Fast Polyhedra Analysis with deep Q-networks

Jakub Kotal

Bachelor Thesis
Novemeber 2018

Supervisor:
Gagandeep Singh, Prof. Dr. Martin Vechev

ETH zürich

Abstract

We use deep Q-networks to increase the performance of numerical static analysis whilst maintaining sufficient precision. In this work, we build upon an existing framework that uses linear function approximation in order to model a decision policy that leverages precision loss for performance gain inside of polyhedra analysis. We expand this framework with the use of a nonlinear Q-function approximation. We then further optimise this method by testing new features, reward functions, as well as other problem specific optimisations. We also try to increase the flexibility of the proposed algorithm. Finally, we train and test our resulting algorithm, with which we are able to achieve a considerable gain in both performance and precision.

Contents

Introduction

1

Our ever-growing reliance on computers has led to the explosion of newly written software. This growth is paralleled by the expansion of its complexity. Furthermore, an ever-increasing number of the structures surrounding us have become controlled by this software. From simple things such as automatic doors to vastly more complex and important systems such as self-driving cars, medical software or nuclear weapons software. In some of these applications a program bug could simply result in a slight nuisance, however in others, its existence could have major consequences. The implications can be illustrated by some infamous examples where bugs such as buffer overflows, rounding errors or division by zeros have led to the explosion of the rocket Ariane 5 or lethal radiation overdoses caused by Therac-25. As such, due to the importance of these devices and the risk associated with potential flaws, one can see the necessity of their invulnerability and its verifiability. Unfortunately, as the complexity of these programs increases, their verification can no longer be simply done by hand and the need for complete and formal verification methods becomes critical.
Static analysis is one of the possibilities to combat these problems. Its task is to analyse a program without its execution. Doing so, it discovers weaknesses inside the code, weaknesses that could lead to vulnerabilities. As such, it helps with debugging and provides a better notion of safety about the code. In recent years, static analysis has seen a growing commercial use, especially in safety-critical fields. Recent advances in this field use clever mathematical properties in order to increase the capabilities of such methods. Other techniques leverage the domain of artificial intelligence(AI) to increase the fields capacities. For, until AI is capable of writing invulnerable software for us, we can at least use it to verify ours.

1.1 Problem Statement

The main focus of numerical static analysis is to observe the effects that program expressions and statements have on the variables inside of the program. In order for these methods to work precisely, a numerical abstract domain is needed that can capture the relationships between the variables. An important attribute of a domain is its expressivity, that is to say how complex are the relationships between variables that the domain can represent. The more expressive a domain is, the more complicated these can become. The most expressive domain is the polyhedra domain that can represent any linear constraint between variables. However, its expressivity comes at a price. It has high space and time cost, having worst-case exponential complexities in both. Although other domains that do not suffer from these problems also exist such as OCTAGON [Miné 2006], ZONE [Miné 2001] or PENTAGON [Logozzo and Fähndrich 2010],

they only achieve this by losing expressivity. Various methods exist that attempt to increase the performance of the polyhedra domain. Some of these methods exchange some precision loss to increase the performance. A reinforcement learning approach exists for this problem, this method removes certain constraints during the analysis to reduce its complexity. However, removing the wrong constraint can have a disastrous effect on the overall precision, making the removing procedure a very sensitive task.

1.2 Goals

The goal of this thesis is firstly to adapt some of the recent developments in new areas of reinforcement learning in order to use them in polyhedra analysis. Secondly, we attempt to optimise said methods with some problem-specific knowledge. With this we hope to increase the efficiency of the analysis and learn an action selection policy that achieves the correct balance between precision and performance, hopefully outperforming pre-existing methods.

1.3 Related work

The concept of using reinforcement learning to decide between different operators with various levels of abstraction has been explored in [Singh et al. 2018]. This work used Q-learning with a linear function approximation, for the estimation of the Q-function.
The domain of parametric program analysis also explores the idea of leveraging precision loss for performance gains with works like [Oh et al. 2015, Oh et al. 2014, Liang et al. 2011]. These works attempt to tune the precision and cost of the given analysis according to the program being analysed. The main difference between the above-mentioned works and ours is that they do not adaptively tune the analysis while it is being conducted. The work [Chen et al. 2008] uses a floating point representation of the polyhedra to increase performance with the loss of some precision as well.
Other works increase the performance of static analysers without the loss of precision
[Singh et al. 2015, Singh et al. 2017]. However, these analyses remain slower than ours as with our method we were able to outperform [Singh et al. 2018], which in turn already outperformed [Singh et al. 2017].

1.4 Structure of this Document

Chapter 2 briefly introduces reinforcement learning, explaining the important concepts and its main goals. It will also describe some of the recent advances inside the field and the achievements of these methods.

Chapter 3 addresses Polyhedra analysis. It will explain its usefulness inside static analysers and how it works. The chapter also introduces some of the modifications made to the domain, in order to reduce its shortcomings.

Chapter 4 addresses the work done throughout this thesis. It will first address how the two previous chapters were combined and the different choices behind the design of the basic algorithm. The chapter also discusses the different methods tested to further optimise the results.

Chapter 5 discusses the different experiments that were run whilst finding the optimal combination of parts for the algorithm. The chapter also presents the results of the execution of the finalised versions of our algorithms on different benchmarks.

Chapter 6 will be a final conclusion on what was achieved during this thesis and the results we were able to obtain. The chapter also contains a short discussion of a few possibilities for further improvement.

Reinforcement Learning

2

Reinforcement Learning [Kaelbling et al. 1996] is a very general problem of machine learning studied in a multitude of different fields, such as game theory, information theory or statistics. It is a very broad concept and the foundations are the following. An agent interacts with a given environment, at its disposal, it has a set of various actions. When an action is performed it receives a reward. The goal of the agent is to devise an action selection policy that should maximise the cumulative reward of the selected actions.

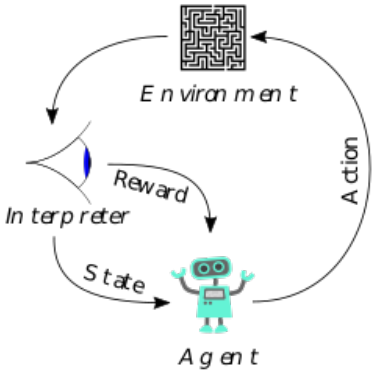

Figure 2.1: *Concepts of RL illustration [Wikipedia contributors 2018]*

2.1 Concepts

In order to be able to solve a problem with reinforcement learning (RL), it first has to be mapped to the following RL concepts:

- A set of agent states S, with the initial state $s_0 \in S$
- A set of actions A
- A function giving the reward of performing an action from s_t to s_{t+1}, $r(s_t, a_t, s_{t+1}) \in \mathbb{R}$

During the execution of the program, the agent will first start in the initial state s_0. Then, at each timestep $t = 0, 1, ...$, the agent will pick an action $a_t \in A$, this action will then be executed, the agent will move from s_t to s_{t+1} and receive the corresponding reward $r(s_t, a_t, s_{t+1})$. This process will be repeated until a final state is reached. In RL, state transitions typically satisfy the

Markov property. This property states that the next state s_{t+1} only depends on the current state s_t and on the action taken at this state a_t. We call the sequence of actions and states from the initial to the final state an episode. The goal of the agent is to devise an action selection policy that maximises the cumulative reward, which is measured as the sum of the rewards inside of an episode. We will demonstrate this reward with the following example.

Example 2.1
Let's assume we have an episode with two timesteps and an action set with only two possible actions. In the following tree, we can see the different actions that we can take at each state and their respective reward.

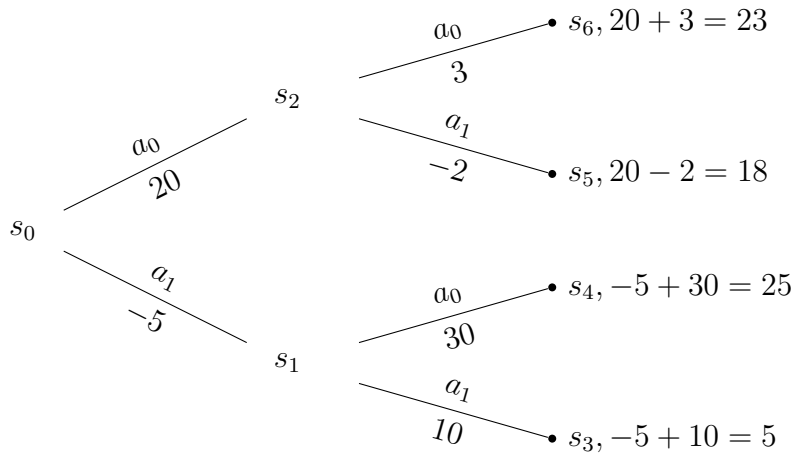

Figure 2.2: *Example of an execution episode with two possible actions*

The cumulative reward is the summed reward of all the actions taken during an episode. This is shown at the end of the leaf nodes in the example above. Therefore, an ideal decision policy would choose the actions $\{a_1, a_0\}$ in that order for the above episode.

2.2 Q-function

We call Q-function or quality function, a mapping $Q : S \times A \to \mathbb{R}$ that specifies the cumulative reward of picking an action a_t in state s_t. If this function was known, the reinforcement learning problem would become quite trivial, as it would simply suffice to pick, at every time step, the action with the highest Q-function value.

