

# Project report for AML: A machine learning approach for 2048

Wenyao JIN      wen-yao.jin@student.ecp.fr  
Meiqi GUO      mei-qi.guo@student.ecp.fr

March 27, 2017

## Abstract

In the frame of the course AML, this report will present methods, experiments, difficulties encountered, during our approach for the game of 2048. Results obtained and future directions will also be discussed at the end. Our partition of work is Wenyao JIN:60%, Meiqi GUO:40%.

## 1 Introduction

### 1.1 Game rules

The highly addictive game 2048 enables single player to play on a square board of 4x4 tiles. Each tile contains a number which is always a integer to the power of 2. Player have 4 possible actions of direction to make at each step: left, right, up, down. When an action is made, the game will push every tile to the direction given by the player, and merge tiles that have the same number by multiplying the tile by 2. Every when an action is taken, the game will regenerate a 2 or 4 at a random empty tile. When no more empty tile is on the game, and no more action can be made to change the board, the game stops. The goal of the game is to get a 2048 tile. See Figure 1 for an illustration.

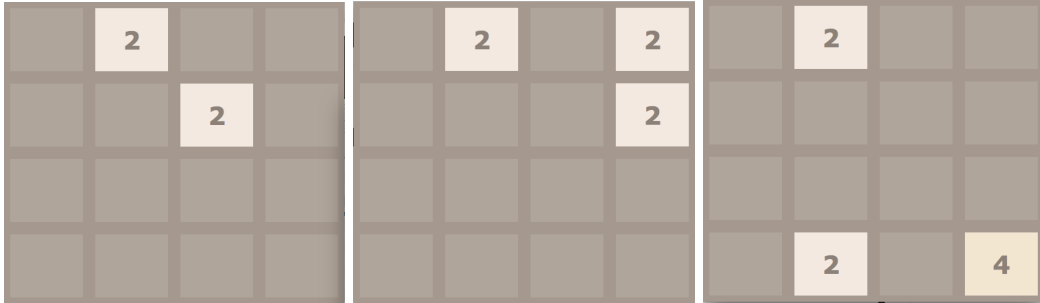


Figure 1: game rules explanation: step 1 (left), step 2 (middle), step 3 (right)

### 1.2 Game score

Apart from the goal of the game, a score is also kept to give players the possibility to score for larger tiles after hitting 2048. The score is incremented by the formula  $X(\log_2 X - 1)$  every time a tile of  $X$  is made. We found the score rule very reasonably designed, because it gives bigger weight on larger tiles. Moreover, we wanted to learn our game as it is, so we kept the score system as the reward for our learning agent.

## 2 Methods

In this section we represent every methods tested in general and their parameters.

## 2.1 Temporal difference learning

The temporal difference learning (TDL) is the most discussed method during the course, it use an update function to modify state value function  $V^\pi : S \rightarrow R$  with reward  $r$ :

$$V(s) = V(s) + \alpha \delta Tr$$

$$\delta = (r + V(s_{next}) - V(s))$$

where  $Tr$  is the eligibility trace introduced in the course.

Other method such as Monte-Carlo tree search are also possible. However, due to the time and resources of this project, we finally chose to use the TDL frame. In the following subsection, we will discuss 2 different policy and their update functions that we used in this TDL frame.

### 2.1.1 Q learning

The Q-learning algorithm propose a learning on action-state value:

$$Q(s, a) = Q(s, a) + \alpha(r + \max_{a_{next} \in A} Q(s_{next}, a_{next}) - Q(s, a))$$

Q-learning agent don't have any extra information beside the next state and reward it receives after each taken action, thus the agent don't have any knowledge about how the game works. This is why, we need to sometime adjust reward feedback to prevent the agent from bugging sometime. For example in 2048, we have certain configuration that certain action won't affect the game. In such scenario, a negative reward needs to be affected to teach the agent to stop test certain action.

