Vrije Universiteit Amsterdam

Bachelor Thesis

DeepSAT: A Machine Learning Approach for Solving the Boolean Satisfiability Problem

---

Author: Fabrizio Galli Acosta (2589816)

1st supervisor:	Jörg Endrullis
2nd supervisor:	Femke van Raamsdonk
3rd reader:	Erman Acar

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 11, 2019

Abstract

In this paper, the classic NP-Complete *Boolean Satisfiability Problem*, or SAT, is interpreted as a reinforcement learning task. The model, called *DeepSAT*, successfully solves instances of the SAT problem with a limited number of input variables. This paper illustrates how reinforcement learning is applied to the SAT problem, discusses the findings observed and limitations of the model, and provides supporting theoretical background for the topic.

Keywords— SAT, Boolean Satisfiability Problem, Machine Learning, Deep Learning, Reinforcement Learning, NP-Complete Problems.

1 Introduction

Consider the scenario where a salesperson has to visit a number of houses to offer his products and return home at the end of the day. Ideally, she would like to return home as early as possible to spend time with her family, hence, it would make sense for her to choose the shortest possible route to do so.

If this salesperson has to visit n houses, she would have to consider n possible houses to visit first, and, for each of these, she would have to consider $n-1$ possible houses to visit immediately after, and so on until there is only one house to choose from. In mathematical terms, she would have to consider $n!$ possible routes. The number of possible routes grows very rapidly as the number of houses increases, indeed, the number of possible routes grow at an exponential rate with respect to the number of houses, in fact, if the number of houses to visit was only 60, there would be more possible routes to choose from than there are atoms in the universe.

The scenario described above is a well known problem in computer science, called *The Travelling Salesman Problem*. The *TSP* belong to a class of problems called *NP-Complete Problems*, which are problems that are prominent in many computing applications, from logistics to theoretical physics. Finding solutions for these problems requires searching within astronomically large spaces. How fast these problems can be solved is one of the biggest unresolved problems in computer science, and most experts believe that these problems cannot be solved efficiently, yet this has to date not been mathematically proven.[2]

In recent years, with advancements in artificial intelligence, computers were able to perform impressive tasks that, short ago, seemed improbable. In 2015, Google's *AlphaGo* was able to defeat, 5-0, a professional *Go* player, a feat that was believed to be, at least, a decade in the future[5]. *AlphaGo* achieved this with the use of *reinforcement learning* and *deep learning*. Deep learning is a recent approach that uses neural network models, technology inspired by the physiology of the human brain, and has achieved impressive performance in tasks such as face recognition, automatic translation, and self-driving car technology.[3]

The research question for this project is the following: *Can a machine learning model learn the solution of an NP-Complete problem?*. To investigate this question, we take the *Boolean Satisfiability Problem* (the first problem proven to be NP-Complete), and develop a method using *reinforcement learning* with the goal of finding solutions for the problem; This method is referred on this paper as *DeepSAT*. In this paper, the background theoretical aspects of the project are first presented to the reader, followed by a detailed description of the application of reinforcement learning to the DeepSAT model, in the context of the Boolean Satisfiability Problem. Finally, the results and observations of the performance and the weaknesses of the model are discussed.

2 Preliminaries

2.1 The Boolean Satisfiability Problem

The Boolean Satisfiability Problem, or SAT, in propositional logic is a simple decision problem that plays a prominent role in Complexity Theory and Artificial intelligence, with important applications in hardware and software verification.[1] The problem is about determining if there exists an interpretation that satisfies a *boolean formula*, that is, if there exists a model in which truth values (*True* or *False*) are assigned to the variables of the formula in such a way that makes the formula evaluate to *True*. For example, the formula $(A \wedge B)$ is satisfiable, while the formula $(A \wedge \neg A)$ is not satisfiable. SAT problems are normally presented in *conjunctive normal form* (CNF), that is, formulae that are conjunctions over disjunctions over literals.

The SAT problem belongs to the class of NP-Complete problem. This means that any problem in the NP-Complete class, which includes a wide range of decision and optimisation problems, are at most as difficult to solve as the SAT problem.

2.2 Reinforcement Learning

In this section, a summary of the background information regarding reinforcement learning, which is relevant to this paper, is presented to the reader. The contents of this section are largely obtained from the book *Reinforcement Learning: An Introduction*, by Richard S. Sutton and Andrew G. Barto.[6]

Typically, machine learning is considered to have two main branches of study: supervised learning and unsupervised learning. In supervised learning, the learner model receives a set of labeled examples as training data from a knowledgeable supervisor, and infers a function to map an input to an output.[4] In unsupervised learning, the learner model receives a set of unlabelled data and finds a structure hidden in this collection of data. Reinforcement learning is different from these two approaches. In reinforcement learning, an agent is learning to map situations to actions, so it can maximise a numerical signal, called the reward. Therefore, reinforcement learning can be considered to be a third paradigm of machine learning.

Reinforcement learning has its roots on dynamic programming, which was developed by Richard Bellman for solving problems that Bellman described to suffer from "the curse of dimensionality", meaning that the computational requirements to find optimal solutions grow exponentially with the number of state variables. Reinforcement learning can be used when the set of possible states is very large, or even infinite.

There exists a number of cases where reinforcement learning has shown to effectively tackle problems where the search space is exponentially large. An instance of this is TD-Gammon[7], a RL artificial network that learns to efficiently play the game backgammon, which has 10^{20} states, and greatly surpasses all previous computer programs on its ability to play the game, performing at the level of the world's best human players.

In reinforcement learning, a learning agent interacts with an environment with the task of maximising a numerical reward. This agent must be able to sense the state of the environment and it takes actions to affect the environment. Each action taken yields a new state of the environment, as well as a numerical reward. The goal of the agent is to learn which sequence of actions will lead to the maximal reward. This interaction is modelled under the formal framework of Markov decision process, described in the following subsection.

2.2.1 Finite Markov Decision Process

The finite Markov decision process, or finite MDP, defines the interaction of a learning agent and its environment in terms of states, actions and rewards. MDPs are a classical formalisation of sequential decision making, where the actions taken by the agent affect not only the immediate rewards, but also the subsequent states of the environment and, through this, future rewards.

MDPs are modelled using the *agent-environment interface*. In this interface, the learner, or decision maker, is called the *agent*. The thing the agent interacts with (everything outside of itself) is called the *environment*. The agent and the environment interact continually over a number of time steps t, on which the agent performs *actions* on the environment, each resulting on a numerical value, called the *reward*. The goal of the learning agent is to maximise this numerical reward over time through its choice of actions.

