Pommerman Fight

BIN CHEN, JIARONG QIU, SAIRAM KAMAL RAJ, SHUWEI SHI, WEI CHENG
{chenbin, jiarongq, kamalraj, shuweish, wcheng49}@usc.edu

1) Goal of our project

Our goal is to train an agent on the Pommerman Environment, and compete with other three simple agents. The actions taken by our agent have to make sense. Possibly, we could lead to develop some different strategies based on different reward functions.

2) Background and overview of the game

Pommerman Description

In the Free For All(FFA) mode, every agent should explore the map and defeat other agents using the bomb. Initially, everyone is born at a corner of the map with the ability to place bombs that explodes in an area of 3x3 cross. And, it needs to destroy the wooden blocks to create the paths to other corners and gain buff for extra bombs and more powerful bombs, thus, it can gain more power to beat other agents. The last one survived is the winner.

RL Problem Formulation

Observations

1. alive: [10,11,12,13], correspond to the agent0 - agent3
2. board: numpy array uint8, 11x11, skip fog which is used in 2v2
    a. Passage 0
    b. Rigid Wall 1
    c. Wooden Wall 2
    d. Bomb 3
    e. Flame 4
    f. Extra Bomb 6
    g. Power Range 7
    h. Can kick 8
    i. Agent0-3, 10 - 13
3. bomb_blast_strength: 11x11 array. Everything outside agents view is fogged out.
4. bomb_life: 11x11.Everything outside agents' view is fogged out. Bomb trigger time
5. bomb_moving_direction: not mentioned ignored
6. Flame_life: time flame last
7. game_type:1, guess is the mode ignored
8. game_env:string name,pommerman.envs.v0:Pomme ignored
9. position: (y,x) tuple
10. blast_strength:int,2
11. can_kick: ignored
12. teammate: ignore, default is an instance of dummy agent9
13. ammo:1
14. enemies: instanced of other agents
15. step_count: may be the count of turns

Action

1. Stop 0
2. Up 1
3. Down 2
4. Left 3
5. Right 4
6. Bomb 5

Reward Shape

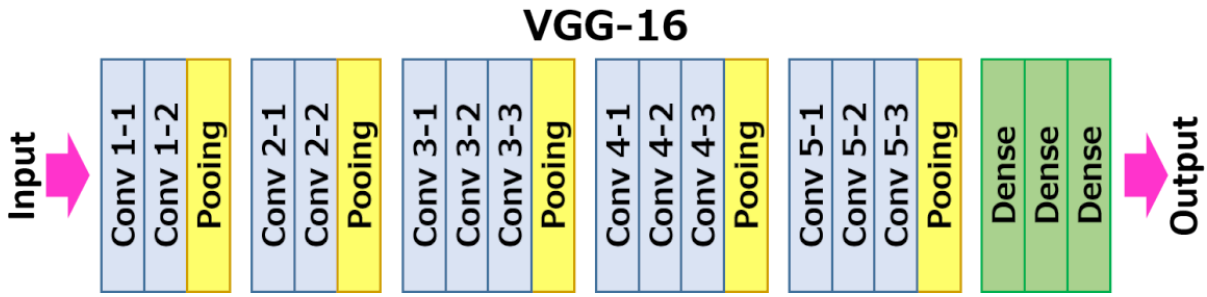
The default reward shape provided in the GitHub playground repo is

1. Win in the end +1
2. Lost in the end -1

3) Prior Research related to our game

Model

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.

Libraries

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 internal Google 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 is primarily developed

