Bachelor thesis
Computing Science

Radboud University

Improving reinforcement learning performance in 2048 using expert knowledge

Author:
Koen Sauren
s1024202

First supervisor/assessor:
Dr. Nils Jansen
N.Jansen@cs.ru.nl

Second assessor:
Dr. Daniel Strüber
d.strueber@cs.ru.nl

August 23, 2022

Abstract

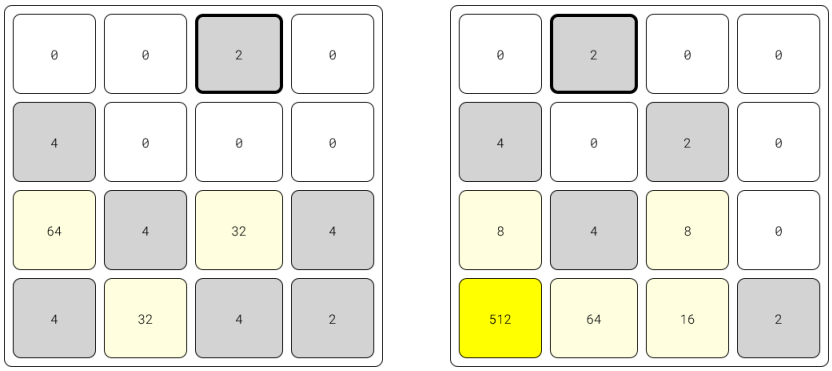
We developed a method for playing 2048 using a deep reinforcement learning agent using expert knowledge. Previous deep RL-based agents only achieved poor performance, never exceeding a 1024 tile. We acknowledge that a specific strategy for moving the tiles is required to win in 2048 and that enforcing such a strategy is a form of enforcing strict safety on the agent. We allow the policy to deviate from the chosen strategy at key states in the game, using expert knowledge to find such states. We define metrics to compare states and actions, as well as a heuristic to embed the expert knowledge into the policy. The construction of our system is inspired by *shielding*, a method used to guarantee safety in reinforcement learning. Using this method, performance increases noticeably, as compared to greedy, strategy-less agents. agents adhering to strict safety. In total, we study three different agents: a greedy agent, which optimises immediate reward, a naive safe agent, which strictly adheres to the strategy, and an agent using expert knowledge. Experiments show that the greedy agent performs poorly and is not able to reliably get a score higher than 256. The naive, safe agent, outperforms the greedy agent, more reliably achieving a 512 tile, but it cannot win the game. The agent using expert knowledge outperforms both. Our agent is able to achieve a 1024 tile reliably and achieves a 2048 tile, thus winning the game, on several occasions.

Contents

Chapter 1

Introduction

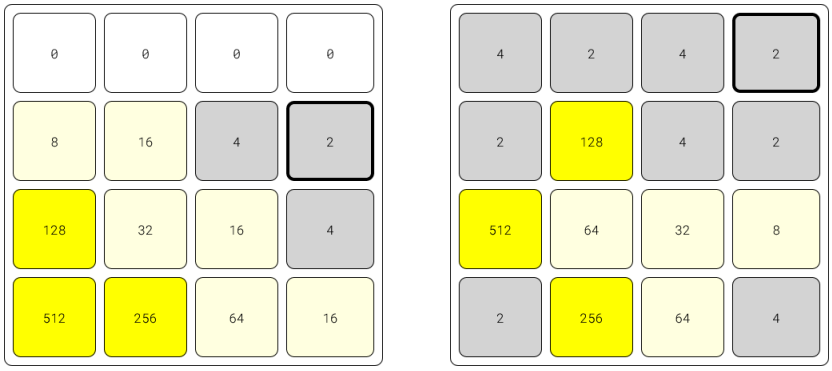
2048 is a sliding puzzle video game, where the player's objective is to combine numbered tiles on a 4x4 grid by sliding them until a tile with a value of 2048 is achieved [8]. After each move, a new, low-valued tile appears on the grid. This ensures that the player does not run out of tiles to combine, but it also makes unsolvable grid arrangements possible. To prevent such unsolvable arrangements, i.e., arrangements where no tiles can be combined and no new tiles can be added, keeping the grid ordered is essential. Doing so will require that the tiles are moved according to a certain strategy.
The grid is small, so a simple strategy, such as keeping the highest value tile in a certain corner, suffices. As an example, consider the grid on the left side of figure 1.1. The board is not ordered, which makes lining up tiles of equal value and merging them near impossible. By extension, this also means that it is nearly impossible to reach 2048. In the grid on the right of 1.1, the highest tile is kept in the bottom-left corner, and by only using LEFT and DOWN (a LEFT-DOWN strategy), the desired ordering is maintained. For consistency, we will always use this strategy going forward. It might now be easy to conclude that an easy way to solve 2048 is by simply marking certain moves as *unsafe*. In the case of the previous example, these moves would be UP and RIGHT.



(a) An unordered grid (b) A well-ordered grid

Figure 1.1: 2 example grid arrangements.

This would be simple to implement, but will not work to solve the game. As there, at some point, will be an arrangement of the grid where the legal moves are not possible, the player will in such a case be forced to make a disallowed move. This will almost certainly reduce the 'ordered-ness' of the grid, and increase the chance of game-over. For an example, see figure 1.2.



(a) (b)

Figure 1.2: In the grid arrangement on the left, UP is the only possible move. The right shows the resulting grid after performing this move and playing a few more moves. While it is sometimes possible to recover from a situation like this, it is more likely to lead to game-over quickly.

Deep reinforcement learning is a tool well suited to solve this problem. The agent can be trained to avoid states that require a disallowed move and learn to recover quickly from unfavorable states. It would even be possible to strictly disallow unsafe moves in states where any safe move is possible. While this generally yields fairly good results, it does not take into account subtle nuances of the game, which human players are able to perceive and use to their advantage. For example, figure 1.3 shows a well-ordered grid, which

allows for both a move left and a move down. However, the more practiced player will quickly see that moving right will be better in this state. Since the bottom row, where the most valuable tile is, would not be affected by any move besides UP, moving right will have few serious consequences for the ordering of the grid, because restoring the ordering when only low-value tiles are not ordered is relatively easy. Moving right will position the two 64 tiles above each other, allowing them to be combined into a 128 tile, right next to the existing 128 tile. These tiles can then be combined into a 256 tile. This progression is shown in the bottom half of figure 1.3. This nuance, however, will not be detected by the naive reinforcement learning agent described previously, because it would not consider an unsafe action if a safe action is available. As a result, in a situation like the one described in figure 1.3, the agent will not choose the unsafe move, even though it is better. This is problematic, as such moves are usually the key to success in 2048. For an agent that performs such technically unsafe, but valuable moves it may be easier to achieve a 2048 tile, thus winning the game, and it may be easier to do so in fewer moves. To train an agent to act in this manner, a new approach is required.

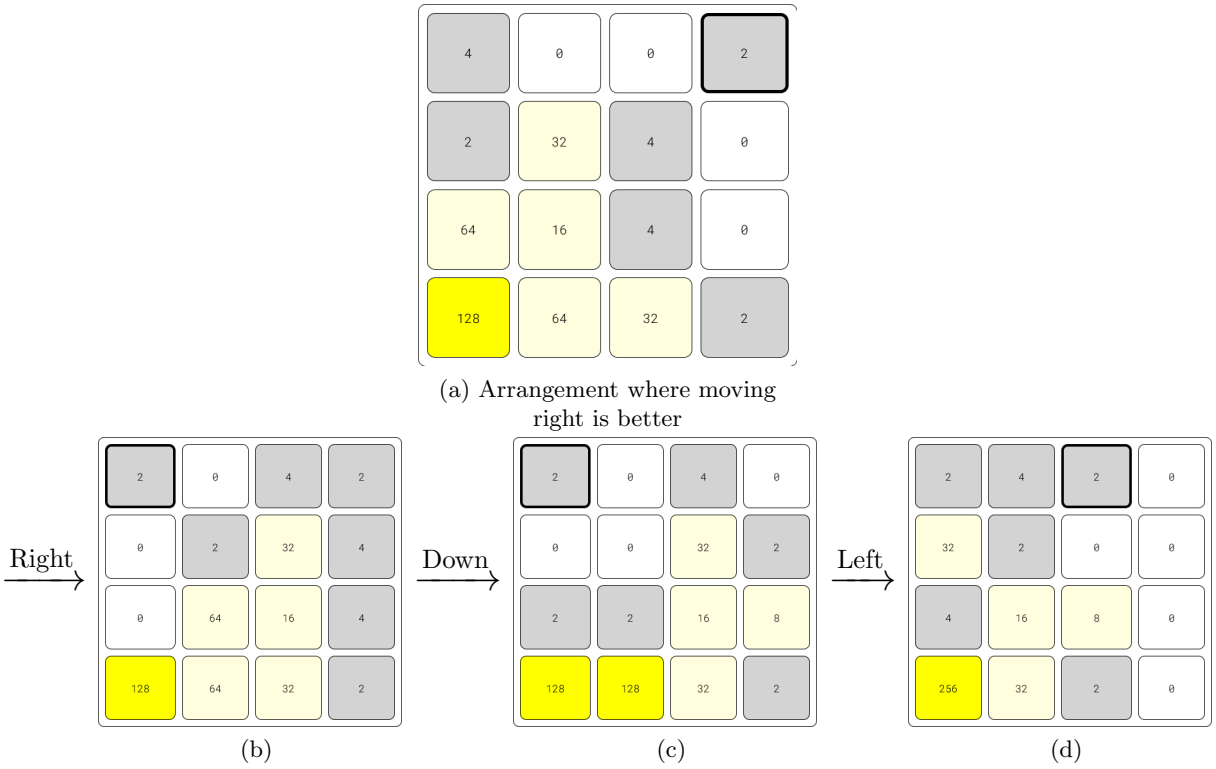


Figure 1.3: Progression of grid arrangements after moving right, showing that merging up to a 256 tile is made easier.

We will now further introduce the problem at hand, and introduce a method to improve the performance of a deep reinforcement learning agent using expert knowledge. The resulting policy will play more like a human player by learning to perceive nuances in the game state and use them to its advantage. This is achieved by introducing a second layer between an agent aiming for strict safety and the environment, which checks if an unsafe action entails an acceptable risk and high potential reward and, if this is the case, overrides the command issued by the baseline agent. We will also show that our approach outperforms both unchecked agents (which only optimise reward) and agents aiming for strict safety, such as the naive agent described previously.

Chapter 2

Preliminaries

2.1 (Deep) Reinforcement Learning

In Reinforcement Learning (RL) [24], a *policy* is learned by an *agent* through interacting with an *environment*. The agent selects an action at each timestep (game state) and the environment responds with the new state resulting from the chosen action, and a *reward*, which is used by the agent to determine if the chosen action was good or bad. The agent's objective is to maximise the cumulative reward. Doing this gives rise to the optimal policy, a set of rules the agent can follow to get the maximum reward in each state. The reward is useful for immediately evaluating states after they arise from some action. In the longer term, however, it is useful to know how valuable a state is to the agent if the optimal policy is followed from that state. To do this, the agent can define a *value function* on the states, which describes how much reward the agent can expect to get in the future if it starts from a certain state.

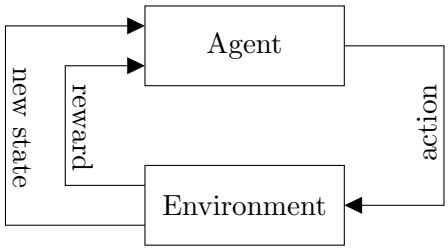


Figure 2.1: Diagram of the interaction between agent and environment in a simple reinforcement learning system.

Formally, we can describe these concepts as follows:
The **policy** followed by the agent at each timestep t, denoted π_t, is a mapping from state-action pairs to the probability that an action is chosen in that state. So, $\pi_t(a|s) = p(A_t = a|S_t = s)$.
The **return** G_t defines a function on a sequence of rewards $R_t, R_{t+1}, \ldots$. It is the agent's objective to maximise the value of this function. Usually, the *discounted return* at some timestep is maximised, defined as

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where γ is the *discount factor*, $\gamma \in [0, 1] \subseteq \mathbb{R}$, which determines the value of future rewards in the present.
Then, the **value function** defines the expected value of a state-action pair in the long term when following some policy π, $v_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a]$.
A **probability distribution** over some set X is a function $\mu : X \to [0, 1] \subseteq \mathbb{R}$, and $\sum_{x \in X} \mu(x) = 1$. Then, $Distr(X)$ denotes all distributions on X.
The **environment** is usually formalised using a Markov Decision Process (MDP), represented by a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where:

- $\mathcal{S}$ is a set of states (the state space).
- $\mathcal{A}$ is a set of actions (the action space).
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to Distr(\mathcal{S})$ is a probability function on state-action pairs. For each such pair, a probability distribution is returned, which defines the probabilities of reaching successor states s' from state s using action a.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the immediate reward function, denoting the reward received after moving from state s to state s' using action a.

