Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

# Reinforcement learning in 2048 game

Bachelor thesis

2018
Adrián Goga

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

# Reinforcement learning in 2048 game

Bachelor thesis

| | |
|---|---|
| Study programme: | Informatics |
| Field of study: | 2508 Informatics |
| Study department: | Department of Informatics |
| Advisor: | prof. Ing. Igor Farkaš, Dr. |

Bratislava, 2018
Adrián Goga

87372509

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

# THESIS ASSIGNMENT

| | |
|---|---|
| **Name and Surname:** | Adrián Goga |
| **Study programme:** | Computer Science (Single degree study, bachelor I. deg., full time form) |
| **Field of Study:** | Computer Science, Informatics |
| **Type of Thesis:** | Bachelor´s thesis |
| **Language of Thesis:** | English |
| **Secondary language:** | Slovak |

**Title:** Reinforcement learning in 2048 game

**Annotation:** The tabular 2D grid game 2048, with only 4 possible actions, attracted attention in machine learning community in recent years. As a result, the top AI approaches (that include planning) yielded very high scores (~10^6) going beyond human capabilities by an order of magnitude. There appeared also a few approaches based on reinforcement learning (RL) trying to master this difficult game with a stochastic component.

**Aim:** 1. Study the literature about reinforcement learning and artificial feedforward neural networks.
2. Implement the chosen RL algorithm and test it, potentially using its variants, on playing 2048 game.
3. Compare the performance of the models and interpret the results.

**Literature:** Russell S., Norwig P. (2010). Artificial Intelligence: A Modern Approach, (3rd ed.), Prentice Hall.
Haykin S. (2009). Neural Networks and Learning Machines (3rd ed.). Upper Saddle River, Pearson Education.
Sutton R., Barto A. (1998). Reinforcement Learning: An Introduction, MIT Press.

| | |
|---|---|
| **Supervisor:** | prof. Ing. Igor Farkaš, Dr. |
| **Department:** | FMFI.KAI - Department of Applied Informatics |
| **Head of department:** | prof. Ing. Igor Farkaš, Dr. |
| **Assigned:** | 11.10.2017 |
| **Approved:** | 30.10.2017 |

doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.......................................................... ..........................................................

Student Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# ZADANIE ZÁVEREČNEJ PRÁCE

| | |
|---|---|
| **Meno a priezvisko študenta:** | Adrián Goga |
| **Študijný program:** | informatika (Jednoodborové štúdium, bakalársky I. st., denná forma) |
| **Študijný odbor:** | informatika |
| **Typ záverečnej práce:** | bakalárska |
| **Jazyk záverečnej práce:** | anglický |
| **Sekundárny jazyk:** | slovenský |

**Názov:** Reinforcement learning in 2048 game
*Učenie hry 2048 pomocou posilňovania*

**Anotácia:** Tabuľková hra 2048 na 2D mriežke, s iba 4 možnými akciami, v posledných rokoch pritiahla pozornosť komunity strojového učenia. Výsledkom toho je, že najlepšie prístupy UI (ktoré zahŕňajú plánovanie) priniesli veľmi vysoké skóre (~10^6), ktoré rádovo presahuje schopnosti človeka. Objavili sa aj prístupy založené len na učení posilňovaním, ktorých cieľom je naučiť sa túto ťažkú hru, ktorá má aj stochastickú zložku.

**Cieľ:** 1. Naštudujte literatúru o učení posilňovaním (RL) a dopredných umelých neurónových sieťach.
2. Implementujte zvolený algoritmus RL a otestujte ho, potenciálne v rôznych variantoch, pri hraní hry 2048.
3. Porovnajte správanie modelov a interepretujte výsledky.

**Literatúra:** Russell S., Norwig P. (2010). Artificial Intelligence: A Modern Approach, (3rd ed.), Prentice Hall.
Haykin S. (2009). Neural Networks and Learning Machines (3rd ed.). Upper Saddle River, Pearson Education.
Sutton R., Barto A. (1998). Reinforcement Learning: An Introduction, MIT Press.

| | |
|---|---|
| **Vedúci:** | prof. Ing. Igor Farkaš, Dr. |
| **Katedra:** | FMFI.KAI - Katedra aplikovanej informatiky |
| **Vedúci katedry:** | prof. Ing. Igor Farkaš, Dr. |
| **Dátum zadania:** | 11.10.2017 |
| **Dátum schválenia:** | 30.10.2017 |

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

študent

vedúci práce

**Acknowledgements:** I would like to thank to my supervisor prof. Ing. Igor Farkaš, Dr. for an interesting thesis assignment and his willingness to help anytime I needed. At the same time, I would like to thank to my family and friends for their unique support.

# Abstrakt

Cieľom práce bolo preskúmať možnosti vytvorenia agenta založeného na učení posilňovaním, ktorý by sa dokázal naučiť logickú hru 2048. Táto hra predstavuje pre agenta neľahkú úlohu, keďže obsahuje prvok náhodnosti. Navrhli sme a otestovali základný model agenta využívajúceho hlbokú doprednú neurónovú sieť, spolu s niekoľkými modifikáciami pôvodnej hry a dvoma spôsobmi kódovania stavu v sade experimentov.

Agenta sme implementovali pomocou knižnice Keras s pozadím TensorFlow v jazyku Python. Na vizualizáciu priebehu učenia sme použili knižnicu Matplotlib. Natrénované agenty sme nechali odohrať 10000 hier a porovnali sme ich výkon s agentom vyberajúcim akcie náhodne. Aj keď sme nedosiahli očakávanú úroveň úspešnosti využívajúc pôvodnú odmeňovaciu funkciu hry, podarilo sa nám agenta naučiť dosahovať relatívne dobré výsledky pomocou jej modifikácie, čo považujeme za zaujímavé zistenie. Najlepší model dokázal dosiahnúť políčko 2048 vo viac ako 7% testovacích hier.

**Kľúčové slová:** Učenie posilňovaním, Umelá inteligencia, Dopredné neurónové siete, Logické hry

# Abstract

The goal of this thesis was to research the possibilities of creating a reinforcement learning based agent, that could learn to play the 2048 board game. This game introduces a difficult task due to its factor of randomness. We designed and tested a base model of the agent using of a deep feedforward neural network, together with a few modifications of the original game and two types of input encoding in a number of experiments.

We implemented the agent using the Keras library with TensorFlow backend in Python language. For visualization of its performance we used Matplotlib library. We let the trained agents play 10000 games and compared their performance to an agent that selects actions randomly. Even as we did not achieve the expected level of performance with the original reward function, we were able to train the agent to achieve reasonably good results using its modification, which we consider an interesting finding. The best model reached the 2048 tile in more than 7% of the testing games.

**Keywords:** Reinforcement Learning, Artificial Intelligence, Feedforward Neural Networks, Board Games

# Contents

# List of Figures

# List of Tables

# Introduction

Reinforcement learning (RL) has been on the rise in the last years, mostly because of its ability to learn how to play games that were not possible to be handled by an AI player before. A notable example of such a game is Go, a grid based puzzle of whose number of possible games is much larger than in chess, even far exceeding the number of atoms in the observable universe. For this reason it was resilient to traditional tree search AI methods until the recent success by Google DeepMind (Silver et al., 2016). Their approach was based on training deep neural networks with reinforcement learning by self play with the combination of tree search and observing human experts, and was able to win the most of the games with the world champion Lee Sedol.

Reinforcement learning is a methodology that resembles the way learning happens in nature. In contrast with the standard AI approaches, like tree search or supervised learning, RL algorithms learn by trying and observing the environment they are placed in, i.e. much like animals or human beings. Therefore it is important in autonomous robotics, where robots have to learn how to deal with their surroundings.

Board games introduce an interesting challenge for reinforcement learning algorithms. They often require planning ahead and sometimes to deal with randomness, which is a difficult setting when the algorithm does not perform exhausting search of possibilities, but rather learns how to play by trial and error. One of such board games that both include randomness and require a portion of planning is 2048. Developed in 2014 by Gabrielle Cirulli as a free web application, it became immediately very popular. Human players are able to learn a strategy that often leads to a good score, if not to winning the game. Our goal was to find out whether RL algorithms can achieve the same level of abstraction.

We chose a particular RL algorithm called Deep Q-learning and performed a set of experiments with the intention to find a setting in which our model can learn how to play the game as well as human players.

# Chapter 1

# Description of the 2048 game

Although the 2048 became viral immediately upon its release in March 2014, the mechanics of the game may not seem immediately obvious. For that matter we will describe the gameplay and create a coherent overview of the game's aspects.

