Battleship AI

Introduction & Related Work

The project is about researching for the algorithms for playing Battleship game. Under the assumption that the ships are placed randomly on the board without violating any rules, we will focus on the algorithm that can hit all the enemy ships in the least steps.

In 2011, Nick Berry designed a probabilistic algorithm that can complete a game in about 55 turns on average. As we will discuss it later in detail, this so far the best traditional algorithm for this game. We treat it as the baseline of our project and try to find out whether popular machine learning models such as CNN or reinforcement learning model like policy gradient is better than traditional method or not.

Of course, this is not a serious scientific research problem, so there is not much related work to talk. Other than Nick's method, our RL approach is mainly inspired by Sue He who uses a DQN model and collects human placed boards for model training.

Code at: https://github.com/JasonSheng1996/Battleship-RL-CNN

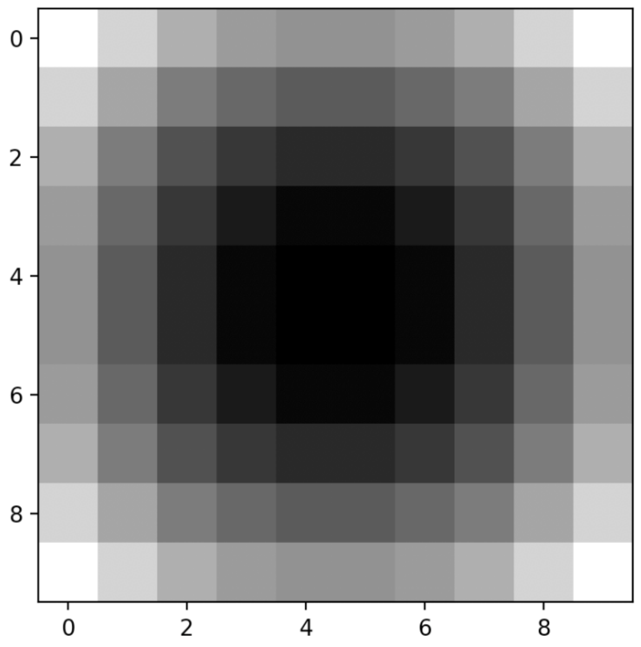
Deterministic and probabilistic strategy for Battleship board game

A simple implementation of this game is to create a stack of potential targets. Initially, the computer is in Hunt mode, firing at random. After a hit, the four surrounding grid squares are added to a stack of 'potential' targets (or less than four if the cell was on an edge/corner). Cells are only added if they have not already been visited (there is no point in re-visiting a cell if we already know that it is a Hit or Miss).

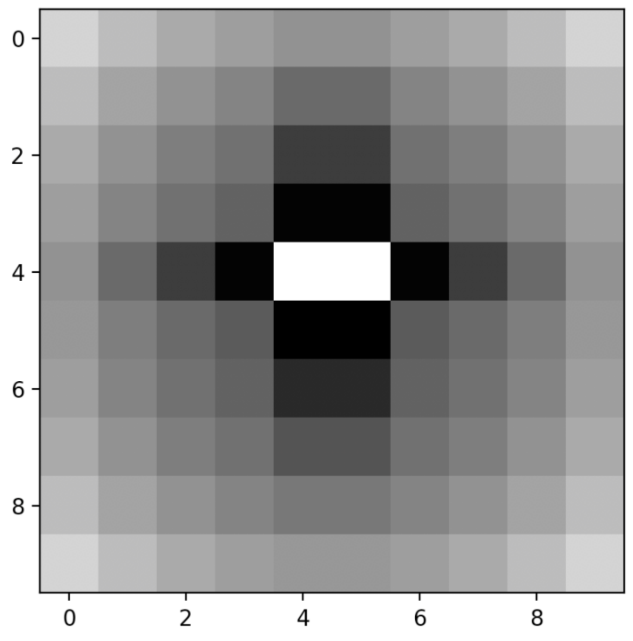
To move further, we can instruct our Hunt algorithm to randomly fire into unknown locations only with even parity. Because even the smallest ship has a length of 2. Once we've sunk the two-unit destroyer, we can change the parity restriction to a larger spacing

