LEARNING CONSENSUS

A PREPRINT

Trenton Bricken
Department of Computer Science
Duke University
trenton.bricken@duke.edu

Rohan Reddy
Department of Mathematics
Duke University
rohan.reddy@duke.edu

Yash Patil
Department of Computer Science
Duke University
yash.patil@duke.edu

Kartik Nayak
Department of Computer Science
Duke University
kartik@cs.duke.edu

December 5, 2019

ABSTRACT

Consensus protocols have proven to be immensely valuable across a wide range of domains, most recently and notably with the advent of bitcoin and other cryptocurrencies [1] [2] [3]. Valuable new consensus protocols that are more efficient and Byzantine fault tolerant emerge regularly. However, creating and validating these protocols does not scale well because every protocol relies upon assumptions about the environment and setup. For example, in the method of communication (e.g. Byzantine Broadcast), reliability of the communication channels (e.g. Partial Synchroneity), and agent identifiability (e.g. Public Key Infrastructure). These assumptions can significantly affect the outcome a consensus protocol's fault tolerance and efficiency. Motivated by these difficulties and continued progress in Deep Reinforcement Learning to learn superhuman strategies across a number of very complex games [4] [5], this research asks the question: can we use Deep Reinforcement Learning to test existing protocols for their robustness, and maybe even discover new ones?

***K*eywords** Byzantine Fault Tolerance · Consensus Protocols · Deep Reinforcement Learning · Neural Networks

1 Background and Motivation

Consensus protocols allow for a set of entities to agree upon some value or sequence of values. Consensus protocols today are used in a wide variety of settings, such as blockchains and state machine replication [1]. There are key assumptions behind each consensus protocol, including the type of faults the protocol can handle (Byzantine or crash), the type of communication, the latency of communication between parties, and the security of the communication. As these assumptions change, so too does the number and type of faults that a consensus protocol can tolerate change. For example, the Ben-Or protocol can tolerate only $f < n/5$ Byzantine faults but it can tolerate $f < n/2$ crash faults [6].

Neural networks are complex models inspired by the neurons in the brain that recognize patterns by approximating arbitrary complex functions. They make use of a technique called back-propagation, an application of the chain rule from calculus, to optimize for any given loss function. In the last decade, neural networks have begun to be used extensively in many applications, including speech recognition, handwriting recognition, and object identification in images.

Reinforcement learning is a set of algorithms and theory focused on how agents can learn intelligent behaviour from experience. The agent evaluates their current situation (the environment, or whatever portion of the environment the agent observes) and takes an action which will yield the highest expected reward, this updates the current state that the agent is in, at which point it must decide to take another action until termination of the simulation.

Today, new consensus protocols are conceived of regularly using clever combinations of a reasonably small set of possible ways to communicate with other agents. Because of the large number of possible environmental conditions to model, we believe that every permutation of these actions has not been tried, leaving more interesting and valuable consensus protocols to be discovered. As such, the goal of this research is to see if we can use the latest advances in deep reinforcement learning to not only re-create basic consensus protocols but also discover new ones.

2 Related Work

To the best of our knowledge, there currently exists no other work using deep reinforcement learning to replicate consensus protocols. There has been research using reinforcement learning to develop optimal strategies for iterated prisoner's dilemma games, but these are fundamentally different in that there are no communication rounds between agents make a decision, and each agent is self interested, having no incentive to collaborate with others [7].

Our inspiration for this work comes from research where two agents, Alice and Bob, are able to learn basic forms of cryptography using secret keys in order to communicate with each other, while preventing Eve who received their communication but without the secret key, from being able to decipher the message [8].

3 Methods

Modular Environment The core assumptions that any consensus protocol makes are:

1. The method of communication, this can include: Byzantine Broadcast (where at each round one agent is selected to send messages to all other agents); Byzantine Agreement (where all agents send messages to each other at each round to commit on one value); and Interactive Consistency (where all agents send values to each other to commit to agree upon a tuple of values).
2. The reliability of the communication channels which are either: Asynchronous; Partially Synchronous; Synchronous.
3. The identifiability of the agents. Messages are either: signed with digital signatures using Public Key Infrastructure (PKI), this allows any agent to verify what message was sent from whom; sent via authenticated channels, this gives the recipient knowledge of who sent the current message but not the ability to verify who sent previous messages that agents claim they received and are passing on. Anonymous channels where the agent has no idea who sent the messages it is receiving.