Further, we have that for all states $s \in \mathcal{S}$ the *available actions* are $\mathcal{A}_s = \{a \in \mathcal{A}|\mathcal{P}(s, a) \neq \bot\}$.

Applying this framework to 2048 gives the following definition for the 4-tuple defined above:

- $\mathcal{S} = \bigcup_g \{g_{x,y} | x, y \in \{1,2,3,4\}\}$, where $g_{x,y}$ is the value of the tile at location (x, y) in some grid g, i.e., $g_{x,y} = \begin{cases} 2^n, \text{ with } n \in \{1, \ldots, 11\} \\ 0, \text{ if the tile at } (x,y) \text{ is empty.} \end{cases}$
- $\mathcal{A} = \{\text{UP, DOWN, LEFT, RIGHT}\}$
- $\mathcal{P}(s, a, s') = \begin{cases} 1, \text{ if } s \neq s', a \in \mathcal{A}_s, \text{ and performing } a \text{ in } s \text{ indeed leads to } s'. \\ 0, \text{ otherwise.} \end{cases}$
- $\mathcal{R}S$ (the reward) can be implemented in various different ways. Implementing this function in a certain way will allow us to push the agent in a certain direction, something we will use later on.

As discussed previously, the objective of the RL agent will be to learn a policy that maximises the expected cumulative return for every state-action pair. To do this, it is necessary for the agent to explore new states, as well as using (exploiting) the value of states it has seen before. However, it is also necessary to prevent the agent from getting stuck in familiar territory, where it only uses state-action pairs it already knows to have high value. To counter this, the agent has to be forced to explore new, potentially better, states. An *ϵ-greedy policy* can be used for this. An **ϵ-greedy policy** is an extension of the 'normal' policy π_t, where we redefine the policy as follows:

$$\pi_{\epsilon-\text{greedy}}(a|s) = \begin{cases} \pi_t(a|s) \text{ with probability 1 - } \epsilon, \\ \text{random value } \in [0,1], \text{ with probability } \epsilon. \end{cases}$$

where $0 \leq \epsilon \leq 1$. Usually the chosen value for ϵ is initially quite high, but decreased as training progresses, because we want to explore as many states as possible in the initial stages of training. Later on, however, we want to exploit the known values of states and actions more.
Several approaches exist for learning the policy π_t, but here we will discuss Q-Learning [16]. This method is well suited to environments with stochastic transitions and rewards, like in 2048. Furthermore, Q-learning is very suitable for environments with discrete action and state spaces, like 2048. In this approach, a *Q-value* is learned for each state-action pair. A higher Q-value then indicates that action a is better in state s. The best action in state s is then the action with the highest Q-value. During training the Q-values are updated iteratively based on what the agent observes during exploration. Formally, we can define this as:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \text{max}_{a'} Q(s', a') - Q(s,a))$$

Here, α is the learning rate and γ is a discount factor which determines how important long-term reward will be. Essentially, the Q-value is updated towards the observed reward plus how good the state we end up in is (the max Q-value of the next state). To learn an optimal policy, i.e, the optimal action for each state, we would need to learn a Q-value for each state-action pair. If $\mathcal{S}$ is sufficiently large, however, this quickly becomes infeasible, like for, for example, 2048. When this is the case, we can approximate the Q-values for each state-action pair by learning a model. This can be done using Deep Q-learning [19][20]. In Deep Q-learning, we learn a neural network, and Q-values are estimated by querying this network with a state as input. As usual with neural networks, we update the weights and biases of the network to optimise a loss function. Internally, two networks are used, a main network and a target network. The parameters of the target network are updated at an interval set by a hyperparameter, using the weights of the main network at that time. The target network is used during training (and after training) to provide stable Q-value estimates for updating the main network. To further improve stability, correlations in training data (between consecutive transitions) are removed by storing transitions in a replay buffer and sampling randomly from this buffer to train the network. Note that our previous definitions for the MDP representing 2048 can be used unchanged with Deep Q-learning.

2.2 Safety for Reinforcement Learning

While reinforcement learning is a powerful tool to achieve certain tasks, it can be hard to train agents capable of operating in safety-critical environments. During exploration, states can be visited which are considered *unsafe*. In certain use cases, the mere possibility that an unsafe state will be reached can be undesirable or even unacceptable. As an example, consider a robot moving through a grid world. The goal state is a certain cell in this world. An adversary moves through the same grid world randomly. If the adversary spots the robot, i.e., it has a direct line of sight, it is game over. To explore the state space safely, the robot will have to avoid states where the game-over condition exists. Since a reinforcement learning agent will aim for optimality and convergence, it is by definition not concerned with safety. Most RL algorithms instead focus on exploration, but they assume that the used MDP is *ergodic*, meaning that any state can be reached from any other state by following a policy that does not end in termination. In certain systems, however, this cannot safely be assumed, as some states are not reachable from all other states during exploration, because those states are unsafe and should not be visited. Marking such states as unvisitable would violate ergodicity, however, so certain properties of the algorithm used may no longer hold. This problem gives rise to *safe explo-*

ration [14] [21] [22] in reinforcement learning. These techniques aim to make exploration safe and by extension provide a safety guarantee on the agent. Besides safe exploration, other techniques may be used for providing such a safety guarantee, some of which may be found in [11]. To provide such a guarantee, it is necessary to incorporate external knowledge of whatever process is being learned into the learning process. By incorporating such knowledge, a guiding hand is introduced, which will help the agent see its own blind spots [9]. For example, it is possible to implement the agent such that it will consult a teacher, which may be a human being, if it cannot decide for itself whether an action might give rise to a catastrophic situation. Conversely, the teacher may monitor the agent and intervene when it deems appropriate. It is also possible to use *apprenticeship learning* [2], where the reinforcement learning agent resembles a supervised learning agent. The teacher provides example policies, which the RL agent can follow and then learn from. There are many different notions of safety for reinforcement learning, as well as many approaches to ensuring safety. A good survey of definitions and methods for safe RL can be found in [11]. One such approach is generally called *Shielding*, and we will focus on that method here.

2.3 Shielding for Reinforcement Learning

Generally speaking, a *shield* in the context of RL is a layer between the agent and the environment, which verifies commands issued by the agent, before they are executed in the environment [3]. The shield can then filter out any unsafe commands from the agent. If the shield detects that a command from the agent will lead to an unsafe state, it can cancel that command and issue a new command, which is not (or less) problematic. Also see figure 2.2.

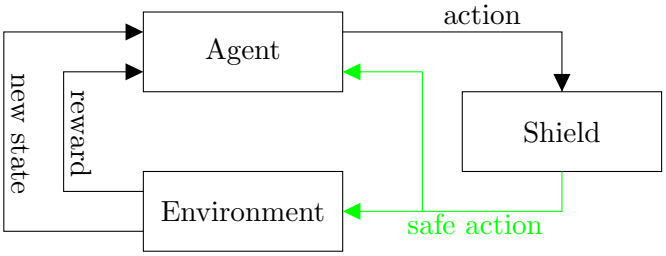


Figure 2.2: Diagram of a shielded reinforcement learning system.

Shielding can be seen as a form of RL with a teacher, as described in the previous section. The key difference is that the shield is in principle always monitoring the agent, whereas teacher-guided RL may only consult the teacher when the agent being trained deems it necessary. Furthermore, in teacher-guided RL the teacher is only active during training, so its advice must be incorporated into the agent's policy during training. Shields, on

the other hand, may also be active when the agent is deployed in some real-world scenario. Critically, combining a shield with a fundamentally flawed agent means that the combined system can act correctly and safely. This notion is used extensively in our approach.

As with other methods for safety in RL, several approaches to shielding exist. Some approaches compute the shield before training starts [3]. The shield can then be deployed in two different ways. It is interesting, for our understanding, to discuss these two briefly. In [3], the authors make the distinction between *Preemptive Shielding* and *Post-Posed Shielding*. A post-posed shield, as shown in figure 2.2, verifies actions selected by the agent, and interferes if the chosen action is unsafe. A preemptive shield, as shown in figure 2.3, will be consulted each time the agent makes a move. The shield will provide a list of safe actions, and the agent will use its own structure and training to determine the optimal move out of the ones available to it.

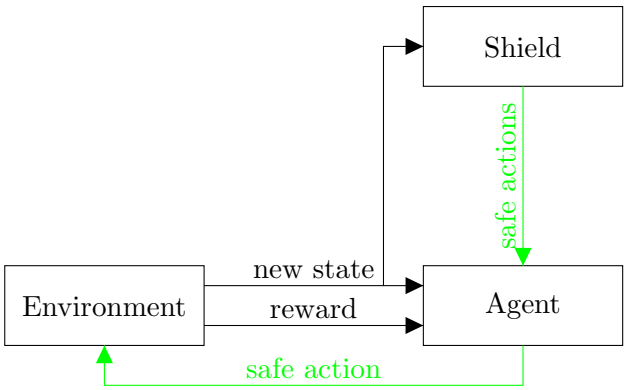


Figure 2.3: Diagram of preemptive shielding.

It is also possible to compute the shield on the fly and update it at an interval during training, like in [23]. Here, the shield is essentially a separate learning agent, which uses a version of the MDP used by the main agent. The benefit of an approach like this is that it is more dynamic, allowing for changes in the environment and behaviour of the agent to be captured in the shield. For complex environments, where dynamics may change over time, this is more effective than a static shield.

Chapter 3

Problem statement

As already stated in the introduction, winning in 2048 requires the use of a strategy. Creating a reinforcement learning agent which can adhere to such a strategy is relatively straightforward; by implementing the reward function such that the two safe moves given by the chosen strategy are heavily incentivised, the agent should quickly learn to adhere to such a strategy. Alternatively, we could explicitly disallow the unsafe moves given by the strategy, unless a safe move is not available. This approach is, however, too straightforward to solve 2048. Using only the two moves given by the strategy at all times results in an agent that is too naive, and will only achieve relatively low scores. A strict safety requirement would mean that no unsafe state can ever be reached, i.e., game-over is not possible. Such a requirement is too strict, as it is usually not possible to achieve the goal state by only making legal moves. It is almost inevitable that we will, at some point, end up in an unsafe state, meaning a state where only illegal moves are possible. This does not have to be very consequential to our ability to win the game, however. As a result, using a shield in the form as it is usually described is not really helpful, as shielding is usually used to (try to) provide a strict safety guarantee. In order to further specify the problem at hand, it is helpful to first define explicitly what an unsafe state is for 2048. We will consider the state after merging two 1024 tiles, the first state where a 2048 tile exists, to be the final 'victory' state for the player. [1] It follows then, that when the board has become saturated with tiles that cannot be merged, this is a game-over state. These states are obviously unsafe, but other states can also be considered unsafe. The left side of figure 3.1, for example, shows a state where the only move possible is UP (which is assumed to be an unsafe move).

[1] Most versions of 2048 allow the player to continue the game beyond this state, however. In these versions the game could theoretically go on forever, reaching a 4096 and 8192 tile and even higher.

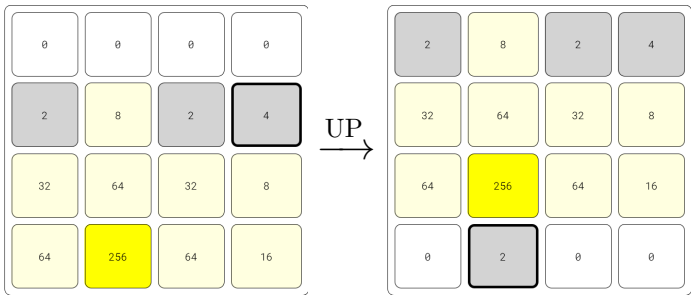


Figure 3.1: An unsafe state and its successor.

