Aprendizaje por Refuerzo: Fundamentos Teóricos y Aplicación al cubo de Rubik

Reinforcement Learning: Theoretical Foundations and Application to the Rubik's Cube

UNIVERSIDAD COMPLUTENSE
MADRID

Carlos Tardón Rubio ctardon@ucm.es
Director: Miguel Palomino Tarjuelo
Curso 2021/2022

Trabajo de Fin de Grado del Doble Grado en
Ingeniería Informática - Matemáticas

Facultad de Informática, Universidad Complutense de Madrid

Resumen

Las técnicas utilizadas y desarrolladas en el área del aprendizaje por refuerzo han ido evolucionando desde sus inicios, a finales del siglo XX. Gracias a los distintos avances en este sector, se han podido resolver problemas cada vez más complicados. La influencia de otras áreas del aprendizaje automático y de la inteligencia artificial han permitido aplicaciones del aprendizaje por refuerzo que inicialmente suponían grandes desafíos por sus requerimientos computacionales. Uno de esos problemas es el que trataremos en este trabajo, que se caracteriza por un gran espacio de estados y un único estado final.

En un primer lugar, se dará una introducción teórica al área del aprendizaje por refuerzo, centrándonos en aquellos aspectos más relevantes en la resolución de nuestro problema. Después, se expondrá una descripción teórica del algoritmo DeepCubeA que fue diseñado especialmente para resolver el cubo de Rubik 3x3x3, caracterizado precisamente por un gran espacio de estados y un único estado final. Por último, diseñaremos e implementaremos una versión del algoritmo DeepCubeA, añadiendo algunos aspectos relevantes de su version anterior (DeepCube), y estudiaremos su comportamiento con los cubos de Rubik 2x2x2 y 3x3x3, y las Torres de Hanói.

Palabras clave

- Aprendizaje por refuerzo
- Cubo de Rubik
- Torres de Hanói
- Redes neuronales
- Aprendizaje automático
- Algoritmo de búsqueda A*
- BWAS

Abstract

The techniques employed and developed in the area of reinforcement learning have been evolving since their origins at the end of the 20th century. Thanks to the various advances in this field, it has been possible to solve increasingly complicated problems. The influence of other areas of machine learning and artificial intelligence has enabled applications of reinforcement learning that initially posed great challenges due to their computational requirements. One such problem is the one we will discuss in this work, which is characterized by a large state space and a single final state.

First, a theoretical introduction to the area of reinforcement learning will be given, focusing on those aspects most relevant to the solution of our problem. Then, a theoretical description of the DeepCubeA algorithm will be presented, that was designed to solve the Rubik's 3x3x3 Cube, which has a large state space and only one final state. Finally, we will design and implement a version of the DeepCubeA algorithm, adding some relevant aspects of its previous version (DeepCube), and we will study its behavior with the Rubik's 2x2x2 and 3x3x3 Cubes, and the Hanoi Towers.

Keywords

- Reinforcement learning
- Rubik's Cube
- Tower of Hanoi
- Neural networks
- Machine learning
- A^* search algorithm
- BWAS

Contents

1 Introducción 4
- 1.1 Antecedentes y Motivación . . . 4
- 1.2 Objetivos y Plan de Trabajo . . . 4

2 Reinforcement Learning: Theoretical Foundations 6
- 2.1 Machine Learning . . . 6
- 2.2 Reinforcement Learning . . . 8
- 2.3 The learning framework and MDP . . . 8
  - 2.3.1 Returns . . . 9
  - 2.3.2 Policies . . . 9
  - 2.3.3 Value Functions . . . 9
  - 2.3.4 Optimality . . . 10
  - 2.3.5 Value Iteration . . . 13

3 DeepCubeA: Theoretical Foundations 14
- 3.1 Description of the problem we want to solve . . . 14
  - 3.1.1 Rubik's Cube . . . 14
  - 3.1.2 Towers of Hanoi . . . 15
- 3.2 Neural Networks . . . 16
- 3.3 DAVI . . . 18
- 3.4 BWAS . . . 21

4 Implementation Decisions 23
- 4.1 Rubik's 3x3x3 Cube Representation . . . 23
- 4.2 Rubik's 2x2x2 Cube . . . 24
- 4.3 Towers of Hanoi . . . 26
- 4.4 Optimal Distances . . . 27
- 4.5 Getting Scrambled States . . . 28
- 4.6 Neural Network Implementation: Original and Modifications . . . 28
- 4.7 DAVI . . . 30
- 4.8 BWAS . . . 33

5 Results 35
- 5.1 Training Models and First Results for the 2x2x2 Rubik's Cube . . . 35
- 5.2 State generation distributions . . . 37
- 5.3 Completing the training for the 2x2x2 cube . . . 40
- 5.4 A new way of getting scrambled states: adaptative-K . . . 41
- 5.5 Final 2x2x2 trained models . . . 42
- 5.6 Admissibility and Consistency of the heuristics for the 2x2x2 cube . . . 43
- 5.7 Final 2x2x2 BWAS results . . . 44
- 5.8 Towers of Hanoi . . . 47
- 5.9 3x3x3 Rubik's Cube . . . 49

6 Conclusión 52

References 53

7 Appendix 55
- 7.1 Computer Specifications . . . 55
- 7.2 Model statistics, first training . . . 55
- 7.3 Model statistics, four trained models . . . 56
- 7.4 Hanoi: solutions found . . . 59

1 Introducción

1.1 Antecedentes y Motivación

Desde su invención en 1974 múltiples han sido las técnicas ideadas para resolver el Cubo de Rubik. Inicialmente, estos métodos han ido enfocados a la resolución manual del mismo en el menor tiempo posible. En la década de los 80 surgió un método popularmente conocido como método de Fridrich, que se usa a día de hoy en las competiciones de *speedcubing*. Sin embargo, estas técnicas manuales están pensadas para poder ser memorizadas y ejecutadas por un humano, y por tanto suelen ejecutar más movimientos de los realmente requeridos para su resolución.

Diversos métodos para resolver el Cubo de Rubik en el menor número de movimientos han ido surgiendo. Muchos de ellos aprovechan su estructura matemática de grupo algebraico para así poder aplicar resultados de teoría de grupos [15]. Sin embargo, debido al enorme tamaño del espacio de estados, muchas de estas técnicas requieren de unos elevados costes computacionales o de métodos novedosos que permitan ahorrar recursos. Uno de los primeros algoritmos que surgió para resolverlo en el menor número de movimientos fue el de Korf [16], que hace uso de *pattern databases*, heurísticas almacenadas en forma de tablas de búsqueda, para poder ejecutar una búsqueda IDA*. Sin embargo, las heurísticas empleadas utilizan características muy particulares de este problema y no son fácilmente generalizables a otros problemas similares, es decir, con un único estado final y elevado número de estados.

Con la motivación de resolver eficientemente el problema del Cubo de Rubik, de tal forma que sea generalizable a otros problemas del mismo tipo, han surgido varias soluciones dentro del ámbito del aprendizaje por refuerzo. El principal inconveniente de aplicar las técnicas estándar de aprendizaje por refuerzo en este tipo de problemas es que el refuerzo que permite aprender al *agente* no se encuentra hasta llegar al estado final. Por ello es necesario aplicar metodologías de aprendizaje con currículo [29], en las que se aprende primero de las instancias más sencillas del problema, para ir progresivamente aumentando el nivel de dificultad, imitando por tanto el orden de aprendizaje humano. En nuestro problema, esto se traduce en una versión simplificada de *prioritized sweeping* [19], en el que el agente aprende primero de cubos muy cercanos al estado resuelto, para progresivamente ir propagando la información aprendida a otros estados más lejanos. Esta idea es expuesta en los artículos de los algoritmos DeepCube [1] y DeepCubeA [2], en los que nos basaremos.

1.2 Objetivos y Plan de Trabajo

Los objetivos de este trabajo son varios. En primer lugar, queremos estudiar una introducción teórica al área del aprendizaje por refuerzo, que se detallará en la Sección 2, centrándonos en aquellos aspectos más relevantes en la resolución de nuestro problema. Después, en la Sección 3 se expondrá una descripción teórica del algoritmo DeepCubeA. Otro de los objetivos es obtener una implementación del algoritmo DeepCubeA, extrayendo algunas características de su predecesor, el algoritmo DeepCube, por motivos comparativos. Esta implementación se detalla en la Sección 4. Como último objetivo tenemos el de observar el comportamiento de la implementación obtenida con los Cubos de Rubik 2x2x2 y 3x3x3, y las Torres de Hanói, cuyos resultados se exponen en la Sección 5. Todo el código de este trabajo se encuentra en el repositorio de github [1]. Los pesos de los modelos ya entrenados están en la página de *releases* de ese mismo repositorio [2].

[1] https://github.com/CarlosTard/TFG-deepcube
[2] https://github.com/CarlosTard/TFG-deepcube/releases/tag/models

Introduction

Motivation

Since its invention in 1974, many techniques have been conceived to solve the Rubik's Cube. Initially, these methods were focused on solving it manually in the shortest possible time. In the 1980s, a method popularly known as Fridrich's method emerged, which is still used today in *speedcubing* competitions. However, these manual techniques are intended to be memorized and executed by a human, and therefore tend to apply more moves than are actually required to solve them.

Various methods for solving the Rubik's Cube in the fewest number of moves have been emerging. Many of them take advantage of its mathematical structure as an algebraic group, in order to be able to apply group theory results [15]. However, due to the enormous size of the state space, many of these techniques require high computational costs, or novel resource-saving methods. One of the first algorithms that emerged to solve it in the fewest number of moves was Korf's algorithm [16], which makes use of *pattern databases*, heuristics stored in the form of lookup tables, in order to execute an IDA* search. However, the heuristics employed make use of very particular characteristics of this problem, and are not easily generalizable to other similar problems, i.e., with a single final state and a large number of states.

With the motivation of solving the Rubik's Cube problem efficiently, in such a way that it is generalizable to other problems of the same type, several solutions have emerged within the field of reinforcement learning. The main drawback of applying standard reinforcement learning techniques to this type of problem is that the reinforcement that allows the agent to learn is not found until the final state is reached. It is therefore necessary to apply curriculum learning methodologies [29], in which the simplest instances of the problem are learned first, and then the level of difficulty is progressively increased, thus imitating the human learning order. In our problem, this translates into a simplified version of *prioritized sweeping* [19], in which the agent first learns from cubes very close to the solved state, and then progressively propagates the learned information to other, more distant states. This idea is developed in the papers of the DeepCube [1] and DeepCubeA [2] algorithms, on which we will build.

Objectives and work plan

The objectives of this work are several. First of all, we want to study a theoretical introduction to the area of reinforcement learning, which will be detailed in Section 2, focusing on those aspects most relevant to the resolution of our problem. Then, in Section 3, we will present a theoretical description of the DeepCubeA algorithm. Another objective is to obtain an implementation of the DeepCubeA algorithm, extracting some features from its predecessor, the DeepCube algorithm, for comparative reasons. This implementation is detailed in Section 4. The last objective is to observe the behavior of the implementation obtained with the 2x2x2 and 3x3x3 Rubik's Cubes, and the Hanoi Towers, the results of which are presented in Section 5. All the code for this work is on the github repository [3]. The weights of the trained models are on the releases page of the same repository [4].

[3] https://github.com/CarlosTard/TFG-deepcube
[4] https://github.com/CarlosTard/TFG-deepcube/releases/tag/models

2 Reinforcement Learning: Theoretical Foundations

In this section, I describe the central ideas of reinforcement learning. The concepts presented here are drawn from two main sources: *Reinforcement Learning, An Introduction* by Richard S. Sutton and Andrew G. Barto [25] (chapters 1-6 and 8.4), and *Foundations of Machine Learning*, by M. Mohri, A. Rostamizadeh, and A. Talwalkar [18] (chapters 1 and 17).

2.1 Machine Learning

Machine learning is a branch of Artificial Intelligence that focuses on the study, design and implementation of computer algorithms that use past data to automatically solve a broad variety of problems. Normally, those data consist of a number of **examples**, that is, individual instances of experience that we use to feed our machine learning algorithm. Each example has a set of attributes which we call **features**, and that are the measurable properties of the problem we are trying to solve. In addition, given an example, we could assign a **label** to it, that is, the true solution of our problem, which is the output we expect from our algorithm. In that case, it is a **labeled example**. Otherwise, if we can't assign a label (for example, we may not know the true solution of the problem), we call it an **unlabeled example**.

In this context, a **training set** is a set of examples used to train the algorithm. In addition, a **validation set** is a set of examples, different from the training set, used to tune the **hyperparameters** of the algorithm, that is, the parameters that control the learning process (for example, the learning rate, or the number of neurons in a neural network). Finally, the **test set**, separate from the training and validation sets, is used to evaluate the performance of the trained algorithm, after training it with the selected hyperparameters.

The types of problems that can be solved using machine learning techniques are numerous. Here, I present some examples:

- **Classification**: the problem is to classify an example, assigning a category to it, from a finite set of values. An example of this type of problem is to read facial expressions, that is, to learn whether a face displays happiness, sadness, fear, etc. If there are only two classes, it is called *binary classification*, while we use the term *multiclass classification* when there are more than two categories.
- **Regression**. In regression, we are interested in assigning a real value to each example. A problem that fits in the category of regression is to predict the global average temperature, based on carbon dioxide emissions (when studying global warming).
- **Clustering**: given a set of unlabeled examples, the task is to group examples into clusters that contain similar characteristics. For example, a job-posting website might want to use clustering to group job positions that are similar to each other, in order to ease the process of hiring and applying to jobs.
- **Recommendation** is the problem of showing relevant results to the user based on some criterion, like the expected rating the user would give to it, the expected revenue the user would generate, etc. Following the job-posting website example, a recommendation system would give the most relevant job positions to the user (or even, the relevant clusters from the clustering example), in order to maximize both the satisfaction of the soon-to-be employed and the number of filled positions.
- **Natural language processing (NLP).** This is not a unique problem type, but a wide variety of problems. NLP tasks can include [11]: classifying a text into sentiment categories (identifying opinions embedded in the text), summarizing a text document, sequence to sequence translation, generating text to describe an image, etc. For example, if we are given a molecule image, we could be interested in generating the chemical formula of that particular molecule. In this case, we would have to use image techniques (such as Convolutional Neural Networks) to extract

useful information from the image, and then NLP techniques to generate the text sequence of the chemical formula.

- **Dimensionality reduction** is the transformation of data, from the initial representation into a new representation with fewer dimensions, trying to preserve the information of the original data. It is used in data visualization, noise reduction and data preprocessing. For example, if the images of the molecules in the previous example have noise (or redundant information) produced by the microscope, we can apply dimensionality reduction techniques, previous to the convolutional model, in order to ease the training of the algorithm.
- **Object Detection** is a problem that arises in the field of image processing. The task is to find and identify objects of a certain class in an image. For example, we could be interested in detecting faces with a security camera, and then identifying the name of each person to catch the perpetrators of a crime. If we are interested in recognizing rare items, for example, anomalies in a lung radiography, this problem is also denominated **anomaly detection**. Not all classification algorithms are well suited for the anomaly detection task because of the class imbalance that is inherent to this problem.

For each of the previous types of problems, we could find different learning scenarios. These scenarios differ in the types of the training sets, and in the method to process the training, testing and validation data. Some of the most important learning scenarios are:

Supervised learning. In supervised learning, a labeled training set is given and a model is trained to output the true label for each of the training examples. Classification and regression are typical supervised learning tasks. In fact, for classification tasks, where the number of categories is finite, the algorithm can be trained to output an estimation of the probability that an example belongs to each class, which gives us more information than simply returning a prediction of the label. If we wanted to yield the label of a given example, we could return the class that maximizes the probability that the example belongs to that class.

Unsupervised learning. In this learning scenario the objective is to learn patterns from unlabeled data. Clustering and dimensionality reduction are two examples of unsupervised learning. The advantage of unsupervised learning methods is that they do not require previously generated labels, a process that is expensive and often requires manual intervention. The main disadvantage is that evaluating the performance of an algorithm is more difficult because the labels are not available. In fact, in some unsupervised learning tasks true labels do not even exist.

Semi-supervised learning is an scenario that combines supervised with unsupervised learning. It arises in situations where we have a small amount of labeled data (for instance, when it is very expensive to label examples), and also unlabeled data, and the goal is to make predictions about the unlabeled data. The key is to improve the information given by the labeled data with the distribution of examples in the unlabeled data. For example, a supervised model can be trained with the supervised data. Then, we can use that trained model to make predictions about the labels of the unlabeled set. The obtained labels are called pseudo-labels. Then, that model can be retrained with both the labeled set and the unlabeled set (using the pseudo-labels) to improve the efficiency of the previous model. Various supervised learning tasks can be framed as semi-supervised problems, such as regression or classification.

Transductive learning. Given a labeled and an unlabeled set, the objective is to predict the labels only for the unlabeled examples. This scenario makes it possible to use transduction, which is to derive the labels for the unlabeled examples from the labeled ones, without building any general rules. That is, transduction does not build a model, it only makes predictions based on the training set. An example of algorithm that can be applied in this scenario is the k-nearest neighbors algorithm.

In the following sections, the **Reinforcement learning** scenario is explained.

2.2 Reinforcement Learning

In reinforcement learning we are interested in maximizing a *reward* that an *agent* receives when interacting with an *environment*. This learning scenario is connected to the study of operant conditioning in biology and psychology, where a human or an animal learns or modifies a behavior when a positive (reinforcement) or negative (punishment) *reward* is received. Our *agent* interacts with the *environment* by executing *actions* that change its *state* in the *environment*. The *agent* has to determine the best sequence of *actions* to maximize the *reward* received, that is, it has to design a strategy, or *policy*. With that objective in mind, the *agent* has to favor *actions* that produce the most *reward*. But, in order to discover which are the best *actions*, it also has to select *actions* that it has not chosen before. Because of that, there is a trade-off between *exploration* (to discover the best *actions*), and *exploitation* (to select the *actions* that were effective in producing *reward* in the past), called the *exploration-exploitation trade-off* (also known as *exploration-exploitation dilemma*).

Two sub-scenarios may arise in relation to the knowledge we have about the *environment*. If a perfect model of the *environment* is known, the maximization of the *reward* turns out to be a *planning problem*. That is, if the agent has a model, it can build a policy by considering potential upcoming scenarios before interacting with the real world. Conversely, if the *agent* does not have a model of the *environment*, we have to learn from the *environment state* and *reward* information (received when taking *actions*) to both gain information about the *environment* and improve the *action policy*. In that case, we face a *learning problem*. This *learning problem* can be solved by a *model-based approach*, in which the *agent* first learns a model of the *environment* and then uses it to improve its *policy*, or by a *model-free approach*, in which the information gathered is directly employed to learn a *policy*.

2.3 The learning framework and MDP

In this section the previously mentioned reinforcement learning concepts are formalized. For this, a Markov decision process, or MDP, will be used, which is a theoretical framework that models the *environment* and the interactions with this *environment*.

The *agent* interacts with the *environment* at discrete time steps $t \in \mathbb{N}$. The initial *state* of the *environment* is $S_0 \in \mathcal{S}$, where $\mathcal{S}$ is the set of all the different *states* the *environment* can be in. At each time step t, the *agent* selects and *action* $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(s)$ are all the valid *actions* from state s, and $\mathcal{A}$ is the union of all of those sets, i.e. all possible *actions*. The next time step, the *agent* receives a *reward* $R_{t+1} \in \mathcal{R}$, and it reaches *state* $S_{t+1} \in \mathcal{S}$. Because of that, a sequence of *state, action*, *reward*, *state, action*, *reward*, ... is created $S_0, A_0, R_1, S_1, A_1, \dots$, which is called a *trajectory*. If the *action, reward* and *state* sets $\mathcal{A}$, $\mathcal{R}$, $\mathcal{S}$ are all finite, it is said to be a *finite MDP*.

As previously said, we are going to model the interactions with the *environment* as a Markovian process, that is, the probabilities of the transition and reward random variables $\mathcal{S}_t$ and $\mathcal{R}_t$ will depend only on the current state and the *action* taken from that *state*, but not on previous *states* or *actions*. Because of that, the *state* has to include all the information about the past interaction that affects the future. Starting from *state* s and taking *action* a, the probability of receiving a *reward* r and reaching state s' will be modelled as the conditional probability $p(s', r|s, a) : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. The transition $p(s'|s, a)$ and reward $p(r|s, a)$ probabilities of a finite MDP are defined using the previous conditional probability function:

$$p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \qquad\qquad p(r|s, a) = \sum_{s' \in \mathcal{S}} p(s', r|s, a)$$

Finally, there are environments with a *finite horizon* where the *agent* always finds a *terminal state* and then stops. They are called *episodic tasks* because all the interactions between the *environment* and the *agent* can be divided into *episodes*, finite trajectories that end at time T in the *terminal state*, where T is also a random variable. If a problem does not have a *finite horizon* and the *agent* runs indefinitely, it is said to be a *continuing task*, or a task with an *infinite horizon*.

2.3.1 Returns

The objective of reinforcement learning is to maximize the *expected return*, that is, the expected value of the cumulative *rewards* that the *agent* receives. In the case of *episodic tasks*, this *return* can be defined as the sum of the *rewards* that the *agent* gets over the course of an episode, that is, $G_t := R_{t+1} + R_{t+2} + \cdots + R_T$, where G_t is the sum of the *rewards* received after instant t. However, this approach is not valid for *infinite horizon* tasks, since the sum could be infinite. Assuming the sequence of *rewards* is bounded, we can introduce a *discount rate* $\gamma \in [0, 1)$, so that the return is bounded

$$G_t := \sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) = R_{t+1} + \gamma G_{t+1}$$

The previous return is the *discounted return*. The effect of the *discount rate* is to exponentially decrease the effect of future *rewards*, so values of γ close to 0 give more importance to present rewards than a value of γ close to 1. If $\gamma = 0$, we would only maximize the immediate *reward*. In fact, if we let $\gamma = 1$, the *discounted return* becomes the *episodic task* return. To adopt a notation that is valid both for *finite* and *infinite horizons*, we will use the *discounted return* definition. For *continuing tasks*, $\gamma \in [0, 1)$, while for *episodic tasks* we will choose $\gamma = 1$ and all the returns past the termination time T are defined to be zero, $R_{T+k} := 0$ for $k = 1, 2, 3, \ldots$

2.3.2 Policies

The *policy* is the strategy our *agent* is going to follow to select the *actions* to apply depending on the *state* of the *environment*. Formally, a *policy* π is a probability distribution over the *actions* $\mathcal{A}(s)$, conditioned on the *state* of the *environment* being s. That is, $\pi(a|s)$ is the probability that the *agent* chooses action a when the state of the *environment* is s.

If for any *state* s, there exists an *action* a such that $\pi(a|s) = 1$, that means the *agent* is totally certain about the *action* it is going to choose, because whenever it encounters *state* s, it will choose *action* a. This is called a *deterministic policy*, and induces a mapping from *states* to *actions*. In that case, with an abuse of notation, we will denote $\pi(s) = a$ (where $\pi(a|s) = 1$). If the *policy* is not *deterministic*, it is said to be a *stochastic policy*.

2.3.3 Value Functions

Our objective is to find a *policy* that for all *states* s, it maximizes the expected *return* when starting in s and following the *policy*. To formalize this idea, we will first define the *value function* for a *policy* π that, for each *state* s, returns the expected *return* when starting from *state* s and following the *policy*. This function is called *state-value function*, *policy value function* or simply *value function* under a *policy* π. That is:

$$v_\pi(s) := \mathbb{E}_{A_k \sim \pi}\left[G_t | S_t = s\right]$$

where $\mathbb{E}_{A_k \sim \pi}$ represents the expectancy when taking the *actions* of the *trajectory* by sampling the probability distribution given by π (that is, $A_k \sim \pi$), and t is any time step. It will also be very convenient to consider an *action-value function* for a *policy* π that, for each *state* s, returns the expected *return* when starting from *state* s, applying action a, and following the *policy* thereafter:

$$q_\pi(s, a) := \mathbb{E}_{A_k \sim \pi}\left[G_t | S_t = s, A_t = a\right]$$

The similarity between the *value function* and the *action-value function* is remarkable. Let's write v_π in terms of q_π and the *policy* π.

$$v_\pi(s) = \mathbb{E}_{A_k \sim \pi}\left[G_t | S_t = s\right] = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \mathbb{E}_{A_k \sim \pi}\left[G_t | S_t = s, A_t = a\right] = \sum_{a \in \mathcal{A}(s)} \pi(a|s) q_\pi(s, a)$$

which can be interpreted as considering the *value* of all possible *actions* from *state* s, taking into account how likely it is to take each *action* under policy π. This equation has the particularity of not depending on the function p, so if we already have the *action-value function* q_π, we don't need a model of the *environment* to compute the *value function* v_π. Also, we have that

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) q_\pi(s,a) = \mathbb{E}_{a\sim\pi}\left[q_\pi(s,a)|S_t = s\right] \tag{2.1}$$

Now, let's write q_π in terms of v_π and the *environment* model p.

