Deep Q-Learning Network for Adaptive 2048

Rahul Krishnamoorthy
Department of Electrical and Computer Engineering
rkrishnamoorthy@ucdavis.edu

Vishal I B
Department of Electrical and Computer Engineering
vib@ucdavis.edu

Anirudh Ramchandran
Department of Electrical and Computer Engineering
aniramch@ucdavis.edu

Abstract

The paper proposes a reinforcement learning application to solve a variation of the traditional 2048 game called 'Adaptive 2048'.The adaptive algorithm aims to make the game harder and more challenging to obtain a perfect score of 2048 when compared to a regular 2048 game. A deep Q-learning agent was used to solve this problem since there was almost an infinite number of states, making it infeasible for the agent to solve the problem using just Q-learning. The Deep Q-learning agent was able to obtain a maximum score of 512 after training for a total of 5000 episodes.

1 Introduction

An area of machine learning where the agent gains an understanding of the suitable actions with respect to its environment to maximize its reward is known as reinforcement learning. Unlike supervised learning, where a labelled data set can be used to train the model, in reinforcement learning, the agent is not given the right actions to be taken. Hence the agent tries to discover the best action in every situation by testing them and then learning from the experience. Slowly, through experience the agent would learn to take actions that would yield the highest rewards, this forms the basis of reinforcement learning. An RL problem can be modelled as a Markov chain/Markov's decision process, where the actions taken from one state will cause the transition to a different state or the same state. The main objective of a reinforcement learning agent is to take different actions, visit every possible state and learn the best action. While in unsupervised learning , the learning agent would try to find similarities and differences between data points, reinforcement learning would aim at discovering suitable action sequences to maximize the cumulative rewards of the agent.(1; 3)

1.1 2048: The game

2048 is a single-player sliding block puzzle game where the goal is to slide and combine similar number tiles to reach a maximum sum of 2048 in a single tile(4).The game has four main actions that are right, left, up, and down, which are done using the player's arrow keys on the keyboard. Upon making a move, a new tile will spawn/appear randomly in one of the empty spots on the grid; this new tile could have the numbers 2 or 4. As mentioned, there are four actions, and the tiles slide as far as possible in the chosen direction, they are only stopped either by the grid border or by other tiles.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

If two tiles of the same number collide while moving, they are merged into a tile that will hold the sum of the two tiles. Now in order to add this resultant tile, we would have to strike into another tile of the same number. After a move, a new tile would appear in an empty space, and thus the game would continue.This describes the game-play of a normal or the original 2048 game. We looked into a harder version of the game, known as adaptive 2048.

1.2 Q learning: The basics

Q learning is an off-policy reinforcement learning algorithm that pursues to find the best action to take given a current state.(2) We consider this an off-policy algorithm because it learns actions outside the current policy. Q learning revolves around the notion of updating the Q values or the action-value functions, and this value denotes the value of the performing action 'a' in a given state 's.' The update rule shown below forms the core principle of the Q learning algorithm.

$$Q(S, A) \leftarrow Q(S, A) \ + \ \alpha[R \ + \gamma \ max_a \ Q(S', a) - Q(S, A)] \tag{1}$$

The Q values are iteratively updated for each state-action pair using the bellman's equation until it converges to the Q', the optimal Q value

1.3 Contribution of team-members

- **Rahul Krishnamoorthy:** Designing the state space representation and reward distribution.Designing the adaptive 2048 environment. Designing and training the Neural Network. Tuning and improving the model once validation results were obtained.
- **Vishal I B:** Designing the state space representation and reward distribution.Designing the adaptive 2048 environment. Designing and training the Neural Network. Tuning and improving the model once validation results were obtained.
- **Anirudh Ramchandran:** Designing the adaptive algorithm for the 2048 environment and Training the network and further testing the game-play with the learned policies and recording the observations.

2 Deep Q-learning

Q learning basically creates a cheat sheet for our agent that helps the agent with what actions to perform. While Q leaning may do a great job in small state spaces, its performance may drop considerably in more complex state spaces. In Q learning, the Q values are stored in tables and accessed, but as the number of states increases, it becomes very difficult and to store a large number of states in tables. This is difficult due to the fact that storing large numbers of Q values in memory intensive and also searching such large tables to find the Q values is computationally very expensive. This is when approximate networks come into the picture. Thus, rather than using value iteration to iteratively calculate the Q values we use a function approximator to find out the optimal Q values. This function approximation can be done using an artificial neural network, and the method of combining both reinforcement learning and deep learning is known as deep reinforcement learning.