It is easy to see that this state is very likely to lead to game over. If we move UP, a low-valued tile will be spawned in the bottom row, and it will be very difficult to recover the state ordering after this. This is shown on the right side of figure 3.1.

For other types of states, however, it is not as clear whether or not they are unsafe. Consider, for example, the states in figures 3.2 and 3.3. These states are very similar, but the left state is considered safe, while the right state is considered 'unsafe'. To see that this is the case, we will perform RIGHT, an unsafe move, in both states. The resulting states are shown in the bottom halves of the same figures.

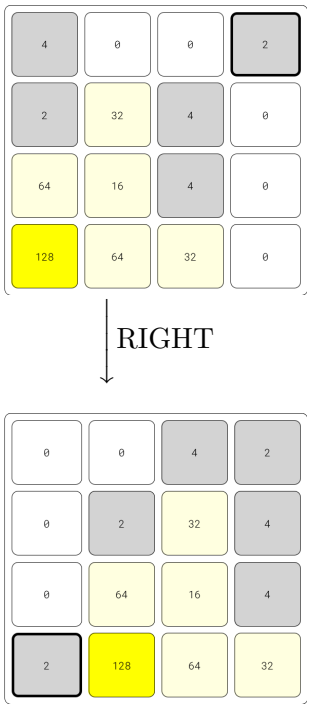


Figure 3.2: An unsafe state and its successor.

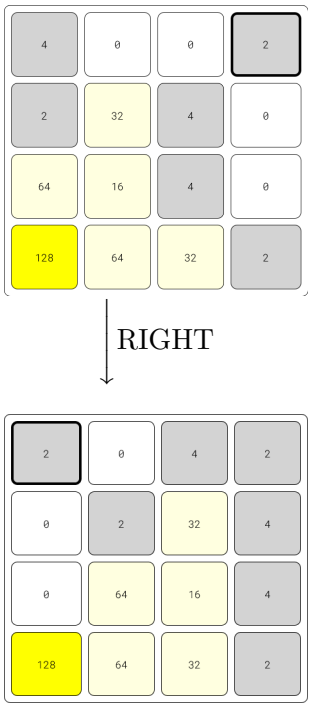


Figure 3.3: A safe state and its successor.

It is clear that the successor state in figure 3.2 is undesirable; a low tile has spawned in the bottom right, where the highest tile should be. The successor state in figure 3.3, on the other hand, is very desirable; using DOWN here will give us a second 128 tile which can then safely be merged with the other 128 tile. Also note that the first state in 3.3 can be achieved by performing DOWN in the first state of 3.1, showing that an unsafe state can sometimes be made safe easily. This point is non-trivial, as the unsafe move (RIGHT) is optimal in the safe state. If the agent strictly avoids unsafe states, DOWN would be chosen in the unsafe state, which is good, but in the resulting state, RIGHT would not be considered. Because of this, an agent aiming for strict safety is not optimal, but an agent which does not do this will also not learn the nuance demonstrated by this example. In practice, the agent will not learn to reliably recover unsafe states first, before taking a risk with high reward (as in figure 3.3).

It might seem then, that removing the strict safety requirement from the agent will solve the problem. By doing this, the agent will learn to achieve an optimal reward in the next state, but it will also be very short-sighted. The agent will not learn to take into account the longer-term consequences of its actions. This problem can be alleviated somewhat by tweaking the discount value γ, but this is an imperfect solution. Solving this problem would require

an agent to learn the difference between safe and unsafe states. But, the agent only learns what action to choose to get an optimal reward in the next state, not what sequence of moves is required to balance safety and reward. So the objective of our agent will be to avoid unsafe states, but if this cannot be done by merely avoiding all unsafe moves, how do we balance safety with taking acceptable risks? As we have seen, simply applying a solution that achieves safety, like a shield, is unsuitable (and also unnecessarily complicated). Instead, a different approach is needed.

Even though 2048 is a relatively niche use case, this problem, balancing the need for safety and the need for taking risks, can in fact be seen in all kinds of real-world use cases. For example, self-driving vehicles are systems where reinforcement learning can come in very useful. In such cases, making the wrong move could have catastrophic consequences, so safety is an obvious requirement. Ideally, safety could be guaranteed by using, for example, shielding. However, shields are not perfect, so we could end up with a system that either does not guarantee safety, which is unacceptable, or a system that is overly cautious. The latter could, for example, cause the system to refuse to enter an intersection if the shield does not deem it perfectly safe to do so. This problem is not solved by lowering the safety standards imposed by the shield, as that could cause unacceptable actions to be taken in different, similar cases. Participating in traffic means that taking risks is unavoidable, but a strict safety guarantee would make it hard to strike a balance. If we use a system adhering to strict safety, it might occur that, when no safe move is available, no choice is made. It is precisely in such situations that a choice *has* to be made, because not doing so could be more dangerous than making a slightly risky choice. In such cases, a system that balances out the strict safety guarantee would be useful.

Chapter 4

Our solution

4.1 Reverse Shielding

To achieve an agent that performs as desired, by balancing safety and adherence to the strategy with taking acceptable risks under certain requirements, we can use the intuition of shielding in a different way. While the final result should not be referred to as shielding, as it does not guarantee safety or correctness, it is helpful to first explain it as a form of (post-posed) shielding, namely *reverse shielding*. Under the 'classic' notion of shielding, the shield acts as a check on the agent by overriding unsafe moves. In our solution, the shield acts as a check on safe moves, by overriding them with an unsafe move if such a move carries an acceptable risk and high potential reward. As stated, this does not guarantee safety, and might thus be better described as infusing an agent with *expert knowledge*. As the concept described closely matches the structure of a shield, however, we will intuitively refer to our method as reverse shielding.
It would of course also have been possible to constrain a greedy agent, which does not consider safety at all, using a 'normal' shield. In this setup, the shield would override unsafe actions. This, however, would also be suboptimal as the strict adherence to safety (as a consequence of the shield) may cause deadlocks, where the shield causes the agent to get stuck in a known safe zone of the state space. The shield only enforces safety and does not enforce that good rewards are obtained or that some target (in our case, reaching 2048) is reached. This is a known problem with shields, discussed in, for instance, Carr et al. [6].

In this architecture, we use a naive agent biased towards safety as a baseline. The majority of moves actually being executed in the game will come from this agent. However, a second layer (the 'shield') checks every safe action selected by the agent and checks if an unsafe move might be better. The second layer might thus be described as being biased towards taking risks.

Note that the naive agent does not only select safe actions, as it is possible to end up in a state where only unsafe actions are possible. In a state like that, the naive agent will act risk-averse, and select the 'safest' unsafe move.

Because the second layer is modelled after a shield, it needs to fulfill two main requirements [3], namely 1) minimal interference, i.e., that it only intervenes when actually necessary, and 2) correctness, i.e., that when it does intervene, it does not issue commands that are disadvantageous. In our case, the commands issued by the second layer are, by definition, considered unsafe, so we adjust this definition to mean that commands issued by the second layer do not carry an unacceptable risk.

Meeting the second requirement (correctness) is achieved by constructing an adequate method for evaluating states and moves. If this requirement is met, the first requirement will also be met, because a safe move will only be overridden by an unsafe move if that unsafe move is not too unsafe, and the metrics used for satisfying the first requirement can also be used for comparing unsafe moves with safe moves. We only need to define a threshold value that defines when the safe move is inferior to the unsafe move.

So, we will need metrics for evaluating states and moves. These metrics can then be combined in several ways to obtain a qualitative comparison. The method chosen should be able to distinguish between unsafe moves that create a real advantage and moves that may contribute in the short term, but are not necessarily more valuable than an allowed move. To find this method, we need to first determine what defines such a move.

4.2 State Score

Firstly, we need a measure of 'goodness' for the state resulting from a move. As discussed, the strategy chosen is aimed at retaining well-orderedness of the state, so a good way to ensure adherence to the strategy would be to encode this property in the agent's reward function. The *state score* is defined for each row and column as follows:

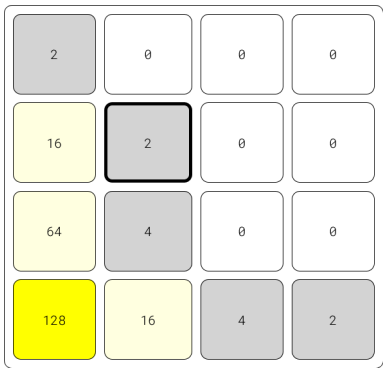
$$\text{StateScore}(S_x) = \begin{cases} 1, \text{if } S_x \text{ is well-ordered according to the chosen strategy} \\ 0, \text{otherwise} \end{cases}$$

where S is the state and x is the index of the row or column. The definition of well-orderedness is usually equivalent to the row or column being monotonically increasing (or decreasing) in some direction. The state score for the full state S is then defined as the sum of the scores for all the rows and

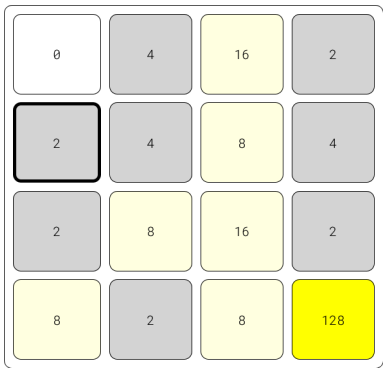
columns, i.e.:

$$\text{StateScore}(S) = \sum_{i=1}^{8}(\text{StateScore}(S_i))$$

For completeness, define the four rows (of a 4x4 grid) to be numbered 1 to 4, where 1 is the top-most row and 4 is the bottom-most row. The columns are numbered in a similar fashion, with 5 being the left-most column, and 8 being the right-most column. As an example, take the LEFT-DOWN strategy we have been using so far (LEFT and DOWN are the legal moves, UP and RIGHT the illegal moves). Figure 4.1 shows a number of states with their state scores according to this strategy.



(a) According to the LEFT-DOWN strategy, this grid is well-ordered and has a state score of 8 (the maximum).



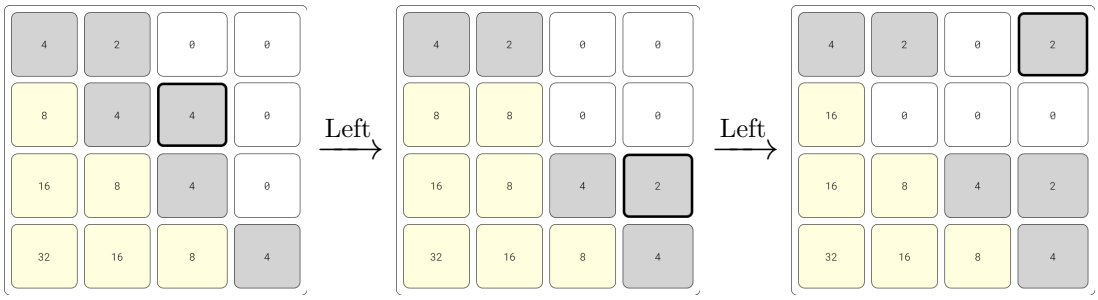
(b) This grid is not well ordered and has a state score of 1 (as only the left-most column is monotonically increasing).

Figure 4.1: 2 example grid arrangements.

Note that, for practical reasons, we also compute the horizontal and vertical state scores separately, in addition to the full state score as described above. The horizontal state score is the sum of the state score of all the rows, and the vertical state score is defined the same way, but for the columns.

4.3 Objective reward

We will also need a more objective measure of the value of a move. This measure will be defined as the sum of the tiles merged because of a move. Observe that this measure is more localised, as it only relates to pairs of tiles, regardless of their position on the grid. The state score, in contrast, is more global, as it relates to the entire state. Choosing the move that merges the tiles with the highest value ensures quick progression toward the objective and prevents the board from filling up with low to medium-valued tiles. Figure 4.2 shows a move progression and the associated move rewards.



(a) Initial state (b) Reward = 8 (c) Reward = 16

Figure 4.2: Move progression and rewards per move.

Going further, we will refer to this measure as the *reward* $\mathbf{R}$ of a move. This measure, however, is not the final form of the reward scheme we will use in the reinforcement learning agent. We then have

$$\mathbf{R} = \sum_{t \in T} (t)$$

where T denotes the values of all tiles created (as a result of merging two tiles) by a move.

