Decision Tree Boosting for Reinforcement Learning

Project Report for CSCI 699 HRI, Prof. Nikolaidis, USC Spring 2021

Nate Sands
njsands@usc.edu

Abstract

Modern autonomous systems rely on machine learning methods, which offer impressive results but often at the cost of interpretability. Of all the ML paradigms, decision tree learners are perhaps the easiest models to interpret. This report outlines a method for training reinforcement learning agents whose controllers are boosted forests of decision trees. It draws inspiration from the AdaBoost framework for supervised learning. Using an oracle agent to narrow the search space and define an objective loss function, it greedily chooses trees which reduce the weighted sum of losses across roll-outs. Experiments in the CartPole RL environment indicate that such a forest may significantly outperform the oracle, even when its constituent trees individually are poor-performing. The use of shallow trees is emphasized to derive controllers which use a minimal number of rules, thereby enhancing their interpretability.

1 Introduction

Trust in autonomous systems derives from the ability to predict their future actions [9], and explain their past actions [10]. This is the core idea behind the (sometimes interchangeable) conceptions of 'interpretability' and 'explainability' in artificial intelligence research. Interpretability is a central concern when human safety is involved, as is the case with robotic systems that drive vehicles, or interact with humans socially or as co-workers.

Deep Reinforcement Learning (DRL) has shown impressive results, particularly in the realms of control problems [6] and game play [11, 12]. It has been used to help robots deftly navigate crowded pedestrian environments with attention to social nuance [5]. It has been the focus of much research in the field of self-driving vehicles [1]. Yet the underlying neural networks in these systems can be difficult to interpret and verify.

In an effort to make learned policies more transparent researchers have explored the idea of using a neural oracle to guide the synthesis of interpretable controllers. In [14] the authors use a DNN to train simple PID-inspired controllers for a simulated race car. In [2] the authors use a neural oracle to synthesize decision tree controllers for classic RL problems like CartPole and Pong. They demonstrate that some properties can be verified for decision trees that are more difficult to verify for DRL policies.

This report outlines a method for synthesizing CartPole agents based on forests of decision trees. It draws inspiration from the AdaBoost framework for supervised learning. The algorithm proposed, called BetaBoost, builds a forest using an oracle agent to narrow the search space. A system of weights is used to recruit trees that perform well in states where the forest is under-performing. This allows a strong agent to be built from several comparably weaker ones.

The experimental section shows that a forest controller composed of of trees with a maximum depth of 2 can outperform a Sarsa-trained oracle by a factor of 20.

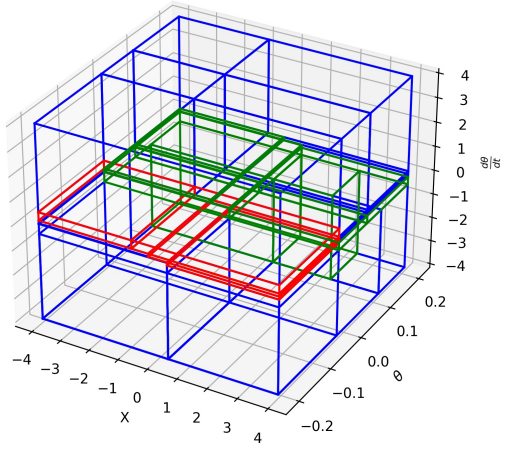

Figure 1. Decision regions in CartPole space (excluding dimension of position).

2 Background and Methods

2.1 Problem Statement

The CartPole environment is a finite-horizon model-free Markov Decision Process, $M = (S, A, R)$. The state space S consists of tuples $(x, \dot{x}, \theta, \dot{\theta}) \in \mathbb{R}^4$, where x is the position of the cart, θ its pole angle, $\dot{x}$ the cart velocity, and $\dot{\theta}$ the pole's angular velocity. The agent receives a reward $R = 1$ for every time step the pole remains upright and the cart remains within bounds. The agent has two possible actions $a \in A$: apply force to the cart in the positive-x direction, or in negative-x direction. The goal is to synthesize a policy $\pi : S \rightarrow A$ that maximizes the reward R. The classic CartPole problem is considered solved if an agent achieves an average reward $R = 200$ over the course of 100 consecutive episodes. But here the objective is to find an agent that achieves the highest possible average up to a maximum of 10000.