Deep Q-Networks are basically neural networks that approximate the Q-value functions. They take in the current state as the input and output the action-state values for all the actions in the action space. The representation of the states varies depending on the application and the input requirement of the neural network. The network specifications also depend on the representation of the states. The network learns the features and hence the representation of the states. Different types of neural networks with different architectures can be used depending on the representation of the states. For example, states represented as images might require a Convolutional Neural Network in order to learn the features in the state representation.There are certain steps involved in the construction and training of a Deep Q Network. The first step is to decide on the representation of the states that would be fed to the network, and the type of network to be used. Once the state representation is done, the neural network is constructed and initialized with random weights.

The next step is the initialization of experience replay memory. At every time step in the episode, the current state of the environment is given as input to the network and the Q values for every action are approximated after forward pass through the network, further, the loss is calculated and the weights of the network are updated through back-propagation of the gradients. The network is trained for

only one epoch at every time step and this might cause the network to over-fit to certain states since there is a high variance in the distribution of the states. To overcome this problem, a buffer memory known as Experience Replay memory is used to train the network with multiple examples at each step. The buffer stores the agent's experience in memory. It stores the current state, action performed, the resulting reward obtained, and the next state. Hence at every time step, the network is trained on a sample of experiences from the replay memory.

After initializing the experience replay memory, the following steps are performed based off our understanding from (5) and these are done iteratively for every step of the episode:

- The current state is initialized and the action is selected in an epsilon-greedy manner, i.e., with a probability 'epsilon', an action is chosen from the action space randomly and with a probability '1-epsilon', an action is chosen from the action space with the maximum action-value. This helps in dealing with the trade-off between exploration and exploitation.
- Once the action is selected, it is executed in the environment and the resulting reward and the next state are observed. These values are now stored in the experience relay buffer.

$$e_t = (s_t,\ a_t,\ r_{t+1},\ s_{t+1}) \tag{2}$$

- After storing the experience in the replay memory, a random batch of experiences have to be sampled from the memory. Experience in the replay memory consists of the current state, the action that was taken, the resulting reward from the action and the next stage of the game after the action was taken.
- Now, since neural networks are good approximators of any function, it is used to evaluate the optimal policy for a particular environment for an agent. The neural network which makes an approximation of this policy is called a "policy network". The policy network thereby approximates the action-state value function to choose the optimal action given the current state to move to the next state. The state that is given as the input to this neural network undergoes forward propagation as in any neural network. Each layer in the network has a non-linearity introduced by it and thereby it extracts the features and passes it forward to the next layer. In the output layer of this neural network, the action-state values for the given state as the input to the neural networks are given as the output. The loss is calculated at this point in the training and the various optimizers like stochastic gradient descent or Adam optimizer or RMSprop can update the weights in the neural network to minimize the loss.

$$loss = (R\ +\ \gamma\ \max_{a'}\ Q(S', a')\ -\ Q(S,\ a))^2 \tag{3}$$

where,

$$target = R\ +\ \gamma\ \max_{a'}\ Q(S', a') \tag{4}$$

$$prediction = Q(S,\ a) \tag{5}$$

The loss is calculated by comparing the Q value in the output with the corresponding optimal Q value or target Q value for the same action. So the Q value evaluated by the neural network is subtracted from the optimal Q value evaluated by the bellman equation for the same state-action pair. But while calculating the optimal Q value using the Bellman equation it is also necessary to calculate a term in the Bellman equation which involves the next state and action pair. Usually, this can be done just by looking at the Q matrix but since the number of states is almost infinite it is not possible to do so. With deep Q learning, the next state is now forward propagated through the network to evaluate the Q value function for all the actions in the action space. This means that it is required to do two forward passes in order to calculate the loss for any state. Over several iterations, this will ultimately approximate the optimal Q value function.