4.4 Mergeable tiles

Finally, we need a measure of the *potential* value of a state. This measure should thus describe how much reward can be obtained if it were possible to perform all 4 moves at once in a state [1]. A good way to do this is to find all the pairs of tiles that can be merged in a state, i.e., all adjacent pairs with equal value. Some pairs may be more valuable than others, for example, a pair of tiles with value 2 is less valuable than a pair of tiles with value 128. To capture this in our measure, we can compute the sum of all mergeable pairs, denoted by $\mathcal{M}$. Further, let $M(g)$ denote all pairs of mergeable tiles, as a pair of coordinates (x, y) and (i, j) in some grid g, and $g_{i,j}$ the value of the tile at column i and row j in g. Then

$$\mathcal{M}(g) \quad = \quad \sum_{p \in M} (g_{p_x, p_y} + g_{p_i, p_j})$$

For an example, see figure 4.3.

[1] Because moves on the same axis are symmetric, we would technically only have to perform 1 move in each direction.

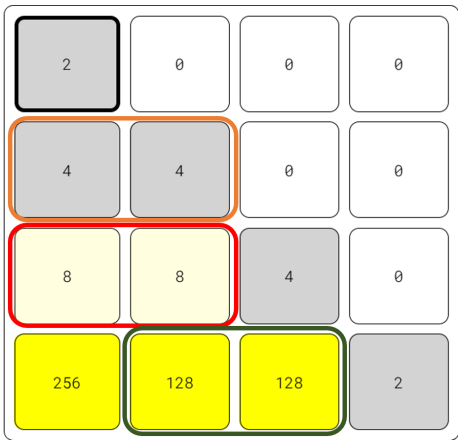


Figure 4.3: In this state, the mergeable pairs have been marked. The total value of all the mergeable pairs, $\mathcal{M}$, is 8 + 16 + 256 = 280.

4.5 Precedence of moves

One further thing worth noting is that not all unsafe moves are created equally. In the strategy we have been using so far, LEFT-DOWN, there are two unsafe moves: RIGHT and UP. It is easy to see that of these two, RIGHT, is the 'least' unsafe. We will demonstrate this with an example. Consider the initial state in 4.4, as shown on the left of that figure. The bottom row is, as a consequence of the LEFT-DOWN strategy, almost always the most critical to success. In 4.4, the bottom row is 'locked', i.e, moving right cannot alter it. Moving right thus has minimal effect on the state score and is very beneficial. Conversely, moving UP is very bad, as it spawns a low-valued tile underneath the highest-value tile. When we then move DOWN again, this tile will still be in place and we will have to work around it. This is indicative of a problem with the move UP under our chosen strategy. We can observe that moving UP is virtually always too risky, and almost never has better value. Because of this, we have to choose to only move DOWN, LEFT, sometimes RIGHT, and UP only if no other move is available. This establishes a certain order of precedence of the moves, which is useful when we have to implement our shield later on. Formally, we can define the 'locked row' phenomenon as follows:

$$\texttt{isRowLocked(i)} = \begin{cases} \mathbf{T}, \text{if } g_{1,i} \neq g_{2,i} \neq g_{3,i} \neq g_{4,i} \\ \mathbf{F}, \text{otherwise} \end{cases}$$

Note that a row does not need to be well-ordered to be locked, and vice-versa.

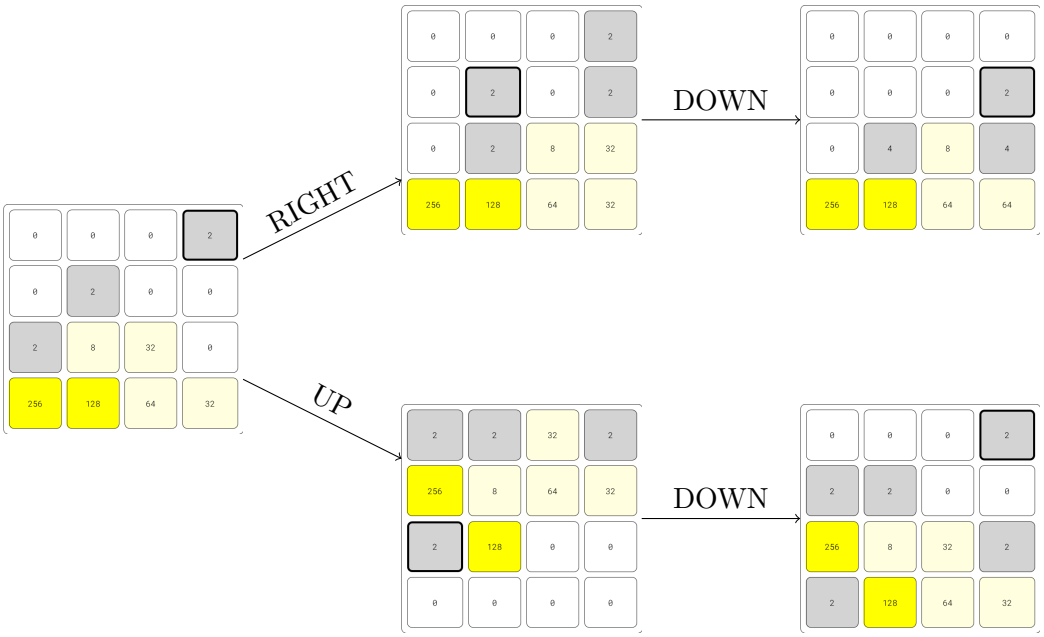


Figure 4.4

With the architecture of the agent and the necessary metrics defined, we can move on to the actual implementation of the agent, and the results it achieves.

Chapter 5

Implementation

For the implementation, we will use TensorFlow Agents [1], on Python 3.8.10. TensorFlow Agents recommends using Reverb [7] as the experience replay system, so we use it here as well. Further, to implement, for example, the grid dynamics and metrics, we used NumPy extensively [15]. We implemented three different agents in total:

- A greedy agent; this agent tries to maximise the reward (as defined in the previous section).
- A naive, safe agent; this agent makes only safe moves, unless a safe move is not possible.
- An optimised agent, based on the naive, safe agent.

The entire implementation is available on GitHub, at `https://github.com/TheRealKS/2048-solver`.
We will now discuss some details of the implementation that are worth mentioning.

5.1 Categorical Q network

Deep reinforcement learning methods tend to be quite unstable during training. This is a notorious problem [5], and an active field of study. Especially while learning a game like 2048, this problem manifests itself. In 2048, there is a high degree of non-determinism, as the placement of new tiles is random. Because of this, the input of actions by the agent naturally causes a high degree of variance in the training data. For Deep Q-learning (the method we will be using) several optimisations exist. Most prominently, we use a Categorical Deep-Q network [4], also known as C51. In this method, the distribution of the return for each state-action pair is learned, instead of just a single Q-value. This distribution is discretised into 51 bins, called atoms (hence the name C51). This requires knowing the range of the returns

$[V_{\min}, V_{\max}]$, the minimum and maximum value the return for a single action can take.
In our problem, this is easy to define beforehand. The lowest reward is 0, for termination. The highest reward is obtained by merging two 1024 tiles, to get the final 2048 tiles. So the range of rewards is $[0, 2048]$. This range does not take into account that more than two pairs of tiles may be merged at the same time. But as it is highly unlikely that we will, say, merge 4 1024 tiles in the same move, we disregard this possibility to avoid increasing the size of the range. As the range becomes larger, the bins will also become large (because the amount of bins is fixed). This will make the distribution estimation less precise.
Once the distribution of Q-values has been learned, the exact Q-value for a state-action pair can be determined by simply taking the mean of the learned distribution:

$$Q(s,a) = \sum_{i=1}^{N} (p_i(s,a) \cdot z_i)$$

Here $p_i(s,a)$ denotes the probability that the return for $Q(s,a)$ lies within the i^{th} bin. z_i denotes the i^{th} atom. Note that N will usually be 51, as that is what the authors of C51 [4] found to be the optimal number of atoms.
The big advantage of using this method is that, instead of trying to learn the average of the distribution of the return, we learn the entire distribution and can then precisely compute the average. In our case, as we have already shown, the return of the same action can vary greatly, because the return will also incorporate longer-term rewards. These longer-term rewards can be vastly different from game to game, based on the overall course of that game. Capturing this variance in the training data is thus very useful, as it means that the average (which is a measure notoriously sensitive to outliers) used to determine the optimal action will better capture all the episodes seen so far. This increases training stability and final performance.

5.2 Reward scheme

As already discussed, the reward scheme for our agent requires special attention. In general, using just the notion of reward as defined previously is too simplistic for non-greedy agents. If we want the agent to learn to adhere to a strategy, and by extension the state ordering, we need to enforce this by incorporating it into the final reward. This can be done by scaling the greedy reward r (the sum of all the merged tiles) by a scalar (defined to be between 0 and 1) s. We can obtain such a scalar easily by performing simple arithmetic on the state score as previously defined.
This means that the reward scheme (as used by all agents except the greedy

one) will be:

$$r' = s \cdot r$$

Using this scheme ensures that the agent will learn to adhere to the strategy, while at the same time also learning to distinguish between the two safe moves, as one of the two can be more valuable than the other, as demonstrated in figure 5.1.

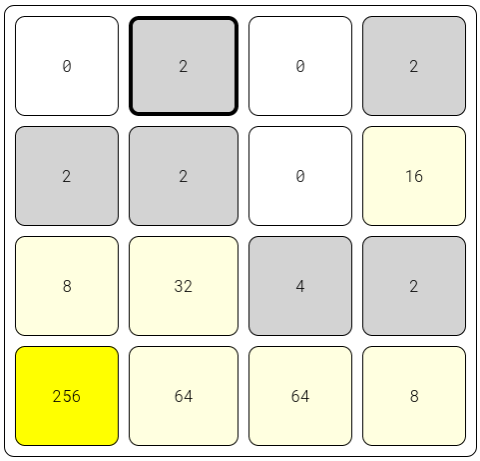


Figure 5.1: In this state, both safe moves (LEFT and DOWN) are available. It is clear, however, that LEFT is more valuable as it will merge the two 64 tiles into a 128 tile.

5.3 Masking

Even though using the reward scheme as defined should cause the agent to learn to adhere to the strategy, it will not learn to do so perfectly. When an unseen state is presented to the neural network, the network will try to predict the optimal action based on previous experience. This means that a certain randomness will be present. Especially in edge cases, an unsafe action might be chosen when a safe action is also available. In such a case, the constraints set by the strategy are violated. Also, enforcing exploration through, for example, an ϵ-greedy policy is necessary to reach optimal performance. But this may also cause unsafe actions to be selected and trained into the network. The reward scheme should disincentive such moves, which generally causes the agent to behave as intended, but in practice it can still occur that an unsafe move is selected by the agent even when that is unacceptable. To remedy this, we will need to be more aggressive in disallowing unsafe moves. Fortunately, TF-agents allows a mask to be defined on the actions available for selection. This is available both for the ϵ-greedy policy used and the agent itself. We will use this feature extensively to prevent

unsafe moves being chosen when the strategy disallows this. Furthermore, it allows us to disallow impossible moves at each step, which prevents the agent from getting stuck and selecting the same move over and over even though that move is not possible (and does not change the grid arrangement). With our heavy emphasis on safe moves during training this might happen, for example, if a safe move is not available and an unsafe move must be chosen (which the agent will be reluctant to do).

5.4 Implementing the reverse shield

For implementing the reverse shield, we have two different options. We could try to learn the behaviour in a separate agent, or we can construct a heuristic to determine if an unsafe move is superior. In our case, we choose the latter. Before this heuristic is computed, however, certain requirements must be met. Imposing these requirements will prevent unnecessary computations. The requirements for when the heuristic should be evaluated on the current state-action pair are as follows:

- The chosen action should be defined as safe by the current strategy.
- In the state under consideration, an unsafe move is available.
- The amount of moves played in the current episode exceeds a certain threshold.

The last requirement is necessary to prevent unsafe moves from being played in the early stages of the game when the groundwork is laid for maintaining good state ordering during the rest of the game. During these stages, it is more vital to maintain safety. A wrong move in the early stages of the game can sabotage the agent for the rest of the game and we should thus not work against the agent by replacing a safe move in the early stages of the game. A good example of why this is important can be found in figure 5.2.

