Solving the game of 2048 with Artificial Intelligence

Final Project: CS541 Artificial Intelligence

By :
Amee Sankhesara
Deepa Varghese
Devyani Shrivastava
Harie Vashini Dhamodharasamy Kalpana
Shweta Korulkar
Vinaya D Bhat

I. Abstract:

The 2048 game is a single-player game released by Italian programmer Gabriele Cirulli that quickly rose to fame because of it's deceptively easy seeming game play. Although it is easy to play, people cannot win the game easily, as they cannot expect where the next tile will come up. In this project, we have implemented the Minimax Algorithm, the Expectimax Algorithm and Reinforcement Learning to create agents that can master the game. For each algorithm, we explain the details of it's implementations and the results observed. This is followed by a Comparison study between the three techniques used.

II. Introduction:

The game is played over a 4x4 board, that initially has a tile of value 2. The existing tiles on the board can be moved in a singular direction either up, down, left or right. With each movement, three things occur, the tiles stop when they either collide with another tile or the border grid, If two tiles of the same number collide while moving, they will merge into a new tile bearing their summation, and a new tile pops up at an unexpected empty slot of value 2 or 4. The game is played as a series of such movements. When there is no empty cell and no more valid move, the game ends. If a tile with number 2048 appears in the board before the game ends, the player wins.

In this project, we attempted to create three different agents that use one of the algorithms: Minimax, Expectimax, Reinforcements Q Learning algorithm.

PEAS

- Performance Measure:Tile with the maximum value on the board
- Environment: 2048 game board of 16 squares in a 4x4 configuration.
- Actuators: Virtual keyboard events 'Up', 'Down', 'Left', 'Right'.
- Sensors: Interaction with game logic.

Environment's Nature: Full Observable, Stochastic, Static, Single Agent, Sequential, Discrete.

III. Minimax Algorithm

The 2048 minimax algorithm has the initial state as 2 values that are randomly generated (value 2 with probability 90% and value 4 with probability 10%) and placed in randomly selected cells. The next move is done by the Player AI who is trying to maximize the score. Player AI can either slide the cells 'Up', 'Down','Left' and 'Right'. Following the Player AI's move is the Computer AI move who is trying to minimize the score. The Computer AI is trying to minimize the score by placing a randomly generated value (value 2 with probability 90% and value 4 with probability 10%) in a random selected cell.

(The game starts off with the initial state, the Player AI who is trying to maximize the score should choose the next move which would lead to a terminal state with highest value.
To do this the Player AI considers all four possible moves possible as shown in figure. For each possible Player AI move, the Computer AI tries to minimize the score in the next step by randomly generating a value(2 or 4) and placing it in cell (consider all such possible cells). This process continues until a certain depth is reached or timeout occurs. Take the states or boards at that depth as the terminal states and calculate the heuristic values of the terminal states. This is a Player AI state and hence we end up choosing a state with maximum heuristic value (call it alpha) and start propagating backwards. The next state in backpropagation is the Computer AI move and gives the value of beta which is the state with minimum heuristic. This process continues till we reach the first Player AI which started this process.)
So the Player AI gets to choose the next move which it thinks would lead to the highest heuristic terminal state. After the Player AI chooses the next move, The computer AI will randomly put a value (2 or 4) in a randomly selected cell in an attempt to obstruct the player AI(trying to minimize the score) . The Player AI has to move next and this process continues until time is up or no moves are possible that is the board is filled with no possible next move. The time limit for each Player AI move was taken to be 0.2 seconds. The Player AI must choose the next move within 0.2 seconds otherwise the game will be over. In each of the 10 cases, timeup did not occur and game ended due to no possible moves left.