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_{A_k\sim\pi}\left[G_t|S_t = s, A_t = a\right] = \mathbb{E}_{A_k\sim\pi}\left[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a\right] && (2.2) \\ &= \sum_{r\in\mathcal{R}, s'\in\mathcal{S}} p(s',r|s,a)\left[r + \gamma \mathbb{E}_{A_k\sim\pi}\left[G_{t+1}|S_{t+1} = s'\right]\right] && (2.3) \\ &= \sum_{r\in\mathcal{R}, s'\in\mathcal{S}} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] && (2.4) \end{aligned}$$

This formula does explicitly involve the *environment* model, but not the *policy* followed (it is implicit through the *value function*). This is because after taking *action* a, we need to know what *state* are we going to be in and the *reward* associated with that transition before being able to utilize the information that the *value function* provides. Following the same methodology, $v_\pi(s)$ can be written in terms of the value of the successors of state s, satisfying the following recursive relationship:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_{A_k\sim\pi}\left[G_t|S_t = s\right] = \mathbb{E}_{A_k\sim\pi}\left[R_{t+1} + \gamma G_{t+1}|S_t = s\right] && (2.5) \\ &= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{r\in\mathcal{R}, s'\in\mathcal{S}} p(s',r|s,a)\left[r + \gamma \mathbb{E}_{A_k\sim\pi}\left[G_{t+1}|S_{t+1} = s'\right]\right] && (2.6) \\ &= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{r\in\mathcal{R}, s'\in\mathcal{S}} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] && (2.7) \end{aligned}$$

Similarly, $q_\pi(s,a)$ can be written in terms of the action-value of the possible successors of state s when applying action a. Therefore, it satisfies the following recursive relationship, that can be obtained by using the relationships between v_π and q_π:

$$\begin{aligned} q_\pi(s,a) &= \sum_{r\in\mathcal{R}, s'\in\mathcal{S}} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] && (2.8) \\ &= \sum_{r\in\mathcal{R}, s'\in\mathcal{S}} p(s',r|s,a)\left[r + \gamma \sum_{a'\in\mathcal{A}(s')} \pi(a'|s') q_\pi(s',a')\right] && (2.9) \end{aligned}$$

Equations 2.7 and 2.9 are the *Bellman equations* of a policy π for v_π and q_π respectively. They represent the value of a *state* in terms of the value of its successors, which can be interpreted as a one-step lookahead from state s. In fact, the *Bellman equation* 2.7 of a policy π is the relationship a function has to satisfy in order to be the *value function* under that *policy*. Respectively, the *Bellman equation* 2.9 of a *policy* π is the relationship a function has to satisfy in order to be the *action-value* function of that *policy*. These equations will be used later as an update rule to solve the policy evaluation problem.

2.3.4 Optimality

As stated at the beginning of the last section, our objective is to find a *policy* that for all *states* s maximizes the expected *return* when starting in s and following the *policy*. Now, we can formalize this goal defining the condition that we are looking for in a *policy*.

Definition 2.1. A *policy* π_* is an *optimal policy* iff, for each state $s \in \mathcal{S}$, $v_{\pi_*}(s) \geq v_\pi(s)$ for any *policy* π.

We will also say that *policy* π_1 is better than or equal to a *policy* π_2 if its expected return is greater than or equal to the expected return of *policy* π_2 for all states. That is, for each state $s \in \mathcal{S}$, $v_{\pi_1}(s) \geq v_{\pi_2}(s)$. Therefore, an *optimal policy* is better than or equal to all the other possible policies.

Example: imagine a one dimensional world made out of 7 cells. We start in a particular cell, and the objective is to move to a goal cell, the central one, in the minimum number of steps. The rules are that, if we are in cell i, we can move to cells $(i+1)\%7$ (right move) or $(i-1)\%7$ (left move). That is, we can move left or right, but the *actions* that take us off the world result in a teleportation to the other side of it. When you are in the goal cell, the episode finishes. Any left or right *action* results in a punishment of -1, and the agent dies after time step 30. As we are in a finite-horizon *environment*, we will take $\gamma = 1$. There are 7 *states*, one for each cell. A non-optimal *policy* π would be, for example, the deterministic *policy* that always takes a right move, and therefore assigns a zero probability of taking the left action. The value function for each *state* is very simple to compute, since it is the sum of *rewards* received until reaching the *goal state*. On the other hand, the optimal *policy* π_* is the one that minimizes the number of moves taken by the *agent*. For cells 0–2 , it always chooses right, and for cells 4–6, it chooses left. The *state-value* functions of these two *policies* are shown in Figure 1.

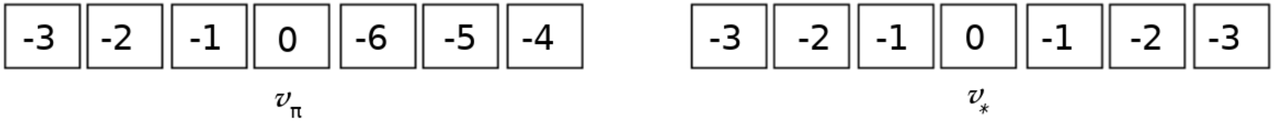


Figure 1: state-value pairs for *policies* π and π_* in the one dimensional world.

Notice that, if an *optimal policy* exists, there could be multiple *optimal policies*, but all *optimal policies* share the same *value function*. This is due to the fact that, if π_1 and π_2 are two *optimal policies*, $v_{\pi_1}(s) \geq v_{\pi_2}(s)$ and $v_{\pi_2}(s) \geq v_{\pi_1}(s)$ for all states, so $v_{\pi_1}(s) = v_{\pi_2}(s)$. This fact can be also stated as $v_{\pi_*}(s) = \max_{\pi} v_\pi(s)$. Therefore, from the relationship 2.4 between *value functions* and *action-value functions*, all *optimal policies* also share the same *action-value function*, which is $q_{\pi_*}(s,a) = \max_{\pi} q_\pi(s,a)$. The following theorem is going to give us an important property about *policies*, from which an important equation for *optimal policies* emerges.

Theorem 2.2. (Policy improvement theorem) Let π_1 and π_2 be two *policies*. Assume that v_{π_1} is bounded (which is trivial for a finite $\mathcal{S}$). Suppose that $\mathbb{E}_{a\sim\pi_2}\left[q_{\pi_1}(s,a)|S_t = s\right] \geq \mathbb{E}_{a\sim\pi_1}\left[q_{\pi_1}(s,a)|s_t = s\right]$ holds for all $s \in \mathcal{S}$. Then, $v_{\pi_2}(s) \geq v_{\pi_1}(s)$ for all $s \in \mathcal{S}$. Moreover, if there is a *state* $s_1 \in \mathcal{S}$ in which $\mathbb{E}_{a\sim\pi_2}\left[q_{\pi_1}(s_1,a)|S_t = s_1\right] > \mathbb{E}_{a\sim\pi_1}\left[q_{\pi_1}(s_1,a)|s_t = s_1\right]$, then the result holds with a strict inequality for at least one *state* s_2, that is $v_{\pi_2}(s_2) > v_{\pi_1}(s_2)$.

Proof: There are two *trajectories* involved in this proof. In order not to confuse them, we denote the *trajectory* created by *policy* π_1 in lowercase, that is, a_k, s_k and r_k, and the *trajectory* created by *policy* π_2 in uppercase letters, that is, A_k, S_k and R_k. For any $s \in \mathcal{S}$, we have

$$\begin{aligned}
v_{\pi_1}(s) &\overset{2.1}{=} \mathbb{E}_{a_t\sim\pi_1}\left[q_{\pi_1}(s,a_t)|s_t = s\right] \leq \mathbb{E}_{A_t\sim\pi_2}\left[q_{\pi_1}(s,A_t)|S_t = s\right] \overset{def}{=} \mathbb{E}_{A_t\sim\pi_2}\left[\mathbb{E}_{a_k\sim\pi_1}\left[G_t|s_t = s, a_t = A_t\right]|S_t = s\right] \\
&\overset{(1)}{=} \mathbb{E}_{A_t\sim\pi_2}\left[R_{t+1} + \gamma\mathbb{E}_{a_k\sim\pi_1}\left[G_{t+1}|s_{t+1} = S_{t+1}\right]|S_t = s\right] \overset{def}{=} \mathbb{E}_{A_t\sim\pi_2}\left[R_{t+1} + \gamma v_{\pi_1}(S_{t+1})|S_t = s\right] \\
&\overset{(2)}{\leq} \mathbb{E}_{A_t\sim\pi_2}\left[R_{t+1} + \gamma\mathbb{E}_{A_{t+1}\sim\pi_2}\left[R_{t+2} + \gamma v_{\pi_1}(S_{t+2})|s_{t+1} = S_{t+1}\right]|S_t = s\right] \\
&\overset{(3)}{=} \mathbb{E}_{A_t\sim\pi_2, A_{t+1}\sim\pi_2}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_{\pi_1}(S_{t+2})|S_t = s\right]
\end{aligned}$$

where in (1) the fact that $G_t = R_{t+1} + \gamma G_{t+1}$ is used in combination with the fact that R_{t+1} does not depend on the distribution π_1 since the *action* at time t, a_t, has been sampled from π_2. In (2), the same process that has been applied to $v_{\pi_1}(s)$ is now applied to $v_{\pi_1}(S_{t+1})$. Finally, in (3), the two

expectations have been merged since they take A_t and A_{t+1} from the same distribution. Therefore, iterating this process for any $N \geq 1$:

$$v_{\pi_1}(s) \leq \mathbb{E}_{A_k \sim \pi_2}\left[\sum_{k=1}^{N} \gamma^{k-1} R_{t+k} + \gamma^{N+1} v_{\pi_1}(S_{N+1}) | S_t = s\right]$$

Because v_{π_1} is bounded, the limit when $N \to \infty$ gives us:

$$v_{\pi_1}(s) \leq \mathbb{E}_{A_k \sim \pi_2}\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} | S_t = s\right] = v_{\pi_2}(s)$$

which gives us the desired result. Moreover, any strict inequality of the theorem hypothesis in a *state* s_1 produces a chain of inequalities, so we can take $s_2 = s_1$, which produces $v_{\pi_2}(s_2) > v_{\pi_1}(s_2)$, as we wanted. □

Lemma 2.3. Let π be a *policy*. If for any *state*, *action* pair (s, a) with $\pi(a|s) > 0$, $a \in \operatorname{argmax}_{a' \in \mathcal{A}} q_\pi(s, a')$ holds, then the policy π is optimal. That is, if π only chooses *action* a from *state* s if a maximizes the *action-value function* $q_\pi(s, \cdot)$, then π is optimal.

Proof: [18, Theorem 17.7] □

We might ask ourselves under what conditions does an optimal policy exist, if there are any. For a finite MDP we have the following result. It is very promising since we are only going to work with finite problems.

Theorem 2.4. Any finite MDP admits an *optimal deterministic policy.*

Proof: In a finite MDP setting, there is only a finite number of deterministic *policies* since only a finite number of *actions* can be chosen from any *state*, and the *environment* has a finite number of *states*. Because of this, we can choose a deterministic *policy* π that, compared to all the other deterministic *policies*, maximizes the quantity $\sum_{s \in \mathcal{S}} v_\pi(s)$. Then, π is optimal. If it were not optimal, lemma 2.3 tells us that there is a state s' where we do not choose an action that maximizes $q_\pi(s', \cdot)$. If we take $a' \in \operatorname{argmax}_{a \in \mathcal{A}(s')} q_\pi(s', a)$ and define π' as

$$\pi'(a|s) = \begin{cases} \pi(a|s) & \text{if } s \neq s' \\ 1 & \text{if } s = s' \text{ and } a = a' \\ 0 & \text{if } s = s' \text{ and } a \neq a' \end{cases}$$

then $\mathbb{E}_{a \sim \pi'}[q_\pi(s', a) | S_t = s'] = \max_{a \in \mathcal{A}(s')} q_\pi(s', a) > \mathbb{E}_{a \sim \pi}[q_\pi(s', a) | S_t = s']$, and $\mathbb{E}_{a \sim \pi'}[q_\pi(s, a) | S_t = s] = \mathbb{E}_{a \sim \pi}[q_\pi(s, a) | S_t = s']$ if $s \neq s'$, which, by Theorem 2.2, implies that $v_{\pi'}(s) \geq v_\pi(s)$ for all *states* s, and that there is a *state* s_2 where $v_{\pi'}(s_2) > v_\pi(s_2)$. Because π' is also deterministic, this contradicts the fact that π maximizes $\sum_{s \in \mathcal{S}} v_\pi(s)$. □

Now, let π_* be an *optimal policy.* From 2.1 we have that $v_{\pi_*}(s) = \mathbb{E}_{a \sim \pi_*}[q_{\pi_*}(s, a) | S_t = s] \leq \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a)$. Also, from the policy improvement theorem, $v_{\pi_*}(s) = \mathbb{E}_{a \sim \pi_*}[q_{\pi_*}(s, a) | S_t = s] \geq \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a)$. This is because, if we had that $\mathbb{E}_{a \sim \pi_*}[q_{\pi_*}(s', a) | S_t = s'] < \max_{a \in \mathcal{A}(s')} q_{\pi_*}(s', a)$ for some *state* s', then there would exist a policy π'_* that would be better than the *optimal.* That is, following the same reasoning as in Theorem 2.4, if we take $a' \in \operatorname{argmax}_{a \in \mathcal{A}(s')} q_{\pi_*}(s', a)$ and define π'_* as

$$\pi'_*(a|s) = \begin{cases} \pi_*(a|s) & \text{if } s \neq s' \\ 1, & \text{if } s = s' \text{ and a = a'.} \\ 0, & \text{if } s = s' \text{ and a} \neq \text{a'.} \end{cases}$$

then $\mathbb{E}_{a\sim\pi'_*}\left[q_{\pi_*}(s',a)|S_t=s'\right] = \max\limits_{a\in\mathcal{A}(s')} q_{\pi_*}(s',a) > \mathbb{E}_{a\sim\pi_*}\left[q_{\pi_*}(s',a)|S_t=s'\right]$ and $\mathbb{E}_{a\sim\pi'}\left[q_\pi(s,a)|S_t=s\right] = \mathbb{E}_{a\sim\pi}\left[q_\pi(s,a)|S_t=s'\right]$ if $s\neq s'$, contradicting the optimality of π_*. Therefore, from these inequalities, we have that $v_{\pi_*}(s) = \max\limits_{a\in\mathcal{A}(s)} q_{\pi_*}(s,a)$ for all *states* s.

From this result, we can derive the **Bellman optimality equation**:

$$
\begin{aligned}
v_{\pi_*}(s) &= \max_{a\in\mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_{a\in\mathcal{A}(s)} \mathbb{E}_{A_k\sim\pi_*}\left[R_{t+1}+\gamma G_{t+1}|S_t=s, A_t=a\right] \\
&\overset{2.4}{=} \max_{a\in\mathcal{A}(s)} \sum_{r\in\mathcal{R},s'\in\mathcal{S}} p(s',r|s,a)\left[r+\gamma v_{\pi_*}(s')\right] \\
&= \max_{a\in\mathcal{A}(s)} \mathbb{E}\left[R_{t+1}+\gamma v_{\pi_*}(S_{t+1})|S_t=s, A_t=a\right]
\end{aligned}
$$

An analogous form of the **Bellman optimality equation** for q_* can be obtained from the fact that $v_{\pi_*}(s) = \max\limits_{a\in\mathcal{A}(s)} q_{\pi_*}(s,a)$ and from 2.4:

$$
\begin{aligned}
q_{\pi_*}(s,a) &= \sum_{r\in\mathcal{R},s'\in\mathcal{S}} p(s',r|s,a)\left[r+\gamma \max_{a'\in\mathcal{A}(s')} q_{\pi_*}(s',a')\right] \\
&= \mathbb{E}\left[R_{t+1}+\gamma \max_{a'\in\mathcal{A}(s')} q_{\pi_*}(S_{t+1},a')|S_t=s, A_t=a\right]
\end{aligned}
$$

2.3.5 Value Iteration

The existence of an *optimal deterministic policy* for finite Markov decision processes demonstrated in theorem 2.4 brings us closer to obtaining an **optimal policy**. The *optimal policy* may not be unique, but all *optimal policies* share the same *value function*. The idea is to obtain the optimal *value function*. **Value Iteration** is an iterative algorithm that aims to approximate this optimal value function given a perfect model of the *environment*. It starts with a random initialization for the approximation of the value function, v_0 (except for the terminal *states*, where this value has to be defined as 0). At each iteration $k+1$, the algorithm updates the value from the previous iteration k by using the Bellman optimality equation as an update rule:

$$
v_{k+1}(s) \;=\; \max_{a\in\mathcal{A}(s)} \mathbb{E}[R_{t+1}+\gamma v_k(S_{t+1})|S_t=s, A_t=a] \;=\; \max_{a\in\mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right] \tag{2.10}
$$

In practice, this process is not guaranteed to converge in finite steps, so normally it is stopped when the left and right sides meet some condition (for example, when they are closer than a threshold). For finite *states*, this process is implemented as any other dynamic programming algorithm, for example by keeping v_k in a table that at each possition s stores $v_k(s)$. Assuming that $v_N \approx v_*$ being N the final iteration, we can obtain a (close to) *optimal deterministic policy* by choosing $\pi(s) = \operatorname*{argmax}\limits_{a\in\mathcal{A}(s')} \sum_{r\in\mathcal{R},s'\in\mathcal{S}} p(s',r|s,a)\left[r+\gamma v_\pi(s')\right]$.

3 DeepCubeA: Theoretical Foundations

The following sections cover the study, design, implementation and analysis of a reinforcement learning algorithm that, in combination with a modified variant of A^* search, learns how to solve problems with large state spaces and only one goal state. This work is based on a paper from 2019 titled *Solving the Rubik's Cube with deep reinforcement learning and search* [2], where the authors introduce an algorithm called DeepCubeA. It makes use of a deep neural network (Sections 3.2 and 4.6) that, by employing a reinforcement learning technique called DAVI (Sections 3.3 and 4.7), is trained to predict how far is a state from the goal state. Then, this network is used as a heuristic to search for the optimal path to the goal. The search algorithm is named BWAS (Sections 3.4 and 4.8), a variant of A^* that weights the contribution of the path cost to the total cost of each node, and that expands a batch of nodes in parallel instead of only one. I am also going to study some details about the previous paper by the same authors called *Solving the Rubik's Cube with Approximate Policy Iteration* [1], mainly some aspects of state generation.

3.1 Description of the problem we want to solve

As mentioned in the introduction, we are interested in problems with a large state space and only one goal state (also called final state or solved state). Complete knowledge of the environment is assumed, and we will be focused mainly on deterministic problems that can be modelled as a finite MDP. Starting from any state, the objective is to reach the goal state applying as few actions as possible, that is, to find the optimal path from an initial state to the goal state. There are many examples of this type of problem, mainly puzzles such as all the Rubik's Cube versions (2x2x2, 3x3x3 [8], 4x4x4, etc), sliding puzzles (the 15 [13], 24, 35, ..., $n^2 - 1$ puzzles), Lights Out [17], Sokoban [6] and the Towers of Hanoi [24].

3.1.1 Rubik's Cube

The Rubik's Cube is one of the problems we are going to solve, and we will focus on the 3x3x3 and 2x2x2 ones.

The **3x3x3 Rubik's Cube** [8] is a 3-dimensional cube-shaped puzzle, that consists of 26 smaller cubes called *cubelets* (or cubies). There are 6 center cubelets (with one sticker each), 12 edge cubelets (2 stickers) and 8 corner cubelets (3 stickers). Each sticker is painted in blue, yellow, orange, red, white or green. In the solved state, all the nine stickers of each face are the same color: the yellow face opposes the white one, the red face is opposed to the orange one, and the blue one opposes the green face. We will use the orientation in figure 2, with the yellow face up, the blue face in front, the white face at the bottom, the red one at the right. As a naming convention, we will call the upper face U, the down face D, the right, R, the left L, the front face F, and B will be the back one. With an abuse of notation, the clockwise 90º degrees turns of each face will be also called U, D, R, L, F and B, while the counter-clockwise 90º degrees turns will be called U', D', R', L', F'and B', as explained in figure 3. Notice that the center cubelets remain fixed after appling each of the 12 actions; this will be important in the implementation section.

The notation explained before is called quarter-turn metric because all the actions are 90º turns, so if we want to perform a 180º rotation we will need to apply two 90º actions. There is also a half-turn metric that adds six actions to the quarter-turn metric set that correspond to the 180º turn of a face, and are represented as U2, D2, R2, L2, F2 and B2. Notice that an optimal path in the half-turn metric (one that uses the minimum possible actions) between a state and the solved state is never worse than an optimal path in the quarter-turn metric. That is, the quarter-turn metric one has to perform at least the same number of actions, or even more. God's Number is the maximum number of actions we need in order to optimally solve any state of the cube. That is, an optimal path between

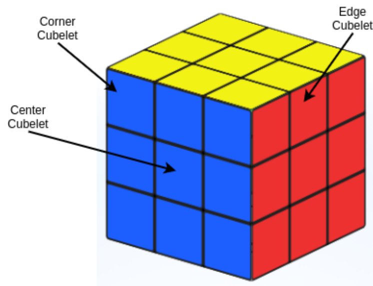


Figure 2: Solved 3x3x3 Cube.

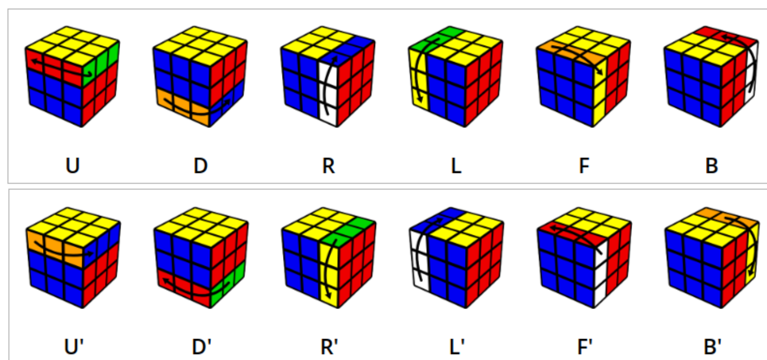
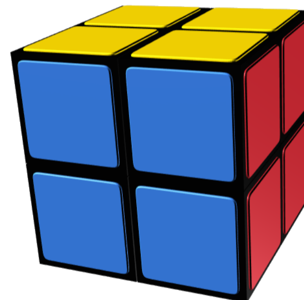


Figure 3: Rubik's Cube Move Notation. [7]

any state and the solved state has a length less than or equal to the God's Number, but never worse. God's Number for the 3x3x3 Rubik's Cube in half-turn metric is 20 [26], while for the quarter-turn metric it's 26 [27]. We will use the quarter-turn metric.

There are 43,252,003,274,489,856,000 [23] different possible states of the 3x3x3 Rubik's Cube. Let's consider the 2x2x2 Rubik's Cube, a similar problem with far fewer states.

The **2x2x2 Rubik's Cube** is a 3-dimensional cube shaped puzzle, very similar to the 3x3x3 one. It has no centers and no edges, just 8 corners with 3 stickers each. We will use the same notation as in the 3x3x3 version both in the quarter-turn and the half-turn metrics.



The God's Numbers for this cube are 14 using the quarter-turn metric and 11 using the half-turn metric, and it has 3,674,160 possible states (these results are easily obtained from the optimal solver in Section 4.4).

If we wanted to build an **optimal solver** we could implement a simple breadth-first search for the 2x2x2 cube, as we will see in Section 4.4. However, building an optimal solver with a simple BFS for the 3x3x3 cube is impossible due to the high number of states. Because of that, a better solution is needed. In 1997, Korf [16] introduced a strategy based on Iterative Deepening A^* (IDA^*). His algorithm uses pattern databases, lookup tables that store multiple heuristics (in this case, one table for the corners, and two tables of six edge cubelets each), and by combining them (by using a max operator), Korf obtains an admissible heuristic. However, according to the DeepCubeA authors, this solver is very slow. Although better alternatives have been found (for example the one used to find the God's Number of the 3x3x3 cube [22]), our purpose is to implement a near-optimal algorithm using reinforcement learning techniques.

3.1.2 Towers of Hanoi

The Towers of Hanoi[24] is a mathematical puzzle that consists of m posts and n disks of different sizes, with $m \geq 3$, and we will refer to it as $H(n, m)$. In the initial state, all disks are stacked on the first post, in increasing order in size from top to bottom. The goal is to move all disks from the first post to the last one. The rules are simple: only one disk may be moved at a time, taking the upper disk from one of the towers and placing it on an empty rod, or on top of another stack. A disk can only be placed on top of a bigger disk, maintaining the increasing order in sizes.

