SAT Solving via Artificial Intelligence

CO600 Group Project

Autumn Peles

Muhammad Lutufullah

Mohammad Abdin

Aivaras Stupuras

1. Abstract

In this project we aim to explore a machine's ability to reason on satisfiability problems using reinforcement learning. We will be developing an environment that feeds a random CNF formula with each training episode, and the agent is tasked with simplifying the formula efficiently to reach a correct solution. In the following sections we will be giving a background to our project before going into our planning and execution.

2. Introduction

RL has been gaining lots of attention lately, with projects like AlphaZero and AlphaGo beating the highest level of players [1]. While it might seem more intuitive to use supervised learning for this problem since it can be thought of as a classification problem (i.e a cnf can be either satisfiable or unsatisfiable) we chose to go with reinforcement learning to explore whether or not an agent will be able to develop unique SAT solving techniques that can beat modern SAT solvers.

3. Background

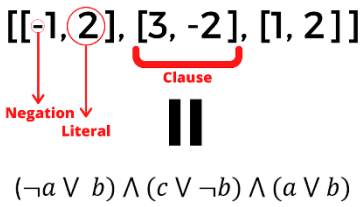


Figure.1 Formula in conjunctive normal form (CNF)

We will be working with and trying to satisfy problems in conjunctive normal form in this project so it is important to know what they are, what they consist of and how they work.
Propositional logic formulas consist of a combination of variables and various operators such as AND (also known as conjunction, ∧), NOT (also known as negation, ¬) and OR (also known as disjunction, ∧). The boolean satisfiability problem is the problem of determining whether an interpretation exists that satisfies the given boolean formula provided. The variables in the formula are replaced by values such as TRUE or FALSE until the formula as a whole becomes TRUE which means the formula can be deemed as 'satisfiable'. A literal is known as the variable and can be either positive or negative. A clause is known as the disjunction of literals or a single literal. A clause becomes satisfied by any interpretation that satisfies at least one of its literals and is falsified for any interpretation that falsifies each literal in the clause. For a formula to be in conjunctive normal form it has to have a conjunction of different clauses or a single clause. The CNF formula is said to be satisfiable for an interpretation which satisfies all of its clauses and it is falsified for any interpretation which falsifies at least one of its clauses.

3.1 Deep Learning Neural Networks

Broadly, deep learning refers to a subset of learning algorithms that involve less directed means of learning patterns in the data they're given. Though these techniques can be applied to both supervised and unsupervised learning, deep networks tend to be given more or less raw data, with very little, if any, guidance on how to extract features from that data. Such networks have been shown to be capable of learning patterns far more obscure and difficult to describe than those an engineer might typically be able to explicitly program or point towards, and are often more useful to the network for accomplishing its given task as well. As a result, deep learning is a powerful paradigm to employ. Our task specifically deals in problems that have been standardised to a very raw form already in CNFs, and as such it's a highly appropriate situation in which to utilise a deep learning algorithm such as the Deep Q Network we chose.

3.2 Markov Decision Process (MDP)

The markov decision process is the structure used by a reinforcement learning method. It consists of five components namely *agent, environment, state, action and reward.*

In this process, the agent will make decisions and interact with the environment and these interactions will keep occurring over time. At each time step the environment will be in some sort of state which the agent will have an idea of. After observing this, the agent will decide and choose an action to perform. The environment will then undergo a transition and move into a new state and as a result the agent will be given either a positive or negative reward based off the previous action it performed

This process of selecting an action from a given state, transitioning to a new state, and receiving a reward happens sequentially over and over again, which creates something called a trajectory that shows the sequence of states, actions, and rewards.

Throughout this process, it is the agent's goal to maximize the total amount of rewards that it receives from taking actions in given states. This means that the agent wants to maximize not just the immediate reward, but the cumulative rewards it receives over time.

3.3 Related works

In addition, after doing research on this topic we found that the topic of 'AI for SAT-solving' has been explored by postgraduate students at Stanford university, where they used single-bit supervised learning [2]. So we thought it would be more valuable and interesting if we explored a reinforcement learning approach to this particular domain.

We will be using Python for this project as it seemed to be the ideal choice given that most machine learning frameworks are mainly built on it.

3.4 Environment overview

The environment will hold the logic behind the execution and also store information like the current state of the CNF, and also provide feedback for the agent though a reward value. Every RL environment consists of an action space and an observation space, the action space defines what action input to expect from the agent and the observation space defines the feedback or 'state' that the agent is in.

