SOLVING THE N-PUZZLE USING DEEP REINFORCEMENT LEARNING

By

Dudley Spence

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfilment of the requirements

for the degree of

MASTER OF SCIENCE

15 September 2021

Abstract

The software discussed in the following dissertation was designed to act as an agent that uses deep reinforcement learning to learn to play and complete the famous N-puzzle. Due to the N-puzzle having such a significant state space with the 8-puzzle having 10^{13} solvable states, simple reinforcement techniques such as value iteration would not be a viable option. Therefore, I chose to use a Deep Q-Network, with the project's focus being on solving the 8-puzzle and the 15-puzzle.

The network can solve most of the 8-puzzle combinations, failing on only a small percentage of the higher difficulty puzzles. The network can also solve some of the lower difficulty 15-puzzles. However, the network is designed so that with enough computing power, it is fully capable of learning to play N-puzzles of a larger scale, such as the 24-puzzle. Once trained, the software can then take permutations of the puzzle as input and depending on the degree of difficulty of the puzzle, will provide the user with a set of step-by-step instructions to reach the solution.

The heuristic Manhattan distance and the instances of linear conflicts are integral parts of the reward system used by the network during training. They reward behaviours that both solve the puzzle and optimise the number of actions taken to reach the solution. The network uses revolutionary training methods that are at the forefront of artificial intelligence research, such as experience replay memory which is one of DeepMind's most promising discoveries.

The software is coded using python and uses a training loop script that controls the interaction between the environment, the deep q-network/agent and the experience replay memory. The TensorFlow library is used to create the deep neural network model and allows training parameters such as the weights to be saved and loaded. This allows the network to be efficiently utilised for solving puzzles inputted by a user without any need to retrain the network. Evaluation of the network was limited to the average number of puzzles solved at each approximate starting difficulty.

There is no clear way to accurately calculate the optimum number of actions to solve each puzzle; therefore, the optimality of the solutions produced by the network could not easily be evaluated.

Since the development of the deep q-learning algorithm in 2013, there have been constant attempts to apply the algorithm to a vast number of problems. Unlike other reinforcement learning approaches, I was only able to find one example of the N-puzzle being solved with a DQN and this approach gave no indication of state representation, or the reward system used, only the network architecture. Therefore, I can presume my project is unique as the only software that uses a DQN combined with the reward system and the environment state representation I designed.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

[illegible]

Signed

Dudley Spence

List of Figures

Figure 1: Example 8-Puzzle..8

Figure 2: Reinforcement Learning Decision Process 9

Figure 3: Tensorflow Network Architecture..18

Figure 4: Coded Training Loop..19

Figure 5: In-situ Evaluation..20

Figure 6: Example Output of Solver Script...22

Figure 7: 8-Puzzle Training Evaluation - Early Stage (left) & final (right)..............................24

Figure 8: 15-Puzzle Final Evaluation of Training 25

Figure 9: Example 8-puzzle..34

Figure 10: Diagram showing the software structure 35

Contents

Chapter 1. Introduction 8

Chapter 2. Background and review of literature..10

Chapter 3. Software Design..11

3.1. Methods and Knowledge used for the design...11

3.1.1. Q-Learning...11

3.1.2. The Deep Q-Network 12

3.1.3. Exploration and Exploitation...13

3.1.4. Experience Replay 13

3.1.5. Manhattan Distance and Linear Conflicts.......................................13

3.1.6. One-Hot Encoding..14

3.2. The Design 15

3.2.1. The environment...16

3.2.2. Reward System 16

3.2.3. The Experience Replay Buffer...17

3.2.4. The Deep Q-Network 17

3.2.5. Training Loop..19

3.2.6. The Puzzle Solver 21

3.3. Technologies used 22

3.4. Coding Standards 23

3.5. Design Ethics 23

Chapter 4. The Results **24**

Chapter 5. Future Developments **27**

5.1. Time and Computational Power 27

5.2. Prioritised Experience Replay.. 27

5.3. Double Deep Q-Network.. 28

5.4. Graphical User Interface 28

Chapter 6. Conclusions **29**

Bibliography **31**

Appendix 1: Installation Guide **32**

Appendix 2: Additional Design Detail **35**

Chapter 1. Introduction

Chapter 1 first provides a brief overview of the N-puzzle and some of the different approaches taken to reach the solution. This is then followed by an introduction to reinforcement learning and how this approach can tackle problems like the N-puzzle.

The N-Puzzle is a sliding tile game that takes place on a k * k grid with ((k * k) – 1) tiles each numbered from 1 to N. The goal is to slide the tiles around each other in a vertical or horizontal direction, using the vacant position to reposition the tiles into ascending order. When referring to the puzzle, I will discuss the 8-puzzle (figure 1) for simplicity; however, all the concepts can be applied to any size N-puzzles.

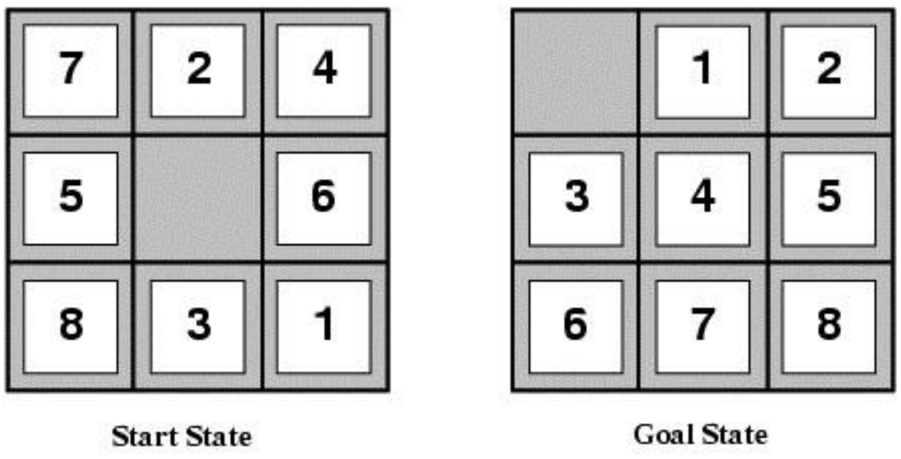


Figure 1: Example 8-Puzzle

There are many approaches to solving the N-puzzle. The most common is heuristic functions such as the A* search algorithm, which uses a combined heuristic using the Manhattan distance and the hamming distance. This is usually quite effective but can take many iterations to find a correct solution to the puzzle. Reinforcement Learning concerns how an agent, such as the player of a game, can learn a policy (strategy) that suggests taking actions in its environment to maximise cumulative reward. Unlike supervised learning, reinforcement learning doesn't require labelled data and instead relies on balancing exploration (uncharted territory) and exploitation.