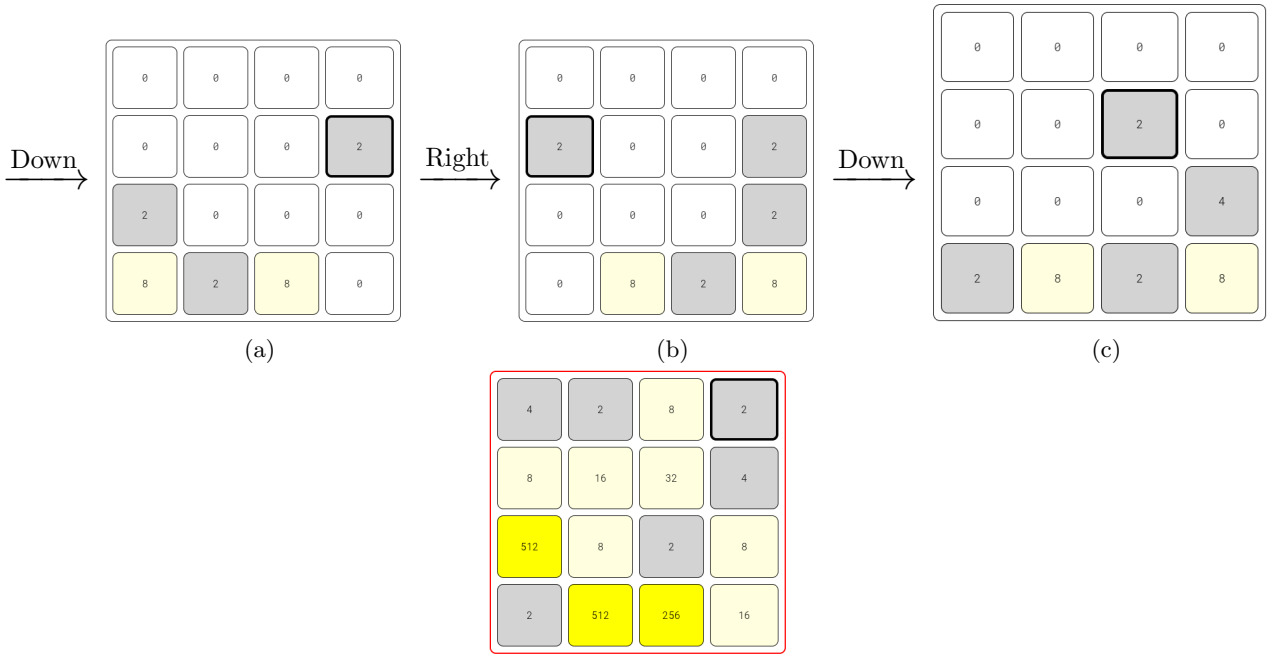


Figure 5.2: a) shows the 9^{th} state of a game. In this state, the chosen (safe) move is replaced by the shield with RIGHT, an unsafe move. As the heuristic used by the shield does not distinguish between low and high-valued tiles, RIGHT was determined to have better value. Because of this, a low-valued tile is moved into the bottom-left corner, where the most valuable tile should be. This tile is still in place in the final state of the episode. If this had not occurred, the agent would have had a better chance of progressing further, and maybe even winning the game, as evidenced by the high values of the other tiles.

Before we can compute the heuristic, we need to perform the safe move and collect metrics (the value of the mergeable tiles and the state score in the state after performing the safe move). Once the requirements for computing the heuristic have then been met, we can perform the unsafe move and collect the same metrics. The main metric we will use to determine if the unsafe move is better is the 'mergeable' metric. The state score is used to determine the risk presented by this unsafe move. Before the heuristic is computed, there is one additional check, namely that the unsafe move did not lead to termination of the game. Such moves are of course never advantageous. The heuristic used is best described using pseudocode, as in algorithm 1. For simplicity, we will refer to the horizontal state score as 'hscore', and the value of the mergeable tiles as 'mergeable'. We will also

use a function called `getNumEmptyRows` which returns the number of empty rows, and a function called `isRowLocked(i)`, which returns whether or not the i^{th} row is locked (as discussed before).

Algorithm 1 Reverse shield heuristic

procedure heuristic(*grid*, safe_hscore, unsafe_hscore, safe_mergeable, unsafe_mergable)
Require: *grid* is the grid after the unsafe move has been performed
1: **if** safe_hscore $\geq$ unsafe_hscore $\wedge$ safe_mergeable $=$ unsafe_mergeable **then**
2: **return** safe move is better
3: **else**
4: zero_rows $\leftarrow$ *grid*.getNumEmptyRows
5: **if** unsafe_hscore $-$ zero_rows $> 0 \wedge$ *grid*.isRowLocked(3) **then**
6: **if** unsafe_mergeable $>$ safe_mergeable **then**
7: **return** unsafe move is better
8: **end if**
9: **end if**
10: **return** safe move is better
11: **end if**
end procedure

Note that this particular version of the heuristic has been written to work for the strategy we've been using throughout this thesis, namely the down-left strategy, but that the procedure is analogous for other strategies. Also note that under our chosen strategy, we only have to evaluate this heuristic for RIGHT. We never consider UP to be a viable alternative if other moves are available, due to the order of precedence as established in 4.5.
Now, we will go through the checks one by one:

- Line 1 contains the first safety check. If the value of the mergeable tiles is the same for both the safe move and the unsafe move, and the horizontal state score after the unsafe move is not better, we err on the side of caution and choose the safe move.

- Line 5 contains the second safety check. This check is primarily to verify that the unsafe move presents an acceptable risk. The idea is that, if the bottom row is locked, the unsafe move will generally present an acceptable risk. To this end, we check that (1) the horizontal state score, not taking into account empty rows (which will always be well ordered), is at least 1, so at least one row is well-ordered, and (2) that the bottom row is locked. It might seem that one of these checks is redundant, as they seem to check for more or less the same thing. In certain cases that may indeed be possible, but in other critical edge

cases the combination of these checks can prevent disadvantageous moves. Under our strategy, the bottom row will generally be well-ordered if there is only 1 well-ordered row. But if a row is well-ordered, it does not mean that it is locked. If, for example, we consider a bottom row like in figure 5.3. We can easily see that performing LEFT (safe) is better. Critically, this row is well-ordered, but not locked, which means it passes the first check, but not the second. If the bottom row is locked, however, but not well-ordered, moving right is suboptimal, as we should be aiming to restore the ordering of the bottom row. This can not be done using RIGHT. As a result, both checks are necessary to make a good assessment.

- Line 6 contains the value check. Here, we check if performing the unsafe move actually has merit. We use the value of the mergeable tiles for this, as discussed before.

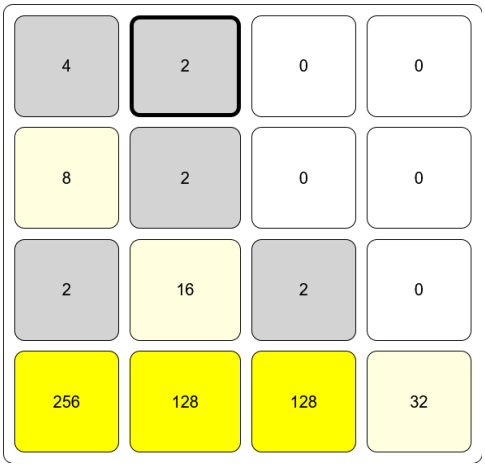


Figure 5.3: In this state, the bottom row is well-ordered, but not locked. This means LEFT is optimal.

With the heuristic defined, we can capture the entire procedure for the reverse shield in the following algorithm:

Algorithm 2 Reverse shield

procedure reverse_shield(*grid*, *previous_move*)
Require: *grid* is the grid returned directly after the previous move
1: prev_state_legal_moves ← *grid*.getLegalMoves
2: *safe_grid* ← *grid*.performMove(safe_move)
3: safe_hscore, safe_mergeable ← *safe_grid*.collectMetrics
4: **if** *previous_move* = LEFT ∨ *previous_move* = DOWN **then**
5: **if** RIGHT is in prev_state_legal_moves **then**
6: *unsafe_grid* ← *grid*.performMove(unsafe_move)
7: **if** ¬*unsafe_grid*.isGameOver **then**
8: unsafe_hscore, unsafe_mergeable ← *unsafe_grid*.collectMetrics
9: **return** heuristic(*unsafe_grid*, safe_hscore, unsafe_hscore, safe_mergeable, unsafe_mergeable)
10: **end if**
11: **end if**
12: **return** safe move is better
13: **end if**
end procedure

For an example of this procedure at work, see appendix A.

5.5 Reverse shield and training

Now that we have defined the reverse shield, we can think about how to integrate it with the agent. As mentioned before, we will be using the naive, safe agent as the baseline agent. One option is to train the naive agent, and then add the reverse shield, so that it will only be active when the policy of the naive agent has already been trained. Alternatively, we could deploy the reverse shield during training. In that way, the agent should learn the behaviour of the shield and be able to function on its own. While this approach has many advantages, it has one key disadvantage. Unsafe moves will need to be allowed at all times, but the main feature of the safe agent is that this is not the case. If we deviate from this notion of strict safety, the agent will learn to perform unsafe moves when that is, in fact, unacceptable. The reward scheme we are using, where the agent is penalised for reducing the state score, should theoretically make the agent replicate the heuristic's behaviour exactly. However, this is not the case in practice, as we have already discussed in section 5.3.
To strike a compromise between these two options, we will deploy the reverse shield during training, but the agent will not observe the moves chosen (i.e.,

use these moves as training data) by the reverse shield. During training, the agent will progress further in the game and observe states that would only occur in later stages of the game. This way, the safe policy is better trained for a wider variety of states, but it will not be necessary to deviate from strict safety.

Chapter 6

Results

In this section, we will evaluate the performance of the three agents and see how they compare.

6.1 Preliminaries

All experiments were run with an Intel Core i7 Processor, 8GB of ram, and an NVIDIA GTX1050Ti as an accelerator, on the Linux 5.11 kernel.
For all agents and all experiments, unless otherwise noted, the same parameters are used. The general setup uses the categorical Q network, as discussed before, with a neural network containing 128 fully connected hidden layers. A standard ϵ-greedy policy is used with the probability for a random action set to 0.1. Below, we will note the values used for the hyperparameters.

- Learning rate: 0.001
- Discount factor γ: 0.95
- Q target network update interval: 50 episodes
- Replay buffer sample size: 64
- Replay buffer capacity: 10000 observations

The data was gathered by evaluating the policy learned every 25 episodes. At each evaluation point, 10 episodes are run by the policy as it is at that point learned by the agent. Of these 10 episodes, the mean length and mean return value is recorded, as well as the maximum tile value reached in any one of the 10 episodes.

6.2 Greedy agent

The greedy agent, as described before, uses the reward metric **R** without any scaling. This agent, as a result, uses a fairly random strategy and does not achieve very good results. Figure 6.1 shows the average reward and average episode length plotted. Figure 6.2 shows a plot of the maximum tile value reached. As can be seen, the agent does not improve much. This fits what we have theorised before, namely that when only the immediate reward is maximised, the overall performance will be quite bad. However, the agent is in fact optimising somewhat as evidenced by the ever-decreasing loss values, as shown in figure 6.3. The agent does manage to achieve a 512 tile sometimes, probably because it learns how to combine high-value tiles and is thus not entirely random, but the most frequent highest tile value is 256. Figure 6.4 shows the return and episode length for 10 episodes played with an entirely random policy (no learning whatsoever). We can now see that the greedy agent is, in effect, not much better than this. To show this further, figure 6.5 shows 3 final states reached by the greedy agent after training 1000 episodes. Figure 6.6 shows the same, but for the random policy.
It is also interesting to note that the results are quite unstable. This is a hard problem to solve. As discussed previously, reinforcement learning is notoriously unstable during training. Especially with a fairly random distribution of returns (the outcome of the game can vary drastically from episode to episode). This also means that applying a policy that worked well in the previous episode might be counterproductive in the next. As we have already seen, the spawning of a single, low-valued tile in an unfortunate position can be disastrous for the outcome of the game. This means that, even with countermeasures in place, the training will unfortunately be relatively unstable. This is a problem that would require further investigation and experimentation to solve.

