Fast Polyhedra Analysis with deep Q-networks

Jakub Kotal

Bachelor Thesis
Novemeber 2018

Supervisor:
Prof. Dr. Otmar Hilliges, Prof. Dr. Martin Vechev

ETH zürich

Advanced Interactive Technologies

Abstract

Acknowledgment

I would like to thank Prof. Dr. Otmar Hilliges for supervising this project. Thanks to him I had the opportunity to get to know many fields in computer science that were unfamiliar to me half a year ago. The last semester was a very challenging and rewarding one. His profound knowledge in many areas was very helpful to me.

Furthermore i would like to express my deep appreciation and gratitude to Christine Ryer-Bringold, my former music teacher. She supported me generously with many hours of supportive advices and conversations about education and music.

Someone else who deserves my deep gratitude is my mother. The innumerable sacrifices she made to support me whenever she could are highly appreciated and I cannot say how thankful I am for everything she did.

Contents

List of Figures

List of Tables

1 Introduction

As technology becomes ever more present in the modern age and tasks become more and more optimised, the more the structures in our lives become controlled by programs. From simple things as automatic doors, too self-driving metros, trains and even cars, to medical software, aviation software and even nuclear weapon software. The more we wish to automise and simplify our lives the bigger will be the power that we will put into the hands of computers and the programs that run on them. As these programs get bigger, writing them becomes more complicated they become longer and the risk of them containing errors increases. However, the safety and invulnerability of some of these systems is critical and needs to be verifiable. Static analysis is a sub-branch of computer science tasked with the analysis of computer programs without actually executing them. It has seen a growing commercial use in the past years in some of these safety-critical domains.

1.1. Problem Statement

Unfortunately, the complexity and size of the programs being used today has grown drastically and the design of a static analyser that can keep up with this growth is not a simple task. Many techniques exist that exploit specifics about the analyses that mange to increase its precision. There are also different types of techniques that try to leverage precision loss against performance gain. However, finding the right balance between these two is not an easy task.

1.2. Goals

The goal of this thesis is to incorporate advanced reinforcement learning techniques inside polyhedra analysis. The goal is then to optimise these methods in order to render the analysis more efficient and outperform other analysis methods.

1.3. Structure of this Document

.

The remaining section of this chapter will introduce some conventions made for the sake of readability and briefness of the complete document.

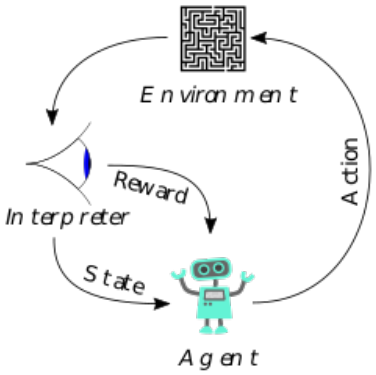
Chapter 2 will describe reinforcement learning and talk about the recent advances in this field. I will as well describe deep Q-networks and some of areas that they have been used.

Chapter 3 addresses Polyhedra analysis and some of the methods used to make it more efficient.

Chapter 4 Will address the algorithm designed and how the two previous chapters were combined, in order to achieve polyhedra analysis with deep Q-networks.

2 Reinforcement Learning

Reinforcement Learning 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 well-defined concept and the foundations are the following. An agent interacts with a given environment, at its disposal, it has a set of various different actions. When an action is performed it receives a reward. The goal of the agent is to devise an action selection strategy that should maximise the cumulative reward of the actions.