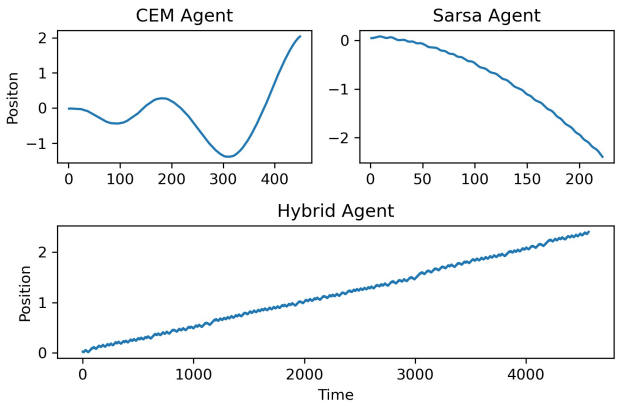

Figure 2. Individual performance of two CartPole agents versus a hybrid model which combines them.

2.2 Reinforcement Learning Paradigms

As a basis for decision tree synthesis, the algorithm uses a version of the cross-entropy method for reinforcement learning [3] to train CartPole agents. Each CEM agent is defined by a vector of weights $\omega = (\omega_1, \omega_2, \omega_3, \omega_4)$ plus a bias term ω_b. The agent has a choice of two actions to take in any state, and this choice is determined by the sign of the dot product of the weight vector with the state variables, plus the bias term. One of the oracles (ZeroDOracle) used in the experiments is a CEM agent [1] The second oracle (SarsaOracle) was trained using the Sarsa algorithm [13] over 100,000 iterations.

2.3 Synthesis of Decision Tree Controllers

Training a decision tree to act as controller for CartPole is straightforward. A previously trained agent (in the experiments, a CEM agent) generates a history consisting of a few thousand state-action pairs. The CART algorithm [7] builds a tree from this data that 'classifies' states as either 'push cart left' or 'push cart right', in a way that mimics the action of the original agent. Fidelity to the original agent is not a strict requirement, however, and will vary according to the quality of the sampled history and the maximum tree depth allowed. I considered trees with a maximum depth ranging from 2 up to 5 in order to reduce the complexity of the synthesized controller.

3 Method

3.1 Motivation

I observed that two different CartPole controllers could sometimes be merged into a hybrid controller that earned higher rewards. In one case, I synthesized decision trees using histories sampled from a Sarsa-trained model and another from a CEM-trained model. Figure 1 shows the their shared decision space. The blue regions show where the two models happened to agree, and the green and red regions are where they disagree. I added the condition that whenever the two agents disagree, a leftward force should be applied. This resulted in roll-outs which earned rewards an order of magnitude greater. Figure 2 shows the difference in performance. (The program specifying the hybrid model is given in the appendix.)

3.2 AdaBoost

The idea of 'ensemble methods' is a familiar one in supervised learning. AdaBoost[8] is a general framework for building accurate classifiers from an ensemble of "weak learners". In the setting of a binary classification problem, a committee C comprising M classifiers is selected. For any given sample x_i, each classifier c_m gives a response $c_m(x_i) \in \{-1, 1\}$. Each member is assigned a coefficient α_m representing the weight given to their individual vote. The final decision of the committee is determined by the sign of the weighted sum of votes, i.e.

$$C(x_i) = \text{sgn}(\alpha_1 c_1(x_i) + \alpha_2 c_2(x_i) + \cdots + \alpha_M c_M(x_i)). \quad (1)$$

The model assigns a weight w_i to each sample in the data set. If the committee misclassifies a sample, its weight is increased. A new member of the committee is selected so as to minimize the sum of weights corresponding to misclassified samples.