2.3 Learning the Q-function

The Q-function, therefore, represents the ideal decision policy that one can make. However, since the Q-function is not known beforehand, the goal of reinforcement learning is to obtain

it. Most RL approaches achieve this by iteratively updating their policy until a suitable one is learned, this is called training. During training, the agent is allowed to interact freely inside the environment, picking actions randomly or with the strategy learned so far. It observes the different rewards it obtains and updates its policy accordingly.
We shall now briefly introduce a few concepts used during training.

Exploration-exploitation tradeoff/dilemma As mentioned above, during training we either pick the best action with the till-now learned policy or we simply pick a random action. The trick is in finding a correct balance between the two, as each has its own particular advantage. This is called the exploration-exploitation dilemma [Yogeswaran and Ponnambalam 2012]. If we always pick the best action, we will converge quickly but the risk of getting stuck in local minima is fairly high. On the other hand, if we only pick random actions, the convergence will be very slow and we might simply diverge and not be able to learn a strategy at all. Most training algorithms start with a high probability of exploration and then, as training progresses, increase the exploitation probability in order to solidify the learned policy.

Learning rate The learning rate is also part of the exploration-exploitation dilemma. With this ratio, we can modify the importance that we give to newly acquired information. The lower it is the harder it will be to overwrite an already learned policy. With a high learning rate, we will adapt more easily to new problems, but this will be accompanied by a higher risk of overfitting.

Discount factor In reinforcement learning, we approximate the Q-function with the following equation:

$$Q'(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = a, a_t = a, \pi] \tag{2.1}$$

where π is the learned policy.
We call γ the discount factor. It represents the importance that we give to the future reward. If $\gamma = 0$ then only the immediate reward will be considered and if $\gamma \approx 1$ then we will learn a policy that will give the same importance to the future rewards as to the immediate reward. As before, one has to choose between a faster convergence and a higher risk of getting stuck in local minima and not learning a global policy.

2.3.1 Q-learning

Q-learning [Watkins and Dayan 1992] is one of the main algorithms used in reinforcement learning. Given any Markov decision process, infinite time and a high exploration rate, Q-learning is guaranteed to learn the ideal decision policy for the particular problem. It works by iteratively updating its policy using a weighted average of the old values and the new information. Q-learning can obtain optimal results on simpler tasks. However, it lacks one key attribute.

2.3.2 Generalisation

Generalisation is the capacity of an agent to make decisions about unvisited states. As said before, Q-learning is guaranteed to learn the optimal policy, but only under the assumption that during training the agent will have explored all states multiple times and all actions will have been taken. If the state space of our problem was to be large, training in this way would be extremely time intensive. Therefore, a very interesting capacity for our agent would be the capability to generalise from trained data, towards previously unseen circumstances. Normal Q-learning lacks this capacity and if it is presented with a previously unvisited state, its policy will be no better than random. In order for a generalisation to be possible, some kind of function approximation has to be done on the Q-function.

2.3.3 Linear function approximation

Several approximation techniques exist. One of the most popular ones, as well as an important technique for the rest of this paper, is Q-learning with linear basis function approximation. This method models the Q-function as a linear combination of basis functions where each basis function assigns a value to a $(s_t \in S, a_t \in A)$ pair and each feature has its own basis function. Such an approximation offers several advantages: it is efficient, converges relatively quickly and the learned policy can be quite easily interpreted. Unfortunately, linear approximations also have three main shortcomings.

- It requires a discrete state space
- It requires a discrete action space
- If the optimal policy is nonlinear, it cannot be learned

Some workarounds for these problems are possible, such as discretising the continuous variables. However, these also cause a loss of information and therefore precision.
Recently, a lot of work has been done on the incorporation of nonlinear function approximators inside of reinforcement learning. These methods do not suffer from the above-mentioned problems and have therefore been able to offer some very promising results on problems that were considered too complex for reinforcement learning in the past.

2.3.4 Deep Q-networks

Deep Q-networks (DQN) [Mnih et al. 2013] are a novel method used for reinforcement learning that has gained a lot of attention in recent years. This is due to the ability that a single algorithm has in achieving very good results on a broad array of tasks and this without the need of any specialised knowledge about the task beforehand. This has led some to consider the method as the first major step towards general artificial intelligence.
What separates deep Q-networks from other reinforcement learning methods is that they use neural networks (NN) (i.e nonlinear functions) to approximate the Q-function. Reinforcement learning has been known to diverge when nonlinear function approximators have been used. Various techniques have been developed to combat the divergence of these methods, such as

neural fitted Q-iteration [Riedmiller 2005] or experience replay memory [Mnih et al. 2015]. These techniques have allowed NN's to become a viable tool for reinforcement learning and to achieve some very remarkable results. Most notably, [Mnih et al. 2015] were able to design an algorithm that, without the need of any prior knowledge about the task, achieved human-level performance on a large number of different Atari games.

Polyhedra Analysis

3

Polyhedra analysis is one of the main tools of static analysis [Cousot and Cousot 1977]. During the execution of a program, its statements and expressions can create relationships between the different variables of the program. Polyhedra are the most expressive way of modelling all these different relations. Even though alternatives for constraint representation exist, none are as expressive as polyhedra. Unfortunately, this domain has worst-case exponential space and time complexity, meaning that using it on most real-world sized applications would very often cause it to either timeout or run out of memory making it very impractical. Therefore, other domains have been more widely used, such as Octagon [Miné 2006], Zone [Miné 2001] or Pentagon [Logozzo and Fähndrich 2010], but all of these are less expressive and therefore less precise by design. In recent years, several techniques have been developed that have managed to speed up polyhedra analysis without causing it to lose precision [Gange et al. 2016, Jourdan 2017, Maréchal and Périn 2017]. Some of these methods are implemented inside the ELINA framework. We shall now briefly present the methods that are relevant to this work.

3.1 Polyhedra representation

One of the techniques used to increase the performance of static analysis involves the way in which we represent our domain [Singh et al. 2015]. Polyhedra, for example, can be represented with both the constraint representation and the generator representation [Motzkin 1953]. To illustrate these different representations I will proceed with the following example. Let's assume we have the following statement:

Example 3.1

$$
\begin{aligned}
&if\ x \geq 1 \wedge y \geq 1 \wedge x - y \leq 0 : \\
&\qquad \ldots \\
&end
\end{aligned}
$$

Inside of the if statement, the polyhedron would have the following shape:

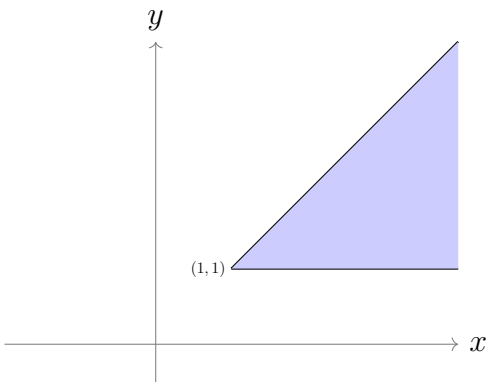

Figure 3.1: *Example of a polyhedron modelling the above constraints*

There are two ways for us to represent this information.

3.1.1 Constraint representation

In constraint representation, we model the polyhedron as the intersection of a finite number of closed half spaces and a finite number of subspaces. The resulting polyhedron can be written as:

$$P = \{x \in Q^n | Ax \leq b \wedge Dx = e\} \tag{3.1}$$

where A, D are matrices and b, e are vectors of natural numbers. Therefore, the constraint representation of the above example would be:

$$C = \{-x \leq -1, -y \leq -1, x \leq y\} \tag{3.2}$$

3.1.2 Generator representation

In order to encode a Polyhedron with the generator representation, we have to model it as the convex hull of three items:

- A finite set $V \in Q^n$ of vertices $v_i \in V$.
- A finite set $R \subseteq Q^n$ of rays. $r_i \in R$ are direction vectors of infinite edges of the polyhedron with one end bounded. Each ray starts from a $v \in V$.
- A finite set $Z \subseteq Q^n$ of lines. $z_i \in Z$ are direction vectors of infinite edges of the polyhedron with both ends unbounded. Each line passes through a $v \in V$.

The result of generator representation on the previous example would have the following form:

$$G = \{V = \{(1, 1)\}, R = \{(1, 1), (1, 0)\}, Z = \emptyset\} \tag{3.3}$$

3.1.3 Polyhedra domain

Once we can represent our polyhedra, to make them useful, a set of operators has to be defined as well. The polyhedra abstract domain consists of the polyhedral lattice: $(P, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, and a set of operators that we can apply on them. The different operators are the following:

- Inclusion test: $P \sqsubseteq Q$, holds if all generators in G_P satisfy all constraint in C_Q
- Equality test: $P = Q$, checks whether the polyhedra are equal
- Join: $P \sqcup Q$, G_{output} is the union of the generators from the input. C_{output} is obtained by incrementally adding the generators from G_Q to the polyhedron defined by C_P
- Meet: $P \sqcap Q$, C_{output} is the union of the constraints in the input polyhedra. G_{output} is obtained by incrementally adding the constraints in C_Q to the polyhedron defined by G_P
- Widening, this operator is applied to accelerate convergence since the polyhedral lattice has infinite height:

$$C_{P \nabla Q} = \begin{cases} C_Q \text{ if } P = \bot; \\ C'_P \bigcup C'_Q, \text{ otherwise;} \end{cases}$$

  where $C'_p = \{c \in C_P | C_Q \vdash c\}$, and
  $C'_Q = \{c \in C_Q | \exists c' \in C_P, C_P \vdash c$ and $((C_P,) \bigcup \{c\}) \vdash c'\}$ where $C \vdash c$, tests wether c can be entailed from constraints in C
- Conditional: let $\otimes \in \{\leq, =\}, 1 \leq 1 \leq n, \alpha \in Q$ then $\alpha x_i \otimes \delta$ adds the constraint $(\alpha - a_i)x_i \otimes \delta - a_i x_i$ to the constraint set C
- Assignment: $x_i = \delta$, first adds x_i to P then augments C with $x_i - \delta = 0$