### 2.1.2 Afterstate

Despite its well known generalization power, Q-learning algorithm don't know about the game rule, which make the learning process very slow. Much care is also needed to prevent the agent from getting stuck. [SZU14] proposed another policy where the agent can calculate the score received from its action, which meanings that it knows about the deterministic part of the game rule:

$$a = \arg \max_{a' \in A} V(s_{after}(s, a')) + r_{after}(s, a')$$

where  $s_{after}(s, a)$  referring to the game state after we take action  $a$  on  $s$ , and  $r_{after}(s, a)$  its reward:

$$s \xrightarrow{a} s_{after}(s, a) \xrightarrow{\text{randomly add 2}} s_{next} \xrightarrow{a_{next}} s_{after}(s_{next}, a_{next})$$

The agent don't know about the stochastic part, so it still needs to learn about the law. Since the agent knows already about it's current reward, the update function will focus on update its afterstate:

$$a_{next} = \arg \max_{a' \in A} V(s_{after}(s_{next}, a')) + r_{after}(s_{next}, a')$$

$$V(s_{after}(s, a)) = V(s_{after}(s, a)) + \alpha(r_{after}(s_{next}, a_{next}) + V(s_{after}(s_{next}, a_{next})) - V(s_{after}(s, a)))$$

Afterstate algorithm resolve completely the problem of bugging which will occur in the q-learning algorithm given a improper representation.

## 2.2 Representations

For the game 2048, if we don't consider playing after obtaining the 2048 tile, we will have 11 states possible for each tile. The entire possible state for the game will be very big:  $11^{16}$ . So simpler representation is very much needed.

We didn't tried to use any representations by approximation functions, because we think the board condition is very complex and subtle to be fully represented by approximation.

Learning from our own game experience. The representation needed, is one which have its focus only on the local area where the largest tile is located. Also, we want a representation to reveal relations between each tile. After searching online, one representation called N-tuple caught our

attention. The following explanation is given by [SZU14].

An N-tuple network consists of  $m$   $n^i$ -tuples, where  $n^i$  is tuple's size. For a given board state  $s$ , it calculates the sum of values returned by the individual n-tuples. The  $i$ th  $n^i$ -tuple, for  $i = 1 \dots m$ , consists of a predetermined sequence of board locations ( $loc_{ij}$ ) for  $j = 1 \dots n^i$ , and a look-up table  $LUT_i$ . The latter contains weights for each board pattern that can be observed on the sequence of board locations. Thus, an n-tuple network implements a function  $f$  :

$$f(s) = \sum_{i=1}^m f_i(s) = \sum_{i=1}^m LUT_i[index_i(s)]$$

where  $index_i(s)$  retrieve from lookup table the weight of  $i$ th tuple associates with the state  $s$ . As you can see, we use a linear representation by summing up all the weights given by the tuples

Several n-tuple network has been used in our project. See Figure 2 for illustration. Tuple 1 is the most intuitive n-tuple network. Each tuple have 4 tiles: horizontal tuples will serve to detect possible high value for left or right actions while vertical ones will detect up or down action. Square tuples will give more focus one local configuration. This tuple have 17 tuple with 4 tiles, which give us a total of  $17 * 11^4 = 248897$  weights. This is the tuple that we found most promising and spent the most of time training.

Tuple 2 is slightly modified from tuple 1 by removing several square tuples. This tuple is conceived because we think we might be more interested in the corner configurations. Hopefully, this may somehow reduce complexity of our model and make improvement. Tuple 3 is taken from [SZU14]. Since this kind of tuple don't have the property of symmetry, it will need special care with symmetric sampling when training. We see that in this network, we have 2 tuples of 6 tiles, this will dramatically increase our total weights:  $2 * 11^4 + 2 * 11^6 = 3572404$  weights. Tuple 4 is taken from [YEH16], this network have  $4 * 11^6 = 7086244$  weights.