The model assigns an error value ϵ_m to classifier c_m equal to

$$\epsilon_m = \frac{W_e}{W} \quad (2)$$

where W_e is the sum of weights corresponding to the samples that c_m misclassifies, and W is the sum of *all* weights. It can be shown that the value of α_m that minimizes the overall error of the committee is equal to

$$\alpha_m = \frac{1}{2} \ln(\frac{1 - \epsilon_m}{\epsilon_m}). \quad (3)$$

When c_m is added to the committee the weights are updated according to

$$w_i \leftarrow w_i e^{\alpha}, \quad (4)$$

if c_m incorrectly classifies x_i, and

$$w_i \leftarrow w_i e^{-\alpha} \quad (5)$$

otherwise. The overall effect is to focus on areas of inaccuracy when the size of the committee is expanded. It is important to note that the above formulation imposes the constraint that every classifier added to committee must have an error less that one-half (the point at which the sign of α_m changes).

[1] ([−0.10417859, −1.48485568, −2.45623463, −2.75488367], −0.01961685693590471). It was trained to stay at 'zero distance' from the origin using a recursive reward strategy.

3.3 BetaBoost

The BetaBoost algorithm is a heuristic method for synthesizing decision tree controllers for CartPole using the AdaBoost framework as a model. Essentially, it approaches the reinforcement learning problem as a supervised learning problem. Every 4−tuple in the CartPole state space is treated as a 'sample', and the 'label' is the expected cumulative reward, or state-value, of an CartPole agent following an ideal policy beginning in that state. In other words, each state $s \in S$ has label $V(s)$.

Just as a supervised learning problem requires a set of samples with known labels to train a classifier, the RL setting requires an oracle to act as a witness to the value $V(s)$ of each state. This role can be played by any previously trained agent, using any training regime (CEM, Q, Deep Q, etc.). The average reward earned by an oracle over multiple roll-outs starting from a state is taken to be a good approximation of the state value. This allows us to set up a cost function in order to measure the accuracy of any new agent we are considering.

In the context of building a forest of decision trees, the cost function lets us add trees that reduce the loss at particular states, using a system of voting coefficients and weights similar to those in AdaBoost.

In broad outline, the algorithm proceeds as follows:

Given an oracle O, generate a history H of states visited by O during multiple episodes of CartPole (each episode beginning at the usual starting state). Determine a distribution over states $P(S = s|H, O)$ using, say, a kernel density estimate. Sample N states from this distribution to form the training set $S_O = s_1, \ldots, s_N$. For each state $s_i \in S_O$, perform roll-outs of the oracle starting from that state, and record its mean reward R^i_O. This is the oracle's best approximation of $V(s_i)$. The set $R_O = \{R^1_O, \ldots, R^{(N)}_O\}$ then becomes the set of labels for S_O.

The next step is to select a pool of candidate trees. The space of such trees is infinite, but one of the functions of the oracle is to bound that search space. The naive assumption is made that a set of trees trained from states that the oracle is likely to visit will on the whole behave similarly to the oracle on all states. So for every state in S_O a shallow decision tree is trained using the procedure outlined in section 2.3.

Once a pool of trees T has been assembled, initialize weights $w_i = 1$ for each state $s_i \in S_O$. For any tree t_m, let $R_{t_m} = \{R^{(1)}_{t_m}, \ldots, R^{(N)}_{t_m}\}$ denote the set of rewards earned by t_m starting from all states s_i, and let L_i denote the loss $L_i = R^{(i)}_O - R^{(i)}_{t_m}$.

On the first iteration, select the tree t_1 from T that minimizes the sum

$$\sum_{i=1}^{N} w_i \mathbb{I}[L_i > 0], \tag{6}$$

(i.e. the sum of all weights corresponding to states from which t_1 earned less than the oracle.)

Calculate the error factor

$$\epsilon_i = \frac{R^{(i)}_O - R^{(i)}_{t_m}}{R^{(i)}_O}. \tag{7}$$

(Note that unlike (2) which calculates the percentage of weighted error over all states, (7) calculates a separate error factor for *each* state).

Use these factors to update the weights

$$w_i \leftarrow w_i e^{\epsilon_i}. \tag{8}$$

Finally, determine t_1's voting coefficient using a rough comparison of its performance to the oracle's:

$$\beta_1 = \frac{\sum_{i=1}^{N} R^{(i)}_{t_1}}{\sum_{i=1}^{N} R^{(i)}_O}. \tag{9}$$

Repeat these steps until the remaining trees $t_2, \ldots, t_M$ have been added to the forest. The action of the forest is determined by

$$F(s_i) = \text{sgn}(\beta_1 t_1(s_i) + \cdots + \beta_M t_M(s_i)). \tag{10}$$

Note that the quantities associated with the AdaBoost framework have a guarantee of optimality. I make no such presumption about this model. It is a rough estimation without a firm theoretical framework as yet, although there is some empirical support for it.