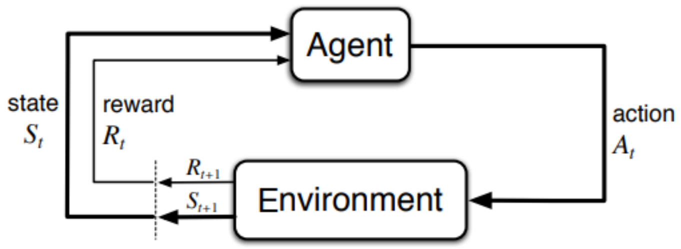


Figure 2: Reinforcement Learning Decision Process

Reinforcement learning has become an effective tool for solving Markov decision processes such as the N-puzzle, a problem in which the next state and the reward can be predicted using only the action that is taken and the information from the current state that summarise the whole history of the environment. Attempting to solve the N-puzzle can offer a unique insight into the potential and limits of reinforcement learning. Reinforcement learning algorithms can start with no knowledge or ability to solve a task, and under the right conditions, achieve performance far superior to that of a human. They can associate immediate actions with the long-term outcomes they produce, sometimes at the expense of immediate reward. This process of forward-thinking is vital to the development of complex strategies, often referred to as the network policy. (Sutton and Barto, 1998)

Deep reinforcement learning combines artificial neural networks with a reinforcement learning framework that helps software agents learn how to reach their goals. An agent is trained without knowing the rules of the game and is told only the reward corresponding to each state's actions. In this project, I aimed to use a deep q-network to train an agent to solve the N-Puzzle and then continue training to optimise the solutions produced by the agent.

The network can solve most permutations of the 8-puzzle and a smaller number of lower difficulty 15-puzzles. With more time and computing power, there is nothing to suggest the network would not be capable of training and successfully solving larger N-puzzles such as the 24-puzzle.

Chapter 2. Background and review of literature

Chapter 2 summarises the literature that was critical for finding the correct approach to solving the N-puzzle. Analysing previous attempts to solve the N-puzzle was a vital step in planning the project’s design, and later provided a comparison for evaluating my own project.

This project required research on previous attempts to solve the N-puzzle and understanding which approaches could be deemed effective. The most common approach to solving the puzzle is using search algorithms. A* search is regarded as the most effective of these approaches (Mathew and The Society of Digital Information and Wireless Communication, 2014). An alternative and even more effective approach is the genetic algorithms, which take inspiration from the genetic processes seen in biology and evolution. For example, exploration occurs via mutation, and exploitation occurs via genetic crossover of two parent genes (Shaban, Natheer Alkallak and Mohamad Sulaiman, 2010).

When searching for methods of solving the puzzle with reinforcement learning algorithms, of which there are few, I was able to find examples of 2 different approaches.

The work done by (Bischoff *et al.*, 2013) used a technique known as local value iteration and was considered effective at solving most permutations of the N-puzzle. Work by (Mehta, 2021) also used local value iteration for the 15-puzzle and found that for low difficulty games, the loss converges and achieves a 100% win rate. The medium and high difficulty games achieve about 43%- and 22%-win rates, respectively.

When searching for previous research into using the DQN algorithm for solving the N-puzzle, I was only able to find one example. The work done by (Agostinelli *et al.*, 2019) used a Deep Q-Network labelled DeepCubeA designed to solve the Rubik’s cube but could be easily adapted to learn to solve the N-puzzle. DeepCubeA was able to solve every test N-puzzle and found the shortest path to the goal 99.4% of the time for the 15-puzzle and 96.98% of the time for the 24-puzzle. It became apparent that DeepCubeA had far greater computing power than was available to me. This allowed the training of an extensive network architecture described as two layers with 5000 and 1000 nodes respectively, followed by four residual blocks, each containing two more hidden layers and each with 1000 nodes. Understanding the scale of DeepCubeA’s network architecture and training facilities was critical for managing the balance between computing power, network size and expectation.

Chapter 3. Software Design

3.1. Methods and Knowledge used for the design

This section will outline the deep reinforcement learning knowledge, methods and techniques used in the development of the software.

3.1.1. Q-Learning

The Q-learning algorithm attempts to determine the quality (Q) of a given action from a given state at maximising future rewards. The Q-learning agent navigates the environment calculating Q[S, A], which is the expected reward of being in state S, and taking action A. These Q-values are stored in a table and updated, and the Q-values will eventually converge as the Q-function becomes more accurate. The algorithm proceeds in discreet rounds.

In every round, t,

- An action is chosen greedily using the "estimated" Q-values

$$\boldsymbol{a_t = argmax_a Q(s_t, a)}$$

- Action $\boldsymbol{a_t}$ is taken giving an observed reward $\boldsymbol{r_t}$, next state $\boldsymbol{s_{t+1}}$
- Update Q-values for $\boldsymbol{s_t, a_t}$

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

The discount factor γ, essentially determines how much emphasis the agent puts on rewards in the future relative to the rewards in the immediate future. If γ equals zero, the agent is short-sighted and only learns about actions that produce an immediate reward.

If γ equals one, the agent evaluates each of its actions based on the sum of all its future rewards, which leads to decisions being made factoring in potentially irrelevant information too far in the future.

The learning rate α determines to what extent newly acquired information overwrites old information.

3.1.2. The Deep Q-Network

The DQN (Deep Q-Network) algorithm was designed by DeepMind in 2013 with the development of a DQN capable of playing several Atari games to a superhuman ability (Mnih *et al.*, 2013). The algorithm is an enhancement of the RL algorithm Q-Learning with a deep neural network.

To tackle problems such as the N-puzzle where the state space is far too large to store Q-values in a table, the Q-function must instead be approximated. This function approximation is made using a deep neural network. The state is used as input and is forward propagated. The network outputs the Q[S, A] values for all the possible actions. Like a regular neural network, the algorithm requires a loss function.

The loss function is the mean squared error of the target Q-values, calculated using the Bellman equation, minus the predicted Q-Value. Unlike in other deep learning methods where the target value is static and does not change, in reinforcement learning, the target does change and is regularly calculated. To regularly generate the target values, two versions of the network are used. One network is the function approximator called the policy network, and the other is the target network and has the same network architecture as the policy network but with frozen parameters.

Every k iterations, where k is a user-defined value, the network parameters of the policy network are copied to the target network.

$$\boldsymbol{Loss} = \left[\boldsymbol{r} + \boldsymbol{\gamma}\, \boldsymbol{max}_{\boldsymbol{a'}} \boldsymbol{Q}\left(\boldsymbol{s'}, \boldsymbol{a'}; \boldsymbol{\theta}_{\boldsymbol{i}}^{-1}\right) - \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a}; \boldsymbol{\theta}_{\boldsymbol{i}})\right]^{\mathbf{2}}$$

$$\boldsymbol{Loss} = (\boldsymbol{Target\ value} - \boldsymbol{Predicted\ value})^{\mathbf{2}}$$

For every input, the state S is provided to the policy network, and the next state S’ is provided to the target network. For the target network, the maximum Q-value is always chosen for the loss function and for the policy network, the Q-value is chosen that corresponds to the action chosen using the ε-greedy policy. Now that we have the loss function (shown above), the optimisation problem is the same as minimising the error function in any other neural network, through backpropagation.