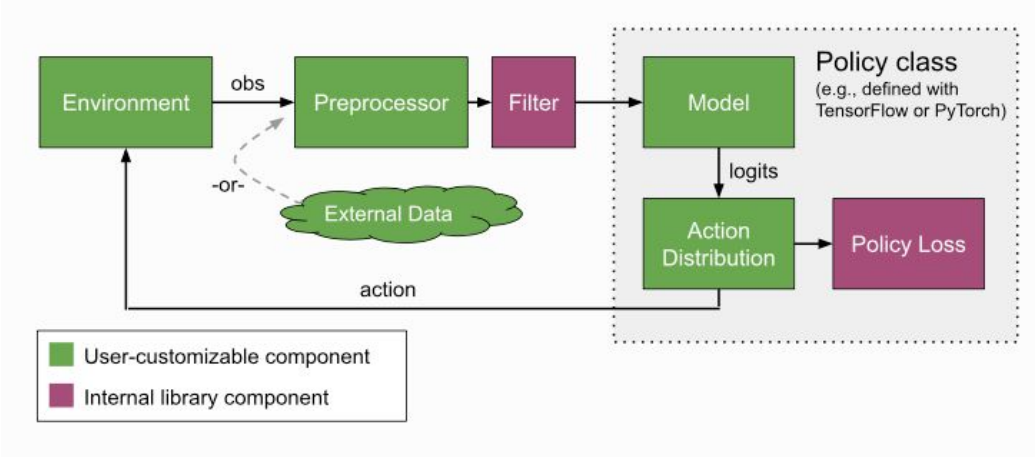
by Facebook's AI Research lab (FAIR).It is free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.

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

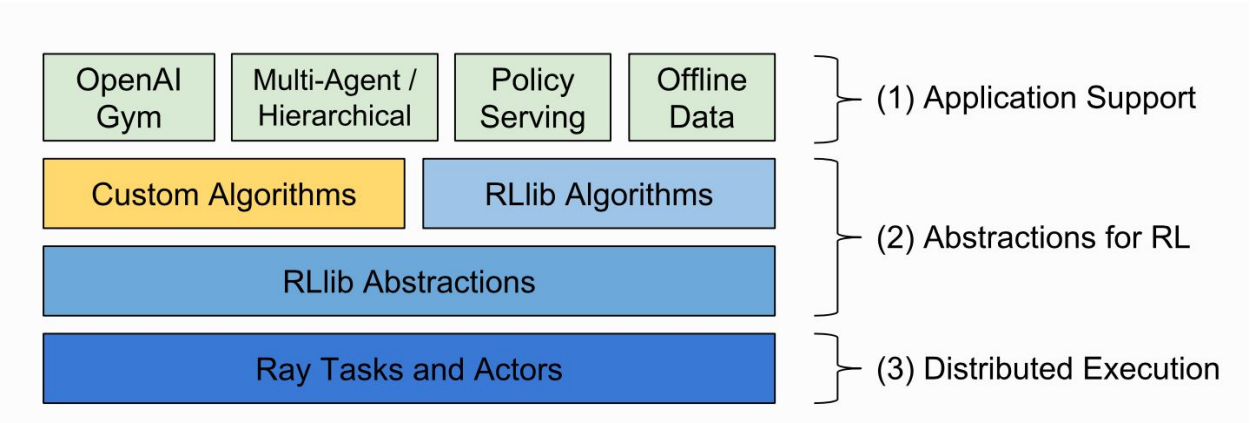
PyTorch provides two high-level features:

- 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.



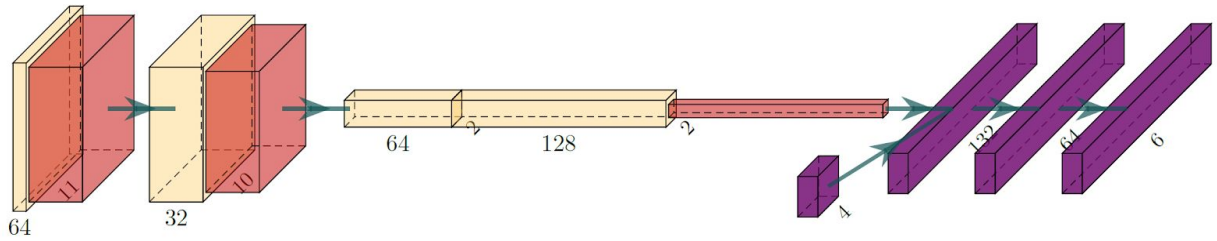
4) Methodology

For training an agent in a Deep Reinforcement Learning game, we should choose one of the algorithms to train the selected model(used to be a neural network in Deep RL) with the Pommerman environment. We use RLlib as our main framework of the whole project. We can directly choose the algorithms we need and build our own model and environment with RLlib dealing with the rest processes such as training the model, logger and multi-threads. Below are the algorithms, models and reward shapes we are using.

Model

By now, we used 3 different models for our agents. We tried a pure fully connected model, simple CNN model and CNN model derived from VGG16. Below are the details of these models.