## 1.1 Game overview

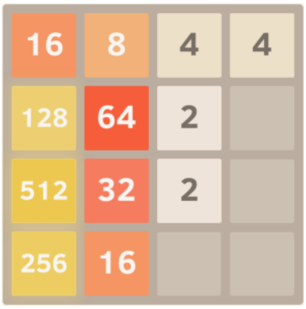
2048 is a single-player puzzle game developed by Gabriele Cirulli [1] during a single weekend. The game gained a significant amount of popularity due to its addictive gameplay style. The game is based on 1024 by Veewo Studio and is similar to Threes! by Asher Vollmer. 2048 is played on a board of size $4 \times 4$ where the player in each turn slides the board in one of the four directions in order to merge tiles of equal values and create tiles of higher values.

Initially, there are two tiles on the board. Their positions are chosen randomly and the values of the tiles are either 4 or 2 with probabilities 0.1 and 0.9, respectively. A player in each turn performs an action from the set $\{Up, Down, Left, Rright\}$. Each such action causes all tiles on the board to slide to the associated direction, stopping only on collision with the border of the board or with another tile. When two tiles of equal value collide, they merge into a tile of their sum. After each such turn, a new tile of value 4 or 2 spawns randomly on a an empty place with the same probabilities as initially. However, performing an action is only possible if that action will slide at least one tile.

For each pair of merged tiles of value $2^x$ the player gets $2^{x+1}$ points. The goal of the game is to obtain the tile of value 2048, but can be played further after reaching it and higher valued tiles can be obtained. [2] In this work, we will assume the goal of reaching the 2048 tile, since it is difficult enough. The game ends when there is no

[1] the original game can be played at `https://gabrielecirulli.github.io/2048/`

[2] the highest recorded tile is 32768



| | | | |
|---|---|---|---|
| 16 | 8 | 4 | 4 |
| 128 | 64 | 2 | |
| 512 | 32 | 2 | |
| 256 | 16 | | |

Figure 1.1: An example of the game state from the original version by Gabrielle Cirulli.

valid action to perform.

It turns out that one of the general strategies most players quickly discover is to keep the board sorted, i.e. to keep the biggest tile in one of the corners. That way the spawning tiles can be easily merged with other low-valued tiles and gradually slid into the bigger valued ones. However, sometimes the only valid move is to slide the highest tile away from the corner and due to the tiles spawning randomly, the newly spawned tile can take the place in the corner where the highest tile was, thus breaking the strategy.

Another minor heuristic is used to speed up the early stage of the game, where it is suitable to simply pick the first few moves at random. As the number of tiles on the board remains reasonably low, the possibility of reaching a terminal state is negligible.

Since the release of the original game, there has been a tremendous amount of different versions that modified the gameplay in various ways, some of them being played on a hexagonal board, using arithmetic operations as tiles, or a version where a player drops tiles in order to prevent AI player from reaching the 2048 tile.

## 1.2 2048 in research

There were many attempts at solving the game by an artificial intelligence. Szubert and Jaśkowski (2014) used Temporal Difference (TD) learning together with N-tuple network to achieve a winning rate of 97%. Here, the N-tuple networks contained predetermined positions on the board with weights associated with each observable sequence. This approach was extended by Wu et al. (2014) by Multi-Stage Temporal Difference (MSTD), adapting the agent to several stages of the game. The MSTD reached the 32768-tile with a rate of 31.75%, a significant improvement compared to 0% by TD. Oka and Matsuzaki (2016) extended previous TD approaches and discovered that the positions of N-tuples significantly influence the performance of the algorithm. Dedieu and Amar (2017) implemented a deep Policy Network, but their agent was not able to achieve the 2048-tile a single time.

# Chapter 2

# Artificial intelligence

In this chapter we present the basic concepts of artificial intelligence (AI) that we will use later in the work. This chapter was written according to Russell and Norvig (2009), Sutton and Barto (2017) and Kvasnička et al. (1997).

## 2.1 Markov decision process

In order to introduce Reinforcement learning, we will first define the finite Markov Decision Process (MDP). MDP is a framework used for modeling decision making in stochastic (i.e. random) environments in many areas, such as robotics or economics. Formally, MDP is a 5-tuple $(S, A, P_.(\cdot,\cdot), P_.(\cdot,\cdot), \gamma)$, where

- $S$ is a finite set of states
- $A$ is a finite set of actions
- $P_a(s, s') = Pr(s_{t+1}|s_t = s, a_t = a)$ is the probability that action $a$ in state $s$ in time $t$ will result in state $s'$ in time $t + 1$
- $R_a(s, s')$ is the immediate reward received after a transition from state $s$ to state $s'$ by taking action $a$
- $\gamma \in [0, 1]$ is the discount factor, which represents the difference in importance between future and present rewards

MDPs model the process of learning by interacting and observing. We call the entity that performs the learning an agent and everything the agent interacts with is called an environment. The agent interacts with the environment on a basis of discrete time steps, each step consisting of taking an action $a_t \in A$, observing an immediate reward $r_t \in \mathbb{R}$ and changing the current state $s_t$ to a new state $s_{t+1}$. In MDPs, the outcome of taking an action $a \in A$ in a state $s \in S$ depends only on the definition

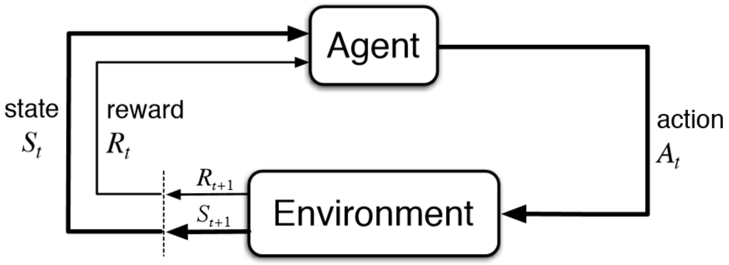


Figure 2.1: An agent interacting with the environment within a Markov Decision Process (Sutton and Barto, 2017).

of the transition probability function $P_a(s, \cdot)$ and not on states that the agent visited before. This crucial property is called Markov property (Sutton and Barto, 2017).

With the agent interacting with the environment comes a notion of discounted future rewards. That is, when we are evaluating the sum of discounted future rewards starting from time $t$ - $R_t$, the importance of each next reward decreases with the magnitude of $\gamma$ as following:

$$R = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{t+n} r_{t+n} + ...$$

The key problem for MDPs is to find an optimal policy $\pi$ ($\pi : S \rightarrow A$), that is a function which maps states to actions such that picking action $\pi(s)$ in the state $s$ at any time will result in maximizing some cumulative function of random rewards called utility. Utility function for a policy $\pi$, $U^\pi(s)$, is usually defined as the expected discounted sum of rewards we get when we start in state $s$ and follow policy $\pi$:

$$U^\pi(s = s_0) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})\right] \tag{2.1}$$

## 2.2 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning (ML) that has been studied by animal psychologists over the past 60 years (Russell and Norvig, 2009). In RL, an agent learns from its own experience, rather than by being presented with desired behavior as in supervised learning. The agent takes actions in an environment (commonly formulated as MDP) an observes rewards, otherwise called reinforcements (hence the name) as a feedback in order to find an optimal (or at least nearly optimal) policy. In RL, the agent does not need to have knowledge about the model of the environment or the reward function, therefore it is also feasible for continuous environments with only partial observability. In RL, the reward also does not need to come immediately after taking an action, but can be delayed.

The three basic types of RL agents as presented in Russell and Norvig (2009) are:

1. A **reflex agent** learns a policy that directly maps states to actions
2. A **utility-based agent** learns a utility function defined on states that measures how good a particular state is and uses it to select the best actions in order to maximize the expected utility
3. A **Q-learning** agent learns an **action-utility function**, commonly defined as the **Q-function** that represents the expected utility of taking a given action in a given state

In RL, the agent has to explore as many states as possible in order to learn how to behave optimally. However, the agent only visits states that its current policy allows him, so we need to encourage him to sometimes select actions that are not optimal according to agent's knowledge. One particular strategy that addresses this problem is called $\epsilon$-greedy exploration, in which the agent takes a random action with the probability of $\epsilon$ and the best action according to the current knowledge with the probability of (1-$\epsilon$). The value of $\epsilon$ can be altered during training.

## 2.3 Feedforward neural networks

We will use the concepts of Feedforward Neural Networks in the design of our agents, therefore we present a brief introduction into their architecture and mechanics. This section was written according to Haykin (2009).