The agent initially explores the environment by testing our different actions, observing the state that they return and the reward, assigning a 'Q' (quality) value for each action, state pair. However, for more complex environments with lot's of actions and states it can be impossible to store the 'Q' value for each action, this is where state-of-the-art deep learning techniques are used to estimate the Q value instead. The agent is initialized with a random policy network, which dictates which action to choose based on the current state of the environment. As the agent explores the environment, the policy network is optimized to maximize the reward.

To translate that into our project, the action space is simply choosing a literal and setting it to either positive or negative, while the observation space is the CNF itself. The agent receives a positive reward if it predicts a problem currently and a negative reward if it gives a wrong prediction, chooses a literal that has been eliminated, or does a simplification that compromises the CNF (i.e chooses a literal making a CNF unsatisfiable when it was initially satisfiable)

It's a common practice in reinforcement learning to make the training process 'end-to-end' which implies not providing any human domain knowledge to the agent or applying feature engineering but rather allow the agent to explore the logic behind CNF formulas and develop its own strategies to solve them.

4. Model Structure

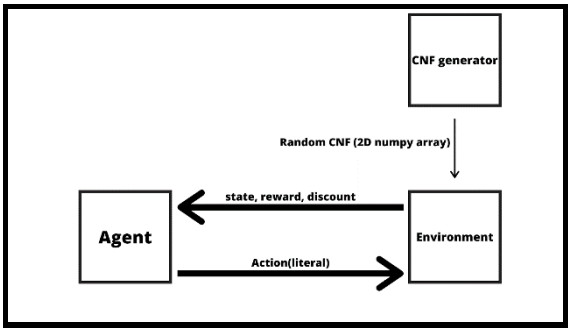


Figure 2. Model abstraction

Figure 1 captures a high level idea of the initial plan for the model. The training occurs in episodes, an episode here is defined as 'solving a CNF formula' once the agent either finishes solving it or performs an incorrect move (i.e choosing a literal that's been removed from the cnf) the episode ends and a new random CNF is generated by the cnf generator.

Within each episode, there can be a number of timesteps. Essentially what happens in each time step is the following:

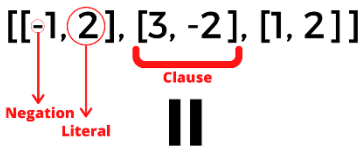
1. The agent chooses a literal or a negation of that literal to set to true and simplify the formula
2. The environment goes through the logic of simplifying the CNF
3. Finally, the environment returns a TimeStep() that contains state, reward, and discount. Data structure will be explained in further detail in the next section.

The 'discount' is a value from 0 to 1 that applies a decaying effect to the reward given to the agent with each timestep. This is usually used to stop the agent from trying to prolong the episode taking advantage of a loop-hole to maximize the reward but in our problem that shouldn't be the case so at least in our initial build of the model we'll just be using a discount value of 1 meaning the reward is stable throughout.

4.1 Data types and formatting

To begin with, we had to decide on the format that our CNF will take. This is crucial because it's the main component that our agent will be interacting with and also observing. After researching what other SAT

solvers use to encode a CNF we saw that the most common format was a 2-D list of integers, where each integer represents a literal, and each list in size the outer list is a clause. This is better illustrated in the following diagram:



$(\neg a \vee b) \wedge (c \vee \neg b) \wedge (a \vee b)$

Figure 3. CNF structure

One concern that we had was that the cnf formula size cannot be dynamic, in deep Q learning the size of the state must be stable but that won't be the case if we simplify the cnf with each timestep. As a solution we decided that instead of removing the literals in the simplification process, we will replace them with zeros.

As previously mentioned, we will be using TensorFlow's tf-agents library for our model. There is one essential data structure that should be noted here which is the TimeStep() object. This class is returned from the environment using one of the three methods ts.termination(), ts.transition(), ts.reset(). Each of these returns a timestep with a different 'step type' if the agent receives a termination time step that tells it that the episode has ended, where a transition would keep it going.