In the following table, we can see the respective complexities of the different operators according to the representation

Table 3.1: *Operator complexity according to the representation*

Operator	Constraint	Generator	Both
Inclusion ($\sqsubseteq$)	$O(mLP(m,n))$	$O(gLP(g,n))$	$O(ngm)$
Join ($\bigsqcup$)	$O(nm^{2^{n+1}})$	$O(ng)$	$O(ng)$
Meet ($\sqcap$)	$O(nm)$	$O(ng^{2^{n+1}})$	$O(nm)$
Widening (∇)	$O(mLP(m,n))$	$O(gLP(g,n))$	$O(ngm)$
Conditional	$O(n)$	$O(ng^{2^{n+1}})$	$O(n)$
Assignment	$O(nm^2)$	$O(ng)$	$O(ng)$

$m = |C|, g = |G|, LP(m,n)$ is the complexity of solving a linear program with m constraints

and n variables
As we can see, no representation is less complex than the other, as some of the operators are faster in one but others in the other. But, as we can see in the last table, when both representations are available all the operators have a polynomial complexity.

Chernikova's Algorithm

The first optimisation that one can do is to keep both representations of the polyhedron and for each operator pick the representation that minimises the time complexity. A conversion between the two representations is possible thanks to Chernikova's algorithm [Chernikova 1968]. However, Chernikova's algorithm still has worst-case exponential complexity and it becomes the new bottleneck for the double representation.

3.2 Polyhedra Decomposition

Another technique used for increasing the efficiency of polyhedra analysis is the use of online decomposition [Singh et al. 2017]. It is based on the observation that during the execution of a program, not all its variables are dependent on one another. Using this observation, we can separate the set of all variables into smaller, independent sets. Therefore, instead of representing the whole set of variables with one large polyhedron, we can instead represent it with various smaller ones.
Let's assume we have a set of variables χ in a Polyhedron P. The set χ can be partitioned as $\pi_P = \{\chi_1, \chi_2, .., \chi_r\}$, $\chi_i \subseteq \chi$. We call the partitioning of the set permissible if no two variables x_i and x_j exist that are in different blocks in π and are related by a constraint in P. We call the $\top$ partition, a partition where all the variables are in one block, i.e no partitioning is done. Similarly, the $\bot$ partition is a partition where each variable has its own block, i.e all variables are independent of one another. Once the decomposition has been done in this way, during the execution of an operator, it only has to be executed on the subset of blocks that contain the same variables. This allows for a very large performance gain in both space and time, giving us the following time complexity for the various operators.

Table 3.2: *Operator complexity with decomposition*

Operator	Decomposed
Inclusion ($\sqsubseteq$)	$O(\sum_{i=1}^{r} n_i g_i m_i)$
Join ($\bigsqcup$)	$O(\sum_{i=1}^{r} n_i g_i m_i + n_{max} g_{max})$
Meet ($\sqcap$)	$O(\sum_{i=1}^{r} n_i m_i)$
Widening ($\bigtriangledown$)	$O(\sum_{i=1}^{r} n_i g_i m_i)$
Conditional	$O(n_{max})$
Assignment	$O(n_{max} g_{max})$

Example 3.2
Let's consider the variables $X = \{x_1, x_2, x_3, x_4\}$ and $C = \{x_1 + 2 \cdot x_3 \leq 10, x_2 = 1\}$.
This polyhedron can be partitioned into three blocks $\pi_P = \{\{x_1, x_3\}, \{x_2\}, \{x_4\}\}$ and the constraints of the corresponding blocks are equal to:

$$C_{P_1}\{x_1 + 2 \cdot x_3 \leq 10\}, C_{P_2}\{x_2 = 10\}$$

Operator decomposition

As we change the model of our domain we must also update the operators inside this domain. For the sake of brevity, I will not go into detail about each of them, but I will concentrate on the join operator as it plays an important role in this paper.
During the execution of the join of P and Q, the join operator can cause the creation of constraints between different blocks of π_P and π_Q. In the worst case, all the blocks can be merged creating the $\top$ partition. In order to reconstruct the $\top$ partition from the blocks the following formula is used:

$$P = P_1 \bowtie P_2 \bowtie ... \bowtie P_r = (C_{P_1} \cup C_{P_2} \cup ... \cup C_{P_1}, G_{P_1} \times G_{P_2} \times ... \times G_{P_r}) \tag{3.4}$$

Due to the cartesian product, building the $\top$ partition can blow up the number of generators and cause the join to be a serious bottleneck of online decomposition.

3.2.1 Reinforcement Learning for polyhedra analysis

Both of the techniques presented in chapters 3.2 and 3.3 achieve considerable performance gain without having to sacrifice any precision. Unfortunately, at some point compromises have to be made. Analysers that tune the precision and cost based on the program they are analysing are called parametric program analysers. Several such approaches already exist. [Oh et al. 2015, Liang et al. 2011, Heo et al. 2016]. As explained in section 3.3.1., one bad constraint can significantly decrease the performance of the whole program. The trick is being able to identify this constraint at the correct time. One of the explored solutions to this problem that we shall explore is training a reinforcement learning algorithm that will make the decisions of when and how to apply some abstractions [Singh et al. 2018].
Intuitively, using reinforcement learning for polyhedra analysis seems quite straightforward. Let's imagine that for the join operator that we have presented above, we have two different versions J_{pr} and J_{pe}. One of these is a slower and more precise join and the other, using some sort of abstraction, is faster but not as precise. The goal of the reinforcement learning method would then simply be to select the correct operator at the right time, thus enabling us to obtain the most precise result in a minimal time. Before this can be done, it is first necessary to initialise polyhedra analysis for reinforcement learning.

Adapting polyhedra analysis for Reinforcement Learning

As explained in section 2.1., reinforcement learning uses a defined set of concepts. In some way, these concepts have to be translated into the domain of polyhedra analysis. A possible mapping can be done in the following way.

Table 3.3: *Mapping of RL concepts to polyhedra analysis*

RL concept	Polyhedra analysis concept
Agent	Static analyser
State $s \in S$	features describing the polyhedron
Action $a \in A$	Abstract transformer
Reward function r	function representing the runtime and precision of a join

A more concrete explanation of the initialisation of these concepts will come in the following chapter.

Linear function approximation methods

Existing methods already exploit this idea [Singh et al. 2018]. This work uses a linear approximation of the Q-function in order to craft its decision policy. Q-learning is a common linear Q-function approximation technique and has shown some very good results when applied to the polyhedra domain.
However, as explained in chapter 2., new reinforcement learning methods seem to be more focused on the use of neural networks for the Q-function approximation, as they offer some advantages that Q-learning cannot. For example, if the Q-function of the particular problem were to be nonlinear, Q-learning will never be able to learn the ideal policy. In these cases, deep Q-networks have a very real opportunity of exceeding the performance of the Q-learning based methods.
The use of nonlinear approximators has not yet been tested on the domain of static analysis. In the following chapters, we shall explore the idea of using such methods in order to construct a new decision policy based on neural networks.

Fast polyhedra analysis with deep Q-networks

4

The goal of this work was to build upon the existing solutions used to make polyhedra analysis more efficient and use deep Q-networks to further improve the performance. Specifically, the goal was to extend the Elina Framework, more precisely the version of it that uses reinforcement learning for polyhedra analysis, in order to replace the Q-learning method with deep Q-networks for the Q-function estimation. Our work can be separated into three major subtasks. First of all, we had to figure out a way of calling a deep neural network from Elina, so that it could be used for the Q-function estimation. Secondly, In order to test out the validity of deep Q-networks on polyhedra analysis and to have a baseline for further optimisation, we implemented the basic deep Q-network training algorithm. Finally, we attempted to further optimise this problem by using task-specific knowledge.

4.1 Incorporating Neural Networks inside Elina

The first part, whilst definitely being the least interesting, was possibly the most laborious and a very key part of the whole work. The problem was that for the Q-value estimation we wanted to use neural network regression from the Keras framework in Python. We opted for this framework as it is very widely used for this type of problem, is relatively easy and intuitive to use whilst remaining very powerful. However, Elina, being written in C, the link between the python and C code had to be made. This was finally achieved by using the Python/C API and embedding the python code into the Elina framework.
One advantage of such a modification to the framework is that we can now learn online. In comparison to [Singh et al. 2018] where the training was done offline. Online training allows the algorithm to pick the best action according to the current decision policy. This is an essential feature for deep Q-networks, as it allows us to set an exploration rate and to reinforce the already learned policy, which is vital to prevent divergence.

4.2 Basic algorithm

In order to get the basic training algorithm running, first, the domain of polyhedra analysis had to be made compatible with the domain of reinforcement learning. As we have discussed in chapter 3.3.1., in order for reinforcement learning to work the following list of items has to be initialised inside the polyhedra domain:

- A set of features describing the polyhedra, to use as states S
- A set of actions A
- A reward function r

Luckily, since a reinforcement learning method [Singh et al. 2018] has already been used for polyhedra analysis we could simply reuse what has already been developed for our baseline algorithm.

4.2.1 Features

The reinforcement learning method uses a set of nine different features.

