building standard RL environments. During the execution, they allow the distributed ways to sample batch and tune hyper-parameters of algorithms based on the Ray library. In brief, we choose RLlib for it brings us the convenience in using RL algorithms to train an agent for Pommerman.



Game Layout

Bomberman is an arcade-style maze-based video game developed by Nintendo Entertainment System on 1985, it contains an agent known as Bomberman who navigates a grid like maze and places bombs to destroy enemies and surrounding wooden walls to obtain power-ups and access to the next stage through a door. Pommerman is stylistically similar to Bomberman and at a higher level it has 4 agents traversing a grid environment with a goal of their team being the last one standing on the board.

Key notable characters of the game environment are:

Agent – Navigates through the grid environment and communicates with the other agents either in an adversarial and cooperative fashion.

Bombs – placed by the agents and upon explosion can destroy anything in their vicinity except the rigid walls.

Rigid Walls – Indestructible and impassable walls that form both the surroundings of the game and might be present in the grid such that the agents cannot pass through these walls but should instead navigate for other passable routes.

Wooden Walls – Walls that are initially impassable but can be destroyed using bombs and some of them acts as containers for power-ups.

In Pommerman, each agent can execute one of 6 actions at every timestep:

1. Stop: This action is a pass.
2. Up: Move up on the board.
3. Left: Move left on the board.
4. Down: Move down on the board.
5. Right: Move right on the board.
6. Bomb: Lay a bomb (Note: This is considered a negative reward action).

Every battle starts on a randomly drawn symmetric 11x11 grid ('board') with four agents, one in each corner. In team variants, teammates start on opposite corners and the game ends when at most one agent remains alive or both players on one team have been destroyed and the winning team is the one who has remaining members.

If there is a communicative scenario, then the agent emits a message every turn consisting of two words from a dictionary of size eight. These words are passed to its teammate in the next step as part of the observation. In total, the agent receives the following observation each turn:

1. Board: 121 Ints. The flattened board. In partially observed variants, all squares outside of the 5x5 purview around the agent's position will be covered with the value for fog (5).
2. Position: 2 Ints, each in [0, 10]. The agent's (x, y) position in the grid.
3. Ammo: 1 Int. The agent's current ammo.
4. Blast Strength: 1 Int. The agent's current blast strength.
5. Can Kick: 1 Int, 0 or 1. Whether the agent can kick or not.
6. Teammate: 1 Int in [-1, 3]. Which agent is this agent's teammate. In non-team variants, this is -1.
7. Enemies: 3 Ints in [-1, 3]. Which agents are this agent's enemies. In team variants, the third int is -1.
8. Bomb Blast Strength: List of Ints. The bomb blast strengths for each of the bombs in the agent's purview.
9. Bomb Life: List of Ints. The remaining life for each of the bombs in the agent's purview.
10. Message: 2 Ints in [0, 8]. The message being relayed from the teammate. Both values are zero only when a teammate is dead or if it is the first step. This field is not included for non-cheap talk variants.

Game Play: The agent starts with one bomb ('ammo'). Every time it lays a bomb, its ammo decreases by one. Bombs placed explodes after 10 timesteps and produces flames that have a lifetime of 2 timesteps and destroys the anything in its blast radius except the rigid walls, and post the explosion agents ammo will increase by one. The agent also has a blast strength that starts at two. Every bomb it lays is imbued with the current blast strength, which is how far in the vertical and horizontal directions that bomb will affect. Bombs destroyed in this manner chain their explosions. A single game finishes when an agent dies or reaches 800 timesteps.

Power-Ups: Half of the wooden walls have hidden powerups that are revealed when the wall is destroyed. These are:

1. Extra Bomb: It increases the agent's ammo by one.
2. Increase Range: This increases the agent's blast strength by one.
3. Can Kick: Permanently allows an agent to kick bombs by moving into them. The bombs travel in the direction that the agent was moving at one unit per time step until they are impeded either by a player, a bomb, or a wall.

Reward Shape:

Initially we trained agents with default reward shape as per the pommerman playground:

1. +1: Win.
2. -1: Lost.