1. Fully connected model:
   The first model we use is a fully connected model with 3 hidden layers of 256, 128, 32 cells. We use this model and A2C algorithm to train for 4 hours and get poor results with almost losing game all the time.
2. CNN with LSTM:
   For the A2C algorithm, agents' action at this time is much related to the previous action. So we use a neural network with 4 13*13 CNN layers plus 2 LSTM layers to train the agent. After 8 hours training, the agent got into the local maximum that only got the corner to hide from the enemies killing themself.
3. More complex CNN model derived from VGG16:



The VGG16 is one of the most famous architecture in 2014 and we fit our model from it to get higher efficiency and better results. It takes all the observation values into consideration including four 2-dimension values and scalar values. We use both tensorflow and pytorch for this model in RLlib and using different algorithms to train the agents.

Reward Shape

The original reward shape only gives reward when agents win or lose their game, which makes it hard and slow to learn how to win a game. It's also easy to get into local maximums such as only evading agents that never try to plant bombs and kill enemies. We modified the reward shape and added more intrinsic reward that guide agents to perform better and training faster.

We modified the reward to take getting reward and destroying wood boxes and enemies into consideration:

Win: +200
Lose: -200
Drop bomb near wood boxes and enemies: # of wood and enemies around the bomb
Move into reward position: 3
Base reward each step: 0

5) 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

DQN (Deep Q-Network)

Based on the concept of finite input and output space, we deployed DQN algorithm which is derived from Q Learning.

It replaces the q value table with a neural network. The input of the network takes the state of the environment (in our case, the status of the board) and the output is the values received when certain actions are taken.

In our case, there are 6 actions, so the output of the network is 6.

In the Q learning, we update the q value using the formula:

Q(s,a) = r + gamma * (max_action(Q(s',a'))

[where; s: state of the current environment, s': next state of the environment a: action taken by the agent, r: reward earned at this moment after take the action, gamma: discounted factor, Q value is immediate reward r + discounted reward(Q(s'a), max_action: the max q value for all the actions]

In DQN, we make one neural network as pivot and train another neural networks according to the bellman equation, and then after a few episodes we update the parameters to the fixed network.

In our implementation, there are two neural networks.

1. Network_eval
2. Network_target

The training process would be like this,
Firstly, we fix the target network, and use network as the standard.

In order to get the expected value for state s(i) to train our network_eval against, we calculate the expected gained value of all actions of state s(i+1), pulsed it with the reward received in state s(i). Even though we know that this value is not actually accurate now, after a couple of updates, the neural network parameters would converge.

store_transsions():

Firstly, the agent plays for several episodes and stores the transitions (s,a,r,s_) into the memory for later learning process, where the

1. s stands for the current state for the game
2. a stands for the action took by the agent when presented with state s
3. r stands for the reward gained by the agent after taking the action
4. s_ stands for the new state reached by the environment

get_action_for_state():

Get the best action for the state s.

Feed the state to the neural network and get the value for each possible action.

Method to avoid local maximum: In 10% of the cases, we generate random actions instead of the action mentioned above.

Advantage:

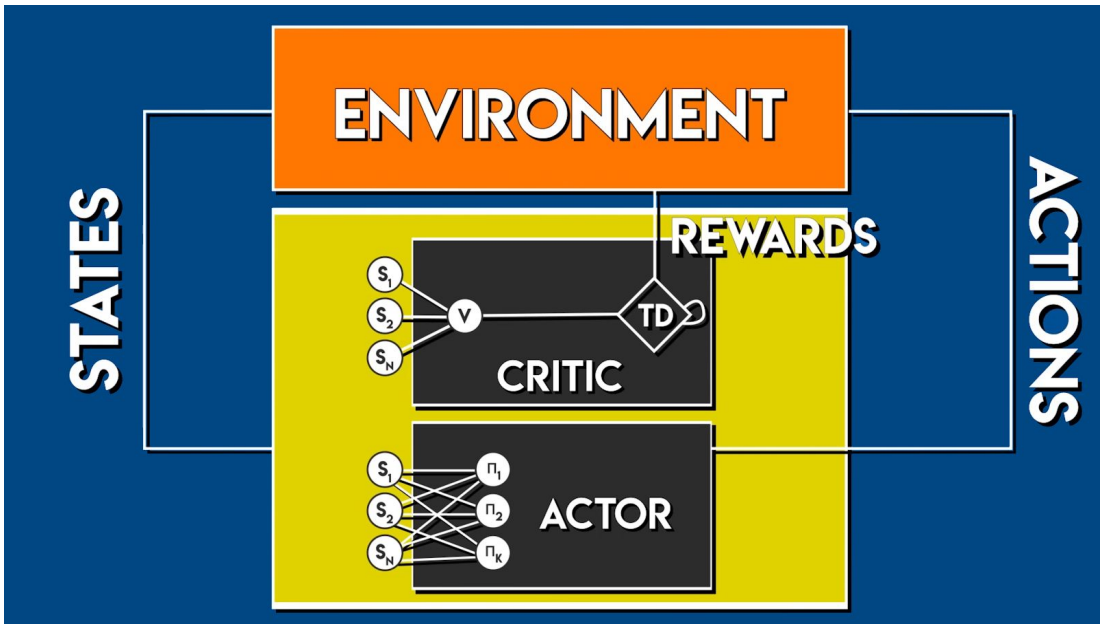
1. Learn from past experiences - It uses a replay buffer to store past experiences and the agent can learn from them.
2. Offline-Learning: Since the history of policies are stored in a buffer, agents can learn offline by performing batch processing on their learnings.

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:

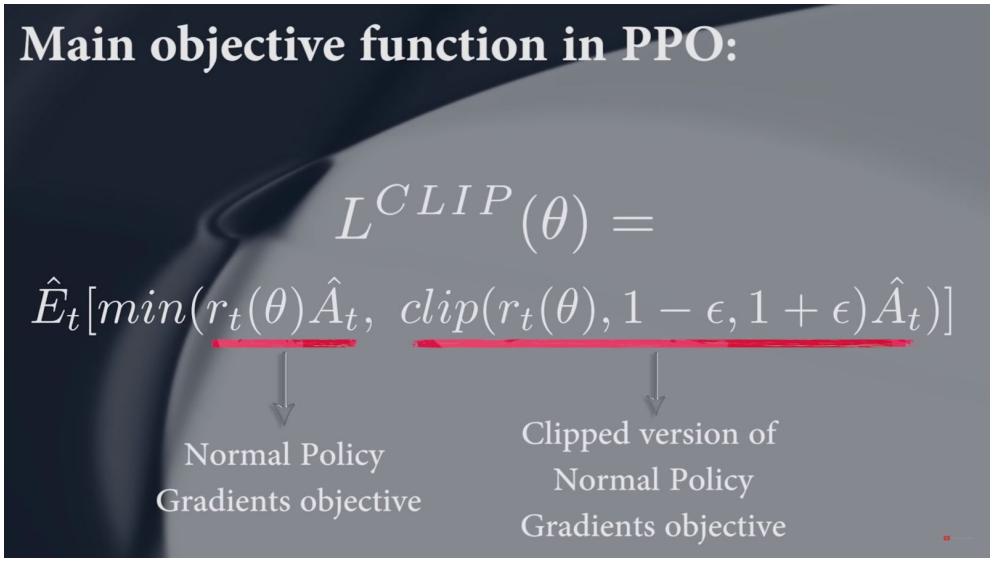
1. Actor Critic models tend to require much less training time than policy gradient methods.

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:

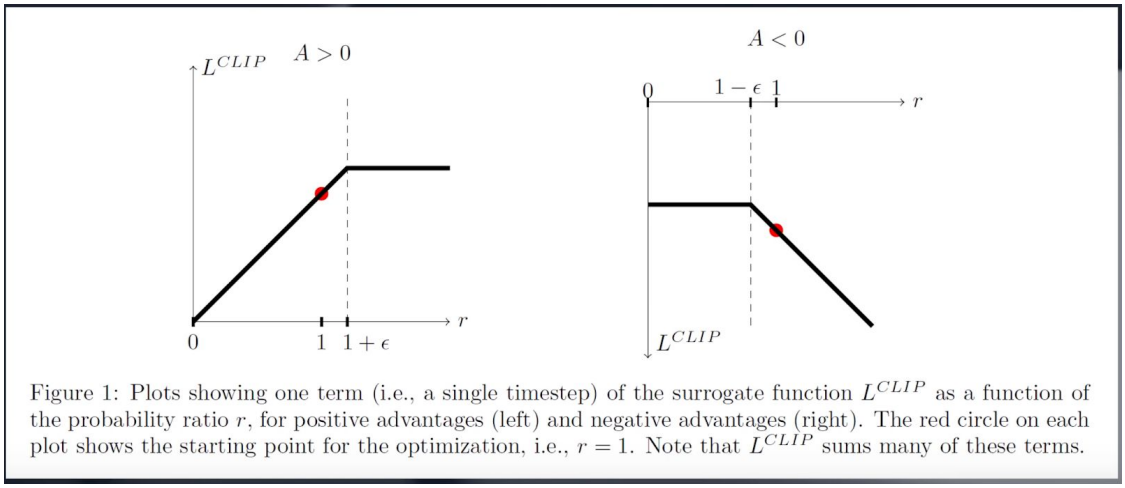
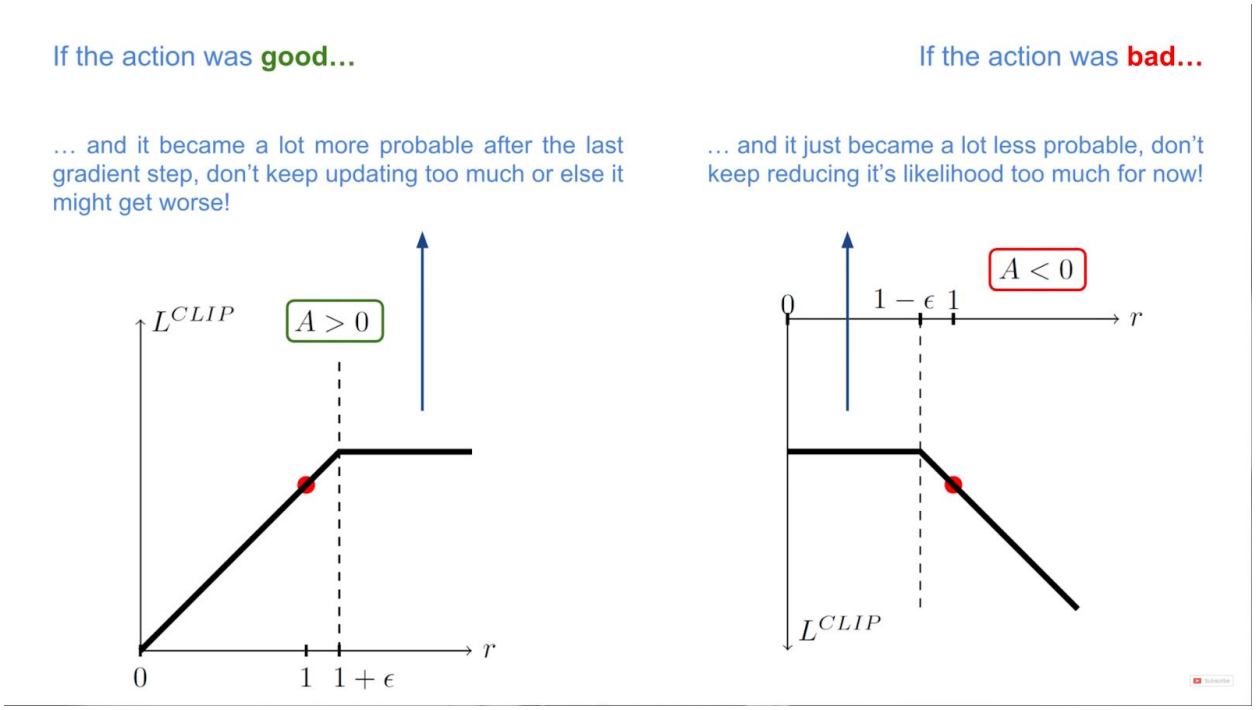


Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{CLIP}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

