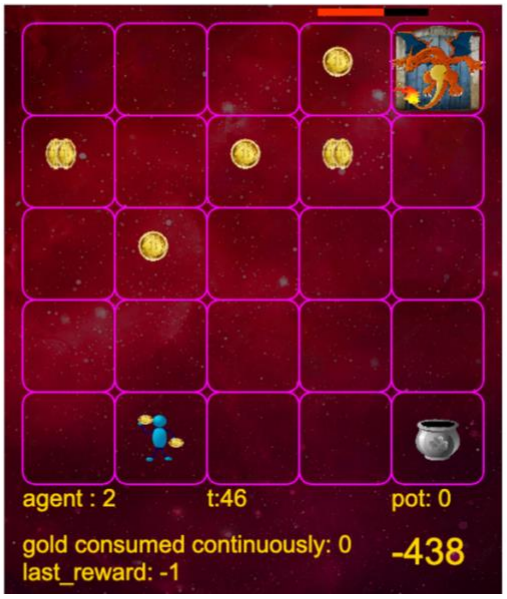
Using Reinforcement Learning to train

A Gold Hunter Agent

Swee Loke – December 2021



Summary

This project demonstrates:

- A game with rewards designed to train the DQN agent to win the game.
- How using special sequence of scenarios helped reducing the number of required training episodes.
- The trained agent can win (escape) around 98% of the time after trained for less than 6000 episodes.
- A graphical implementation that allows the intelligent agent or a human to play interactively.

About the game

The goal is for the agent to get out of the room (door at the upper right corner). The dragon always guards the door, so it is impossible to escape with the dragon at the door. Hence the agent has to collect gold to feed the dragon and let it be occupied near the gold pot in order to escape.

The Dragon

- The dragon loves gold and whenever there is gold in the gold pot, it will move to just above the pot. The dragon only moves up and down along the same column as the gold pot. It moves at the speed of 3 cells per timestep by default (i.e., it only take one timestep to move from door to near the gold pot and vice versa).
- The dragon starts consuming the gold in the gold pot when it is right above the gold pot. The dragon consumes gold coins at the rate of 0.9 per timestep (default). The dragon will move back to the door immediately as soon as the gold pot is empty.

The Agent

- Agent's moves are: Up, Down, Left, Right, None
  Since the game is dynamic, agent can choose to remain on the same square to catch the gold that is dropping.
- The agent collects the dropping gold coins and loads them into the gold pot by visiting the cell where the gold pot is.
- The agent can collect a maximum of 3 gold coins before it needs to unload to the gold pot. The only way for the agent to unload the collected gold is to visit the gold pot.
- If the agent collects (intentionally or unavoidably) more than 3 coins, all the previously collected coins at hand will drop and the agent will faint.
- When the agent fainted, the frozen state will last for 3 timestep and at that time, the agent cannot make any more movement. When the agent is in frozen state, additional coins will not be collected or cause further fainting.

- The agent can faint at most n times (maximum lives, default to 5), shown by the red bar.
- If the agent lost all the lives (due to fainting) or touched by a dragon, the game is terminated.

With the above default values, it is very unlikely for the agent to escape because the dragon consumes the gold very fast and will move back to the door right away

Hence, we added 2 hidden rules to support winning:

- If the dragon consumes at least 10 gold coins consecutively (i.e., not moving away from the pot), its speed will be reduced to 1 square per timestep (shown by the shrinking dragon)
- If the dragon consumes at least 15 gold coins consecutively, it will consume the gold slower after that (0.5 coins per timestep).

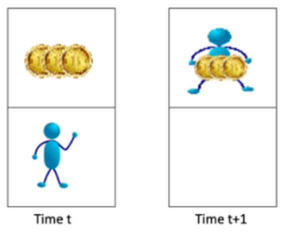
We want to see if we can train the DQN agent to escape.

Features

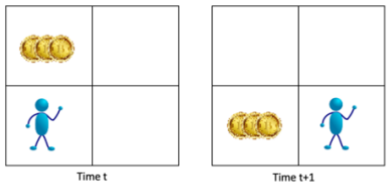
The information shown on the game screen will be encoded as features arrays. They are:

1. Agent location
2. Agent's previous location

   Only the immediate previous location is stored. The game is dynamic, the previous steps are not important in the game. The only reason we keep the previous location is to show the direction of the movement. This is important for the DQN agent to train. When the agent is moving upwards and the gold coin is dropping downwards, the agent will catch the gold in crossing even at no time t, they occupied the same cell.



   Whereas if the agent moves left or right, it will not catch any gold.



3. Is Agent fainted?
4. Agent frozen state countdown
5. Number of lives left

   This is the max lives – number of times the agent has fainted. The game is terminated when it reaches 0.
6. Number of coins the agent is carrying
7. Number of coins in the pot
8. Dragon Location
9. Dragon's previous location
10. Dragon's moving speed
11. Dragon's gold consumption speed
12. Number of coins consumed in succession
13. Gold coin locations and each gold's weight (either one, two or three coins).
14. Gold coin previous locations and each gold's weight (either one, two or three coins).