There are totally two passes through the neural network to approximate the optimal Q value. But the target Q value is unknown at the beginning. But still, the network is used to approximate this target Q value. This process requires a second pass through the network and since it passes through the same network, it uses the same exact weights that have already been used for getting the Q value that has to be approximated in the first place. The problem here is that when the first two passes occur in the network, the loss function is evaluated using the Bellman equation and the optimizer minimizes

the loss function. The loss function gets minimized during each pass by adjusting the weights in each layer of the neural network. When the weights get adjusted in the network, it means that the loss has been lowered than the previous pass but since the weights were adjusted, the target Q value changes during every two passes of the network too. As the network keeps minimizing the loss based on the target values by adjusting the weights, the target values keep shifting to a different value and the model might never converge to the optimal policy. The predicted Q values will move towards the target Q values but the target Q values will also begin to move for every two passes in the same direction and the model might never converge causing instability.e

In order to avoid this problem of the neural network chasing its own tail, a new network called the target network is used to find the target state-action values. The target network is basically a clone of the policy network with the same weights of the policy network frozen. The target network's weights are updated from the policy network every certain number of time steps and are not done during every iteration which is also a hyper-parameter(6). So the first pass occurs through the policy network but the second pass, however, occurs with the target network. So these two values are used to update the Bellman equation in order to approximate the optimal Q value for any state.

3 Implementation

The implementation of a reinforcement learning problem involves the creation of an agent and an environment. The environment is the problem on which the agent is going to work on which will provide the states necessary for the agents to perform the action and when the actions are performed, it provides a reward for that action after moving the environment to the next state. The agent in a reinforcement learning problem is the one trying to solve a particular problem introduced by the environment

3.1 Environment

The environment for a reinforcement learning model simulates the moves and enables the agent to switch from one state to another based on its actions thereby, providing a reward. For a game of 2048, the environment consists of information on what the state would look like at a point and what it would look like after an action has been made and what would be the reward on reaching the new state with the action made by the agent.

3.1.1 Representation of states

The states were represented by a 2D matrix as shown in the Figure 1 in which each element in the game's grid is represented by the corresponding entity in the matrix. The 2048 game consists of 4 rows and 4 columns and that was the shape of the matrix that was created to represent the states as well.

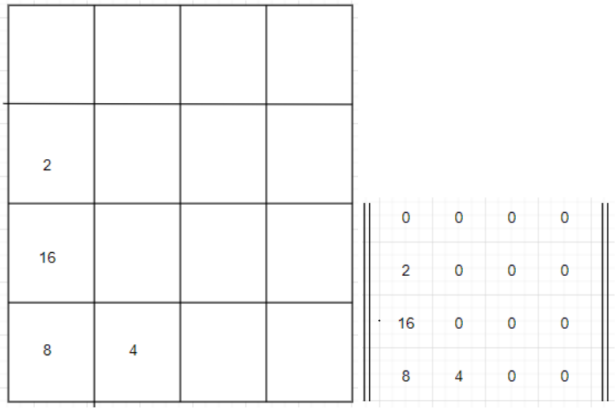

Figure 1: State Representation in Grid and Matrix format

From the image, it can be seen that whenever there is a particular number in the game's grid, there is the same number at the same location in the matrix as well. One has to identify how to represent the empty cells in the grid. Substituting a zero in the empty spaces was the best choice as to when we move from one state to another after an action, similar cells in the grid which are adjacent, needed to be added up. Since adding a zero does not affect the total sum, it was much easier to handle zeros in the empty cells than any other number.

The game starts with a two or a four placed randomly at any two cells in the grid. There should be an equal probability of choosing any two cells in the grid on which a two or four may appear. But even though the cells in which the first two entities appear may be random, it is necessary to choose whether a two or a four appears in those cells. This choice is made at random too but the probability of choosing a two was kept at a much higher value than for getting a four. A two appears as the first two new numbers 90% of the times while a four appears only 10% of the times. This was done to not let the player have an upper hand in the first move itself where the person might start with two fours which can be combined to get an eight easily and is one step closer to victory just by chance.