It can be easily proven that the state space has a size of m^n. The typical choice for m is to have 3 posts. With 3 posts, the problem has only one optimal solution of length $2^n - 1$ [5]. This solution can be described recursively as follows. The objective is to move n disks from the first post, the source,

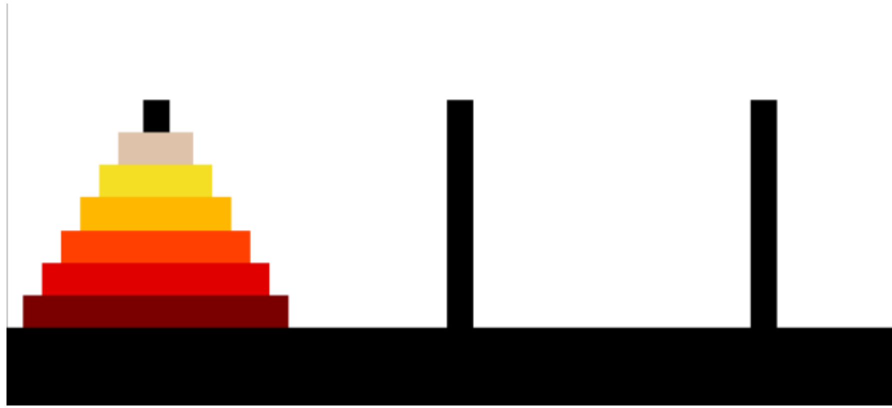


Figure 4: Initial state of a Tower of Hanoi consisting of 3 posts and 6 disks. © Wikimedia Commons User:Trixx / CC-BY-SA-3.0

to the rightmost one, the target, using the central post as an auxiliary tower. The base case is to move 0 disks, which is trivially solved and does not produce any move. The recursive case consists of three steps. First, the smaller $n-1$ disks are moved from the target post to the auxiliary one, which is a sub-problem of the Towers of Hanoi. Thus, it can be solved by recursively calling this procedure, but swapping the roles of the target and the auxiliary posts. The second step is to move the bigger disk from the source to the target post. The last step is to recursively move the $n-1$ disks on the auxiliary post to the target one, using the source post as auxiliary. After the last step, all disks are placed on the last tower in ascending order, so the problem is solved.

The optimal solution length of $2^n - 1$ is going to present a challenge for the algorithm introduced in the following sections, hence improving our understanding of it. For example, with 3 towers and 10 disks, the optimal solution has a length of 1023 moves, and the problem has only 59,049 states. Compared to the 2x2x2 Rubik's cube with a God Number of 14 and 3,674,160 states, the Towers of Hanoi has 1.6% of the states that the 2x2x2 cube has, but with an optimal solution 73 times longer.

3.2 Neural Networks

We are going to use a neural network as a heuristic, so it is essential to fully understand how a neural network works, the purpose of the neural network in the DeepCubeA algorithm, and its architecture. Let's begin with the simplest type of neural network.

A **feedforward neural network** [20] is a sequence of layers of neurons. Each neuron performs an affine transformation of its inputs, and then applies an activation function. Therefore, it is a function $f(x) = \phi(b + \sum w_i \cdot x_i)$, where ϕ is the activation function, x_i the inputs of the neuron, and $w_i \in \mathbb{R}$ and $b \in \mathbb{R}$ are called weights, coefficients or parameters. The activation function is normally chosen to be nonlinear, to make it possible to learn complex tasks. An example of activation function, and the one we are going to use, is **ReLU** [9] (Rectified Linear Unit), which is the function $\phi(x) = \max(x, 0)$. In a feedforward neural network, the first layer of neurons take its inputs from the input of the network, the second layer uses the output of the first layer as inputs, and so on. Finally, the output of the network is the last layer's output. A layer of a feedforward neural network is called fully-connected or densely-connected layer, because it is connected to all the neurons of the previous layer in order to perform the affine transformation of its outputs.

Not every layer of a neural network is a fully-connected layer. Another type of layer we are going to use is the **batch normalization** [12] layer (or batch norm). It applies a transformation that normalizes the input, maintaining the mean close to 0 and the standard deviation close to 1. Ideally, this normalization would be performed over the entire training set, but usually this is impractical with

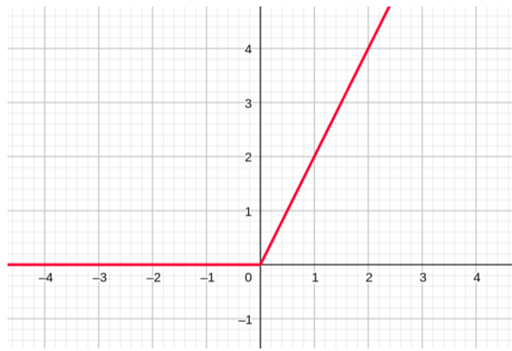
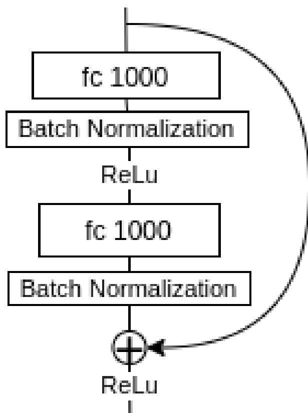


Figure 5: ReLU activation function.

large datasets, so the training set is divided in batches, and the normalization is applied over each batch. This normalization process is conducted to make the neural network fast and more stable, and to improve the generalization ability of the network. The batch normalization layer we are going to use [3] works differently during training and during inference. During training, it normalizes the input taking into account only the current training batch, but not the previous batch of inputs. During inference, it normalizes the input using a moving average of the mean and standard deviation of the batches it has seen during training. Because of this, batch normalization performs well when the training and inference data have similar means and standard deviations.



Residual block, 2 fully-connected layers and batch normalization.

Another technique we are going to use is **skip connections**, also called **shortcuts**. They are connections that skip one or more layers. The arrow in the left figure is an example of a skip connection, where the input is added to the output of the last batch normalization layer. It skips two fully-connected (fc) layers with 1000 neurons each, and two batch normalization layers. The skip connection together with the layers it skips is called a **residual block**. This structure allows training of deep neural networks, neural networks that have many layers, because it reduces the degradation problem [10]. The degradation problem is the decrease of the accuracy of the model when adding more layers to a neural network. This is counter-intuitive, since, in theory, we could approximate the identity function with the added layers. Adding a shortcut in a residual block ensures that we propagate the identity function, thus reducing the degradation problem.

Training and inference, mentioned before, are two of the main operations performed over a neural network. **Inference, or evaluation** means to compute the output of the network given a particular input. During a **supervised training**, we are given a training set with input samples and their expected outputs (called labels). The objective is to update the parameters of a neural network to minimize the distance between the expected output and the output produced by the neural network when given the training set as input. This distance is measured by a loss function, such as the mean squared error, the one that the DeepCubeA paper uses. The **mean squared error** computes the mean of the squared distances between the labels and the output of the network: $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, where y_i are the labels and $\hat{y}_i$ the neural network predictions. The update of the parameters to minimize the loss function is as follows. First, the gradient of the loss function with respect to the parameters is computed using an algorithm called backpropagation. Then, an optimization algorithm is applied. For example, stochastic gradient descent updates the parameters in the opposite direction of the gradient computed by backpropagation. The intuition is that the gradient is the direction of steepest ascent, so the opposite of this gradient points in the direction of steepest descent. In fact,

if the gradient is very small, the weights are not updated, and the network is unable to learn more. Batch normalization reduces this problem, called vanishing gradient problem. A hyperparameter called **learning rate** controls how much to move in the direction of the gradient. A high learning rate produces a faster movement towards the minimum but more instability, whereas a small learning rate produces a slow but stable convergence. There are methods with adaptative learning rates to improve convergence. One of them is **Adam** [14], the one we are going to use. Adam computes bias-corrected estimations of the first and second moments of the gradients. It does so by maintaining moving averages of the uncentered estimates of the mean and the variance, and then applying a bias correction transformation. These estimations are then employed to compute the direction and step size used to update the parameters. These updates are applied by dividing the input array in batches, and for each batch, the updates are applied only when all the gradients and estimations for all the samples in the batch have been computed.

Our neural network is going to act as a heuristic. It takes a state of a particular problem as an input (for example, in the 2x2x2 cube a state is an array of size 144) and it returns a real value that approximates the number of actions to reach the goal state. This number is called value, cost-to-go or distance to goal state, and the value of the goal state is set to 0.

The architecture of the neural network introduced in the DeepCubeA paper is shown in Figure 6, and it works as follows. First, a fully-connected layer is applied. It is very important to have a high number of neurons in this layer (5000 for the 3x3x3 cube), since it is responsible for extracting all the features from the raw state. Then, after every fully-connected layer, a batch normalization is applied to improve generalization and stability of convergence, and a ReLU activation funtion afterwards (this order is suggested in the batch normalization's paper [12]). After this, another fully connected layer of neurons is applied, with its corresponding batch normalization and ReLU layers. Then, the objective is to add more layers to obtain a deep neural network that is capable of learning complex tasks (like the problem we are trying to solve). But, because of the degradation problem, the best way of doing this is to use residual blocks. Figure 6 is the architecture of the original DeepCubeA model, in which there are 4 residual blocks. Each block has two fully connected layers of 1000 neurons each, with their respective batch normalization layers and ReLU activation functions. Because of the skip connection, the last ReLU is applied after adding the output of the batch normalization and the input of the residual block.

Finally, after the residual blocks, there is a fully connected layer of one neuron without activation function, that performs a linear combination of the output of the last residual block. Its purpose is to aggregate all the information found by the previous layers, and produce the final output of the network, that is, the distance to the goal state.

3.3 DAVI

Our purpose now is to design an algorithm to train the neural network so that, for each state it returns the cost-to-go. However, a supervised learning approach is impracticable since we don't know the real distances to the goal state beforehand. A reinforcement learning solution based on value iteration would be to adapt equation 2.10 to our problem. First, we need an MDP.

The MDP of our problem: As mentioned in Section 3.1, we are solving deterministic problems of which we have a complete knowledge of their environment. That is, the probability function p is known, and the problem is deterministic, so the probability function is degenerate, i.e. from *state* $s \in \mathcal{S}$, taking *action* $a \in \mathcal{A}(s)$ always results in the same *reward* r and the agent always reaches the same next state s'. Namely, $p(s', r|s, a) = 1$ and it is $p(s'', r'|s, a) = 0$ for $(s'', r') \neq (s', r)$. In this case, we define $S(s, a) := s'$ and $R(s, a) := r$.

Example: 3x3x3 Rubik's Cube. In this cube, the set $\mathcal{S}$ contains all 43,252,003,274,489,856,000 *states*. The set of *actions* that can be taken from any state is always the same, i.e. $\mathcal{A} = \mathcal{A}(s) = \{$U, D, R, L, F, B, U', D', R', L', F', B'$\}$, except for the goal state, where no action can be chosen $\mathcal{A}(s) = \{\}$. The transition function is given in Figure 3.

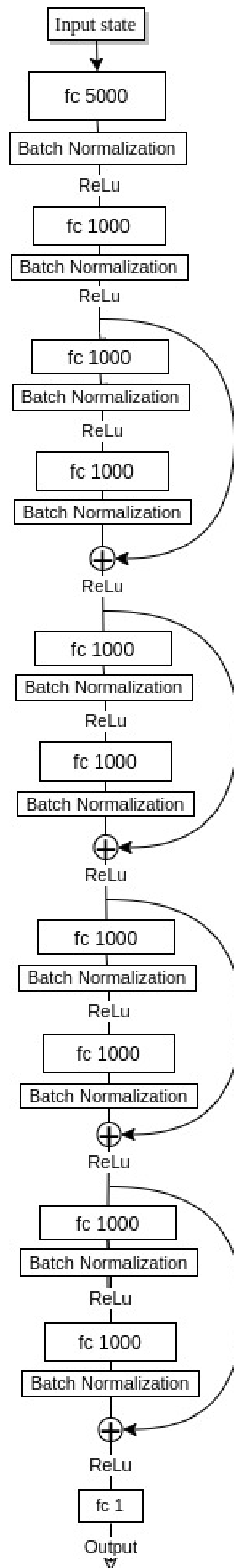


Figure 6: DNN architecture, where fc is a fully connected layer.

Let $J_k(s) := -v_k(s)$ which is the learned cost-to-go at iteration k. By utilizing properties of min and max functions, the value iteration update is as follows:

$$J_{k+1}(s) = -v_{k+1}(s) = -\max_{a\in\mathcal{A}(s)} \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right] = \min_{a\in\mathcal{A}(s)} \sum_{s',r} p(s', r|s, a) \left[-r - \gamma v_k(s')\right]$$
$$= \min_{a\in\mathcal{A}(s)} \sum_{s',r} p(s', r|s, a) \left[-r + \gamma J_k(s')\right] \overset{(*)}{=} \min_{a\in\mathcal{A}(s)} \left[-R(s, a) + J_k(S(s, a))\right]$$

where in $(*)$ it has been taken into account that our main focus is on deterministic problems, so selecting action a from state s takes us to state $S(s, a)$ with a reward of $R(s, a)$. Our objective is to reach the goal state applying as few actions as possible, so one option would be to give the agent constant negative rewards (punishments) until it finds the goal state, as we did in the one-dimensional world example 2.3.4. However, in practice we will operate with positive constant rewards, redefining $R(s, a) := -R(s, a)$ to cancel out the negative sign in the above equation. The value iteration update formula is as follows:

$$J_{k+1}(s) = \min_{a\in\mathcal{A}(s)} \left[R(s, a) + J_k(S(s, a))\right] \tag{3.1}$$

In fact, because we wish to minimize the number of actions taken to reach the goal state, $R(s, a)$ is going to be a constant number, generally chosen to be 1. Also, the discount rate is $\gamma = 1$ because we want to give exactly the same importance to all the actions of a path. In addition, if s is the goal state, we have to set $J_k(s)$ to 0, since every path ends there. Also, from the goal state the set of actions that can be chosen is empty, and the min function is undefined over the empty set. In that case, the formula is changed to $J_{k+1}(s) = 0$.

Nonetheless, there is still a complication about the value iteration approach. The state space of the problem we are solving is very large, so a table based dynamic programming solution to store the learned cost-to-go is not viable. Thus, we have to adopt a neural network approach to reduce the information stored. What we want is to approximate the value iteration update formula 3.1 with the deep neural network described in the previous section. The resulting algorithm is called **DAVI**, Deep Approximate Value Iteration, and it works as follows. We start with `model` and `model_e`, two copies of the same neural network model. `model_e` approximates J_k, and it is used to compute the right side of the value iteration update formula 3.1, while `model` is trained to approximate J_{k+1}. First, a random set of states is generated, and the right side of equation 3.1 is computed (employing `model_e` to evaluate $J_k(S(s, a))$). This is a neural network evaluation, so it takes a lot of computational time, but it can be parallelizable across multiple CPUs (in DeepCubeA's original paper, they use GPUs). Then, the right side of the value iteration formula is used to train the `model` neural network to approximate J_{k+1}. This process of generating states, evaluating and training is repeated until the mean squared error between the right and left hand sides of the formula is beyond a certain threshold ε (we check this condition every C iterations to make the algorithm more stable). When this occurs, it means the approximation of J_{k+1} given by `model` is good enough, and `model_e` parameters are updated with the learned parameters from `model`. This is called a **convergence point**. After it, we continue the previous process, but the effect of updating the `model_e` parameters is that we are now using the approximation of J_{k+1} to evaluate the right hand side, so it means we are now learning J_{k+2}. This algorithm finishes when reaching a maximum number of iterations M, but we will test the model at different convergence points to choose the best.

The first thing we notice is that information is propagated from the goal state to all the other states. Before the first convergence point, the only information we have is that, if s is the goal state, $J_k(s) = 0$ and $J_{k+1}(s) = 0$, so the update formula 3.1 makes `model` approximate the true cost-to-go for states that are only one action away from the goal state. After the first convergence point, `model_e` parameters are updated with the learned parameters from `model`, so we are effectively using that

learned knowledge to train `model`, and it begins to learn the cost-to-go of states that are two actions away from the goal state. In general, after convergence point k, we are learning the cost-to-go of states at distance $k+1$ from the goal, and we are improving the approximation of states at distances less than $k+1$ (since the information in `model_e` is just an approximation). Because of this, it is very important to generate states from a state distribution that allows the reward signal of the goal state to propagate. It has to assign more probability to states that are close to the goal state. I will explore some alternatives for this state distribution in Section 4.5.

The consequence of learning the true cost of states at distance $k+1$ only after convergence point k is that, in problems with longer optimal solutions, we will possibly have to run the algorithm for a longer period of time if we want to learn the true cost of all states. That is why, as stated in Section 3.1.2, the Towers of Hanoi presents a challenge for the algorithm because optimal solutions are very long compared to the space-state size.

The DAVI algorithm uses two models. There are mainly two reasons why the same model is not used for training and evaluation. The first motive is the same as the reason why there is a parameter C that controls the frequency with which the convergence is checked. If we used only one model, the algorithm would be more unstable because the errors in the approximation of J_{k+1} would be directly converted into errors in the evaluation of the right hand side of the formula, without leaving time for the network to rectify those errors. The second reason is that we want our cost-to-go function to correctly order the states according to their distances from the goal state, even if the real cost-to-go function is not correctly predicted. With only one model, the approximation errors would spread more rapidly to some states than to others, thus ruining the order of the cost-to-go function.

3.4 BWAS

Now, our purpose is to search for a path between an initial state and the goal state. As previously mentioned in the Rubik's Cube Section 3.1.1, a simple way of achieving this would be to implement a **breadth-first search** (that only considers the information gathered during the search). This guarantees the path obtained is optimal, but it is only feasible in problems with a small number of states (like the 2x2x2 cube). This is because of the high memory usage due to the need for storing open but not yet expanded nodes.

As we have a neural network trained to predict the distance to the goal state, we could use the neural network as a search heuristic. Therefore, a solution that possibly requires low memory capacity is to start from the initial state and, at each step, only expand the node with the lowest heuristic value from the opened nodes set. This is called **greedy search**, and it runs until we reach the goal state. However, greedy search is not guaranteed to return an optimal path. Thus, we need a way of taking into account both the information gathered during the search and the information stored by the neural network. One way of achieving this is to execute an ***A** search** [28]. It starts from the initial state, and expands the state with the lowest $f(n) = g(n) + h(n)$ score (called f-score), where $h(n)$ is the heuristic value of state n, and $g(n)$ is the g-score, the exact cost of the path (generated by A^*) from the starting state to state n. If the heuristic h is admissible, that is, if it never overestimates the true cost of reaching the goal, A^* returns an optimal path. If it is consistent, that is, if the heuristic value of node N is no greater than the step cost of getting to a neighbor of N plus the heuristic of that neighbor, and the heuristic of the goal state is 0, then the heuristic is admissible, and A^* does not revisit any node. Nonetheless, our heuristic is not guaranteed to be admissible, so it will return non-optimal paths in some cases. In addition, A^* may expand a significant number of states. Because of that, we could insert a weighting factor $\lambda \in [0, 1]$, and expand the state with the lowest weighted f-score, that is, $f(n) = \lambda g(n) + h(n)$. This is called **WAS** (Weighted A^* Search), and the weighting factor λ balances the advantages of greedy search and A^* search. A value of λ close to 0 makes WAS behave almost like a greedy search, achieving potentially longer solutions. A value of λ close to 1 makes WAS behave like A^* Search, with optimal or close to optimal solutions, but probably with

higher memory usage.

Now, another problem arises. Expanding a node involves computing the weighted f-score of all of its children. While the number of possible actions is bounded (12 in the case of a 3x3x3 Rubik's Cube), this process has to be done for every state that we expand until we find the goal state. Our heuristic is the output of a neural network, so computing it is very expensive. However, it is highly parallelizable across multiple computing devices. Because of this, the N states with the lowest weighted f-scores can be expanded, and the heuristic of their children can be computed in parallel. This new algorithm is **BWAS** (Batch Weighted A^* Search), and it is the final method we are going to use to find optimal or near to optimal paths.

4 Implementation Decisions

Now, I will proceed to implement, test and refine my own version of the DeepCubeA algorithm. I will start with the Rubik's Cube representation, in order to have a puzzle to test our algorithm with.

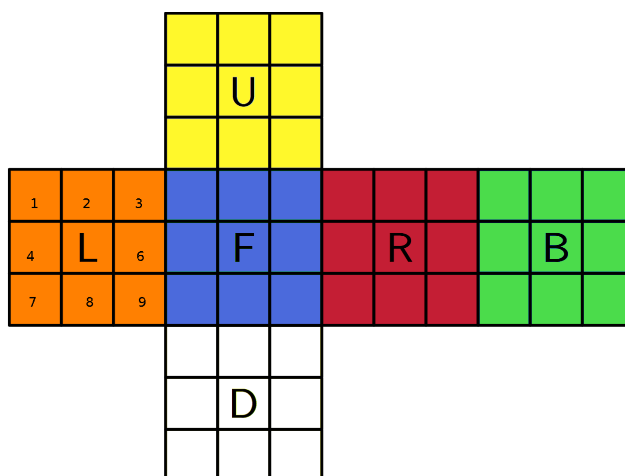
4.1 Rubik's 3x3x3 Cube Representation

The papers of DeepCube [1] and DeepCubeA [2] suggest two different ways of representing the Rubik's 3x3x3 Cube:

- In DeepCube they try to take advantage of specific characteristics of the 3x3x3 cube, so the representation obtained is less generalizable to other problems. Each cubelet is univocally determined by the stickers it contains, and the position of one of those stickers determines the position of the remaining stickers of that cubelet. Moreover, centers are always fixed, so we can focus on the eight corners and the twelve edges, that is, 20 cubelets. For each cubelet, we need to one-hot encode[5] each of the 24 different positions (in the case of edges, 12 locations and 2 possible orientations, while the corners have 8 possible locations and 3 orientations). Therefore, the state has a dimension of $20 * 24 = 480$.
- In DeepCubeA the color of each sticker is represented with a one-hot encoding. For example, the 3x3x3 cube has 6 colors and $6*9$ stickers so the representation has a size of $(6*9)*6 = 324$. This representation is easily generalizable to other problems because, for example in the $H(4,3)$ puzzle, for each disk we can represent the tower in which it is located as a one-hot encoding. As there are 4 disks, and 3 towers, the state has a dimension of $4*3 = 12$.

In this work we will use a one-hot encoding for each sticker, as proposed in the DeepCubeA paper, due to the smaller size of the state representation and the ease of generalization to other problems.

Our front (F) face is going to be the one with the blue center and the upper (U) face the one with the yellow center (with a standard color layout, the remaining faces have their color univocally identified), as showed in the right figure.



The input of the DeepCube's and DeepCubeA's neural networks is a flat array, so we are going to need to flatten the cube in order to represent the state. The process is as follows. The order is Left, Up, Front, Down, Right and Back; we will first represent the Left face, then the Upper face, etc. For each face, we start with the top-left sticker and we move to the bottom-right sticker, going from top to bottom, and from left to right (the order is showed in the figure). As previously mentioned, the color of each sticker is represented in one-hot, for the 3x3x3 cube. There are 6 colors, and in the goal state each color is univocally assigned to a face, so we can use the same order of the faces (L, U, F, D, R, B) to numerate the colors. This numeration is the one we are going to use to create the one-hot representation, so for example blue is in the front face, the third face, so its representation is [0, 0, 1, 0, 0, 0]. Because the 3x3x3 cube has its centers fixed, the first center is always going to be represented as [1, 0, 0, 0, 0, 0, 0] (orange), the second is [0, 1, 0, 0, 0, 0, 0] (yellow), etc. The final representation of the state is the concatenation of all the one-hot sticker representations, in the order

[5] If we have an integer i, $0 \le i < T$ for some T, the one-hot encoding of i is a group of T bits, where the i-th bit is 1 and the rest bits are 0

specified before. In fact, in the general case, a NxNxN Rubik's Cube has 6 colors and 6 faces with $N * N$ stickers on each face, so it can be represented in the same way, with a state size of $6^2 * N^2$.

To run the DAVI algorithm, it will be essential to generate a large number of states as in the paper it is fed with 10,000 million cubes. Because of that, the implementation of the operations for creating a cube in the goal state and the application of an action over a state must be efficient in terms of computational time and memory.

Generating the goal state: As will be seen in Section 4.5, the states fed to the DAVI algorithm are generated from the final state (that is, the state in which all stickers on each face are the same color). It is therefore necessary to have a way to generate the state that represents the goal state in an efficient way. I have studied three alternatives: generate it with Python's list comprehensions, with NumPy's matrix operations, or from a copy of a NumPy array that contains the (already generated) final state.

```
def _solved1():
    return [1 if i==j else 0 for (j, k, i) in it.product(range(6), range(9), range(6))]

def _solved_one_hot():
    identities = np.tile(np.eye(6,dtype=int),9)
    return identities.reshape((6*9*6))

_state_copy = _solved_one_hot()
def _solved_copy(cube_rep=_state_copy):
    return cube_rep.copy()
```

The execution time of the three functions was measured with a million calls. `_solved1()` took 20.7979 seconds, `_solved_one_hot()` 8.8266 seconds, and `_solved_copy()` 0.3389 seconds. Thus, the generation of a new solved state will be implemented as a copy of an array containing the representation of the final state, since an improvement by a factor of x26 over the second fastest way is achieved.

4.2 Rubik's 2x2x2 Cube

In order to test and refine our algorithm, we need a manageable problem. As we have seen in Section 3.1.1, the Rubik's 3x3x3 Cube has a number of states of the order of 4.3×10^{19}. This huge number of states makes it hard for DAVI to converge because the network has to be able to distinguish many different states from each other. Moreover, if we want to build an optimal solver to test our algorithm, any simple solution (such as BFS) is unfeasible. Thus, we will first implement the DeepCube algorithm for the Rubik's 2x2x2 Cube. The 2x2x2 cube can be optimally solved with a simple breadth-first search in a laptop. In fact, we can store a dictionary that maps each state with its distance to the goal-state, and that is precisely how we are going to build an efficient optimal solver. This cube is not mentioned in the original papers (DeepCube [1], DeepCubeA [2]), so we will face some issues for which I will propose some solutions.