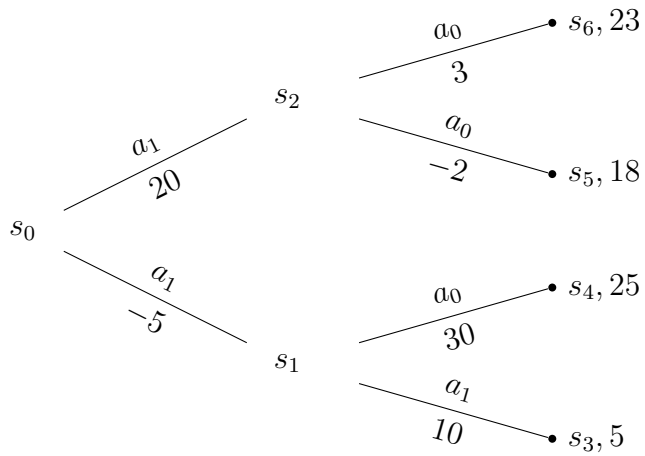
2.1. Concepts

In order to be able to solve a porblem with reinforcement learning, this problem 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 action 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 $a_t \in A$ then the action will be executed the agent will move from s_t to s_{t+1} and receive the reward $r(s_t, a_t, s_{t+1})$. This process will be repeated until a final state is reached. 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 picking strategy so as to maximise the cumulative reward. I will demonstrate the idea of the cumulative reward with the following example.

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



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-value.
Unfortunately, as in most real world cases the state space is very large or even infinite. Computing the Q-function exactly is very close to infeasible.

2.3. Q-function approximation

The goal of reinforcement learning is to obtain the best possible Q-function approximation. To achieve this, the agent is allowed to interact freely inside of the environment, picking actions randomly or with the till now learned policy. It observes the different results it obtains and updates its policy accordingly.
I shall now talk about a few concepts used during training.

Exploration-exploitation
As mentioned above, during training we either pick the best action with the till-now learned

policy or we just pick a random action. The trick is in finding a correct balance between the both of these as both of these have their advantages. If we always pick the best action we will converge quickly but the risk of getting stuck in a 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 find a strategy at all. Most training algorithms start with a high probability of exploration and then as training progresses increase the exploitation probability as to solidify the learned policy.

Learning rate

The learning 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 a already learned policy.

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 feature reward. If $\gamma = 0$ then only the immediate reward will be considered and if $\gamma \approx 1$ then we will look for a policy that will maximise the cumulative reward. The same as before one has to choose between a faster convergence and a higher risk of getting stuck in local minima.

2.3.1. Q-learning

Q-learning is one of the main algorithms used in reinforcement learning in order to approximate the Q-function. It modelles the Q-function as a set of basis function where each basis function assigns a value to a (state,action) pair and each future has its own basis function. Q-learning offers several advantages. It is efficient and converges relatively quickly. The learned policy can be quite easily interpreted. By definition, Q-learning uses a linear function approximation. In most casses this should not cause much of a problem. But, in the case where the ideal decision policy is non-linear, Q-learning will never achieve optimal results.

2.3.2. Deep Q-networks

Deep Q-networks are a novel method used for reinforcement learning that has gained a lot of attention in the recent years. The reason why it has gained so much notice lately, is due to the ability of a single algorithm being able to achieve very good results on a broad array of tasks without the need for any specialized knowledge about the task beforehand, leading some to call

it as the first major steps towards general artificial intelligence.
What separates deep Q-networks from other reinforcement learning methods, is that they use neural network in order to approximate the Q-function. Neural networks are non-linear function, reinforcement learning has been known to diverge when non-linear function approximators have been used. However, the development of new technique such as experience replay memory has allowed them to become a viable tool for reinforcement learning and to achieve some very remarkable results, most notably in domains such as video games.

3 Polyhedra Analysis

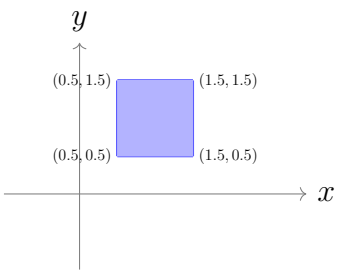
Polyhedra analysis is one of the main tools of static analysis. During the execution of a program variables can become bounded, either by numbers or by other variables. Polyhedra are one of the most expressive ways of modelling all the possible values that variables can have as well as the different dependencies between them. Different alternatives exist for constraint representation instead of polyhedra, but the polyhedra domain is by far the most expressive. Unfortunately, it has worst case exponential space and time complexity meaning that using it on most real-world applications would cause it to either timeout or to run out of memory making it very impractical. Therefore, other domains have been more widely used such as octagon, zone or pentagon, but all of these are less expressive and therefore less precise by design. In the recent years several techniques have been developed that have managed to speed up polyhedra analysis without the loss of precision.