3.1.2 Action space

After the first state, the action made by the agent moves the game from one state to another. The action space was provided for the agent to choose among an action from left, up, right or down. Left was assigned a number 0, up was assigned 1, right was assigned a 2 and down was assigned a number of 3. Whenever an agent chooses any of these numbers the corresponding action was performed which moves the environment from one state to another. When two similar numbers are adjacent to one another or if they are separated by zeros, they can be combined together and the result will be the addition of two numbers. In order for this to happen the action should be in such a way that it should bring these two similar elements together. An action does not just enable two similar cells to combine, but it also pushes the cells towards the wall along which the action is oriented. So, an action would move cells towards a particular wall and in doing so, if there were similar elements next to one another, it would add them both and replace the one nearer to the wall towards which the action was oriented as the sum. If there was an element next to it, it would bring it one step closer to the wall even for zeros. But for zeros, an additional zero should be added at the opposite wall signifying an empty cell. Figure 2 represents an example of an action being made in the game. A left action by an agent/player resulted in combining two 2's and two 16's to give the next state with a reward of 36.

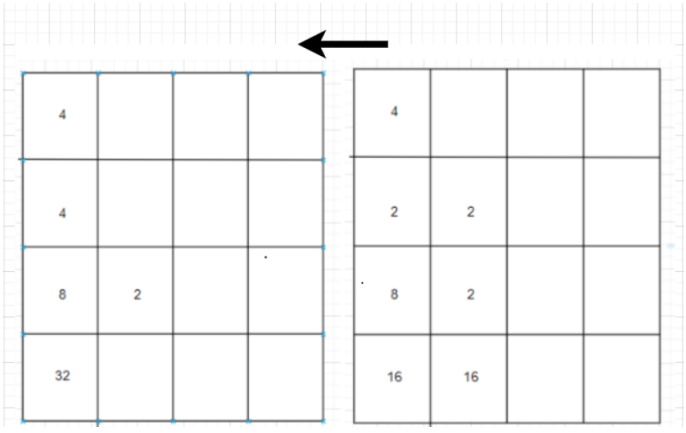

Figure 2: Sample action in a game of 2048

When two numbers combine, there is going to be one less number/cell in the game and when multiple numbers combine, at some point there may not be any more cells to combine. This is the reason why in the game of 2048, a new number which maybe a 2 or a 4 is introduced every time the grid moves to a new state. The new number can appear anywhere in the grid and just like the initial two numbers, there is a higher chance (90%) of obtaining a two rather than a four.

3.1.3 Allocating rewards

One of the most important purposes of the environment is to provide rewards for each action taken by the agent. This is one of the most important functions of the environment as it would guide the agent to learn what is important to learn from the environment and what is not. The reward supplied by the environment acts as a guide to the reinforcement learning agent. The bellman equation to update the state value function consists of various elements but one of the most important factors is the reward. The reward influences the Bellman equation to obtain the action-value function and thus it helps to learn the optimal action to take given a state. The rewards were given in such a way that it encourages the agent to act in such a way that it combines two similar cells to get a cell of higher value. A higher reward was given for combining cells with a larger value than for a lower value. This enables the agent to combine similar cells more and more until it reaches 2048. The sum of all new cells that were formed on combination action was given as the reward. This means that when two 2s combine and when two 4s combine on performing a particular action, the total reward is the sum of the new entities that were obtained from this combination which is 12. Also, when an action resulted in a new state in which no cells combined, a reward of 0 was provided by the environment. This was done to encourage the agent to combine the cells more. When the game ends with the agent obtaining 2048 in the environment, then a reward of 2048 is given as the reward. But it was also important for the environment to penalize when the agent loses the game with no more possible moves. This was done by providing a reward of -1 by the environment. On training the agent with these conditions, the agent should be able to understand the basic goal of the game and hopefully will learn more subtle nuances and strategies in order to win the game. These are the actual rules of the game but we have designed an adaptive mode for this game which makes the game even harder.

3.1.4 Adaptive 2048

The main strategy by which people play the game is to place the largest elements along one of the corners. This is done because these large numbers are of no use when combining with the smaller numbers say, a 2 or a 4 which are the numbers that always appear in a new random cell. So the strategy is to combine the smaller elements at the center of the grid and when the numbers become bigger, they can then be combined with the larger elements in the corner and kept there. With this strategy, it is very easy for an agent to win so we made the environment adapt in such a way that it does not allow an agent to proceed with this strategy. The environment is designed to have these attributes that disrupt this strategy:

- The algorithm does not create a new element in a random spot anymore. It now looks for places that are adjacent to an already existing element to drop the new element. This is done to restrict the freedom of movement of the cells which a player might exploit to move the cells around to a more comfortable position for them to combine.
- Now that these cells can only appear adjacent to already existing elements, it looks for all the elements in the grid where an adjacent spot is available to create a new element. The algorithm now finds the largest number among these adjacent elements and drops a 2 near the largest element it can find. But it is important to find if the largest number is greater than 4. If not, the algorithm is actually helping the player by putting similar elements near each other. So it draws a 2 near the largest element if it is greater than 4.
- If the largest adjacent element is a 4, it drops a 2 next to it and if the largest number is a 2, then it drops a 4 next to it.
- Another important feature is that the algorithm completes rows and columns. What this means is that whenever a row or column needs only one element for it to be filled, it fills that row or the column. Since the strategy of the player is to play along a wall or a corner, by completing rows or columns, the player is forced to move the largest element away from the wall and thereby exposing it. This enables a 2 or 4 to appear next to it and thereby disrupting the player's entire strategy.

One important thing to consider is the preference of these features. Some of these features should have a higher preference to occur when the opportunity shows itself to disrupt the strategy. So the first preference is to always fill the columns or rows as this option occurs the least number of times but it completely disrupts the game play. The next preference is to always drop the new element next

to the largest element considering whether the largest number is greater than or lesser than 4. On doing all these steps, an adaptive algorithm or 2048 was implemented.

3.2 Agent

Now that the environment has been created, the agent has to be configured so that it can start playing the game and learn a strategy to win it. The agent that was created for this purpose devised a deep Q learning algorithm to solve this problem. With an understanding of the deep Q learning algorithm that was explained previously, the deep Q learning agent was created.

3.2.1 Policy network

The first step in creating the deep Q learning model was to create the policy network, which would approximate the optimal state-action values. A convolutional neural network (CNN) was designed in order to perform this operation. CNN's have a shared weight architecture and also possess translational invariance. The agent needs an excellent approximator to approximate the Q values, and a CNN does a very good job of approximating based on the loss function. We used the Keras library in python to design the neural network.The input for Keras is expected in a specific manner. The input must be in the form of a 3D matrix, with the first dimension being the number of channels for the input data, which is 1 for this case. The second and the third dimension of this matrix should represent the rows and columns of the input data, which is 4x4 in this case. So the input data has the shape of 1x4x4, which was given as input to the neural network. The states were converted into the above-mentioned shape before giving it as an input to the neural network. (7)

The first convolution layer had 32 filters with the kernel size being 2x2 and having a stride length of 1. A reLu activation function was used in the first layer. A reLu activation function provides the non-linear function that is required to obtain the features which can then be given to the next layer, also enabling the gradient to move faster during gradient descent. The second layer has the same activation function as before, but this time it has 64 filters but with the same kernel size and stride length. After passing through two convolution layers, the data is then flattened so that it can be sent into a fully connected layer. The fully connected layer had a size of 256, which means that it would give 256 size features after passing through it. The dense layer had a reLu activation function too. The feature vector that was obtained from here was then sent into another dense layer, which acts as the output layer with a reLu activation function. Since the action space had only 4 elements signifying each possible action, the neural network gave 4 outputs, which means that there is a Q value corresponding to each action in the action space. Whenever a neural network is used, it needs a loss function to minimize or maximize in order to reach the optimal output. The loss function that has to be minimized here is the mean square error between the Q values given by the policy network and the target values given by the target network. Another loss function is known as "Huber loss" can also be used. Huber eliminates any dramatic change in the difference calculated. These dramatic changes often hurt the RL agent. Huber loss clips the difference if it's above a certain threshold value. Due to certain library version errors and lack of availability of resources, the Huber was not implemented but is a part of the future work for this project. The model was created as shown in Figure 3 (8)

3.2.2 Target network

The disadvantages of using the same network to evaluate the target Q value, which was mentioned in the previous section, called for the creation of a separate target network. The target network has the same parameters as that of the policy network when it is created initially. It has the same design and shares the same initial weights. The weights get updated after a specific point of time.