For the state representation we will use the same technique as in the 3x3x3 case. The color of each sticker in a face is represented in a one-hot encoding. There are 6 faces, and thus, 6 different colors. In each face there are 2x2 stickers, so this results in a state representation of size $6 * 4 * 6 = 144$.

As in the 3x3x3 cube, there are 12 possible actions in the 2x2x2 one: six clockwise 90º turns: F (front face clockwise turn), B (back face turn), L (left face turn), R (right face turn), U (upper face turn), D (down face turn), and the counter-clockwise turns: F', B', L', R', U', D'. But, if we tried to follow exactly the same approach as in the 3x3x3 cube we would run into a problem. In the 3x3x3 cube, centers are fixed, that is, actions don't change the position of the six centers. This guarantees us that, after applying a list of actions, we don't end up with a rotated version of the goal state because the centers are always in the same position. That is very important because we need to check whether or not two states are equal in an efficient way, so we don't want to have to rotate the cube multiple times for each comparison. In the 2x2x2 cube, things are different because there are no fixed centers

so if, for example, we apply the actions R, L' to the solved cube, we get again the solved cube, but rotated. This issue does not only complicate things when comparing states, but also when training

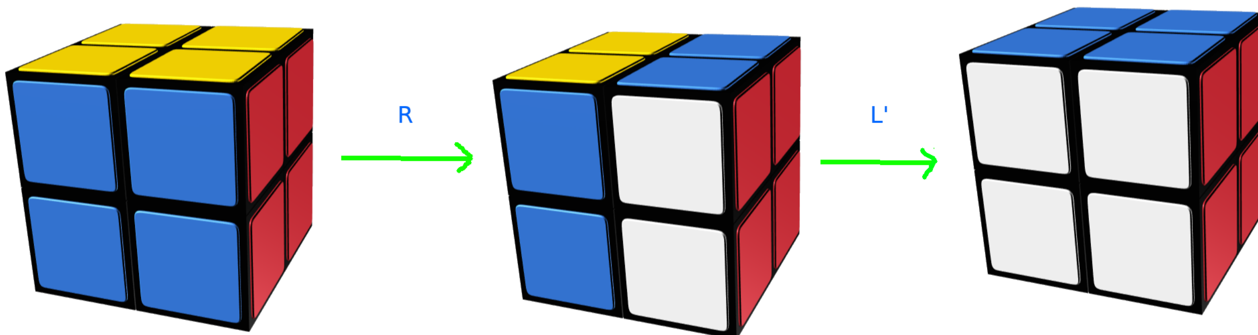


Figure 7: Actions R L'. We get the solved cube, but rotated.

the algorithm. If the goal state (left cube in figure 7) and the rotated goal state (right cube in figure 7) do not have exactly the same representation, the neural network will think they are different states and it will have to distinguish a larger number of states than necessary. There are multiple ways of solving this problem:

- We do not solve it, that is, we consider that a rotation of a state is a different state. This is the simplest option, but we will not consider this solution because of the reasons above.
- Design a state representation that is independent of rotations.
- To fix a corner when applying actions. If a corner is fixed, for example the white-blue-orange corner, no action can rotate the cube.

We will study and implement the third option, because of its simplicity an efficacy. But, how do we fix a corner? We will fix the white-blue-orange corner, the bottom left corner in the front face of the left cube in figure 7 (the procedure is analogous for other corners). There are six actions that affect that corner: L, L', D, D', F, F', and the other six actions preserve the position of the corner: R, R', U, U', B, B'. The trick here is that, for every action that changes the position of the corner, there is an equivalent action that does preserve the position. For example, if we apply L to the solved cube, we end up in the same state as if we would have applied R (except for a rotation). Actually, for every clockwise (counter-clockwise) action over a face, if we turn the opposite face clockwise (counter-clockwise), we get the same state. The complete list of equivalent pairs of actions is: (F, B), (F', B'), (L, R), (L', R'), (D, U) and (D', U'). With this in mind, solving the issue in figure 7 is very easy: we will only work with the six actions that do not affect the white-blue-orange corner: R, R', U, U', B, B', because the other six actions are actually equivalent to the selected ones. In fact, this solution has the advantage that we have reduced the number of actions to apply, so training the algorithm and predicting the sequence of moves to solve the cube are both going to be more efficient.

To efficiently implement BWAS, we will need a **hash function** to use states as keys of dictionaries. Because numpy arrays are easily represented as strings using the Python's built-in function `str`, we could take advantage of the fact that Python strings are hashable to build a simple way of hashing states. But because Python's dictionaries store a list of the keys and our states are one-hot encoded, we can actually achieve a better solution in terms of memory usage. First, the one-hot encoding of the color of each sticker is transformed to an integer representation using an argmax operator, to lower the memory usage. Second, the resulting array is transformed to a bytes representation that allows us to easily recover the state we are representing. The resulting operation is valid for both the 3x3x3 and 2x2x2 cubes, and it is:

```
np.argmax(state.reshape(-1,6), axis=1).tobytes()
```

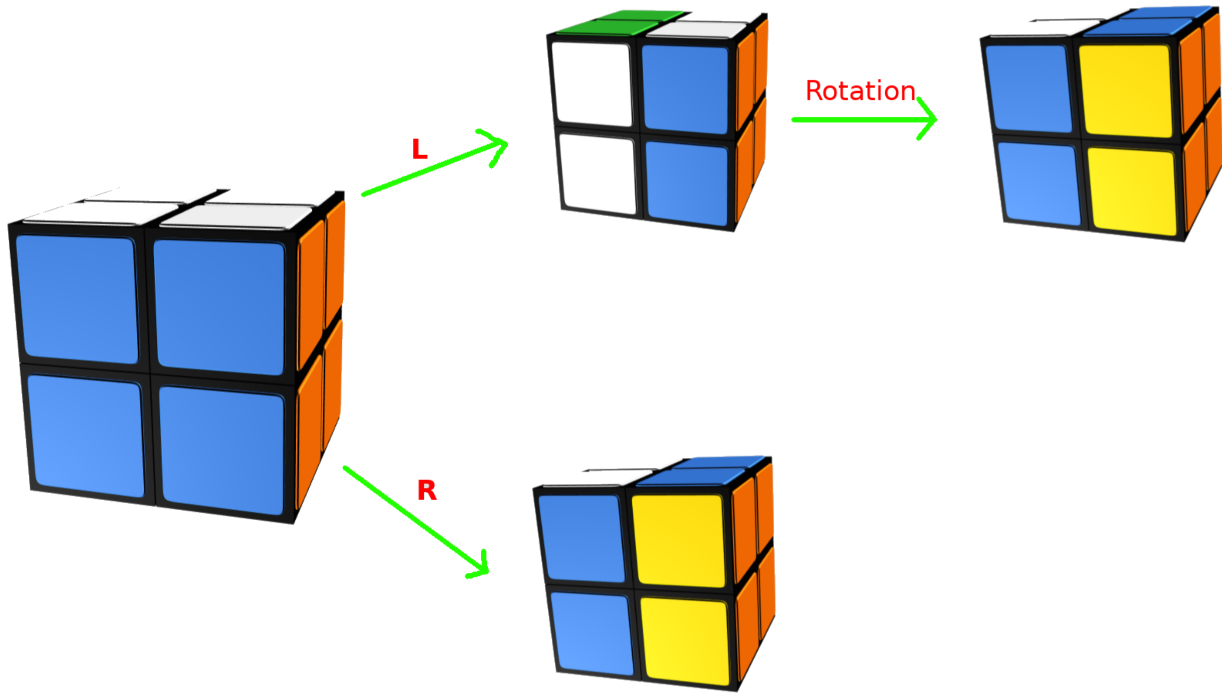


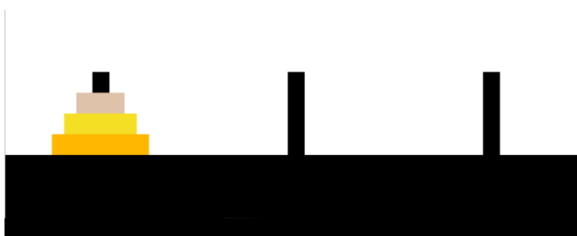
Figure 8: Action R produces the same state as L, except for a rotation of the entire cube.

For example, the 3x3x3 cube converted directly to an string with the Python's built-in function `str` would have a size of 706 bytes, but applying the above operation results in only 465 bytes. This allows the optimal solver in Section 4.4 to run in a computer with less than 16GB of RAM for the 2x2x2 Rubik's Cube. The operation can be inverted to retrieve the state as a numpy array as follows:

```
tf.one_hot(np.frombuffer(bytes_hash, 'int64'), 6,dtype='int64').numpy().reshape(-1)
```

4.3 Towers of Hanoi

To represent a state of a Towers of Hanoi $H(n, m)$, for each disk we are going to represent the tower in which it is located as a one-hot encoding. There are n disks, and for each disk we need m integers to represent their respective towers, so the size of the state representation is $n * m$. In the vector that represents the state, the order of disks is always the same: we begin with the one-hot encoding of the smallest disk, and we end with the biggest one. In fact, if $n' < n$, this representation allows us to easily map states of $H(n', m)$ disks into states of $H(n, m)$ by setting the tower of the last $n - n'$ disks to the one-hot representation of the target tower. If we have an algorithm to solve every state of $Hanoi(n, m)$ (not just the initial one), this mapping allows us to solve $Hanoi(n', m)$ by reutilizing the same algorithm.



Initial state for $H(3, 3)$.

$H(3, 3)$ initial state mapped to a $H(4, 3)$ state.

The action to move disk d from the top of the tower i and placing it on tower j is represented as the tuple (d, i, j). In fact, one could argue that only i and j are necessary to represent the action, since

we always move the top disk of a post and place it on top of another one. But, because of the way that we represent states, if we carry the information of which disk we are moving, executing an action means only changing positions $d * m + i$ and $d * m + j$ of the array. For example, assume that we have $Hanoi(2, 3)$, 2 disks and 3 towers. The smaller disk is on tower 1, so it is represented as [0, 1, 0] and the bigger disk on tower 0, so it is [1, 0, 0], thus the state representation is [0, 1, 0, 1, 0, 0]. If we wanted to move the smaller disk, $d = 0$, to the third tower, $j = 2$, that would be action $(0, 1, 2)$, we would only have to change positions $d * m + i = 1$ (to put a 0, which means the disk has left tower 1), and $d * m + j = 2$ (to put a 1, meaning that disk 0 is now on tower 2).

The way of hashing the Towers of Hanoi states is analogous to the Rubik's Cube method.

4.4 Optimal Distances

For problems that have few states, like the 2x2x2 cube or Hanoi with a small number of disks, we can build an optimal solver just by calculating the true distances to the goal for every state of the problem. Without taking into account any of the particular characteristics of the problem, it can be done by performing the following breadth-first search. The optimal distances are stored in the dictionary `optimal`, hence the need for having a way of hashing states. Starting from the goal, thanks to the use of a FIFO queue, states are explored ordered by their distances to the goal state. For each state s that we expand, with a distance of *cost* to the goal state, if any of its neighbors s' (the states that can be reached by applying a single action from s) has not been previously visited, it is added to the queue to expand it later. Because it has not been previously visited, an optimal path from s' to the goal state is to optimally go from the goal to s, and then from s to s' with an extra action, so we can update the dictionary `optimal` with its optimal distance, which is $cost + 1$.

```
def optimal(puzz, file_name):
    s = puzz.goal_state()
    optimal = {puzz.hash_state(s):0}
    q = deque()
    q.append((s,0))
    while len(q) > 0:
        s, cost = q.popleft()
        for a in puzz.valid_actions(s):
            next_s = puzz.action(s, a)
            hash_s = puzz.hash_state(next_s)
            if hash_s not in optimal:
                optimal[hash_s] = cost + 1
                q.append((next_s, cost+1))

    save_optimal(file_name, optimal)
    return optimal
```

After calculating the optimal costs for all the states, the dict is saved in a file, to trade memory usage for future constant time lookups of the calculated distances.

After computing the optimal values for the **2x2x2 cube**, we can extract some useful information. For example, the number of states at each distance from the goal state:

Distance to goal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# States	1	6	27	120	534	2256	8969	33058	114149	360508	930588	1350852	782536	90280	276

As we can see, the number of states at distances 10, 11 and 12 is very large. Because of that, the training of the models will probably present some difficulties when reaching convergence point 9 or above. Also, there are only 276 states at a maximum distance to the goal state, so, for example, the probability of sampling one of those states from a uniform distribution will be very low.

The results for **Hanoi**$(3, 3)$ are a little bit different. The classes are better balanced, with the states at the maximum distance being the most numerous:

Distance to goal	0	1	2	3	4	5	6	7
# States	1	2	2	4	2	4	4	8

For **Hanoi**$(4,3)$, the first 8 classes have the same number of states as in $Hanoi(3,3)$. Also, the biggest class, with 16 states, is the one with states at a distance 15, the maximum distance for this problem. This fact, in combination with the long distance from the initial to the goal state, is going to present a challenge for the DAVI algorithm when running it to learn a Hanoi problem with a high number of disks.

Distance to goal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
# States	1	2	2	4	2	4	4	8	2	4	4	8	4	8	8	16

4.5 Getting Scrambled States

To feed the DAVI algorithm, we need to get a set of states to train the neural network. The papers DeepCube [1] and DeepCubeA [2] suggest two different ways of obtaining those states.

- In DeepCube, they generate the states starting with the solved cube and, by applying K random actions to it, a sequence of K states is obtained. This procedure is applied l times to generate a training sample with $N = K * l$ cubes.

```
def get_scrambled_deepcube(l, K, puzzle):
    res = []
    for i in range(l):
        s = puzzle.goal_state()
        for j in range(K):
            s = puzzle.action(s, random.choice(puzzle.valid_actions(s)))
            res.append(s)
    return res
```

- In DeepCubeA, each training state x_i is obtained by applying k_i random actions to the solved cube (keeping the last obtained state), where k_i is obtained from a discrete uniform distribution over the interval $[1, ..., K]$, where K is fixed (the original authors chose $K = 30$ for the 3x3x3 cube).

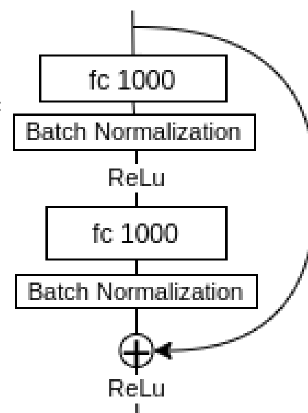
```
def get_scrambled_deepcubea(B, K, puzzle):
    res = []
    for i in range(B):
        ki = random.randint(1,K)
        s = puzzle.goal_state()
        for j in range(ki):
            s = puzzle.action(s, random.choice(puzzle.valid_actions(s)))
        res.append(s)
    return res
```

The distributions of states generated by these two functions change depending on the choice of the parameter K. With the objective of learning the simplest instances of the problem first, this parameter will be chosen accordingly. For further explanations of this distribution, see Section 5.2.

4.6 Neural Network Implementation: Original and Modifications

To implement the neural network, we are going to use the Keras framework because it provides a default implementation for all the types of layers we are going to use: Dense (fully connected) layers of a given size, a BatchNormalization layer that performs the batch normalization operation, and an Add layer that adds the output of two different layers. The dense layer also provides an argument to specify the activation function, but we are interested in applying the activation function after the batch normalization, so we have to use a ReLU layer, which is a non-trainable layer.

```python
def residual_block(x_input, layers_resblock, resblock_layer_size, batch_norm):
    x = x_input
    for i in range(layers_resblock - 1):
        x = layers.Dense(resblock_layer_size)(x)
        if batch_norm:
            x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)
    x = layers.Dense(resblock_layer_size)(x)
    if batch_norm:
        x = layers.BatchNormalization()(x)
    x = layers.Add()([x_input, x])
    x = layers.ReLU()(x)
    return x
```



Residual block of 2 fully connected layers with batch normalization.

Notice that one of the hyperparameters controls the number of fully connected layers the residual block (also called resblock) has, and another, called `batch_norm`, is a boolean that manages whether or not the batch normalization layer is included. This is going to be very useful when creating different versions of the model.

The function that creates the entire model uses the function above to create residual blocks, and in addition to the hyperparameters of the `residual_block` function, it has `input_shape` (that specifies the size of the state representation of the puzzle we are solving), `first_layers_neurons`, which controls the number of neurons of the first layers (the fully connected layers that are previous to the residual blocks), and `n_resblocks`, the number of residual blocks of the network. Notice that the fully connected layers of the residual blocks are going to have the same number of neurons as the last of the fully connected layers (the layer before the residual blocks in Figure 6), because we need to perform the add operation, so the residual connection has to be the same size as the output of the layer inside the residual block.

```python
def _nn_model(name, input_shape=324, first_layers_neurons=(5000,1000),
              n_resblocks=4, layers_resblock=2, batch_norm=True):
    inputs = keras.Input(shape=input_shape)
    x = inputs
    for n in first_layers_neurons:
        x = layers.Dense(n)(x)
        if batch_norm:
            x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)

    for i in range(n_resblocks):
        x = residual_block(x, layers_resblock, first_layers_neurons[-1], batch_norm)

    output = layers.Dense(1)(x)

    model = keras.Model(inputs, output, name=name)
    return model
```

Now, I am not only going to build the DeepCubeA model (figure 6), but also three other models. We are going to call the original DeepCubeA model `DNN_0` (13,763,001 trainable parameters for the 2x2x2 cube), and it is the model obtained when calling the above function with default parameters. The second model is `DNN_noBatchNorm` (13,735,001 trainable parameters), and it is exactly the same model as `DNN_0`, but without batch normalization. The objective with this model is testing how well the batch normalization performs, or if we are going to have to remove it. The third model is a reduced version called `DNN_reduced_0` (6,698,601 trainable parameters), with only 3000 and 800

neurons in the first two fully-connected layers (the fully connected layers that are previous to the residual blocks), and 3 residual blocks instead of 4, with 800 neurons in each of the dense layers of the residual blocks. The purpose of this model is to test how the simplified version of the original model performs in terms of generalization capacity, lightness of the model (less time to perform a prediction, and lower memory usage), and ability to converge to an usable solution when training the DAVI algorithm. I also propose a fourth model, `DNN_reduced_0_noBatchNorm` (6,681,401 trainable parameters), that is exactly like the reduced model but without batch normalization.

4.7 DAVI

Now, we are going to implement the DAVI algorithm described in Section 3.3. The first thing we need is a way of generating batches of data from an array of input states. Because the outputs of the network and the gradients can be computed in different threads for all the states in a batch, dividing the input array in batches effectively improves the performance of training and evaluating our neural networks. Usually, this batch generation is automatically performed by Tensorflow and Keras, the frameworks we are going to use to implement it. But, according to the official documentation, if we implement our own way of generating batches of data subclassing `tf.keras.utils.Sequence`, the utilities of the Keras framework allow us to tune the number of processes that Keras spawns, and to use multiprocessing instead of multiple threads. This is exactly the parallelization we want, explained in Section 3.3.

To divide the array `data` into batches of size `batch_size`, we need to implement two methods: `__len__()` and `__getitem__(i)`. The former has to return the number of batches in our array and the latter returns the i-th batch. The first thing we have to notice is that the number of elements in `data` might not be divisible by `batch_size`. In that case, the last batch would have less than `batch_size` elements. Returning the i-th batch is very straightforward because a batch is a slice of the `data` array of `batch_size` elements, so the i-th batch starts at position $i *$ `batch_size` and ends at position $(i+1) *$ `batch_size` $- 1$ (or lower if it is the last batch and the length of `data` is not divisible by `batch_size`). To implement `__len__()`, if the length of `data` is divisible by `batch_size`, the number of batches is the exact division of the length of `data` by `batch_size`, but we have to apply the ceil operator to treat the general case.

```
class Dataset(tf.keras.utils.Sequence):
    def __init__(self, data, batch_size):
        self.data = data
        self.batch_size = batch_size

    def __getitem__(self, i):
        batch = self.data[i*self.batch_size:(i+1)*self.batch_size]
        return batch

    def __len__(self):
        return math.ceil(len(self.data) / self.batch_size)
```

From Section 3.3, our objective is to approximate the value iteration update formula:

$$J_{k+1}(s) = \min_a \left[R(s,a) + J_k(S(s,a))\right]$$

The function `next_cost` computes the right hand side of the update formula for an array of states `X`, using the neural network `model_e` to approximate J_k. The intuition here is that, for every state in `X`, we have to compute its neighbors, evaluate J_k by calling the `predict` function of `model_e`, and apply a min operator. But, because the prediction of a neural network is time consuming, this process can be more efficient if we compute all the J_k values in parallel for all the neighbors of every state in `X`. Thus, the first `for` loop stores all the neighbors of every state in an array, then a Dataset is

created to divide that array in batches, and the `predict` function of `model_e` is used to compute J_k in parallel across different processes. Finally, the last `for` loop applies the min operator over all the predicted values of the neighbors. The first `if` inside the `for` loop checks if `state` is the goal state, in which case we have to impose that the min operator returns 0. The second `if` checks if any of the neighbors s' is the goal state, in which case we directly know that $J_k(s') = 0$, and because in practice we are using constant rewards $cost := R(s, a)$, the value iteration update formula tells us that $J_{k+1}(s) = \min_a [R(s, a) + J_k(S(s, a))] = cost$.

```
def next_cost(model_e, X, goal_state, puzzle, batch_size=32, workers=4):
    y = []
    next_states = []
    for state in X:
        next_states.extend([puzzle.action(state,a) for a in puzzle.valid_actions(state)])
    d = Dataset(np.array(next_states), batch_size) # used for multiprocess predicting
    preds = model_e.predict(d, workers=workers, use_multiprocessing=True) + puzzle.cost()
    i = 0
    for state in X:
        num_possible_actions = puzzle.number_actions(state)
        state_next_states = next_states[i:i+num_possible_actions]
        state_preds     = preds[i:i+num_possible_actions]
        i += num_possible_actions
        if np.array_equal(state, goal_state):
            y.append(0)
            continue
        if list(filter(lambda s: np.array_equal(s, goal_state), state_next_states)):
            y.append(puzzle.cost())
            continue
        y.append(min(state_preds)[0])
    return y
```

Now, we have all the ingredients to implement the DAVI algorithm. We start by initializing Adam, the adaptive learning rate method we are using, by setting the learning rate. In this case, we are going to use 0.0002 since a higher value would worsen the stability of convergence. Then, the mean squared error is set up. We are using two neural networks, `model` and `model_e` with the same architecture: we are going to train `model` and we will use `model_e` for predicting and computing the right hand side of the value iteration formula. At the start, `model` is already initialized (either with the default initialization that the framework provides, or with the weights of a past training). Also `model_e` is cloned from `model` to have the same architecture. If `model` was default initialized, that would be sufficient, but it might have the parameters of a past training, so we also have to copy the weights into `model_e`, to take advantage of that information.

```
    optimizer = keras.optimizers.Adam(learning_rate=lr)
    loss_fn = keras.losses.MeanSquaredError()
    model_e = keras.models.clone_model(model) # model_e with parameters theta_e, used to predict
    model_e.set_weights(model.get_weights()) # clone_model does not copy weights
    model.compile(
        optimizer=optimizer, loss=loss_fn, metrics=[], weighted_metrics=None,
        run_eagerly=False)
```

Now, we arrive at the main loop of DAVI. A maximum of M iterations are run, counting from `initial_it` (which is a parameter needed if we want to continue a past training). This loop can be divided in two steps: generating the training and validation data and training the neural network, and extracting the loss from the training history and checking for convergence.

```
for iteration in range(initial_it, initial_it + M):
    # generate data and train the neural network
    # get validation loss to check for convergence, and store model weights
```

To generate the data, first we have to generate scrambled states using one of the methods described in Section 4.5. The parameter `K` controls the maximum number of random actions applied to the

goal state, and B and B_val control the number of times that operation is repeated for generating the training and validation data respectively. Then, the function next_cost is applied to compute the approximations of the right hand side of the value iteration update formula, both for the training and validation states, using model_e. Next, the model neural network is trained to approximate J_{k+1}. This is done by calling its function fit, that performs supervised training using the approximations as labels.

```
# generate data and train the neural network
X = puzzle.get_scrambled(B, K)
X_val = puzzle.get_scrambled(B_val, K)
y = next_cost(model_e, X, goal_state, puzzle)
y_val = next_cost(model_e, X_val, goal_state, puzzle)

history = model.fit(
        x=np.array(X), y=np.array(y),
        batch_size=None, epochs=1,
        verbose=2, callbacks=None,
        initial_epoch=0, steps_per_epoch=None,
        validation_data=(np.array(X_val), np.array(y_val)),
        )
```

Finally, the loss information is retrieved from the history returned by the fit method. Every C iterations, that loss information is stored, and the convergence of the model is tested, which consists in checking if the validation loss has fallen below the threshold ε. If that is the case, we have a new convergence point, and the weights of model_e are updated with the recently learned parameters from model. To continue the training if something fails, the weights are also stored on disk, but only every CC convergence points. CC is a new parameter, not present in the DeepCube or DeepCubeA papers, that is very useful to save disk space when solving problems that generate a lot of convergence points like the Towers of Hanoi. Therefore, this constant CC is set to 1 for the Rubik's Cube puzzles, and to 10 for the Towers of Hanoi.

```
# get validation loss to check for convergence, and store model weights
loss = history.history['loss'][0]
val_loss = history.history['val_loss'][0]
losses["loss"].append(loss)
losses["val_loss"].append(val_loss)
if iteration % C == 0:
    nn.save_losses(model.name, str(puzzle), iteration, save_path, losses)
    if val_loss < eps:
        model_e.set_weights(model.get_weights())
        if (conv_point-1) % CC == 0:
            nn.save_weights(model, str(puzzle), iteration, save_path)
        conv_point += 1
```

On a final note about the data generation step, the validation set is randomly selected at each iteration to avoid a difficulty related to the convergence of the algorithm that a fixed set may cause. This is not supervised learning, we don't have the true values that the final neural network needs to learn, so the validation data checks how close we are to approximate the value iteration update formula. If we used a fixed set, we might choose a very poor one (and we wouldn't have a way to detect that it is not a good validation set). For example, if we are learning cubes at a distance of 12 from the goal state (and we have already learned the ones at lower distances), a validation set with only cubes at a distance of 7 from the goal state would mean that, depending on how well our model approximates the true cost-to-go, we would probably detect a false convergence (if we had already learned those states, the loss would be lower than ε). And, because the validation set is fixed, if we have learned a decent approximation of the cost-to-go for those states, once we detect a false convergence we will re-detect a false convergence next time it is checked. In the original paper, the authors don't mention if they use a fixed or a dynamic validation set. They also do not state how they solve the false convergence problem. To try to solve it, I decided to choose a different validation set at each iteration. The downside is that there might be overlap between the validation

and training sets, but the validation is only checked every C iterations, so this minimizes the probability of a false convergence. The overlap is due to the fact that both the training and validation sets have to be randomly selected from their distributions (in this case, the two distributions are the same, since training and validation sets are generated as in Section 4.5) that assign non-zero probability to each state, because we want states at all possible distances from the goal state (and we don't know in advance the distribution of distances over the state space). Also, to accomplish this, we have to choose a value of `K` higher than the maximum number of actions we need to reach all states.