In this scenario it was observed that the agent being trained tend to remain safe always and executed stay put action on all steps. Hence, we made certain changes to the reward values and trained the agents to obtain smart agents that move during the game but do not plant bombs and agents that moved and also planted bombs. Reward values that resulted in the training of agents were:

1. no penalty – if an agent plants a bomb and its explosion does not kill any enemies.
2. 0.02 pts – if agent explores previously unexplored positions.
3. 0.05 pts – if the agent picks up power-up that increases bomb's blast strength.
4. 0.05 pts - if the agent picks up power-up that increases its number of ammos.
5. 0.5*number of woods – if an agent plants a bomb and its explosion destroys wooden walls.
6. 0.5/100 near the enemy - if agent moves towards its enemies.
7. 0.5*impact_of_bomb_blast based on bomb strength – If agent places a bomb and it blasts.
8. 0.5 pts – if an agent plants a bomb and kills 1 enemy.
9. 1.0 pts - if an agent plants a bomb and kills 2 enemies.
10. 1.5 pts - if an agent plants a bomb and kills 3 enemies.

Further, we also customized the winning and loosing reward shapes for an agent to make its learning more efficient by providing the following:

1. +200: Win.

2. -200: Lost.

Technologies Leveraged:

Software and Libraries used:

1. PyTorch – A free open source machine learning library initially developed for Facebook's AI and used for scalable distributed training and performance optimization in our research.
2. RLlib – Open Source library used for reinforcement learning offering abstractions for highly scalable and unified API for variety of applications like TensorFlow and PyTorch
3. Gym – Toolkit for developing and comparing RL algorithms. Supports in training agents to play Pommerman.
4. Github – Platform enabling our community of developers to discover, share and build Pommerman games and train the agents.
5. Ray – High performance distributed execution framework used for large scale machine learning and reinforcement learning applications like Pommerman.
6. Pip – Standard software management system used to install and manage software packages written in Python.
7. Conda – Open source cross-platform language-agnostic package manager and environment management system used to create virtual environments and install packages for Pommerman.
8. Numpy – Python library used for large multi-dimensional arrays and matrices along with collection of high level mathematical funcions to operate and performs computations in Pommerman's grid like environment.
9. TensorFlow – End-to-End open source machine learning platform used to develop and train ML models using tensors.
10. Keras – Neural Network library which is open source and capable of running on top of TensorFlow and designed to enable fast experimentations with deep neural networks.
11. Convolutional Neural Networks – Regularized versions of multilayer perceptrons used to form fully connected networks i.e; each neuron in a layer is connected to all neurons in the next layer.
12. Logging – Used to track events when pommerman runs.

Reinforcement Learning Algorithms:

Dynamic Environment, Partial Observability and self dependence on current policy due to performing actions based on agents interaction with the environment makes reinforcement learning a challenge.

Hence, Data distributions over observations and rewards are constantly changing resulting in major instability during the training process

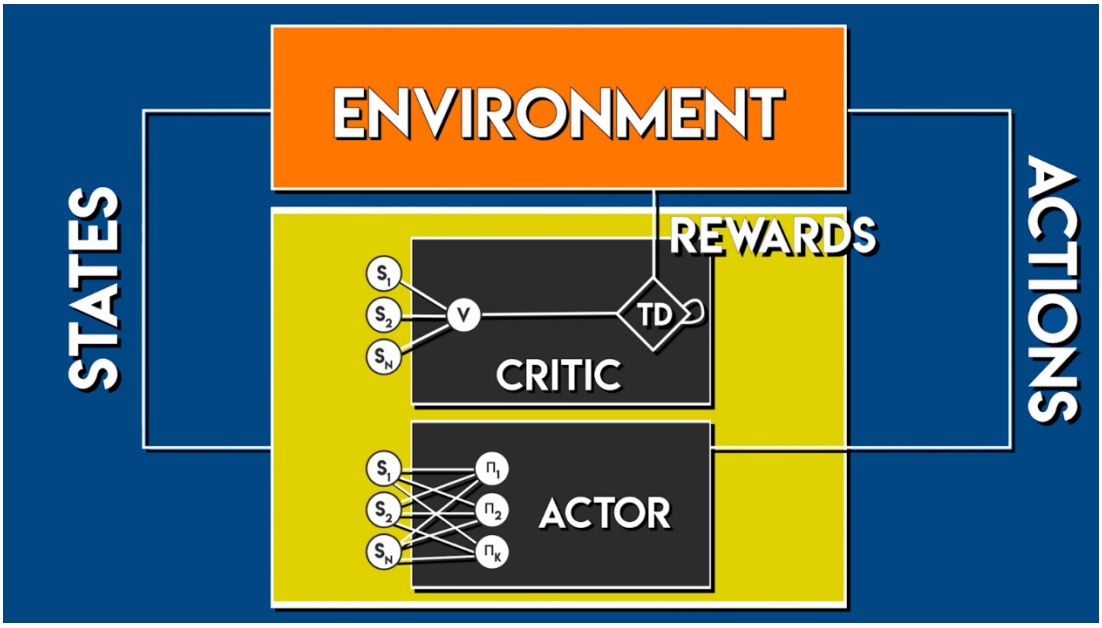
A2C (Advantage Actor Critic Model)