But here is a better algorithm based on probability. The basic idea of this approach is described in this blog[1] but without ready-made implementation. The idea is we start in the top left corner and try placing it horizontally. If it fits, we increment a value for each cell it lays over as a 'possible location' in which there could a ship. Then we try sliding it over one square and repeating … and so on until we reach the end of the row. Then we move down a line and repeat. Next we repeat the exercise with the ship oriented vertically. We're always looking for the most likely location for a ship to be located in based on the information we already know. For my implementation, please refer to **probablistic_ship.py**.

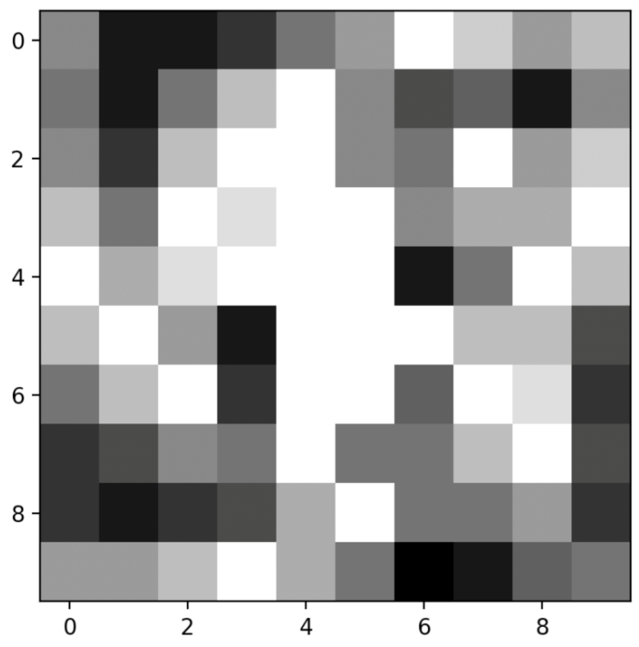
[1] http://www.datagenetics.com/blog/december32011/?fbclid=IwAR3C8ordqQQqtEdk0MGAAvIJ-sl-vvQtLoOEeGjTnVCzw0x8IL4JivmI0dE)