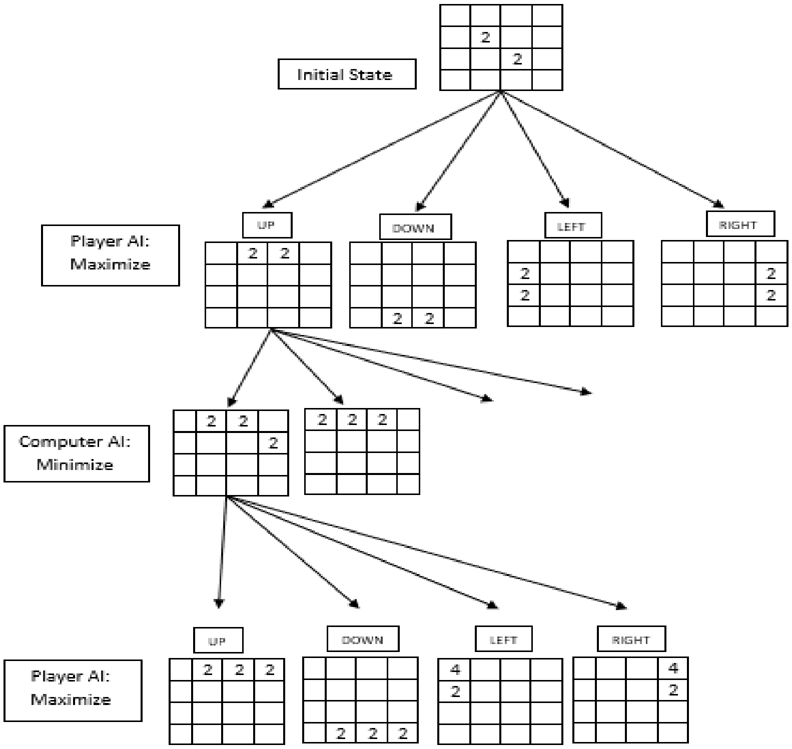


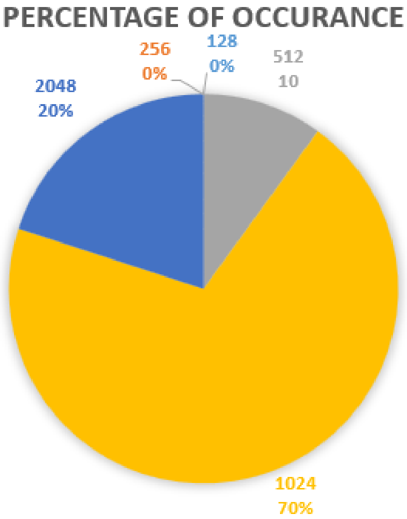
Figure 2: Minimax search tree for 2048

The terminal states will have the utility value which is based on heuristics. The utility function used in the minimax algorithm is the weighted sum of the following heuristics:

- Monotonicity of board: This heuristic tries to ensure that the values of the tiles are all either increasing or decreasing along both the left/right and up/down directions. This heuristic alone captures the intuition that higher valued tiles should be clustered in a corner. It will typically prevent smaller valued tiles from getting orphaned and will keep the board very organized, with smaller tiles cascading in and filling up into the larger tiles.
- Smoothness of board : This heuristic alone tends to create structures in which adjacent tiles are decreasing in value, but of course in order to merge, adjacent tiles need to be the same value. Therefore, the smoothness heuristic just measures the value difference between neighboring tiles, trying to minimize this count.
- Emptiness of board : There is a penalty for having too few free tiles, since options can quickly run out when the game board gets too cramped.
- Maximum four tiles of board : In this heuristic all the 16 values of board will be stored in array and array will be sorted and max 4 values from sorted array will be returned.