Similar to Deep Q-Network i.e; finite input and output space, we implemented A2C (Actor - Critic) algorithm to train our agent in this multi-agent game. In this algorithm, the actor takes in the current environment state and determines the best action to take i.e; actor is the policy being optimized,

the critic plays the evaluation role by taking in the environment state and action performed by the actor and returns the action score i.e; critic is the value function which is being learned.

Analogy: Much like how a child playing in a park can perform multiple actions and it's parent can either compliment or critic the child's actions, our agent initially performs any randomly chosen action from the list of actions and the critic part of the algorithm evaluates the reward score for each action and hence helps in determining the best possible action.

Then, by deploying chain rule, the agent eventually learns to perform the right actions based on the feedback obtained from the critics and updates the policy distribution in the direction suggested by the critic.



In A2C: the reward functions are calculated using the formula:

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

[where; s: state of the environment, a: action taken by the agent, pie: expected value obtained by performing an action a on state s at time t and A is advantage function]

Method to avoid local maximum: Reward obtained by performing an action a in state s at time t is reduced by a baseline factor using Advantage Function A, which is computed as:

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

[where; s: state of the environment, r: reward earned till now based on the policy generated, gamma: discounted reward and Vv is the average of Q-Values obtained for all actions in the given state at time t]

Advantages:

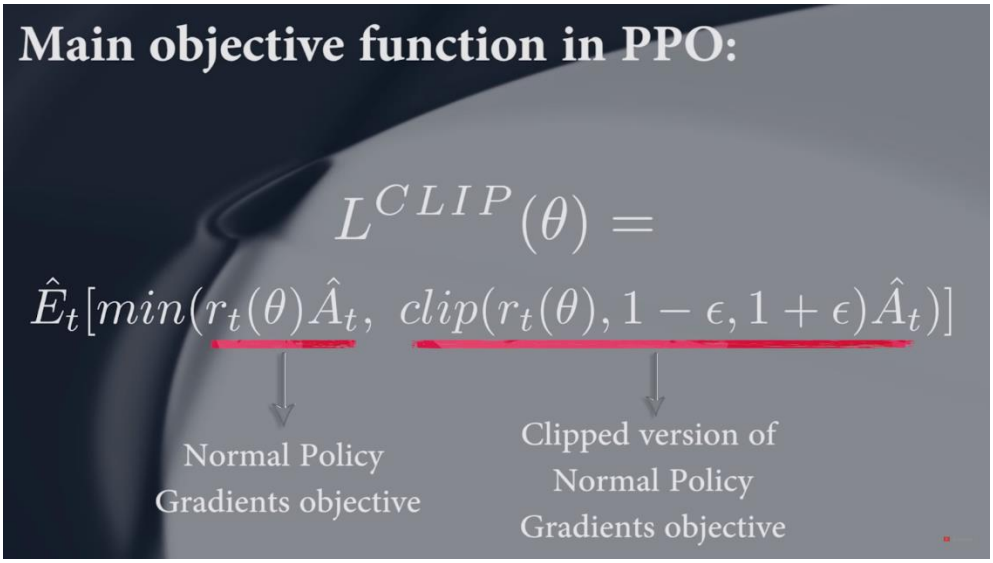
1. Actor Critic models tend to require much less training time than policy gradient methods.

PPO (Proximal Policy Optimization)

Developed at OpenAI, based on the principles of policy optimization and value based functions of A2C framework and "trusted region" of TRPO algorithm, PPO algorithm is used to overcome the disadvantages of vanilla policy gradient methods. It is also easier to implement, samples efficiencies and easy to tune.

PPO utilizes fixed-length trajectory segments and during each iteration N parallel actors, collects T timesteps of data and constructs a surrogate loss optimized with mini batch SGD or Adam optimizers for K epochs. Further, based on the rewards obtained from the N actors, an advantage function is calculated (thus involves actual computation and no guessing) and overcomes noisy advantage function values.