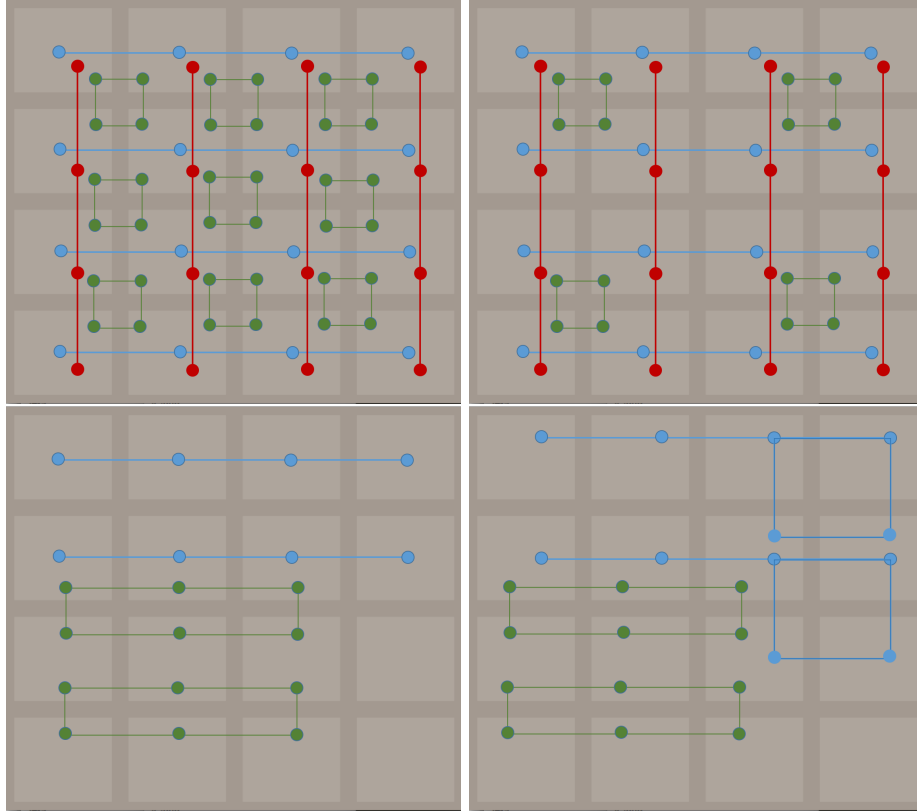


Figure 2: tuples figure explanation: tuple 1 (upper left), tuple 2 (upper right), tuple 3 (lower left), tuple 4 (lower right)

## 2.3 Exploration

Since the game is by its nature stochastic, exploration may not be necessarily needed. We've however implemented  $\epsilon - greedy$  for each method to test its impact on performance. For Q-learning, every act we have a probability of  $\epsilon$  to make a random act. For afterstate, if an afterstate is never visited, we will choose it at a probability of  $\epsilon$ . An alternative to this, is linearly decrease epsilon to make the agent exploit more and more as the training goes on.

## 2.4 Symmetric sampling

Since the game is symmetric, it means that at each step, we can rotate and transpose the game to do updates 8 times(4 rotation\*2 transpose). As mentioned earlier, tuple 3 and tuple 4 need symmetric sampling.

## 2.5 Other Representations

2048 needs long term strategy, because to reach 2048 tile, we need at least 1024 steps. Eligibility trace won't be able to cover this kind of strategy. Other man made representations may be able to boost the learning. For example, we notice that monotony, which is applied by human players to reach for higher score, is a sign for good game state. So by giving a bonus to monotone tuples weights can encourage agent to adopt a better strategy. In this game, we've made a possibility to give monotony bonus on tuple 1. See Figure 3 for explanation.

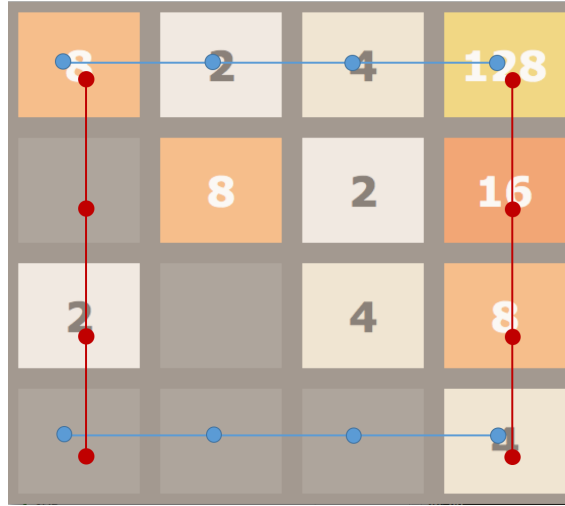


Figure 3: The tuple on the right and the one on the bottom may get extra reward

## 3 Comparison of methods

To this state, we have several parameters and methods to choose. See Table 1 for illustration. In this section we will illustrate our result for different parameters(Our final winner parameter is thickened in the table 1). However, we wasn't able to test every combination, due to the lack of time. We will use a stepwise exploration method. To evaluate performance of agents at each time, we take the average of last 100 episode played, each score being log to the largest tile obtained.