4.2 Environment

Our environment will hold all the logic for solving the cnf and also provides feedback for the agent. The core of any reinforcement learning environment is in its action space & observation space, which denotes the array of actions that can be taken by our agent and the structure of the state to be observed by the agent, respectively. In our case, these are:

- Action space: array of shape (1,) with a minimum value of -N_LITERALS, and a maximum value of N_LITERALS+1 where N_LITERALS is the number of variables contained in our cnf. Note that we've added a +1 to the positive value, this is because the agent will also have the choice of making a prediction on the cnf at any point, where an input of '0' predicts unsatisfiable, and an input of 'N_LITERALS+1' is satisfiable.
- Observation Space: this would simply be a 2-dimensional array containing the cnf, as previously discussed in the data types section. However with deep Q learning the observation space and action space must be dimensional aligned, this forced us to return to state as a flattened version of the original 2-d array. This will be further discussed in the limitations but to briefly highlight the problem the agent here is not aware of the separation point between clauses, but we do maintain a fixed clause size throughout training. Therefore, it is possible for the agent to learn the separation point of the clauses while training.

The environment gets instantiated with a randomly generated cnf formula and the agent then is required to simplify it and give a prediction once it reaches a good level of confidence. The primary method used for this execution is given by our step() function. The step() function takes in an action input from the given action space, if the agent chooses to simplify the formula using a literal 'p' we delete every clause in S containing and every occurrence in the cnf of -p. This simplification process is partly taken from the DPLL algorithm. But since we want to have a fixed size for the observation space, instead of deleting these literals we replace them with zeros. If the action given is a prediction (i.e '0' for unsatisfiable or N_VARIABLES+1 for satisfiable) the training episode ends and the agent is rewarded accordingly.

4.3 Reward system

We wanted to make the training as 'end-to-end' as possible, since the aim for our agent is to explore how it can solve a cnf. Based on that notion, the agent only gets a positive reward when it makes a correct prediction and a negative reward whenever it makes an invalid move or a wrong prediction. An invalid move would be for example attempting to simplify by a literal that was eliminated, or simplifying a cnf to unsatisfiability when initially it was satisfiable. All these moves are checked using the pysat library.

4.4 Pysat

Our environment will include a pre-existing sat solver which will solve the CNF formula and the result will be used to judge the actions made by the agent by making comparisons. We will be using MiniSat which is a simple, well documented implementation suitable for educational purposes, supports incremental SAT and has mechanisms for adding non-clausal

constraints. By virtue of being easy to modify, it is a good choice for integrating as a backend to another tool, such as a model checker or a more generic constraint solver.

4.5 Agent

We will be using the standard implementation provided by TF-Agents for a Deep Q Network. The Deep Q Network agent can be used in any environment which has a discrete action space. A substantial part of a Deep Q Network Agent is a QNetwork, a neural network model that can learn to predict QValues (expected returns) for all actions, given an observation from the environment. We will use ***tf_agents.networks.*** to create a QNetwork. The network will consist of a sequence of ***tf.keras.layers.Dense*** layers, where the final layer will have 1 output for each possible action. The various hyperparameters such as the learning rate, number of iterations, cnf size etc can be changed and modified to see how it affects the results and the loss in our case by experimentation and will be discussed more in the testing section. There will be two environments defined, one each for evaluation and training. As mentioned above, the policy is created such that it will make random actions for each time step as it trains. This is done using ***tf_agents.policies.random_tf_policy***. The policy is evaluated using the average return which is the sum of rewards during an episode in the environment while the policy is being run. After several episodes are completed an average return is created.

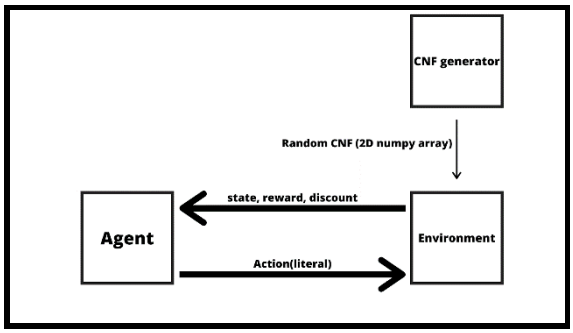


Figure 2. Model abstraction

Figure 1 captures a high level idea of the initial plan for the model. The training occurs in episodes, an episode here is defined as 'solving a CNF formula' once the agent either finishes solving it or performs an incorrect move (i.e choosing a literal that's been removed from the cnf) the episode ends and a new random CNF is generated by the cnf generator.

Within each episode, there can be a number of timesteps. Essentially what happens in each time step is the following:

4. The agent chooses a literal or a negation of that literal to set to true and simplify the formula
5. The environment goes through the logic of simplifying the CNF
6. Finally, the environment returns a TimeStep() that contains state, reward, and discount. Data structure will be explained in further detail in the next section.