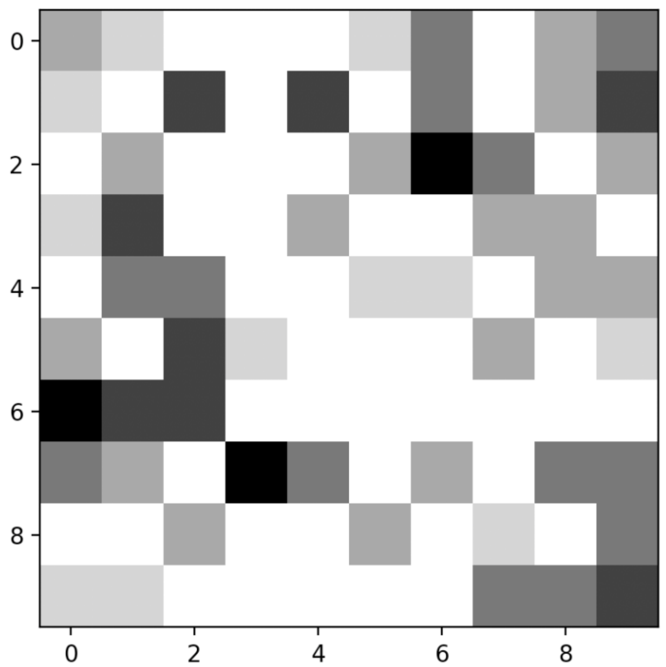
Initial State



Makes two hits

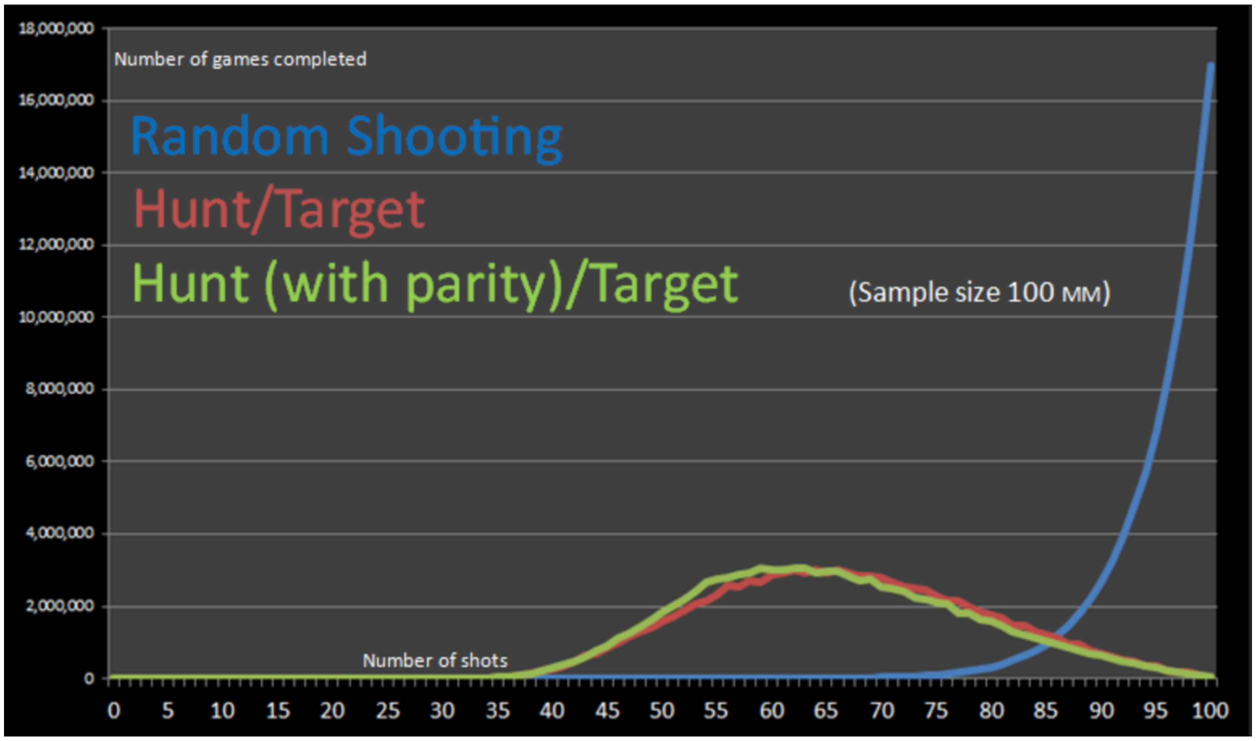


Searching for the next hit

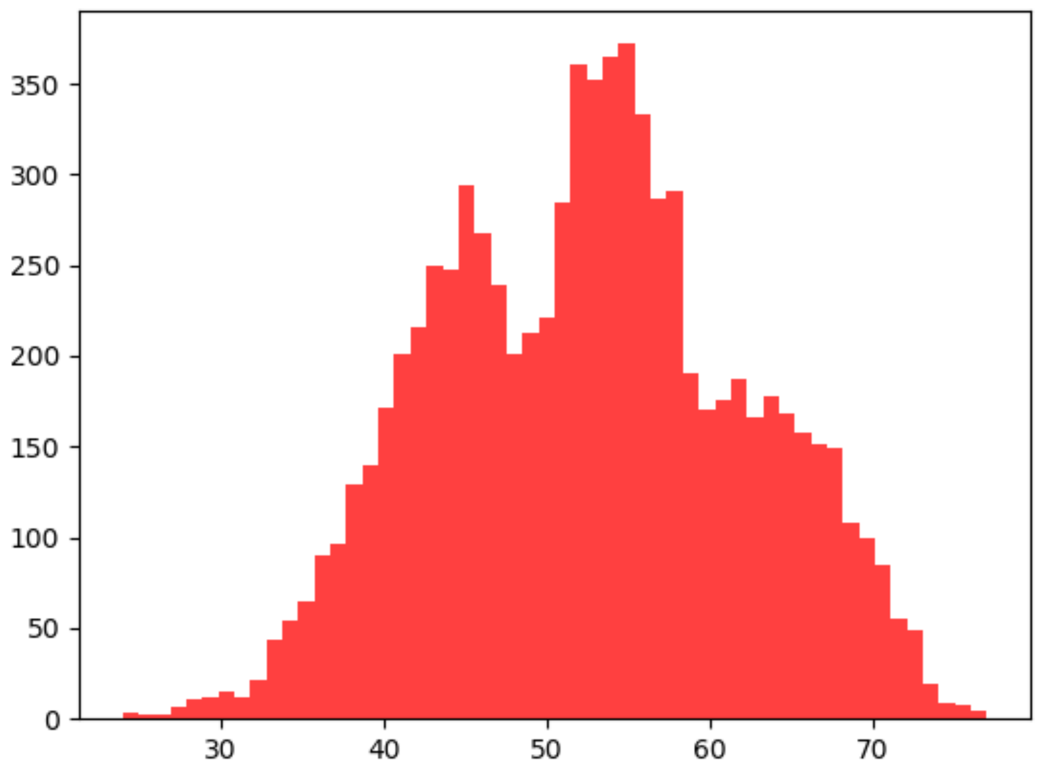


Sinks all ships (another round of game)

Result:



From http://www.datagenetics.com/blog/december32011/?fbclid=IwAR3C8ordqQQqtEdk0MGAAvIJ-sl-vvQtLoOEeGjTnVCzw0x8IL4JivmI0dE



Histagram: 8000 simulations of using probablistic method. The average step is 56.64
X_axis: number of steps to finish a game. Y_axis: number of games.

CNN approach

The probabilistic approach shed light on the upcoming machine learning approach. For the battleship game, the basic idea behind probabilistic method is to choose the next cell which is most likely to be a hit. This idea naturally maps to the generalization of a multiclass classification problem: given an input, choose the most likely label given our training data target. Here the input is an "image", or a state in the game with the game board marked as 0 (unvisited), -1 (miss) or 1 (hit), the output is a discrete probability density function of all cells (we pick the one with largest probability as our next try cell) and the training data target is the ground truth – where the opponent's ships are placed in a game.

The difference between the probabilistic approach and the machine learning approach lies only in its applications rather than idea. In the probabilistic approach, in each turn we recalculate a density function to determine our next move by trying to put ships in available cells. However, we didn't consider the ship overlapping placement problem in order to lower the complexity of this algorithm. The upper bound of all possible placements for ships=[2,3,3,4,5] on a 10*10 board is 200^5, which is quite large if we take overlapping into consideration for our probabilistic approach.

Therefore, we assume that the machine learning approach will get better results, i.e. average steps to finish a game, by learning enough samples to shorten the difference between overlapping and non-overlapping ones.