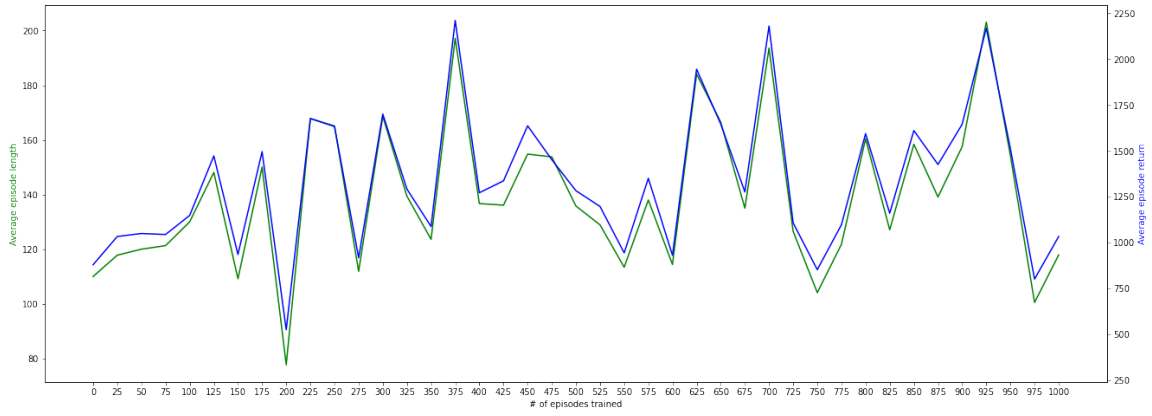


Figure 6.1: Average return vs average reward of the greedy agent.

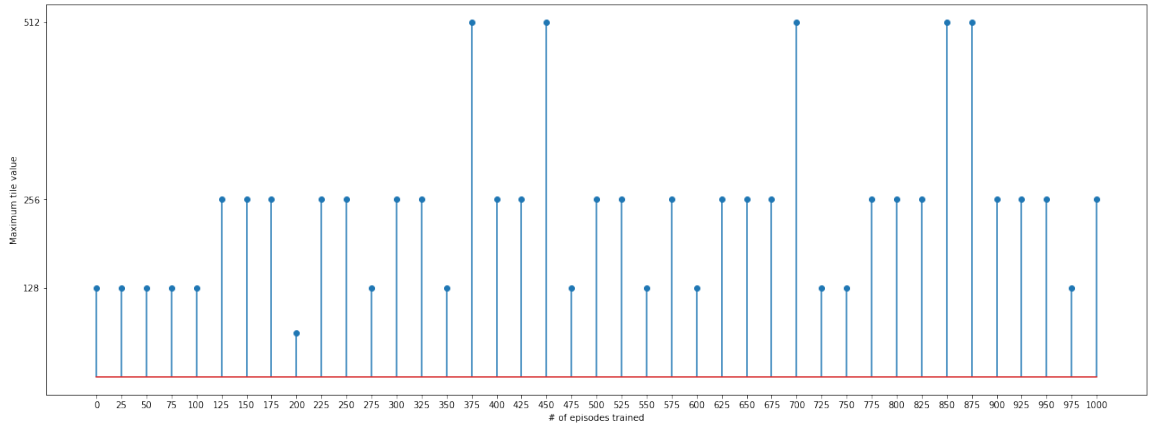


Figure 6.2: Maximum tile values achieved by the greedy agent.

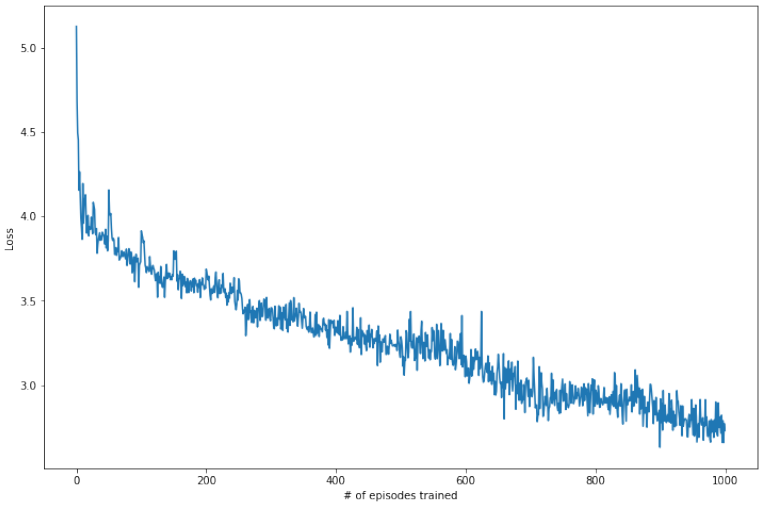


Figure 6.3: Loss achieved by the greedy agent.

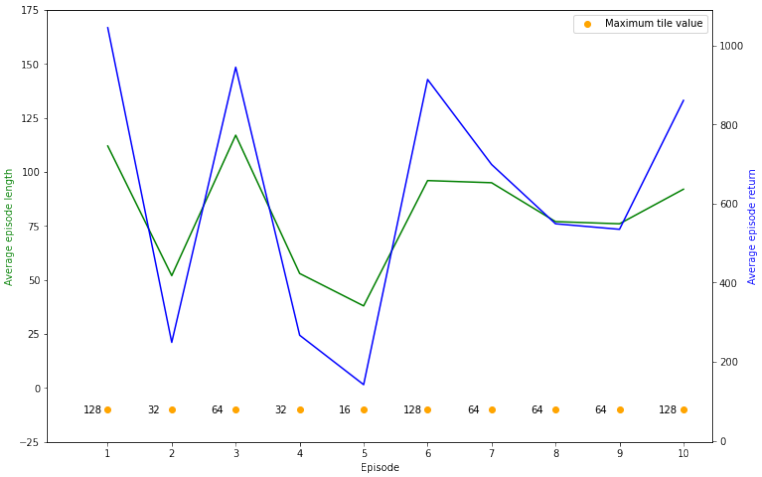
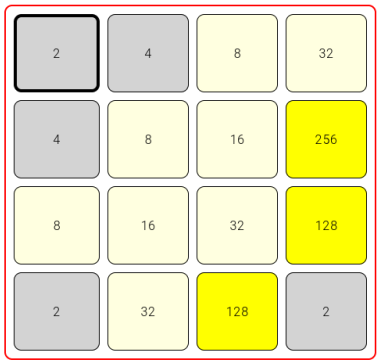
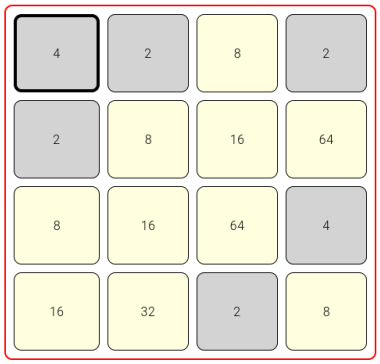


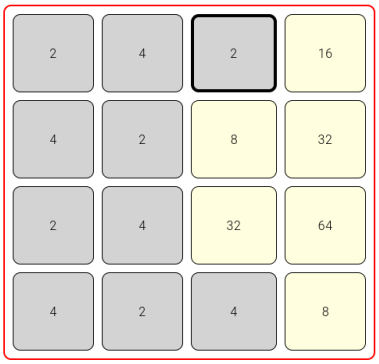
Figure 6.4: Results achieved by a random policy.



(a) Highest tile = 256

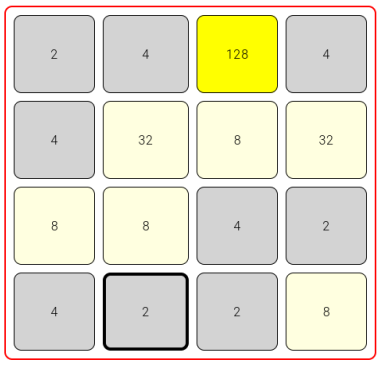


(b) Highest tile = 64

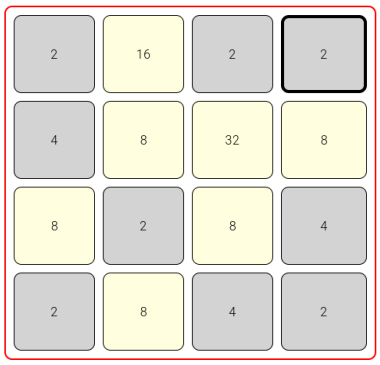


(c) Highest tile = 64

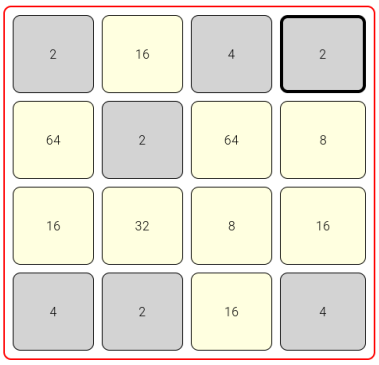
Figure 6.5: Final states achieved by the greedy agent.



(a) Highest tile = 128



(b) Highest tile = 32



(c) Highest tile = 64

Figure 6.6: Final states achieved by the random agent.

6.3 Naive agent

The naive agent plays only safe moves unless such a move is unavailable. This means that for this agent, safety is strictly observed. This agent achieves much better results already than the greedy agent. While the variance in the return is still high, an upward trend is visible. We can see in figure 6.8 that the maximum achieved tile is never lower than 256, and 512 is reached more regularly. Most importantly, however, the kinds of states reached are much more usable and have higher potential. Figure 6.9 shows the final state of a game played by the naive agent. Clearly, this state is much more favourable than any of the states achieved by the greedy agent (as shown in figure 6.5).

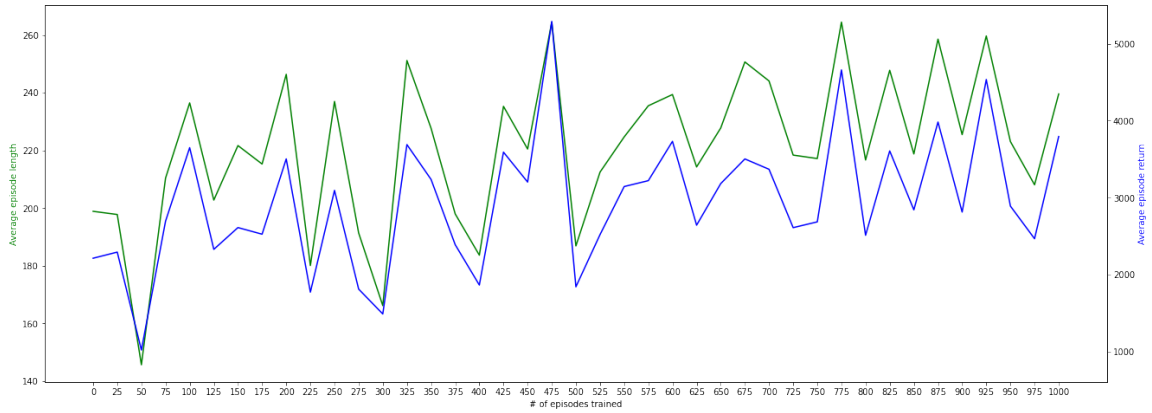


Figure 6.7: Average return vs average reward of the naive agent.

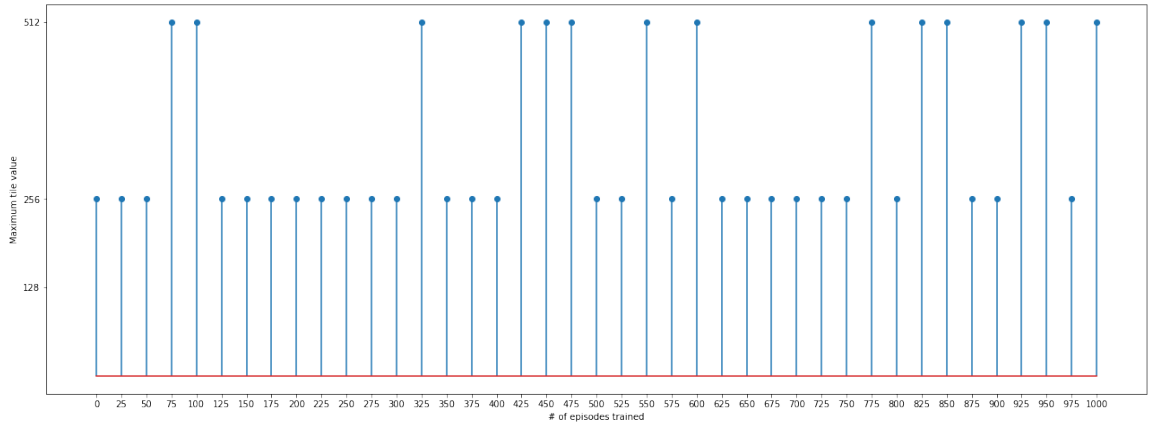


Figure 6.8: Maximum tile values achieved by the naive agent.