3.1. Polyhedra representation

One of these techniques involves separating the way we represent our polyhedra. Polyhedra can be represented with both their constraint representation and their generator representation. To illustrate these different illustrations I will proceed with the following example. Lets assume we have the following set of assertions:

$$assert(y \leq 1.5)$$
$$assert(y \geq 0.5)$$
$$assert(x \leq 1.5)$$
$$assert(x \geq 0.5)$$

and their respective polyhedron would have the following shape:



We can represent this information in two different possible ways.

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 subspace. The resulting polyhedron can be written as:

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

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

$$\{-x \leq -0.5, x \leq 1.5, -y \leq -0.5, y \leq 1.5\}$$

3.1.2. Generator representation

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

- Set of vertices $V \in Q^n$.
- Set off rays where one end is bounded and that start from a vertex, modelling the edges of the polyhedron.
- set off lines modelling the infinite edges of the polyhedron with both ends unbounded.

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

3.2. Polyhedra domain

Now that we can represent our Polyhedra we can do some interesting calculations with them. The polyhedra abstract domain consists of the polyhedral lattice: $(P, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, and a set of operators that we can apply onto the different polyhedra. The different operators are the following:

- Inclusion test: $P \sqsubseteq Q$
- Equality test: $P = Q$
- Join: $P \sqcup Q$
- Meet: $P \sqcap Q$
- 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 $C \vdash c$, test 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
- Assignement: $x_i = \delta$, 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

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 ($\bigtriangledown$)	$O(mLP(m,n))$	$O(gLP(g,n))$	$O(ngm)$
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 faster than the other. As some operators are quicker in one but others in the other. But, as we can see in the last table, when both representations are available all operators are polynomial.

Chernikova's Algorithm

The first optimisation that one can do is keep both representation of the the polyhedron and for each operator picking the representation that minimises the time complexity. A conversion between the two representations is possible thanks to Chernikova's algorithm.

3.3. Polyhedra Decomposition

Another technique used increase the efficiency of polyhedra analysis, is that of online decomposition. It is based on the observation that during the execution of a program, not all it's variables are dependent on one another. Using this observation, we can separate the set of all variables into independent sets. Therefore, instead of having to represent 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 iff $\chi_i \cap \chi_j = \emptyset$,

$\forall i \neq j$. 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 are influenced by it. This allows for a very large performance gain. Giving us the following time complexity for the various operators.

Table 2 Asymptotic time complexity of Polyhedra domain operators 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 2.1
I'll illustrate online decomposition with the following example.

3.3.1. Decomposition operators

As we change the model of our domains we must equally update the operators inside of these domains. For sake of brevity I will not go into detail about each of them, but I will only talk about the join operator as it plays an important role in the work of this paper.
During the join of P and Q, most of the time their factors will not be equal($\pi_P \neq \pi_Q$). Therefore, we have to remake their partitions in order for their factors to be equal $\pi = \pi_P \sqcup \pi_Q$. During most joins of a normal execution this will not cause much problem, but during some cases the join can merge all blocks producing the $\top$ partition. Causing all the performance benefits of decomposition to be lost.

Example 2.2

3.4. Reinforcement Learning for polyhedra analysis

Both of the techniques presented in chapters 2.3 and 2.2, manage to achieve considerable performance gain without having to sacrifice any precision. Unfortunately, at some point compromises have to be made. Many different algorithms have been proposed that try their best at minimising the precision loss and maximising the resulting performance gains.