We have built a modular environment where different combinations of the above assumptions can be easily turned on and off. Currently we have implemented Byzantine Agreement, Authenticated Channels, PKI and Synchronous communication with plans to implement additional environmental assumptions in the future.

Training Process

See Figure 1 for an example round of our most simple environment where there are three honest agents. In a given simulation, each agent is initialized with a random value that can be interpreted as the value that they would prefer to commit to. In each round of the simulation, the agents are able to send the value they were initialized with to all of the other agents, or to commit to a value of 1 or 0. The actions decided upon and states received from the other agents all must be encoded as "one-hot" vectors in order to be fed into the neural network which decides the agent's optimal next action. After every agent takes an action, the environment resolves all of them by updating the states of every agent to reflect these actions. A new round begins when the agents get to decide again what action they want to take next. Every simulation terminates when all of the honest parties have committed to a value. At this point, depending on the values the honest parties committed to, a reward or penalty is given to the honest and Byzantine agents which is used to update their neural networks and teach them what actions to perform more or less of in the future.

Agent Rewards

For a consensus protocol to work it must satisfy two criteria:

1. Consistency - all honest parties must commit to the same value.
2. Validity - if all honest parties are initialized with the same value, they must commit to it. This is to prevent the agents from always committing to a particular value, independent of what their initial value is.

If the honest agents satisfy consistency and validity, they receive a +1, otherwise they receive a -1 and vice versa for the Byzantine agents trying to break consensus.

For reasons outlined in Section 4 (Results) we modified these rewards and added additional ones in attempting to get the agents to efficiently learn the consensus protocols.

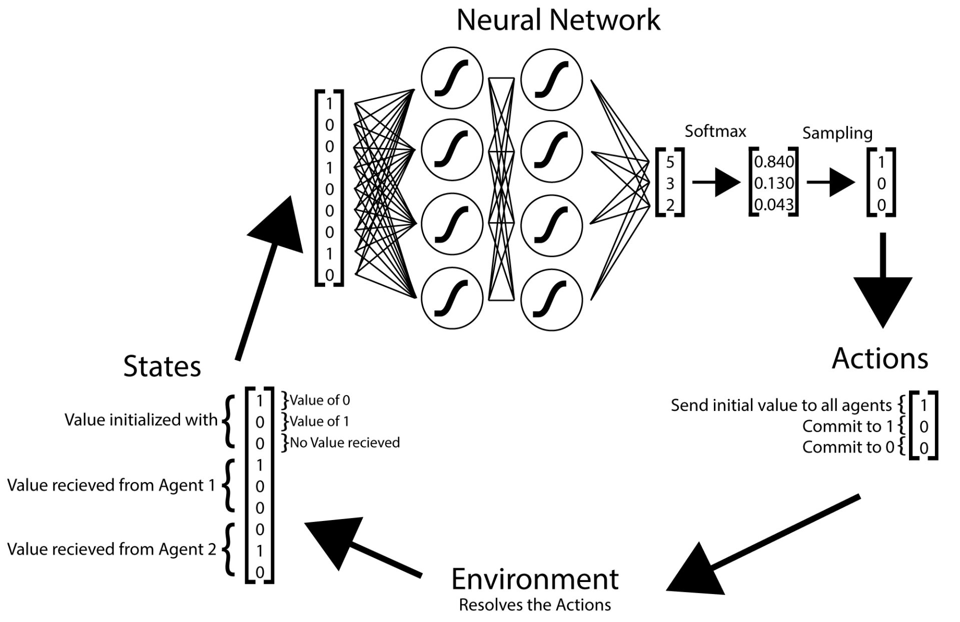

Figure 1: The training process for the perspective of Agent 0 during an example round of the consensus protocol. This is not the first round because if it were then under "States" there would be values of 1 for the positions corresponding to having received no values from Agents 1 and 2. In this particular example, where all of the other agents are honest and can be trusted, the agent should commit to a value of 0 because it is the majority value.