### 2.3.1 Learning methods

There are two distinct methods of how artificial neural networks (ANNs) can be learned. **Supervised** learning is a type of learning in which an output function is inferred by processing labeled training data. The training data consist of a set of pairs, each pair consists of an input object and a desired output. In **unsupervised** learning there are no desired outputs and the network has to recognize some hidden structure in the presented data. In the next sections, we focus on the supervised type of learning.

### 2.3.2 Neurons

Basic building elements for ANNs are neurons, which receive one or more inputs and accordingly produce an output. A neuron computes the weighted sum of the inputs, commonly referred to as *net*, which is then passed through a non-linear function known

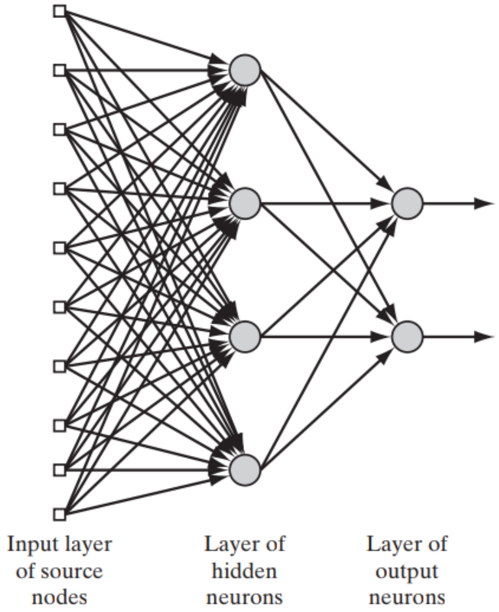


Figure 2.2: A neural network consisting of three layers (Haykin, 2009).

as activation function or transfer function. The network is created by connecting outputs of neurons to inputs of other neurons, forming a directed, weighted graph.

Neurons usually possess a so called bias unit which is added to the weighted sum of inputs to allow the network to shift the output values along the x-axis. We can treat the bias unit as another weight if we augment the input with another dimension of fixed value $x_{n+1} = 1$. The output from a neuron can be therefore written as

$$y = f(\sum_{i=1}^{n+1} w_i x_i) = f(net) \tag{2.2}$$

Where $f$ is the activation function. Activation functions are usually sigmoid functions (named by their shape, which in the form of a letter 'S'), but may also take form of other non-linear functions. A desirable property of activation functions is to be continuously differentiable so we can use them in gradient-based training methods. Below, we give example of two commonly used activation functions

$$S(x) = \frac{1}{1+e^{-x}} \tag{2.3}$$

$$R(x) = \max(0, x) \tag{2.4}$$

Function $R(x)$ (2.4) is called Rectified Linear Unit (ReLU) and it is not differentiable at 0, but it is still usable for gradient-based as it is rarely exactly at 0.

### 2.3.3 Forward propagation

Feedforward Neural Networks are arranged into layers of neurons, with the same activation function used within a layer. The output of each neuron in each layer is connected to inputs of some neurons in the next layer, therefore the graph of connections is acyclic (as opposed to Recurrent Neural Networks [1]). When each neuron in a layer is connected to each neuron in the next layer, we call it full connectivity. The layers between the input layer and the output layer are called hidden layers. In order to get an output from a neural network, the information at the input must flow progressively through all neurons in all layers directly or indirectly connected to the input neurons. This process is called a forward propagation or forward pass, meaning that the input flows forward across the layers of the network. To demonstrate a forward pass, assume a three layer network. Let us define the output for the $k$-th neuron in the hidden layer, assuming full connectivity:

$$h_k = f(net_{h_k}) = f_{\text{hid}}(\sum_{j=1}^{n+1} w_{jk} x_j) \tag{2.5}$$

where $n$ is the number of neurons in the first (input) layer, $x_j$ is the value of the $j$-th neuron in the input layer (note that $x_{n+1} = 1$ is the augmentation of the input for the bias unit) and $w_{jk}$ is the weight between the $j$-th neuron in the input layer and $k$-th neuron in the hidden layer. Then the output of the $i$-th neuron in the output layer of the network is defined as:

$$y_i = f(net_{y_i}) = f_{\text{out}}(\sum_{k=1}^{m+1} w_{ki} h_k) \tag{2.6}$$

where $m$ is the number of neurons in the hidden layer.

### 2.3.4 Training process

The major issue we need to address is how to update the weights of an ANN in order to obtain better results in the future. Let $P = \{(\vec{x}^1, \vec{d}^1), (\vec{x}^2, \vec{d}^2), ..., (\vec{x}^n, \vec{d}^n)\}$ be the training set, where $\vec{x}^i \in \mathbb{R}^m$ is the input and $\vec{d}^i \in \mathbb{R}^k$ is the desired output for $1 \geq i \geq n$. Firstly, we need choose an error function $E$, that would serve as a notion of how close the outputs from an ANN are to desired ones. There are many types of error functions, one of the most common is the mean squared error (MSE) defined as:

$$E = \frac{1}{2n} \sum_{i=1}^{n} (d_i - y_i)^2 \tag{2.7}$$

[1] more on recurrent neural networks can be found in (Haykin, 2009)

where $y_i$ is the actual output from the network. Note that the factor of 2 is put there for convenience to cancel out with the power of 2 when differentiating the function. Then, to be able to know the direction to which we need to change the weights, we need to know what is the gradient $\nabla_{w_{ij}} E$ of the error $E$ with respect to each of the weights $w_{ij}$:

$$\nabla_{w_{ij}} E = \frac{\partial E}{\partial w_{ij}} \tag{2.8}$$

By knowing this, we can change each of the weights by a small amount in the opposite direction of the gradient according to the update rule:

$$w_{ij}(t+1) = w_{ij}(t) - \eta \nabla_{w_{ij}(t)} E \tag{2.9}$$

where $w_{ij}(t)$ denotes the value of a weight $w_{ij}$ at a training iteration $t$ and $\eta \in [0,1]$ is the learning rate parameter, which prevents large updates that can cause instability in the network. The gradient of the error function is computed using the Backpropagation algorithm (described in Chapter 4 in Haykin (2009)).

### 2.3.5 Training optimizations

The training algorithm presented in the section before uses the whole training set $P$ in each training iteration. This is inefficient when the training set is large, so instead a randomly chosen subset (named a batch) is used for training each iteration. Due to its randomness this method got named Stochastic Gradient Descend (SGD).

Another improvement is based on an observation that the weight updates often oscilate for a long time before reaching a value that is satisfactory. The updates can be alleviated by adding a momentum, i.e. a difference between the current value and the value before the previous update, multiplied by a momentum rate parameter $\mu \in [0,1]$:

$$w_{ij}(t+1) = w_{ij}(t) - \eta \frac{\partial E}{\partial w_{ij}} + \mu(w_{ij}(t) - w_{ij}(t-1)) \tag{2.10}$$

The learning rate parameter $\eta$ can also be altered during training. It was shown by Hinton et al. (2012) that exponentially shrinking the learning rate leads to better results. Also, we can tune the learning rate so that each network weight has its own value of $\eta_{w_{ij}}$, which is, for example, the case of RMSprop algorithm. In the experimental part of this thesis we used Adam method (Kingma and Ba, 2014), which extends this idea further by adding a momentum to the update rule.

### 2.3.6 Network pruning techniques

One of the commonly encountered problems in ANN training is overfitting. Overfitting is when the network learns the details of the training set so well that it also learns the noise and does not generalize beyond the training patterns as we would require. Regularization is a set techniques for preventing this kind of behavior. We distinguish four different types of regularization.

The first type is called data augmentation. It is common in deep learning that the overfitting happens due to small training sets. Depending on the problem, the training set can be sometimes artificially augmented, i.e. we can add more new training samples by modifying the ones we already have. This is common in visual data, where various changes to pixel colors, rotations and scaling produce new data.

The second type of regularization is called early stopping. The idea is very simple, we stop the training right when the network is sufficiently trained and still can generalize well. To know when to stop the training, we divide the training set further into an estimation set and a validation set. The network is trained only on the estimation set and regularly tested on the validation set whether it still performs well on data it has not been trained on.

Another type of regularization is called a Dropout layer (Srivastava et al., 2014). Dropout layer can be inserted after an input layer of any hidden layer, and it consists of selecting a portion on the input neurons and ignoring their outputs during a forward and a backward pass in one training iteration. Besides speeding up the training process, the network is prevented from overfitting, because the neurons do not co-adapt too much.