As shown in example 2.2 one bad constraint can significantly decrease the performance of the whole program. The trick is being able to identify this variable at the correct time. One of the potential solutions to this problem that we shall explore in this paper is training a reinforcement learning algorithm in order to decide when to apply abstractions.

3.4.1. Adapting polyhedra analysis for Reinforcement Learning

In order to be able to apply reinforcement learning, we first have to express polyhedra analysis in terms of reinforcement learning. That is, we need an agent, a set of actions, a reward and a set of features.
Conceptually the problem is quite simple. We will have our agent, the static analyser, executing the analysis of a program. During analysis it will have a set of actions at its disposal. Each of these actions will have a varying performance/precision capability. The goal of the agent will be to choose the correct degree of abstraction at the correct point in time. The reward has to be some sort of compromise between the precision and the performance of the given action.

3.4.2. Existing methods

Existing methods already exploit this idea. Using techniques such as Q-learning in order to create a decision policy. Q-learning is a common linear Q-function approximation technique. However recent RL methods concentrate on the use of non-linear Q-function approximation such as neural networks. Whilst non-linear Q-function approximations have been known to be unstable or divergent in the past, development of new techniques such as experience replay has helped make them a viable tool for a series of reinforcement learning problems. In the following chapters, we shall explore the idea of using such a technique for polyhedra analysis.

Fast polyhedra analysis with deep Q-networks

4

The goal of my 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, my goal was to extend the Elina Framework, the version of it that used reinforcement learning for polyhedra analysis, in order to use deep Q networks for the Q function estimation. The work I have done can be separated into three major subtasks. First of all, I had to figure out a way of calling a deep neural network form Elina, so that I could use it for the Q function estimation. Secondly, the most interesting part was writing the training algorithm. This included choosing an adequate feature vector, modifying the reward function and testing various action selection policies. Finally, once my training algorithm was written and the link to Elina created, I could train, optimise and last of all test my algorithm.

4.1. Incorporating Neural Networks inside Elina

The first part, whilst definitely being the least interesting was probably the most laborious and I very key part of the whole work. The problem was that for my Q function estimation I wanted to use neural network regression from Keras in Python. I decided to use 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, I somehow had to make the link between my Python code and the C code of Elina. I finally achieved this by using the Python/C API and embedding the python code into the Elina framework. Initialising all the python code as well as importing all the necessary libraries such as Keras, is done in the $op\_pk\_manager\_alloc()$ function so that it doesn't have to be done every time we call the join function. Inside the join function we simply have to call the learning or prediction algorithms.

4.2. Training Algorithm

Once this has been done, I could finally proceed to the more interesting part of writing the training algorithm. I experimented with various different training algorithms in order to obtain the best possible results, the last of which I will describe in the following paragraphs. First of all, I shall describe a broad outline of the different steps taken and afterwards I will get into more detail about the specific feature selection, reward function modelling and action selection policy.

4.2.1. Separating the problem into two

I decided to separate the problem into two independent subproblems. The objective of Polyhedra Analysis is to obtain the most precise result in the quickest way possible. Therefore, I separated these two objectives with two different q networks, one estimating the precision of the result and the other the time complexity of getting there. There are two main advantages to dividing the problem in such a way. Firstly, the features estimating precision and the ones estimating time complexity may very well be different. The separation therefore allows us to use only the necessary features for each subsystem making the learning and prediction less complex and therefore more precise and converge faster. Secondly, two separate subsystems allow for a more flexible algorithm post training. Since during training, we will have trained two different Q function estimators, during prediction we will have a greater possibility of optimizing our results by changing the importance we give to each subsystem at different points in time. I will get into more detail about this in the section about the action selection policy.

4.2.2. Problems

During the testing of the different training algorithms I ran into two major problems. The first of which was the divergence of the action selection strategy. Meaning that neural networks were not able of creating a consistent strategy but would be picking different actions all the time. The second of which was the divergence of the predicted Q towards infinity. In order to combat these problems, I used a combination of different methods.