Reinforcement Learning Algorithm We implemented REINFORCE which is an on-policy model-free algorithm that is the precursor to the state of the art Proximal Policy Optimization algorithm that we plan to implement going forwards [9]. REINFORCE is the same as PPO except for clever tricks to reduce variance during training and take larger update steps, allowing for faster, more efficient learning [10].

Neural Networks Neural networks are used to learn the policies of the honest and Byzantine agents. There is one network used for all of the honest parties and one for all of the Byzantines. These networks are feed forward and fully connected, they use ReLU activation functions. We are able to easily change the number of layers and hidden units per layer but have been using one layer of 16 units and a second layer of 8 for now. We have been using the Adam optimizer for stochastic gradient descent during training. These networks have been implemented using the deep learning framework PyTorch, code is available upon request.

Below is an algorithm that outlines the training process for the consensus algorithm at a high level. The following notation is used: e training epochs, s simulations per epoch, α the learning rate, R set of rewards, E environment to be used including the number of honest and Byzantine agents, H set of hyper-parameters for the neural networks.

Algorithm 1 Training process for learning consensus

```
Input: $e$, $s$, $\alpha$, $R$, $E$, $H$
Output: Honest and Byzantine policies, $\pi_H$, $\pi_B$
$\pi_H, \pi_B \leftarrow$ InitPolicies($H$)          // initialize the neural network policies
for i in (1,...,e) do
    $\Theta \leftarrow 0$          // initialize a buffer to store simulation trajectories
    for j in (1,...,s) do
        $\Theta[j] \leftarrow$ RunSimulation($\pi_H, \pi_B, E$)          // run a simulation with the current policies
    end
    $\Omega \leftarrow$ ComputeRewards($\Theta, R$)          // computing rewards for each of the simulations
    $\nabla_\theta \pi_H, \nabla_\theta \pi_B \leftarrow$ ComputeGradients($\pi_H, \pi_B, \Omega$)
    $\pi_H = \pi_H - \alpha \nabla_\theta \pi_H$          // update the policies
    $\pi_B = \pi_B - \alpha \nabla_\theta \pi_B$
end
return $\pi_H, \pi_B$
```

4 Results

4.1 Honest Consensus

The most simple environment is where there are three honest agents and no Byzantine agents. In this environment, the action space for agents is to either send the value they were initialized to, commit 0, or commit 1. The optimal protocol for the honest agents to learn is to send the value they are randomly initialized with in the first round and then commit to whatever the majority value of all the agents is (including their own initial value). In our initial implementation of this algorithm, agents committed to the same value every time, which allowed them to be correct 87.5% of the time. This is because if the agents always committed to a value of 0, only 12.5% of the time will they break validity when they are all randomly initialized to a value of 1 ($0.5^3 = 0.125$). In order to get the agents to learn to commit to the majority value and escape this local optima, we added more rewards to the agents: penalizing an agent for committing in the first round (when they have not received any values from other agents); adding a round penalty for the honest agents to learn to not continuously send values and commit as fast as possible; increasing the validity penalty to remove this local optima. The rewards used are detailed in the table below.

Honest Consensus Rewards				
Consistency Violation	Validity Violation	Majority Violation	Correct Commit	Round Penalty
-1	-2	-1	+1	-.03

After making these modifications, honest agents were able to learn how to agree upon the majority value by epoch 500. By this epoch, each iteration took 2 rounds, in which each agent first sent its initialized value to the other agents in the and then committed to the majority value in the second round. See figure 2 for the honest win percentage as the agents are trained.

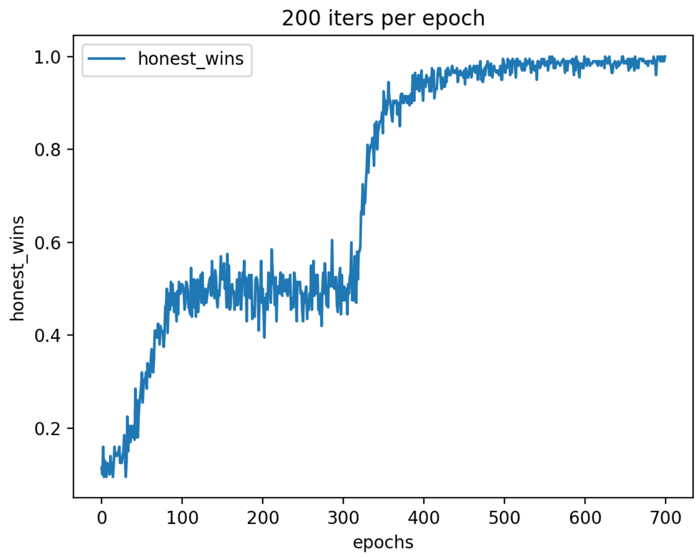

Figure 2: Honest win percentage (where the honest agents satisfy consistency and validity) in the honest consensus setting