### 3.1 Policy

See Figure 4 for illustration. We can see that in short time training, Afterstate policy performs way better than Q-learning which is just slightly better than a random agent. Plus, Q-learning has a serious problem of bugging due to our local tuple representation: the agent won't be able to detect efficiently an action is futile. It will need to penalize each tuple with the same negative reward until they turn nearly negative to escape from the trap. This will undo all the learning that's been done for these tuples. This is more dangerous if the learning rate is changed to  $t^{-1}$

Parameter/method	choice
Policy	Q-learning, <b>Afterstate</b>
Learning rate	<b>0.0025</b> $t^{-1/2}$ , $0.005t^{-1/2}$
Forget rate	<b>0.0</b> , 0.3, 0.5, 0.8
Epsilon	<b>0.0</b> , 0.01, $0.01 - t/2000$
Symmetric sampling	True, <b>False</b>
Tuple	<b>tuple1</b> , tuple2, tuple3, tuple4
Mono	<b>0.0</b> , 0.025

Table 1: Parameters to choose

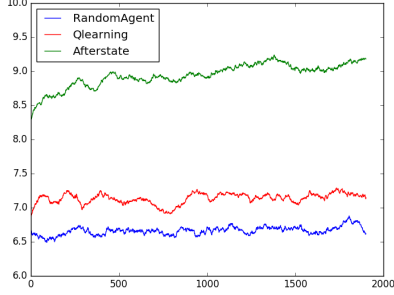


Figure 4: performance

Policy	Q-learning	Afterstate
Learning rate	$0.0025t^{-1/2}$	$0.0025t^{-1/2}$
Forget rate	0.0	0.0
Epsilon	0.0	0.01
Symmetric sampling	False	False
Tuple	tuple 1	tuple 1
Mono	0.0	0.0

Table 2: parameters

instead of  $t^{-1/2}$ . Afterstate don't have this problem because it knows about the game rule and can easily avoid futile action.

### 3.2 Tuple

See Figure 5 for illustration. From this graph we can see clearly that tuple outperform other tuples. What's remarkable is that, tuple 3 is train with symmetric sampling, so it's trained 8 time more than tuple 1. From [SZU14], we see that tuple 3 may have the potential to outperform tuple 1 in the long run, but due to limits of computational power, we will settle with tuple 1.

### 3.3 Forget rate

See Figure 6 for illustration. Apart from forget rate, other parameters are fixed as in the table. We can see that the 0.0 and 0.3 outperform other rates, while it seems to be indiscernible between the two. We will try to train them for another 9000 episode to see if there will be any difference.

### 3.4 Exploration

See Figure 7 for illustration. In this part, exploration (linear decrease) is tested, together with the unsettled choice between 0.0 and 0.3 forget rate. We can see that this three models are indiscernible

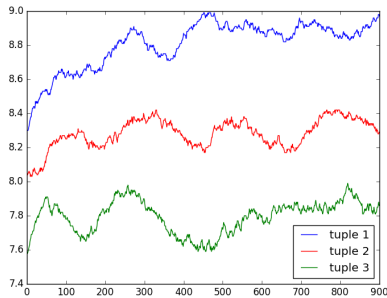


Figure 5: performance

Tuple	tuple 1	tuple 2	tuple 3
Policy	Afterstate	Afterstate	Afterstate
Learning rate	$0.0025t^{-1/2}$	$0.0025t^{-1/2}$	$0.0025t^{-1/2}$
Forget rate	0.0	0.0	0.0
Epsilon	0.0	0.0	0.0
Symmetric sampling	False	False	True
Mono	0.0	0.0	0.0

Table 3: parameters

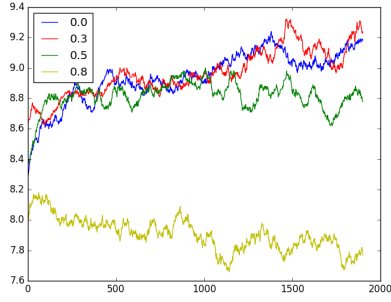


Figure 6: performance

Forget rate	0.0,0.3,0.5,0.8
Policy	Afterstate
Learning rate	$0.0025t^{-1/2}$
Epsilon	0.0
Symmetric sampling	False
Tuple	tuple 1
Mono	0.0

Table 4: parameters

after 10000 test, we have no choice but take the simplest model: forget rate at 0.0 without any exploration.