The last type of regularization is weight penalty. Weight penalizing tries to prevent the weights from growing too large and consequently causing overfitting. There are many ways of how to realize weight penalizing, but one of the most popular is to add a term to the cost function that penalizes large weights as following:

$$\hat{E}(w) = E(w) + \frac{1}{2}\lambda \sum_{i} |w_i| \tag{2.11}$$

where $\lambda \in [0, 1]$ is a parameter that determines the strength of the penalization. This way of regularization is called $L_1$ regularization. There is also $L_2$ regularization, which is defined as:

$$\hat{E}(w) = E(w) + \frac{1}{2}\lambda \sum_{i} w_i^2 \tag{2.12}$$

# Chapter 3

# Q-learning

In this chapter we present and describe the idea of Q-learning, which is one of the most popular reinforcement learning algorithms, as well as some of its recent improvements.

## 3.1 Q-learning

Q-learning is a model-free (i.e. that the agent does not need to have a model about the environment) reinforcement learning algorithm. The aim of Q-learning is to learn an optimal policy by learning a state-action value function called Q-function. This function, taking a state $s$ and an action $a$ as parameters, is defined as the expected cumulative reward we get when we take an action $a$ in state $s$ and then follow the optimal policy. When we are dealing with a Q-function in terms of some policy $\pi$, we write it as $Q^{\pi}$. The optimal Q-function, $Q^{*}$, is the maximum expected cumulative reward achievable from state $s$ by taking an action $a$:

$$Q^{*}(s = s_0, a) = \max_{\pi} \mathbb{E}\left[ r_0 + \gamma r_1 + \gamma^2 r_2 + ... = \right] = \max_{\pi} \mathbb{E}\left[ \sum_{t=0}^{\infty} \gamma^t r_t \right] \tag{3.1}$$

where $\gamma$ is the discount factor. $Q^{*}$ satisfies the following Bellman equation:

$$Q^{*}(s, a) = \begin{cases} r, & \text{if s' is a terminal state} \\ r + \gamma \max_{a'} Q(s', a'), & \text{otherwise} \end{cases} \tag{3.2}$$

where $r$ is again the immediate reward obtained by the transition from state $s$ to state $s'$ by taking action $a$.

The optimal policy in Q-learning is learned by starting from a random policy and performing update rules to obtain better estimates of $Q^{*}$ and eventually converging to it. The update rule is defined as:

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha \left[ r + \gamma \max_{a'} Q_t(s',a') - Q_t(s,a) \right] \tag{3.3}$$

where $\alpha \in [0,1]$ is a learning rate. It has been proven in Watkins and Dayan (1992) for $0 \leq \alpha < 1$ and any MDP, that $Q_n \rightarrow Q^*$ as $n \rightarrow \infty$, but in some cases it is extremely slow.

### 3.1.1 Deep Q-Network

As Q-learning can be generally applied to any MDP, it is not always seen as a feasible solution from a practical point of view. The most common encountered problem is that the number of possible states is too high to be stored in any affordable memory, often even exceeding the physical possibility of storage.

The solution for reducing the size of the Q-table is to learn an approximation of the Q-function using a neural network model. This allows us to trade the memory complexity of a Q-table for that of a neural network, which can be significantly smaller. In exchange we get worse estimation properties as the convergence to optimal policy is now not guaranteed and at times also hard to reach.

This way, the Q-function is also a function of the weights of the neural network $\theta_t$, therefore being $Q(s,a;\theta_t)$. The loss function is then defined as:

$$L(\theta) = \Big( r + \gamma \max_{a'} Q(s',a';\theta) - Q(s,a;\theta) \Big)^2 \tag{3.4}$$

This led to the design of so called Deep Q Networks (DQN), which are essentially multi-layered neural networks that accept a state $s$ as an input and output a vector of action values $Q(s,.;\theta)$.

### 3.1.2 Double Deep Q Network

In the standard Q-learning, the Q-values sometimes tend to be overestimated (Hasselt, 2010). This itself would not be a problem if all the Q-values would be overestimated by the same value, but that is not the case. Since it is the difference between the action values that is important, selecting overestimated values results in overoptimistic estimations of next values and therefore poor policies.

The proposed solution for this problem by Hasselt (2010) is to decouple action evaluation and selection of the best action in the max operator. The original solution dealt with the basic tabular Q-learning, but it was extended to Deep Q Networks in Wang et al. (2016). Decoupling in DQN can be done by maintaining two independent networks for both tasks. The prediction network estimates the action values for the current state and the target network selects the best action. The loss function then becomes:

$$L(\theta) = \Big(r + \gamma Q(s', \operatorname*{argmax}_{a'} Q(s', a'; \theta^-); \theta) - Q(s, a; \theta)\Big)^2 \tag{3.5}$$

This network architecture is called Double DQN (DDQN). To ensure the decoupling, we use the same network for selection as for evaluation, but with different set of weights. Therefore we maintain weights $\theta$ for the evaluation network and $\theta^-$ for the target network. Each $t$ iterations of the learning loop we copy the weights from the evaluation network to the target network and keep it fixed since. The evaluation network is trained according to the standard DQN update.

### 3.1.3 Dueling DQN architecture

In many environments it is common that there is a significant number of states in which we do not need to know the state-action values for all actions as some state-action values might be irrelevant for a given state. To address this problem, a network architecture called Dueling Deep Q Network was proposed in Wang et al. (2016).

The key concept of the solution was to separate the network architecture into two streams of fully connected layers, one for estimating the state-value $V(s)$ and the other to estimate the action-values, defined as an advantage function $A(s, a)$. These two streams are then combined together at the final layer by a certain kind of aggregation to form an estimation of $Q(s, a)$. The naive approach of aggregating the two streams together would seem to be simply a summation of their output values as follows:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha) \tag{3.6}$$

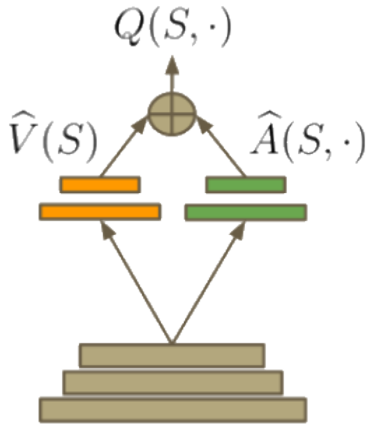


Figure 3.1: Dueling DQN architecture with the first 3 layers in fully-connected fashion and then separated into $V(s)$ and $A(S, \cdot)$ streams which are finally combined into a $Q(S, \cdot)$ value.

where $\alpha$ are the parameters of the advantage stream and $\beta$ of the value stream. However, it is necessary to observe that $A(s, a; \theta, \alpha)$ is actually $|\mathcal{A}|$ values in total, so to compensate for that we would have to add $V(s; \theta, \alpha)$ into the summation for every action. Another thing to note is that this way is not possible to obtain the advantage and the value function values from the output, because there are infinitely many ways of summing values that result in the same Q-value. Therefore it is not a good way to estimate the state-action values like this. A better approach presented in the original paper was to remove the advantage of the chosen action from $A(s, .; \theta, \alpha)$ values in the sum as follows:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \Big(A(s, a; \theta, \alpha) - \max_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha)\Big) \tag{3.7}$$

When we aggregate the streams like this, we can get the state value and the advantage values from $Q(s, a; \theta, \alpha, \beta)$, because the following equation holds

$$\arg\max_{a' \in \mathcal{A}} Q(s, a'; \theta, \alpha, \beta) = \arg\max_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha) = V(s; \theta; \beta) \tag{3.8}$$

Hence, the Q-value is a combination of two unique stream values. Also, an alternative approach is to subtract the mean from the advantage function values:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \Big(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha)\Big) \tag{3.9}$$

It is important to note that as this technique only alters the architecture of neural network, it can be used together with the Double DQN.

### 3.1.4 Experience replay

A basic DQN agent learns from every experience $e_t = (s, a, r, s')$ that is encountered and does not process it further on. This is often infeasible as in many environments experiences that come sequentially one after another are strongly correlated and the agent is not presented with sufficient variance of transitions. Mnih et al. (2013) overcame this obstacle by storing experiences in a buffer pool $\mathcal{D} = \{e_1, e_2, ..., e_n\}$ from which it draws samples to learn from with uniform probability.

### 3.1.5 Prioritized experience replay

While using experience replay is a good way of providing an agent with a variance of independent experiences to learn from, it is not always efficient to draw the learning samples randomly, as some experiences are more beneficial for the network than others. A better approach appears to be to use some kind of prioritization scheme according to some criteria. One such criterion is the TD-error $\delta$ that indicates how unexpected a certain experience for the network is and so we can conclude that the bigger TD-error, the more the network learns from such experience.