We randomly select the training set at every iteration, so if we didn't want to have overlap, we would have to store every generated state (not only the ones generated at this iteration, but also all the previous ones) and generate a validation set that does not contain any of the training states. We can compare states in $\mathcal{O}(\text{state_size})$ (for example, by using a Python set), but we would quickly run out of memory if we used this technique. In fact, the overlap between validation and training sets is not a big problem in this particular implementation. We are only using the validation set to compute the *loss* that is compared against ε. This *loss* measures how well the model has learned the values of states taken from the distribution of states generated in Section 4.5 (for further explanations of this distribution, see Section 5.2). If there is overlap, it only means the loss for the states in the intersection is probably low, depending on how well our model has learned the cost-to-go for those states. Although the probability of a false convergence is not zero, it is unlikely to concatenate multiple false convergences in a row because we are continuously sampling our validation set. In the case of a false convergence the model we use to make predictions (`model_e`) is updated faster, but the rate of convergence can be changed by adjusting ε.

4.8 BWAS

To implement BWAS, a priority queue `open_states` is used to store the open states, that is, the ones seen but not yet expanded. This priority queue stores tuples of three elements, sorting those tuples lexicographically in ascending order. We are interested in retrieving the state with the lowest f-score and also, the tuple has to carry the state, so the tuples are of the form (f-score, counter, state), where counter is an increasing integer that breaks any f-score ties, to prevent comparison between states. Also, I store the best g-score found for the open states in a dict. To build the path from the initial state `state_0` to the goal state, we need a way to remember what actions we have to take, so the g-score dict also stores the last action taken to reach any state with the current best g-score.

```
open_states = PriorityQueue()
found = False
# state -> (cost, action) (the action from parent_state to state)
g_score = {puzzle.hash_state(state_0): (0, "")}
# f(x) = lamda*g(x) + h.predict(x) where g(x) is the cost of the path to x and
# h(x) is the heuristic
open_states.put((0 + h.predict(np.array([state_0])), 0, state_0))
# PriorityQueue sorts lexicographically
count = 1 # To break any possible ties.
if np.array_equal(state_0, puzzle.goal_state()):
    found = True
```

After all the initializations we arrive at the main loop, which runs until the goal state is reached. At each iteration, N states are expanded, the heuristics of their children and the new g-score and f-score are used to determine if those children have to be added to `open_states`.

```
while not found and not open_states.empty():
    # Expand states and compute heuristic
    # Add new states to open_states queue
```

First, N states are removed from the priority queue, and if any of them is the final state, the algorithm finishes. Then, as we did in the DAVI algorithm, all of the N states are expanded and the heuristics of the neighbors of those N states are computed in parallel at the same time.

```
# Expand states and compute heuristic
states_to_expand = []
i = 0
while i < N and not found and not open_states.empty(): # BWAS: expand N states instead of 1
    f_score, _, state = open_states.get()
    if np.array_equal(state, puzzle.goal_state()):
        found = True
        break
    states_to_expand.append(state)
    i += 1
if found:
    break
child_states = []
for state in states_to_expand:
    child_states.extend([puzzle.action(state,a) for a in puzzle.valid_actions(state)])
# Heuristics are computed in parallel for all the children of the N nodes
d = Dataset(np.array(child_states), batch_size) # used for multiprocess predicting
heuristics = h.predict(np.array(child_states), workers=n_processes, use_multiprocessing=True)
```

Then, for each of the neighbors of each of the expanded states, the new g-score and f-scores are computed from the g-score of the parent and the previously calculated heuristic value. A child is inserted into the `open_states` if it was not previously visited or if the g-score just computed improves the previous g-score. That guarantees that the f-score is also better than the previous one, since the heuristic value of a state does not change during the BWAS execution.

```
# Add new states to open_states queue
i = 0
for state in states_to_expand:
    actions = puzzle.valid_actions(state)
    num_possible_actions = len(actions)
    new_g_score = g_score[puzzle.hash_state(state)][0] + 1
    for j, a in enumerate(actions): # Each of the N states has len(action) childs
        child = child_states[i + j]
        child_h = heuristics[i + j]
        key = puzzle.hash_state(child)
        if not key in g_score or g_score[key][0] > new_g_score:
            g_score[key] = (new_g_score, a)
            open_states.put((lamda*new_g_score + child_h, count, child))
            count += 1
    i += num_possible_actions
```

Finally, when the loop ends we have to rebuild the solution, that is, the path from the initial state to the goal state. It is built backwards, from the goal to the initial state. The g-score dict stores the action needed to move from the second to last state in the path to the goal state, so taking the inverse action leaves us in that second to last state. Repeating this procedure, we reach the initial state, and the path is just the sequence of actions we have seen but in reverse order.

```
# Build solution
state = puzzle.goal_state()
length, a = g_score[puzzle.hash_state(state)]

actions = []
while a != "":
    actions.append(a)
    state = puzzle.inv_action(state, a)
    _, a = g_score[puzzle.hash_state(state)]
print(f"   Expanded {num_expanded} states")
return length, actions[::-1]
```

5 Results

5.1 Training Models and First Results for the 2x2x2 Rubik's Cube

I have executed the DAVI algorithm to train the four models for the 2x2x2 cube, and the goal now is to test them in order to decide whether or not to continue training. The same parameters of the DAVI algorithm have been used for all the models: $\varepsilon = 0.05$ (controls the error threshold to update the evaluation model), $C = 10$ (how often to check for convergence), $M = 10000$ (max. number of iterations). To generate states, `get_scrambled_deepcubea` is used with $K = 20$ (max. scrambles per state), $B = 1000$ (number of states the algorithm sees per iteration), and finally, $B_val = 200$ (the number of states for validation, used to calculate the loss to check for convergence). The next table shows a summary of the training process, where # model updates is the number of convergence points.

Model	Total Iterations.	# model updates	Model updates iterations
`DNN_0`	2950	10	40, 70, 90, 130, 160, 240, 400, 740, 1200, 2080
`DNN_noBatchNorm`	2850	11	10, 20, 30, 40, 50, 70, 130, 250, 650, 1080, 1790
`DNN_reduced_0`	2830	10	50, 60, 80, 100, 140, 290, 550, 1010, 1610, 2440
`DNN_reduced_0_noBatchNorm`	3910	10	10, 20, 30, 40, 50, 70, 130, 370, 830, 2030

The approximate training times are: 17 hours for `DNN_0`, 16 hours for `DNN_noBatchNorm`, 10 hours for `DNN_reduced_0` and 13 hours for `DNN_reduced_0_noBatchNorm`. The first thing that is worth mentioning is that the models without batch normalization update their evaluation models faster. For example, the tenth update of the `DNN_0` model happens in the iteration 2080, while the eleventh update of `DNN_noBatchNorm` is in the iteration 1790. This means that, with less cubes fed into the algorithm, it is probable that the noBatchNorm version is able to distinguish cubes at distance 10 from cubes at distance 11. Remember from the DAVI explanation that each update of the evaluation model means the model is learning to classify states that are one step further from the goal state. Because of that, at the last convergence point, the `DNN_noBatchNorm` algorithm was learning to predict the cost-to-go of states that are 11 moves from the solved cube, while the other three models were learning to distinguish cubes at distance 9 from cubes at distance 10. In the quarter-turn metric, there are states that are 14 moves from the solved cube, so if we wanted to predict them correctly, we would have to continue training the models. However, running the algorithms for longer has its own issue. Whereas the first updates happen in the first iterations (for example, the original model `DNN_0` updates 8 times before iteration 1000), the later updates are more expensive to achieve (`DNN_0` only updates two more times between iterations 1000 and 2950). In fact, the model `DNN_reduced_0_noBatchNorm` has been trained for 3910 iterations (a thousand more than the rest, because it is the cheapest one, with less trainable parameters), and the last update is the tenth one in the iteration 2030, which is a symptom that it is finding it more difficult to converge.

Let's examine the training and validation losses. In all the loss figures, the horizontal dotted line represents $\varepsilon = 0.05$, the black circles show when the evaluation model updates happened, the cyan color is the validation loss, and the magenta color the training loss. Notice that figure 9 shows that the validation loss spikes up very frequently. In fact, there is a spike between iterations 200 and 250 that raises the validation loss to a mean squared error of over 2. It is due to the fact that, in order to check for convergence, the validation loss changes at each iteration (for further explanations, see section 4.7), so this was expected.

Also, notice the black circles over the horizontal dotted line. Those are the convergence points, where the evaluation model is updated with the parameters of the model being trained. Because of this update, the train and validation losses spike just after the convergence point, as the network now has to learn to assign the correct cost-to-go of states that are one step further from the goal state.

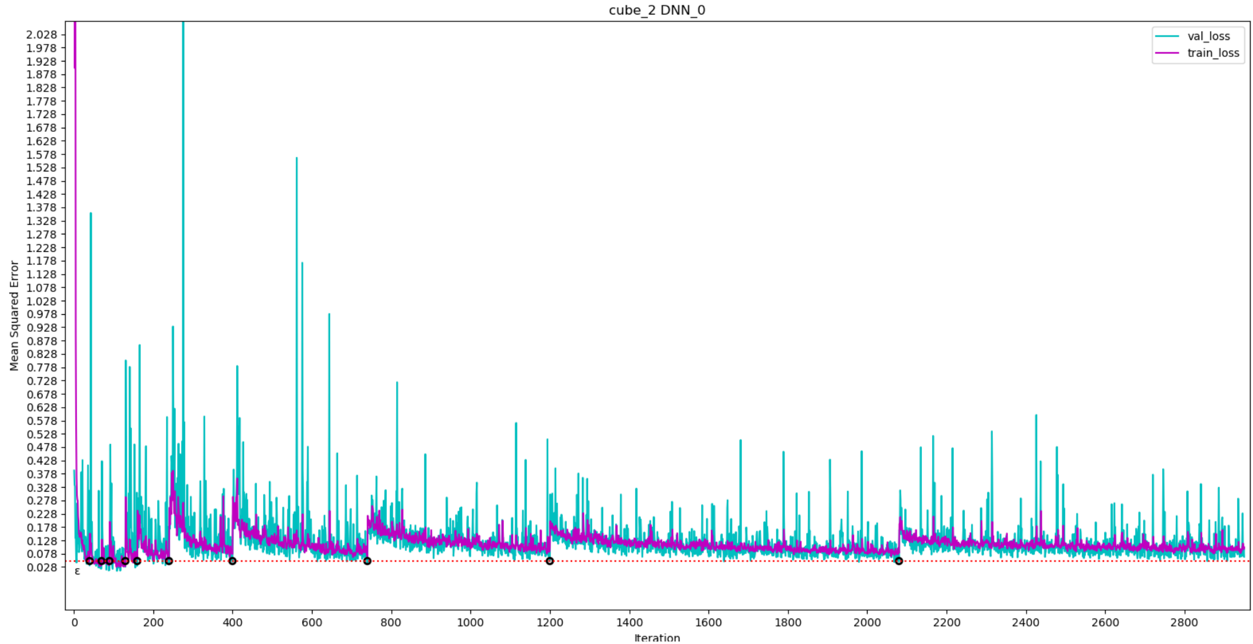


Figure 9: Training (magenta) and validation (blue) losses of DNN_0 model until the iteration 2950.

Another fact that figure 9 reveals, and that we already knew, is that the model updates take more time as the training progresses. Let's examine the last two convergence points of our four models, to see whether they are near their maximum capacity, or if they can still learn.

As we can see in figure 10, models with batch normalization (left) have longer spikes, which indicates that they perform worse over the validation sets than the ones without batch normalization. Moreover, the loss of models without batch normalization (right) falls faster, while the ones with it take more iterations to converge (notice that the loss of `DNN_0` after the last convergence point holds above the red line (epsilon), almost without decreasing). However, any difficulty in converging at this point of the training does not mean they are reaching their maximum learning capacity since they have already updated 10 times (11 in the case of `DNN_noBatchNorm`), so they are learning states at a distance of 11 (12) from the goal state and, according to table 4.4, these distances have 1350852 (782536) states, which is a big percentage of the total number of states.

Finally, I have also measured the performance at the last convergence point of these four models against the optimal solver, taking all states into account. The aim is to discover how these models perform in the task of predicting the correct cost-to-go without yet computing the actual path with BWAS. Thus, the optimal solver is only used to compute the length of the optimal paths. The procedure is to compute the optimal cost-to-go and the model output (predicted cost-to-go) for each state, and compute the mean absolute error between the optimal and the predicted values. The mean absolute error of `DNN_0` at iteration 2080 is `3.4841`, that of `DNN_noBatchNorm` at iteration 1790 is `2.5468`, that of `DNN_reduced_0` at 2440 is `3.0558`, and that of `DNN_reduced_0_noBatchNorm` at 2030, `2.5733`. It is clear (at least for now) that models without batch normalization perform better. Nevertheless, we are also going to continue training models with batch normalization, to test how they perform as a heuristic for the BWAS search. In Figure 11, the mean absolute error per distance of the predicted cost-to-go is shown. From this figure, it is evident that, for the four models, the error increases the farther away we are from the solved cube. As the distances 10, 11 and 12 are the ones with more states, they are the ones which contribute more to the above losses. Also the losses are very high for the distances 12, 13 and 14, but this was expected since we are in the update number 10 (11 in the case of `DNN_noBatchNorm`). In fact, for the distances 12, 13 and 14, the network is learning

they are at a distance 11 (12), so a mean absolute error of more than 1 was expected. But, as DAVI employs the model after the last convergence point to make predictions, the error is accumulated at each convergence point, thus the increasing error in figure 11.

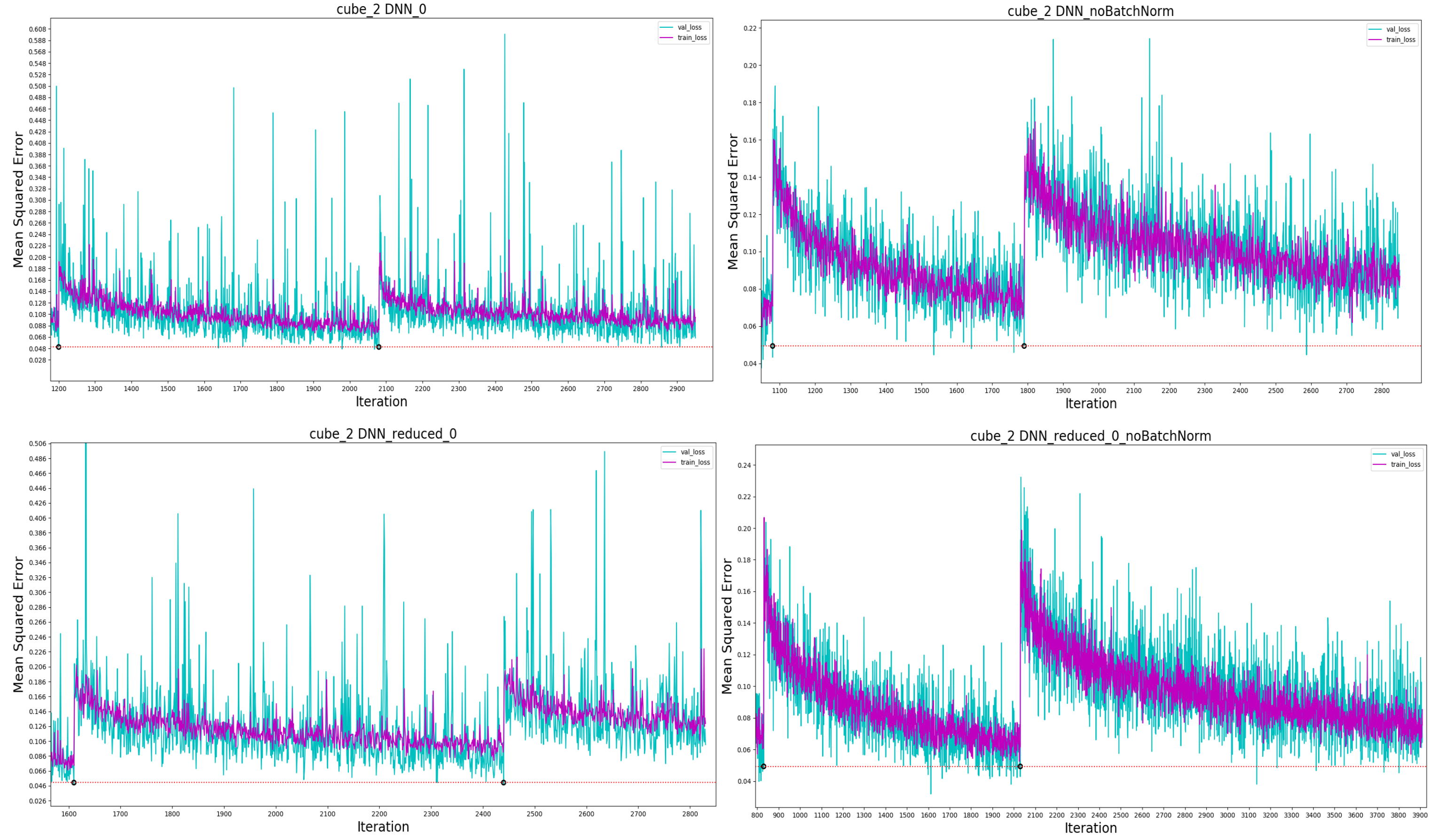


Figure 10: Last two convergence points of the four models (models without batch normalization have been zoomed in).

5.2 State generation distributions

Now, we are going to study the distributions of states taken from the functions in Section 4.5. The goal is to examine the effect of the parameter K in both distributions (which controls the number of random actions applied to the goal state), and compare the DeepCube and DeepCubeA methods to choose one of them for future trainings. Because we have an optimal solver, all the tests in this section have been performed over the 2x2x2 cube. The objective of these functions is to generate states to train the neural network models with the DAVI algorithm, so the states generated by these functions have to allow the reward signal to propagate from the goal state to to the other states. Thus, it has to assign more probability to states that are close to the goal state, in order to learn the simplest instances of the problem first. This is the first obstacle, since in Section 4.4 we already discovered that the real state distributions for the 2x2x2 cube and the Towers of Hanoi for 3 and 4 disks are very unbalanced. In particular, for the 2x2x2 cube the majority of states are at distances 8-13, so a uniform distribution would not be very useful to execute the DAVI algorithm since it would assign a very low probability to states at distances less than 8. On the other hand, the DeepCubeA way of generating states gives us the following distributions for $K=$ 17, 20 and 23.

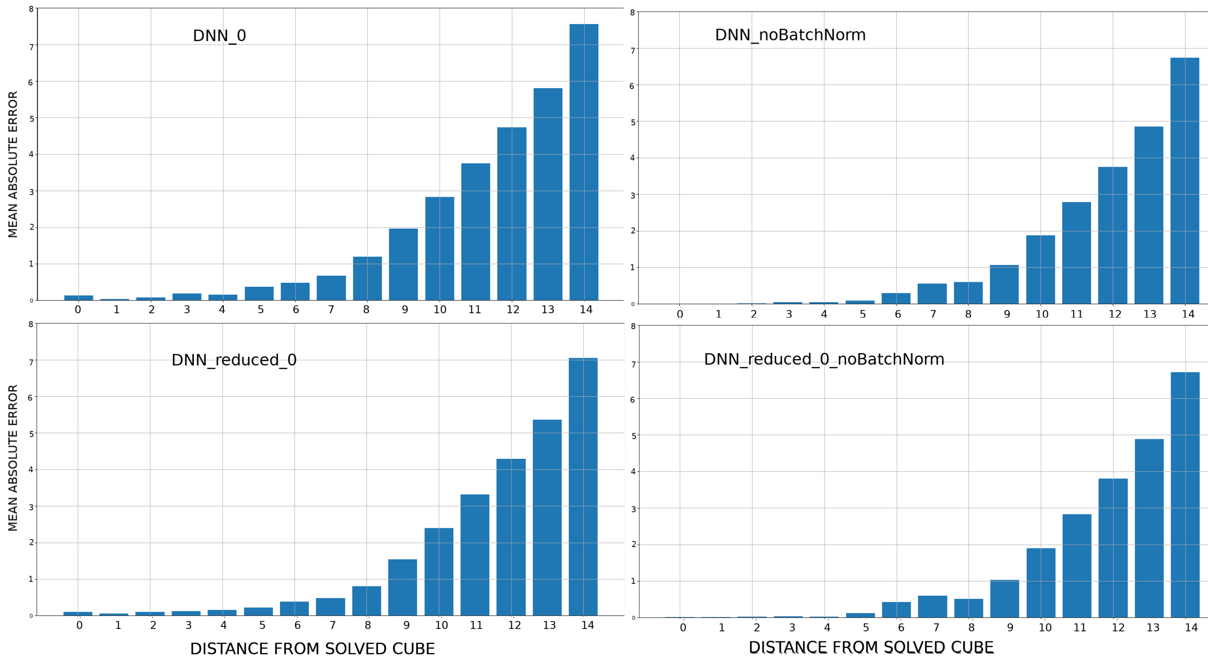


Figure 11: Mean absolute errors per distance from goal state.

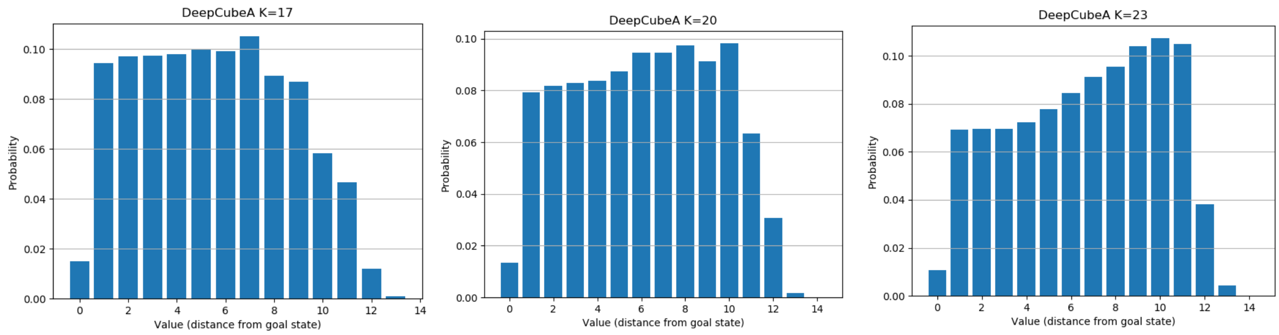


Figure 12: State distribution for `get_scrambled_deepcube` for different values of K.