- The number of blocks.
- The minimal, maximal and average size of a block.
- The minimal, maximal and average size of the generator set of the union of input factors corresponding to a block.
- The number of variables with finite upper and lower bound.
- The number of variables with one finite and one infinite bound.

We will start off by reusing the same features. However, we will modify their bucketing, since, due to the nature of the Q-learning method, it is very restrictive. The new bucketing will be discussed in section 4.3.2.

4.2.2 Actions

As we discussed in section 3.2.1., the bottleneck of the decomposed analysis is the join. Therefore, the set of actions will be composed of various joins of different precision and performance. First of all, the cost of the joins depends on the size of the blocks. Therefore, bounding this size with a threshold would increase the performance. These four different thresholds are used: $threshold \in [5, 9], [10, 14], [15, 19], [20, \infty)$
Once we have decided on the size of a threshold, we have to determine what to do if the block has a greater size than the threshold. Different possibilities exist for splitting a large block into smaller ones, nonetheless, in essence, one has to pick a subset of constraints to remove from the block so as to make two subsets of its variables independent for it to be further partitionable. The following three constraint removals are used:

Stoer-Wagner min-cut

The first technique uses an algorithm based on the simple idea of removing the minimal amount of constraints in order to be able to split the block into two separate permissible partitions [Stoer and Wagner 1997].

Weighted constraint removal

The second and third constraint removal techniques are based on the same principle. They associate a certain weight to each constraint and then remove the constraint with the highest one. Two different weight distribution techniques are considered.
In the first one, we compute for each variable $x_i \in X$ the number of constraints it appears in, we can call this n_i. Then, for each constraint c_i, we set its weight to the sum of n_i of all the variables that are in the constraint.
In the second method, we first compute for each pair of variables $x_i, x_j \in X$, n_{ij} the number of constraints containing both x_i and x_j. The weight of the constraint is then the sum of all the n_{ij} of all the pairs of variables x_i, x_j contained in the constraint.
The idea behind removing the constraint with maximal weight is that most likely the variables with a large weight also occur in other constraints and therefore will not become unbounded once this constraint is removed. The precedent allows us to retain precision.

Merging blocks

The next three actions consist of various block merging strategies. The idea is to select different blocks and merge them together as long as the resulting blocks remain below the upper threshold limit, in order to increase the precision of the subsequent join. The following three block merging strategies are used:

- No merge, no blocks are merged
- Merge smallest first, we first merge the smallest two blocks together. We then remove the smallest block and continue as long as the resulting merge remains below the threshold.
- Merge small with large, similar to the previous strategy but this time we merge the smallest block with the largest.

In total, we have four different thresholds, three different constraint removal algorithms and three different block merging strategies. We can mix and match these together as we please, which means that in total we have $4 \cdot 3 \cdot 3 = 36$ different actions we can pick from.

4.2.3 Reward

As a reminder, the objective of the reward is to guide our learning policy, rewarding it when it takes actions towards our global goal and penalising it when it does otherwise. Therefore, the reward developed by the Q-learning method was the following:

$$r(s_t, a_t, s_{t+1}) = 3 \cdot n_s + 2 \cdot n_b + n_{hb} - \log_{10}(cyc) \tag{4.1}$$

Where n_s is the number of variables with a singleton interval (i.e $x_i \in [a, b], a == b$) inside the resulting polyhedron after the join. n_b is the number of bounded variables and n_{hb} the number of semi-bounded variables of the resulting polyhedron. cyc is the number of cycles required to perform the join.

4.2.4 Back to deep Q-networks

As we have finally initialised polyhedra analysis for reinforcement learning, we can proceed to the design of the training algorithm. Under its most basic form, the training algorithm could simply be the following. Pick a random action or the action with the maximal predicted Q-function, observe the reward and then retrain the network with the observed reward plus the discounted future reward on the given action. However, this very basic algorithm does not work very well as it runs into two major complications.

Divergence of decision policy The first of which is the divergence of the action selection policy. Meaning that the neural networks are not capable of creating a consistent policy, but rather pick different actions almost randomly most of the time.

Divergence of the action value prediction The second problem is that the values of the predictions from the estimators would diverge towards infinity. This is also known as the exploding gradient problem.

These problems are not new and are a fundamental problem when trying to use nonlinear functions for the Q-function approximation. Several techniques exist to combat these problems, many of these reduce the complexity of the network and/or the problem in general. However, new methods have been developed recently that allow the training of the estimators without limiting the size of the neural network, the number of possible actions and allow the training to be done online. We decided the use the following two concepts inside of our algorithm.

4.2.5 Experience replay memory

Experience replay memory [Mnih et al. 2015] is a biologically inspired mechanism. Its goal is to give the estimator a very basic concept of a memory and instead of directly learning from the current events happening, the agent learns from a random subset of its memory. More formally, during training, an array of a certain size filled with the past memory objects is kept. Each memory object contains the following items: $mem(s_t, a_t, r_t, s_{t+1})$. The current state, the action taken at this state, the observed reward and the next state. For training, we then simply pick a random subsample from the memory array and train from this data.
The objective of the memory is to introduce more variety into the training data. It arises from the observation that during the execution of a program, a particular strategy would be optimal for a certain period of time and afterwards another one would be the new optimal. This causes two major problems. Firstly, it is not very time efficient as we do not gain much information by learning from the same data. Secondly, as there is low diversity in the training data, the decision strategy would often change abruptly. This either leads to the neural network getting stuck in local minima or simply to diverge and not obtain a policy.
Picking a random subsample from memory helps increase the variance amongst the training data allowing the network to learn a more global policy.

4.2.6 Separating target from max Q estimators

The other method used to reduce the divergence of the predicted Q-values towards infinity was to reduce the correlation between the training and the prediction data [Mnih et al. 2015]. This is done because, since the networks are being fitted partly on the maximum Q-value estimation, diverging towards infinity reduces their error and therefore is a valid strategy that the networks can exploit. In order to reduce this correlation, the networks predicting the maximum Q-value and the ones predicting the Q-value can be separated. The weights of the maximum Q-value estimators are then updated every n steps in order for them to remain up to date.

Further modifications were also made on the neural network level in order to reduce these problems. We will further discuss these in section 4.4.
Once these modifications have been made, the basic training algorithm has the following form.

Algorithm 1: Basic DQN Training algorithm

function learn $(S, A, r, \gamma, \alpha, \phi, N, l\_freq, b\_size, \varepsilon, update\_nn\_freq)$;
Input :
$S \leftarrow states, A \leftarrow Actions, r \leftarrow reward,$
$\gamma \leftarrow discount\ factor, \alpha \leftarrow learning\ rate,$
$\phi \leftarrow$ set of feature functions over S and A
N $\leftarrow$ size of memory, l_freq$\leftarrow$ learning frequency,
b_size $\leftarrow$ batch size, $\varepsilon \leftarrow$ random action probability,
update_nn_freq $\leftarrow$ frequency of updating max Q estimators
Output:
$\theta \leftarrow$ trained weights of the estimator
$\theta =$ initialise random weights and learning rate α
$\theta_{max} = \theta$
$M =$ initialise an empty memory stack
for *each episode* **do**
  start from initial state $s_0 \in S$
  for $t = 0, 1, 2, ...$ **do**
    Initialise new memory item m_t
    With prob. ε: $a_t = rand(1, 36)$, with prob. $(1 - \varepsilon)$: $a_t = max_{arg}(Q(:, s_t))$
    observe next state s_{t+1} and $r_t(s_t, a_t, s_{t+1})$
    Set $m_t.a = a_t, m_t.s_1 = s_t, m_t.s_2 = s_{t+1}, m_t.r = r_t$
    if $M_{size} \geq N$ **then**
      del M_0
      $M_N = m_t$
    else
      push m_t on M
    if $t \mod l\_freq = 0$ **then**
      select a random batch of size b_size from M
      compute $Q(:, s_t)$ estimation with θ
      compute $Q(:, s_{t+1})$ estimation with θ_{max}
      set $Q(a, s_t) = Q(a, s_t) + \gamma * max(Q(:, s_{t+1}))$
      Fit weights θ with new training data
    if $t \mod update\_nn\_freq = 0$ **then**
      set $\theta_{max} = \theta$
  end
end
return θ

4.3 Deep Q-networks for faster Polyhedra analysis

Once the basic version of the deep Q-network algorithm was designed we were able to train it, test it and obtain our first set of results. The discussion of these will be done in chapter 5. These results can from now on be used as a baseline and we will attempt to further improve them.
In the search for a more efficient algorithm, we decided to modify the training algorithm with the use of some problem-specific knowledge, as well as further optimisations to the feature vector and reward functions. The modifications made will be discussed in the rest of this chapter.