Without alpha-beta pruning:

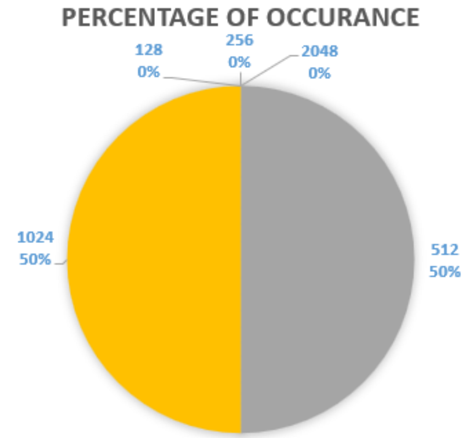
When the Player AI has to choose the next move, it forward propagates to a certain depth and then chooses the next move which it thinks would lead to the final state with highest heuristic value ultimately. Without alpha-beta pruning the minimax algorithm will consider all states at the cutoff depth and then choose the next move after back propagation.
The following results were obtained:
Number of times played: 10



With alpha-beta pruning:

The main goal is to decrease the number of nodes to evaluate by comparing alpha and beta values and “pruning” sub trees. With alpha-beta pruning the minimax algorithm will calculate the alpha value for Player AI states (initially alpha = -infinity for every state) and beta value for Computer AI states (initially beta=infinity for every state). Once the alpha becomes greater than

or equal to beta for a particular state, the downward subtree will be pruned. This way the number of states considered by the Player AI to select the next move would be reduced.The following results were obtained:
Number of times played: 10



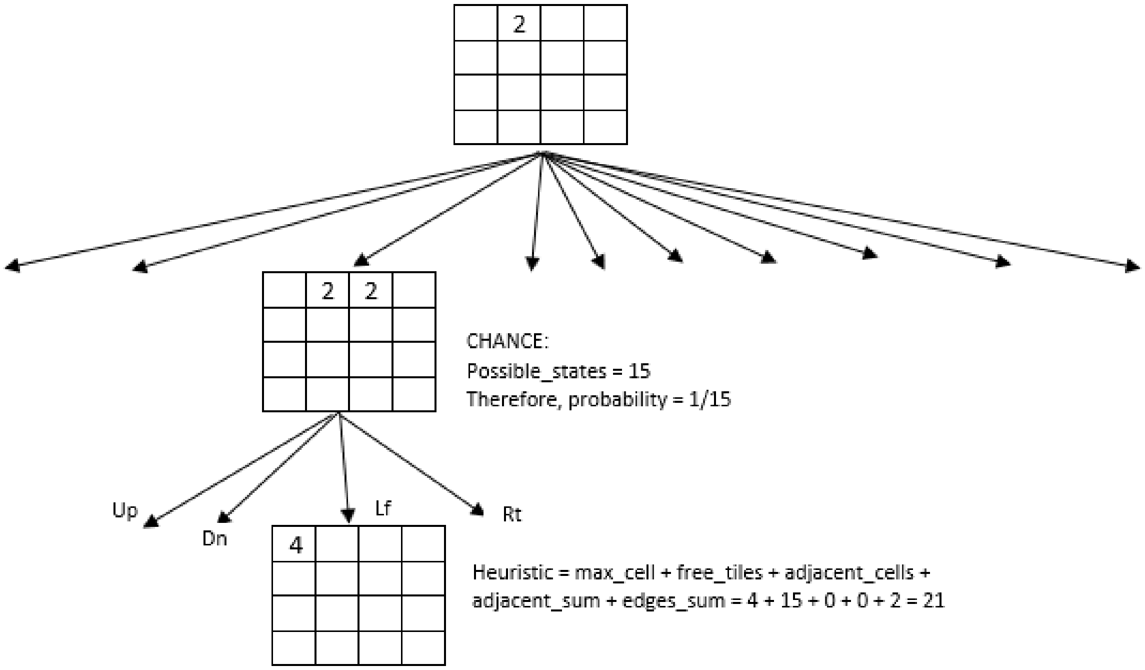
Results:

Comparing the results, here are few observations are made: the Player AI is playing optimally but the computer AI is not optimal and has some amount of randomness associated with it. The Player AI chooses the next move assuming that the computer AI is playing optimally that is the Player AI assumes the worst possible move that can be done by the computer AI. During the computer AI's turn its going to chose its move randomly (placing 2 or 4 randomly). Ideally, if both the players are playing optimally pruning should not affect the result. But since one player is playing optimally and another playing has some amount of randomness associated to it we are getting different results with pruning and without pruning.