These figures measure the probability of generating a state at a particular distance. They have been created by generating random states (68,000 states for K=17, 80,000 states for K=20 and 92,000 states for K=23) with the `get_scrambled_deepcubea` function, and then computing the true distances to the goal state by using the optimal solver. The first thing to notice is that, for K= 17, 20 and 23, it assigns relatively high probability to states at distances 1-7, at least compared to the small number these states represent in the real distributions. The effect of K is also shown. While $K = 17$ assigns less than a 9% probability for states at distances 8 and 9, and less than a 6% for states at distances 10 and 11, raising K decreases the probability of getting states at lower distances, and increases the likelihood of reaching states at greater distances. For $K = 23$, there is more than a 10% probability of getting a state at a distance of 9 and more than a 10% probability for states at distances 10, while for each class of states below distance 5, the probability of getting a state is lower than a 7% (for each class). This effect has to be taken into account when selecting the optimal K for a particular problem. Because we have already trained models with 10 convergence points or more for the 2x2x2 cube, and after convergence point d the reward signal has already propagated to states at distances d or lower, it is in our best interest to choose a K value that allows us to learn states at distances 10 or higher while also having medium-high probabilities for states less than 10 in order to improve the approximation of the cost-to-go for those states. Because of that, we will use $K = 20$ for completing the training of the models of the previous section. Now, let's compare these distributions with the ones created by the `get_scrambled_deepcube` function, which generates states in the way described in the DeepCube paper.

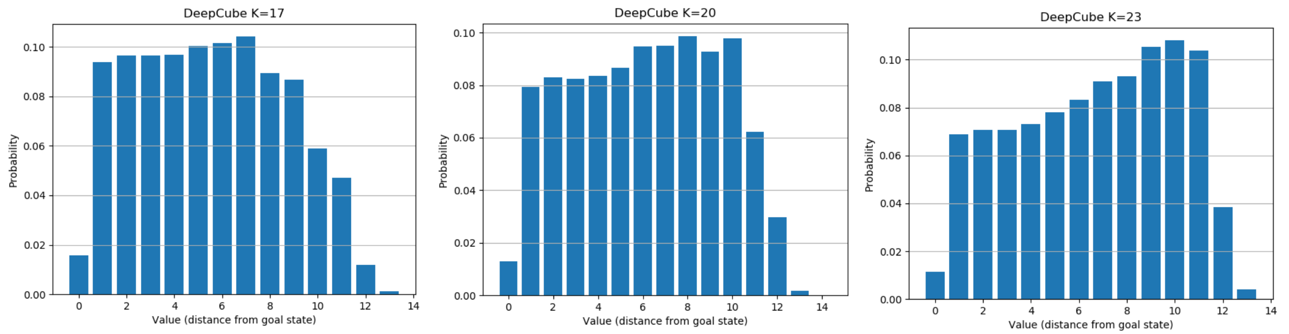


Figure 13: State distribution for `get_scrambled_deepcube` for different values of K.

As can be seen, these distributions look identical to the previous ones. In fact, it's hard to see the differences. To compare them in more detail, we can count the total number of states at each distance generated by the DeepCubeA and DeepCube methods respectively. Then, for each distance d to the goal we subtract the number of states generated by DeepCubeA that are at distance d minus the number of states generated by DeepCube that are at that distance. This process is performed over the same number of states than the previous figures (between 68,000 and 92,000 states). The results are shown in Figure 14 where a height of n for a distance d means that DeepCubeA has retrieved n states more than DeepCube for that particular distance. With this in mind, the difference between these two methods seems very small. For K=17, there is an outlier at distance 6 with a difference of approximately -150 cubes, which only represents the 0.22% of the total number of generated cubes, while the other differences are bounded between -100 and 100. For K=20, all the errors are between -150 and 150, and for K=23 there is an outlier at distance 8 of more than 200 cubes, but it only represents a 0.25% of the total cubes generated for this value of K. Overall, for any of these values of K, the total of absolute differences, taking into account all distances, is below 3%. Although there may be some statistical differences between these two ways of generating states, I have not found any clear pattern.

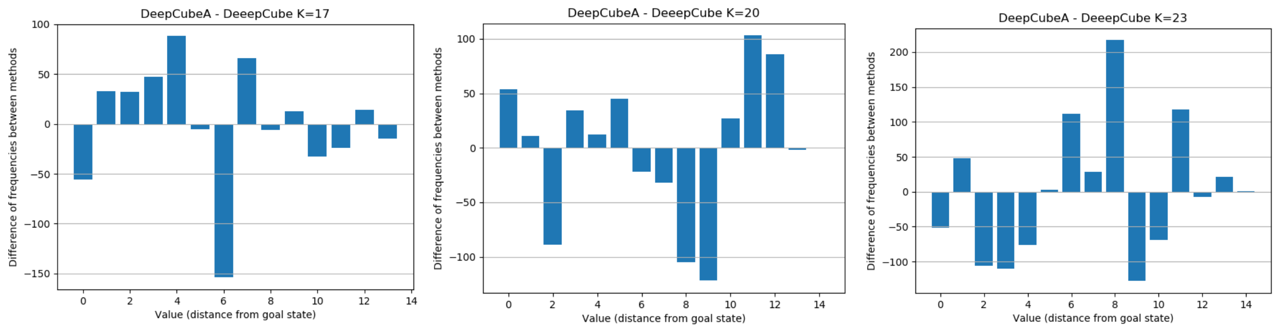


Figure 14: Comparison between the two ways of generating states

With all this in mind, in the next sections we are going to generate states with the `get_scrambled_deepcube` function, which is the one used in the DeepCube algorithm. Although there is a slight statistical difference and the DeepCubeA paper is more recent, there is a benefit in using this function in terms of time complexity. While the two functions run in approximately the same time ($\mathcal{O}(B * K)$ when calling both functions with $l = B$, where l is the parameter of `get_scrambled_deepcube` and B the one of `get_scrambled_deepcubea`), the DeepCube one returns $l * K$ states while the DeepCubeA function returns only B. The impact of this fact in the performance of the training algorithm is magnified when using larger values of K, so we will also choose the `get_scrambled_deepcube` function when training for the 3x3x3 Cube and the Hanoi puzzles, since they need higher values of K.

5.3 Completing the training for the 2x2x2 cube

I have trained the four models of Section 5.1 for longer. In total, DNN_0 has been trained for 60 hours, and the others for approximately 43 hours each. The DNN_noBatchNorm model has been trained until convergence point 14. In theory, because the maximum optimal distance of state is 14 in the 2x2x2 cube, this means that the reward signal has propagated to all possible states. The three other models have been trained until convergence point 13, because, as seen in Section 5.1 the most promising one was DNN_noBatchNorm in terms of the convergence speed and the mean absolute errors. Also, to execute an A^* search, it is enough to distinguish states at a distance of 13 or less. This is because there are very few states at distance 14, and at convergence point number 13, states at distances 13 and 14 are not distinguished by the model, which does not ruin the possibility that the model, used as a heuristic for the search, is admissible. All models have been trained with the same hyperparameters as in the previous training, but using get_scrambled_deepcube for generating states, because of the reasons discussed in the previous section. After the last convergence point, all models have been trained for longer (until the iteration shown in the column named Total Iterations), with the intention to achieve more convergence points, and to study the behaviour of the loss.

Model	Total Iterations	# model updates	Model updates iterations (convergence points)
DNN_0	10200	13	40, 70, 90, 130, 160, 240, 400, 740, 1200 , 2080, 3140, 4050, 7550
DNN_noBatchNorm	7590	14	10, 20, 30, 40, 50, 70, 130, 250, 650, 1080, 1790, 2540, 3290, 6950
DNN_reduced_0	12440	13	50, 60, 80, 100, 140, 290, 550, 1010, 1610, 2440, 3850, 5880, 9900
DNN_reduced_0_noBatchNorm	13660	13	10, 20, 30, 40, 50, 70, 130, 370, 830, 2030, 3660, 6140, 11080

In Figure 15, the last convergence points of the four models are shown. As in Figure 10, the spiking up continues. Also, the loss here seems to be more horizontal, taking more time to converge than in Figure 10. This is due to multiple facts, one of the main ones being that now the models are in later convergence points, so they are required to distinguish more distances. It also contributes to this fact that errors are accumulated, so being at convergence point 13-14 means not only learning more distances than when we were at convergence points 10-11, but also correcting those errors from past iterations. Finally, in Figure 15, it is shown that after the last convergence point (the second black circle over the red dotted line), the loss of the four models continues decreasing, meaning that is very possible that it would generate more convergence points if trained for longer. However, it is not necessary to achieve more convergence points because, as will be seen in Section 5.7, more convergence points does not always mean that we are going to achieve better results when using these models as a heuristic function for the BWAS search.

I have measured the mean absolute error of the output of the last convergence point of these models compared to the true distances, taking all states into account, as previously done in Section 5.1. DNN_0 has a mean absolute error of 2.1626 at iteration 7550, DNN_noBatchNorm has 1.7688 at iteration 6950, DNN_reduced_0 has 1.9219 at 9900, and DNN_reduced_0_noBatchNorm at 11080, 1.6683. All algorithms have reduced their errors compared to the previous measurements. Furthermore, DNN_noBatchNorm converges in fewer iterations than the others, while DNN_reduced_0_noBatchNorm is the one with the smallest errors. In fact, the two reduced models have outperformed their full counterparts, but they have also been trained for more iterations. This is because, except DNN_0, all the models have been trained for approximately the same period of time, but reduced versions are faster when it comes to predicting and training, which means they iterate faster. Additionaly, the mean absolute error per distance in Figure 23 has also been reduced with respect to the one in Section 5.1 for all models.

However, there is an inconvenience with these models. Although they are going to perform well as a heuristic for the BWAS algorithm (see Section 5.7), they are trained by feeding them with 1000 states per iteration. That means DNN_noBatchNorm at convergence point number 13 has seen 3,290,000 states, while the other models have seen more than 7.5 million states at that same convergence point.

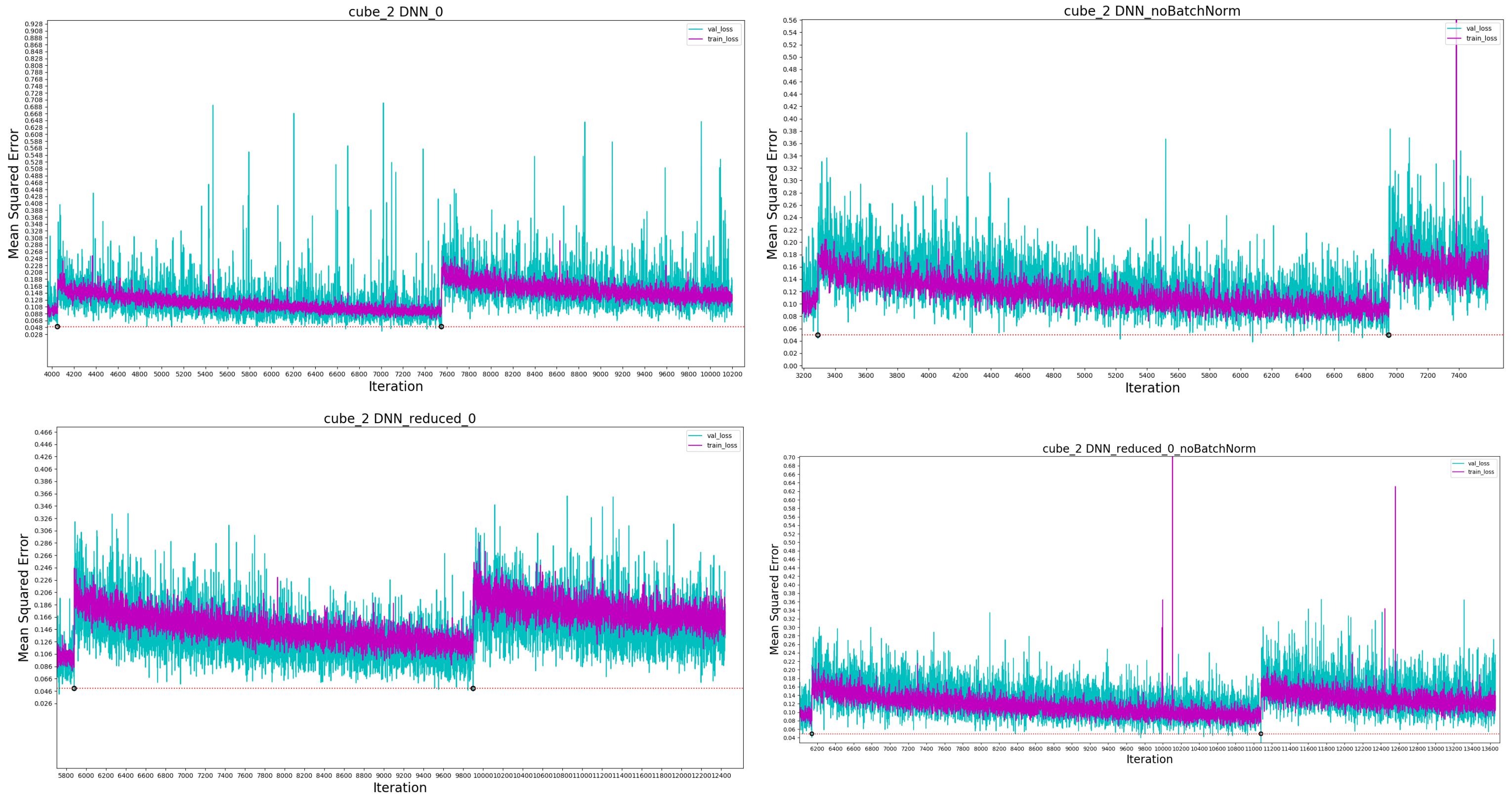


Figure 15: Last two convergence points of the four models.

Compared to the number of states of the 2x2x2 cube, 3,674,160, this means that `DNN_noBatchNorm` has almost visited all possible states, while the other models have re-visited multiple states to achieve 13 convergence points. For the 2x2x2 cube this is not a problem, but we aim to solve more difficult puzzles with more states like the 3x3x3 cube, so visiting all states is not feasible. Two solutions are going to be studied: to increase ε in order to produce more convergence points in less iterations, or to invent a new way of generating states with which the reward signal is propagated more efficiently. I will study both options.

5.4 A new way of getting scrambled states: adaptative-K

From the knowledge acquired from training our first models, and the theory behind the DAVI algorithm, a new and improved way of generating states can be created. The true reward signal reaches states at a distance of d only after convergence point $d-1$. Assuming that J_0 is initialized to 0, and the rewards are constant and set to 1, the value iteration update $J_1(s) = \min_{a \in \mathcal{A}(s)} [1 + J_0(S(s,a))] = 1$ tells us that `model` should learn that all states are at a distance of 1, which is only true for few states. Then, after the first convergence point, J_1 is a function that approximates the constant value of 1, so the value iteration formula tells us that J_2 should learn that all states at a distance of 2 or higher are at a distance of 2. In this way, after convergence point c, the network is learning that all states at distances $c+1$ or higher are at a distance of $c+1$ (plus or minus all the approximation errors). Thus, we are re-feeding the network with different (and incorrect) values for the same state, which possibly leads to the slowing of training. The idea for our new way of generating states is to use the function `get_scrambled_deepcube` as before, but having a different value of `K` at each convergence point, called `effective_k`, and now the hyperparameter `K` acts as an upper bound for `effective_k`. The parameter `effective_k` has to take advantage of the fact that the true reward signal reaches states at a distance of d only after convergence point $d-1$, so it can be set to the convergence point number + 1. However, this can lead to some instability, which can be mitigated by adding a second hyperparameter, `conv_point_offs` (it gives better results when set to 1 or 2, depending on the

problem). The final formula for `effective_k` is:

$$\texttt{effective_k} = \min(\texttt{K}, \texttt{conv_point} + 1 + \texttt{conv_point_offs})$$

Thus, by taking the minimum, K acts as an upper bound for `effective_k`, which is useful if we know in advance the maximum distance to the goal state. For example, for the 2x2x2 cube, the maximum distance is 14, so probably there is no point in increasing `effective_k` above $K = 20$ (values less than 20 would assign a very low probability to states at a distance of 14). The effect of `conv_point_offs` is to retrain some states, but not all. For example, a value of 1 means that after convergence point $d - 1$ we will be training states at distances of up to d with the correct reward signal, but also states at a distance of $d + 1$ with an incorrect cost-to-go of d. The correct reward signal would arrive one convergence point later. This hyperparameter helps to bring the cost of those states closer to their true values.

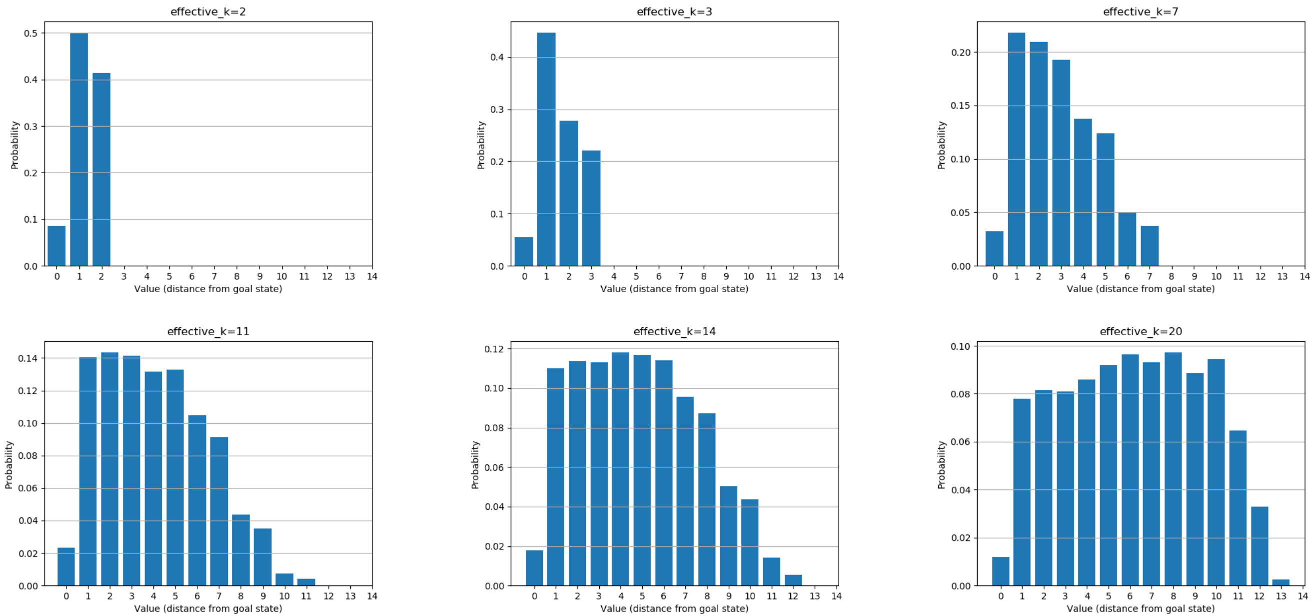


Figure 16: State probability distributions for different values of `effective_k`.

The state probability distributions for different values of `effective_k` are shown in Figure 16. As intended, lower values give all the probability to states closer to the goal state, while increasing this value means giving gradually more importance to far states. This technique is closer to a curriculum learning approach than the previous ways of generating random states. At the initial convergence points, the value of `effective_k` is small, thus learning simple states, the ones closer to the goal state. As the training progresses, more and more convergence points happen, thus increasing the `effective_k`, which leads to learning more difficult states.

5.5 Final 2x2x2 trained models

To test our two solutions for the inconvenience presented in Section 5.3 (the need to feed the algorithm with all the states if we want to achieve convergence), I have trained three other models. They are all based on the `DNN_reduced_0` architecture. This is done mainly for two reasons. First of all, the 2x2x2 Rubik's Cube is not a big puzzle, so a reduced version of the original architecture should suffice, taking into account that the reduced versions have achieved smaller errors. Also, the model `DNN_reduced_0` is chosen, and not the version without batch normalization (`DNN_reduced_0_noBatchNorm`) because it performs better when using it as a heuristic for BWAS (see Section 5.7). The model named `DNN_final_2x2x2` has exactly the same architecture as `DNN_reduced_0` and has been trained with the same hyperparameters, but a value of $\varepsilon = 0.07$ is chosen instead of $\varepsilon = 0.05$ to achieve more convergence points. On the other hand, `DNN_final_2x2x2_nr` and `DNN_final_2x2x2_nrr` are trained

using adaptative-K, the new way of getting scrambled states. A value of `conv_point_offs` = 1 has been chosen for this problem. As this new way causes the loss to decrease more quickly, we can exploit this fact by checking for convergence more frequently. Thus, `DNN_final_2x2x2_nr` has been trained with $C = 5$, so convergence is checked every 5 iterations. To double check that the change of C is effective, a model `DNN_final_2x2x2_nrr` has been trained with adaptative-K and exactly the same hyperparameters as `DNN_final_2x2x2_nr`, but choosing $C = 10$, which is exactly the value used for training the models in Section 5.1.

Model	Total Iterations	# updates	Model updates iterations (convergence points)
`DNN_final_2x2x2`	7460	16	40, 70, 90, 100, 120, 150, 230, 510, 980, 1380, 1990, 2350, 2680, 3330 5460 7460
`DNN_final_2x2x2_nr`	2850	13	40, 45, 55, 65, 80 115, 265, 315, 470, 680, 795, 1315, 1490
`DNN_final_2x2x2_nrr`	2920	13	40, 60, 100, 110, 160, 230, 400, 440, 450, 580, 840, 1250, 1640

Notice that `DNN_final_2x2x2` has achieved 16 convergence points, and the 13th and 14th ones happen at iterations 2680 and 3330, which is already an improvement compared to the version trained in Section 5.3. Also, `DNN_final_2x2x2_nr` and `DNN_final_2x2x2_nrr` achieve 13 convergence points with only 1490 and 1640 iterations, clearly showing the benefits of the adaptative-K method because they have only been fed with less than half of the states the 2x2x2 cube has. Furthermore, `DNN_final_2x2x2_nr` has converged faster than `DNN_final_2x2x2_nrr`, which shows the effect of checking for convergence more frequently.

5.6 Admissibility and Consistency of the heuristics for the 2x2x2 cube

We are going to use the previous models as the heuristic for BWAS, so a natural question that arises is if they are admissible or consistent. If so, an A^* algorithm (which is a BWAS search with $\lambda = 1$ and $N = 1$) always returns the optimal solution, and if it is consistent, it also finds it without revisiting any nodes. I have measured the degree in which any of the models trained is admissible or consistent, for the last four of their convergence points. I have done so by generating 40,000 random states (by calling `get_scrambled_deepcube` with $l = 2000$ and $K = 20$). Then, with the help of the optimal solver, the percentage of the states that are admissible is computed, that is, the ones in which the heuristic does not overestimate the true cost-to-go. Also, for the non-admissible states I have computed the average absolute overestimation, which is always 0.304 or lower. Finally, the percentage of states that are consistent is computed. That is, the percentage of states in which the heuristic value is not greater than the step cost of getting to a neighbor of that state plus the heuristic of that neighbor.

Model	Iteration	% Admissible States	Avg. Overestimation	% Consistent states
`DNN_0`	2080	81.885	0.116	70.79
-	3140	87.407	0.174	66.957
-	4050	81.572	0.235	66.255
-	7550	79.022	0.296	67.63
`DNN_noBatchNorm`	1790	71.375	0.222	59.842
-	2540	71.827	0.298	63.632
-	3290	53.58	0.216	57.36
-	6950	58.7	0.26	54.5
`DNN_reduced_0`	2440	75.417	0.169	69.56
-	3850	65.17	0.208	63.805
-	5880	69.48	0.246	60.025
-	9900	76.535	0.289	49.225
`DNN_reduced_0_noBatchNorm`	2030	70.792	0.304	61.607
-	3660	70.765	0.342	61.367

-	6140	62.785	0.266	57.685
-	11080	49.705	0.195	59.79
DNN_final_2x2x2	1990	86.47	0.251	57.63
-	2350	79.177	0.213	67.615
-	2680	80.812	0.29	61.007
-	3330	76.87	0.255	66.712
DNN_final_2x2x2_nr	680	74.412	0.213	65.645
-	795	66.46	0.208	70.187
-	1315	71.315	0.244	58.772
-	1490	73.172	0.243	66.21
DNN_final_2x2x2_nrr	580	83.147	0.257	69.82
-	840	80.565	0.272	68.25
-	1250	71.19	0.221	65.11
-	1640	64.005	0.225	57.292

As can be seen, training the algorithms for longer does not always mean they are going to be closer to constitute a consistent or admissible heuristic. In fact, the opposite happens with `DNN_reduced_0_noBatchNorm` and `DNN_final_2x2x2_nrr`. Also, the results for the final algorithms are remarkable. For example, more than 80% of states are admissible for the `DNN_final_2x2x2` at iteration 2680, and more than 61% of states are consistent. The most admissible model is `DNN_0` at iteration 3140, with 87.407% of the states being consistent and an average overestimation of 0.174, while the most consistent one is also `DNN_0` at iteration 2080. Finally, the models with batch normalization seem to outperform the ones without it, so now it is clear why the original authors of the DeepCubeA paper chose to include batch normalization layers into their models.

5.7 Final 2x2x2 BWAS results

Up to this section, we have only considered the results of the neural network models. Let's use these models as heuristics for the BWAS search. BWAS is adjusted by two hyperparameters. N controls how many states are expanded at the same time to compute their heuristics in parallel, and λ is in charge of regulating the weight of the g-score. If λ is close to 0, the value of the heuristic is given more importance, so BWAS acts like a greedy search (depending on the choice of N). If it is 1, it is closer to an A^* search (also depending on the choice of N). The first objective is to test all models to study which one of them to select as a final model. But our resources are limited, we can not yet afford to perform a grid search to choose the best N and λ hyperparameters for all the models. Instead, we will have to choose only one value for them. In the DeepCubeA paper, they mention that the best hyperparameters for the 3x3x3 cube are $\lambda = 0.6$ and $N = 10000$. We will choose $\lambda = 0.7$ to test all the models of the 2x2x2 cube, and we have to choose a lower N, $N = 5$ because of computational resources restrictions (having a higher value would expand more states, and would mean computing the heuristic value of many more states in parallel).