4.3.1 Separating the problem into two

The first problem specific modification was dividing the problem into two independent subproblems. Fundamentally, the objective of Polyhedra Analysis is to obtain the most precise result in the quickest way possible. These are two independent objectives. Our goal was, therefore, to separate these two tasks and create two independent subsystems, one focusing on making the results as precise as possible, and the other on the time complexity of getting there.
Such a division of the problem offers two main advantages. Firstly, we obtain a greater flexibility during training. The parameters of each system can be tuned for its specific needs, for example, the characteristics of the neural network or the discount factor. Furthermore, the optimal features used for precision and performance estimation are most likely different. The separation, therefore, allows us to only use the necessary features for each subsystem making the learning and prediction less complex and therefore more precise and converge faster. The reward as well can be fine-tuned for the needs of the specific subsystem.
Secondly, two separate subsystems also allow for a more flexible algorithm post training. Since during training, we will have two different Q-function estimators, during prediction we will have a greater possibility of optimising our results by changing the importance we give to each subsystem at different points in time according to the specific needs. We will get into more detail about this in section 4.3.4.
The pseudo code for the training algorithm with separated estimators has the following form:

Algorithm 2: Separated DQN Training algorithm

function learn $(S, A, r, \gamma, \alpha, \phi, len, l\_freq, b\_size, \varepsilon, update\_nn\_freq, SA)$;

Input :
$S \leftarrow states, A \leftarrow Actions, r \leftarrow reward,$
$\gamma \leftarrow discount\ factor, \alpha \leftarrow learning\ rate,$
$\phi \leftarrow$ set of feature functions over S and A
len $\leftarrow$ size of memory, l_freq$\leftarrow$ learning frequency,
b_size $\leftarrow$ batch size, $\varepsilon \leftarrow$ random action probability,
update_nn_freq $\leftarrow$ frequency of updating max Q estimators,
SA $\leftarrow$ action selection algorithm

Output:
$\theta_1 \leftarrow$ trained weights of neural network for performance
$\theta_2 \leftarrow$ trained weights of neural network for precision

$\theta_1 =$ initialise random weights and learning rate α
$\theta_2 =$ initialise random weights and learning rate α
$\theta_{1\_max} = \theta_1$
$\theta_{2\_max} = \theta_2$
$M =$ initialise an empty memory stack
for *each episode* **do**
  start with initial states $s_{pr\_0} \in S, s_{pe\_0} \in S$
  for $t = 0, 1, 2, ...$ **do**
    Initialise new memory item m_t
    With prob. ε: $a_t = rand(1, 36)$, prob. $(1 - \varepsilon)$: $a_t = max_{arg}(SA(Q_{1,2}(:, s_t)))$
    Observe next state s_{pr_t+1}, s_{pe_t+1} and $r_{pe}(s_{pe\_t}, a_t, s_{pe\_t+1}), r_{pr}(s_{pr\_t}, a_t, s_{pr\_t+1})$
    Set $m_t.a = a_t, m_t.s_{pr\_1} = s_{pr\_t}, m_t.s_{pe\_1} = s_{pe\_t}, m_t.s_{pr\_2} = s_{pr\_t+1}, m_t.s_{pe\_2} = s_{pe\_t+1}, m_t.r_{pr} = r_{pr}, m_t.r_{pe} = r_{pe}$
    if $M_{size} \geq len$ **then**
      del m_0
      $M_{len} = m_t$
    else
      push m_t on M
    if $t \mod l\_freq = 0$ **then**
      select a random batch of size b_size from m
      compute $Q_{1,2}(:, s_t)$ estimation with θ_1, θ_2
      compute $Q_{1,2}(:, s_{t+1})$ estimation with $\theta_{1\_max}, \theta_{2\_max}$
      set $Q_{1,2}(a, s_t) = Q_{1,2}(a, s_t) + \gamma * max(Q_{1,2}(:, s_{t+1}))$
      Fit weights θ_1, θ_2 with new data from this batch
    if $t \mod update\_nn\_freq = 0$ **then**
      set $\theta_{1\_max} = \theta_1, \theta_{2\_max} = \theta_2$
  end
end
return θ_1, θ_2

4.3.2 Feature selection

The goal of our algorithm is to be able to choose the correct abstractions on the polyhedra so that their join will be efficient and precise. For this to be possible, the algorithm has to have a concrete knowledge about the polyhedra being analysed. It gets this knowledge through the feature vector. Therefore, in order to maximise the precision of our algorithm, we want to describe the polyhedra as closely as possible, so that our algorithm can make its decisions precisely. There are two main ways of increasing the precision of our feature vector, by adding new features or by making the existing features more precise. However, it is important not to expand this too much, or the algorithm will no longer be able to select the relevant information and the decision policy will diverge.
With regards to the old set of features. One big inconvenience of using Q-learning is that, since we want to represent our Q-function with basis functions, the size and dimensions of our feature vector are very limited. This inconvenience is a lot less important when using methods such as deep Q-networks, as the input to these can be continuous. This allowed us to make two major changes.

Adding new features

First of all, we reused the nine features from the Q-learning algorithm. As a reminder, these features are the following: the first seven features are used to characterise the complexity of the join. They are the number of blocks, minimal, maximal and average size of the blocks and the minimal, maximal and average size of the generator set. The last two features are used to characterise the precision of the inputs and they are the number of variables with a finite upper and lower bound, as well as the number of variables with a finite upper or lower bound, in both Polyhedra.
We further extended this set with four new features in order to further increase the description accuracy of the feature vector. The selection of these features was done by trial and error, with an experimental observation of an increase, or lack thereof, of accuracy. It is also possible to check whether a particular feature is useful once training is finished by analysing the neural network and observing the impact a particular feature has on the decision policy. However, since at the end our network has a total of four layers, this made its analysis somewhat complicated.
In the end, we decided to add a total of four new features, three of which are used for modelling the precision and one for the complexity. For the complexity, we added the number of variables. As for the precision, we added the number of unconstrained variables, the number of variables with a singleton interval and finally, the sum of the values the bounded variables can have (i.e. an approximation of the circumference of the polyhedron). We also attempted to use features that would have perhaps modelled the precision more accurately, such as most notably approximating the volume of the polyhedra. Unfortunately, one must also consider the complexity of computing the features. The computation of the volume approximation was far too time consuming, which greatly increased the learning time making the feature not viable.

Bucketing

Due to the limitations of Q-learning the old algorithm uses a very restrictive bucketing policy. Deep reinforcement learning does not suffer from such restrictions. It was, therefore, possible for us to remove the bucketing from the feature selection. We still decided to implement a version of bucketing inside of the algorithm for three main reasons.

- For features with very big values, an exact precision is not needed. For example, for the feature that approximates the circumference of the polyhedron, its total value can be very big and if it is a million and one or just a million does not impact the resulting precision much, therefore bucketing does not impact it either.
- Bucketing greatly decreases the learning time and helps convergence.
- Bucketing allows to control the impact of the different features on the decision policy. More precisely, we wanted to ensure that the said decision policy does not give more importance to some features simply because they are larger in value than others. In total, we have thirteen different features and they can all have very different values. For example, the number of blocks varies mainly between one and ten and the circumference of the polyhedra can go up to 10^9. This does not mean that the circumference has a greater impact on the overall description of the polyhedron. Bucketing assures that all features have a similar impact on the decision policy.

In order to decide the values of the bucketing that would be used, the algorithm was run on a big number of benchmarks computing the maximal, minimal and average values the different features could have. Finally, they were scaled in such a way, so that they would all approximately remain between the bounds of zero and ten.
We describe the final features in the following table. The first column describes what the particular feature represents. The second, the complexity of extracting it. The third, an approximate range of values the feature can have during a normal execution. Finally, the last column describes how we scale our features. $round(x/y, z)$ means that we first divide the feature by y and then round it to the nearest z.

Table 4.1: *Features used in the DQN algorithms*

Feature	Extraction complexity	Approximate range	Scaling
$\|\beta\|$	$O(1)$	1-10	x
$min(\|\chi_k\| : \chi_k \in \beta)$	$O(\|\beta\|)$	1-50	$round((x/5), 0.5)$
$max(\|\chi_k\| : \chi_k \in \beta)$	$O(\|\beta\|)$	1-50	$round((x/5), 0.5)$
$avg(\|\chi_k\| : \chi_k \in \beta)$	$O(\|\beta\|)$	1-50	$round((x/5), 0.5)$
$min(\|\bigcup G_{P_m}(\chi_k)\| : \chi_k \in \beta)$	$O(\|\beta\|)$	1-10000	$round((x/1000), 0.1)$
$max(\|\bigcup G_{P_m}(\chi_k)\| : \chi_k \in \beta)$	$O(\|\beta\|)$	1-10000	$round((x/1000), 0.1)$
$avg(\|\bigcup G_{P_m}(\chi_k)\| : \chi_k \in \beta)$	$O(\|\beta\|)$	1-10000	$round((x/1000), 0.1)$
$\{\|x_i \in X : x_i \in (-\infty, \infty)$ in $P_m\|\}$	$O(ng)$	1-100	$round((x/10), 0.5)$
$\{\|x_i \in X : x_i \in [l_m, \infty)$ in $P_m\|\}+$ $\{\|x_i \in X : x_i \in (-\infty, u_m]$ in $P_m\|\}$	$O(ng)$	1-50	$round((x/5), 0.5)$
$\{\|x_i \in X : x_i \in [l_m, u_m]$ in $P_m\|\}$	$O(ng)$	1-50	$round((x/5), 0.5)$
$\{\|x_i \in X : x_i \in [u_m, u_m]$ in $P_m\|\}$	$O(ng)$	1-50	$round((x/20), 0.2)$
$\|X\|$	$O(\|\beta\|)$	1-200	$round((x/5), 0.5)$
$\sum(u_m - l_m) : u_m, l_m \in \{[l_m, u_m]$ in $P_m\}$	$O(ng)$	$2 * 10^9$	$round((x/2 * 10^8), 0.01)$

One thing to note is that opposed to the reinforcement learning algorithm, these features do not have a maximal possible value. We believe that this increases the precision of the Q-function estimation, especially for the approximation of the complexity. Certain features can have very large values and we believe that when this happens, it has a major impact on the complexity of the join. However, these very large values were ignored by the bucketing of the Q-learning algorithm.