IV. Expectimax Algorithm

The Expectimax algorithm is a variation of the minimax algorithm, except that we add chance nodes between agent nodes and opponent nodes. Minimax search tree classifies two player as maximizer and minimizer, where maximizer tries to maximize the overall utility of the game, while minimizer does the opposite. A value is associated with each state of the game. The difference in expectimax search is that instead of the minimizer trying to minimize the overall utility of the game, the “chance” node considers a turn of an unpredictable nature, and the value associated to the “chance” node is the probability that the said occurence will occur. The algorithm searches all possible states of the game for every possible chance occurrence given a certain depth, and evaluate how good the state is using heuristic functions. After getting the value, the algorithm carry the value up through the branches, and evaluate the value of each “chance” node, and finally the agent, which is represented as a “max” node, will choose the action to take so as to go down the route of the highest valued chance node's branch.

In our 2048 problem, we define the “max” node as the artificial intelligent agent, while the “chance” node as the game itself, which randomly place a newly generated tile of value 2 on any empty grid on the board after each move. The agent first simulate four action that it could take resulting in four different states. For each such state, it considers the “chance” of a 2 appearing in the remaining empty spaces. For each “chance” node we evaluate its value by considering the 4 possible actions that can be taken that results in 4 new states. In our expectimax search, we only consider the case that a new tile with value 2 is generated, we ignore the case of value 4 tiles, because it occurs at a low frequency. This assumption reduces the out branch of each “chance” node by half, dramatically improving the searching speed.



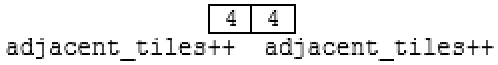
CHANCE:
Possible_states = 15
Therefore, probability = 1/15

Heuristic = max_cell + free_tiles + adjacent_cells + adjacent_sum + edges_sum = 4 + 15 + 0 + 0 + 2 = 21

Figure 2: expectimax search tree for 2048

Heuristics:

To calculate the evaluation function, apart from the existing heuristics of smoothness of board, maximum tile, number of free tiles (emptiness of the board), we added a new approach where, we also calculated the sum of how many tiles directly adjacent to an equal tile. For simplicity, these are all counted twice since we perform the calculations for each tile. We found that the existing heuristics, even when put together might rate two different states as having the same evaluation, although by mere observation we can see that one is better than the other. This heuristic ensures that if two tiles of the same value are adjacent to each other, then they weigh in the overall value of the state. Thus, for states where such pairs of tiles are higher, their overall value is higher as well, to indicate that these states are more desirable, compared to some other state that has a similar heuristic value. The sum of adjacent tiles is added to the evaluation function. To illustrate,

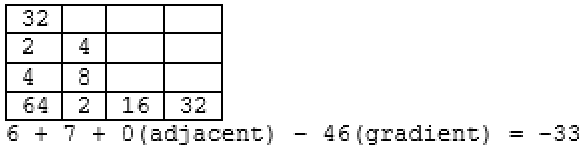


4	4

```
adjacent_tiles++  adjacent_tiles++
```

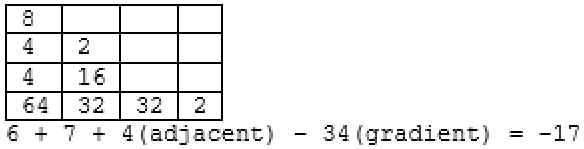
So states are better when they have more adjacent tiles. Combined with the smoothness of board function, this heuristic gives a better idea of the value of the state.