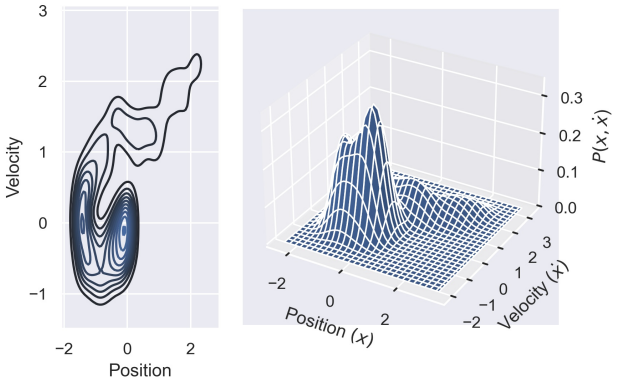

Figure 3. Contours of kernel density estimate for SarsaOracle's states with respect to position and velocity.

4 Experimental Results

I utilized the CartPole environment provided by OpenAI Gym [4] to conduct the experiments. The purpose was to observe the performance of forests grown using the above framework. Performance was measured every time a tree was added to the forest, using the average reward over 100 episodes as a metric. An episode terminated when the cart's

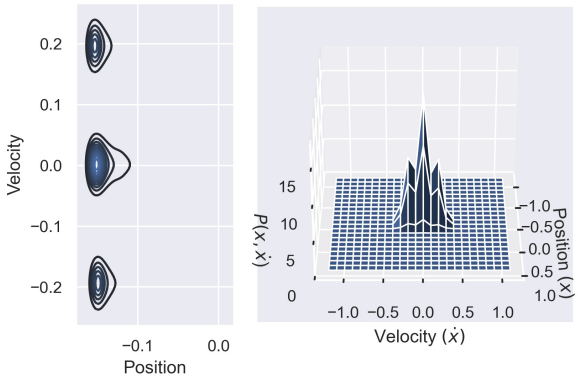

Figure 4. Contours of kernel density estimate for ZeroDOracle's states with respect to position and velocity.

position or pole angle went out of bounds, or when the agent reached a maximum reward of $R = 10000$. Each forest had a maximum tree depth $MaxDepth \in \{2, 3, 4, 5\}$. Forests were allowed to grow up to a maximum size of 10 trees.

The experiments were conducted once each using SarsaOracle and ZeroDOracle. Histories consisting of 10000 state-action pairs from each oracle were used to generate kernel density estimates (see figures 3 and 4). For each forest, 300 states were sampled to form the basis for tree synthesis.

4.1 SarsaOracle

SarsaOracle has an average return of approximately $R = 800$. As figure 5 indicates, the BetaBoosted forests tended to increase in performance as more trees were added, and reached maximum returns between $R = 4000$ and $R = 5000$. Table 2 summarizes these results. Worthy of note is the forest MaxDepth2, which earned a maximum reward almost 20 times the average of any one of its component trees (see table 1). Also note that in one case (MaxDepth5), a forest *suppressed* a tree that performed better than its collective average.

4.2 ZeroDOracle

ZeroDOracle has a flawless record of $R = 10000$ for each of 100 episodes. Thus any candidate tree can never hope to beat it, which diminishes the influence of the weights w_i on tree selection. This provides an interesting contrast with SarsaOracle. On the one hand, ZeroDOracle produced an exceptional *tree* (MaxDepth4, $R = 8000$), but none of the forests gained performance when more trees were added in the way that they did with SarsaOracle. It remains to be seen if revising the algorithm to utilize a system of weights compatible with high performing oracles would restore this behavior.