3.1.3. Exploration and Exploitation

If the agent was to choose its actions based solely on the policy, it would never learn any complex strategies and would make all its actions based on only the immediate reward. This usually results in the agent selecting the same move repeatedly.

To avoid this premature convergence to a weak strategy, randomness must be introduced to allow the agent to explore the environment in order to develop a more complex strategy that considers the longer-term rewards associated with some actions. This process is called an ε-greedy policy. Epsilon is a user-defined probability of selecting a random action (exploration) rather than using the policy to select the action (exploitation).

3.1.4. Experience Replay

As humans, our best decisions aren't based only on the new experiences we have had, and instead are based on a combination of new and old experiences stored as episodic memories that can be learned from. For example, when learning to play a game such as tic-tac-toe, if a player loses, the strategy used for the following game should not be based solely on the previous game. Instead, the player should try to play a strategy that is a culmination of the experiences of all the games they have played. Similarly, for the DQN agent, experience replay acts as a data generating process, collecting experiences to learn from and storing them in a replay memory buffer of fixed size.

These experiences are stored as tuples containing < S, A, S', R, D> state, action, next-state, reward and done, where done is a Boolean value holding True if the experience resulted in the terminal state. These experiences slowly fill the replay memory buffer with the newest pushing out the oldest.

Each epoch, the agent samples a random batch of experiences and uses the batch as the training sample data. This random sampling avoids using data that is too correlated such as sequential experiences. (Fedus *et al.*, 2020)

3.1.5. Manhattan Distance and Linear Conflicts

The Manhattan distance (or the taxi-cab distance) is a heuristic value meaning it tells the algorithm which path will provide the solution as early as possible. The heuristic function for

Manhattan distance is computed by counting the number of moves along the grid that each tile is displaced from its goal position and summing these values over all tiles (excluding the vacant tile 0). This gives a rough estimate of the number of moves from the goal state. Mathematically, if the position of the tile is $x = (a, b)$ and the position of the same tile in the terminal state is $y = (c, d)$ then

$$\boldsymbol{Distance} = |\boldsymbol{a} - \boldsymbol{c}| + |\boldsymbol{b} - \boldsymbol{d}|$$

One of the drawbacks of relying on the Manhattan distance as an estimate of cost is its lack of consideration of linear conflicts in the board. Two tiles 'a' and 'b' are in a linear conflict if they are in the same row or column, their goal positions are in the same row or column, and the goal position of one of the tiles is blocked by the other tile in that row.

When linear conflicts occur, one tile must move out the way to allow the other to pass and then move back again. Therefore, for every linear conflict in the state, two is added to the distance value of the board.

3.1.6. One-Hot Encoding

When approaching a problem with machine learning, understanding what type of data is being used is critical for successful network predictions. Most input data will either be categorical or numerical. When considering the N-puzzle, it is important to remember that the numbers on each tile are simply labels and the value of the number has no relation to the tile it is on. Therefore, the N-puzzle input data can be considered categorical data rather than numerical data.

There are two methods for pre-processing categorical data, which are integer encoding and one-hot encoding. Integer encoding only works when the categories follow a natural order, such as small=1, medium=2, and large=3 meaning medium and large (2 and 3) can be considered more similar than small and large (1 and 3). However, when the categories don't follow a natural order, such as in a classification problem where blue=1, green=2 and yellow=3, if we inputted a yellow tile, then the network would assume a natural order and consider a classification of green to be more correct than blue when in fact both are equally incorrect.

Similar to the colours, there are many versions of the N-puzzle that have no numbers on the tiles and instead have image segments, with the goal being to produce the original image. Therefore, to avoid the data being treated as being integer encoded by the network, the data must be pre-processed using one-hot encoding.

One-hot encoding converts the categories into a binary format. Looking at the 1-D state format there are 8 tiles and one vacant position. Each tile will be represented by an array of length 8 made up of zeros and ones where the position of the one identifies the tile. For example [0, 1, 2, 3, 4, 5, 6, 7, 8] would first be represented as:

```
[[0, 0, 0, 0, 0, 0, 0, 0]
 [0, 1, 0, 0, 0, 0, 0, 0]
 [0, 0, 1, 0, 0, 0, 0, 0]
 [0, 0, 0, 1, 0, 0, 0, 0]
 [0, 0, 0, 0, 1, 0, 0, 0]
 [0, 0, 0, 0, 0, 1, 0, 0]
 [0, 0, 0, 0, 0, 0, 1, 0]
 [0, 0, 0, 0, 0, 0, 0, 1]]
```

Prior to being inputted to the network, this would then be reshaped into a one-dimensional array of shape (1, 72) or, for this example:

[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1].

3.2. The Design

The software design can be separated into two, the training software and the front-end software. The training part of the software consists of four main elements, the environment, the DQN, the replay buffer, and the training loop. The front end of the software consists of a puzzle-solving script that uses the trained network to solve requested puzzles. In this section, I will be discussing how each aspect of the software was designed and developed.

3.2.1. The environment

The environment script is responsible for generating new starting states, maintaining the current state, applying actions to the current state and calculating the reward of a Q(s, a). When formulating a DQN, our unique contribution is in choosing the reward structure and state representation.

How can a 2-dimensional puzzle board be represented in code? This was one of the first questions that were asked when designing the software. The environment is represented as a 1-dimensional array of numbers ranging from zero to N, where 0 represents the vacant position on the puzzle board. For example, for an 8-puzzle, the solved state would be [0, 1, 2, 3, 4, 5, 6, 7, 8].

How can starting puzzles be generated that are the desired difficulty and always be solvable?

Each puzzle state has a property distance which is computed as a combination of the Manhattan distance and the linear conflicts. When generating starting puzzles, random actions are taken until the distance of the board is greater than the user-defined parameter, difficulty. The puzzle is generated by working backwards from the goal state to prevent any puzzles from being generated that are not solvable. Depending on the selected action, a tile swaps index position with the vacant tile 0.

3.2.2. Reward System

When an action is taken, the reward of the resulting state is calculated and provided to the DQN. Initially, the reward system used was simply -9 and then +1 for every tile that is in its final position. However, this was reward system was ineffective, likely due to the constant requirement to move tiles out of their terminal position in order to increase the overall number of tiles in their terminal position.

The reward system needed to give a better estimate of the cost of the current state, and so heuristic functions were the most logical solution. I instead introduced the heuristic Manhattan distance as the reward system. After each action, the reward for the transition would be the negated Manhattan distance with an additional -1 for any actions that would not be possible on the physical board and result in the state remaining the same such as moving in the direction of the edge of the puzzle. This was effective for training puzzles of low difficulty;

however, once the difficulty reached ~10+, the training plateaued, and the agent was unable to solve the puzzles. This was likely due to the introduction of linear conflicts in the puzzles when the difficulty surpasses 10. Therefore, my final reward system also introduced an additional -2 for each linear conflict in a state. This showed immediate results and the training continued past difficulty 10.