4.2.3. Action replay memory

The first of which, was using an experience replay memory. During training, an array of the last 1000 memory objects is kept. Each memory object stores the following data: the current state features for both time complexity and precision, the action taken, the respective reward for both time complexity and precision after this action, and finally the next state features for both precision and time complexity once the action has been taken. Every n joins, we pick a random subsample of fifty memory objects from the array, compute the targets and refit the neural networks.
The objective of this is to homogenize the training data. I observed that during the execution of a program, often a particular strategy would be optimal during a certain period of time and afterwards another one would be the new optimal. This caused two major problems. First of all, it is not very time efficient as we do not gain much information by learning from the same data. Secondly, as there was low diversity in the training data and the decision strategy would frequently change it either led to the neural network getting stuck in local minima or simply to diverge and not obtain a global policy.
Picking a random subsample from the last 1000 joins helped to increase the variance amongst the training data allowing the network to learn a more global policy. The fitting of the neural networks only begins after a certain number of joins has been done already. I decided to do this in order to decrease the training time and not give a greater probability of training on the

first joins of a program, since the polyhedra are mostly relatively small at the beginning and therefore not the most interesting ones to learn from.

4.2.4. Q-estimator separation

Another modification I made, in order to reduce the divergence of the predicted Q values towards infinity, was to reduce the correlation between the prediction of the maximum Q value and the Q value prediction. I did this because, since the networks were being fitted based on the maximum Q value estimation, diverging towards infinity reduced their error and therefore was a valid strategy that they would use. In order to reduce this correlation, I separated the networks predicting the maximum Q value and the ones predicting the Q value. I would only update the weights of the networks predicting the maximum values every two hundred and fifty steps, so that they would still remain up to date. I also made some modifications on the neural network level to reduce this problem which I will get into inside the neural network characteristics part.

Pseudo code of the training algorithm

Algorithm 1: DQN Training algorithm

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

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
len $\leftarrow$ size of memory, l_freq$\leftarrow$ learning frequency,
b_size $\leftarrow$ batch size,
update_nn_freq $\leftarrow$ frequency of updating max Q estimators

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
$\theta_2 =$ initialise random weights
$\theta_{1\_max} =$ initialise random weights
$\theta_{2\_max} =$ initialise random weights
$m =$ initialise an empty memory
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
    Take action a_t according to the action selection algorithm
    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 $Q(:, s_t)$ estimation with θ_1, θ_2
      compute $Q(:, s_{t+1})$ estimation with $\theta_{1\_max}, \theta_{2\_max}$
      set $Q(a, s_t) = Q(a, s_t) + \gamma * max(Q(:, s_{t+1}))$
      Fit weights θ_1, θ_2 with new computations 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. Actions

As for the actions, I used the same actions as were already implemented inside of the reinforcement learning algorithm. As we discussed in 3.3.1., the bottleneck of the decomposed analysis is the join. Therefore, our set of actions will be a various set of joins of different precision and performance.
First of all, the cost of the joins depends on the size of the block. Therefore, bounding the the size of the block 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 equally decide on what to do if the block has a greater size than the threshold. Their are different possibilities of how to split a large block into smaller ones, but basically, one has to pick a subset of constraints to remove from the block so that all its constraints are no longer dependant and then it can be further partitioned. The following three constraint removals are used:

Stoer-Wagner min-cut

This uses the basic idea of removing the minimal amount of constraints in order to be able to split to block into two separate permissible partitions.

Weighted constraint removal

The second and third constraint removal techniques are based on the same principal. They associate a certain weight to each constraint and then remove the constraint with the highest weight. Two different weight distribution techniques are considered.
In the first one, we first 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.
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 constraints x_i, x_j contained in the constraint.
The idea of removing the constraint with maximal weight, is that, most likely the variables occurring in the constraint are also bounded by other constraints and therefore will not become unbounded once the constraint is removed.