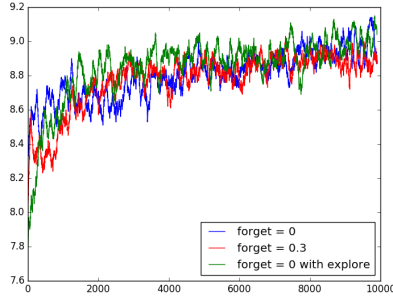


Figure 7: performance

Epsilon	0.0,0.0,0.01-t/2000
Forget rate	0.0,0.3,0.0
Policy	Afterstate
Learning rate	$0.0025t^{-1/2}$
Symmetric sampling	False
Tuple	tuple 1
Mono	0.0

Table 5: parameters

### 3.5 Symmetric sampling

See Figure 8 for illustration. We see that random sampling actually performs worse than a simple model. This may caused by its symmetric training which in a way, encourage exploration other than exploitation. A good sign is that it's continuously improving it's performance despite the average low score. It may outperform the simple model in the long run. However, here we won't use symmetric sampling anymore.

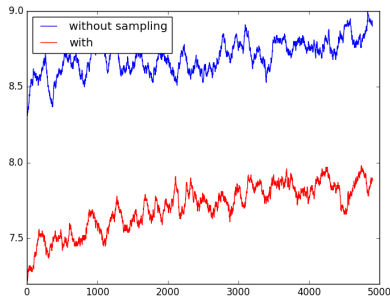


Figure 8: performance

Symmetric sampling	True	False
Epsilon	0.0	0.0
Forget rate	0.0	0.0
Policy	Afterstate	Afterstate
Learning rate	$0.0025t^{-1/2}$	$0.0025t^{-1/2}$
Tuple	tuple 1	tuple 1
Mono	0.0	0.0

Table 6: parameters

### 3.6 Monotony

See Figure 9 for illustration. Clearly, the monotony bonus did not help the agent to learn. This is very counter-intuitive, but experiment speaks for itself. This extra-layer of bonus may perturbed the well studied update function system, or it's causing other problem by prioritize monotony.

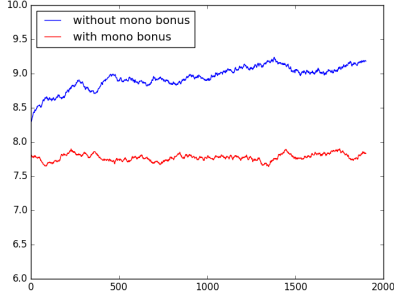


Figure 9: performance

Mono	0.0	0.0025
Epsilon	0.0	0.0
Forget rate	0.0	0.0
Policy	Afterstate	Afterstate
Learning rate	$0.0025t^{-1/2}$	$0.0025t^{-1/2}$
Symmetric sampling	False	False
Tuple	tuple 1	tuple 1

Table 7: parameters

### 3.7 Learning rate

See Figure 10 for illustration. The 0.0025 learning rate clearly outperforms 0.005. We didn't have results for learning rate linearly decreasing through time, because such configuration often cause bug and halt the system. Plus we figure that, the system needs a certain forget rate to be able to converge to the good strategy in the end.

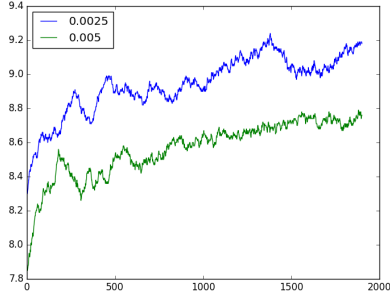


Figure 10: performance

Learning rate	$0.0025t^{-1/2}$	$0.005t^{-1/2}$
Mono	0.0	0.0
Epsilon	0.0	0.0
Forget rate	0.0	0.0
Policy	Afterstate	Afterstate
Symmetric sampling	False	False
Tuple	tuple 1	tuple 1

Table 8: parameters

## 4 Final result and discussion

Figure 11 is the final result that we get from the train 50000 times. Our agent is able to get 1024 for 50% of the time, and it's still improving (We are training our model for another 100000 games to see the result, we will update once it's finished).