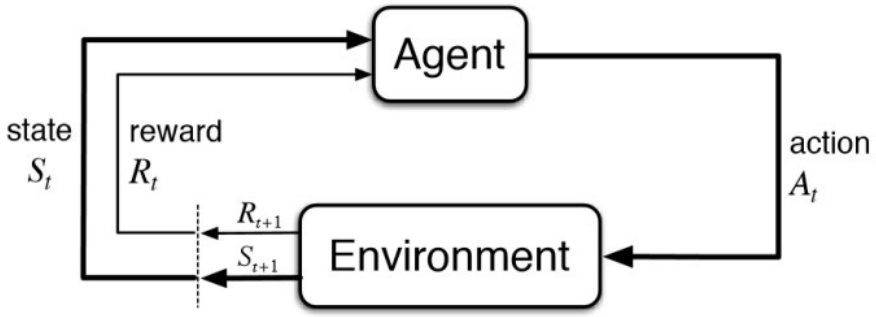

Figure 1: The Markov Decision Process Model.

The sequence of actions occur over a discrete number of time steps, $t = 1, 2, 3, \ldots$. In each time step, the agent receives a representation of the state of the environment, $S_t \in \mathcal{S}$, and performs an action, $A_t \in \mathcal{A}(s)$, on the basis of the current state. On the following time step, the agent receives a reward, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and a new state of the environment, S_{t+1}.

In a finite MDP, the sets of states, actions and rewards (S, A, R) created by the interaction of the agent and the MDP have a finite number of elements. These sets give rise to a sequence, or episode, that has the following form:

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$$

where T is the terminal time step.

2.2.2 Episodes and Returns

Tasks that break down into a finite sequence of states are called episodic tasks, in which one episode is a run from time step t to time step T (the terminal time step). During an episode, a collection of rewards is observed in each time step, denoted by $R_{t+1}, R_{t+2}, \ldots, R_{T-1}, R_T$. When performing the optimisation task, we are looking to maximise expected return, i.e., the aggregation of the observed rewards, denoted by G_t. The most simple definition of the expected return is the sum of all observed rewards:

$$\begin{aligned} G_t &= R_{t+1} + R_{t+2} + \cdots + R_{T-1} + R_T \\ &= \sum_{k=t+1}^{T} R_k \end{aligned}$$

An additional concept is added to the notion of expected return, that is, the discount rate. The discount rate, denoted by γ, is a parameter used to determine the present value of future rewards. This extended definition of returns is called the discounted expected return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots + \gamma^{T-t-1} R_T$$
$$= \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

with $0 \leq \gamma \leq 1$. With the discount rate, a reward received k time steps in the future is worth only γ^{k-1} as much as it would if it were received in the current time step, which gives more weight to the actions taken in the beginning of the sequence.

2.2.3 Policy Gradient

The policy, denoted by π, of a reinforcement learning model determines the behaviour of an agent at a given time. The policy is, thus, a mapping from the state of the environment to the action to be taken. This mapping gives a probability distribution for the actions that can be taken. If the agent is following policy π at time step t, with a state of the environment $S_t = s$, then the probability of taking action $A_t = a$ is denoted by $\pi(a|s)$.

For policy gradient, the notion of parametrised policy is used, where $\boldsymbol{\theta} \in \mathbb{R}^d$ is the policy's parameter vector, with dimensions d. Thus, we write $\pi(a|s, \boldsymbol{\theta})$ for the probability that action a is taken given that the environment is in state s with parameter $\boldsymbol{\theta}$. The policy can be parametrised in any way, as long as $\pi(a|s, \boldsymbol{\theta})$ is differentiable with respect to its parameters. The gradient of the probability distribution $\pi(a|s, \boldsymbol{\theta})$ is denoted by $\boldsymbol{\nabla}_\theta \pi(a|s, \boldsymbol{\theta})$.

The goal of the model is to learn the parameters of the policy, for this purpose an objective function, or *performance measure*, $J(\boldsymbol{\theta})$ is defined, which has to be maximised with respect to the parameters. The method used in policy gradient to learn the parameters is *stochastic gradient ascent* on J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\boldsymbol{\nabla} J(\boldsymbol{\theta}_t)}$$

where α is the learning rate (or step size) of the update, and $\widehat{\boldsymbol{\nabla} J(\boldsymbol{\theta}_t)}$ is a stochastic approximation of the gradient of the performance measure J with argument $\boldsymbol{\theta}_t$.

The *policy gradient theorem* gives an exact analytic expression for the gradient of the performance measure with respect to the policy parameter. The expression given by the policy gradient theorem can be further converted to yield the following identity:

$$\boldsymbol{\nabla} J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[G_t \frac{\boldsymbol{\nabla}_\theta \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right].$$

2.2.4 REINFORCE: Monte Carlo Policy Gradient

Given the mathematical identities described in the previous section, the stochastic gradient ascent update can be written in the following terms:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\boldsymbol{\nabla}_\theta \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}.$$

Intuitively, this formula expresses that the increment is proportional to the product of the return G_t and a vector $\boldsymbol{\nabla}_\theta \pi(A_t|S_t, \boldsymbol{\theta})$, the gradient of the probability of taking the chosen action, divided by the probability of choosing that action, $\pi(A_t|S_t, \boldsymbol{\theta})$. The vector is the direction in parameter space that most increases the probability of repeating action A_t on future visits to state S_t. This

vector is divided by the probability of taking action A_t in order to not give more advantage to actions that are most frequently chosen.

The *Monte Carlo Policy Gradient* is an algorithm to learn the parameters of a model, in order to maximise the objective function $J(\boldsymbol{\theta})$. The pseudo code of this algorithm, named REINFORCE, is given below.

Algorithm .1 REINFORCE

Data: a differentiable policy parameterisation $\pi(a|s, \boldsymbol{\theta})$
Initialise policy parameter $\boldsymbol{\theta} \in \mathbb{R}^d$
while *True* **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following policy π
  for *each time step t in the episode* **do**
    $G_t \leftarrow$ discounted expected return from time step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha G_t \boldsymbol{\nabla}_\theta \ln \pi(A_t|S_t, \boldsymbol{\theta})$
  end
end

Note that the last term is written in the compact form $\boldsymbol{\nabla}_\theta \ln \pi(A_t|S_t, \boldsymbol{\theta})$, using the identity $\boldsymbol{\nabla} \ln x = \dfrac{\boldsymbol{\nabla} x}{x}$.