However, greedily selecting the experiences according to the TD-error can lead to over-fitting since the most erroneous experiences would be replayed at a very high frequency and the least erroneous effectively never. The fact that we update the errors for experiences that were replayed does not help much, because the error usually shrinks at only small pace. Finally, when a replay memory is implemented as a sliding windows, hence deleting the oldest experiences to make place for new ones, this would mean losing many potentially valuable experiences.

A solution proposed in Schaul et al. (2016) finds a balance between randomly selecting samples and using a purely greedy strategy. The solution lies in using a stochastic method to sample prioritized experiences. Let us define a probability of an experience being sampled as

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.10}$$

where $p_i$ is the priority of $i$-th experience. The parameter $\alpha$ determines the magnitude with which the priorities are taken into consideration (note that $\alpha = 0$ refers to the greedy case). The two variants of assigning priorities to experiences that were proposed in the original paper are:

- Direct, proportional priority $p_i = |\delta| + \epsilon$ where $\epsilon$ is some small constant that prevents transitions from not being sampled when their TD-error becomes zero

- Indirect, rank-based priority $p_i = \frac{1}{\mathrm{rank}(i)}$ where rank$(i)$ is the position of the $i$-th transition in the pool buffer when sorted according to $|\delta|$.

This prioritization scheme changes the distribution of the transitions and affecting learning in a negative way. This is corrected by importance sampling weights:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \tag{3.11}$$

where $N$ is the number of experiences in the pool buffer and $\beta \in [0, 1]$ is the amount of compensation for the non-uniform probabilities. We can use these weights in a Q-learning update rule as a multiplier of the $\delta$ error.

# Chapter 4

# Implementation

In this chapter we will explain the motivations behind choosing the tools for our experimental purposes, as well as for implementing the game environment.

## 4.1 Game environment

The original 2048 game by Gabriele Cirulli is implemented in Javascript. While it is possible to write AI agents in Javascript or in another language and interact with the web application externally, we chose to use Python implementation of the game as the intention was to write agents in this language. C++ was also considered due to its efficiency, but we later put aside this idea, mostly because debugging and changing C++ code for ML algorithms can be time demanding, but also because there is not as wide variety of ML libraries for C++ as is for Python. Therefore, the focus was to implement the game environment with sufficiently simple API (Application Programming Interface) that would be intuitive to use when developing AI agents. We dropped the need for a GUI, because a simple text-based board representation suffices our needs for debugging purposes.

The board is represented as a 2-D numpy array of integer values representing the exponents of the tile values. One exception is that the value 0 represents an empty place on the board. The game function `step(action)` in game environment performs one move on the board and returns a news state, a reward obtained by the state transition and a boolean value indicating whether the games has ended. The function `reset()` resets the game environment to a beginning state and `exponentiate()` returns an actual game board with tiles as powers of two as in the original game.

## 4.2 AI agents

For our agents implementation, we used machine learning libraries `TensorFlow` and `Keras` for Python. The first, simpler experiments were written using `TensorFlow`, but as the models became complex and we needed to change the code often, we switched to `Keras` with `TensorFlow` as its backend for convenience.

Although we have written most of the code by ourselves, we used some parts from external sources. Namely, we used and existing implementation of prioritized experience replay memory from `OpenAI baselines` [1], which is a set of implementations of reinforcement learning algorithms made for the research community with an MIT license. For the basic, not prioritized experience replay memory, we used `collections.deque` from Python's standard library.

We implemented the agents as base classes, whose parameters can be easily altered. Each agent class has a `_build_model(self)` function that creates and returns the neural network for approximating the Q-function, along with defining the optimizer and the loss function for training. The function `choose_action(state)` samples the best action from the output of the network based on the `state` as an input, with the probability of `self.epsilon` of taking a random action. The function `store_experience(state, action, reward, new_state, done)` simply stores the whole state transition into a memory pool, either prioritized or not. In agents that make use of the double DQN architecture, the function `update_target()` copies all the weights from the evaluation model to the target model. All agents possess functions `save_model()` and `restore_model()` for simple serialization, and are trained using the `train()` function, of which parameters depend on the type of the agent. For serializing data outside of the model, as for example the scores throughout the time, we used a `Pickle` library. Additionally, all the graphs we plotted were made with `Matplotlib` plotting library.

[1] available at `https://github.com/openai/baselines`

# Chapter 5

# Experiments

In this following chapter we present the results of simulations, differing in reward designs and state encoding.

## 5.1 Experimental settings

As we discover in the following experiments, the original game of 2048 cannot be grasped by reinforcement learning as easily as, for example, Atari games, due to its stochastic nature. For that purpose, we introduce modifications of the game that would improve the potential for the learning of the agents.

### 5.1.1 Performance measurements

While an intuitive measurement technique seems to be the total cumulative score at the first glance, we will rather keep track of the sum of the board tiles at the end of each game. Such measurement has lower variance and better assesses the performance of the agent considering the goal of obtaining the 2048 tile. Therefore, when we refer to a 'score' in the following text, we mean the latter definition. Simultaneously, we will also keep track of the maximum tile obtained throughout each game.

### 5.1.2 Invalid moves

In the original version of the game, moves that do not change the position of any tile on the board are simply not permitted. This creates an issue we need to address. One possible option is to introduce a negative reward for each invalid move. Alternatively, we can treat invalid moves like valid moves, with a new tile spawning randomly afterwards. After the consideration, we decided to pursue the second path, since making the environment for the agents more 'friendly' would probably lead to smoother learning curve.

### 5.1.3 Reward functions

The standard reward in the original version of the game is the sum of the values of newly created tiles by merging. It is important to note that DQN algorithms work considerably better when the rewards are in the range $[-1, 1]$. This is because scores between games do not vary that much, which prevents the agent from making large updates on the network. So to make the learning more stable, we will scale the rewards to the range $[-1, 1]$ making the Q-values relatively small. For the original reward, this can done by dividing it by 80, since 8 is the upper bound for the maximum number of pairs of tiles that can be merged and the maximum value for a tile on board is 10 ($2^{10} = 1024$ and when 2048 appears the game ends). However, this upper bound is not very tight since we will never merge 8 pairs of tiles, so we can lower the factor arbitrarily and in case it exceeds the value of 1, we simply take 1 as the reward.

Since our original intention was to learn the state-action value function without any interference in the reward function other than scaling, as we will later see, some changes needed to be made in order to achieve better learning progress. For that matter, introducing a reward of $-1$ for reaching a terminal state seems like a good idea, because it will encourage the agent to play for a longer time. Also, the original reward may appear noisy to the agent. Keeping this in mind, we devised three alternative reward functions $R_1$, $R_2$ and $R_3$ in order to achieve better stability in learning:

$$R_1(s, s') = \begin{cases} 1, & \text{if obtained 2048 tile} \\ (\#\mathit{tilesMerged}(s, s') - 1)/8, & \text{otherwise} \end{cases}$$

$$R_2(s, s') = \begin{cases} -1, & \text{if } s' \text{ is a terminal state} \\ \min(\#\mathit{tilesMerged}(s, s')/4, 1), & \text{otherwise} \end{cases}$$

$$R_3(s, s') = \begin{cases} -1, & \text{if } s' \text{ is a terminal state} \\ 1, & \text{if } R(s, s') \geq \alpha \\ R(s, s')/\beta, & \text{otherwise} \end{cases}$$

where $R(s, s')$ represents the original reward in terms of exponents instead of powers of 2 and in case of $R_3$ the parameters $\alpha$ and $\beta$ are chosen in advance. Functions $R_1$ and $R_2$ should be able to eliminate most of the noise for the cost of reducing the complexity of the game, while $R_3$ is closer to the original reward function, but has potential to be learned more smoothly than the original.

### 5.1.4 State encoding

As neural networks work best with the inputs scaled so that each neuron receives an input from range $[0, 1]$ (or $[-1, 1]$), we employed 2 different ways of encoding a game state into a form suitable for a neural network input. [1]

1. The most intuitive strategy is to encode a state as a vector of length 16 with exponents of the tiles as its elements - values between 0 and 11 (with 0 representing an empty place on board). However, to provide a better input to a neural network architecture, we normalize this vector by dividing it by 11 to obtain values in the range $[0, 1]$.

2. This encoding strategy uses the idea of Gray code. Gray code is such an ordering of binary numbers that each two successive numbers differ only in one bit. This encoding could help the NN to identify the symmetries in the game board more easily. Therefore, each exponent of a tile value is encoded into a 4-bit gray code, which are then concatenated into a vector of length $16 \cdot 4 = 64$.