Crucially, however, analysing these states once again shows that the strictly safe policy is sub-optimal. In figure 6.9, the tiles are available to form a 128 tile, which might have allowed us to continue the game. Further analysing this game shows that game-over would have been averted if the reverse shield had been in place. Consider the initial state in figure 6.10, which is the state a few moves before the final state in figure 6.9. In this state, choosing RIGHT would have merged the two 32 tiles into a 64 tile, and positioned that tile above the existing 64 tile. Because the naive agent will never consider such a move, however, this is not done and the agent chooses LEFT. As a result, the two 32 tiles are still merged, but the resulting tile is not positioned correctly and the game ends soon after. It is clear then, that while the naive agent is much better than the greedy agent, there is room for improvement. We will now see that using the reverse shield does just that.

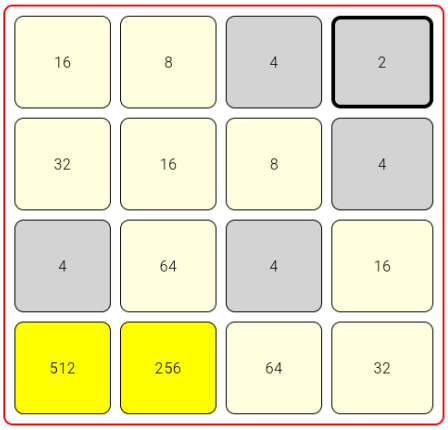
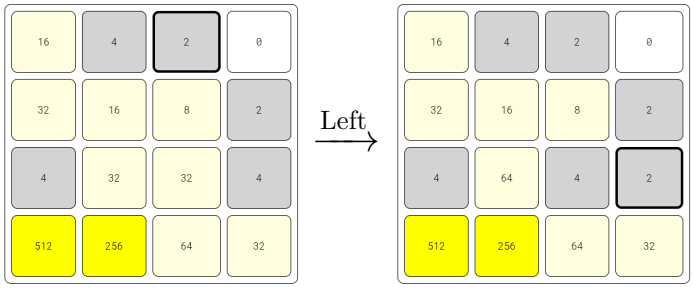


Figure 6.9: Final state achieved by the naive agent.



(a) (b)

Figure 6.10: Move progression chosen by the naive agent.

6.4 Reverse shielded Agent

This agent achieves by far the best results. Here, we trained for 500 episodes total. Figure 6.11 shows the average reward vs average lengths. Stability is, once again, not great, but an upward trend is visible, and after 225 episodes, a 2048 tile is achieved, as shown in figure 6.12. After this, the agent stabilises somewhat, before performance peaks quite high and falls back down again. A long, annotated move progression can be found in appendix A.

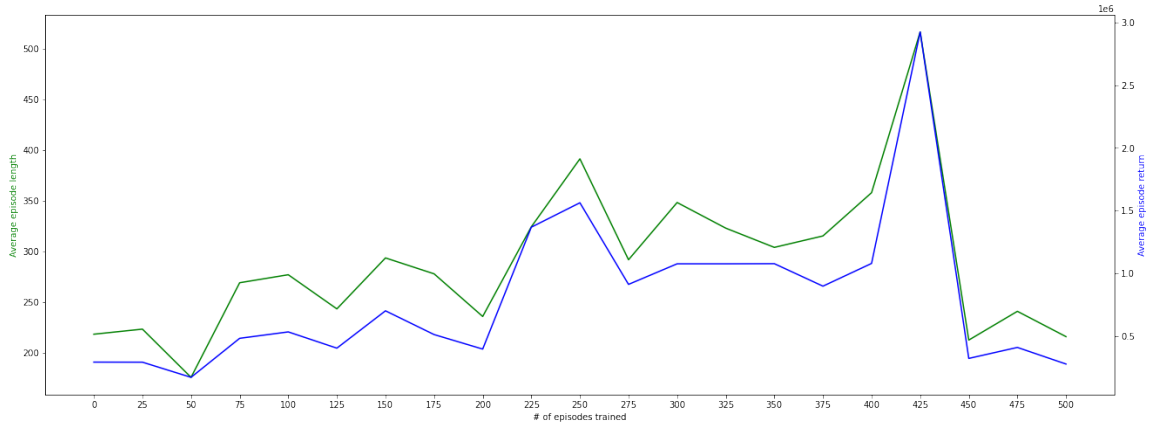


Figure 6.11: Average return vs average reward of the reverse shielded agent.

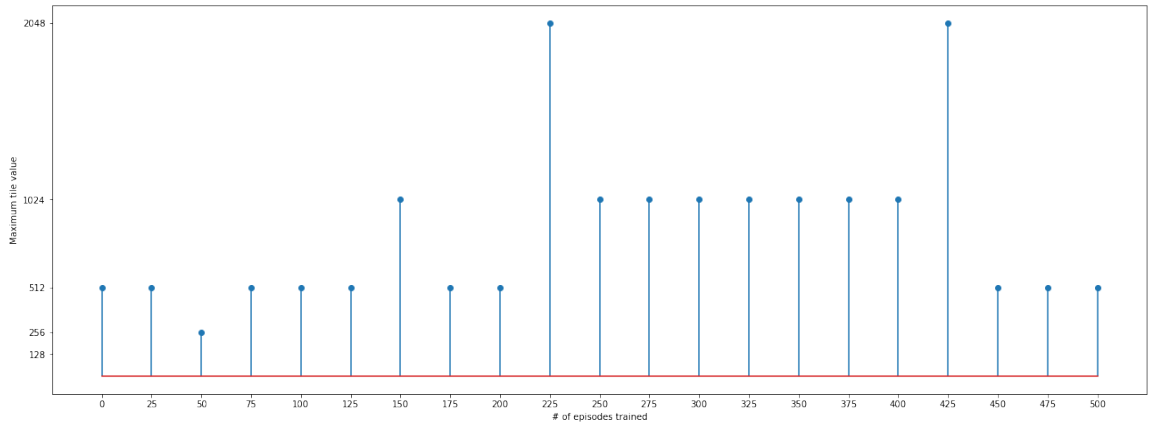


Figure 6.12: Maximum tile values achieved by the reverse shielded agent.

Most importantly, however, a 2048 tile is achieved twice, and this agent is able to easily reach a 1024 tile. Also note that the measure of the maximum tile, while more concrete than the average cumulative return, does not take into account whether, for example, two 1024 tiles exist on the board, but it is not possible to merge them. The maximum tile value measure does not

capture this. The average length measure can do a better job here, but as evidenced by figures 6.11 and 6.12, the number of moves needed to reach a certain tile might differ greatly. If the agent is inefficient, i.e., it creates a lot of high-valued tiles that cannot be merged, the average length, as well as the average return, will be high, but the maximum tile value may still be quite low. The reverse shielded agent should be more efficient than the naive agent. To show this, we trained both agents for 250 episodes and recorded the average of the sums of all the tiles on the board at each evaluation. Figure 6.13 shows the results of this.

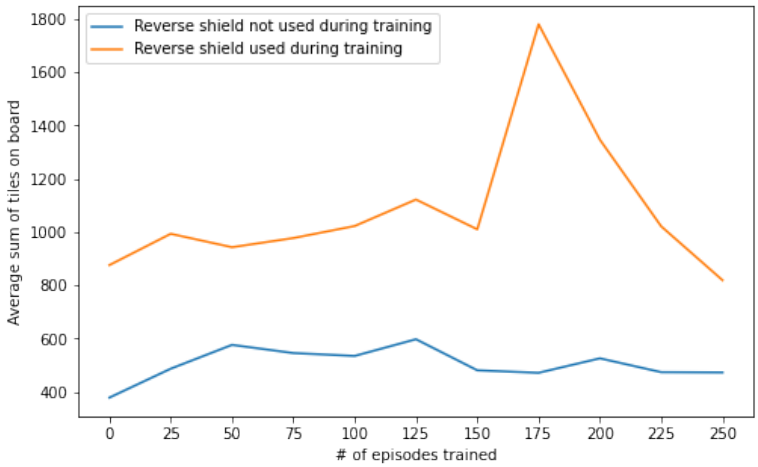


Figure 6.13: Sum of the tiles on the board of the reverse shielded agent and the naive agent.

It is very clear that the reverse shielded agent always performs much better than the naive agent. For completeness, see figure 6.14 for the maximum tile values achieved by both agents. Here, it can be seen clearly that the higher sum of tiles value corresponds to a higher maximum tile value in most of the cases. This shows that the reverse shielded agent is able to create more high-valued tiles, and while they may not always be mergeable, this is something that a further improved agent could take advantage of.

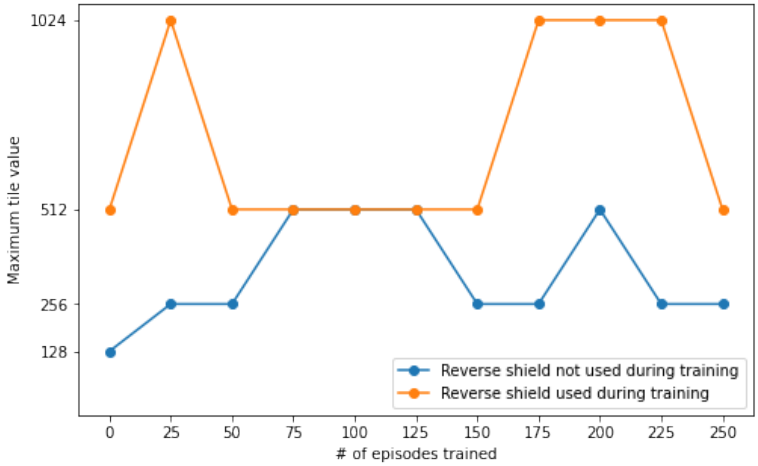


Figure 6.14: Sum of the tiles on the board of the reverse shielded agent and the naive agent.

Most importantly, however, is that we want our agent to create opportunities and then use them to its advantage. To show that the reverse shielded agent does this best, figure 6.15 shows how many times tiles of values between 32 and 1024 were lined up, i.e., able to be merged, for each of the 3 agents. Each agent was trained for 500 moves. The results were taken as an average of the results for 10 games played after training had finished.

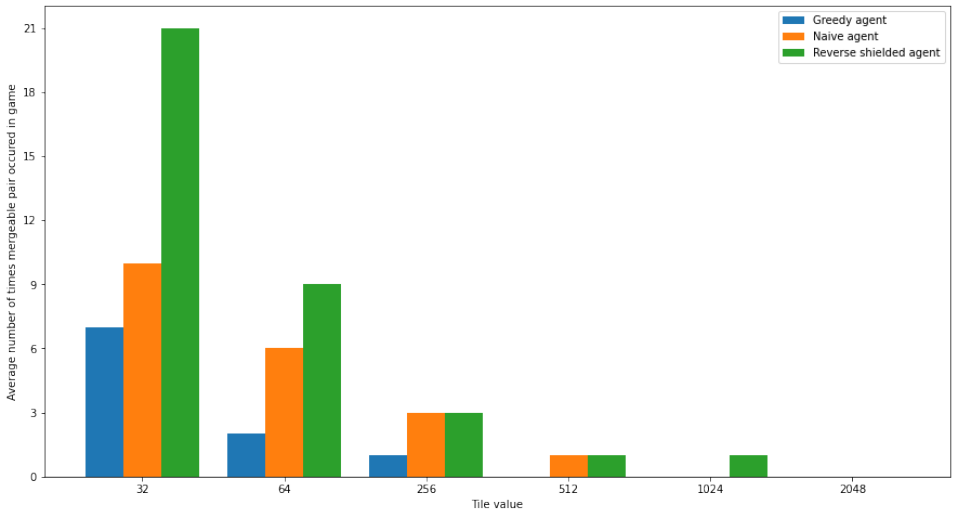


Figure 6.15: Bar plot showing how often certain pairs of tiles were lined up to be merged. The greedy agent never lines up tiles with a value higher than 256, and none of the agents achieve this for 2048 tiles.