The 'discount' is a value from 0 to 1 that applies a decaying effect to the reward given to the agent with each timestep. This is usually used to stop the agent from trying to prolong the episode taking advantage of a loop-hole to maximize the reward but in our problem that shouldn't be the case so at least in our initial build of the model we'll just be using a discount value of 1 meaning the reward is stable throughout.
tool, such as a model checker or a more generic constraint solver.

5. Aims

The primary aim of this project was to create a self-learning sat solver which would potentially outsmart its competitors and become more efficient over time as well as giving the best solution possible.We were also hoping to potentially find new, interesting ways that CNFs can be solved even though they might not be as efficient as it is interesting how AI will approach the problem. One of the main ambitions for our project was Chess AI because it took one most complex game that exists and revolutionized how it can be played with perfect accuracy and efficiency. It clearly showed that AI can be extremely efficient and beat its creators if trained the right way.

6. Development stages

6.1 Iteration 1

Introduction

The focus of the first iteration was to get a better understanding of TensorFlow training and create a basic agent and environment based around it. From our meetings, we created tasks that we worked on below.

Features and tasks

CNF generator

To initially start the process of training our AI, we needed the CNFs for it to be properly trained therefore during this iteration we worked on creating the generator that will feed our agent with randomly

generated CNFs on a 50/50 satisfiable/unsatisfiable scale.

PySAT

Implemented PySAT toolkit in our system for our CNFs to be solved.

Environment

For our agent to be trained we needed to build an environment to get things started. The environment will be feeding the agent randomly generated CNFs to be trained with reward values.

Challenges

One of the main challenges that we have faced during this iteration was the formula to produce a generated CNF as our current one kept crashing the kernel and when we tried to fix it we ran into a loop.

Results

Found out that generating CNFs on a 50/50 satisfiability scale is problematic and will cause kernel crashes so for this iteration of the system we have decided to run only satisfiable CNFs as it does not cause a kernel crash.

6.2 Iteration 2

Introduction

During this iteration we have focused on making an agent and implementing py-sat toolkit in place, as well as fixing the CNF generator formula so it can produce on 50/50 scale.

Features and tasks

Agent
To build the agent we have followed a TensorFlow for generic skeleton structure of the code during this iteration and adapted it to match our needs.

PySAT
Moved to different SAT solver available in the library in hopes it will fix the kernel crash.

CNF generator
During this iteration of the system, we have decided that training a model with CNFs that are satisfiable is bad because it will train the system to guess instead of solving it. We worked on the formula that will stop the kernel crashes.

Challenges
The challenges that we have faced were kind of difficult because we were not accustomed to the kernel crash errors in our previous projects, so to find an actual problem in the code we had to go over the code step by step to find the problem which was a learning experience.

Results
Managed to solve the formula with step-by-step debugging. Agent has been coded and implemented with the environment. Kernel crashes were solved by switching sat solver available in PySAT library.

6.3 Iteration 3

Introduction

This iteration we mainly focused on getting every problem fixed so we can start training our model.

Features and tasks

Agent
Agent has been implemented within the environment and now runs without any kernel crashes or loops.

CNF Generator
CNF generator is able to produce CNFs that are satisfiable and satisfiable on 50/50 scale

Challenges
Some of group members during this iteration have faced problems with their library of TensorFlow being outdated which made them unable to run the code, however this quickly has been resolved with the pip upgrade.

Results
We have produced a system that can now be trained.

7. Testing

To test and optimise our program, we ran several tests with different combinations of hyperparameters. For each hyperparameter,
we tried multiple values, and for each combination, we ran our program three times and took the average result to improve the reliability. The table compiling these results can be see below [Figure.3]