Merging blocks

The next three actions are various block merging strategies. The idea is to select different blocks and merge them together as long as the resulting blocks remain below the threshold 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 me merge the small-

est 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 $4 \times 3 \times 3 = 36$ different actions we can pick.

4.4. Feature selection

With regards to the features I decided to use for the q function estimation. As a reminder, the objective of the features is to make possible the estimation of the q function. Therefore, in this particular case, the objective is to find features about the polyhedra that would help in the prediction of the precision and time complexity of their resulting join. The first nine features I used are the same as the reinforcement learning version of my algorithm and they remained unchanged in my version of the algorithm except for a change in their bucketing, to which I will get later. The first seven features of these features are used to characterize 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 characterize 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.

One of the advantages of using deep reinforcement learning over reinforcement learning, is that we can increase the number of features without greatly increasing the complexity of the training. This allowed me to add four new features to the set, in order to further increase the accuracy of the predictions. The selection of these features was done by trial and error, with an experimental observation of an increase, or lack thereof, off accuracy. It is also possible to check whether a particular feature is useful once training it finished by analyzing the neural network and observing the impact a particular feature has on the end result. However, since at the end my network had a total of four layers, this made its analysis somewhat complicated.

At the end I added a total of four new features, three of which are used for modelling the precision and one for the complexity. For the complexity I added the number of constraints. As for the precision I added the number of unconstrained variables, the number of exactly constrained variables and finally total amount of values all the bounded constraints can have (i.e. an approximation of the circumference of the polyhedra). I also tried using some 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 complex, which greatly increased the learning time making the feature not viable.

Another advantage of deep reinforcement learning is that the Q function is approximated with a neural network and not a matrix. The problem with the matrix representation is that increasing the number and size of features greatly increases the dimensions of the matrix. Causing quite large constraints on the size of the future vector that one can use. Deep reinforcement learning does not suffer from this. Because of this it was possible for me to remove the very constrictive bucketing method of the reinforcement learning algorithm. I still implemented a version of bucketing in my algorithm for three main reasons.

Firstly, different levels are necessary for different features. For example, for the feature that

approximates the circumference of the polyhedra, its total value can be very big and and if it is a million and one or just a million doesn't impact the resulting precision much therefore bucketing makes sense.
The second reason, for the use of bucketing is that it greatly decreases the learning time and helps convergence.
Finally, the last reason is that the decision policy doesn't give more importance to some features simply because they are greater than others. In my total of thirteen different features, they all can have very different values. For example, the number of blocks varies mainly between zero 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 precision of the polyhedra. Bucketing assures that all features have a similar impact on the decision policy.
In order to decide the values of the bucketing I would use, I ran my algorithm and a big number of benchmarks computing the maximal, minimal and average values the different features could have. Finally, I scaled them so that they would all approximately remain between the bounds of zero and ten. And bucketed in the following fashion:

Feature	Extraction complexity	Approximate range	Scaling
$\|\beta\|$	$O(1)$	1-10	$x/1.$
$min(\|\chi_k\| : \chi_k \in \beta)$	$O(\|\beta\|)$	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 $P_m\|\}$	$O(1)$	1-200	$round((x/20), 0.2)$
$\|X\|$	$O(ng)$	1-50	$round((x/5), 0.5)$
$\sum(u_m - l_m) : u_m, l_m \in \{[l_m, u_m]$ in $P_m\}$	$O(ng)$	$2 * 10^9$	$round((x/2 * 10^8), 0.01)$

One thing to note is, that as opposed to the reinforcement learning algorithm, my features do not have a maximal possible value. I 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 I believe when this arrives, it has a major impact on the complexity of the join. However, these very large values were ignored by the bucketing of the reinforcement learning algorithm.