3.2.3. The Experience Replay Buffer

The replay buffer is responsible for storing new experiences into the replay memory that has a user-defined capacity and will push old experiences out for new ones when the memory reaches maximum capacity. The second function of the replay memory is selecting a random batch of experiences from the buffer and pre-processing them for input to the DQN.

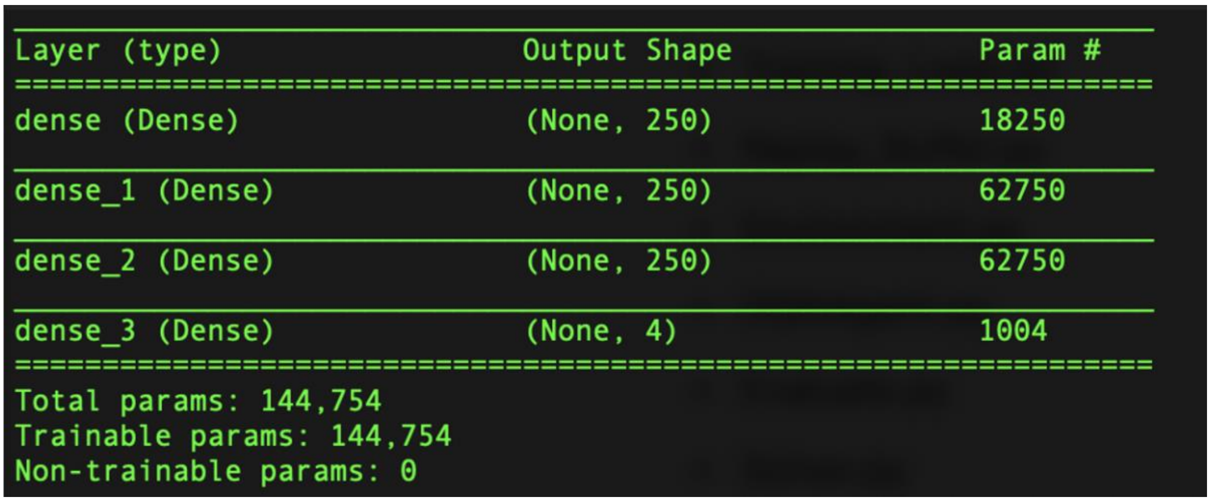
Initially, the replay buffer memory could hold 5000 experiences. During the training process, the agent would occasionally be making progress until suddenly it lost all the progress it was making as if it had forgotten what it had learned. This was due to a phenomenon known as catastrophic forgetfulness.

Catastrophic forgetfulness occurs when the size of the replay buffer memory is not large enough, and so the older experiences are frequently pushed out of the buffer (forgotten) by new experiences collected by the agent. Occasionally, experiences that were vital for learning a complex strategy will be forgotten, and the progress made by the agent will go backwards. This phenomenon restricts the agent to a cycle of learning simple strategies that are then forgotten and relearned.

This problem was immediately mitigated by increasing the buffer size from 5000 to 100,000 and maintaining a batch size of 100.

3.2.4. The Deep Q-Network

The DQN class has many user-defined hyperparameters; each can be altered for optimisation of the network and can adapt the network to learn to solve different puzzle sizes. Both the policy network and the target network have identical network architectures (figure 3).



```
______________________________________________________________________
Layer (type)                 Output Shape              Param #
======================================================================
dense (Dense)                (None, 250)               18250
______________________________________________________________________
dense_1 (Dense)              (None, 250)               62750
______________________________________________________________________
dense_2 (Dense)              (None, 250)               62750
______________________________________________________________________
dense_3 (Dense)              (None, 4)                 1004
======================================================================
Total params: 144,754
Trainable params: 144,754
Non-trainable params: 0
```

Figure 3: Tensorflow Network Architecture

The network has an input layer of N+1 nodes and three dense hidden layers, each with 250 nodes and finally an output layer of 4 nodes. This is a small network in comparison to the DeepCubeA network previously discussed.

The design of the network involved balancing both being large enough that it can handle more complex learning but also a small enough network that allows for extensive training of the network given the time and computing power available.

During the network design, several different activation functions were tested, and the Relu function consistently produced the best results. The DQN involved a pre-processing function capable of converting the 1-D input array to a one-hot encoded N(N+1) array made up of only zeros and ones.

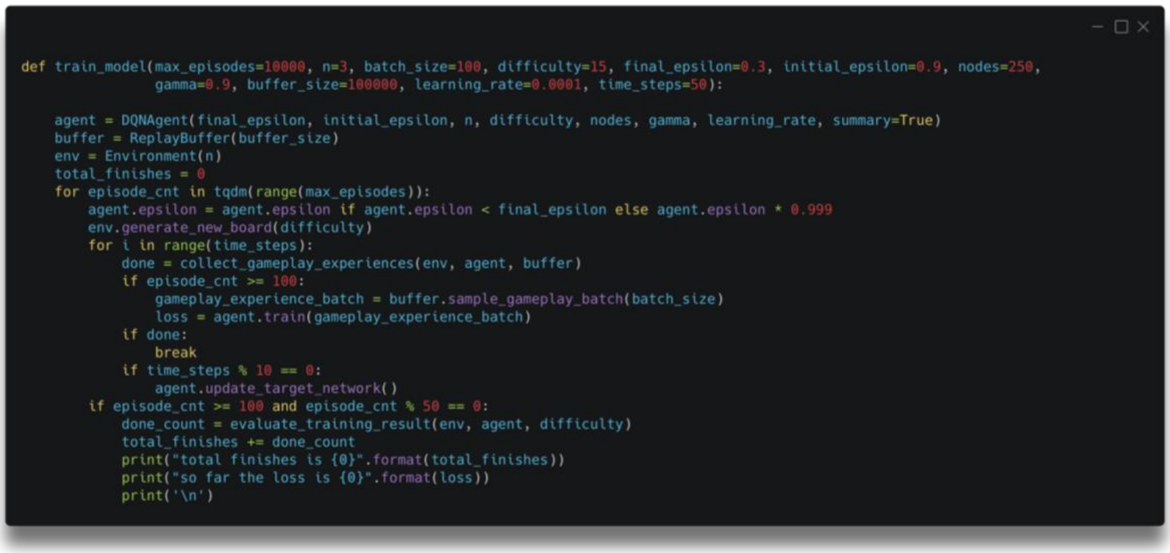
The training function of the DQN takes a sample batch of experiences as input and forward propagates the current states to the policy network and the next states to the target network. The loss is then calculated using the mean squared error function, and the policy network weights are updated using the TensorFlow Adam-optimization gradient decent.

When required, the target network parameters can be updated to match the policy network parameters. The discount factor is 0.9, meaning the network will consider the reward of actions several steps ahead.