Main objective function of PPO is calculated as:



[where; E: Expectation objective (calculated using batches of trajectories, r: reward obtained by following the policy, A: Advantage function, epsilon: clipping factor (usually 0.2 to clip the reward function between 0.8*r and 1.2*r]

Advantage Function:

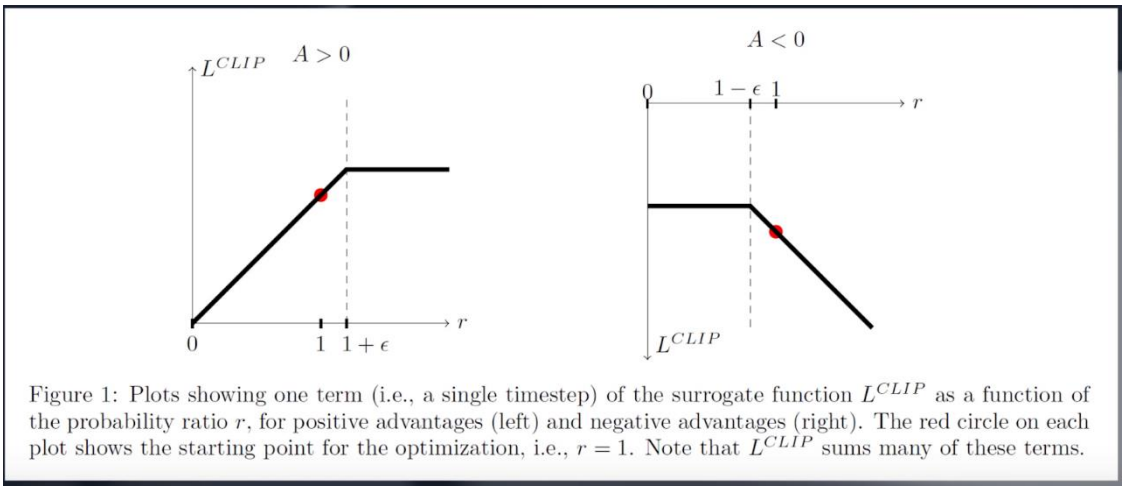
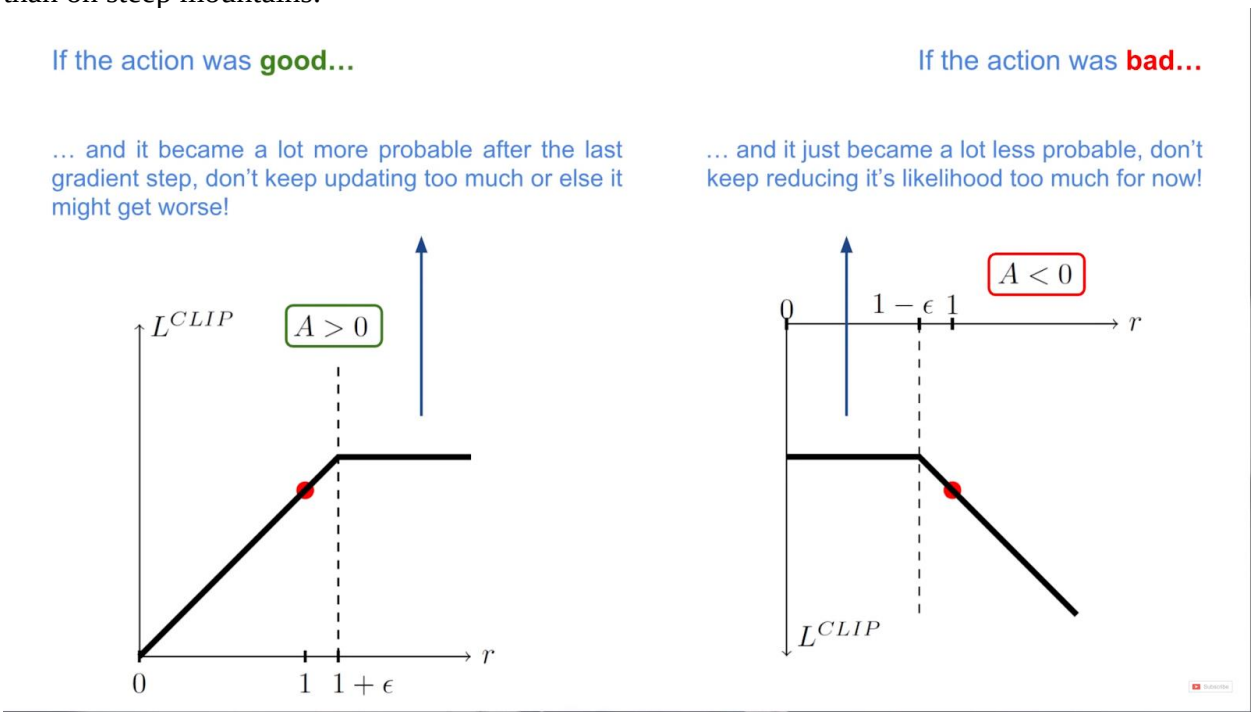


Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{CLIP} sums many of these terms.