4.3.3 Reward Function modelling

Due to the nature of the new algorithm, the modelling of the reward function was also separated into two subproblems. Firstly, the reward for the precision estimator and then the reward for the performance estimator.

Performance reward

Modelling the complexity is fairly simple and very straightforward, as simply counting the number of cycles needed for the computer to execute the join perfectly models the joins complexity and is exactly what we want to optimise. One thing to note is that we want this reward to be as small as possible, two simple solutions for this are either to invert it or to negate it. After some experimentation with both of these, we decided that negating it produces better results.

This is probably due to the fact, that inverting the reward makes it have a nonlinear curve and its derivative loses importance the higher the CPU cycles are, which is not something we want to model. Another thing to note is that modelling the complexity reward in this way gives us another advantage over the reinforcement learning version of the algorithm. The Q-learning algorithm took the $\log_{10}$ off the CPU Cycles in order for the complexity and the precision reward to have similar values. We believe that this produces a similar problem as inverting the reward. The reward is no longer linear and its differences lose importance the higher it gets. Once again, this is not something that we want to model. The final rewards have the following form:

$$r_{pe}(s_t, a_t, s_{t+1}) = -1 \cdot cyc \tag{4.2}$$

Where cyc is the number of CPU cycles needed to perform the join.

Precision reward

As to the second reward, modelling the precision of the resulting join. This is considerably more difficult than modelling the complexity as there is no trivial element giving us the precision of our resulting polyhedron. In order to choose the reward, we proceeded by intuitively picking a small set of options and verifying them experimentally. In the end, we tested out three different rewards.
We took the first one from the reinforcement learning algorithm, that is, its objective is to maximise the number of variables with a singleton interval, bounded and half bounded variables. The second is a further extension by penalising the number of values a bounded variable can have in the resulting Polyhedron. The final reward function is slightly different. In this one, we penalise the loss of a variable with a singleton interval, as well as the loss of a bounded or half bounded variable and penalise the number of values a bounded variable can have. We take the logarithm of the amount of values a variable can have, as this values can be very big and we do not want it to have a higher influence on the reward than the other values. The final rewards have the following form:

$$r_{pr_1}(s_t, a_t, s_{t+1}) = 3 \cdot n_s + 2 \cdot n_b + n_{hb} \tag{4.3}$$

$$r_{pr_2}(s_t, a_t, s_{t+1}) = 3 \cdot n_s + 2 \cdot n_b + n_{hb} - \log_{10}(|n_b|) \tag{4.4}$$

$$r_{pr_3}(s_t, a_t, s_{t+1}) = 3 \cdot (n_{s_f} - n_{s_i}) + 2 \cdot (n_{b_f} - n_{b_i}) + (n_{hb_f} - n_{hb_i}) - \log_{10}(|n_b|) \tag{4.5}$$

Where n_s is the number of variables with singleton interval inside the resulting polyhedron after the join. n_b is the number of bounded variables and n_{hb} the number of semi-bounded variables of the resulting polyhedron. n_{x_f} is the number of variables after the join and n_{x_i} before the join.

4.3.4 Action selection algorithms

As discussed before, the Q-function estimation was separated into two independent subproblems. As previously described, this allows a certain amount of benefits that would not be possible otherwise. However, it also imposes one major complication. These two subproblems are

not totally separable, as once we have predicted the two Q-function estimations, we have to somehow merge the information contained in both of these predictions and pick an ideal action accordingly. We tested several different action selection algorithms in order to find the one that maximises precision and performance the most. One thing to note is that it is at least partially possible to test out these algorithms post training. Simply put, we do not have to train using these in order to measure their performance afterwards. However, this can only work if we train in a purely random manner. If we were to train using one selection algorithm and then test with another, this would surely deteriorate the results for the other algorithm. The fact that we are able to test the selection algorithms after random training is still very helpful as the training time is relatively high and having to train for each algorithm would be very time intensive.

Algorithm 1

The first selection algorithm we used is perhaps the most intuitive one. First, we scale both predictions, the performance one between [-1,0] and the precision one between [0,1]. We then add the values together and pick the action with the maximal value.
Whilst seeming fair this type of selection has a couple of fundamental flaws. First of all, reinforcement learning estimates the best action to take in order to maximise the long-term objective, however, it has less of a guarantee towards the second and third to best actions. This means that if the network is well trained, the chances that the action with the maximal Q-value prediction is also the best action to take at this point in time should be quite high. However, the guarantee that the action with the second highest Q-value prediction is the second-best action is much smaller. Using this selection, the algorithm frequently chooses not the best action but the second best or the third etc... thus reducing the probability that they are also good actions. The second problem with this type of selection algorithm is that scaling the reward function causes a potentially important loss of information. Let's demonstrate this with an example, let's say that we have four joins j1,j2,j3,j4. J1 takes one second, j2 ten seconds, j3 one hour and j4 ten hours. Let's also say that j1 is more precise than j2 and j3 is more precise than j4. This algorithm is going to treat the difference between j1 and j2 the same as the one between j3 and j4, even though the gain in precision might be worth the loss in performance for the case of j1 and j2, whilst this would probably not be the case for j3 and j4.

Algorithm 3: Action selection algorithm 1

1 **function** select1 (Q_{pr}, Q_{pe});
  Input :
    Q_{pr} ←array of Q-function estimations for precision,
    Q_{pr} ←array of Q-function estimations for performance
  Output:
    $a \leftarrow$ action to take
2 $Q_{pr} = Q_{pr}/max(Q_{pr})$
3 $Q_{pe} = Q_{pe}/min(Q_{pe})$
4 $a = max_{arg}(Q_{pr} + Q_{pe})$
5 **return** a

Algorithm 2

The second algorithm used, that we designed specifically to confront both problems of the previous algorithm, proceeds as follows. We set a threshold for the performance. We then look at the action that has the maximal Q-value for precision. If this actions Q-value for performance is above the threshold we take it, otherwise we take the second-best action and continue until we find an action that is above the threshold.
Once again, this algorithm seems quite good at first glance and maybe, if executed perfectly, it would be the best option. Nevertheless and much like the previous one, it has some fundamental problems. First of all, picking a threshold is not a very simple task. The predictions are done with regression neural networks, the activation function of the last layer is linear, this means that the output values are unbounded and they do not have a direct correlation with Real-time CPU cycles. In order to pick a threshold, we had to experimentally try different possible values and observe the resulting precision and adjust accordingly, however, this threshold would vary according to the benchmark and therefore choosing an optimal one was a very difficult task on its own. Another problem of this algorithm is that if no action has a Q-performance estimation above the threshold the algorithm will choose the action with the worst precision, and this one does not even have to have the best performance estimation.

Algorithm 4: Action selection algorithm 2

```
1 function select2 (Q_pr, Q_pe, PE_thresh);
      Input  :
               Q_pr ←array of Q-function estimations for precision,
               Q_pr ←array of Q-function estimations for performance,
               PE_thresh ←performance threshold
      Output:
               a ← action to take
2     while max(Q_pr) ≠ 0 do
3         a = max_arg(Q_pr)
4         if Q_pe(a) ≥ PE_thresh then
5             break;
6         Q_pr(a) = 0
7     end
8     return a
```

Algorithm 3

Another possibility is to slightly modify the previous algorithm in order to minimise its shortcomings. This time we set both a threshold for the performance and a certain number x. If the action that has the highest precision estimation is under the threshold for performance, we consider the x best actions according to their precision and take the one that has the highest performance estimation.
The problem of picking an appropriate threshold remains the same and this time we have the further problem of picking a correct value for x. However, the case that all actions are under the threshold is no longer a problem. It is worth noting that this algorithm also has a greater time complexity, however, this is still greatly outweighed if the correct action is taken.

Algorithm 5: Action selection algorithm 3

```
1  function select3 (Q_pr, Q_pe, PE_thresh, N_act);
       Input  :
                Q_pr ←array of Q-function estimations for precision,
                Q_pr ←array of Q-function estimations for performance,
                PE_thresh ←performance threshold,
                N_act ←number of actions to consider if below threshold
       Output:
                a ← action to take
2      a = max_arg(Q_pr)
3      if Q_pe(a) ≤ PE_thresh then
4          a_max = a
5          val_max = Q_pr(a)
6          Q_pr(a) = 0
7          for i ∈ N_act do
8              a = max_arg(Q_pr)
9              if Q_pe(a) ≥ val_max then
10                 a_max = a
11                 val_max = Q_pe(a)
12             Q_pr(a) = 0
13         end
14     return a_max
```

Algorithm 4

Finally, the last selection algorithm that we will introduce is based on the previous one but altered in order to reduce the importance of choosing a correct parameter. This time we begin by scaling the performance prediction to [-1,0]. We set a threshold between [0,1]. We pick the action that has the highest Q-precision estimation. If the absolute value of this action's performance estimation minus the maximal performance estimation is below the threshold, then we pick this action, otherwise we pick the second-best precision action until we find one that is below the threshold.
This time, since we compare to the best performance action we are guaranteed to be below the threshold at some point, therefore, in the worst case, we will pick the action with the best performance. The threshold is also a lot easier to set as it is bounded. Unfortunately, once again we have the problem caused by the scaling of the performance estimation. However, after some experimental observation, we observed that when the join is fast, all the Q-performance estimations tend to be quite close together, which hopefully renders the problem inconsequential.