3 The DeepSAT Model

DeepSAT is a reinforcement learning model whose goal is to find solutions for instances of the Boolean Satisfiability Problem, or SAT problem. The network is trained for each specific instance of the SAT problem, and seeks to learn the solution, assuming there exists one, for that specific problem. Therefore, the network trained on one instance problem does not generalise to other problems.

The intuition behind the models is to map the truth values of the input variables on one state vector $\mathbf{s}$, and, following the *policy* π, assume the truth value of one variable at a time. This *guessing* of the truth values of the variables are the *actions* taken by the agent. The model generates *episodes* by taking a sequence of actions (guessing truth values of variables), and, in doing so, transforming the state of the environment (representation of the truth values on a *statevector*). During an episode, the agent collects *rewards* proportional to the number of variables already guessed. The episodes terminates when the environment finds that the assumption of the truth value of a variable is not consistent with the input CNF.

Given that the rewards are proportional to the number of variables guessed, the network will seek to maximise the amount of guessed variables. If the number of guessed variables is equal to the number of input variables, and the environment finds that this interpretation of variables is consistent with the input CNF, then a solution has been found for the input SAT problem.

In this section, the different components of the DeepSAT model are presented to the reader.

3.1 The SAT Solving Machine

From the most abstract level, we can think of DeepSAT as a machine that takes a certain input problem, processes it, and produces an output. As it is normally the case with conventional SAT solvers, the machine takes as input a set of logical clauses in CNF form and a positive integer, n, representing the number of variables. This model, however, differs to other conventional SAT

solvers in that it will not produce an output of 'YES' if the instance problem is satisfiable and 'NO' if the problem is not satisfiable. This is given because the DeepSAT learner will explore stochastically the search space of the problem and try to converge onto a solution; But, if this solution does not exist, then the learner will remain searching indefinitely. With this considered, the model gives the following program function:

DeepSAT :: CNF(*set of clauses*) $\rightarrow$ n(*Integer*) $\rightarrow$ **Maybe** *Yes*.

An observation is that it is also possible that the model produces no output for a SAT problem where a solution exists, but the agent has not found it.

3.2 The Environment

To model the SAT problem as a reinforcement learning task an environment where the task is performed has to be defined. The elements comprising the Environment class are the *state*, the *action space*, and the *rewards*. Each of these components are described bellow.

3.2.1 The State of the Environment

The environment models the truth values the input variables can take. An input variable can be in one of two states: True or False. The set of all possible combination of state variables has size of 2^n, where n is the number of input variables, which is also the search space of an instance of the SAT problem.

For this model, the state of the variables are represented on a *one hot vector* $\mathbf{s}$. For each variable, there exists two elements in the state vector, one for its *True* state and another for its *False* state. These states can be either *on*, value of one, or *off*, value of zero, depending on whether an assumption of that truth value has been made. The state space has the form:

$$\mathbf{s} \in \mathcal{S} = [x_0, x_1, x_2, x_3, \ldots, x_{2n-2}, x_{2n-1}]$$

with $||\mathcal{S}|| = 2n$, where n is the number of input variables.

In the initial state of the environment, all elements of the state vector have value of 0. During an episode, some variables will be assumed to have some truth value, when this happens, the corresponding element in the vector will turn from 0 to 1.

Consider the example of a SAT problem with three input variables a, b and c: the state vector has the form $\mathbf{s} = [x_0, x_1, x_2, x_3, x_4, x_5]$, representing the states $[a, \neg a, b, \neg b, c, \neg c]$. At the initial time step, $t = 0$, we have $\mathbf{s}_0 = [0, 0, 0, 0, 0, 0]$, and if, for instance, at time step 3, $t = 3$, we have assumed $a = True$, $b = True$, and $c = False$, the state vector would be $\mathbf{s}_3 = [1, 0, 1, 0, 0, 1]$.

It can be observed that all possible states of the variables can be represented on the state vector s, therefore, if there exists a solution for a SAT problem, this solution can be mapped on the state vector.

3.2.2 The Action Space

The actions space is the set of all possible actions that can be performed on the environment. The actions for the case of this SAT solver are to assume that any of the variables are either *True* or *False*. The actions have a one-to-one mapping to the state vector $\mathbf{s}$, so the action space has the form:

$$\mathbf{s} \in \mathcal{A} = \{a_0, a_1, a_2, a_3, \ldots, a_{2n-2}, a_{2n-1}\}$$

with $||\mathbf{a}|| = ||\mathbf{s}||$, for a SAT problem with n input variables.

Considering the previous example of a SAT problem with input variables a, b, and c, the action space for this case will be:

$$\begin{aligned}\mathcal{A} =&\{a_0 = Assume\ a, a_1 = Assume\ \neg a, a_2 = Assume\ b,\\ &a_3 = Assume\ \neg b, a_4 = Assume\ c, a_5 = Assume\ \neg c\}\end{aligned}$$

For simplicity, in this version of the model all states, $\mathbf{s} \in \mathcal{S}$, have the same action space, even if some variables have already been guessed.

3.2.3 The Rewards

The reward corresponding to a certain state of the environment is given by

$$reward\,(\mathbf{s}) = \begin{cases} sum\,(\mathbf{s}) & if\ consistent \\ 0 & otherwise \end{cases}$$

The term *consistent* refers to the cases where there exists no contradictions between the interpretation of the variables in the state vector and the input CNF (this is verified before calculating the reward).

The term $sum\,(\mathbf{s})$ makes sense in terms of optimisation, because we want to maximise the number of assumed variables, while maintaining the consistency property. If the vector has a summation equal to the number of input variables, and the consistency property is maintained, then a solution for the input SAT problem has been found.

3.3 The Agent

The agent is the part of the model where the learning process takes place. The role of the agent is to select an action, $a \in \mathcal{A}$ (i.e. guessing the truth value of an input variable), from a probability distribution, $\pi(A|S, \boldsymbol{\theta})$, obtained by following policy π. During one episode run, the agent records the perceived rewards $\mathcal{R}$ from the environment, and also the natural logarithm of the gradient of the policy at each time step, $\boldsymbol{\nabla}_\theta \ln \pi(A|S, \boldsymbol{\theta})$, for performing the parameters update. After each episode run is finished, the agent updates the parameters using the REINFORCE algorithm described in section 2.