4.2 Byzantine Breaking

This environment has 3 agents: 2 honest agents and 1 Byzantine agent. In this simulation, the 2 honest agents were trained to follow the protocol in the honest consensus setting and reliably reach consensus over the majority value they were initialized to. The weights for this network were then frozen such that the honest agents always followed the protocol of sending their initial value in round 1 and committing in round 2, with no changes. A Byzantine agent then replaced one of the honest agents in this environment. The Byzantine agent could see the states of all other agents and could send 0, send 1, or send no value to any subset of agents. The Byzantine agent's goal is to prevent the honest agents from reaching consensus; as such, all of its rewards are positive 1 for having agents violate consistency, validity, the majority value, and the correct commit value.

Byzantine Breaking Rewards [Honest Reward, Byzantine Reward]				
Consistency Violation	Validity Violation	Majority Violation	Correct Commit	Round Penalty
[-1, 1]	[-2, 1]	[-1, 1]	[1, -1]	[-.03, .03]

Within 200 epochs, the Byzantine agent learned to send different values to the agents, breaking their ability to commit to the correct majority initial value. The rewards over time for the Byzantine and honest agents are inverses of each other and are denotes in Figure 3 and Figure 4. When training both honest and Byzantine agents together, the honest agents decide to always commit to the exact same value, which gives them a 50 % win rate in the presence of a Byzantine adversary (this win is only when both of the honest agents are initialized to the same value which occurs with a frequency of: $2 * 0.5^2 = 0.5$).

4.3 Avalanche

The Avalanche family of consensus protocols [11] relies on a meta-stable mechanism, providing a probabilistic safety guarantee in the presence of Byzantine faults. The protocols operate by having nodes repeatedly sample k other nodes in the network to query their lock values. We introduced the action of sampling and querying k nodes to our honest agents' action space, with the hope that the honest agents will rediscover the simplest Avalanche protocol, Slush. In our simulations, the agent can either commit 0, commit 1, or sample from k agents. We ran simulations with five honest agents in the environment with a k value of 3. In the trivial case of randomly initializing 4 agents to v and one agent to v' , agents learn to consistently sample all other agent values for the maximum round length, committing to the majority value (Figure 5). In order to achieve this, we tuned the reward parameters to give greater penalties for consistency, validity, and majority violations.

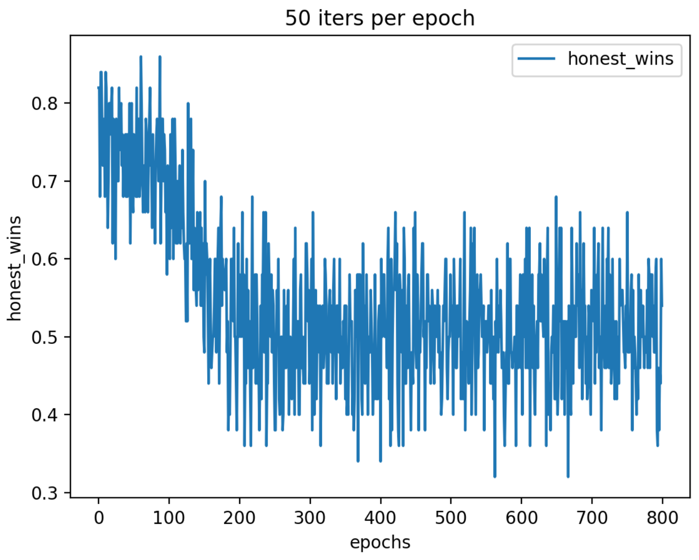

Figure 3: Honest win percentage in the Byzantine Breaking setting.