The target network has to update its weights occasionally but not during every single pass through the network like the policy network. So after the replay phase which is the phase in which a batch of values was sampled from the replay memory and then trained upon, the target network's weights get updated. The target network's weights are updated in a way that its weights are not exactly identical to the policy network and it does not entirely retain its old weights. The algorithm calculates a weighted average of both of these weights using a parameter called 'tau' to update the new weights. When the value of 'tau' becomes 1, the policy network weights are cloned exactly and when it is 0, the old target network weights are retained. A tau value of 0.1 was chosen for the agent to update the target network's weights.

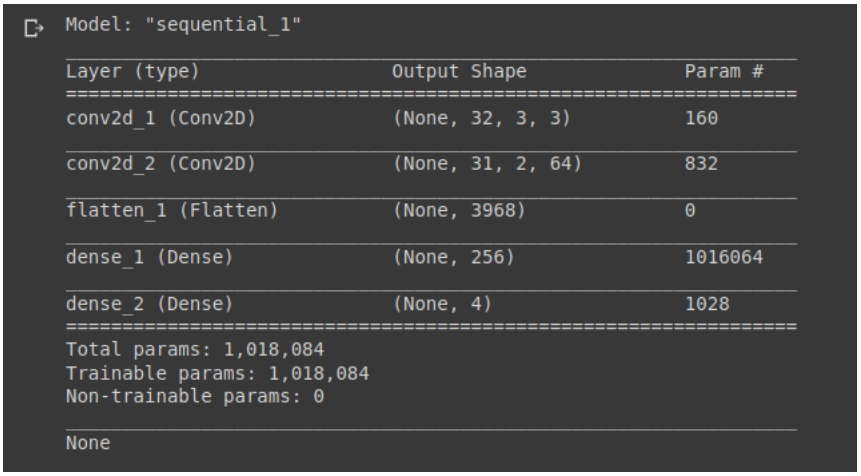

Figure 3: Model summary

3.2.3 Epsilon decay

It is essential for the agent to explore as much as it exploits what it has learned. It is crucial to exploit to move to a meaningful state where it can then explore further. It is also vital for the agent to explore because it introduces newer states to the agent where it can learn more. The agent was designed to explore at the beginning, but as the time- steps proceed, it tends to exploit more. It is important to explore in the beginning as it enables the agent to learn what the optimal action would be at the start of the game. When it learns this, it can move much deeper into the game. This is done using epsilon decay. The epsilon value starts at 1, and after each time step, the epsilon value gets multiplied by the decay constant, which is 0.99. This enables the epsilon value to go lower after each time step and thereby enabling the agent to exploit more. But it is important to set a minimum value of epsilon too as, after a few 100 time steps, the epsilon value may get so low that the agent always exploits, and thus it may not learn anything new. The minimum value of epsilon was set to be 0.05. When the agent explores, it chooses an action at random, and when the agent exploits, the agent chooses the action corresponding to the maximum Q value that is given by the neural network approximator. The epsilon value is reset to 1 after every episode and the decay starts from the beginning. This part of the algorithm provides the details to the agent on how to act at each point in time of the execution of the code.

3.2.4 Hyperparameters

During the training of the model, the model took really long to train, which means that there were a lot of time steps in an episode. It was hence approximated to be a continuous task rather than an episodic task with a gamma value of 0.9. The learning rate enables the model to tell how large the step should be during the process of gradient descent. If the learning rate is too large, it means that the model may not be able to find the global minima, and if it is too large, it will take very long to train. Considering these trade-offs, a learning rate of 0.005 was applied to the training. An RMSprop optimizer is used to train the policy network. RMSprop scales the learning rate so that it go through the saddle point faster.

3.2.5 Experience Replay

Once the agent acts based on the Q value it obtains from the policy network, or if it chooses an action at random, it moves from one state to another, scoring a reward. So all these parameters are stored in a vector and sent into the replay memory. A replay memory acts as a chunk of memory that stores all the states that have previously occurred, along with the actions that were taken and the resulting states and rewards and also the information on whether there is still a possible move available after reaching the new state. The experience replay was set to a maximum length of 2000 which means that it can store 2000 experiences (current state, action, reward, next state, Done). The network does not train until the memory is filled up to at least the size of the batch size. And once the length of the memory

is greater than or equal to the batch size, it samples 'batch size' number of samples from the memory and trains on them. But once the memory is full, no more experiences are added to the memory and the network samples from the same pool of experiences to train the network. This might not expose the model to more recent experiences. So, in order to get a new set of experiences, the memory was cleared after it reached 2000 experiences and loaded with new and more recent experiences.(8)