4.5. Reward Function modelling

Due to the nature of my algorithm, the modelling of the reward function was separated into two separate subproblems. Firstly, the modelling of the complexity reward and then the modelling of the precision reward. Modelling the complexity is fairly simple and very straight forward, as it calculating the number of cycles needed for the computer to do the join perfectly models the complexity and is exactly what we want to improve on. 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. I experimented with both of these and found that negating it produces better results. I assume this is due to the fact that inverting the reward, make it have a nonlinear curve and derivative loses importance the higher the CPU cycles are, which is not something we want. Another thing to note is that modelling the complexity reward in this way give us another advantage over the reinforcement learning version of the algorithm. The reinforcement learning algorithm took the logarithm of 10 of the CPU Cycles in order to have both the complexity and the precision reward to have similar values. I believe that this produces a similar problem as when we inverse the reward. The reward no longer linear and its differences loose importance the higher it gets. However, this is not something that we want to model.
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 polyhedron. In order to choose the reward, I proceeded in intuitively picking a small set of options and verifying them experimentally at the end. I took the first one from the reinforcement learning algorithm, that is, the objective is to maximize the amount of exactly bounded, bounded and half bounded variables. The second is a direct extension of the first but penalizing the number of unbounded variables. The third is a further extension by penalizing the amount of values a bounded variable can have in the resulting Polyhedra. The final, reward function is slightly different. In this one, I penalize the loss of an exactly bounded, bounded or half bounded variable. Reward the loss of a bounded variable and penalize the amount of values a bounded variable can have. I also tried basing a reward function on an approximation of the volume of the resulting Polyhedra, unfortunately, similarly to the case of the features, the complexity of this computation was too high which made it too impractical to use in practice.

4.6. Action selection algorithms

As discussed before, I separated the Q function estimation into two separate subproblems. As I already mentioned earlier, this allows a certain amount of benefits that would not be possible otherwise. However, it also imposes one major complication. These two problems 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 estimations and pick an ideal action accordingly. I tried out a few 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. That is to say that we do not have to train using these in order to measure their performance afterwards. This only works if we train purely randomly however. As if, if we were to train using one selection algorithm and then test with another, this would surely deteriorate the results. 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 I used is perhaps the most intuitive one. First scaling both q function estimations. The performance one between [-1,0] and the precision one between [0,1]. Adding the values together and picking the action with the maximal value. Whilst seeming very 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, it however has less of a guarantee about the second to best and third to best action. 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 to take is much smaller. Using this selection algorithm tends to quite often not choose the best action but the second best or the third etc... making the probability that they are also good actions also a lot smaller. The second problem with this type of selection algorithm is that scaling the reward function causes a loss of information that could be very important. I will demonstrate this with an example, let's say 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 and more precise than j2 and j3 is more precise than j5. 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 to loss in performance for the case of j1 and j2, whilst this would probably not be case for j3 and j4.

Pseudo code of algorithm 1

Algorithm 2: 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 directly confront both problems of the previous algorithm proceeds as follows. We set some sort of 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. Again, this algorithm seems quite good at first glance and maybe if it was executed perfectly it would be the best option, but like the one before it has some fundamental problems. First of all, picking a threshold is not a very simple task. The q function estimator is a regression neural network, the activation function of its last layer is linear, this means that the output values are unbounded and they do not have a direct correlation with Realtime CPU cycles. In order to pick a threshold, I 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.

Pseudo code of algorithm 2

Algorithm 3: 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 some of its short comings. This time we set both some sort of 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 take 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.

Pseudo code algorithm 3

Algorithm 4: 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 I will talk about here is based upon the last one and made in such a way as to reduce the importance of correct parameter choosing. This time we begin by scaling the performance prediction to [-1,0]. We set a threshold to [0,1]. We pick the action that has the highest precision q 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, in the worst case we will pick the action with the best performance. The threshold is also a lot easier to set since it is bounded. Unfortunately, once again we have the problem caused by the scaling of the performance estimation. However, after some experimental observation, I saw that when the join is fast all the q performance estimations tend to be quite close together, so hopefully this will not be all too much of a problem.