Algorithm 6: Action selection algorithm 4

function select4 $(Q_{pr}, Q_{pe}, PE_{thresh})$;
Input :
Q_{pr} ←array of Q-function estimations for precision,
Q_{pr} ←array of Q-function estimations for performance,
PE_{thresh} ←performance threshold
Output:
$a \leftarrow$ action to take
$Q_{pe} = Q_{pe}/min(Q_{pe})$
while $True$ **do**
  $a = max_{arg}(Q_{pr})$
  if $Q_{pe}(a) + PE_{thresh} \geq max(Q_{pe})$ **then**
    break;
  $Q_{pr}(a) = 0$
end
return a

4.4 Neural network characteristics

As to the characteristics of the employed neural network. We started with a relatively small one and then grew it progressively as the number and size of the features were expanded. In the final version, we use a fully connected neural network with four hidden layers of two hundred nodes each. All of whom have thirteen inputs, one for each feature and thirty-six outputs, one for each action. The first three layers use the Relu activation function and the last one uses a linear activation since the rewards can be either negative or positive. We use stochastic gradient descent for updating the weights of the nodes and clip its norm to 1. We do this in order to avoid the exploding gradients problem and prevent the predicted values from diverging towards infinity. To calculate the loss, the mean squared error is used. The characteristics of the neural networks are based on other such projects, as these types of networks seem to be the most widely used ones for deep reinforcement learning. It is worth noting that we decided not to do much parameter optimisation on the neural networks as the training time for the algorithm is quite long which would make the parameter optimisation a very tedious task. In the end, we used the same network models for both performance and precision prediction as well as for the basic DQN algorithm.

4.5 Training

The algorithm was trained in multiple stages. During the first stage, only random actions were picked. This was done so that the algorithm would not get stuck in local minima but have the most global policy possible. We also did not have too large time constraints about the length of training so we could afford to proceed in this way in order to get the best possible results. Throughout the next stages, the training was separated into two. One concentrated

on the precision and the other on the performance. All the actions were no longer chosen at random, instead, the one with the highest predicted value was selected. For this, we used the first action selection algorithm and scaled the values with a factor of two depending on the corresponding goal we were focusing on. The probability with which we chose a random action was progressively decreased throughout training. Thus, reinforcing the already learned strategy and increasing the convergence speed. Once these stages were complete, we would have two separate systems each optimised for their own task. In the final stage, we would then combine both of the systems and do a final training phase with the chosen action selecting algorithm as described above. We did this in order to further optimise the decision policy for the particular action selection algorithm.
During training, we encountered two major complications that we dealt with as follows. The first of these was that, especially during random training, the likelihood of getting stuck on a join and not finishing the analysis of a benchmark was quite high. To confront this we set a timeout of thirty minutes on each benchmark. This was done in the aim of finding a compromise between having enough time to get to interesting joins and not wasting too much time in case the analysis gets stuck on a certain join. The other problem was that the analysis is very sensitive. That is to say, one bad action can have severe consequences on the end precision of the analysis. This means that training with a high probability of exploitation was essential for the estimators and had to be given enough time, but not too much so that we would avoid overfitting.
This method of training is quite time intensive. We decided to proceed in this way since time was not a big factor but the end results were more important. A more efficient training strategy can surely be found if this is needed.

Experimental Results

5

As described in the previous chapter, different strategies were elaborated for both the reward function and the action selection algorithm. Training all possible combinations and then observing the end precision would be a viable way of finding the optimal combination of these. However, due to the training and testing complexities of the algorithms, this would be very time intensive. Instead, we designed a set of experiments with the hope that they would be able to indicate which methods are more optimal.

Precision measuring

In order to measure the precision of the result, we observe the fraction of program points at which the Polyhedra invariants generated by our various algorithms is semantically the same as the one generated by ELINA using online decomposition.

Training/Testing set

The algorithms were trained on 328 different benchmarks following the steps mentioned in subsection 4.5. The testing set consists of 81 different benchmarks that were chosen part randomly and part due to their complexity. To avoid overfitting, the testing and training sets do not overlap.

5.1 Reward function selection

As a reminder, if we view the problem as a decision tree and the role of reinforcement learning is to pick the best path at each node, the reward function can be viewed as the compass of the reinforcement learning algorithm that should lead it to the end in such a way as to maximise the global goal.
We decided to measure the effectiveness of each reward function by observing if the direction it steered the algorithm to was indeed correctly maximising this global goal. To achieve this, we would first train an algorithm with each of the different reward functions. Once this was done, we modified the Q-learning version of the algorithm so that during testing it would measure each of the different reward functions. As a reminder, the Q-learning algorithm was trained

with the following reward function:

$$r(s_t, a_t, s_{t+1}) = 3 \cdot n_s + 2 \cdot n_b + n_{hb} - \log_{10}(cyc) \tag{5.1}$$

We then ran the Q-learning algorithm measuring each of the new reward functions during the execution. The decision of whether or not the reward function was good was then made with the following rule:
If the reward of the deep Q-network algorithm was higher than the one of the Q-learning algorithm, but its overall precision was not higher. This would imply that the deep Q-network was correctly learning to maximise the reward, but that this reward was not maximising the overall results of the algorithm and therefore it was not a good reward.
As a reminder, the rewards tested are the following:

$$r_{pr_1}(s_t, a_t, s_{t+1}) = 3 \cdot n_s + 2 \cdot n_b + n_{hb} \tag{5.2}$$

$$r_{pr_2}(s_t, a_t, s_{t+1}) = 3 \cdot n_s + 2 \cdot n_b + n_{hb} - \log_{10}(|n_b|) \tag{5.3}$$

$$r_{pr_3}(s_t, a_t, s_{t+1}) = 3 \cdot (n_{s_f} - n_{s_i}) + 2 \cdot (n_{b_f} - n_{b_i}) + (n_{hb_f} - n_{hb_i}) - \log_{10}(|n_b|) \tag{5.4}$$

We obtain the following reward for each join, during the execution of three different benchmarks:

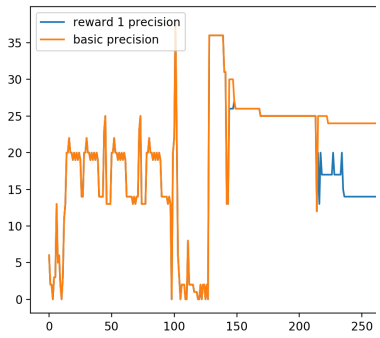

Figure 5.1: *Reward one*

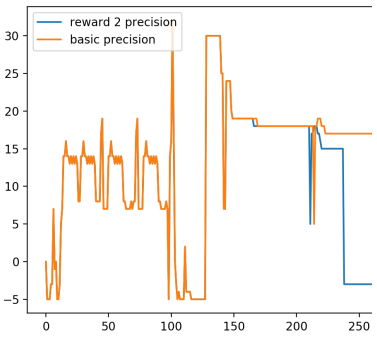

Figure 5.2: *Reward two*

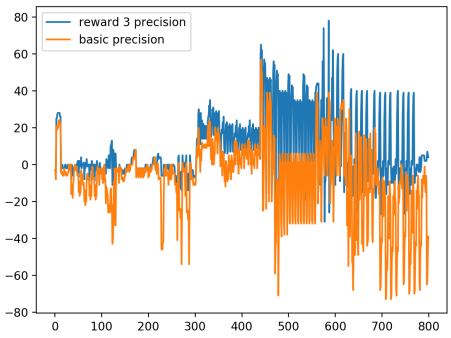

Figure 5.3: *Reward three*

The orange line represents the measured reward whilst running the Q-learning algorithm and the blue line the measured reward of the DQN trained with the corresponding reward. The overall precision of the end polyhedra is the following: For Figure 5.1. 99% for Q-learning and 96% for DQN. For figure 5.2. 99% for Q-learning and 94% for DQN. For figure 5.3. 97% for Q-learning and 90% for DQN. Whilst the graphs behave as expected in the first two scenarios, that is that the reward is lower for the algorithm with lower overall precision as well. In the last graph, we can see something unexpected. Whilst the DQN algorithm has a higher reward throughout most of the execution of the program with an average of 9 versus -8 for the Q-learning algorithm. This indicates that it should also have a better overall precision. However, its overall precision is lower. This tells us that whilst the DQN has learned a correct strategy for maximising the reward, this reward does not guide it to the end goal of maximising the final precision and therefore reward three is not an optimal reward function. These experiments do not give us much information on the differences between reward one and two as they both

behave as they should. This is to be expected as reward one and two are similar.
It is worth noting that whilst these experiments do give us some interesting information about the effectiveness of the different reward functions, the results are heavily dependant on the benchmark we run them on and therefore the results cannot be fully trusted. However, with the information gathered by these experiments, as well as some overall testing we decided to use the second reward function in further experiments.

5.2 Action selection algorithm

The second problem was selecting the best action selection algorithm. In section 4.3.4., four different action selection algorithms were proposed. In order to select the best one of these, we proceeded by training a DQN using each of them and then comparing the overall precision. Not all the training had to be done separately for the four algorithms. Firstly, we could train estimators from only random actions and then simply specialise each of them with their own action selection algorithm. We then compared the results on a set of seven different benchmarks.