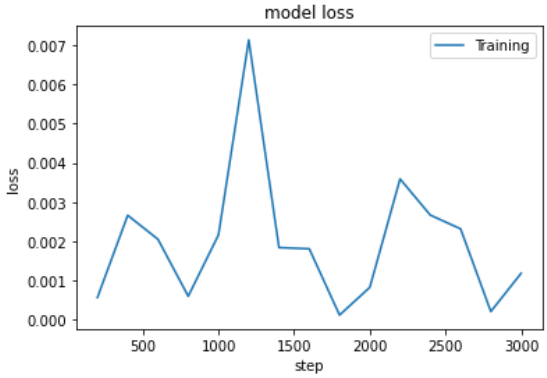
Batch Size	Learning Rate	FC Layer Parameters		Final Loss			Average
64	0.001	100	50	0.0005584529135	0.0005998211564	0.0001745332265	0.0004442690988
64	0.01	100	50	0.003779500956	0.0003476087586	0.0003476087586	0.001491572824
64	0.1	100	50	0.01178383082	0.01412098296	0.01997499168	0.01529326849
32	0.001	100	50	0.000137885203	0.0002075384109	1.06E-05	0.0001186909682
32	0.01	100	50	0.0006001596921	0.001596646965	1.41E-58	0.0007322688859
32	**0.1**	**100**	**50**	**0.015753543**	**0.04664067179**	**0.01716773957**	**0.02652065145**
128	0.001	100	50	8.05E-05	0.000136854389	0.000374398136	1.97E-04
128	0.01	100	50	0.000346992776	9.75E-05	0.0001639090915	0.00020278593
128	0.1	100	50	0.01273574959	0.01438765693	0.006270032842	0.01113114646
64	0.001	200	100	4.13E-06	8.27E-06	0.0002350011055	8.25E-05
64	0.01	200	100	9.53E-05	9.03E-05	0.0007202536217	3.02E-04
64	0.1	200	100	0.01103180461	0.01145917457	9.03E-05	0.007527088055
32	**0.001**	**200**	**100**	**0**	**0.0001347607322**	**2.51E-07**	**0.00004500399461**
32	0.01	200	100	0.0003283910919	0.001147955889	0.01949498057	0.006990442518
32	0.1	200	100	0.03185731173	0.003789396491	0	0.01188223607
128	0.001	200	100	3.31E-05	5.64E-05	0.000210732047	1.00E-04
128	0.01	200	100	0.0003692329919	0.0003882106976	2.49E-05	0.0002607846588
128	0.1	200	100	0.008150991052	0.007847649045	0.008418227546	0.008138955881
						Lowest value:	**0.00004500399461**
						Highest value:	**0.02652065145**

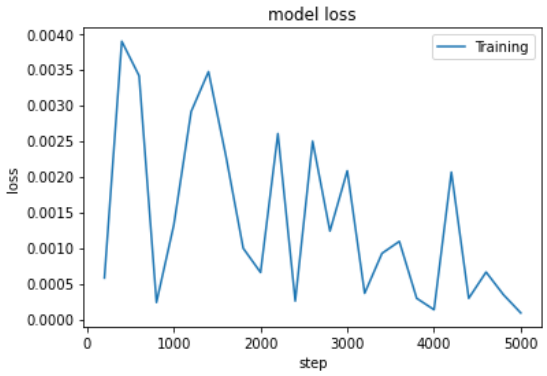
[Figure.3 Loss testing table]

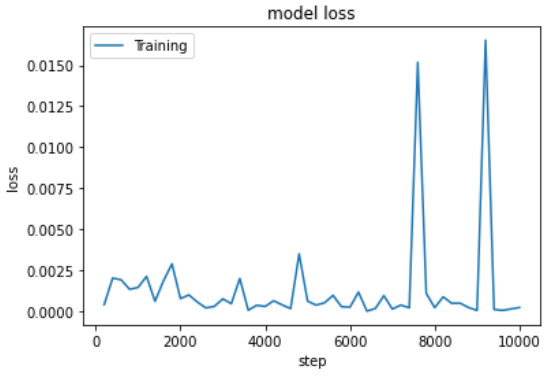
We ran each test for 20,000 cycles and recorded the final loss. This allowed us to get a holistic view of how each of the hyperparameters affected the model's learning capability and extract useful patterns which would prove useful for further tuning. For example, the model consistently performed better with a smaller learning rate, so it would be a good idea to try some even smaller learning rates and see where the limit to that is. The batch size, on the other hand, has less conclusive evidence for the best value - both the lowest and highest recorded average final loss values came from a model with a batch size of 32, and there didn't appear to be a strong consistency between a higher or lower batch size and better performance, on average, meaning that further investigation would be required - more data may either show a pattern that is not yet evident, or provide further evidence that the batch size is actually relatively unimportant to the model's effectiveness.