3.2.5. Training Loop

The training loop follows the algorithm seen immediately below, followed by the coded version, figure 4, beneath that.

```
1. Initialise the replay memory (100,000 capacity)
2. Create an instance of the agent class
3. Create an instance of the environment class
4. For each episode:
    1. Generate a new starting state
    2. For each timestep:
        1. Select an action using exploration or exploitation.
        2. Execute the selected action in the environment.
        3. Observe the reward and next state
        4. Store this experience as a tuple in the replay memory. < S, A, S', R, D>.
        5. Sample a random batch from the replay memory.
        6. Pre-process the states from the batch.
        7. Pass the batch of states to the policy network.
        8. Calculate the loss between the output Q-values and the target Q-values
            ♦ This requires a second forward pass to the network using the next state
        9. Gradient descent updates weights in the policy network to minimise loss.
        10. Every ten timesteps, the target network is updated to the policy network
```



```python
def train_model(max_episodes=10000, n=3, batch_size=100, difficulty=15, final_epsilon=0.3, initial_epsilon=0.9, nodes=250,
                gamma=0.9, buffer_size=100000, learning_rate=0.0001, time_steps=50):

    agent = DQNAgent(final_epsilon, initial_epsilon, n, difficulty, nodes, gamma, learning_rate, summary=True)
    buffer = ReplayBuffer(buffer_size)
    env = Environment(n)
    total_finishes = 0
    for episode_cnt in tqdm(range(max_episodes)):
        agent.epsilon = agent.epsilon if agent.epsilon < final_epsilon else agent.epsilon * 0.999
        env.generate_new_board(difficulty)
        for i in range(time_steps):
            done = collect_gameplay_experiences(env, agent, buffer)
            if episode_cnt >= 100:
                gameplay_experience_batch = buffer.sample_gameplay_batch(batch_size)
                loss = agent.train(gameplay_experience_batch)
            if done:
                break
            if time_steps % 10 == 0:
                agent.update_target_network()
        if episode_cnt >= 100 and episode_cnt % 50 == 0:
            done_count = evaluate_training_result(env, agent, difficulty)
            total_finishes += done_count
            print("total finishes is {0}".format(total_finishes))
            print("so far the loss is {0}".format(loss))
            print('\n')
```

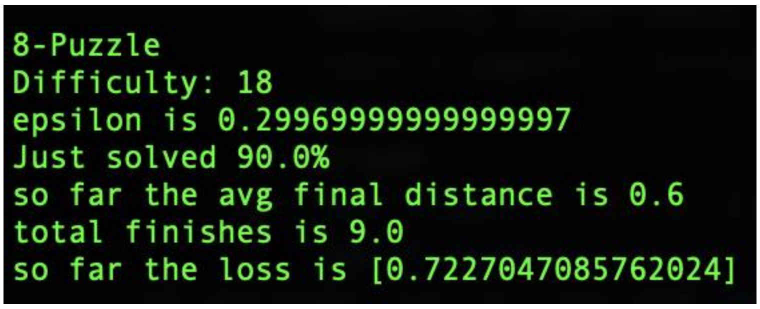
Figure 4: Coded Training Loop

The software will complete 100 iterations of the loop without batch sampling to allow the replay buffer to fill with enough experiences for the first sample batch. Initially, when designing the software, the epsilon value was fixed at 0.3. However, it became apparent that for the agent to quickly learn the rules of the game and explore strategies, this value needs to be higher for the initial iterations.

The epsilon value for the ε-greedy policy starts at 0.9, but for every iteration, the epsilon value decreases by a factor of 0.999. This ε-decay allows for greater exploration of the state space at the start of the training process and greater exploitation of the policy once training has progressed and is closer to a maximum in the state space (Thrun, 1992). Every ten timesteps within an epoch, the parameters of the policy network are copied to the target network.

Due to lack of computing power, the training process is slow and so having an in-situ training evaluation is vital for making small alterations to the network to optimise training speed and quality.

The in-situ evaluation (figure 5) is produced by providing the agent with ten puzzles and using only the policy network; the agent must attempt to solve the puzzles each in under 50 moves.



```
8-Puzzle
Difficulty: 18
epsilon is 0.29969999999999997
Just solved 90.0%
so far the avg final distance is 0.6
total finishes is 9.0
so far the loss is [0.7227047085762024]
```

Figure 5: In-situ Evaluation

For more difficult puzzle sizes and puzzle difficulties, training could occur for a long period of time before any puzzles were solved. This means, using only the total number of solved puzzles, it would not be clear if the training is making progress. Therefore, it is important to include the variable ‘average final distance’, which gives the difficulty of the resulting puzzle state when all moves have been taken. This gives a gauge of how close the puzzles are to the solved solution.

Each evaluation also gives a percentage ‘just solved’ of the 10 attempted puzzles. The training process was accelerated by starting on lower difficulty puzzles and only increasing the puzzle difficulty once three consecutive evaluations have been completed with a ‘just solved’ rate of greater than 80%.

3.2.6. The Puzzle Solver

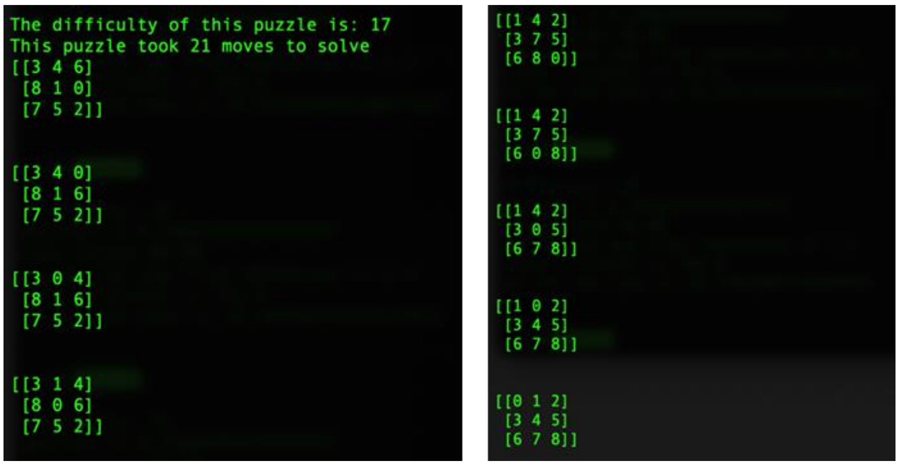
The puzzle solver script is the front end of the software that allows a user to input a puzzle in the form of a number of length N+1. The software creates an instance of the policy network and imports the latest saved network parameters. The user-defined puzzle state is then first checked for solvability. If the puzzle is deemed solvable is inputted to the policy network.

The policy network then has 50 actions to solve the puzzle. If the puzzle is solved in under 50 moves, the step-by-step solution will be printed for the user, with each step being displayed as a 2-dimensional array. If the solution is solvable, the function only needs to perform sequential forward propagations from the starting state until the terminal state is reached or 50 actions have been taken.

The complexity of this solving function is introduced when processing the input state to check if it is solvable. The first step is to find the number of inversions that are in the starting state. An inversion is found by representing the board as a 1-dimensional array. Two tiles (a, b) form an inversion if a comes before b but a > b, ignoring 0 as it represents a vacant position. For example, in the array [0, 2, 1, 3, 4, 5, 6, 7, 8], there is one inversion (2, 1). The following rules can be used to determine if the puzzle state is solvable.