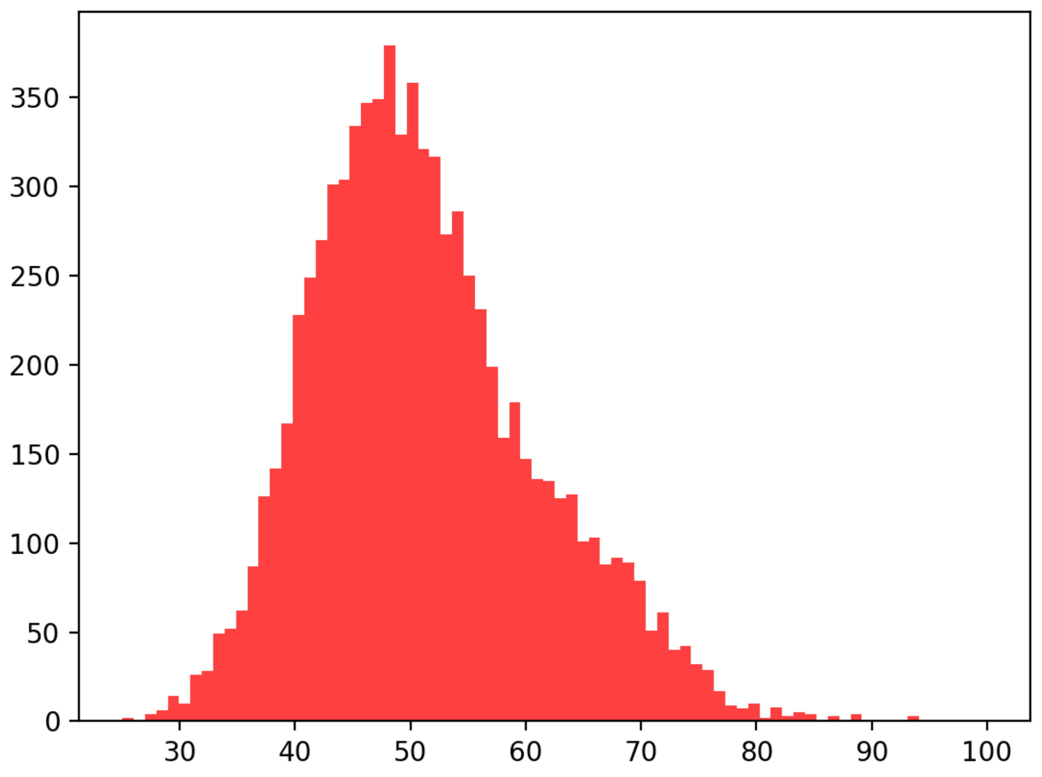
We have tried two different machine learning algorithms. One is convolutional neural network and the other one is reinforcement learning.

Let's first see the CNN approach. The input data for the network is a batch (for example, 1024) of one-channel image (10*10 pixels), which is a state in a game, i.e. the game board in which each cell marked as 0 (unvisited), -1 (miss) or 1 (hit). The label/target for each input is the ground truth of that game, i.e. the place where opponent's ships are placed.

The network we used has roughly three convolution layers and one output/flatten layer: the first layer has 32 filters, the second layer has 64 filters and the third layer has 5 filters. The flatten layer will has the output of shape (1,100).

The final output will be a discrete probability density for all cells on a 10*10 board (of course it is flattened here). For example, if the input is of shape (1,1,10,10), i.e. one image, the output will be of shape (1,100). Each time we will pick the cell with the highest probability as our next move:

After training for around 50,000 games, the loss converges. We let the trained model to play 8,000 games and the average step to finish a game is 51.18. The distribution is:



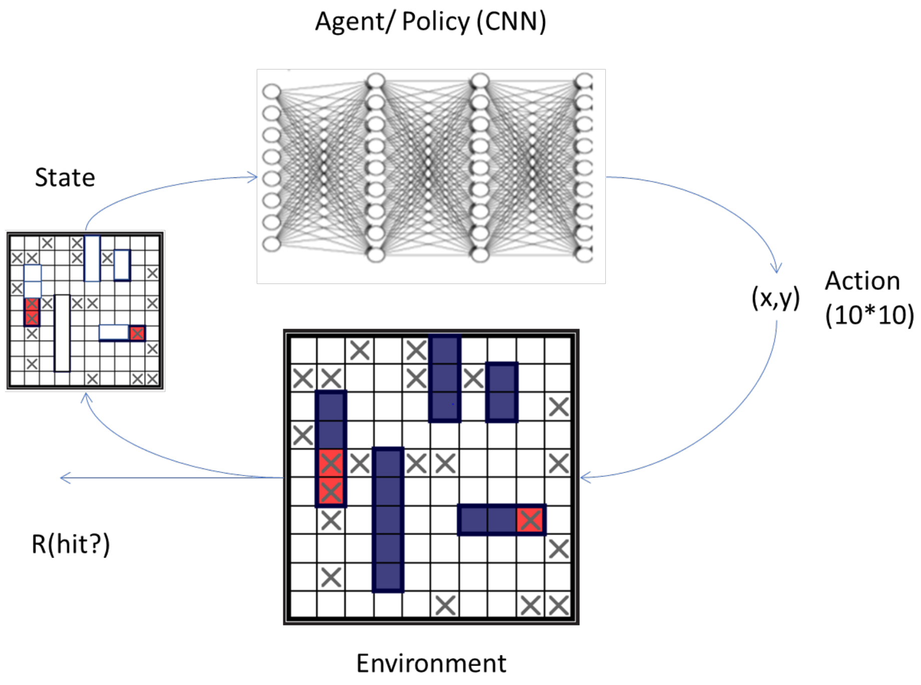
Histagram: 8000 simulations using CNN model.
X_axis: number of steps to finish a game. Y_axis: number of games.

Compared to the probabilistic approach, the performance improves by around 5 steps in average. This model learned and considered the ships overlapping placement problems as we discussed above.

To test the code, please refer to **convnet** file.

Reinforcement Learning

Both probabilistic approach and CNN approach use "greedy" algorithm. They simply shoot the position with largest probability of having the enemy ship. However, we are not so sure that such algorithm is the best. Therefore, the third method we implemented is reinforcement learning. We would like to see if such model can find a global optimum.



We implement an environment with the ground truth and the standard battleship rules. It takes the action we choose and then gives us the reward based on whether we get hit and generate the next state.

On the agent side, we use a CNN network as the actor. It uses the current board as input and outputs the probability of taking different actions. The CNN model has just one 32 channel convolution layer followed by a fully connected layer, which is relatively simpler than the one we used in the pure CNN approach. For training, we used policy gradient.

Policy gradient

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run it on the robot)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

The critical thing about training with policy gradient is designing a proper reward function. Ideally, we should choose a reward function so that it not only reflects our target of finishing the battleship game with as less steps as possible but also gives an unbiased, low variance estimate for expected reward in future (Q-value).

During our experiments, we have tried the following:

1. Raw reward

$$\sum_{t=1}^{T} r(s_t, a_t)$$

   where $r(s_t, a_t) = 1$ if hit, -1 otherwise.

2. Reward to go

$$\sum_{t'>t}^{T} r(s_{t'}, a_{t'})$$

3. Base line

$$\sum_{t=1}^{T} r(s_t, a_t) - E(R)$$

   where E(R) is the expected reward for random strategy.

4. Base line 2

$$\sum_{t=1}^{T} r(s_t, a_t) - E'(R)$$

   where E(R) is the expected reward for probabilistic strategy.

All these reward functions suffer from similar problems. In one episode, a move of hit may get a very close reward to the one of the next missing move. In other word, we don’t reinforce those good moves enough. Therefore, we need to design a reward function that can reflect how much better we can at each step. (Let the reward function does the advantage function job.)

5. Advantage

$$\sum_{t'>t}^{T} (r(s_t, a_t) - E(r(s_t, a_t)))$$

or

$$\sum_{t'>t}^{T} (r(s_t, a_t) - E(r(s_t, a_t))) * 0.99^{t'-t}$$

At each step we subtract the expected reward one can get by shooting randomly from the actual reward. We also tried to calculate the expected reward by current policy using Monte Carlo way.

Unfortunately, none of the above functions works. When using these reward function, the policy gradient training will not converge over 30000 episodes.

6. The one we use:

$$\sum_{t'>t}^{T}(r(s_t, a_t) - E(r(s_t, a_t))) * 0.5^{t'-t}$$

   The training success when we reduce the discount to 0.5. We are telling the agent at current state that only the performance of next few steps is your credit or fault. To be honest, we are not entirely sure this is the right thing, but it does succeed in the experiment.