The advantage function might be positive for all the cases when our selected action has better outcome than the expected return or negative in all other cases.

Analogy:

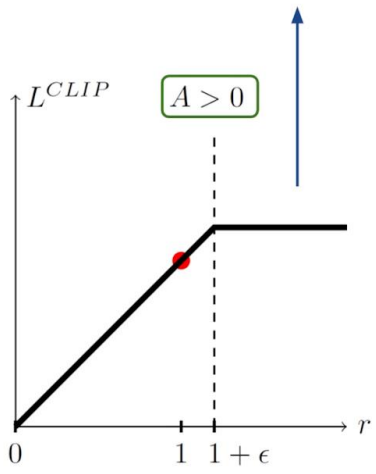
The Advantage function helps in sampling efficiencies and leads to larger or frequent hops on flat surface and small hops on steep surface. Similar to how hopping or jumping on flat terrain is easier than on steep mountains.



Condition for termination of program:

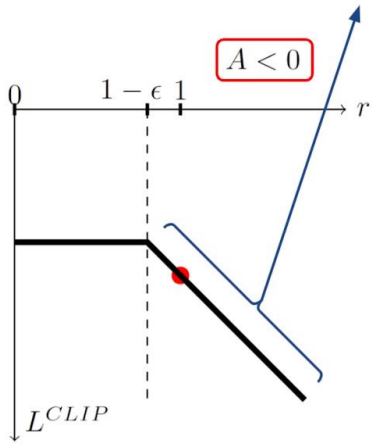
If the action was **good…**

… and it became a lot more probable after the last gradient step, don't keep updating too much or else it might get worse!



If the action was **bad…**

… and it just became a lot **more** probable, we would really want to undo our last update!



Advantages:

1. Requires less space - In contrast to Deep-Q Network, agents in PPO do not learn from past experiences saved on replay buffers but instead it learns directly from the environment.
2. Sample Efficiency - PPO algorithm can sample the efficiencies at each stage and if the advantage function does not lead to large changes in the calculated rewards then it can perform multiple or long hoppings i.e; on flatter surface and when there are large changes i.e; when the surface is steep then it can automatically perform small hops and hence attempts to the reach the optimal value.

APPO (Accelerated Proximal Policy Optimization):

An algorithm formed on the basis of PPO and Nesterov's Accelerated Gradient (NAG) is an algorithm still under research and utilized to speed up training process using momentum. It uses the concept of training rounds i.e; series of updates to our policy network after collecting a certain number of experiences. For Example; we can define 1 training round to be 4 epochs after playing 2 episodes of the game.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

[where; vt: Exponentially Weighted Average at timestep t, gamma: decay factor, eta: learning rate, θ: weights]

Disadvantages of PPO:

1. It is a reactive algorithm and hence slows the training process. We decided to experiment with proactive algorithm to enable early learning.

In PPO, Clipping only takes effect after the policy gets pushed outside of the range (it also depends on the sign of advantage). As such, it is a reactive algorithm because it only restricts movement once the policy has already moved outside of the specified bounds. This means that PPO does not ensure the new policy is proximal to the old policy because rt(θ)rt(θ) can easily move well below 1−ϵ1−ϵ or above 1+ϵ1+ϵ. As we will see later in this post, by using the accelerated gradient concept behind NAG, we can design a proactive algorithm which anticipates how much the policy will move. We can then use the anticipated move to better control our policy and keep it roughly within the bounds.

2. Using a ratio to measure divergence from the old policy handicaps updates for low probability actions.

The bounds that we set for the policy in PPO is based off of a ratio, which I do not like. The denominator $\pi\theta old$ matters a lot because if it is too small, then learning is severely impaired. Let's use an example to show you what I mean. Imagine two scenarios:

1. Low probability action: $\pi\theta old(st,at)=2\%\pi\theta old(st,at)=2\%$
2. High probability action: $\pi\theta old(st,at)=70\%\pi\theta old(st,at)=70\%$

Now let's assume that $\epsilon=0.2\epsilon=0.2$. This means that we restrict our new policy to be $1.6\%\leq\pi\theta\leq2.4\%1.6\%\leq\pi\theta\leq2.4\%$ for the first scenario and $56\%\leq\pi\theta\leq84\%56\%\leq\pi\theta\leq84\%$ for the second scenario. Under the first scenario the policy can only move within a range that is 0.8% wide, while in the second scenario the policy can move within a range that is 28% wide. If the low probability action should actually have a high probability, then it will take forever to get it to where it should be. However, if we use an absolute difference, then the range in which the new policy can move is the exact same regardless of how small or large the probability of taking an action under our old policy is.