After it stores all this information, the algorithm then moves to the learning part. In all the previous steps, it was discussed how an agent acts given the state, but after this step, it is essential for the neural network to learn from these steps. The neural network does this learning part in batches. The batch size that was chosen for this purpose was 50. This means that whenever the replay memory's size is lesser than the batch size, it continues to perform actions and records these values, but not train the network. But when the replay memory's size goes beyond 50, the agent samples random samples equal to the batch size from the replay memory and trains the network. Each sample has its own state, which is passed through the target network, which then gives the target Q value to which the policy network has to approximate its values, which are taken as the maximum of the Q values obtained by it. If the game has ended, the agent updates the target values to be equal to the reward obtained but if the game is not yet over, the target for the action is taken as the gamma multiplied by the target Q added by the reward. Even though it is given that mean square error between the target Q values and the policy generated Q values is the loss function to be minimized, the target values are obtained from the equation mentioned above, which is the Bellman equation. So indirectly, it is the Bellman equation that is getting optimized in this neural network by minimizing the mean square error between target and policy generated Q values. This process is done for each and every sample sampled from the replay memory.(9)

Once all this is done, the agent replaces the next state reached on the environment via the actions as the current state and then passes this state through its algorithm to find the optimal action. This process gets repeated until the episode ends which is signaled by a parameter called 'done'. It gets updated each time after moving to a new state and gets stored in the replay memory as well. This process discussed in this section elaborates on what happens during each time step in an episode. Once the end of an episode is reached, the agent repeats the same process over several iterations till satisfactory results are obtained from the agent.

4 Results

The agent was allowed to play the adaptive version of 2048 games for thousands of episodes. This was done so that the agent explores the game more and more during each iteration and learns the strategy to play the game. The agent played the game for a total of 5000 episodes. Over the 5000 episodes that the agent played, it secured the scores as shown in Table 1.

Table 1: Distribution of max scores obtained over a span of 5000 episodes

Score Obtained	Number of occurrences	Percentages of occurrences
512	5	0.1%
256	574	11.48%
128	3921	78.42%
64	4909	98.18%
32	5000	100%
16	5000	100%
8	5000	100%
4	5000	100%

For a few times, we continued to use the maximum value obtained from the policy network to decide the action the agent should perform. On doing this, we were able to observe that the game gets stuck at a single point where the agent performs a sub optimal action and gets stuck there forever. This process of getting stuck at one part of the game can occur even if the agent performs one sub optimal action which makes the next state the same as the current state. The agent will continue to perform this action forever because this is the action with the maximum action-value. So instead of using

'argmax' all the time, the agent was asked to perform epsilon greedy action even during testing. But the epsilon value was not kept at a high value, it was kept as low as 0.05 in the test case. This was done because a deterministic policy like an 'argmax' function may get stuck at a sub optimal local optima. Sometimes it is necessary to move away from the local optima to reach the global optima and an epsilon greedy function enables the agent to do this. It enables the non-deterministic policy to break out of the infinite loop to move towards the global optima.

It can be seen from Table 1 that the agent could not possibly reach the max score of 2048. The agent, however, was able to reach 512 a couple of times and was able to reach a max score of 256 and 128 during the majority of its episodes. With the adaptive algorithm in place, it places a new block every time and eventually surrounds the largest number in the grid and hence does not allow it to combine further and hence discourage combining larger values. It is possible to reach 2048 only when large numbers combine but since this was heavily restricted by the algorithm, it couldn't possibly reach the max score. Our team members played multiple games of 2048 with the adaptive algorithm in place and could never possibly reach the maximum score. We were able to reach a max score of 1024 a couple of games but most games ended with a high score of 512. Considering that only a score of 1024 was reached with 'human-level' intelligence, it makes sense that the reinforcement learning agent could only reach a max score of 512. Another important restriction to improvement was a lack of computation power. The model continued to train for days but could only reach 5000 iterations and we believe that the model could perform much better if it was able to run for more than 10000 iterations.

5 Future work

These are some of the implementations that can be imposed in the future to improve the model. We believe that these changes can have a positive impact on the reinforcement learning model and it would be interesting to continue the work beyond towards these directions.