The position related features are encoded in the np array reflecting the locations, whereas the values global are encoded with the same value across the entire 5x5 array.

For example, the agent's current and gold locations are encoded as:

```
agent_loc:
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]

gold_loc_w: (the number indicate the number of coins)
[[0. 0. 3. 3. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
```

```
   [0. 2. 0. 0. 0.]]
```

The agent’s carried gold coins (on hand) are encoded as it is not dependent on any location.

```
agent_gold_count:
[[2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]]
```

Rewards

We use different rewards to influence the actions picked by the agent. We can even train the agent in some epoch using different rewards value and then resume training with different set of values.

Reward Type	**Value**	**Purpose**
`NORMAL_STEP`	-1	Normal step cost, a negative cost to discourage agent wasting time repeating steps that does not bring value
`ESCAPE`	5000	Winning because the agent escaped. Agent may have 0 – 2 gold.
`ESCAPE_WITH_MAX_GOLD`	8000	Winning as the agent escape with 3 gold coins. We have this special case to see if Agent will optimize to take 3 gold coins when escaped.
`CATCH_GOLD`	1 * num of coins	We want to encourage this action, so set a positive reward. But catching gold itself has not much value unless the gold is loaded, so it is a small
`LOAD_GOLD`	10 * (2 **num of coins)	We want to encourage the agent load maximum number of gold coins instead of 1 by 1. So we reward much higher for 3 coins.
`LOAD_GOLD_BEFORE_EMPTY`	50	In addition to loading gold, we also reward the agent if the agent manages to load the coin before the pot is empty. This is to help agent discover that feeding the dragon continuously pay off.
`FAINTED`	-200	We want to discourage fainting as fainting 5 times will cause the game to be over.
`DEAD`	-1000	The game is terminated when the agent has fainted 5 times or the agent met the dragon on the same cell.
`REACH_MAX_GOLD`	300	This reward is given when the agent has loaded (cumulative) the max_number (we set to 100) coins. Originally, I thought we can use this to teach agent to learn to load many gold coins into the pots. But this turns out to be not useful, so it is no longer in use.

CNN Models

Started with initial CNN layer with 25 filters, and later kept reducing it and finally use only 7 filters.

The input layer is (5, 5,14).

```
_________________________________________________________________

Layer (type)                 Output Shape              Param #
=================================================================

conv2d (Conv2D)              (None, 5, 5, 7)           105
_____________________________________________________________

flatten (Flatten)            (None, 175)               0
_________________________________________________________________

dense (Dense)                (None, 128)               22528
_________________________________________________________________

dense_1 (Dense)              (None, 128)               16512
_________________________________________________________________

dense_2 (Dense)              (None, 64)                8256
_________________________________________________________________

dense_3 (Dense)              (None, 32)                2080
_________________________________________________________________

dense_4 (Dense)              (None, 5)                 165
=================================================================

Total params: 49,646

Trainable params: 49,646

Non-trainable params: 0
```

Training the DQN agent

- Using the default environment to train the agent will take long time, and even after thousands of epochs, the agent would not have won any meaningful number of games for any useful training towards the end goal. It will require training for a long time before it stumbled into a winning sequence by chance (as it needs to feed the dragon 10 coins continuously with the dragon not leaving the spot in order to slow down the dragon).

- Earlier on in the training, I fixed the epsilon to a high value for the first 1500 epoch, but that didn't help. Agent needs to leverage what it learns earlier on to help getting to the winning path, instead of mostly random actions. Hence it is preferrable (for this game at least) for agent started using what it learns to generate training samples that will help it to move closer to the scenario required for winning (i.e., feeding the dragon at least 10 coins continuously). Hence it is important not to give epsilon any special treatment (i.e., fix the epsilon at high value) earlier on the game.

- Smaller batch size influences the earlier training progress. It seems like at the earlier stage, it is critical to use smaller batch size so that we can start training the agent early, and the earlier advantage is compound. Using a batch size of 256, the agent has won only very few games in the first 1000 episodes (with special initial state). And on average, the agent won fewer games in the first 2000 episodes with using batch size 256 vs 128. However, the advantage diminishes after 3000 episodes. Larger batch size improves the performance faster in the later training.

- More complex models (more neurons/layers) do not guarantee to achieve better performance in the same amount of training time. I started with 25 CNN filter, and then reduced to 10, and then 7 (with even fewer layers), and the model train fasters (on each epoch) and achieve similar performance.

- Since the game is too hard for the agent to train solely by random explorations, we will need to train the agent with some initial environment (instead of the default) to improve the chance of winning.

  Earlier I used the special reward to encourage the agent collecting gold coins and load them to the gold got. But the agent ended up only loading the pot over and over and did not know the goal was to escape.

- We help it by creating the scenario near the end (close to win the game), so that the agent can win some games by randomly reaches the door and that can help it train further.