In this section, the different components of the agent of the DeepSAT model are described.

3.3.1 The Policy Network

The probability distribution for the possible actions to be taken, , is the output of a policy network that computes the following *feed forward propagation* function:

$$\hat{\mathbf{y}} = softmax\,(\mathbf{x} \cdot \mathbf{W})$$

This function has a number of components which will be described next.

$\mathbf{x} \in \mathbb{R}^m$: the input vector (state of the environment).

$\mathbf{W} \in \mathbb{R}^{m \times o}$: the parameters $\boldsymbol{\theta}$.

$\hat{\mathbf{y}} \in \mathbb{R}^o$: the output vector (actions to be taken).

The input vector $\mathbf{x}$ is the concatenation of the state vector $\mathbf{s}$, at the current time step of the episode, with the *bias node*. The bias node, $b = 1$, is added to the input vector for preventing that the output of the network collapses to zero when all the elements of the state vector have a value of 0, that is, at the initial time step of each episode. Thus, the length of the input vector $\mathbf{x}$ is equal to the length of the state vector $\mathbf{s}$ plus 1:

$$||\mathbf{x}|| = ||\mathbf{s}|| + 1 = m$$

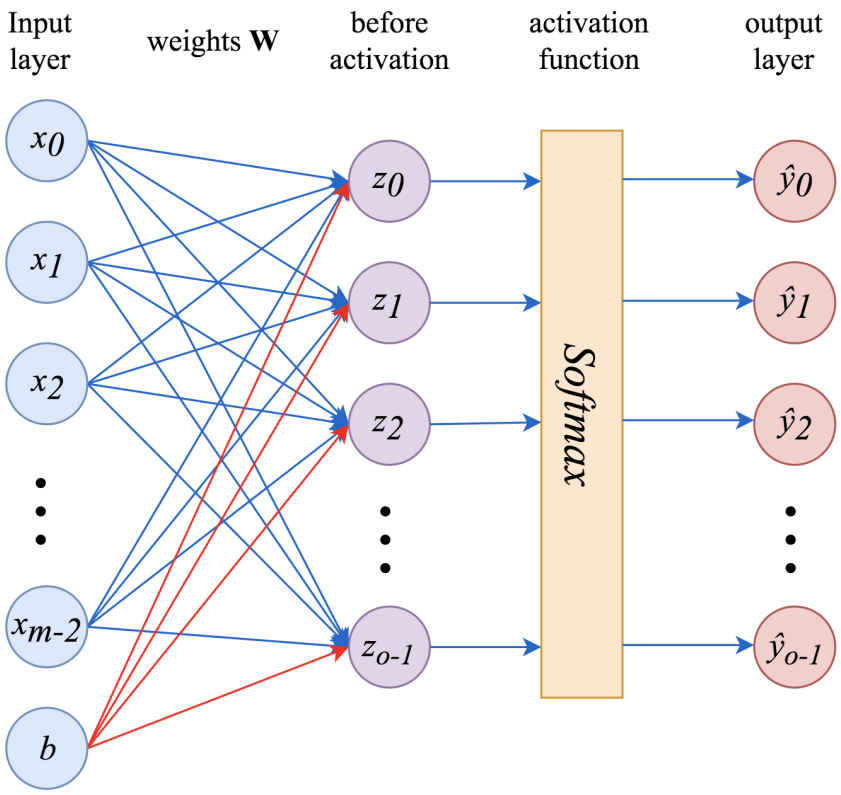

Figure 2: Architecture of the policy network π. The network maps a state of the environment $\mathbf{x}$ to a probability distribution of actions to be taken $\hat{\mathbf{y}}$.

The output vector $\hat{\mathbf{y}}$ is the result of the forward-propagation operation. The output vector is the same as the probability distribution of choosing an action for all $a \in \mathcal{A}$:

$$\hat{\mathbf{y}} = \pi(A|S, \boldsymbol{\theta})$$

The output vector has a one-to-one mapping to the action space, so we have

$$||\hat{\mathbf{y}}|| = ||\mathcal{A}|| = o.$$

The matrix $\mathbf{W}$ represents the weights of the neural network connecting the input nodes with the output nodes. These weight correspond to the parameters $\boldsymbol{\theta}$ that have to be optimised during the learning process. Since there are m input nodes and o output nodes, the size of the matrix is $m \times o$.

To better describe the feed-forward operation, let us rewrite the function as follows:

$$\hat{\mathbf{y}} = softmax\,(\mathbf{z})$$
$$\mathbf{z} = \mathbf{x} \cdot \mathbf{W}.$$

Here the vector $\mathbf{z}$ is introduced, which is the result of the dot product of the input vector $\mathbf{x}$ with the matrix $\mathbf{W}$. The *softmax* function takes as input the vector $\mathbf{z} \in \mathbb{R}^o$ and normalises it into a probability distribution consisting of o probabilities. The *softmax* function is defined by the formula

$$softmax\,(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} = \hat{y}_i.$$

3.3.2 The Gradient of the Policy

For updating the parameters of the network, the gradient of the policy for the selected action has to be computed, $\nabla_\theta \pi(A|S, \boldsymbol{\theta})$. To obtain the gradient of the function $softmax\,(\mathbf{x} \cdot \mathbf{W})$, the Jacobian matrix of the softmax function is computed. The Jacobian matrix contains the derivatives for all the actions in the action space, and, since we are only interested in the action that was selected by the agent, only the row corresponding to the chosen actions is selected. By application of the chain rule, the dot product of vector extracted from the Jacobian matrix is multiplied by the input vector $\mathbf{x}$ yields the gradient of output $\hat{\mathbf{y}}$.

3.4 Episodes

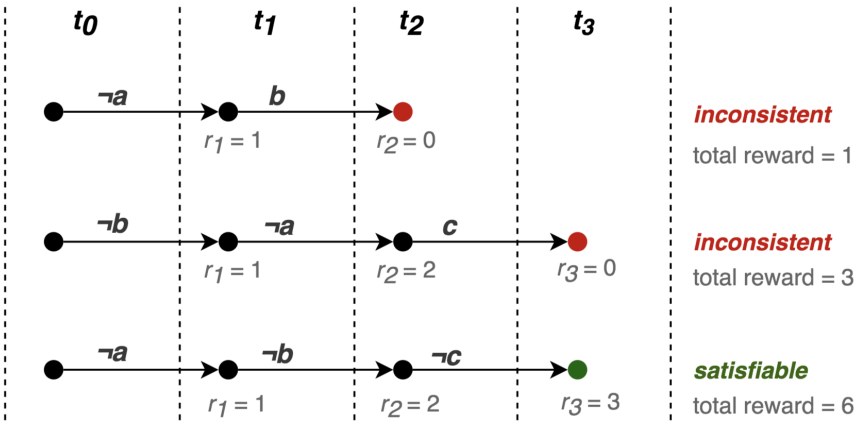

Figure 3: Example episodes for input problem with $CNF = \{\neg a \wedge \neg b \wedge \neg c\}$. The arrows represent the actions taken by the agent (guessing the truth value of one variable) and the transition from one state to the next. The number below the dots represent the reward received in each given time step.