Consider the example:



32			
2	4		
4	8		
64	2	16	32

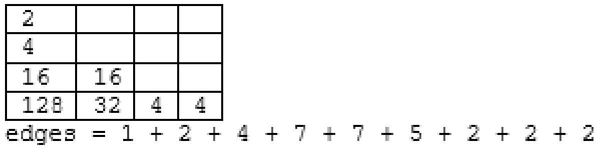
```
6 + 7 + 0(adjacent) - 46(gradient) = -33
```



8			
4	2		
4	16		
64	32	32	2

```
6 + 7 + 4(adjacent) - 34(gradient) = -17
```

The experience with this game will tell you the best strategy is to place the largest tile in the corner and keep other tiles on the edge surrounding it. Thus we modified the monotonicity heuristic so as to push the tiles to either of the four corners. This is done by simply adding the log of tiles on each edge which has the effect of counting the corner twice, so having higher valued tiles at the corner gives a higher value of the board. Thus encouraging the tiles the maximum valued tiles to collect at the edges.



2			
4			
16	16		
128	32	4	4

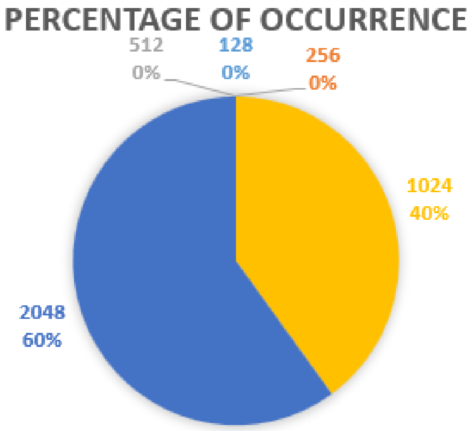
```
edges = 1 + 2 + 4 + 7 + 7 + 5 + 2 + 2 + 2
```

Thus, the evaluation function in Expectimax, returns a weighted sum of edge related heuristic, smoothness of the board, emptiness of the board, maximum tile and the sum of adjacent tiles of equal value. After experimenting with a few different weights for each heuristic, we found a set of balanced weights that resulted in an agent that won at the game most of the time.

Results:

From 10 trials, our agent reached the 2048 tile 60% of the time at depth of 2, and it always got at least 1024 (40%). The highest score (heuristic value) achieved was 45,722. The detailed results are in the chart below. Before we attempted to optimize the heuristic, the AI reached 2048 maybe

5-10% of the time – so it definitely improved quite a bit, especially by being able to reach 4096 occasionally.



V. Reinforcement Learning

Reinforcement learning is used in applications where there is no single correct approach to solve the problem. Teaching computers to play games is a complex learning problem that mostly depends on the game complexity and representational complexity, which has recently seen increased attention towards this field. The approach is based on Q learning. The design involves the use of neural networks as the function approximation method. Like most deep Q learning models, the heart of any reinforcement learning agent is the reward function. In this paper reward functions are designed to train the model to learn the best playing moves. But in 2048 there are 4 random components, that is, initial configuration of the game, addition of random tiles after very move, exploration of the agent and unavailability of moves.This approach is based on prioritized experience replay where the model trains to learn best game-playing strategy from the experiences collected.

Q-Learning

Q learning is the fundamental algorithm in the field of reinforcement learning which works by maximizing the expected value of the total payoff. In Q learning, the agent tries to learn the optimal policy from its history of playing the same game previously. The history is the experiences from the previous runs which is stored in the format (S_0,A_0,R_0 , S_1) where (S_0 indicates the state in which the agent was in and A_0represents the action that was taken ,R_0 is the reward it had received for taking the action, while S_1 is the state reached due to A_0.The agent's goal, roughly speaking, is to maximize the total amount of reward it receives in the long run.
In Q-learning we define a function Q*(s,a) representing the discounted future reward when we perform action 'a' in state 's', and continue optimally from that point on.