1. If N is odd, then the puzzle instance is solvable if the number of inversions is even.
2. If N is even, the puzzle instance is solvable if
    - The vacant position is on an even row counting from the bottom (e.g. second last), and the number of inversions is odd.
    - The vacant position is on an odd row counting from the bottom, and the number of inversions is even.
3. For all other cases, the puzzle instance is not solvable.

For example, a puzzle has been inputted to the software, which has determined it to be solvable and have a difficulty of 17. The solution is then provided to the user in the format seen in figure 6 below.



```
The difficulty of this puzzle is: 17
This puzzle took 21 moves to solve
[[3 4 6]
 [8 1 0]
 [7 5 2]]


[[3 4 0]
 [8 1 6]
 [7 5 2]]


[[3 0 4]
 [8 1 6]
 [7 5 2]]


[[3 1 4]
 [8 0 6]
 [7 5 2]]
```

```
[[1 4 2]
 [3 7 5]
 [6 8 0]]


[[1 4 2]
 [3 7 5]
 [6 0 8]]


[[1 4 2]
 [3 0 5]
 [6 7 8]]


[[1 0 2]
 [3 4 5]
 [6 7 8]]


[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

Figure 6: Example Output of Solver Script

3.3. Technologies used

This software was written using object-oriented programming in the python programming language in the PyCharm IDE. I used several additional python libraries such as Tensorflow, Keras and Matplotlib.

Tensorflow is needed to build the neural network framework required for the deep reinforcement learning algorithm. TensorFlow provides the necessary tools for building complex and powerful neural networks while avoiding the need for direct coding of the fine details of the deep learning process.

Matplotlib is used for creating data visualisations in python, such as the graphs produced by the network evaluation script.

The training of the neural network was performed using a Macbook Pro (16”, 2019) with a 2.6GHz 6-Core Intel Core i7 processor and 16GB of RAM. As previously mentioned, this processing power is negligible in comparison to that of DeepCubeA who sought to train a similar network.

In an attempt to improve computational power, I experimented with training the DQN using AWS cloud computing. During the training optimisation process, the network was trained on

AWS virtual machine with 9 vCPU; however, this proved to be not enough as training speed increases were negligible. Unfortunately, the acquisition of virtual machines with GPU capabilities is not free.

3.4. Coding Standards

Throughout the development on the software, I made sure to maintain certain coding standards, using the PEP8 coding standard as an informal reference. For example, all class names follow the CapsWords convention but all function names and variables are lowercase with underscores between words.

Each of the function and classes are summarised using a doctoring that states the parameters involved and the outputs of the function. When reading unfamiliar code, simplicity is critical to aiding the understanding. For that reason, I have tried to keep functions simple where possible and use relevant variable names. The lack of code repetition shows efficiency in the code and makes following the code an easier process.

3.5. Design Ethics

For this reinforcement learning algorithm, large amounts of state data are required. In the case of the N-puzzle, there are so many states that finding a dataset containing all the states would be impossible. Often for puzzles such as the N-puzzle, the state data can be self-generated by the agent. Therefore, the lack of use of human data voids any requirement for ethical approval.

Chapter 4. The Results

For this section, I will be evaluating the networks' ability to solve puzzles of different difficulties to understand the effectiveness of the network training process.

The last part of the software is the network evaluation script. To evaluate the progress of training, the software iterates through each difficulty and generates 100 puzzles.

For the 8-puzzle, there are 22 difficulties, and for the 15-puzzle, there are 56 difficulties. The agent is given 50 moves to solve each puzzle. The graphs show the percentage of the 100 puzzles solved at each of the difficulties. This gives a strong indication of the training progress; however, the difficulty is based on an estimate of the number of moves from the goal state and so is not a fully accurate variable.

In the graphs below, figure 7, show the evaluations at different stages in the training process.

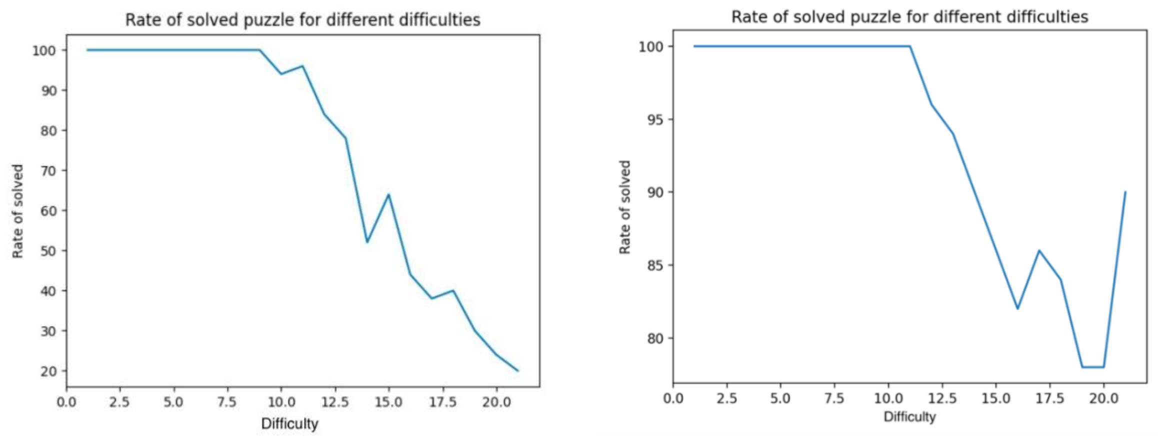


Figure 7: 8-Puzzle Training Evaluation - Early Stage (left) & final (right)

Focusing on the final graph for the 8-puzzle shown on the right of figure 7, the DQN can solve puzzles of difficulties up to and including 11 100% of the time. This % rate of solving then decreases down to 80% by difficulty 18.

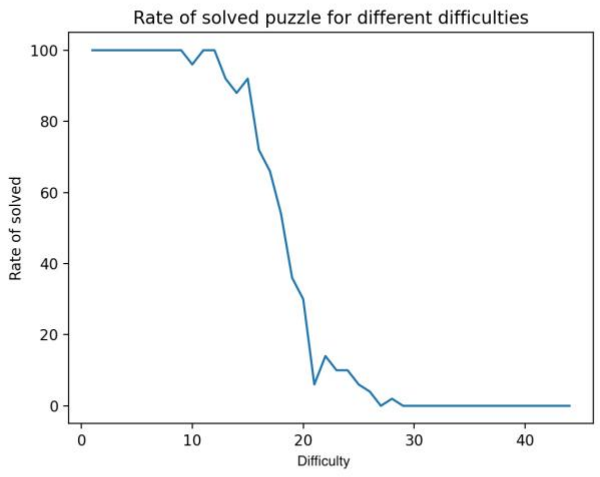


Figure 8: 15-Puzzle Final Evaluation of Training

For the 15-puzzle, figure 8, the DQN was able to solve 90-100% of the puzzles up to a difficulty of 18, then drops to 10% for boards of difficulty 20.

Computational power is the main limiting resource when training the DQN both to train a larger network but also when generating starting puzzles. As the agent learns and trains more difficult puzzles, the time taken to generate the puzzles greatly increases. For example, it can take up to 500 random actions to generate a 15-puzzle of difficulty 30.

During training, it was apparent that despite having 50 moves to solve the puzzle, none of the puzzles took 40-50 moves to solve. This showed that regardless of the number of actions taken by the agent, the policy would produce polar outcomes meaning the agent could either solve the puzzle or it couldn't. This is because for each forward propagation, the current state is provided to the network which uses the policy as its strategy to determine which action to take. Occasionally, due to incomplete training, when the network forward propagates a state, the action suggested by the network is the reverse of the action suggested when the network forward propagates the next state. This means the network will go back and forth thinking the most optimum move is to go back one move then forward again. During training, this cycle cannot last due to the epsilon value encouraging occasional exploration and therefore the eventual discovery of a correct strategy. During the evaluation however, the network only uses exploitation of the policy meaning for some states, an unrefined policy can lead to this looping affect.

Unfortunately, there is no simple way to calculate the optimum number of moves required to solve each puzzle, and so the optimality of the solutions calculated by the DQN cannot be properly evaluated. For every one of the 8-puzzles, each puzzle can be solved in 30 moves or less, with only two puzzles requiring 30 moves to reach a solution. The DQN can solve both of the most difficult puzzles, one in 33 moves and one in 39 moves.

Chapter 5. Future Developments

This section sets out some of the potential enhancements that could be developed in future iterations of the software.

5.1. Time and Computational Power

The key limitation of training the current network is computational power. With more time, I would intend on training the DQN to be able to solve every 8-puzzle and most of the 15-puzzles. This would only be possible by either training the DQN on a more powerful computer and for a longer period or making changes to the DQN that increase the training efficiency.

For example, the DeepCubeA network (Agostinelli *et al.*, 2019) trained for 1 million iterations on two NVIDIA Titan V GPUs, with six other GPUs used in parallel for data generation. In total, the DNN saw 10 billion examples during training. The training was completed in 36 hours. As the only other example of a DQN being used for the N-puzzle, this is a huge difference in computing power to that of the facilities I have access to and shows the importance of such power for achieving the results of DeepCubeA.