An episode is a sequence of actions taken by the agent that produces a change in the state of the environment after each action. In the context of the DeepSAT model, an episode is a sequence of assignments of the truth values of the input variables made by the agent. One episode consists of a discrete number of time steps t. When the episode begins, $t = 0$, no variables have an assigned truth value. For every time step t in the sequence, the agent assigns the truth value of exactly one input variable. After the agent assigns the value of an input variable, the environment computes the reward and checks if this is consistent with the input CNF. If the environment finds that the assigned variables are consistent with the input CNF, then the next time step, $t + 1$, is computed. The episode terminates when the environment finds inconsistency between the variable assignments and the input CNF, or when all the variables have been assigned and this assignment is consistent with the input CNF (the input problem is satisfiable).

4 Implementation

The DeepSAT model is implemented using the Python programming language, with the use of the external package NumPy for vector and matrix operations. Two main classes are implemented for the model: Agent and Environment. The main functions of these classes are described below. See

Appendix 1 and 2 for the complete code of the program. Additionally, two baseline agent classes are implemented for comparing the performance.

4.1 Functions of the Environment Class

1. Method: Init

   Input: *num_var* : Integer, CNF : List of Integer Array.

   This method stores the input CNF and the number of variables for further use.

2. Method: Consistent

   Input: *instances*: Dictionary(Integer, Boolean), CNF : List of Integer Array.

   Output: *consistent*: Boolean, *sat*: Boolean.

   This method evaluates the input CNF with the instantiated variables to look for contradictions. If a contradiction has been found the value of *consistent* is set to $False$, otherwise the value of *consistent* is set to $True$. If the variable *consistent* is $True$, and the number of variables in the *instances* dictionary is equal to the number of input variables, then the CNF is satisfiable and the variable *sat* is set to $True$.

3. Method: Reward

   Input: *consistent*: Boolean.

   Output: *reward*: Integer.

   Computes and returns the reward given by the formula described in section 2.

4. Method: Reset

   This method sets the state of the environment to the initial state.

5. Method: Step

   Input: action: Integer.

   Output: *reward*: Integer, *consistent*: Boolean, *sat*: Boolean.

   This method changes the state of the environment with respect of the input *action* variable. This new state is verified to be consistent with the input CNF by invoking the *Consistent* method. After, the reward is obtained by invoking the *Reward* method. If the state is observed to be inconsistent, the *Reset* method is invoked. Returns the *reward*, the observed consistency of the state and a variable stating if the input CNF is satisfiable.

4.1.1 Functions of the Agent Class

1. Method: Init

   Input: *environment*: Environment, *learning_rate*: Float, *discount_rate*: Float, *random_weights*: Boolean.

   This method initialises the weights of the parameters and stores the environment object and the hyper parameters. The weights can be initialised at random, if the *random_weights*

parameter is set to $True$, otherwise, all weighs are initialised with value of 0.5.

2. Method: Policy

   Input: *state*: Integer Vector, w: Float Matrix.

   Output: y: Float Vector.

   This method computes the feed forward propagation of the policy and outputs the probability distribution over the action space.

3. Method: Choice

   Input: y: Float Vector.

   Output: *action*: Integer.

   This method stochastically selects an action given the probability distribution of the input vector. An observation to be made is that the input vector contains the probabilities for all actions in the action space; However, this method only selects an action from the subset of actions corresponding to variables that have not been previously selected.

4. Method: Gradient

   Input: y: Float Vector, *action*: Integer, *state*: Integer Vector.

   Output: *grad*: Float Vector.

   This method computes the gradient of the policy with respect to the selected action.

5. Method: Reinforce

   Output: *sat*: Boolean, *steps*: Integer.

   This method obtains the probability distribution of the action space by invoking the *Policy* method, chooses one actions by invoking the *Choice* method, performs the actions on the environment by invoking the method *Step* of the *Environment* class, computes the gradient of the policy by invoking the *Gradient* method, calculates the natural logarithm of the gradient and stores it, alongside the reward received by the *Environment* class. This process is repeated until the episode terminates, either by observing that the state is inconsistent with the input CNF, or by observing that the input CNF is satisfiable. After the episode has ended, the parameters are updated following the REINFORCE operation described in section 2. The method returns the number of steps of the episode and a boolean variable stating if a solution for the SAT problem has been found.

6. Method: Train

   Input: *episodes*: Integer.

   Output: *sat*: Boolean, *rollouts*: Integer.

   Generates episodes, by invoking the Reinforce method, until a maximum number of episodes is reached or until a solution has been found for the SAT problem. Returns the total number of episodes run and a boolean value stating if a solution has been found.

4.2 Baseline Agent Classes

4.2.1 Random Agent

This agent selects an action from the action space at random.

4.2.2 Greedy Agent

This agent is similar to the *Agent* class described above, except for the *Choice* method. The Greedy Agent deterministically selects an action by choosing the action with the highest probability

$$action = \underset{a}{argmax}\ \pi(a|s_t).$$

5 Findings

5.1 Evaluation

5.1.1 Methodology

For testing the models, four input problems are selected. Three of these problems are generated at random using an online resource called the *Tough SAT Project*[1]. The randomly generated problems have respectively 20 input variables with 80 clauses, 30 input variables with 100 clauses, and 40 input variables with 200 clauses; These problems will be referred here as 20SAT, 30SAT and 40SAT. A fourth input problem is used for testing, which will be reffered to as 105SAT. 105SAT is a benchmark, very hard, combinatorial problem for block design, this problem has 105 input variables with 320 clauses.[8] All four input problems are satisfiables.