- **Huber loss:** Huber loss can be used instead of mean square error since Huber will reduce any dramatic changes in the error. Due to library version errors, Huber loss was not implemented.
- **More iterations:** Due to computational constraints, the number of iterations was limited to 5000 episodes, but ideally, a Deep Q network requires tens of thousands of iterations for the Q values to converge to a decent level.
- **Deeper networks:** Deeper networks help in learning deeper underlying features and might speed up the learning.
- **More exploration:** The more number of states the agent visits, the higher chances of it finding the optimal action-value estimates.
- **Prioritized experience replay:** Samples of experiences are randomly sampled from the memory buffer. But the samples that are already close to the target should be given less priority than the ones with a larger error. The samples must be ranked according to the error calculated and selected by the rank.
- **Dueling DQN:** The objective of Dueling DQN is to reduce the effect of states that are not valuable. It uses two estimators to learn which states are valuable. This can avoid the cost of learning the effect of actions on that state.

6 Conclusion

Thus, a reinforcement learning algorithm called Deep Q-learning was used to solve a difficult version of the game called 'Adaptive 2048'. After the agent has trained for a total of 5000 episodes in the adaptive 2048 environment, it was able to score a maximum score of 512, 0.08% of the times.

References

[1] Bajaj Check,Prateek,and Prateek Bajaj.(Dec 2019) “Reinforcement Learning.” GeeksforGeeks, www.geeksforgeeks.org/what-is-reinforcement-learning/.

[2] Sutton,Richard S,and Andrew G Barto,(2018). Reinforcement learning": An introduction. MIT press.

[3] Bhatt.S,(2019),Reinforcement Learning 101 Retrieved from https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

[4] 2048 Video Game,(March 2020) Wikipedia, Wikimedia Foundation, en.wikipedia.org/wiki/2048_(video_game)

[5] “Training a Deep Q-Network-Reinforcement Learning.” Deepliazrd,deeplizard.com/learn/video/0bt0SjbS3xc.

[6] Cappiello, Antonio.,(25 Feb. 2020), “Physics Control Tasks with Deep Reinforcement Learning.” Paperspace Blog, Paperspace Blog, blog.paperspace.com/physics-control-tasks-with-deep-reinforcement-learning/.

[7] Muriuki, G. (2018, August 25). Deep Learning; Personal Notes Part 1 Lesson 3: CNN theory; Convolutional filters, Max pooling... Retrieved from https://towardsdatascience.com/deep-learning-personal-notes-part-1-lesson-3-cnn-theory-convolutional-filters-max-pooling-dbe68114848e

[8] Levine, Zachariah. "Learning 2048 with Deep Reinforcement Learning."

[9] Dedieu, Antoine, and Jonathan Amar.,(2017), "Deep reinforcement learning for 2048." Conference on Neural Information Processing Systems (NIPS),, Long Beach, CA, USA.

Appendices

The code contains two classes (for the environment and the agent), the main function and a test function. The functions implemented by our team are marked with ' ###### TEAM CODE' beside the function definitions.

The code was run on Google Colab, ipython notebook which had most of the libraries pre-installed. The training the agent took us upward of 33 hours for 1000 episodes. The 'trials' variable in the main function can be altered to change the number of episodes tested.

The following are the steps to be followed in order to run the program:

- **Step 1:** Copy the source code that we have attached to the ipython notebook. And run the cell.
- **Step 2:** You will see that the training has started and the number of episodes starts printing along with the number of moves made in every episode. The training might take multiple hours depending on the number of episodes.
- **Step3:** Once the training is done, run the following codes in two different cells to mount the drive and load your model from the drive.

  For mounting the drive:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

  For loading the model:

```
from keras.models import load_model
model_save_name = 'mod.h5'  #your model name here
                  #for our code mod-(trial number).h5 is the model name
path = F"/content/gdrive/My Drive/{model_save_name}"
model = load_model(path)
```

- **Step 4:**Call the __test__(model) function from another cell with the name of your loaded model. The number of episodes can be modified to the desired number of episodes. This will start printing the number of episodes. Once all the episodes are done, the max tile, the mean number of moves and the mean score are printed.

  >Max tile - This is the maximum value reached by the game.
  >Mean score - This is the average score from all the episodes.
  >Mean moves - This is the average of the number of moves made by the agent in all the episodes.