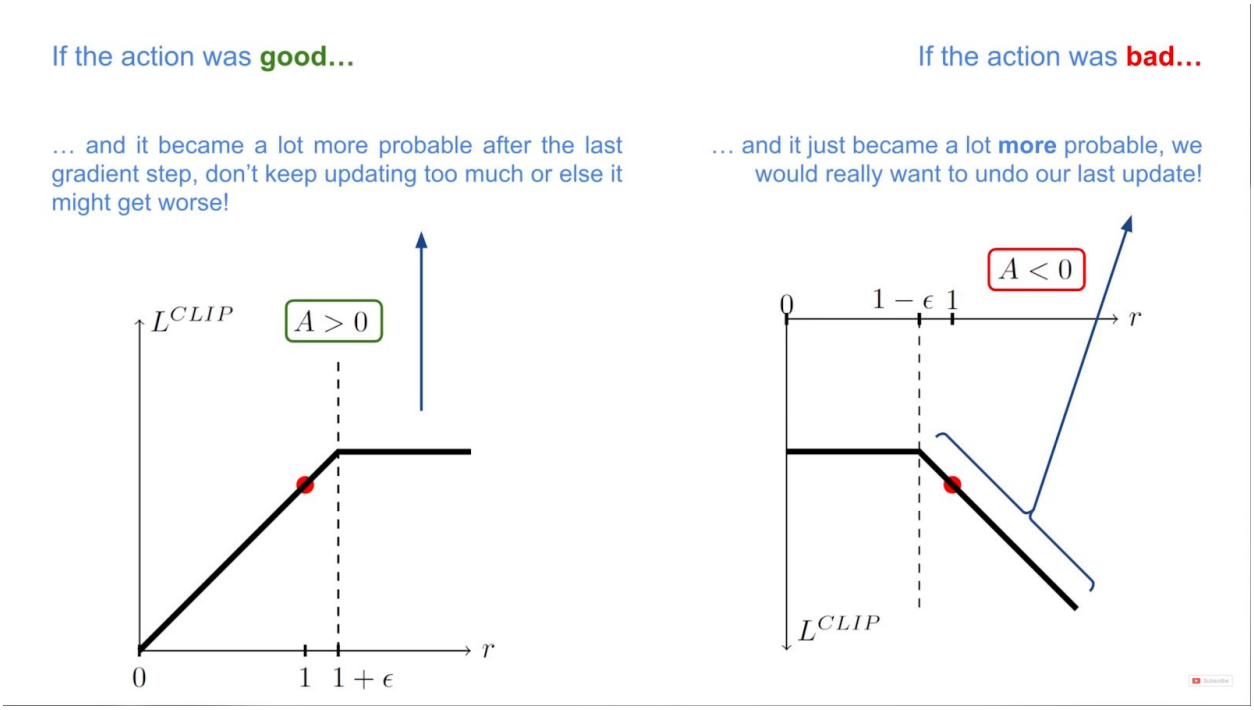
The advantage function might be positive for all the cases when our selected action has better outcome than the expected return or negative in all other cases.

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:



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.

6) References

1. https://gym.openai.com
2. https://www.pommerman.com
3. https://openai.com/blog/gym-retro/
4. http://gym.openai.com
5. https://github.com/M-J-Murray/MAMEToolkit
6. https://drive.google.com/drive/folders/1Czzi1ePTfXIxg99evnx5oVB2Yv6Sh6SE
7. https://github.com/MultiAgentLearning/playground/tree/master/docs
8. https://pdroms.de/files
9. https://github.com/MultiAgentLearning/playground
10. https://pytorch.org/
11. https://www.tensorflow.org/
12. https://morvanzhou.github.io/tutorials/machine-learning/torch/4-05-DQN/
13. https://morvanzhou.github.io
14. https://medium.com/@curiousily/solving-an-mdp-with-q-learning-from-scratch-deep-reinforcement-learning-for-hackers-part-1-45d1d360c120

A2C Algorithm References:

1. https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f
2. https://www.reddit.com/r/MachineLearning/comments/6unujm/d_conceptual_differences_a2c_ppo_reinforcement/
3. https://www.youtube.com/watch?v=w_3mmm0P0j8&t=516s

PPO Algorithm References:

1. https://www.youtube.com/watch?v=5P7I-xPq8u8&t=35s
2. https://www.reddit.com/r/MachineLearning/comments/6unujm/d_conceptual_differences_a2c_ppo_reinforcement/
3. https://www.youtube.com/watch?v=vQ_ifavFBkI

Papers references:

1. https://people.eecs.berkeley.edu/~shah/docs/pommerman.pdf
2. https://arxiv.org/ftp/arxiv/papers/1907/1907.06096.pdf
3. Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).