For measuring the performance, each problem was executed by each of the three agents (DeepSAT, Random and Greedy), collecting the number of steps of each episode during a training execution. This process is repeated 100 times for each agent-problem pair, thus the evaluation is performed with 100 samples for each pair. A maximum of 100.000 episodes is assigned to each agent for solving a problem, if this number is reached, then the program halts and the agent is said to be unsuccessful in solving the problem.

Additionally, the performance is evaluated (only for the DeepSAT model) for different options of hyper parameters. One hundred samples are collected for each hyper parameter test case, and the results are presented as an average of this.

5.1.2 Results

The DeepSAT model was successful in finding solutions for the 20SAT, 30SAT and 40SAT problems. The model demonstrates a learning curve, where the number of variables guessed in each episode increases as time progresses. The model, however, was not successful at finding a solution for the hard 105SAT problem. For the 105SAT case, the model has a steep learning curse at the beginning of training, but this learning curve flattens after some time, making the bot to stop learning. There are many potential problems that could make it hard for the model to find solutions to larger, or harder, problems, some of which are discussed in the next section.

Figure 4 plots the learning curve of the three bots. It can be observed a clear learning curve for DeepSAT, specially as the problem gets larger. In comparison, the random bot has a rather flat learning curve, as it is to be expected. The greedy bot has some learning at the beginning of training, but after a short time it selects the same sequence of actions repetitively, so there is no further progress after this point, and its learning curve becomes completely flat.

Table 1 shows the performance of each bot for each respective problem. The DeepSAT is successful at finding solutions in (almost) all runs of the 30SAT and 30SAT problems, doing so

[1] https://toughsat.appspot.com/

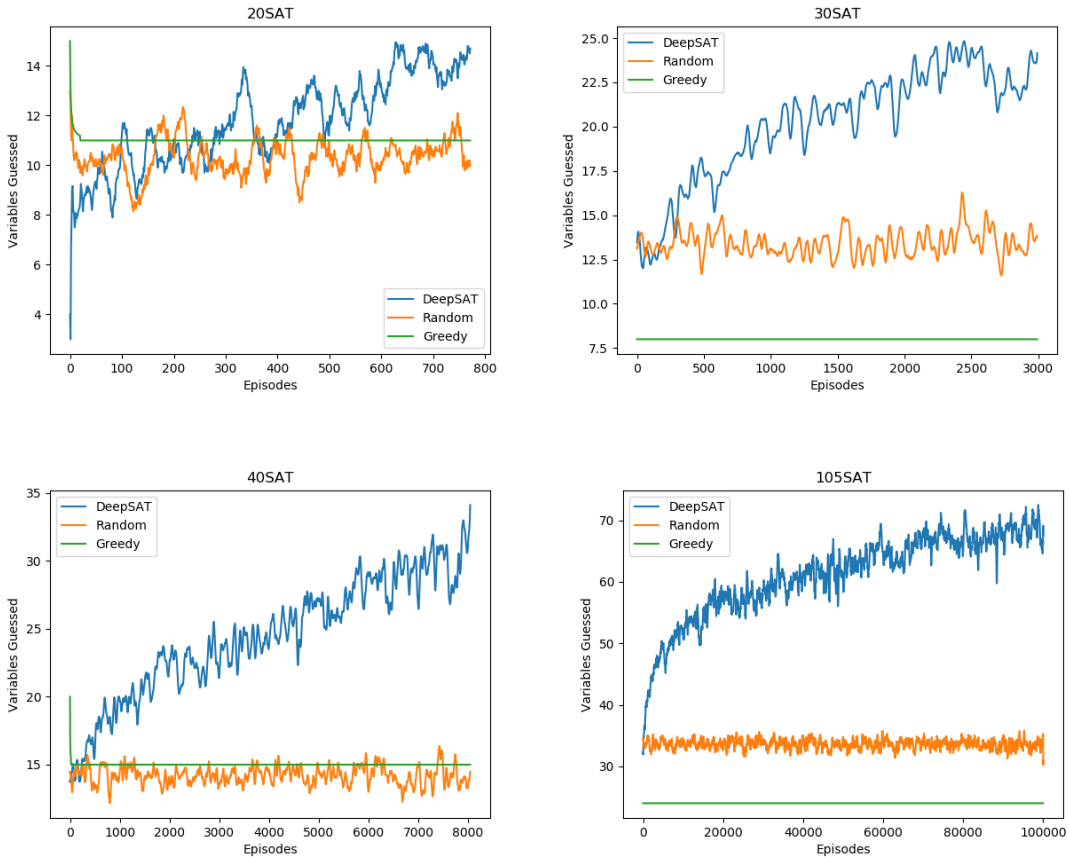

Figure 4: Learning curve for the three agents. The curves are plotted up to the episode number where one of the agents first finds a solution (or maximum number of episodes is reached for the 105SAT case). It can be appreciated that the margin between the learning curves become larger as the input problems become larger as well.

	20SAT		30SAT	
	Episodes (mean)	Solved	Episodes (mean)	Solved
DeepSAT	736	100%	2873	97%
Random	8321	100%	52388	26%
Greedy	-	0%	-	0%
	40SAT		105SAT	
	Episodes (mean)	Solved	Episodes (mean)	Solved
DeepSAT	11971	74%	-	0%
Random	-	0%	-	0%
Greedy	-	0%	-	0%

Table 1: Performance measure of learning agents for training executions with a limit of 100000 episodes. One hundred samples were taken for each agent-problem pair.

much faster than the random bot. The random bot does not find solutions for the 40SAT problem, whereas DeepSAT finds solutions for this problem in 74% of the samples. The greedy bot finds no solution for any of the problems.

5.1.3 Hyper Parameter Evaluation

The DeepSAT agent takes as argument three hyper parameters before training. These hyper parameters are:

- α : the learning rate which determines the size of the *step* for updating the weights,
- γ : the discount rate which indicate how much weight is given to future rewards, and
- $random\_weights$: which indicates whether the weights are random at initialization, or if they will all have equal weight (0.5).

Figure 5 plots the performance of the model using different measures of hyper parameters in solving the 40SAT problem. For the α hyper parameter, there is a clear better performance at $\alpha = 0.0025$, which was the value used for evaluation. For the γ and $random\_weights$ hyper parameters, there is not such a clear distinction. However, for the evaluation these hyper parameters were set as $\gamma = 0.99$ and $random\_weights = False$.

5.2 Problems

Although the DeepSAT model finds solutions for random problems with a limited number of input variables and clauses, the models does not always converge on solutions for satisfiable problems. For a reinforcement learning model to be able to efficiently tackle NP-Complete problems, a number of these problems, in the context of DeepSAT, are discussed in this section.