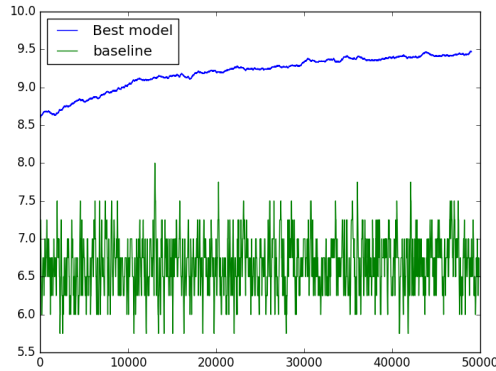


Figure 11: Best model

One obstacle that's prevent the game from getting 2048 is its the monotony problem. See Figure 12 for some result of testing. The agent is clearly stuck in the local minimal of not seeking

monotony on this left side. In some experiments, we can see that the agent even start with a monotony pattern and voluntarily choose to be in this kind of arrangement. This configuration is more probable to be reached, and a monotone configuration will not have any advantage over it before the game passes 512. It's very hard for the system to plan its strategy to this far.

A long learning time may improve this defect because optimal 1024 configuration will propagate inversely through time to encourage current state. But the probability is very low, so that this may demand a lot of training. Another idea is to add some extra features to encourage monotony, but surprisingly our approach did not work.

Another idea comes from observing the training evolution and finding that the score fluctuates through time, often the agent is been able to get 1024 for a row, but at one point it drops back to 512 or even 256 and stay there for several episode before it starts to shoot 1024 again. We can also see the fluctuation from the figures that we showed before, the performance is very unstable even though we've average the score on 100 episodes. The guess is that, high scores falsely give some irrelevant tuple (tuples with small numbers) a big reward, thus make then behave bizarrely in other circumstances. One way to improve this problem is to discriminate tuples by only rewarding tuples with the biggest value. However, we didn't had time to test this one.

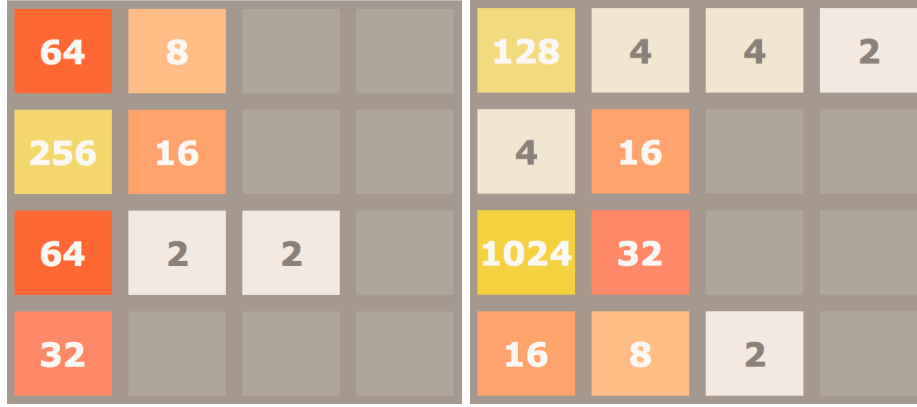


Figure 12: Some result of experiences

## 5 Annexe

Please feel free to test our python files by launching in command line. See [13](#) for the options available. If you want to see our AI play, just tap the last line on the figure.

```
JINdeMacBook-Air:2048 jinwenyao$ python3 puzzle.py -h
Usage: puzzle.py [options]

Options:
  -h, --help            show this help message and exit
  -p POLICY, --policy=POLICY
                        0 for random, 1 for qlearning, 2 for afterstate
                        [default: 2]
  -g TD_LAMBDA, --TD=TD_LAMBDA
                        TD_lambda the forget coefficient [default: 0]
  -a ALPHA, --alpha=ALPHA
                        alpha the learning rate [default: 0.0025]
  -t TRAIN, --train=TRAIN
                        number of training episodes [default: 2000]
  -s, --symmetric        symmetric sampling [default: False]
  -e EPSILON, --epsilon=EPSILON
                        epsilon the exploration rate [default: 0]
  -u TUPLE, --tuple=TUPLE
                        the tuple to use [default: 0]
  -c CONTINUES, --continue=CONTINUES
                        the file to continue training
  -d, --display          display result
  -m MONO, --mono=MONO  bonus for monotonicity [default: 0]
  -v, --verbose          print training steps for the first episode
JINdeMacBook-Air:2048 jinwenyao$ python3 puzzle.py -d -v
```

Figure 13: Command line options



## References

- [SZU14] Marcin SZUBERT. Temporal difference learning of n-tuple networks for the game 2048. *Computational Intelligence and Games (CIG), 2014 IEEE Conference*, pages 1–8, 2014.
- [YEH16] Kun-Hao YEH. Multi-stage temporal difference learning for 2048-like games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2016.