### 5.1.5 Base model description

We describe the basic structure of the model for our agents. We used a combination of double DQN with dueling architecture, using the mean type of merging the value stream with the advantage stream. We also added proportional prioritized experience replay. For the approximation of the Q-function, we used a considerably deep network. The sizes of the hidden layers were set to `[300, 300, 200, 200, 100]`. The activation functions of all hidden layers before separating into streams is set to ReLU. The value stream was then constructed by adding a fully connected layer of 60 neurons and 1 output neuron for the state value. Similarly, the advantage stream consists of a fully connected layer of size 100 and then 4 output neurons for action values. The first fully connected layer in each stream has a ReLU activation function and output layers are linear, to allow the Q-function to also have negative values.

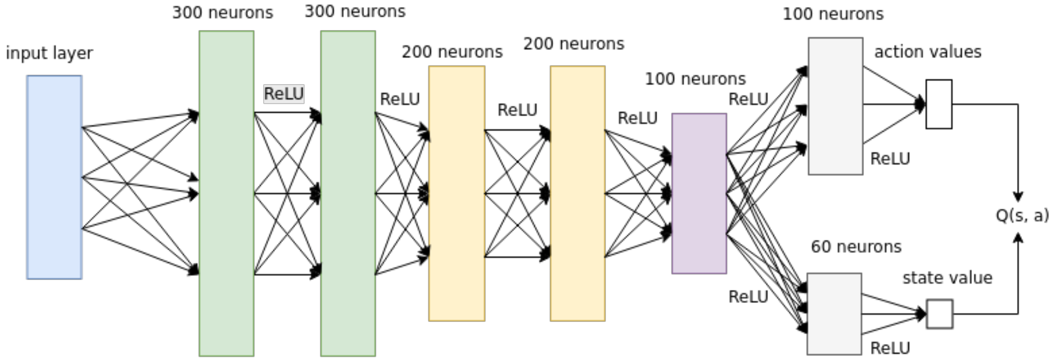


Figure 5.1: An architecture of the neural network used to approximate the Q-function.

[1] We also experimented with one-hot encoding which turned out to be less successful than the normalized board

Other choices for our model include:

- ADAM algorithm to optimize the stochastic gradient descend (Kingma and Ba, 2014). Although good results were achieved using RMSPROP when learning to play Atari games (Mnih et al., 2013), ADAM was developed a year after their effort and should be able to eliminate some of RMSPROP'S drawbacks.

- Mean squared error (MSE) as the loss function.

- $L_1$ regularization with a very small regularization rate of $10^{-6}$, as our network has a considerably larger number of weights.

We used the learning rate $\alpha = 0.0001$ and the discount factor $\gamma = 0.995$. We updated the target network every 50 steps of the game and performed a training update each 32 steps with a batch size 32. After each step of the game, we store the associated experience into a memory pool. The memory pool's maximum size was set to 10000 (after reaching this limit, each new experience replaces the oldest one). For exploration, we used the $\epsilon$-greedy strategy, with $\epsilon$ initially set to 1 and decaying after each training update with magnitude of 0.9999 to the minimal value of 0.1 that was kept until the end of the training.

### 5.1.6 Training and testing

Our model has a large number of parameters to alter and so it is computationally very difficult to find the right combination. Performing an automated grid search or a random search of parameters would be too costly. For this reason, in our experiments we had to mostly rely on the recommended parameters from similar algorithms as well as our intuition and try to set the parameters right on the first time. Nevertheless, we did many experiments of which we present those which were trained for a sufficiently long time and we consider them referential. After we trained each agent for a sufficiently long time, we let it play 10000 games and arranged the counts of maximum obtained tiles into a table.

## 5.2 Experiment 0: Random Agent

To have some reference baseline for our agents in the following experiments, we will first measure the performance of a random agent. Since the results of the randomly played games are independent, we plot the resulting scores into a histogram.

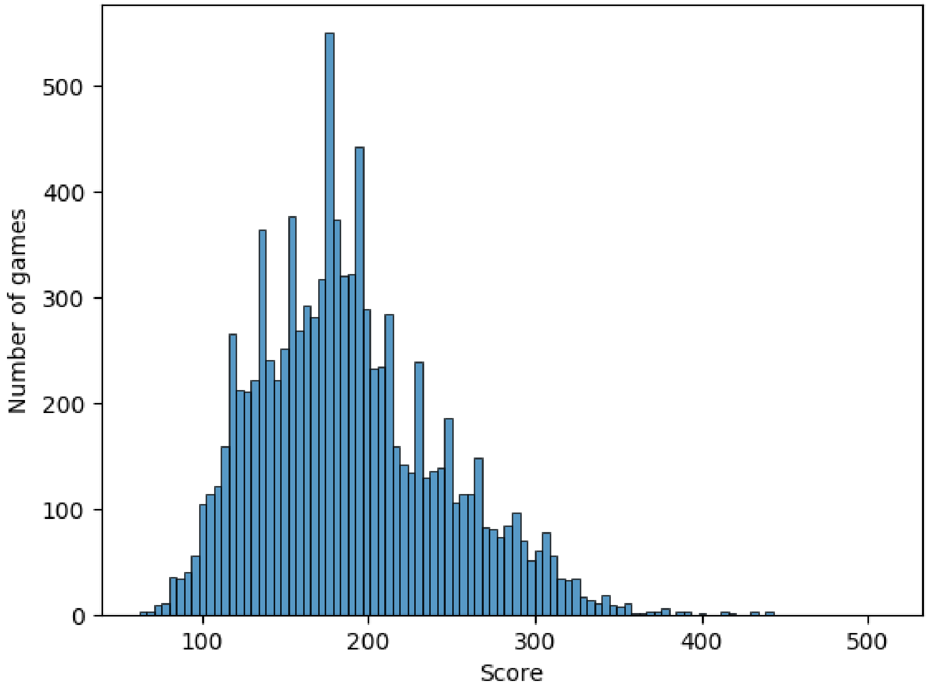


Figure 5.2: A histogram of scores over the 10000 random games sample. By the score we mean the the sum of the tiles at the end of each game.

| **Max. tile** | **2** | **4** | **8** | **16** | **32** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Game count** | 0 | 0 | 3 | 273 | 2373 | 5192 | 2113 | 46 | 0 | 0 | 0 |

Table 5.1: Distribution of 10000 random games according to the maximum tile achieved.

From figure 5.2 it is evident that the most of the random games ended up with a score slightly bellow 200 and only a small portion achieved 300 or more. Moreover, as we can see in table 5.1 a tile of 512 or higher was not reached in a single game and 256 was reached in only 46 games.

## 5.3 Experiment 1

In this experiment we chose the Gray encoding method with the original reward function, rescaled as: $\min(\frac{R(s,s')-5}{10}, 1)$. We also added a reward of 1 for obtaining the 2048 tile and no penalty for losing the game.

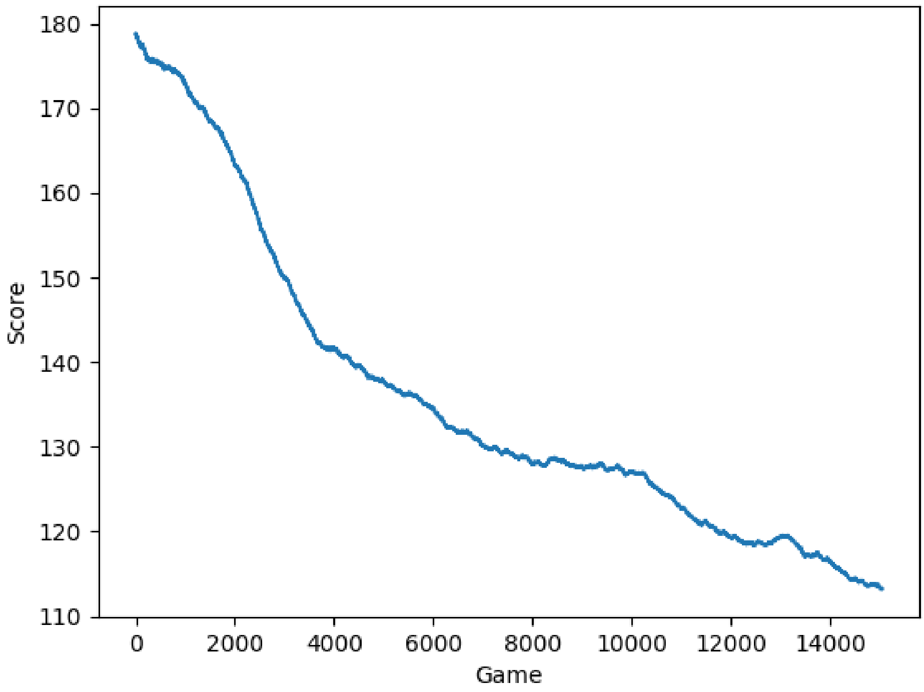


Figure 5.3: The training progress of experiment 1. The scores are smoothed out by the average of 2000 consecutive results.

| **Max. tile** | **2** | **4** | **8** | **16** | **32** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Game count** | 0 | 0 | 82 | 4125 | 4859 | 917 | 17 | 0 | 0 | 0 | 0 |

Table 5.2: Distribution of 10000 testing games played by the first agent after the training, according to the maximum tile achieved.