$$Q^* (s_t, a_t) = max_\pi \; R_{t+1}$$

Rewrite as the Bellman Equation:

$$Q^*(s,a) = r + \gamma max_a Q^*(s' , a')$$

If we have Q*(s, a) then:

$$\pi\ (s)\ = \pi^*(s) = argmax_a Q^*(s,a)$$

However, we do not know Q*(s,a); therefore we must estimate it with a non-optimal function Q(s,a). This enables us to define Q*(s,a) as

$$Q^*(s,a) = max \quad Q^\pi(s , a)$$

The whole idea behind Q-learning is that the Bellman equation can be used iteratively to improve our approximation of the optimal Q-function.It tells us that the maximum future reward is the reward the agent received for entering the current state *s* plus the maximum future reward for the next state *s′*. The gist of Q-learning is that we can iteratively approximate $Q*$ using the Bellman equation described above. The Q-learning equation is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

where *α* is the learning rate that controls how much the difference between previous and new Q value is considered and is $\gamma$the discount factor and satisfies $0 \leq \gamma \geq 1$. As the agent explore more and more of the environment, the approximated Q values will start to converge to $Q*$.

Deep Neural Networks

Deep learning is a computer software that mimics the network of neurons in a brain. Deep learning uses deep neural networks, the word deep means the network join neurons in more than two layers. Deep learning algorithms are constructed with connected layer.The first layer is called the Input Layer, This layer literally inputs information for the neural network to process. The last layer is called the Output Layer, This layer simply brings together the information from the last hidden layer of the network to output all the information needed for the program. All layers in between are called Hidden Layers, These layers do all the processing for neural networks. Each hidden layer is composed of neurons and the neurons are connected to each other. A neuron processes and propagates the input signal it receives from previous layer to the next layer.Weights and biases are the learnable parameters of the deep learning models. These neurons(nodes) receive information from the previous layer’s nodes, multiply it by weight and then add a bias to it.The strength of the signal given the neuron in the next layer depends on the weight, bias and activation function. Weights are initialized stochastically, or randomly. Using feedback from the environment, the neural network can use the difference between its expected reward and the ground-truth reward to adjust its weights and improve its interpretation of state-action pairs.

$$y = \sum_i w_i x_i + b$$

Network consumes large amounts of input data and operates them through multiple layers.Each layer represents a deeper level of knowledge, i.e., the hierarchy of knowledge. The network can learn increasingly complex features of the data at each layer.

Here, we have used neural network to learn Q value for each action in the specific state.The input is first flattened and then we implement a neural network with 4 layers of dense and three LeakyReLu layers. The four dense hidden layers are of sizes 128,32,8 and 1. The input is a vector with size 16 (4*4), which represents all tiles value, the output is a vector of size 4, and each is associated with Q value after taking 4 different actions on this state.

In the first dense hidden layer, we use ReLu, h : x− > (x) + activation function. The activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input.

Activation Function:

The rectified linear (ReLU) activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero. We have used LeakyRelu which is a variant of ReLU. Instead of being 0 when z<0, a leaky ReLU allows a small, non-zero, constant gradient α(Normally, α=0.01). Here,the three LeakyReLu layers are provided with a value of 0.001.

There is one dropout layer in between which is provided with rate 0.125. Dropout is used to set a fraction rate of input units to 0 (drop the input units) at each update during training time,which helps prevent overfitting.

Optimizer:

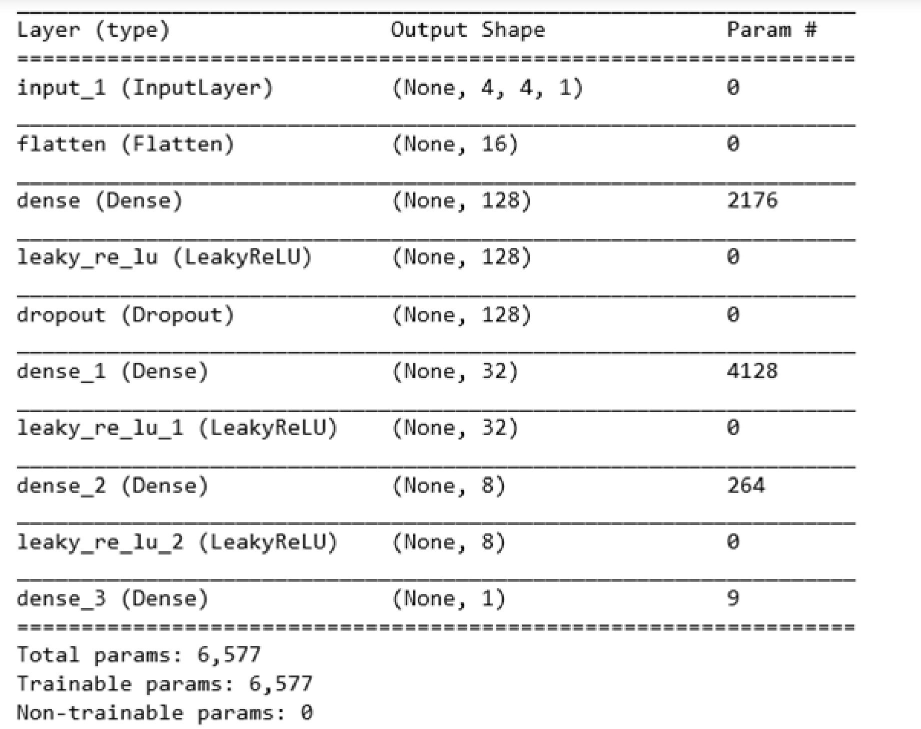
We have used RMSprop optimizer, this optimizer restricts the oscillations in the vertical direction.Therefore, we can increase our learning rate and our algorithm could take larger steps in the horizontal direction converging faster. The value of momentum is usually set to 0.9, we have used mean_squared_error for loss function.

The output values from the last layer is used to decide the action to be taken, All four Q value outputs are compared and the maximum value is chosen. It is possible that the best move cannot change the state, in that case we will use the second max output value as our action to execute. If all four moves cannot change game state, game is over.

For the back propagation part, we use Q learning formulation to calculate the new Q value after each action.

$Q(s, a) = (1 - \alpha)\, Q(s, a) + \alpha\, (R(s) + \gamma \max(Q(s, a))$

Network Architecture:



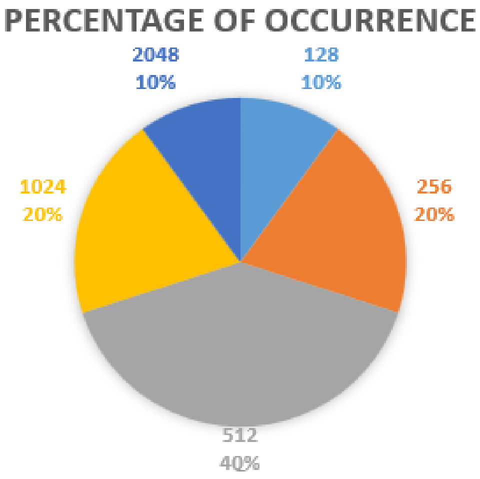
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 4, 4, 1)           0
_________________________________________________________________
flatten (Flatten)            (None, 16)                0
_________________________________________________________________
dense (Dense)                (None, 128)               2176
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 128)               0
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                4128
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32)                0
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 264
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 8)                 0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 9
=================================================================
Total params: 6,577
Trainable params: 6,577
Non-trainable params: 0
```

Reward function:

Reward function is critical in this method, because that is used to decide the better action. Therefore, all possible move at that state are done and the rewards are calculated. Then we mean-center and change value in to a range in [−50, 50], therefore we can give a positive value to a good action and give a negative value to a poor action. Reward function is defined as the value for all merges after taking action.

Results:

For reinforcement learning, using deep q-network, our agent achieved the max-tile of 2048 only once, on running the program 10 times.Our agent learned to make the tiles of 256 and 512 very easily, during this phase the agent played the game optimally by having largest tile in the corner. We believe the reason the agent mastered this strategy was due to availability of large number of free tiles due to which random move done at this stage is recoverable in terms of strategy learnt, but as the agent progressed to make tiles of higher value like 1024 and 2048 it struggled. This is because there were very few free tiles available. For a trial of 10 runs, we achieved the max tile of 512 for maximum number of times.The detailed results are given as below:



VI. Comparative Analysis

After 10 trials for each approach, we compared the above methods from multiple aspects

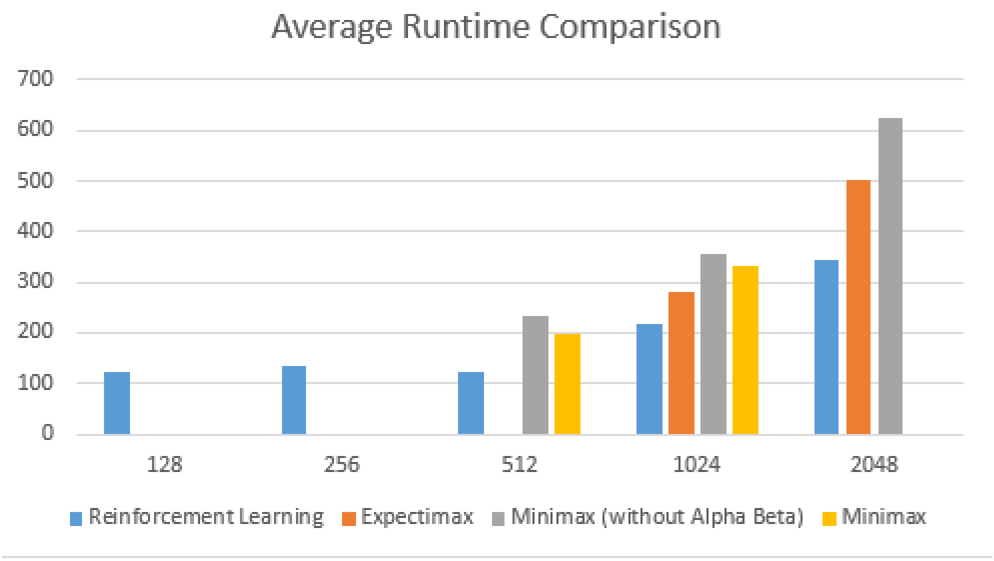
- The Maximum Valued Tile achieved by each algorithm
- Execution time for the milestones, for each algorithm

Expectimax reaches the final goal of getting the maximum tile of 2048, maximum number of times as compared to the other two. The least valued tile that it gets is 1024. Therefore, the chances that Expectimax will win are the highest.

But, if we compare the run times of each algorithm for the milestones such as 128, 256, 512, 1024 and 2048, Reinforcement agent takes the least amount of time.

Minimax (with or without alpha beta pruning) tends to take the most amount of time compared to the other two. But it has always managed to give at least a maximum tile of 1024.

NOTE: X-axis: Maximum tile achieved in a run
Y-axis: Average time taken to complete game, in seconds


Average Runtime Comparison
700
600
500
400
300
200
100
0
128
256
512
1024
2048
Reinforcement Learning
Expectimax
Minimax (without Alpha Beta)
Minimax

References

https://medium.com/@curiousily/solving-an-mdp-with-q-learning-from-scratch-deep-reinforcement-learning-for-hackers-part-1-45d1d360c120

https://cs.uwaterloo.ca/~mli/zalevine-dqn-2048.pdf

https://pdfs.semanticscholar.org/6f96/a3907eb44852dce15fc12d6931e8237dc2e2.pdf

http://cs229.stanford.edu/proj2016/report/NieHouAn-AIPlays2048-report.pdf

https://home.cse.ust.hk/~yqsong/teaching/comp3211/projects/2017Fall/G11.pdf

https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

https://medium.com/intro-to-artificial-intelligence/deep-learning-series-1-intro-to-deep-learning-abb1780ee20

https://www.guru99.com/deep-learning-tutorial.html

https://medium.com/datadriveninvestor/deep-reinforcement-learning-4149def586bb

https://github.com/ParkerD559/2048-Expectimax-AI

https://github.com/shahsahilj/2048

Artificial Intelligent: A modern approach Third Edition by Russel and Norvig.