After calculating $\pi\theta pred$, we can see if it moves outside of the bounds $\pi\theta old(at|st)\pm\epsilon$. If so, then we can use Mean Squared Error (MSE) as the loss function for those samples and move $\pi\theta$ towards the bound that it crossed. On the other hand, if it is within the range, then you can update $\pi\theta$ with the regular policy gradient method.

It can be noticed that by using the method above, an if statement splits the mini-batch into two smaller batches: one to be trained with MSE and the other to be trained with a policy gradient loss. If you don't want to split up your mini-batch with an if statement during training, then you can update the whole mini-batch with the following loss function:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left(\pi_{\theta}^{(i)} - \text{clip}(\pi_{\theta_{\text{pred}}}^{(i)}, \pi_{\theta_{\text{old}}}^{(i)} - \epsilon, \pi_{\theta_{\text{old}}}^{(i)} + \epsilon)\right)^2$$

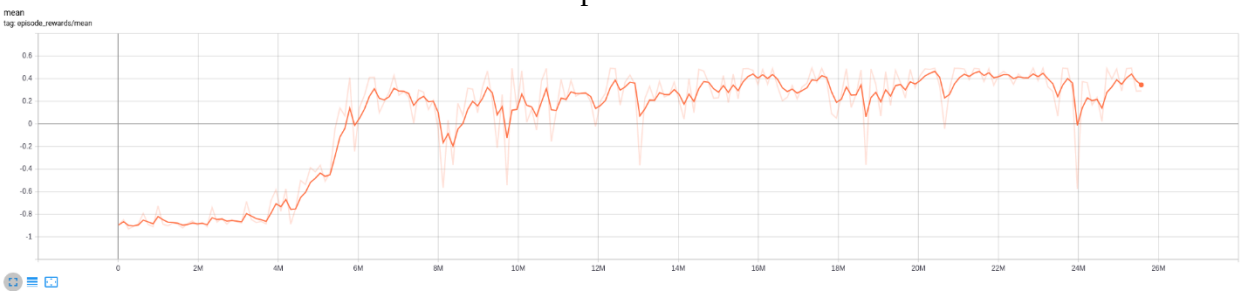
Randomized Agent:

General Deep RL algorithm that performs exploration through a stochastic policy. Actions are randomly sampled from the action likelihood distribution provided by the neural network resulting in apparently random selection of actions.
Though a naïve method to train an agent, we attempted this method to understand how agent would behave and it was observed that in most of the cases the agent executes stay put action and does not perform any action or it randomly moves across the board without placing any bombs.

It was extremely difficult to train the agents using this method because of the large batch size and epochs required and it did not result in sufficient performance and hence we explored other RL algorithms.

Results and Analysis

Based on the research and performance analysis of the A2C, PPO and APPO algorithms, we observed that for A2C algorithm the mean episode reward begins at -0.9 (approximately) and does not increase until 3.7 million episodes but after that it makes a steep increase to approximately 0.3 and then makes a gradual increase to upto 0.4 for further episodes. It can be further observed that the curve tends to remain flat till 26 million episodes.



Comparing several other reward and loss functions we observe the below graph for A2C algorithm with agents that do not place bomb:

While the agent learns to place the bomb the above reward and loss values can be found to have the following changes:

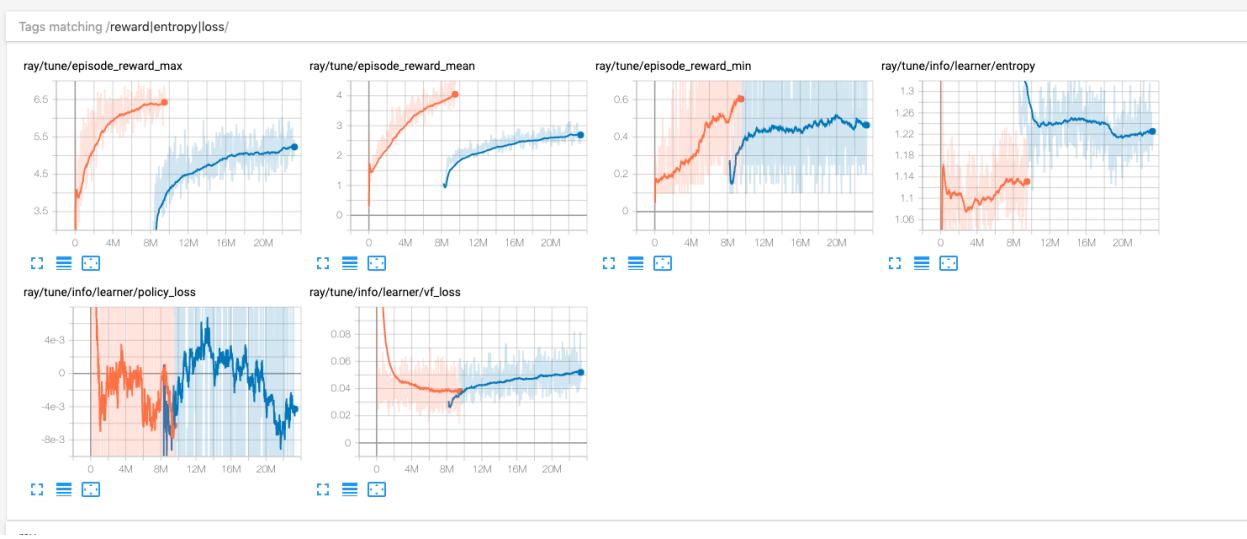
Few key points worth noticing are:

1. The maximum episode for agents that do not place bomb begins at -0.75 and increases to around 0.4 in 4 million episodes and tends to gradually increase later and remain flat at 0.6 (approximately). However, for agents that learnt to place bombs the maximum episode reward even though starts at a much higher value of around -0.5 and further decreases to around -0.75 it quickly increases to a higher value of 2.5 (approximately). This might be the result because the action of placing a bomb is considered as negative loss.
2. Similarly, it can also be observed that the episode mean rewards and episode minimum rewards are also comparatively higher with values greater than or equal to 1.0 approximately for agents that can place bombs and only about 0.4 for agents that cannot place bombs even though the latter agents begin with slightly higher mean and minimum rewards. This concludes that agents which can place bomb have higher performance and learnt better.
3. Action Loss, Dist_Entropy loss and Value loss for agents that place bomb are either flat across several million episodes or drops whereas if agents do not place bomb then it can be observed that the loss increases in few early episodes like in 4 millionth episode the action loss and value loss increases rather than decreasing.

Comparison of PPO algorithm's performance was performed with 2 different tasks:

Task2: Agents move around the pommerman grid but does not place bombs represented by orange line.

Task3: Agents move around the pommerman grid and places bombs represented by blue line.



Further, the results have few key aspects to be observed:

1. It can be observed that the rewards for task 2 and task 3 increases at comparatively same rate. However, the rewards for task 2 are higher than task3.
2. The minimum rewards of task3 increases significantly in the beginning but later becomes flat with approximate value of 0.5.
3. Entropy, Policy and VF losses for task 2 begins much earlier than task3.
4. Entropy loss for task2 is much greater than the entropy loss for task3.
5. Policy loss for task3 is tends to increase higher than for task2.
6. Vf_loss for task3 seems to increase slightly from 8 million episodes to 20 million episodes but it constantly decreases for task2.

Limitations

Preliminary analysis shows that that the pommerman environment is very challenging for reinforcement algorithms and we do not yet know the optimal solutions in any of the variants. Difficulties arise due to several factors like:

1. Agents in pommmerman can reach a local optimum and avoids exploding itself by learning to never use the bomb action. In the long term, this is ineffective because the agent needs to use the bomb to destroy other agents.
2. Without a very large batch size and a shaped reward, neither of Actor-Critic Model nor Proximal Policy Optimization (Schulman et al. 2017) learned to successfully play the game against the default learning agent (‘SimpleAgent’).

Conclusions

Agents that place bomb has rewards that always increases and performs much better than those that cannot place bombs even though their initial values were lower. Lower initial values can be attributed to the action of placing bomb being a negative loss in line with pommerman research papers.

Performance of PPO and APPO algorithms is found to be better than A2C with the rewards for the former algorithm reaching over 6 whereas for the latter reaches only to about 2.5. Hence, it can be concluded that PPO is a better algorithm to train pommerman agents.