The slope in figure 5.3 is caused by the decaying of $\epsilon$, as at the early stage random moves are better than those of an untrained model. However, this agent appears to be unable to learn how to play better and as we can see from table 5.2, it performs even worse than the random agent.

## 5.4 Experiment 2

In this experiment, we used the $R_2$ reward function and tried the Gray encoding once more. The result we obtained is much better, even though the learned policy is still far from winning the game.

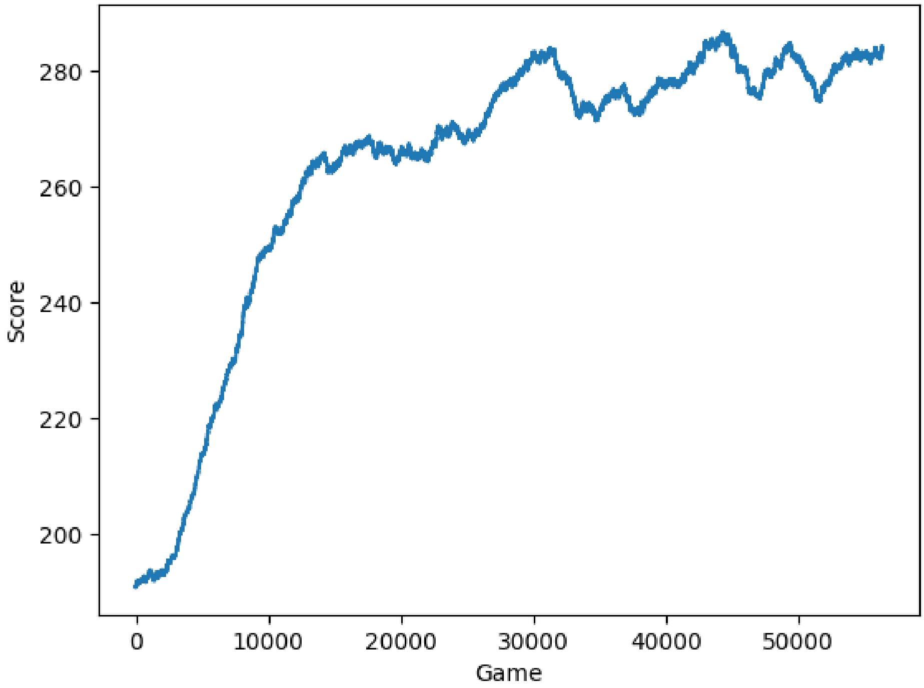


Figure 5.4: The training progress of experiment 2. The scores are smoothed out by the average of 2000 consecutive results.

| **Max. tile** | **2** | **4** | **8** | **16** | **32** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Game count** | 0 | 0 | 1 | 35 | 702 | 3732 | 4463 | 1056 | 11 | 0 | 0 |

Table 5.3: Distribution of 10000 testing games played by the second agent after the training, according to the maximum tile achieved.

Although we only achieved the 512 tile in only 11 games out of 10000, it is an improvement over the first agent, which failed to achieve the 256 tile a single time.

## 5.5 Experiment 3

In this experiment, we chose the $R_3$ reward function for $\alpha = 8$ and $\beta = 60$ with the first type of encoding. The progress we achieved is much more consistent.

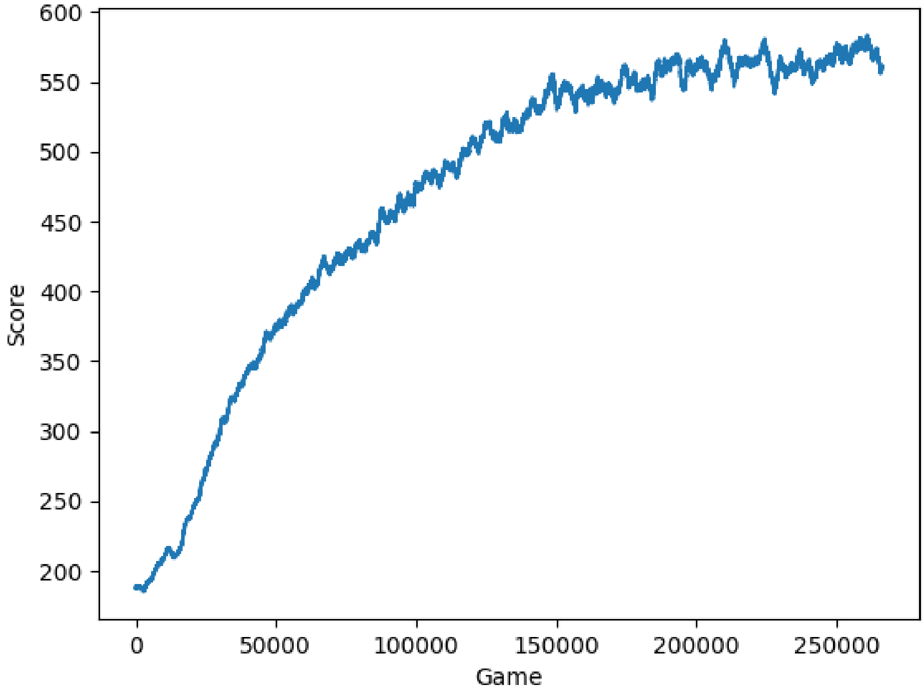


Figure 5.5: The training progress of experiment 3. The scores are smoothed out by the average of 2000 consecutive results.

| **Max. tile** | **2** | **4** | **8** | **16** | **32** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Game count** | 0 | 0 | 0 | 0 | 10 | 102 | 661 | 3058 | 5158 | 1011 | 0 |

Table 5.4: Distribution of 10000 testing games played by the third agent after the training, according to the maximum tile achieved.

Even though experiment 3 never reached the 2048 tile, it is much closer to it than the previous model, and also significantly more consistent at reaching the 512 tile.

## 5.6 Experiment 4

In the final experiment, we used the normalized encoding as in the previous one, but this time we used the $R_1$ reward function.

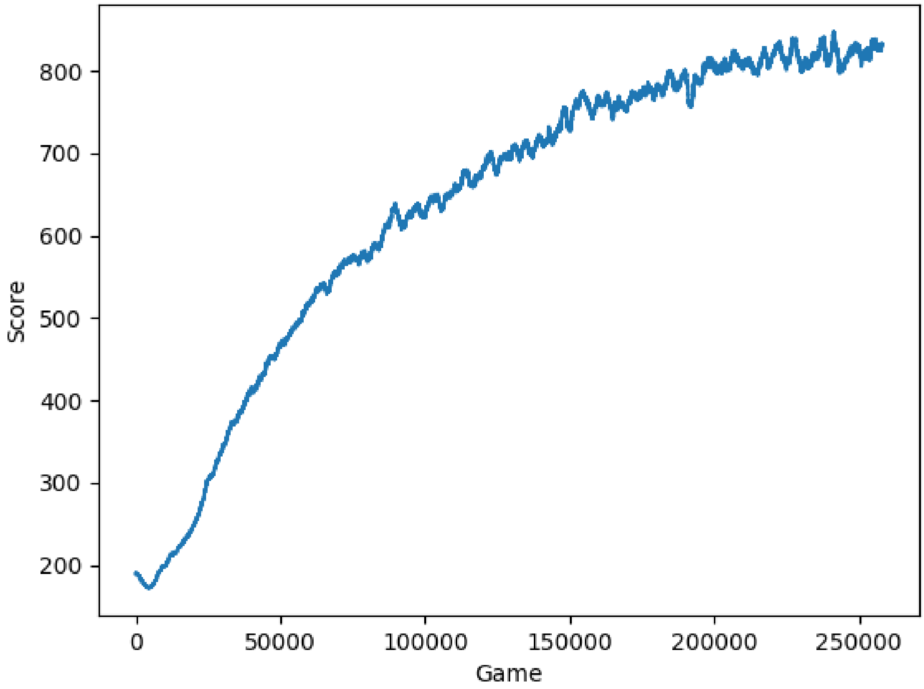


Figure 5.6: The training progress of experiment 4. The scores are smoothed out by the average of 2000 consecutive results.

| **Max. tile** | **2** | **4** | **8** | **16** | **32** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Game count** | 0 | 0 | 0 | 0 | 0 | 4 | 17 | 361 | 3430 | 5408 | 780 |

Table 5.5: Distribution of 10000 testing games played by the fourth agent after the training, according to the maximum tile achieved.