With the right facilities to train my software, I would ideally run a separate computer solely for generating the puzzle state data and the other for training the generated data. With a network that trains faster, restrictions on the size of the network and the learning rate can be reduced to produce a more complex and efficient network.

5.2. Prioritised Experience Replay

One potential change that I would introduce is the inclusion of prioritised experience replay. Currently, the DQN uses uniform experience replay, meaning experience transitions are uniformly sampled from the replay memory.

When humans learn, we don’t give equal significance to our memories; instead, we prioritise the memories that held the most significance to the task we are trying to learn. This process of prioritising experiences was applied to the DQN algorithm by DeepMind, meaning more important memories were replayed more frequently, allowing more efficient learning.

When comparing DQNs used to play Atari games, the network using prioritised experience replay outperformed the network using uniform experience replay in 41 out of 49 games (Schaul *et al.*, 2016).

5.3. Double Deep Q-Network

The classic DQN network has been shown to often suffer from a substantial overestimation of action values when under some conditions. A new double deep q-learning algorithm has since been developed and has been shown to greatly reduce most of the overestimation.

This new algorithm can lead to far greater performance for some learning tasks (van Hasselt, Guez and Silver, 2015). I would be interested to see what effect implementing this algorithm has on the puzzle solving capabilities of the DQN.

5.4. Graphical User Interface

While the solver.py script is effective as a front end for the software and can provide step-by-step solutions to the puzzles, the interface is extremely simplistic. With more time, I would use the TKInter python library to construct a user-friendly interface where tiles are dragged and dropped into their position on an empty board for input. The step-by-step transitions would be animated and could be easily moved through using the forward and backward arrow keys.

Chapter 6. Conclusions

In summary, I have created a Deep Q-Network that is capable of being trained to solve the N-Puzzle. The network uses an ε-greedy policy to promote both exploration of the environment and exploitation. The input state data is one-hot encoded and the network outputs action value estimates. Training follows the deep q-learning algorithm and uses experience replay to store experiences and then train from sample batches in order to reduce the short-sighted effect of using correlated data. The DQN used a reward system based on the combination of the Manhattan distance heuristic and the number of linear conflicts. The training parameters are saved and can be reloaded into the network allowing training to continue from a checkpoint. The software uses a python script called Puzzle_Solver.py to ask users for a puzzle state and attempt to solve the puzzle using the trained network. If successful, the script will provide the user with step-by-step instructions on how to solve it.

The only part of the project plan I was unable to attempt due to restrictions on time was the development of a graphical user interface. If I was to extend the project for further development, I would create a GUI that would interact with the Puzzle_Solver.py script to provide an enjoyable user experience when finding solutions to combinations of the N-puzzle.

The network was trained on both the 8-puzzle and the 15-puzzle and showed promising results. Most of the 8-puzzles could be solved even up to the highest difficulty and many of the 15-puzzles could be solved but only the lower difficulty puzzles. Despite the agent being limited to solving low difficulty 15-puzzles this is by no means a failure and is simply the result of not having the computing resources available to push training further. With most methods of solving the N-puzzle if the solution cannot be found in a set number of actions, then increasing the number of actions can result in a solution that is just not very optimum. Training results showed that for the DQN if a puzzle cannot be solved, allowing more actions to solve the puzzle will not affect the outcome, as the policy used by the network is fixed.

Taking into consideration the lack of computational power available, I think relative to a huge network like DeepCubeA, my DQN produced far better results than I expected. I hope to continue the network training via a subscription to an AWS virtual machine and am also interested to see what changes the introduction of prioritised experience replay will make to the network’s capabilities.

The reward system I designed and the method of representing the state and implementing the actions has not been done before when using a DQN to solve the N-puzzle making my project rather unique. This has made the project even more exciting and I look forward to seeing further developments in the field of deep reinforcement learning that enable more efficient training and more effective learning.

Bibliography

Agostinelli, F. *et al.* (2019) ‘Solving the Rubik’s cube with deep reinforcement learning and search’, *Nature Machine Intelligence*, 1(8), pp. 356–363. doi:10.1038/s42256-019-0070-z.

Bischoff, B. *et al.* (2013) ‘Solving the 15-Puzzle Game Using Local Value-Iteration’, p. 13.

Fedus, W. *et al.* (2020) ‘Revisiting Fundamentals of Experience Replay’, *arXiv:2007.06700 [cs, stat]* [Preprint]. Available at: http://arxiv.org/abs/2007.06700 (Accessed: 24 August 2021).

van Hasselt, H., Guez, A. and Silver, D. (2015) ‘Deep Reinforcement Learning with Double Q-learning’, *arXiv:1509.06461 [cs]* [Preprint]. Available at: http://arxiv.org/abs/1509.06461 (Accessed: 13 September 2021).

Mathew, K. and The Society of Digital Information and Wireless Communication (2014) ‘EXPERIMENTAL COMPARISON OF UNINFORMED AND HEURISTIC AI ALGORITHMS FOR N PUZZLE AND 8 QUEEN PUZZLE SOLUTION’, *International Journal of Digital Information and Wireless Communications*, 4(1), pp. 143–154. doi:10.17781/P001092.

Mehta, A. (2021) ‘Reinforcement Learning For Constraint Satisfaction Game Agents (15-Puzzle, Minesweeper, 2048, and Sudoku)’, p. 19.

Mnih, V. *et al.* (2013) ‘Playing Atari with Deep Reinforcement Learning’, *arXiv:1312.5602 [cs]* [Preprint]. Available at: http://arxiv.org/abs/1312.5602 (Accessed: 7 September 2021).

Schaul, T. *et al.* (2016) ‘Prioritized Experience Replay’, *arXiv:1511.05952 [cs]* [Preprint]. Available at: http://arxiv.org/abs/1511.05952 (Accessed: 13 September 2021).

Shaban, R., Natheer Alkallak, I. and Mohamad Sulaiman, M. (2010) ‘Genetic Algorithm to Solve Sliding Tile 8-Puzzle Problem’, *JOURNAL OF EDUCATION AND SCIENCE*, 23(3), pp. 145–157. doi:10.33899/edusj.2010.58405.

Sutton, R.S. and Barto, A.G. (1998) *Reinforcement learning: an introduction*. Cambridge, Mass: MIT Press (Adaptive computation and machine learning).

Thrun, S.B. (1992) *Efficient Exploration In Reinforcement Learning*. USA: Carnegie Mellon University.

Appendix 1: Installation Guide

The project package contains a text file six python scripts and two folders containing the pretrained network parameters for the 8-Puzzle and the 15-Puzzle:

- README.txt
- Training_Loop.py
- Environment.py
- DQNAgent.py
- Evaluate.py
- Puzzle_Solver.py
- Folder: training_8_puzzle_250_nodes
- Folder: training_15_puzzle_500_nodes

Preliminary Installation Commands:

```
pip install tensorflow

pip install keras

pip install tqdm

pip install matplotlib

pip install numpy

pip install collections

pip install python
```

Training Guide:

- "N" is the puzzle size you are wanting to train on so for the 8-puzzle, N=8. Difficulty is an estimate of the optimum number of moves taken to solve the puzzles the agent will be using for training.
- A lower difficulty parameter will result in faster training as the puzzles are easier but also much faster to generate. Bear in mind if the difficulty is set higher than the maximum difficulty for that puzzle the puzzle boards will never be generated, and training won't progress.

- There are already pre-trained saves included in the software, one for the 8-Puzzle and one for the 15-puzzle should you wish to use the pre-trained networks and continue the training.

- **To begin training:** open the command prompt and run the command…

  ```
  python Training_Loop.py
  ```

- From there, you will be asked to input the N value for the puzzles you would like to train using. For example, for the 8-puzzle, you should input 8.
- You will be then asked to provide a training difficulty. The difficulty should not be set higher than 21 for the 8-puzzle and 50 for the 15-puzzle. I would advise a training difficulty of 15 for the 8-puzzle.
- Finally, you will be asked if you wish to restart training or continue from the last training checkpoint. You must type any of "yes", "no", "y" or "n". If you select yes to restart training, all previous training data for that puzzle size will be deleted.
- The training should then begin and will automatically save the network parameters for future use.
- The first time the network is trained, I would advise not interrupting training until ε-decay has finished and the epsilon value, stated in the in-situ evaluations, has reached the final epsilon value (default is 0.3).

  To make changes to the other hyper-parameters of the DQN, open the Training_Loop.py file, and at the bottom of the script, choose the desired network hyper-parameters.

Network Evaluation Guide:

- To evaluate the network training, simply run the command…
  ```
  python Evaluate.py
  ```
- Input the size N of the puzzle you would like the network to use for evaluation.

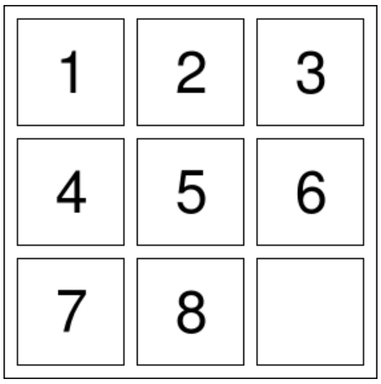


Figure 9: Example 8-puzzle

Guide to solving puzzles with your trained network:

- Firstly, you will need to convert the puzzle you wish to solve into the correct format.
- Figure 8 shows an example 8-puzzle that can be written as a list of length N., e.g. figure 8 = 123456780
- Once you have converted your puzzle into the correct format. On the command line, run the command…
  python Puzzle_Solver.py
- Then type in the reformatted puzzle and press enter to begin attempting to solve the puzzle.

Appendix 2: Additional Design Detail

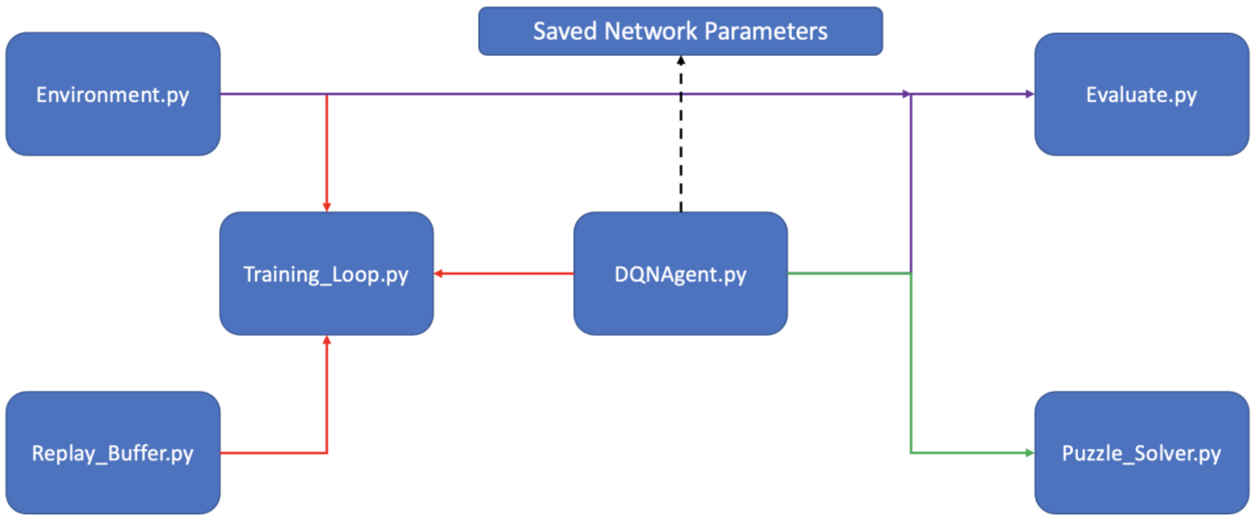


Figure 10: Diagram showing the software structure