Here, we can again clearly see that the naive agent outperforms the greedy

agent and that the reverse shielded agent outperforms the naive agent. Interestingly, we can see that at some point, the reverse shielded agent lined up two 2048 tiles, but did not merge them. This shows that while the reverse shielded agent performs well, it could still do better. The underlying strategy of the naive agent, could, for instance, be improved. Then, the two 1024 tiles might have been combined and the game would have been won. It is also possible that in that particular state an unsafe move would have merged the two tiles, but that the heuristic used did not evaluate that move as better.

Chapter 7

Related Work

Ever since 2048 was first released in march of 2014, solving the game using a computer player has been a popular subject of study. As such, many different approaches have been described, with a wide variety of results. Besides published work, many different efforts using a variety of different methods can be found across the internet. Early efforts used tree search, with carefully designed evaluation functions. These methods, therefore, use *expert knowledge*, much like our reverse shielding approach.
Later methods use non-deep temporal difference learning (TDL), such as [25] and [26]. In the former of these, the authors were able to win (achieve a 2048 tile) in 97% of games. In the latter, the authors used TDL in a 3-stage approach, performing TDL on a different aspect of the game at each stage. Using this method, they are able to always achieve a 2048 tile, and even achieve a 32768 tile in 10.9% of games. Later techniques, such as the one described in [13], managed to increase this to 72%.
Further, attempts have been made to solve 2048 using supervised learning, such as in [17]. Here, a network is trained to classify states into 4 classes, corresponding to the 4 moves. The training data was obtained by running a number of strong, TDL-based agents. This approach was successful, achieving a 2048 tile in 98.8% of played games.
The first attempt at using deep reinforcement learning for 2048 used a standard deep-Q network [12]. Here, the authors based their method on the first version of deep-Q learning, as described in [19]. This method uses a convolutional neural network, which was suitable for playing Atari games in [19], but this method did not work well for 2048. The highest tile achieved using this method was 1024. The authors conclude that a convolutional network is not suitable for a game like 2048, because the state is compact and patterns in the grid arrangement are not very meaningful. In [10], the authors also tried to use deep-Q learning, but also did not achieve a 2048 tile. The authors also discuss incorporating expert knowledge, by implementing a state evaluation function, like the state score we have described. Using this

metric, they try to learn human-like playing behaviour by using the state evaluation metric to enforce adherence to a strategy, much like we do in our approach. Ultimately, however, this specific approach was unsuccessful.
So far, the best, published results have been achieved using TDL. It is worth noting, however, that several users on the internet claim they have been able to achieve very high win rates using deep-Q learning, like in [18]. Here, the author claims to achieve 1024 in all games, and 2048 in 40% of games. The specific results claimed here, however, could not be verified, because the specific implementation used was not publically available.

Chapter 8

Conclusion

In this thesis, we have investigated improving reinforcement learning performance for agents playing 2048 by using expert knowledge. We have made a number of key observations, namely:

1. To win in 2048, a strategy is required. Throughout this thesis, we have used the LEFT-DOWN strategy.

2. However, deviation from the strategy is needed at certain key points in the game.

3. Learning such a nuanced strategy using deep reinforcement learning requires a new approach.

Enforcing the strategy in the naive agent has been shown to yield superior results, as opposed to a greedy agent. We have found that in order to solve the key problem - learning a nuanced strategy - incorporating expert knowledge into the policy through a process similar to shielding worked well. Through our method, we are able to reliable achieve a 1024 tile, and on occasion we are even able to win the game by achieving a 2048 tile. This presents an improvement over previous techniques, though TDL-based methods are still superior. The main shortcomings of our technique are that the training process lacks stability, a common problem in deep RL. Implementing countermeasures, such as the categorical Q-network, have helped somewhat, but improving the training stability is an important topic for future work.
Another topic for future work is learning the nuanced policy directly into the final policy, instead of requiring a second layer to be active at all times. Currently, the shield-like structure requires a rigid heuristic to be evaluated for every single move, and the expert knowledge used is not learned into the agent's policy. Learning the interventions of the reverse shield into the final policy would require relaxing the requirement that the agent adheres to the strategy, which yields inferior performance. Lastly, the code used by the

agent is very suitable for research and evaluation uses in its current form. However, training performance could be increased by optimising the code. Doing so could make training more episodes feasible, and might then as a result also improve performance and stability.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Pieter Abbeel and Andrew Y. Ng. Exploration and apprenticeship learning in reinforcement learning. In *Proceedings of the 22nd International Conference on Machine Learning*, ICML '05, page 1–8, New York, NY, USA, 2005. Association for Computing Machinery.

[3] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. *CoRR*, abs/1708.08611, 2017.

[4] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017.

[5] Lucian Buşoniu, Tim de Bruin, Domagoj Tolić, Jens Kober, and Ivana Palunko. Reinforcement learning for control: Performance, stability, and deep approximators. *Annual Reviews in Control*, 46:8–28, 2018.

[6] Steven Carr, Nils Jansen, Sebastian Junges, and Ufuk Topcu. Safe reinforcement learning via shielding for pomdps. *arXiv preprint arXiv:2204.00755*, 2022.

[7] Albin Cassirer, Gabriel Barth-Maron, Eugene Brevdo, Sabela Ramos, Toby Boyd, Thibault Sottiaux, and Manuel Kroiss. Reverb: A framework for experience replay, 2021.

[8] G Cirulli. 2048. `http://gabrielecirulli.github.io/2048/`, 2014.

[9] Jeffery A Clouse and Paul E Utgoff. A teaching method for reinforcement learning. In *Machine learning proceedings 1992*, pages 92–101. Elsevier, 1992.

[10] Antoine Dedieu and Jonathan Amar. Deep reinforcement learning for 2048. `http://www.mit.edu/people/adedieu/pdf/2048.pdf`. [Online, accessed 07-10-2021].

[11] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[12] H Guei, T Wei, JB Huang, and IC Wu. An early attempt at applying deep reinforcement learning to the game 2048. In *Workshop on Neural Networks in Games*, 2016.

[13] Hung Guei, Lung-Pin Chen, and I-Chen Wu. Optimistic temporal difference learning for 2048. *CoRR*, abs/2111.11090, 2021.

[14] Alexander Hans, Daniel Schneegaß, Anton Maximilian Schäfer, and Steffen Udluft. Safe exploration for reinforcement learning. In *ESANN*, pages 143–148. Citeseer, 2008.

[15] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[16] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *In Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163. Morgan Kaufmann, 1994.

[17] Kiminori Matsuzaki. A further investigation of neural network players for game 2048. In *Advances in Computer Games: 16th International Conference, ACG 2019, Macao, China, August 11–13, 2019, Revised Selected Papers*, page 53–65, Berlin, Heidelberg, 2019. Springer-Verlag.

[18] Anav Mehta. Reinforcement learning for constraint satisfaction game agents (15-puzzle, minesweeper, 2048, and sudoku). *CoRR*, abs/2102.06019, 2021.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[21] Teodor Mihai Moldovan and Pieter Abbeel. Safe exploration in markov decision processes. *CoRR*, abs/1205.4810, 2012.

[22] Martin Pecka and Tomas Svoboda. Safe exploration techniques for reinforcement learning – an overview. In Jan Hodicky, editor, *Modelling and Simulation for Autonomous Systems*, pages 357–375, Cham, 2014. Springer International Publishing.

[23] Stefan Pranger, Bettina Könighofer, Martin Tappler, Martin Deixelberger, Nils Jansen, and Roderick Bloem. Adaptive shielding under uncertainty. *CoRR*, abs/2010.03842, 2020.

[24] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 9(5):1054, 1998.

[25] M. Szubert and W. Jaśkowski. Temporal difference learning of n-tuple network for the game 2048. In *IEEE Computational Intelligence and Games 2014*, pages 1–8, Dortmund, Germany, 2014. IEEE Press.

[26] I-Chen Wu, Kun-Hao Yeh, Chao-Chin Liang, Chia-Chuan Chang, and Han Chiang. Multi-stage temporal difference learning for 2048. In Shin-Ming Cheng and Min-Yuh Day, editors, *Technologies and Applications of Artificial Intelligence*, pages 366–378, Cham, 2014. Springer International Publishing.

Appendix A

The reverse shield at work

On the following page is a sequence of moves achieved by the reverse shielded agent.

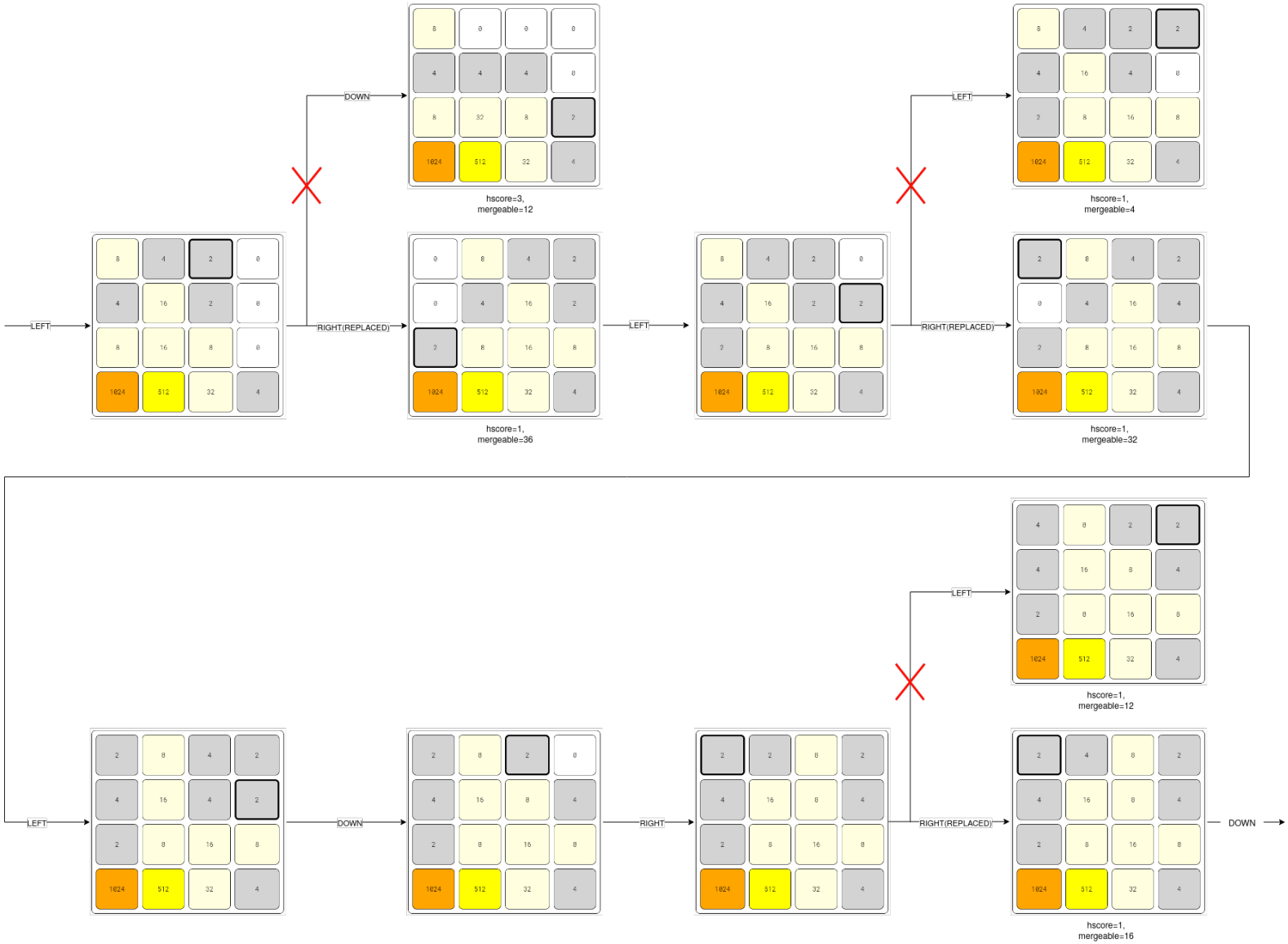


Figure A.1: Sequence of moves as achieved by the reverse shielded agent.