Table 5.1: *Results of experiments with different action selection algorithms*

Benchmark	Q-learning	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
driver-media	99.0	98.6	97.6	93.2	98.0
linux-kernel-locking-spinlock	99.9	94.4	63.6	79.9	95.0
linux-usb-dev	58.2	51.8	60.3	56.5	59.3
linux-kernel-locking-mutex	77.1	73.2	95.5	95.6	76.1
complex-emg-drivers-net	57.1	97.3	94.6	96.0	96.9
complex-emg-drivers-media	57	50.1	58.9	55.4	54.9
complex-emg-linux-alloc-spinlock	44.9	72.2	69.9	91.8	72.0

As to the choice of benchmarks for this experiment. We had several criteria. First, we picked benchmarks where achieving a high level of accuracy was not too easy in order to have more informative results. Second, we picked benchmarks that were fast to compute so that the testing time would be fast. We also picked fast benchmarks because the goal was to find an algorithm that maximises the precision as each of these algorithms can be then optimised with their own parameters in order to ameliorate performance and in this way we only had to concentrate on a single goal.
As to the results of these experiments. Unfortunately, there is no overall best algorithm. As discussed in section 4.3.4., each of these algorithms has its advantages and shortcomings. However, the main objective we were looking for from these experiments was finding an algorithm that would be stable. We can see that algorithms two and three can obtain very good results on some benchmarks. Unfortunately, their performance is far from optimal on others. This is due to the fact that both of these are strongly threshold dependant and picking an optimal threshold

that would work well for all benchmarks is very difficult, making both of these algorithms not practical. Afterwards, between the first and fourth algorithm, the last one outperforms the first on average and therefore is the one we will use from here on. Whilst the last algorithm does not have the best precision on some benchmarks, its stability overall benchmarks made it the preferred candidate.

5.3 Final algorithm

Once both decisions were made, the architecture of the final separated algorithm was fully decided and could, therefore, be further trained and optimised.
For the testing of the different algorithms, we decided to run them on a large set of benchmarks. In this way, we could see if they managed to learn a global policy or simply find local minima. In the end, we settled on a set of 81 different benchmarks of varying sizes and complexities. Due to the relatively large number of benchmarks, we set the timeout to be quite short, more specifically thirty minutes. This was done so that the experiments could be run in a reasonable amount of time. For each benchmark, we saved the resulting invariants of the analysis and the runtime.

We test the benchmarks using four different analyses:

- Using the decomposed Polyhedra domain, to have a baseline for precision.
- RL trained with Q-learning
- Basic deep Q-network algorithm
- Deep Q-network with separated estimators

The following table displays the results of the experiments. It shows the average of the precision percentages when compared to the abstraction-less analysis. The total runtime on all benchmarks, the number of benchmarks that the analysis timed out on. The number of benchmarks where the algorithm achieved 100% precision. Finally, for the reinforcement learning algorithms, the number of benchmarks where they outperform the other RL algorithms with regards to precision.

Table 5.2: *Results with final algorithms*

Results on 81 bench.	Elina	Q-learning	DQN	Separated DQN
Avg. prec. (%)	100	87.1	94.9	91.6
Runtime (hr.)	18:28	14:05	15:35	11:21
n° timeouts	33	23	23	17
n° bench. 100% prec.	81	13	27	18
n° bench. most precise		1	36	2

Result discussion

As shown in the results, we were able to outperform the Q-learning method with both the deep Q-network and the separated deep Q-network algorithms with regards to accuracy. What is interesting to note, is that the regular DQN outperforms the separated one in terms of precision. This can most likely be explained by the action selection algorithm. Even though in 5.2., we attempted to pick an algorithm that would be as stable as possible and be able to maintain good results overall benchmarks. The results show that on a few benchmarks the separated DQN performs very poorly, thus reducing its overall precision. However, it is worth noting that the separated DQN outperforms all the other algorithms in terms of performance, as it manages to finish the analysis of all the benchmarks quite a lot faster than the others, most notably, it is faster than the Q-learning algorithm whilst remaining more precise as well. The separated DQN also has the least timeouts out of all the analyses. In terms of precision, however, the normal DQN algorithm is the most precise on average overall benchmarks. It is also more precise on 36 of the 81 benchmarks and is only beaten on one benchmark by Q-learning and on two by the separated DQN, the precision remains the same on the rest. It also achieves 100% precision on more than double the number of benchmarks than the Q-learning method does.

Conclusion

In this work, we managed to build a framework for the training and testing of various deep reinforcement learning algorithms inside polyhedra analysis. We could then design a series of such algorithms. These algorithms ranged from very global reinforcement learning methods to ones more optimised with problem-specific knowledge as well as having a greater flexibility towards the various subproblems. We then tested out the different subparts of our algorithms we had come up with, in the goal of finding an optimal combination.
Once our algorithms were designed, we could then test them on a broad array of different benchmarks. They outperformed other preexisting reinforcement learning methods on both accuracy and performance.
It is also worth noting that we highlighted the overall effectiveness of the global deep Q-network algorithm, as after all, it was able to craft a decision policy that was the most accurate. Even with the design of new training algorithms that used more domain-specific knowledge, we could only increase the performance at the loss of some precision.
Future research for this work could be done in several directions. Techniques such as CNN's could be investigated for automatic feature extraction from the polyhedra itself. Other works such as [Dyer et al. 1991, Kim et al. 2004] could also be used for the approximation of the volume of the polyhedra. Both techniques could then be simultaneously used for expanding the features as well as the reward for precision. Other than that, optimisations to the neural network itself could be undertaken as this was not explored much in this work. An expansion of the action set, most notably the size and number of different thresholds, perhaps even using a continuous action set for the size of the threshold could increase the precision. Finally, separating the features, NN's and making them more task-specific, would increase the overall performance.

Bibliography

CHEN, L., MINÉ, A., AND COUSOT, P. 2008. A sound floating-point polyhedra abstract domain. In *Asian Symposium on Programming Languages and Systems*, Springer, 3–18.

CHERNIKOVA, N. 1968. Algorithm for discovering the set of all the solutions of a linear programming problem. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki 8*, 6, 1387–1395.

COUSOT, P., AND COUSOT, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM, 238–252.

DYER, M., FRIEZE, A., AND KANNAN, R. 1991. A random polynomial-time algorithm for approximating the volume of convex bodies. *Journal of the ACM (JACM) 38*, 1, 1–17.

GANGE, G., NAVAS, J. A., SCHACHTE, P., SØNDERGAARD, H., AND STUCKEY, P. J. 2016. Exploiting sparsity in difference-bound matrices. In *International Static Analysis Symposium*, Springer, 189–211.

HEO, K., OH, H., AND YANG, H. 2016. Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In *International Static Analysis Symposium*, Springer, 237–256.

JOURDAN, J.-H. 2017. Sparsity preserving algorithms for octagons. *Electronic Notes in Theoretical Computer Science 331*, 57–70.

KAELBLING, L. P., LITTMAN, M. L., AND MOORE, A. W. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research 4*, 237–285.

KIM, Y. J., VARADHAN, G., LIN, M. C., AND MANOCHA, D. 2004. Fast swept volume approximation of complex polyhedral models. *Computer-Aided Design 36*, 11, 1013–1027.

LIANG, P., TRIPP, O., AND NAIK, M. 2011. Learning minimal abstractions. In *ACM SIGPLAN Notices*, vol. 46, ACM, 31–42.

LOGOZZO, F., AND FÄHNDRICH, M. 2010. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. *Science of Computer Programming 75*, 9, 796–807.

MARÉCHAL, A., AND PÉRIN, M. 2017. Efficient elimination of redundancies in polyhedra by raytracing. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, Springer, 367–385.

MINÉ, A. 2001. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects*. Springer, 155–172.

MINÉ, A. 2006. The octagon abstract domain. *Higher-order and symbolic computation 19*, 1, 31–100.

MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., PETERSEN, S., BEATTIE, C., SADIK, A., ANTONOGLOU, I., KING, H., KUMARAN, D., WIERSTRA, D., LEGG, S., AND HASSABIS, D. 2015. Human-level control through deep reinforcement learning. *Nature 518* (Feb), 529 EP –.

MOTZKIN, T. S. 1953. The double description method, in contributions to the theory of games ii. *Annals of Mathematics Study 28*.

OH, H., LEE, W., HEO, K., YANG, H., AND YI, K. 2014. Selective context-sensitivity guided by impact pre-analysis. *ACM SIGPLAN Notices 49*, 6, 475–484.

OH, H., YANG, H., AND YI, K. 2015. Learning a strategy for adapting a program analysis via bayesian optimisation. In *ACM SIGPLAN Notices*, vol. 50, ACM, 572–588.

RIEDMILLER, M. 2005. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, Springer, 317–328.

SINGH, G., PÜSCHEL, M., AND VECHEV, M. 2015. Making numerical program analysis fast. In *ACM SIGPLAN Notices*, vol. 50, ACM, 303–313.

SINGH, G., PÜSCHEL, M., AND VECHEV, M. 2017. Fast polyhedra abstract domain. In *ACM SIGPLAN Notices*, vol. 52, ACM, 46–59.

SINGH, G., PÜSCHEL, M., AND VECHEV, M. 2018. Fast numerical program analysis with reinforcement learning. In *International Conference on Computer Aided Verification*, Springer, 211–229.

STOER, M., AND WAGNER, F. 1997. A simple min-cut algorithm. *Journal of the ACM (JACM) 44*, 4, 585–591.

WATKINS, C. J., AND DAYAN, P. 1992. Q-learning. *Machine learning 8*, 3-4, 279–292.

WIKIPEDIA CONTRIBUTORS, 2018. Reinforcement learning — Wikipedia, the free encyclopedia. [Online; accessed 28-November-2018].

YOGESWARAN, M., AND PONNAMBALAM, S. 2012. Reinforcement learning: exploration–exploitation dilemma in multi-agent foraging task. *Opsearch 49*, 3, 223–236.