In Experiment 4 the agent was able to win the game in 7.8% training games, which is our best result.

# Chapter 6

# Discussion

## 6.1 Analysis of results

We were able to progressively achieve better results by modifying the reward function we used together with the encoding of the input. As we can see in figure 6.1, the third agent plays significantly better than random. Even as we did not find a model that could learn a policy which would achieve the 2048 tile regularly, it is possible that it can be done by the Deep Q Learning with another combination of parameters. In our experiments, we were limited to explore only a small part of the parameter space.

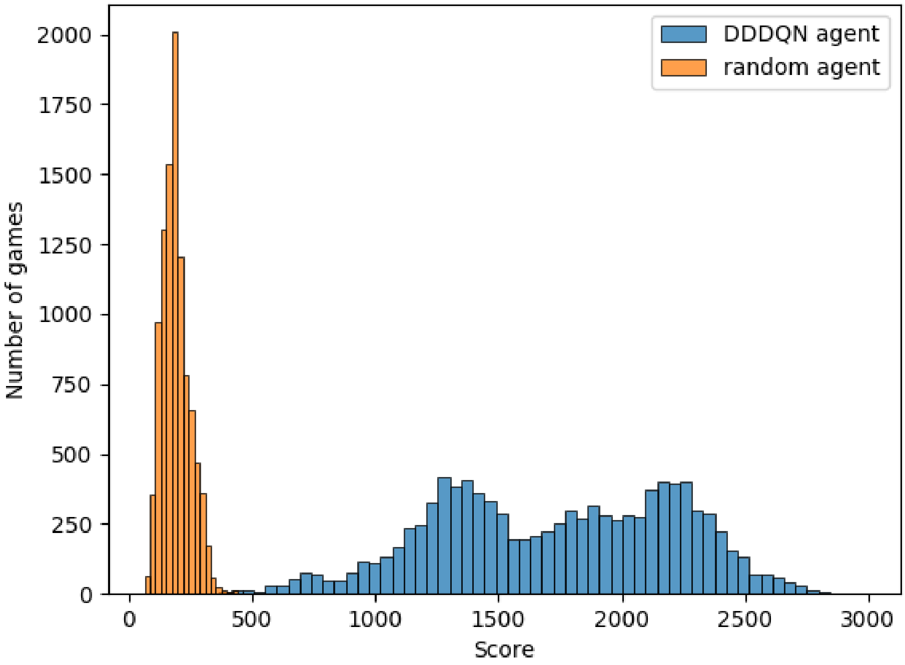


Figure 6.1: The comparison of the distributions of 10000 games played by a random agent and the best agent we trained (experiment 4).

In figure 6.1 we can see that the scores of the best agent have bimodal distribution, with one peak around a score of 1350 and the other over 2250. The explanation for this is that an agent often reaches two tiles of 512, but not always it is able to merge them into 1024 tile. When it does not merge them, it usually leads to the end of the game soon because of the space they occupy, otherwise it frees space for the game to continue.

A problem we encountered during the training is that sometimes the agent gets stuck at a point of reaching very similar score every time and therefore not learning. It happens because the agent's current policy is not good enough to reach higher scores and to get out of this position, it has to wait for the epsilon greedy strategy to trigger suitable random moves. However, the value of $\epsilon$ is almost always at its minimum when the problem occurs. This could be solved by tracking the progress of learning, increasing $\epsilon$ again when necessary and letting it decay afterwards as usual.

## 6.2 Future work

The next step in improving the performance of the algorithms would be adding more hidden layers to the advantage stream and the value stream of the network, to achieve more robust estimations of the state action value. Further tweaking the reward function may help too, as well as the parameters of our base model. Importantly, we can also assume an alternative version the game where a player has a possibility to step undo the last action, which significantly simplifies the game.

Although we achieved results that are close to our goal, our model is not the current state-of-the-art in reinforcement learning algorithms. We believe that the quantile regression DQN (Dabney et al., 2017) may achieve learning better policies and even potentially deal with the noise in the original rewards. Another improvement of the efficiency of learning is the use of delayed rewards in n-step Q-learning. This method uses the discounted reward over $n$ steps to update the $Q(s, a)$ value. N-step Q learning can be extended to Asynchronous n-step Q-learning (Mnih et al., 2016) using multiple actor-learner threads. Also, further exploring the possibilities of the Policy Network model (Dedieu and Amar, 2017) can bring success.

We trained our agents using only CPU power, which is slow with models that have a large number of parameters. A significant speed-up can be achieved by training the agents on a graphics card, for example with CUDA technology by NVIDIA. Asynchronous methods of learning (Mnih et al., 2016) also improve the speed by the factor of simultaneously running threads, where each thread employs an independent agent.

# Conclusion

We designed, implemented and tested a number of agents based on the Deep Q Network RL algorithm, enhanced by the combination of dueling and double DQN architecture and prioritized experience replay. Even though we did not believe that our agents would achieve the same level of performance as the tree search techniques, it was unexpected that the agents are unable to grasp the game's mechanics from the original reward function. It turns out that the game of 2048 is a difficult challenge for reinforcement learning algorithms, due to the high variety of rewards which may appear noisy and the factor of randomness of the transition function.

Even though our best performing agent can win the game by achieving the 2048 tile at times, human players are more quicker at discovering the right moves that often lead to winning. This tells us that the game requires a deeper level of abstraction and planning forward than the most of Atari games, in which the associated rewards usually come immediately, or in a short time.

Nevertheless, we showed that encoding the state as Gray code is not helpful as the network fails to recognize patterns in state representation where each merge of two tiles results in a tile that differs in only one bit from those it was created from.

We believe that the 2048 game is an interesting testing environment for reinforcement learning algorithms. To achieve better results, enhancing the pure RL approach with a supervised training as in AlphaGo (Silver et al., 2016) seems promising.

# Bibliography

Will Dabney, Mark Rowland, Marc G Bellemare, and Rémi Munos. Distributional Reinforcement Learning with Quantile Regression. *arXiv preprint arXiv:1710.10044*, 2017.

Antoine Dedieu and Jonathan Amar. Deep Reinforcement Learning for 2048. *31st Conference on Neural Information Processing Systems (NIPS 2017)*, 2017.

Hado V. Hasselt. Double Q-learning. In *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

Simon S. Haykin. *Neural networks and learning machines*, volume 3. Pearson Upper Saddle River, NJ, USA, 2009.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. RMSProp: Divide the gradient by a running average of its recent magnitude. *Neural networks for machine learning, Coursera lecture 6e*, 2012.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, 2014.

Vladimír Kvasnička, Ľubica. Beňušková, Jiří Pospíchal, Igor Farkaš, Peter Tiňo, and Andrej Král'. *Úvod do teórie neurónovỳch sietí.* Iris, 1997.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement learning. *CoRR*, abs/1312.5602, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pages 1928–1937. PMLR, 2016.

Kazuto Oka and Kiminori Matsuzaki. Systematic Selection of N-tuple networks for game 2048. In Aske Plaat, Walter Kosters, and Jaap van den Herik, editors, *Computers and Games*, pages 81–92, Cham, 2016. Springer International Publishing.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations*, 2016.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. In *Journal of Machine Learning Research*, volume 15, pages 1929–1958, 2014.

Richrd S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction (in preparation), 2017.

Marcin G. Szubert and Wojciech Jaśkowski. Temporal Difference Learning of N-tuple networks for the game 2048. In *Proceedings of the 2014 Conference on Computational Intelligence and Games*, CIG 2014, pages 373–380. IEEE, 2014.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-Learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *ICML'16*, pages 1995–2003, 2016.

Christopher J.C.H. Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992.

I-Chen Wu, Kun-Hao Yeh, Chao-Chin Liang, Chia-Chuan Chang, and Han Chiang. Multi-stage temporal difference learning for 2048. In *Technologies and Applications of Artificial Intelligence*, pages 366–378. Springer, 2014.