Each of the tests has been performed by generating 10,000 states by calling `get_scrambled_deepcube`. For each of the four last convergence points of all the previous models, BWAS has been executed by using the model as a heuristic for the search. The percentage of optimal paths found by each of the algorithms is shown in the next table.

Model	Iteration	% Optimal solutions found
BWAS + `DNN_0`	2080	99.409
-	3140	99.089
-	4050	98.018
-	7550	97.857
BWAS + `DNN_noBatchNorm`	1790	98.128
-	2540	97.697
-	3290	96.7867
-	6950	96.4364
BWAS + `DNN_reduced_0`	2440	99.069
-	3850	98.978
-	5880	98.408
-	9900	97.037
BWAS + `DNN_reduced_0_noBatchNorm`	2030	98.1781
-	3660	97.757
-	6140	97.2472
-	11080	97.557
BWAS + `DNN_final_2x2x2`	1990	98.55
-	2350	99.089
-	2680	98.958
-	3330	96.8768
BWAS + `DNN_final_2x2x2_nr`	680	98.99
-	795	99.04
-	1315	98.768
-	1490	98.178
BWAS + `DNN_final_2x2x2_nrr`	580	98.595
-	840	97.2972
-	1250	96.506
-	1640	97.017

First, notice that every model finds the optimal solution for more than 96.4% of the states. This is a successful result; in the original paper they managed to find the optimal path 60.3% of the time for the 3x3x3 cube. Second, the model that performs better is `DNN_0` at iteration 2080, which is precisely the model with the highest percentage of consistent states. It finds the optimal solution 99.4% of the time, which is very close to being perfect. Also, the models with batch normalization seem to perform better, just as with admissibility and consistency. In addition, the `final` models achieve good results. The choice of training a model with the adaptative-K and a reduced value of C gives a good quality result, since `DNN_final_2x2x2_nr` at iteration 795 has achieved a 99.04% success rate having been fed only with 795,000 cubes in total. Now, we will choose the four best performing models, `DNN_0`, `DNN_reduced_0`, `DNN_final_2x2x2` and `DNN_final_2x2x2_nr` at their best iterations: 2080, 2440, 2350 and 795. The objective is to compare their performance at every distance. Each of the 10,000 cubes generated before were classified by their distance to the goal state, and then, the percentage of optimal solutions found is computed per distance. All of the four models achieve a 100% success for states at distances 0-6. Also, they have all found the optimal path for a randomly generated cube at the maximum distance, 14. The percentages of optimal solutions found for cubes at distances between 7 and 13 are shown in the table below.

Model	Iteration	7	8	9	10	11	12	13
BWAS + `DNN_0`	2080	100	99.774	98.302	98.007	99.274	100	100
BWAS + `DNN_reduced_0`	2440	100	100	99.151	96.279	98.298	100	100
BWAS + `DNN_final_2x2x2`	2350	100	99.771	98.555	97.130	98.074	100	100
BWAS + `DNN_final_2x2x2_nr`	795	99.622	99.083	98.590	96.928	98.215	100	100

First of all, none of these models fall below 96.279% for any distance. In addition, the majority of the errors occurs in states at distances between 8 and 11, having a 100% success rate for states at higher distances. This could be because distances between 8 and 11 amount for the majority of states. Also, remember that `DNN_reduced_0`, `DNN_final_2x2x2` and `DNN_final_2x2x2_nr` have the same architecture, but have been trained differently. A higher value of ε produced a slightly faster convergence rate for `DNN_final_2x2x2`, which achieves almost the same results as `DNN_reduced_0`. But the biggest change is when training it with the new way of generating states, adaptative-K. `DNN_final_2x2x2_nr` does not underperform `DNN_reduced_0` by more than 0.6% for any of the distances, while having been trained only with a third (795,000 states) of the states that `DNN_reduced_0` saw during its training (2,440,000 states).

Taking all this information into account, it is clear that `DNN_0` (the original model in the DeepCubeA paper), and `DNN_reduced_0` (the reduced version of this model) are the ones that achieve better results in absolute terms. However, because it has been trained with a novel way of generating states, and because in Hanoi we are going to be interested in generating convergence points at a fast rate, we are going to study `DNN_final_2x2x2_nr` in more detail. Because it is only one model, we can perform a grid search to discover how N and λ impact the percentage of optimal solutions found, the number of nodes expanded, and the average running time for each input. A grid search works by generating all the possible combinations of N and λ values (from a list of possible values for one of those parameters), and computing the performance of the BWAS algorithm for any of those combinations. As possible values of N, I chose 3, 5, 7, and for λ, 0.2, 0.5, 0.7 and 1. To compute the performance, we use 506 states. 500 states have been randomly chosen by calling `get_scrambled_deepcube` with $K = 20$, and 6 states have been manually chosen at distances 9, 10, 11, 12, 13, 14. For each N, λ pair, the percentage of optimal solutions, the average number of expanded nodes and the average execution time is computed.

λ \ N	3	5	7
0.2	66.996	72.529	75.296
0.5	92.094	92.292	92.292
0.7	99.407	99.407	99.407
1.0	100	100	100

% of optimal solutions found by BWAS + `DNN_final_2x2x2_nr` at iteration 795.

All tests with $\lambda = 1$ result in an optimal path, which is a better result than the one we already had. In fact, the percentage of optimal solutions found is inversely proportional to λ, which was expected since a greater value of λ allots more importance to the true information discovered while executing the search, that is, the g-score. Also, higher values of N retrieve more optimal solutions, especially with $\lambda = 0.2$. With this value of λ, a BWAS search with $N = 1$ would behave almost like a greedy search. Higher values of N make the algorithm expand more states per iteration, thus increasing the likelihood of finding the goal state sooner. In fact, in the limit, if we used a value $N \approx \infty$, all the states in the priority queue would be expanded at each iteration of the BWAS algorithm. In that case, it would behave almost like a breadth-first search. First, the goal state would be expanded. Then, all the states in the priority queue would be expanded, which are exactly the states at a distance of 1. Then, all states at a distance of 2, and so on. In that case, all the states would be visited by

increasing distances, and the value of λ would only control the order in which the states at the same distance would be visited.

λ \ N	3	5	7
0.2	233.0177	214.3458	218.5849
0.5	155.01778	161.2628	168.4782
0.7	158.54	165.936	173.0454
1.0	358.16996	363.7964	369.223

Average number of expanded nodes per state by BWAS + `DNN_final_2x2x2_nr` at iteration 795.

While a value of $\lambda = 1$ allows us to find all optimal paths, it does so by expanding a large number of states. As can be seen in the previous table, it expands more than the double of the states expanded by $\lambda = 0.7$ and $\lambda = 0.5$. Also, on average, $\lambda = 0.2$ expands more than 214 nodes per input state, which can be due to the lengthy solutions it finds which result in expanding more nodes in total. The effect of N can be noticed: for larger N, more states are expanded on average, but the effect is not as influential as the effect produced by λ.

λ \ N	3	5	7
0.2	5.6917	3.4137	2.9184
0.5	3.8875	2.5969	2.4892
0.7	4.0779	2.7218	2.488
1.0	8.7950	6.5999	5.10435

Average execution time in seconds per input state by BWAS + `DNN_final_2x2x2_nr` at iteration 795.

With respect to the execution time, N is very influential, which is exactly the effect we were looking for when designing the BWAS algorithm. Raising it from 3 to 7 divides the execution time by 1.5–1.9, depending on the value of λ. While $\lambda = 1$ achieves the best results, it does so by doubling the execution time of the other values of λ. Also, $\lambda = 0.2$ is very time consuming due to the additional expanded nodes.

In conclusion for the 2x2x2 Rubik's Cube, the DAVI algorithm to train a heuristic for the BWAS search works effectively. The new way of getting scrambled states, adaptative-K, in combination with the reduced version of the neural network architecture and BWAS with $\lambda = 0.7$ and $N = 7$ allow us to find the optimal solution for more than 99% of states in less than 2.5 seconds per state, having been trained only with 795,000 states. Actually, by compromising the execution time, we can raise the percentage of optimality by taking $\lambda = 1.0$.

5.8 Towers of Hanoi

I have trained the DAVI algorithm for Hanoi with 3 towers and 4, 7 and 10 disks. Since the optimal solution from the initial state has a length of 2^{n-1}, it is in our best interest to try to generate convergence points more frequently. Thus, we are going to use the adaptative-K method of generating states for training all the models from now on. Also, the parameter CC is important in order not to run out of memory. The effect is that it saves only one model out of every CC convergence points. For $H(4, 3)$ it is set to 1, while for $H(7, 3)$ and $H(10, 7)$ it is set to 10.

For $H(4, 3)$, I have trained the reduced model with the adaptative-K method, $B = 200$, $B_val = 20$, $K = 2^5$, $C = 2$, $\varepsilon = 0.05$, `conv_point_offs`$= 2$. The value of $B = 200$, the number of states the neural network is fed per iteration, may seem low, but this problem has only 81 different states, so it is actually very high. The optimal path has a size of 15, thus, we want to achieve at least 15 convergence

points. This is realized at iteration 346, when the network has been fed with 69,200 states, which means it has seen every state 800 times or more. This iteration is achieved after 15 minutes of training. To study this problem, we are going to choose the last three convergence points, 298, 334 and 346.

For $H(7,3)$, I have trained `DNN_0` with the adaptative-K method, $B = 400$, $B_val = 40$, $K = 2^7$, $C = 2$, $CC = 10$, $\varepsilon = 0.05$ and `conv_point_offs`=2. It has achieved 142 convergence points, requiring 1542 iterations and 145 minutes of training. To study it, convergence points at iterations 1320, 1354 and 1424 are chosen. This time, I do not select the last iterations because we need to explore some of the other convergence points. This is due to the large number of convergence points that the DAVI algorithm generates for $H(7,3)$.

Finally, for Hanoi with 3 towers and 10 disks, I have tried to train it with both the complete model `DNN_0` and the reduced one `DNN_reduced_0`. The initial attempt for training `DNN_0` with $B = 500$, $B_val = 50$, $K = 2^{10}$, $C = 2$, $CC = 10$, $\varepsilon = 0.05$, `conv_point_offs=2` resulted in 582 convergence points at iteration 10,000. This problem only has 59,049 states, so it should be possible to generate more convergence points in less iterations. I tried changing the model to `DNN_reduced_0`, and changing $B = 200$, $B_val = 50$, $C = 1$, to check more often for convergence and to feed less states to the model per iteration. I also set $\varepsilon = 0.07$ to increase the number of convergence points. This only achieved 342 convergence points in 10,000 iterations (which is not a bad result because of the change in the parameter B). With the exact same setting but raising the value of ε to 0.1, 842 convergence points are achieved at iteration 12084. This is a much better result, but the network has re-visited multiple times the same states. Finally, with $\varepsilon = 0.5$, after 3 hours, 1011 convergence points are achieved at the end of iteration 3572. We are going to study convergence points at iterations 3477, 3542 and 3572. The final results for all the Hanoi puzzles are shown in the following table.

Problem	Model	# updates	Model updates iterations (convergence points)
$H(4,3)$	`DNN_reduced_0`	15	74 88 92 102 118 136 188 214 222 236 258 264 298 334 346
$H(7,3)$	`DNN_0`	142	72-1542
$H(10,3)$	`DNN_reduced_0`	1011	2-3572

In the following table, the convergence and admissibility results are shown.

Problem/Model	Iteration	% Admissible States	Avg. Overestimation	% Consistent states
$H(4,3)$ `DNN_reduced_0`	298	78.443	0.294	46.678
-	334	53.306	0.296	44.243
-	346	69.490	0.443	51.475
$H(7,3)$ `DNN_0`	1320	93.0519	0.084	64.762
-	1354	94.1703	0.014	59.973
-	1424	90.6109	0.101	68.083
$H(10,3)$ `DNN_reduced_0`	3477	91.76977	0.817	73.4321
-	3542	97.367	0.02	75.8105
-	3572	92.175	0.428	81.922

The model for $H(4,3)$ seems to achieve worse results in terms of admissibility and consistency, while the ones for $H(7,3)$ and $H(10,3)$ reach admissibility levels above 90%. Also, the average overestimation of `DNN_0` at iteration 1354 is very low, 0.014. In combination with an admissibility rate over 94%, this model performs very well in terms of admissibility, but the consistency is 59.973%, below other iterations for the same problem. Lastly, the model for $H(10,3)$ achieves great consistency percentages, above 73.4%, which is a result that outperforms those of the 2x2x2 Rubik's Cube.

Finally, we will study if these algorithms can find the optimal path for their respective Hanoi problems. For the BWAS search, we will use $N = 7$, and values of $\lambda \in [0.3, 0.5, 0.7, 1.0]$. As explained in Section 3.1.2, Hanoi not only has a goal state, but also an initial one. We are mainly interested in

analysing if these models can find the optimal path from the initial state, which is also one of the states that are at a maximum distance from the goal state. The results are affirmative. All iterations of the model trained as a heuristic for solving $H(4,3)$ using a BWAS search with $N=7$ find the optimal path of length 15 described recursively in Section 3.1.2. It does so by expanding between 73 and 76 states for values of λ between 0.3 and 1, finding the goal state in 0.6 to 0.7 seconds. For $H(7,3)$, the BWAS search finds the optimal solution of length 127, for values of $\lambda > 0.3$, in an execution time between 15 and 20 seconds, and expanding between 1676 and 2028 states depending on the value of λ. A sub-optimal solution of length 131 is found by iteration 1354 of `DNN_0` with $\lambda = 0.3$, which can be found at Appendix 7.4. After 104 successful actions, it spends more actions than it should when trying to move disks 0, 1 and 2 to the auxiliary (central) tower. Those disks were on top of the target tower, and they had to be moved to the auxiliary one in order to place the 4th disk on top of the target tower. Instead of moving the uppermost disk, disk 0, to the auxiliary tower, the search decides to back off and place it on the initial one. Maybe at this point it tries to solve the puzzle as best as it could, but because disk 0 is blocking the initial tower, rather than undoing this mistake, it decides to pull out the disk number 1 from the target tower, placing it on the auxiliary tower instead of the initial one where it should have been put. Finally, it undoes these two mistakes by spending two more actions, which leaves a total length of $127 + 4 = 131$ steps.

For $H(10,3)$, a BWAS search using iterations 3477, 3542 and 3572 of `DNN_reduced_0` as a heuristic also finds the optimal solution. It does so in 404 seconds, expanding 56010 states when using $\lambda = 0.7$. This is sub-optimal, since this problem has a total of 59049 different states, so it has almost expanded the entire space state. However, from this solver we can extract useful information about the Hanoi problems $H(n,3)$ with $n \leq 10$. Remember from Section 4.3 that we can map states of $H(n,3)$ into states of $H(10,3)$, and then solve these states with the model trained for $H(10,3)$, which is a very simplified version of **transfer learning** [4]. All instances of $H(n,3)$ with $n = 1,2,3,4,5,6,7,8,9$ have been optimally solved by all of the three selected iterations (3477, 3542 and 3572) of the model trained for $H(10,3)$, with λ values 0.5, 0.7 and 1.0. Furthermore, iteration 3572 with $\lambda = 0.7$ optimally solves $H(4,3)$ in 0.695 seconds, expanding only 64 states, which is a better result than the one achieved with the model specifically trained for this problem. Also, it solves $H(7,3)$ in 13.075 seconds, expanding only 1505 states, which also improves the solution by the `DNN_0` model trained for $H(7,3)$.

5.9 3x3x3 Rubik's Cube

Finally, I have trained the original model of the papers, `DNN_0`, for the 3x3x3 Rubik's Cube but using adaptative-K as the way to generate random states. To execute the DAVI algorithm for the 3x3x3 Rubik's Cube, the authors of the DeepCubeA paper [2] use two NVIDIA Titan V GPUs to train the neural network, with six other GPUs used in parallel for data generation. They completed the neural network training in 36 h. I will only use the laptop described in Appendix 7.1. The hyperparameters I will use are $K = 30$, $CC = 1$, $B = 10020$, $B_val = 1020$, $C = 50$, `conv_point_offs`$= 2$ and $\varepsilon = 0.05$. In the original paper they use C=5,000, but my resources are more limited (see Section 7.1), so I have to check for convergence more often to generate more convergence points.

Problem	Model	# updates	Model updates iterations (convergence points)
3x3x3 Cube	`DNN_0`	10	50 100 150 200 250 300 350 500 700 1400

As can be seen, the results in terms of the number of convergence points are below the ones achieved for the other puzzles. The God number for the 3x3x3 Rubik's Cube is 26 in the quarter turn metric, so we would need at least 26 convergence points to propagate the reward signal to all the possible states. Nevertheless, according to the authors that discovered the God number for this cube, [21], at distance 10 or less from the goal state there are 6,701,836,858 states, which is a significant amount of states compared to the 2x2x2 cube and the Towers of Hanoi for 10 disks or less. To get to iteration 1400, 56 hours of training were needed, and the model was fed with 14 million cubes. That is, on average, each iteration takes 2.4 minutes, so if we wanted to feed the algorithm with 10^{10} states, as they did in the DeepCubeA paper [2], we would have to train it for 998,004 iterations in total, which

would take approximately 2,495,000 minutes, that is, more than 1700 days. Although we are using a different method for generating states, which could improve the number of iterations needed, we would have to use better hardware if we wanted to train it completely.

As we have not built an optimal solver for this cube, we can not check if this model is admissible or the percentage of states for which it finds an optimal solution. But the consistency can be easily checked by the same method as before. With a test of 30,000 states generated by calling `get_scrambled_deepcube` with $K = 30$, the results are shown in the following table:

Model	Iteration	% Consistent states
`DNN_0`	350	93.9566
`DNN_0`	500	93.6433
`DNN_0`	700	90.836
`DNN_0`	1400	86.77

These results seem promising, although a good part of the consistent states may be consistent because the network has not yet seen them during its training, since these models have only seen less than a $3.24\text{x}10^{-11}\%$ of the total number of possible states for the 3x3x3 cube. These results may worsen if we continued the training. In fact, iteration 350 achieves a 93.9566% while iteration 1400 only gets a 86.77%, which possibly indicates a downtrend.

Finally, to test how well the algorithm solves the cube, we will use `DNN_0` at iteration 1400, since it is the last convergence point, with the intention of being able to solve cubes at a distance of at least 10. Also, the hyperparameters are set to $N = 7$ and $\lambda = 0.6$. The value of N is selected because of the hardware limitations I have, while the value for λ is the one that produces better results in the original DeepCubeA paper. In [27], a cube at a maximum distance, 26, and the list of actions to reach that state, which they call *Superflip*, are provided. These actions are D', D', L, R, D', L, R', D, U', R', U', U', L', R', B, F, U', F', F', L', R, U', U', F', U', U', and the resulting cube is shown in Figure 17.

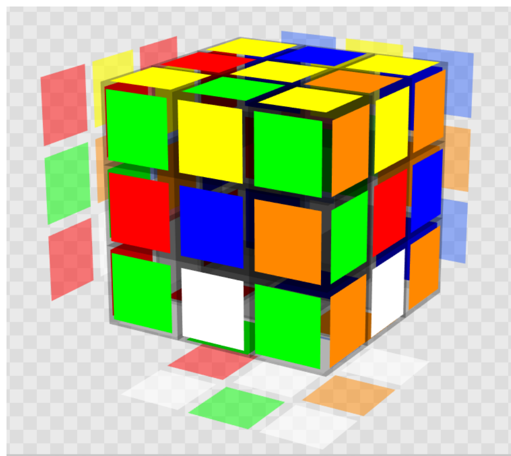


Figure 17: State at a distance of 26 from the goal state. Generated by twizzle.

To test the performance of BWAS with our trained heuristic, we apply the following process. We generate a test cube by applying the first action, D', to obtain a cube at a distance of 1. Then, we apply the first two actions to obtain a cube at a distance of 2, etc. In general, after applying q of these actions, we get a state at a distance of q from the goal state, since the list of actions forms an optimal path; the objective is to check if BWAS can find this optimal path. The behavior of the algorithm is very good: it finds the optimal path for the cubes at distances between 1 and 15. For the ones at distances lower or equal to 11, it finds an optimal path in less than 1.11 seconds, expanding 101 states or less. After that, the time spent and the number of nodes expanded increase exponentially. For the cube at 12 moves from the goal, it takes 5.65 seconds and it expands 472 nodes. For the one at 13

moves, 29.444 seconds and 2341 nodes expanded. For the cube at 14 moves from the goal, it takes 153.349 seconds and it expands 10895 nodes. Finally, the one at 15 moves takes 1157.79 seconds and 69793 nodes are expanded. This performance drop can be due to the exponential ramification of the number of states of the 3x3x3 cube, combined with the inexperience of the neural network model with those states. It has only converged 10 times, so the reward signal has not reached states at distances higher than 10. Moreover, `conv_point_offs` is set to 2, which means it has never seen states at distances higher than 12, thus it is probably not a good heuristic when trying to build a path from those states. For comparison, in the DeepCubeA original paper they use GPUs with 64GB of RAM to parallelize the BWAS search. By choosing $\lambda = 0.6$ and $N = 1000$, they obtain a solution for a randomly generated cube (at any distance from the goal) in an average time of 214.03 seconds. Their solutions have an average length of 21.36 moves, and the BWAS search expands 4,516,478 nodes on average.

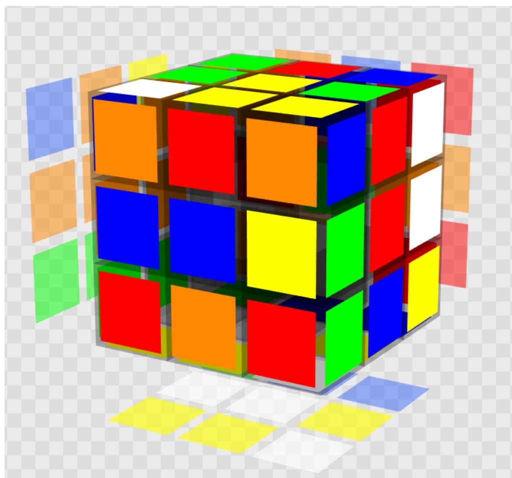


Figure 18: State at a distance of 15 from the goal state. Generated by twizzle by applying actions D', D', L, R, D', L, R', D, U', R', U', U', L', R', B to the solved cube.

Now, let's focus on one of those states. The cube at distance 15 is reached by applying D', D', L, R, D', L, R', D, U', R', U', U', L', R', B to the goal state. The solution found for this cube is B', R, L, U', U', R, U, D', R, L', D, R', L', D', D'. There is a clear pattern here. First, notice that U', U' is equivalent to U, U, since they both perform a 180º turn of the upper face. The same holds with D', D' and D, D, etc. Having this in mind, the solution found by the solver arises from group theory. We have built the cube by applying actions $abc \cdots z$. A way of returning to the original cube is to compute the inverse element of $abc \cdots z$, which is obtained from the inverses of these actions, but in reverse order, that is $z^{-1}y^{-1} \cdots a^{-1}$. This is what our algorithm has done (having in mind that D' is the inverse of D, U' is the inverse of U, etc), but combining the fact that the element D'D', and the element U'U', are its own inverse, that is, the orders of these elements are both 2.

6 Conclusión

Incluso habiendo técnicas previas para resolver un problema concreto, siempre es positivo intentar aplicar metodologías nuevas para así comprender mejor el problema y construir soluciones eficaces y aplicables a otras situaciones. Ejemplo de ello es el algoritmo implementado en este trabajo, el cual hemos aplicado a problemas con solución conocida, como es el caso de los cubos de Rubik, que suponen un reto todavía por la dificultad para obtener soluciones óptimas. El haber estudiado el problema de las torres de Hanói, con solución óptima conocida para el caso de 3 torres, nos ha permitido comprender en mayor profundidad el algoritmo. Esto en esencia posibilita su posterior aplicación en problemas de mayor complejidad, no tratados en este trabajo por sus altos requerimientos computacionales.

Hemos visto como una idea perteneciente al paradigma del aprendizaje por refuerzo, como es el *value iteration*, puede combinarse con otras técnicas del aprendizaje automático, como son las redes neuronales, y con algoritmos de búsqueda en grafos. Esto nos ha permitido construir un algoritmo compuesto que resuelve un problema inabarcable con cualquiera de estas técnicas por separado.

Conclusion

Even if there are previous techniques to solve a particular problem, it is always positive to try to apply new methodologies in order to better understand the problem and build effective solutions applicable to other situations. An example of this is the algorithm implemented in this work, which we have applied to problems with known solutions, such as Rubik's Cubes, which still pose a challenge due to the difficulty of obtaining optimal solutions. The study of the Hanoi Towers problem, with a known optimal solution for the case of 3 towers, has allowed us to understand the algorithm in greater depth. This in essence makes possible its later application in problems of greater complexity, not treated in this work due to its high computational requirements.