5.2.1 Problem 1: Expensive computation of episode steps

During an episode roll-out, at each time step t, the state has to be checked for consistency with the input CNF to determine if the time step $t + 1$ can be computed. To make this consistency check, the environment walks trough all the clauses of the input problem. This is problematic, specially when the number of variables and clauses is high. The result is a slow execution of the program during training.

This problem, however, may be overcome if the learning curve of the agent does not deaccelerate during training. If the learning curve were to maintain a steep slope as the number of episodes

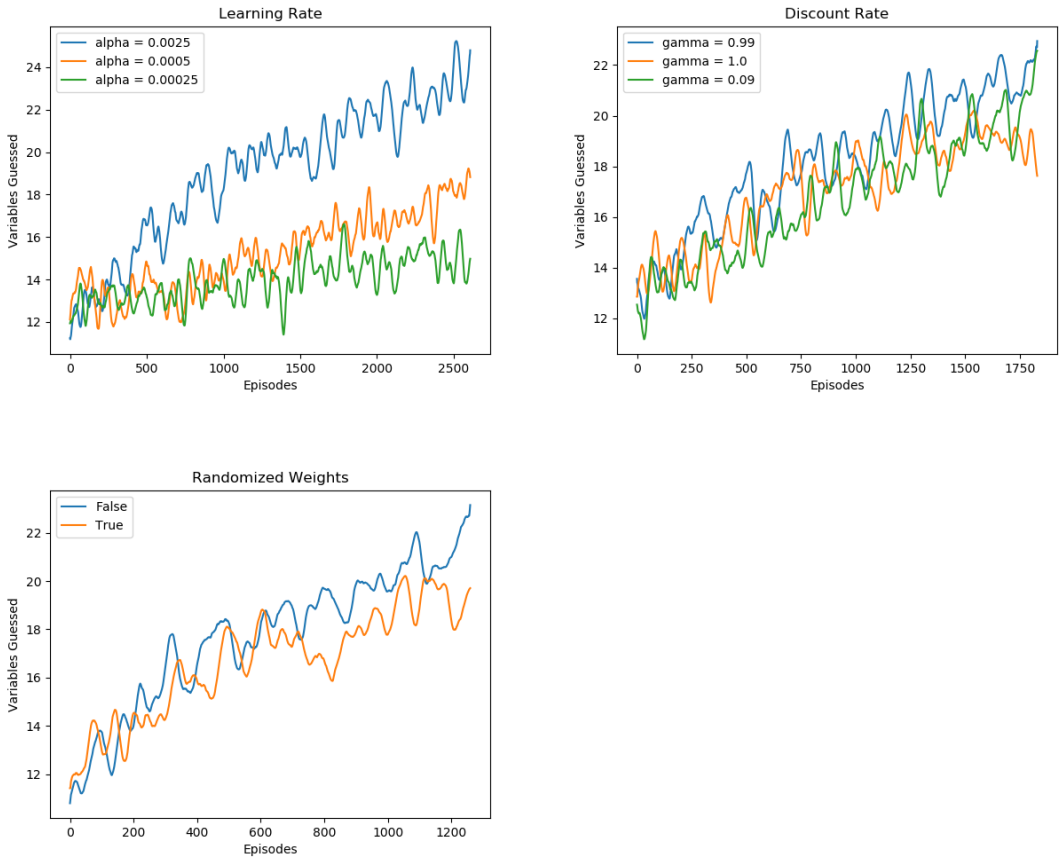

Figure 5: Hyper parameter evaluation. Learning curve of DeepSAT for different measures of learning rate and discount rate, and whether the initial parameters are randomized.

executed progress, this polynomial cost of checking for consistency could be afforded if a solution is found fast enough.

5.2.2 Problem 2: Exploration vs. Exploitation

The *Exploration vs. Exploitation Problem* is a fundamental problem in reinforcement learning that has been intensively studied by mathematicians for many decades, yet it remains unresolved.[6] As the agent progresses in training, and maximizes the expected rewards by its choice of actions, it will become more prone to choose actions that are already known to produce highest rewards. The problem here is that there may be different sequence of actions that result in even higher rewards, but the agent moves away from this selection of actions and settles for the known rewards, that is, it exploits the known rewards. In order to find new higher rewards, the agent would have to explore new sequences of actions, but this cannot be done without first going through paths that yield lower rewards.

In the context of the DeepSAT model, the agent does not always converge on solutions because it learns sequences of actions that guesses an elevated number of variables, but not all the variables. For example, in one instance while training on the 105SAT problem, the agent found a sequence of actions that guessed correctly 100 variables, but no solution was to be found with this interpretation of the variables; However, the agent persisted on choosing this sequence of action, thus stop learning and failing to find a solution for the problem.

5.2.3 Problem 3: No conflict resolution

Consider the hypothetical scenario where there exists an input problem with clauses: $(A \vee \neg B) \wedge (C) \wedge (B)$. Now suppose that the agent selects the sequence of actions $(\neg A \rightarrow C \rightarrow B)$, then the interpretation of variables become inconsistent with the input clauses. In this sequence of actions, the agent will receive a positive reward for the first two actions $(\neg A \rightarrow C)$, and no reward for the last action (B). However, the mistake of the agent was in the selection of the first action $(\neg A)$ and the agent got reinforced on the selection of this action anyways.

This is a problem that can also impede the agent to find effective solutions. There exists known algorithms that perform *conflict driven clause learning*, and backtracking, e.g. *Davis-Putnam-Logemann-Loveland algorithm*. These approaches could be a way to tackle the problem. In further research, some intuition from these known algorithms may be borrowed to overcome this problem.

6 Conclusions

Although the version of DeepSAT presented in this paper is not able to compete with state-of-the-art SAT solvers, we have observed that, nevertheless, a machine learning approach for SAT solving is a feasibility. The field of deep learning, and machine learning in general, is broad and there is an ample supply of models that may also fit NP-Complete problems, in isolation or in combination with the model presented in this paper. Let the DeepSAT model be taken as a benchmark and an example of how NP-hard problems can be approached using machine learning. As we have seen in recent years, machine learning have been doing impressive leaps in advancement, and one can speculate that this trend will continue over the years, which makes appropriate to incorporate classical logic problem to the context of this growing field. This project was successful in addressing the research question: *Can solutions for the SAT problem be learned?*. While with limited capabilities, a clear and elegant learning curve was demonstrated by the agent, which suggests that there exists potential in the intersection between NP-Complete problems and machine learning. Perhaps, if further research on this topic proves successful, the pervasive belief held by experts on the *P vs. NP* debate may take a more optimistic turn.