Avalanche Rewards				
Consistency Violation	Validity Violation	Majority Violation	Correct Commit	Round Penalty
-4	-4	-10	1	-.03

However, in the case where we have 3 agents that are randomly initialized to v and 2 agents randomly initialized to v', the agents get stuck in a local minimum and commit to either 0 or 1 in every run after sampling for the max round length (Figure 6)

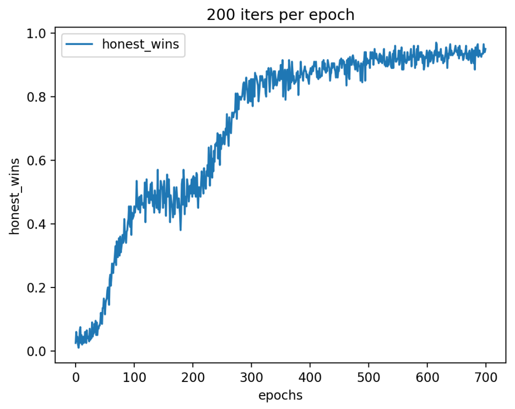

Figure 4: Honest win percentage in the Slush environment, four agents are initialized to the same value and one agent differs

4.4 Public Key Infrastructure (PKI)

Lastly, we integrated a PKI system where each agent sends not only their own value to every other agent but also the values of every other agent that they received values from in the previous round. Because the agents are required to do this (if they did not then it would be clear that a particular agent was Byzantine and should be ignored) the honest parties have the chance to infer which agent is Byzantine by the fact that it is sending different values to different agents.

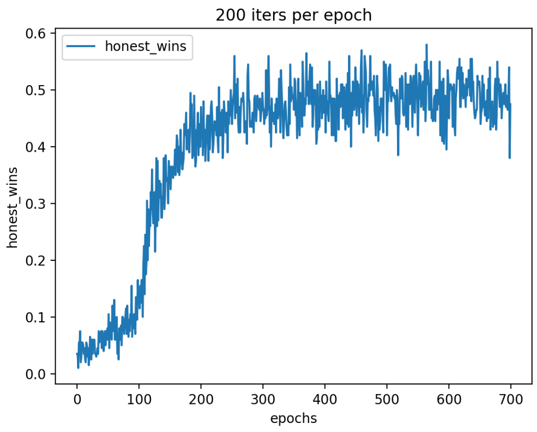

Figure 5: Honest win percentage in the Slush implementation with three agents initialized to v and two agents initialized to v

In a setting with one Byzantine and three honest agents, learning which agent is Byzantine should enable them to ignore the Byzantine agents value and commit to the majority value of the other honest agents.

This inference is significantly more complex that those tested previously and is currently being debugged. We suspect that implementing a more powerful learning algorithm such as PPO [10] in addition to continuing to try many possible hyperparameters, will allow for discovery of the correct protocol.

5 Conclusion

In this paper we demonstrate that agents can learn to replicate simple consensus protocols. We show that honest agents can learn the most efficient way to agree upon some value by tuning their rewards to match the requirements of validity and consistency. We also show that a Byzantine adversary can learn to break the protocol in cases where they are not all initialized to the same value. In addition, we trained agents to replicate the Slush protocol by sampling a set number of agents' value. There remains much work and opportunities left to pursue in the future, investigating different protocols, expanding the action space of agents, and even seeing if we agents can discover new protocols.

Acknowledgements

We would like to thank our professor, Dr. Kartik Nayak for his guidance and mentorship on this project, as well as the open source software contributors, including but not limited to: Numpy, Pandas, Scipy, Matplotlib, Pytorch, and Anaconda/Jupyter notebooks [12].

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[3] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis.

[5] OpenAI. Openai five. `https://blog.openai.com/openai-five/`.

[6] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

[7] Marc Harper, Vincent A. Knight, Martin Jones, Georgios Koutsovoulos, Nikoleta E. Glynatsi, and Owen Campbell. Reinforcement learning produces dominant strategies for the iterated prisoner's dilemma. *CoRR*, abs/1707.06307, 2017.

[8] Martín Abadi and David G. Andersen. Learning to protect communications with adversarial neural cryptography. *arXiv*, 2016.

[9] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[11] Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. `https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV`.

[12] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, page arXiv:1907.10121, Jul 2019.