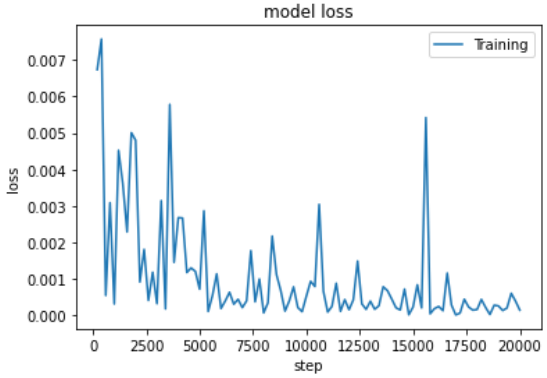
We tuned each parameter and ran each test manually, which definitely led to it taking much longer than it could have if it had been automated. However, we opted against devoting time to creating code to automate the testing, as we didn't plan to do more testing beyond what is shown in the table above, and believed it would be wasteful. Additionally, it could've introduced further bugs that could have led to invalid results. That said, if further testing were to be done, and over many more combinations of hyperparameters, then it would be a good idea to develop a good and robust function for doing this automatically, and perhaps even recording the results into the table automatically, so this would be a point to consider were further work to be carried out.

Next, we present a number of graphs that demonstrate some important things to consider about the learning process.









As can be seen, the model's error, or loss, is both highly prone to large fluctuation (shown most clearly in the first and third graphs), but also trends towards getting lower over time (shown more clearly in the second and fourth), as well as more consistently low the further it goes (shown best in the fourth graph, which runs for the largest number of cycles). These are useful things to know when

working with the model and could even provide motivation for accommodating this in future work - for example, the model could retain the best state it has ever reached in order to provide robustness against an unlucky random fluctuation away near the end. Had the last graph instead stopped at the large peak shortly after 15000 steps, the good results surrounding it would have been lost. While repetition of the training can also provide this robustness, running the program multiple times is costly in both time and energy, and would need to be carefully considered were this program applied on a larger scale. Furthermore, this information provides justification for running the program for the length of time that we did, as the model tends to get better results more consistently the longer it is able to be trained. The precise limit to this trend is something that could be investigated more thoroughly, but the broad strokes are still useful heuristics to have learning during this process, and as such we felt were worth highlighting.

8. Limitations & Future work

After numerous testing and research we can reflect back on some of the limitations we faced. Our biggest concern was with the way we structured the problem as mentioned in section 4.1, framing the cnf as a 2-dimensional array of integers could have been problematic since our policy network will associate the value of the integer with a magnitude instead of a variable label. To elaborate, assuming that we have two cnf formulas with only 2 variables each, if one cnf has the variables labelled '1' & '2' while the other has them label '1', '40', the output of the final layer will largely differ between these two cnfs. Another limitation to note is the fact that the returned state for the agent is a flattened version of the cnf which eliminates any information about the separation between clauses. It should be possible however for the network to learn this separation given that our training structure is fixed. We also tested training our agent on cnfs with only 1 variable per clause and observed how largely it would affect the model's accuracy but as we expected the results weren't largely different.

Furthermore, given these limitations and as we have gained more knowledge about reinforcement learning throughout this project we can hypothesize some changes to our approach which can lead to better results. Firstly, we would change how the problem is encoded, while a 2-D array seems to be an intuitive structure for SAT solvers, it might not be a reliable input format for our agent's policy network. Graph neural networks (GNN) may have proved to be useful for encoding the problem. Similar to how computer vision reinforcement learning projects utilize convolutional networks to understand and recognize patterns in a frame/image, and since it's common to represent CNFs as graphs where each node is a literal, GNNs can be used to extract something meaningful from these representations and feed it to the agent. Lastly, it would have been useful to add visualization and tests to better observe what the agent is doing. We were only able to view the cnf updates with each timestep but it would be worth implementing automatic tests that would detect if the agent is applying unit-propagation or tautology elimination for example.

9. Conclusion

To conclude, we built a system that uses a deep Q network to solve CNFs with effort. The main aims of this project were to produce a self-sufficient system, that if trained well will produce a model that is more efficient than 'hand crafted' CNF solvers out there. Proving that AI-built systems can be more efficient and give a new-look at the problem at hand. CNF Solver overall has been a great learning experience with a lot of positives and negatives. Our solver has started to show serious progress in how a well-trained model can compete with other CNF solvers in the market. All tool kits and technologies that were used for the project were modern and up to market standards. Regarding the innovation of our project, it lies within reinforcement learning and SAT solving by hand because it still must be trained with the right model in mind there is still some manual labour that has to be put into this project for it to be fully successful.

Acknowledgements

Our group is deeply appreciative of guidance given to us by our supervisor Stefan Kahrs, as well as the TensorFlow and Py-Sat team for providing us with the resources.

References

[1] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., . . . Hassabis, D. (2017). *Mastering Chess and Shogi by Self-Play with a.* London: DeepMind.

[2] *Supervision.* Redmond: Microsoft Research.