Table 1. MaxDepth2 Forest

Tree	Beta Value	Mean Reward
0	0.271	226.24
1	0.339	129.87
2	0.280	238.89
3	0.236	95.83
4	0.209	91.32
5	0.292	191.1
6	0.201	84.16
7	0.237	131.6
8	0.271	129.97
	Forest Mean Reward:	4408

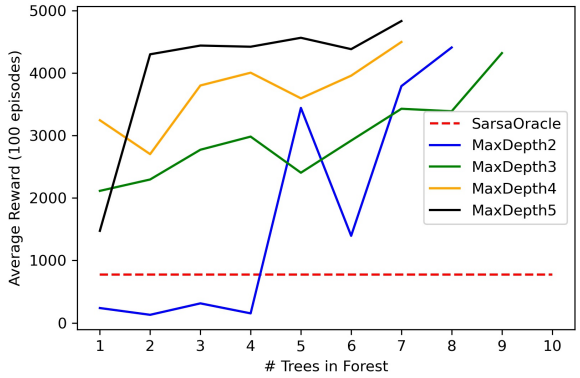

Figure 5. Growth of forests using SarsaOracle. Graphs end at forest size (max size 10) with peak performance.

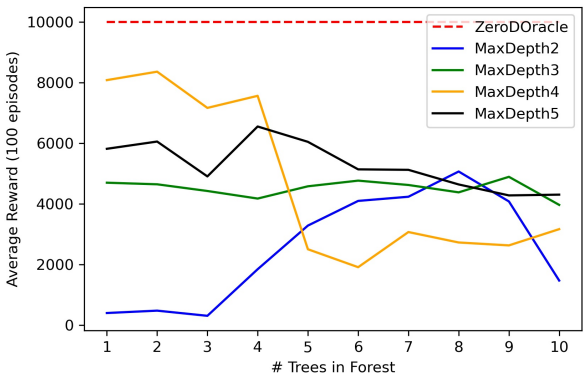

Figure 6. Growth of forests using ZeroDOracle.

5 Conclusion

I have outlined a procedure for generating a CartPole controllers composed of shallow decision trees. In most cases

Table 2. Oracle and Forest Agents

Agent	Avg Reward	Max	Min	Std	Best Tree Avg	# Trees	Nodes
SarsaOracle	772	1345	433	165	-	-	-
MaxDepth2	4408	10000	155	3686	239	8	36
MaxDepth3	4318	10000	438	3519	3848	9	64
MaxDepth4	4496	10000	336	3425	3848	7	139
MaxDepth5	4831	10000	560	3309	5808	6	135

these performed better than their component trees individually, and significantly outperformed one oracle agent used to guide their synthesis. The procedure lacks formal guarantees, however, and it is uncertain if it can used to generate controllers that approximate high-performing oracles. The goal of this research is to explore the synthesis of interpretable controllers that are predictable and verifiable.

References

[1] S. Aradi. 2020. Survey of Deep Reinforcement Learning for Motion Planning of Autonomous Vehicles. *ArXiv* abs/2001.11231 (2020).

[2] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *NeurIPS*.

[3] P. D. Boer, Dirk P. Kroese, Shie Mannor, and R. Rubinstein. 2005. A Tutorial on the Cross-Entropy Method. *Annals of Operations Research* 134 (2005), 19–67.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, J. Schneider, John Schulman, Jie Tang, and W. Zaremba. 2016. OpenAI Gym. *ArXiv* abs/1606.01540 (2016).

[5] Y. Chen, Michael Everett, M. Liu, and J. How. 2017. Socially aware motion planning with deep reinforcement learning. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), 1343–1350.

[6] S. Collins, A. Ruina, Russ Tedrake, and M. Wisse. 2005. Efficient Bipedal Robots Based on Passive-Dynamic Walkers. *Science* 307 (2005), 1082 – 1085.

[7] T. Hastie, R. Tibshirani, and J. Friedman. 2016. *The Elements of Statistical Learning*.

[8] T. Hastie, R. Tibshirani, and J. Friedman. 2016. *The Elements of Statistical Learning*.

[9] Been Kim, O. Koyejo, and Rajiv Khanna. 2016. Examples are not enough, learn to criticize! Criticism for Interpretability. In *NIPS*.