We have seen how an idea belonging to the reinforcement learning paradigm, such as *value iteration*, can be combined with other machine learning techniques, such as neural networks, and with graph search algorithms. This has allowed us to build a composite algorithm that solves a problem that would be unmanageable with any of these techniques separately.

References

[1] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. “Solving the Rubik’s cube with approximate policy iteration”. In: Proceedings of International Conference on Learning Representations (ICLR). 2019.

[2] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. “Solving the Rubik’s cube with deep reinforcement learning and search”. In: *Nat Mach Intell 1, 356–363* (2019).

[3] *BatchNormalization layer*. URL: https://keras.io/api/layers/normalization_layers/batch_normalization/.

[4] Jason Brownlee. “A Gentle Introduction to Transfer Learning for Deep Learning”. In: (). URL: https://machinelearningmastery.com/transfer-learning-for-deep-learning/.

[5] Brian Davey. *Proofs by induction*. URL: https://www.amsi.org.au/teacher_modules/pdfs/Maths_delivers/Induction5.pdf.

[6] Dorit Dor and Uri Zwick. “SOKOBAN and other motion planning problems”. In: *Computational Geometry* 13.4 (1999), pp. 215–228. ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(99)00017-6. URL: https://www.sciencedirect.com/science/article/pii/S0925772199000176.

[7] Dylan Wang. *Rubik’s Cube Move Notation*. URL: https://jperm.net/3x3/moves.

[8] John Ewing and Czes Kosniowski. “Puzzle It Out: Cubes, Groups and Puzzles”. In: *Cambridge: Press Syndicate of the University of Cambridge. p. 4. ISBN 0-521-28924-6* (1982). URL: https://books.google.com/books?id=s1E4AAAAIAAJ&pg=PA4.

[9] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artifcial Intelligence and Statistics (eds Gordon, G., Dunson, D. and Dudík, M.) 315–323* (2011).

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015.

[11] *Hugging Face Models*. URL: https://huggingface.co/models.

[12] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *In Proceedings of International Conference on Machine Learning (eds Bach, F. and Blei, D.) 448–456* (2015).

[13] Woolsey Johnson and William Story. “Notes on the ”15” Puzzle”. In: *American Journal of Mathematics* 2.4 (1879), pp. 397–404. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2369492 (visited on 05/10/2022).

[14] Diederik Kingma and Jimmy Lei. “Adam: a method for stochastic optimization”. In: *Proceedings of International Conference on Learning Representations (ICLR) (eds Bach, F. and Blei, D.)* (2015).

[15] Herbert Kociemba. “The Two-Phase-Algorithm”. In: (). URL: http://kociemba.org/cube.htm.

[16] Richard Korf. “Finding Optimal Solutions to Rubik’s Cube Using Pattern Databases ”. In: *AAAI-97 Proceedings* (1997).

[17] “Lights Out”. In: *wikipedia* (). URL: https://en.wikipedia.org/wiki/Lights_Out_(game).

[18] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. “Foundations of Machine Learning”. In: *The MIT Press* (2018).

[19] Andrew Moore and Christopher Atkeson. “Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time”. In: *Machine Learning* 13.1 (Oct. 1993), pp. 103–130.

[20] Sazli Murat. “A brief review of feed-forward neural networks”. In: *Communications, Faculty Of Science, University of Ankara* 50 (Jan. 2006), pp. 11–17. DOI: 10.1501/0003168.

[21] Tomas Rokicki and Morley Davidson. “God’s Number is 26 in the Quarter-Turn Metric”. In: (2014). URL: https://cube20.org/qtm/.

[22] TOMAS ROKICKI, HERBERT KOCIEMBA, MORLEY DAVIDSON, and JOHN DETHRIDGE. “THE DIAMETER OF THE RUBIK’S CUBE GROUP IS TWENTY”. In: *SIAM J. DISCRETE MATH* (2013).

[23] Scott Vaughen. *Counting the Permutations of the Rubik’s Cube*.

[24] Paul Stockmeyer. “Variations on the Four-Post Tower of Hanoi Puzzle”. In: (1994).

[25] Richard Sutton and Andrew Barto. "Reinforcement Learning, An Introduction". In: *The MIT Press* (2018).
[26] Thomas Rokicki. *God's Number is 20*. URL: https://www.cube20.org/.
[27] Thomas Rokicki. *God's Number is 26 in the Quarter-Turn Metric*. URL: https://www.cube20.org/qtm/.
[28] Sai Varsha. "A Star Algorithm". In: (2014). URL: http://cs.indstate.edu/~skonakalla/paper.pdf.
[29] Xin Wang, Yudong Chen, and Wenwu Zhu. "A Survey on Curriculum Learning". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), pp. 1–1. DOI: 10.1109/TPAMI.2021.3069908.

7 Appendix

7.1 Computer Specifications

All tests and results have been computed on a laptop with the following specifications:

- OS: Ubuntu 18.04.5 LTS, Linux: 4.15.0-176-generic
- Processor: Intel Core i7-8550U CPU 8 cores @ 1.80GHz cache size: 8192 KB
- RAM: 16GB
- SSD

7.2 Model statistics, first training

In all the following figures, the horizontal dotted line represents $\varepsilon = 0.05$, the black circles show when the evaluation model updates happened, the cyan color is the validation loss, and the magenta color the training loss.

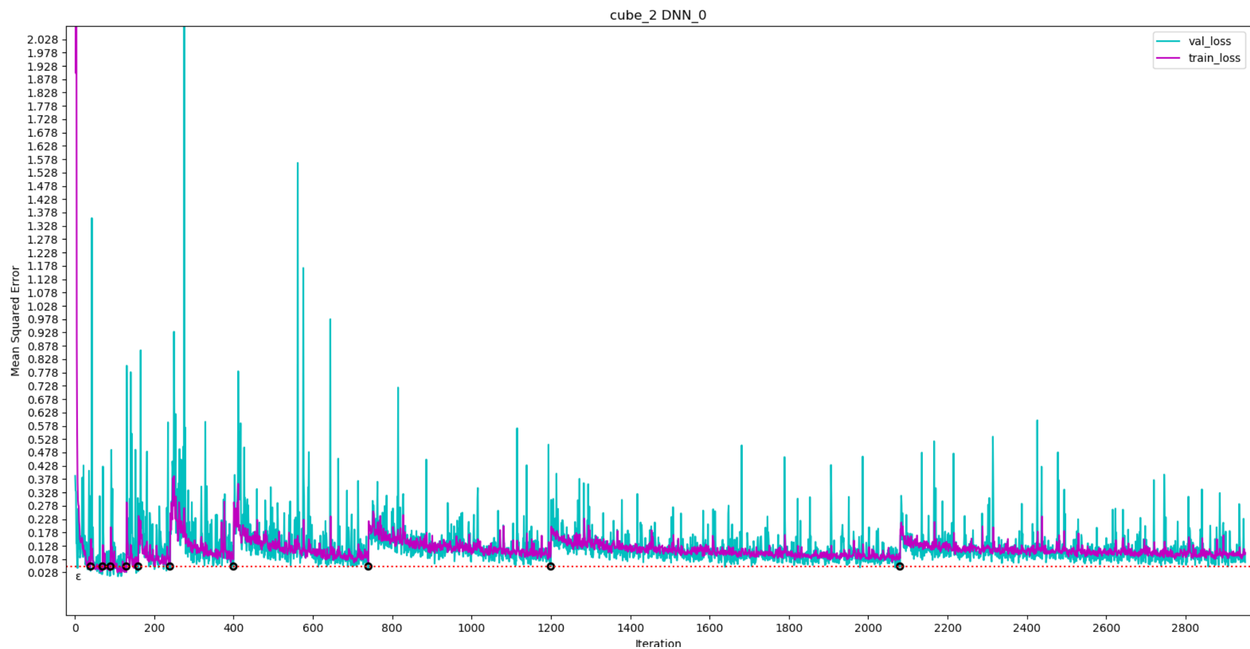


Figure 19: Losses of DNN_0 model until the iteration 2950.

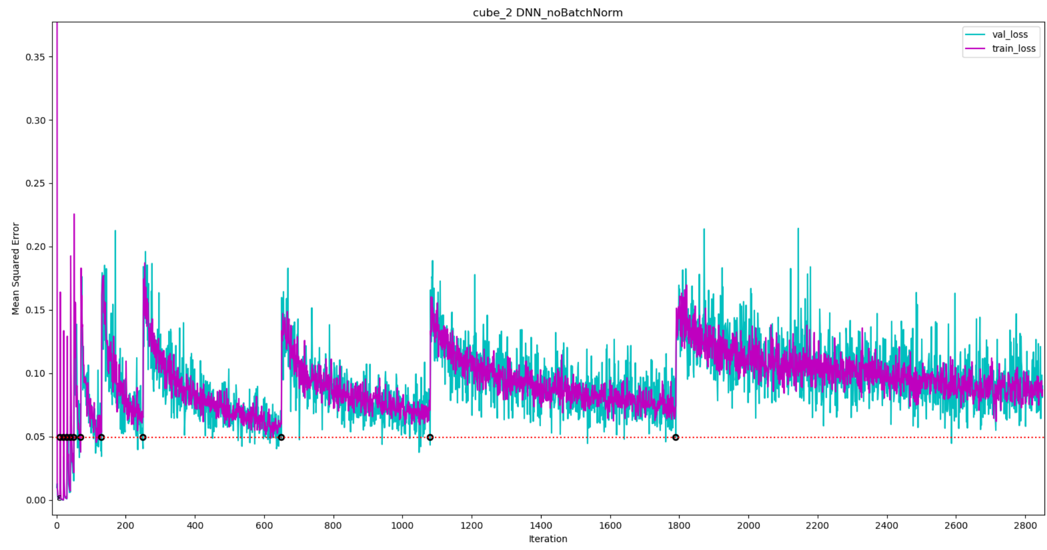


Figure 20: Losses of `DNN_noBatchNorm` model until the iteration 2850.

7.3 Model statistics, four trained models

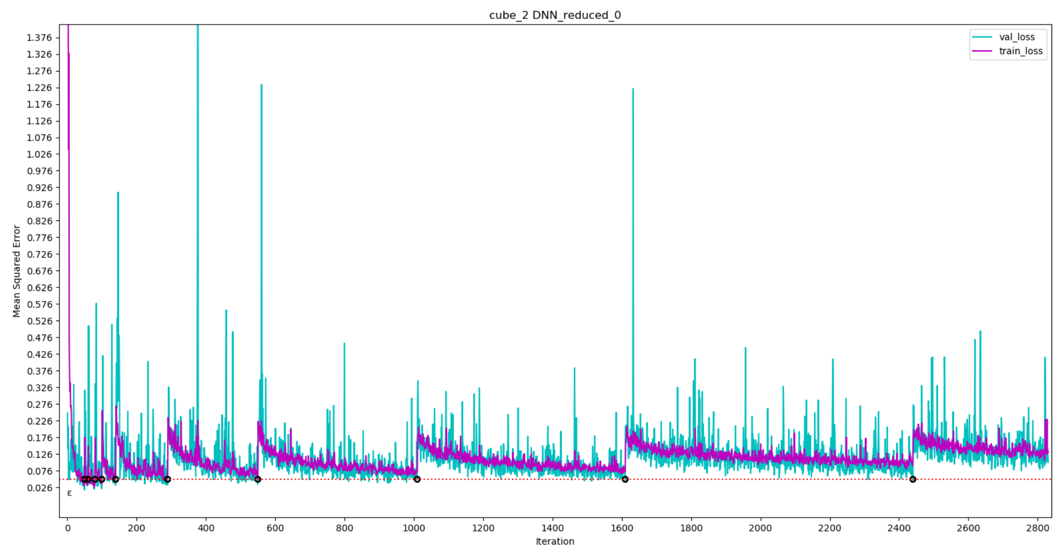


Figure 21: Losses of `DNN_reduced_0` model until the iteration 2830.

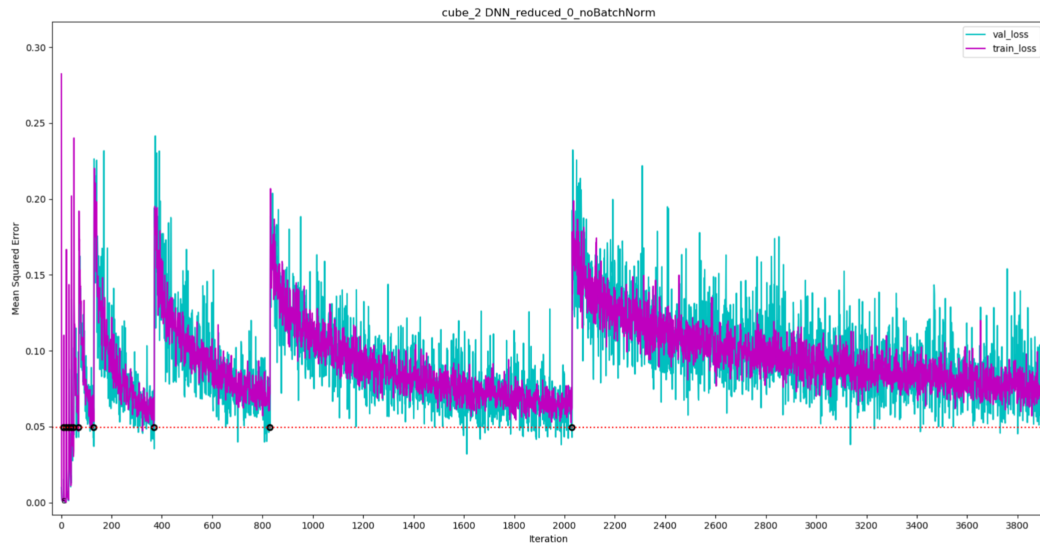


Figure 22: Losses of `DNN_reduced_0_noBatchNorm` model until the iteration 3910.

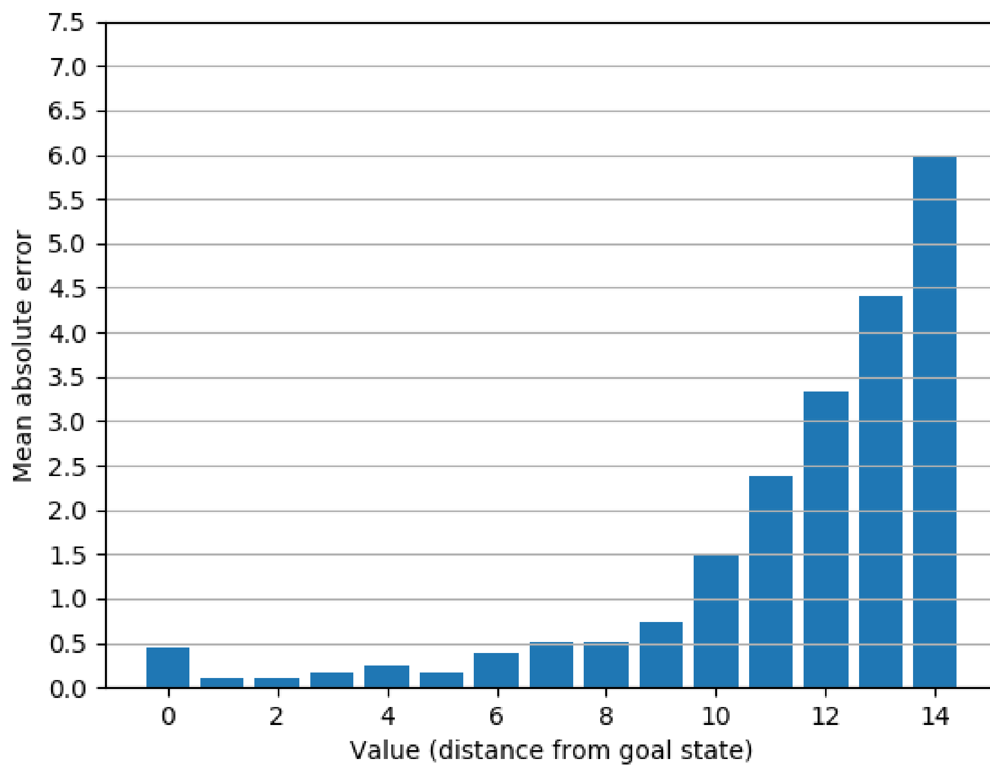


Figure 23: Mean absolute error for the last convergence of `DNN_0` (iteration 7550).

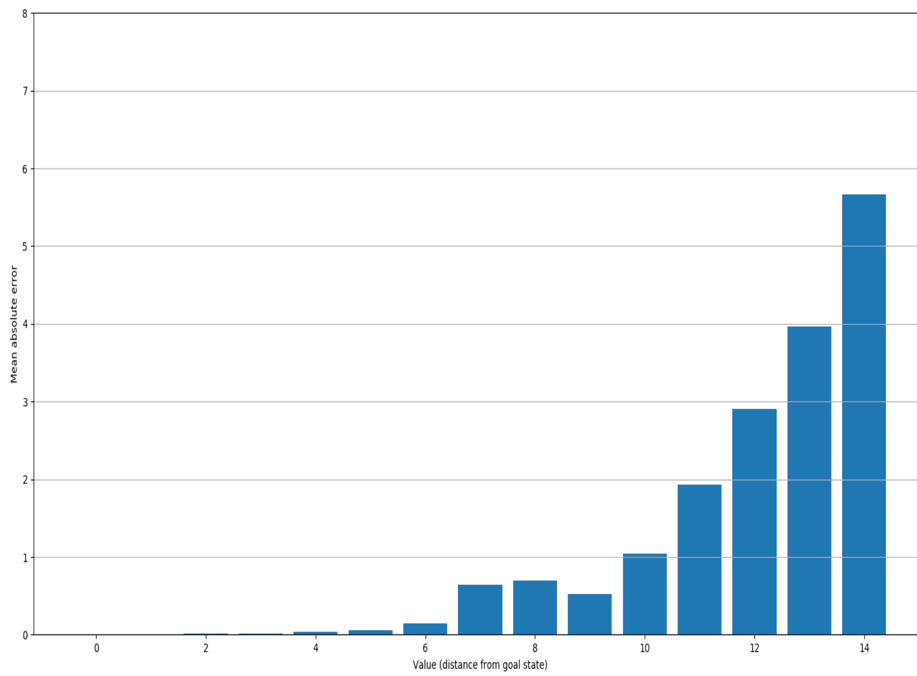


Figure 24: Mean absolute error for the last convergence of `DNN_noBatchNorm` (iteration 6950).

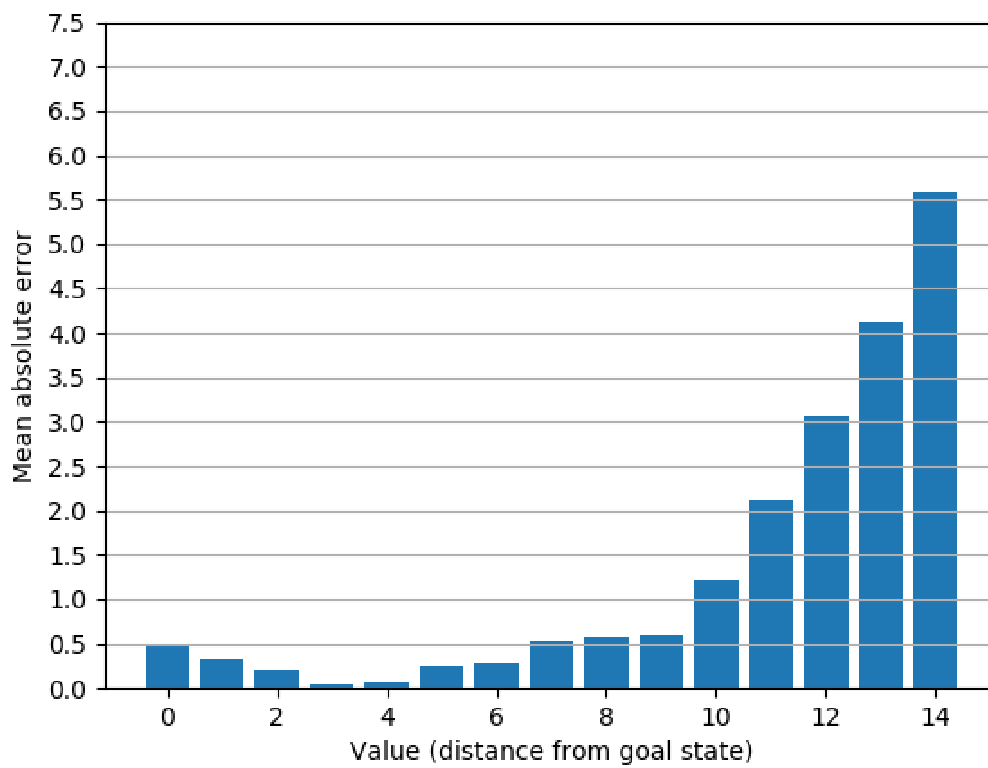


Figure 25: Mean absolute error for the last convergence of `DNN_reduced_0` (iteration 9900).

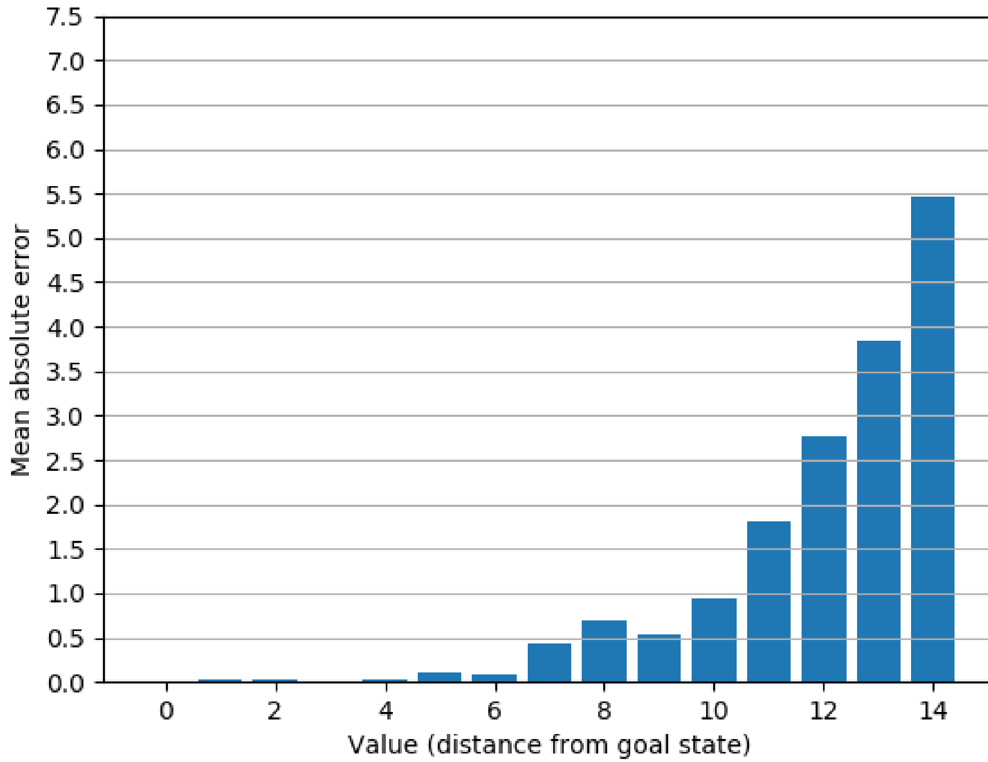


Figure 26: Mean absolute error for the last convergence of `DNN_reduced_noBatchNorm` (iteration 11080).

7.4 Hanoi: solutions found

The next one is the solution to $H(7,3)$ found by a BWAS search $N = 7$, $\lambda = 0.7$ using `DNN_0` at iteration 1354 as a heuristic. It is a list of 127 actions, which applied to the initial state give us the final state. Each action is represented as (d, i, j), where we move disk d from tower i to tower j. The tower where all disk are initially placed is tower 2, while the final tower, the one where we want to place all disks, is tower 0.

[(0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (3, 2, 1), (0, 0, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (4, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (3, 1, 0), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (5, 2, 1), (0, 0, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (3, 0, 2), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (4, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (3, 2, 1), (0, 0, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (6, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (3, 1, 0), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (4, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (3, 0, 2), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (5, 1, 0), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (3, 2, 1), (0, 0, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (4, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (3, 1, 0), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0)]

The following list is the sub-optimal solution to $H(7,3)$ found by a BWAS search $N = 7$, $\lambda = 0.3$ using `DNN_0` at iteration 1354 as a heuristic.

[(0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (3, 2, 1), (0, 0, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (4, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2),

(0, 0, 1), (1, 0, 2), (0, 1, 2), (3, 1, 0), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (5, 2, 1), (0, 0, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (3, 0, 2), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (4, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (3, 2, 1), (0, 0, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (6, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (3, 1, 0), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (4, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (0, 2, 0), (1, 2, 1), (0, 0, 1), (3, 0, 2), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (5, 1, 0), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (3, 2, 1), (0, 0, 2), (1, 0, 1), (0, 2, 1), (2, 0, 2), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 2, 1), (0, 0, 2), (1, 0, 1), (0, 2, 1), (4, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0), (2, 1, 2), (0, 0, 1), (1, 0, 2), (0, 1, 2), (3, 1, 0), (0, 2, 0), (1, 2, 1), (0, 0, 1), (2, 2, 0), (0, 1, 2), (1, 1, 0), (0, 2, 0)]