We faced stringent resource limitation due to the current COVID-19 situation and hence could not use our game pipe lab systems or super computers provided to us thus further limiting the experiments and researches on the algorithms that can be used for our research. However, we are thankful to professor Mike Zyda and the teaching members of the course for the tremendous help and support provided.

Future Work

Research game competitions disappear for two reasons - either the administrators stop running it or participants stop submitting entrants. This can be due to the game being 'solved', but it could also be because the game just was not enjoyable or accessible enough. On the contrary according to the research paper "Pommerman: A Multi-Agent Playground" pommerman is highly intuitive, easy to integrate and fun to watch and stated as the most intuitive game played by AI agents and they strongly feel that research in pommerman has a long life ahead.

Ratings provided for pommerman in this paper are as follows:

Game	Intuitive?	Fun?	Integration?
Bridge	1	3	5
Civilization	2	3	1
Counterstrike	5	5	2
Coup	4	5	5
Diplomacy	1	4	3
DoTA	3	5	2
Hanabi	2	3	5
Hearthstone	1	4	1
Mario Maker	4	5	3
Pommerman	**5**	**4**	**5**
PUBG	5	5	1
Rocket League	5	4	1
Secret Hitler	4	4	3
Settlers of Catan	4	3	3
Starcraft 2	3	5	5
Super Smash	5	5	1

Beyond the surface hyperparameters like board size and number of walls, early forays suggest that there are many aspects of the game that can be modified to create a rich and long-lasting research challenge. These include partial observability of the board, playing with random teammates, communication among the agents, adding power-ups, and learning to play with human players.

Further, imitation learning can be combined with Monte Carlo Tree Search and Deep Reinforcement Learning to train the agents and according to the paper: "Safe Deep RL with Shallow MCTS" by Borealis AI, this can use A3C as a baseline algorithm. Hence, we plan to use these techniques to further train our agent and analyze the performance. This has the effect of providing action guidance, which in turns improves the training efficiency.

Further we also plan to combine supervised learning loss with the Q-learning loss within the DQN algorithm to pre-train and determine the performance of our pommerman agent. showed that their method achieves good results on Atari games by using a few minutes of game-play data where a pre-trained network can focus more on policy learning using DQN.

References

Technologies Documentations:

1. Gym – A Toolkit for developing and comparing RL algorithms.
2. Gym Retro – A platform for Reinforcement Learning research on Games.
3. MAME RL – Python library used to train reinforcement learning algorithm on almost any arcade game.
4. PyTorch – An open source machine learning framework that accelerates the path from research prototyping to production deployment.
5. TensorFlow – An end-to-end open source machine learning library used to develop and train models.
6. Ray – A fast and simple framework for building and running distributed applications.

Algorithms reference documents:

1. Understanding Actor Critic Methods and A2C (Chris Yoon).
2. Conceptual differences A2C and PPO (u/Delthc).
3. Actor Critic Algorithms (Siraj Raval).
4. Proximal Gradient Methods and Proximal Policy Optimizations (PPO): diving into Deep RL (Arxiv Insights).
5. An Introduction to Proximal Policy Optimization (PPO) in Deep Reinforcement Learning (Udacity-Deep RL).
6. Accelerated Proximal Policy Optimization (Brandon Da Silva).

Github repositories referenced:

1. Pommerman Playground (fschlatt and cinjon).
2. MultiAgentLearning Playground (PBarde and cinjon).
3. Pytorch Pommerman RL (rwightman).
4. Bomberman deep reinforcement learning challenge in PyTorch (Kazyka Eugene).
5. Pommerman (Jameson Thai).

Research Papers referenced:

1. Multi-Agent Strategies for Pommerman (Dhruv Shah, Nihal Singh and Chinmay Talegaonkar).

2. Automated Gaming Pommerman: FFA (Ms. Navya Singh,Mr. Anshul Dhull, Mr. Barath Mohan.S, Mr. Bhavish Pahwa, Ms. Komal Sharma).
3. Real-time tree search with pessimistic scenarios (Takayuki Osogami, Toshihiro Takahashi) – IBM Research Tokyo.
4. Safer Deep RL with Shallow MCTS: A Case Study in Pommerman (Bilal Kartal∗ , Pablo Hernandez-Leal∗ , Chao Gao, and Matthew E. Taylor) – Borealis AI Canada.
5. Pommerman: A Multi-Agent Playground (Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, and Kyunghyun Cho and Joan Bruna).