Pseudo code algorithm 4

Algorithm 5: 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
$Q_{pr} = Q_{pr}/max(Q_{pr})$
$Q_{pe} = Q_{pe}/min(Q_{pe})$
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

4.7. Neural network characteristics

As for the characteristics of the neural network that I used. I started with a relatively small neural network and the grew it progressively as I expanded the number of vectors and the size that these can have. In the final version I use a fully connected neural network with four hidden layers of two hundred nodes each. They have each 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. I use stochastic gradient descent for updating the weights of the nodes and I clip its norm to 1. I do this in order to avoid the exploding gradients problem and prevent the predicted values to diverge towards infinity. I use the mean squared error in order to calculate the loss, I as well clip this to [-1,1] in order for the predicted values to not diverge towards infinity. I based the characteristics of my neural networks based on reading as these types of networks seem to be the most widely used ones for deep reinforcement learning. It is worth noting that I have not done much parameter optimization on my neural networks as the training time of the algorithm is quite long which would make the parameter optimization a very tedious task. It is also worth noting that I use the same network models for both performance and precision prediction.

4.8. Training

The training of the algorithm was done in multiple stages. During the first stage, only random actions were picked. This was done in order for the algorithm not to get stuck in local maxima but have the most global policy possible. I also did not have any time constraints about the length of training so I 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 concentrating on the precision and the other on the performance. All the actions were no longer chosen randomly, but now the action with the highest predicted value was chosen. The probability with which we chose a random action was progressively decreased throughout training. This was done in order to reinforce the choice of the best action and increase convergence speed. Once these stages finished, we would have two separate systems each optimized for their own task. In the final stage, I would then combine both of the systems and do a final training phase with the chosen action selecting algorithm as described above. I did this in order to further optimize the decision policy for the action selection algorithm.
This method of training is quite time intensive. I proceeded 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.

5 Implementation

details about implementation. explain key parts of the project in terms of the implementatio

5.1. Software Components

key components

5.2. Graphics

explain parts of graphics, openGL etc.

5.3. Sound Analysis

implementation of the frequency analysis

5.3.1. Songs

data structure used as song

5.3.2. Notes

data structure used as note (graphical note explained in subsection graphics so only explain normal note)

5.3.3. Timing

in what sense was the timing aspect implemented

5.3.4. Midi Playback

midis generated with jMusic library, offered as an additional help for the user

5.3.5. Noise Detection

thresholding, show graphics about the choice of the constant used to threshold

5.4. Game Controller

aspects related to the controller

5.4.1. Player

data structure used as player

5.4.2. Score

score saved. what is the score

5.5. User Interface

how does the interaction between user and application work.

Evaluation

6

evaluation of the project. user study etc.

6.1. Methodology

general information, in what aspects the system was evaluated

6.2. Experiment: Comparison of two Prototypes

explain idea behind controlled comparison study

6.2.1. Setup

setup of this user experiment

6.2.2. Results

discuss results of the controlled comparison

6.3. Experiment: Cognitive Walkthrough

explain idea behind cognitive walkthrough, what aspects

6.3.1. Setup

setup of the cognitive walkthrough

6.3.2. Results

results of the cognitive walkthrough

6.4. Long term study

explain idea behind long term study, what would be the goal of this experiment

6.4.1. Setup

setup of the long term user study

6.4.2. Discussion

explain issue of time, was not enough to actually perform the study

6.5. Limitations

what are the most significant limitations in the project, how could the be improved in future work

Conclusion and Future Work

some final sentences about the project and some ideas of future improvements

Appendix

A

in here we put all questionnaires, guides etc.

A.1. Controlled Comparison

questionnaire and guide, script

A.1.1. Guides

guide for controlled comparison

A.1.2. Questionnaire

questionnaires

A.2. Cognitive Walkthrough

questionnaire and guide, script

A.2.1. Guides

guide for cognitive walthrough

A.2.2. Questionnaire

questionnaires

A.3. Long Term Study

questionnaire and guide, script

A.3.1. Guides

guide

Bibliography