Network training issue:

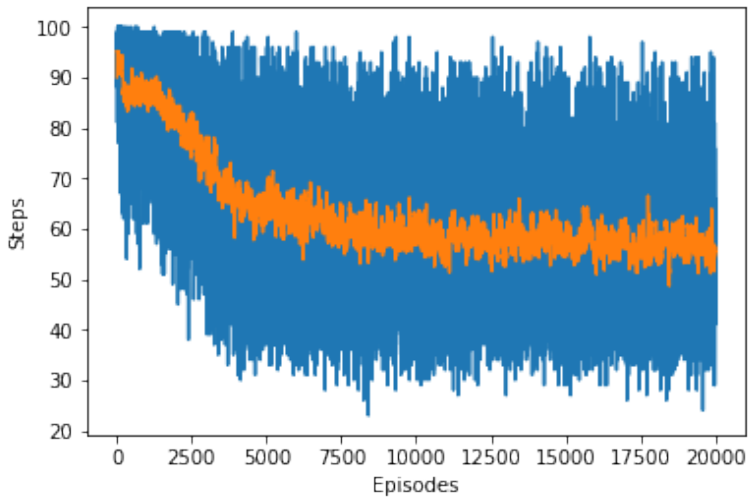
$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\left(\sum_{t'=1}^{T} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})\right)$$

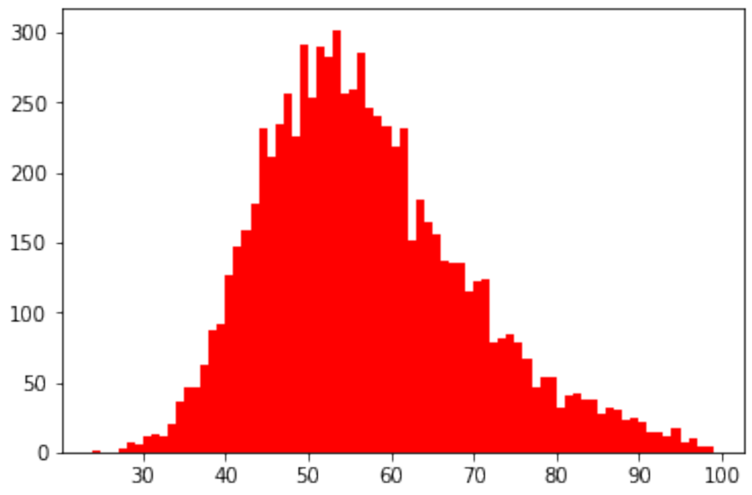
If the network is initialized with real bad parameter or the training is misled by some of previous training sample, the network may output $\pi_\theta\ (a|s) = 0$. The tragedy happens when we have to choose that action to take and eventually we get "inf" or "nan" in the next round of backpropagation.

To solve such problem, we need to prevent some single sample from changing the parameters a lot, especially when there is gradient explosion. The trick we use is called Gradient Clipping. The idea is very simple. We simply cap the gradient to a Threshold value to prevent the gradients from getting too large.

Result

With all the efforts we discussed above, here is the final result for reinforcement learning.





After training for around 20,000 games, let the trained model to play the 8,000 games same as the CNN approach did and the average step to finish a game is 55.54.

Future work

Based on all the experiments we did, we do feel the hardness of designing a proper reward function. Luckily, we do have the way to avoid this difficulty. That is using Actor-Critic, using another machine learning model to replace the human-designed reward function. For example, we can let the estimation and policy share one CNN which takes the board as input and output both reward estimation and action probabilities. (See https://github.com/spro/practical-pytorch/blob/master/reinforce-gridworld/reinforce-gridworld.ipynb)

Teamwork

For implementation part, Jiacheng focuses on RL and Shengchao focuses on CNN and traditional approach.