References

[1] The configurable sat solver challenge (cssc). *Artif. Intell.*, 243(C):1–25, February 2017.

[2] Emile Aarts and Jan K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[3] Holger H Hoos et al. Beyond programming: the quest for machine intelligence. 2017.

[4] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[5] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[6] R.S. Sutton. *Reinforcement Learning*. The Springer International Series in Engineering and Computer Science. Springer US, 1992.

[7] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[8] Allen Van Gelder and Ivor Spence. Zero-one designs produce small hard sat instances. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, pages 388–397, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

A Source Code: Environment Class

```
import numpy as np
import collections

class environment():
  working_CNF =  []
  input_CNF = []
  state = []
  instances_V  = {}

  def __init__(self, num_of_var, CNF):
    self.state = np.zeros(2 * num_of_var)
    self.input_CNF = CNF
    self.working_CNF = [c.copy() for c in self.input_CNF]

  def get_state_with_bias_node(self):
    return np.append([1], self.state)

  def get_number_of_variables(self):
    return int(len(self.get_state())/2)

  def get_action_space_size(self):
    return len(self.state)

  def get_state(self):
    return self.state

```

```
  def reward(self, consistent):
    if not consistent:
      return 0
    return sum(self.state)

  def consistent(self, instances):
  # Returns consistent = False if a contradiction has been found in the
    clauses
  # Retruns consistent = True otherwise
  # Returns sat = True if CNF is satisfiable
    sat =  False
    CNF = self.working_CNF

    for c in CNF:
      size = len(c)

      for literal in c:
        key = abs(literal)
        value = literal > 0

        if key in instances:
          if value == instances[key]:
            CNF.remove(c)
            break
          else:
            size -= 1
        else:
          break

      if size == 0:
        return False, sat

    if (len(instances) == self.get_number_of_variables()):
      sat = True

    return True, sat

  def step(self, action):
    self.state[action] =  1
    self.instances_V[int(action/2)-1] = (action % 2 == 0)

    consistent, sat = self.consistent(self.instances_V)
    reward = self.reward(consistent)

    if not consistent:
      self.reset()

    return reward, consistent, sat

  def reset(self):
    self.state = np.dot(self.state, 0)
    self.instances_V = {}
    self.working_CNF = self.working_CNF = [c.copy() for c in self.input_CNF]
```

B Source Code: Environment Class

```
import numpy as np
import copy
import environment
import statistics

class agent():
  env = None
  w =  None
  name = "Standard Agent"
  #Hyperparameters
  NUM_EPISODES = 10000
  LEARNING_RATE = 0.0025
  GAMMA = 0.99
  random_weights = False

  def __init__(self, environment, LEARNING_RATE = 0.0025, GAMMA = 0.99,
    random_weights = False):
    self.env  = environment
    self.LEARNING_RATE = LEARNING_RATE
    self.GAMMA = GAMMA
    self.random_weights = random_weights
    self.initialize_weights()

  def reset(self):
    self.initialize_weights()
    self.env.reset()

  def initialize_weights(self):
    n = self.env.get_action_space_size()

    if self.random_weights:
      self.w = np.random.uniform(-1,1,(n + 1,n))
    else:
      self.w = np.full((n + 1, n), 0.5

  def policy(self,state,w):
  # Performs the softmax forwards propagation
  # Gives as output a probability distribution
    z = state.dot(w)
    exp = np.exp(z)
    return exp/np.sum(exp)

  def softmax_grad(self,softmax):
  # Vectorized softmax Jacobian
      s = softmax.reshape(-1,1)
      return np.diagflat(s) - np.dot(s, s.T)

  def gradient(self, y, state, action):
  # computes the gradient of the policy
    dsoftmax = self.softmax_grad(y)[action,:]
    return state[None,:].T.dot(dsoftmax[None,:])

  def get_legal_actions(self):
  # Auxiliary method for choice()
```

```
54      state = self.env.get_state()
        actions = []
56
        for i in range(0,len(state),2):
58        legal = 1 - (state[i] + state[i+1])
          actions.append(legal)
60        actions.append(legal)
        return actions
62
    def choice(self, y):
64  # Creates a new probability distribution excluding the illegal actions
    # Returns random choice of action (given the distribution), and name of
      variable
66
        legal_actions = self.get_legal_actions()
68      legal_actions_probs = np.multiply(legal_actions, y)

70      average_difference = (sum(y) - sum(legal_actions_probs)) / sum(
      legal_actions)
        probs = [((legal_actions_probs[i] + average_difference)*legal_actions[i
      ]) for i in range(0,len(legal_actions))]
72
        return np.random.choice(self.env.get_action_space_size(), p=probs)
74
    def train(self, episodes, verbose):
76  # Executes rollouts until solution is found or until max_num of episodes
      reached
        sat = False
78      scores = []

80      for e in range(0,episodes):
          sat, step = self.reinforce()
82        scores.append(step)

84        if ((e+1) % 100 == 0):
            if verbose:
86            print("LOG: Rollouts completed: ", (e+1), "Guessed: ", statistics.
      mean(scores[-100:]))

88        if sat:
            break
90
        return sat, len(scores)
92
    def reinforce(self):
94    grads = []
      rewards = []
96
      while True:
98      state = self.env.get_state_with_bias_node()

100     # FORWARD PROPAGATION
        probs = self.policy(state,self.w)
102
        # choose action to take
104     action = self.choice(probs)
```

```
106         # agent performs action on environment, gets observations back
            reward, consistent, sat = self.env.step(action)
108
            # COMPUTING THE GRADIENT
110         grad = self.gradient(probs,state, action)
            log_grad = grad / probs[action]
112
            # store data for parameters update
114         grads.append(log_grad)
            rewards.append(reward)
116
            if not consistent:
118           break

120         if sat:
              return True, reward
122
        for i in range(len(grads)):
124       # Update the weights given by the REINFORCE formula
          # Loop through everything that happend in the episode and update
        towards the log policy gradient times **FUTURE** reward
126
          self.w += self.LEARNING_RATE * grads[i] * sum([ r * (self.GAMMA ** t)
        for t,r in enumerate(rewards[i:]) ])
128
        return False, rewards[-2]
```