  We do it by

  - On the first 1000 episodes, we set the gold pot count to 20, so that that will keep the dragon near the gold pot. It can also consume the gold in successions. That will increase the chance to win. This is the easiest scenario for random exploration to win.
  - On the 2nd 1000 episodes, we set the gold pot counts to the episode number mod 15. That will introduce a range of low to high gold counts and the agent may win some with increased difficulty than the first 1000 episodes.

After the first 2000 episodes, the agent is trained on the same default environment as the previous training case. As we can see, it starts winning some on its own without further help.

- It is more important to have good features than to build complex models with average features.
- I tried tuning different hyper parameters earlier, but because the agent rarely wins any games in the first few 4000 episodes (with default setting), the different parameters did not help winning the game.

Model Performance

The DQN has achieved amazing performance even with training < 5000 episodes. It wins above 90% when testing with 1000 episodes.

Epoch	**% Wins of 1000 games**	**Win with NO max gold**	**Win with max gold**
4000	0.9050	0.61	0.39
5000	0.9830	0.57	0.43
6000	0.9730	0.58	0.42
7000	0.9760	0.53	0.47

By training more episodes, the agent did not increase the % of escape per 1000 games, but it increase the times it escapes with max gold coins.

Game design trade off

When designing the agent, we need to fine tune the speed of the dragon. If it moves or consumes the gold coins too quickly, the agent will never win. If too slowly, the agent can always win in few steps. In order to make this game challenging, we need to incorporate the possibility of reducing the speed in certain scenarios (i.e., after consuming certain number of coins consecutively).

Hence, we also include the following as features

- The speed of dragon moves
  Technically these values can be derived from the movements of the dragon's previous location and its current location. But it will take too long for the agent to infer that info, so we spell out the speed as a feature.

- The rate of dragon consumes coins
  We need to include the rate because there is no previous pot gold count as a feature. Instead of using previous count and current count, we added the rate as a feature based on the same reasoning as above. It would take the agent too long to learn the correlation of gold coins consumed in succession and the rate of coins in pot decreasing.

No need to add features preventing loops (like in the Wumpus' case) because the constant moving parts (gold dropping) of the game, it will always move the game to a different state.

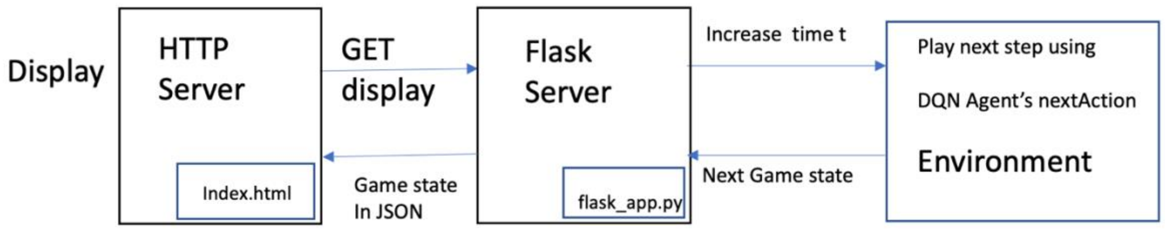
Future enhancement

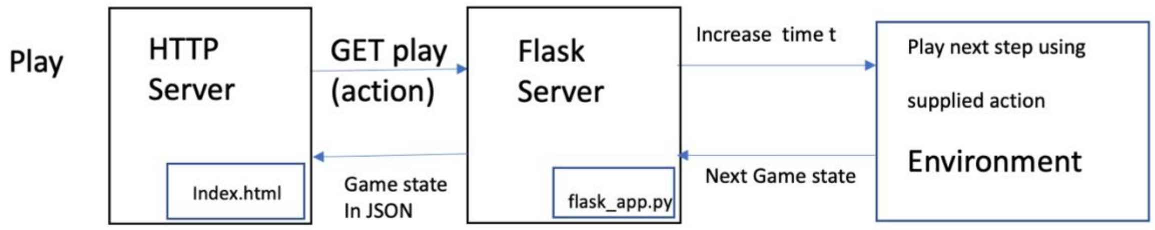
After testing the game, I realize that because of the speed of the dragon moves, agent cannot go too far to collect gold coins. Hence it mainly stays within column 3 to 5. If we were to redesign the game, we would put the door and the dragon on the middle column, so that the agent can move both left and right to collect more coins.

GUI Game Implementation

The game GUI is implemented in Java script and PIXI (executed by HTTP server). The core environment and agent are implemented in Python (executed by Flask server). The HTTP server makes REST API call to the Flask server and receive the states (for rendering) in JSON.

The same architecture is used for rendering games played by human or automatically by the agent.





Conclusion

When I design the game, I did not expect that we would be able to train the agent to play the game in such high performance within fairly short training time (<6000 epochs). This is all made possible by discovering how to create winning scenarios by setting the initial environment. So it is critical not only to design a good model/features, but also need a training strategy.