[10] T. Miller. 2019. Explanation in Artificial Intelligence: Insights from the Social Sciences. *Artif. Intell.* 267 (2019), 1–38.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, Andrei A. Rusu, J. Veness, Marc G. Bellemare, A. Graves, Martin A. Riedmiller, A. Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, A. Sadik, Ioannis Antonoglou, Helen King, D. Kumaran, Daan Wierstra, S. Legg, and D. Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (2015), 529–533.

[12] D. Silver, Aja Huang, Chris J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, S. Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (2016), 484–489.

[13] R. Sutton and A. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd. ed.). The MIT Press, Cambridge, MA. 129–130 pages.

[14] Abhinav Verma, V. Murali, R. Singh, P. Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. *ArXiv* abs/1804.02477 (2018).

A Hybrid Agent

```
// hybrid of cem-trained agent and sarsa-trained agent

// each agent votes on an action, which sometimes results in a tie
vote(t, a1, a2) :- agent1_vote(t, a1), agent2_vote(t,a2).
tie(t, a1, a2) :- vote(t, a1, a2), a1 != a2.

// always move left in case of a tie vote
action(t, 0) :- vote(t, a1, a2), a1 + a2 = 0.
action(t, 0) :- vote(t, a1, a2), a1 = 0, a2 = 1.
action(t, 1) :- vote(t, a1, a2), a1 + a2 = 2.
action(t, 0) :- vote(t, a1, a2), a1 = 1, a2 = 0.

// cem agent
agent1_vote(t, 0) :- state(t, _, s1, _, s3), s3 <= -0.013720870949327946, s1 <= 0.35592521727085114,
                                             s3 <= -0.03167750872671604.
agent1_vote(t, 0) :- state(t, _, s1, _, s3), s3 <= -0.013720870949327946, s1 <= 0.35592521727085114,
                                             s3 > -0.03167750872671604.
agent1_vote(t, 0) :- state(t, _, s1, _, s3), s3 <= -0.013720870949327946, s1 > 0.35592521727085114,
                                             s3 <= -0.13628841191530228.
agent1_vote(t, 1) :- state(t, _, s1, _, s3), s3 <= -0.013720870949327946, s1 > 0.35592521727085114,
                                             s3 > -0.13628841191530228.
agent1_vote(t, 0) :- state(t, _, s1, _, s3), s3 > -0.013720870949327946, s3 <= 0.10513358563184738,
                                             s1 <= -0.27255984395742416.
agent1_vote(t, 1) :- state(t, _, s1, _, s3), s3 > -0.013720870949327946, s3 <= 0.10513358563184738,
                                             s1 > -0.27255984395742416.
agent1_vote(t, 1) :- state(t, _, s1, _, s3), s3 > -0.013720870949327946, s3 > 0.10513358563184738,
                                             s1 <= -0.8283147513866425.
agent1_vote(t, 1) :- state(t, _, s1, _, s3), s3 > -0.013720870949327946, s3 > 0.10513358563184738,
                                             s1 > -0.8283147513866425.

// sarsa agent
agent2_vote(t, 0) :- state(t, _, _, s2, s3),  s3 <= -0.44452863931655884, s2 <= 0.11774696037173271.
agent2_vote(t, 1) :- state(t, _, s1, s2, s3), s3 <= -0.44452863931655884, s2 > 0.11774696037173271,
                                              s1 <= 2.3634891510009766.
agent2_vote(t, 0) :- state(t, _, s1, s2, s3), s3 <= -0.44452863931655884, s2 > 0.11774696037173271,
                                              s1 > 2.3634891510009766.
agent2_vote(t, 0) :- state(t, _, _, s2, s3),  s3 > -0.44452863931655884, s2 <= -0.023333095014095306,
                                              s3 <= 0.4449678659439087.
agent2_vote(t, 1) :- state(t, _, _, s2, s3),  s3 > -0.44452863931655884, s2 <= -0.023333095014095306,
                                              s3 > 0.4449678659439087.
agent2_vote(t, 1) :- state(t, _, _, s2, s3), s3 > -0.44452863931655884, s2 > -0.023333095014095306,
                                             s2 <= 0.06978285312652588.
agent2_vote(t, 1) :- state(t, _, _, s2, s3), s3 > -0.44452863931655884, s2 > -0.023333095014095306,
                                             s2 > 0.06978285312652588.
```