Communicating Agents Seeking Information

David James

2019-01-28

Table of Contents

Personal Motivations

I am fascinated by:

- simulation as a tool
- building agents that learn
- understanding multi-agent systems
- understanding how communication affects ↑

My educational background:

- Liberal Arts + ECE (undergraduate)
- Public Policy + CS (graduate)

Research Question

How do decentralized* communicating agents behave?

- What behavior patterns emerge?
- How do incentives affect behavior?
- Do agents reach global optima?
- What are the algorithmic challenges?

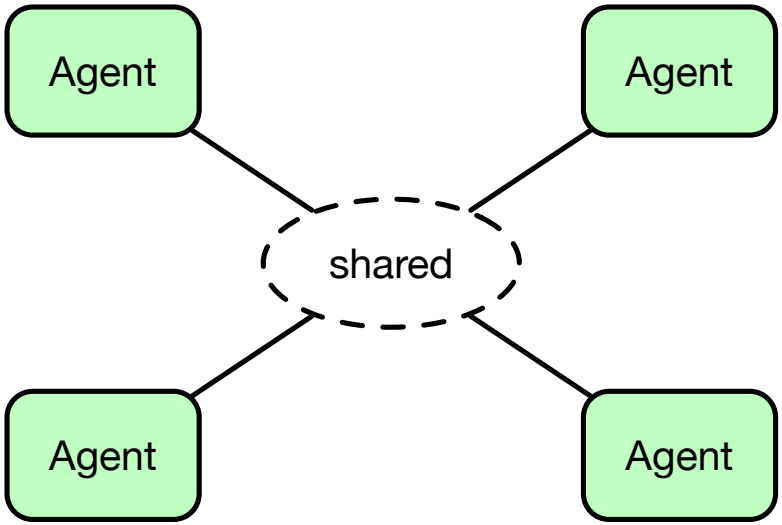
Design Process

What kind of simulation can explore these questions?

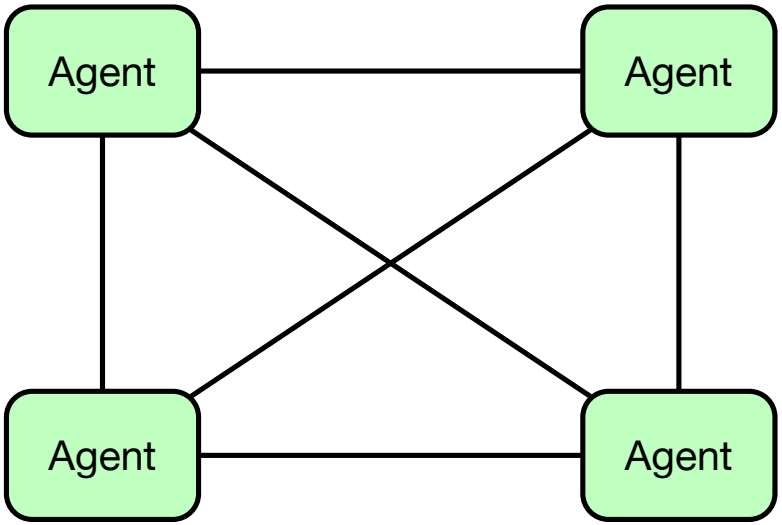
Let's walk through some design questions:

- How does communication work?
- How does learning (optimization) work?

Communication Type

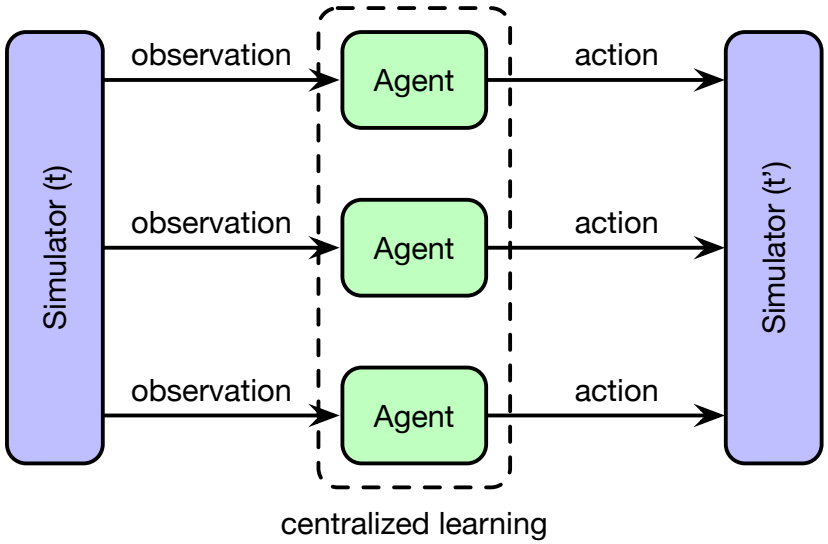


Shared Communication

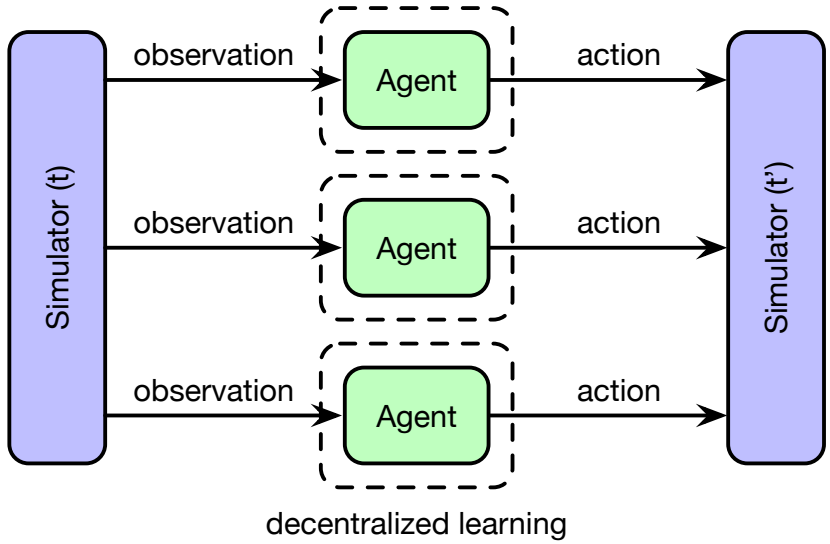


Private Communication

Learning Type



Centralized Learning



Decentralized Learning

Table of Contents

Simulation Overview

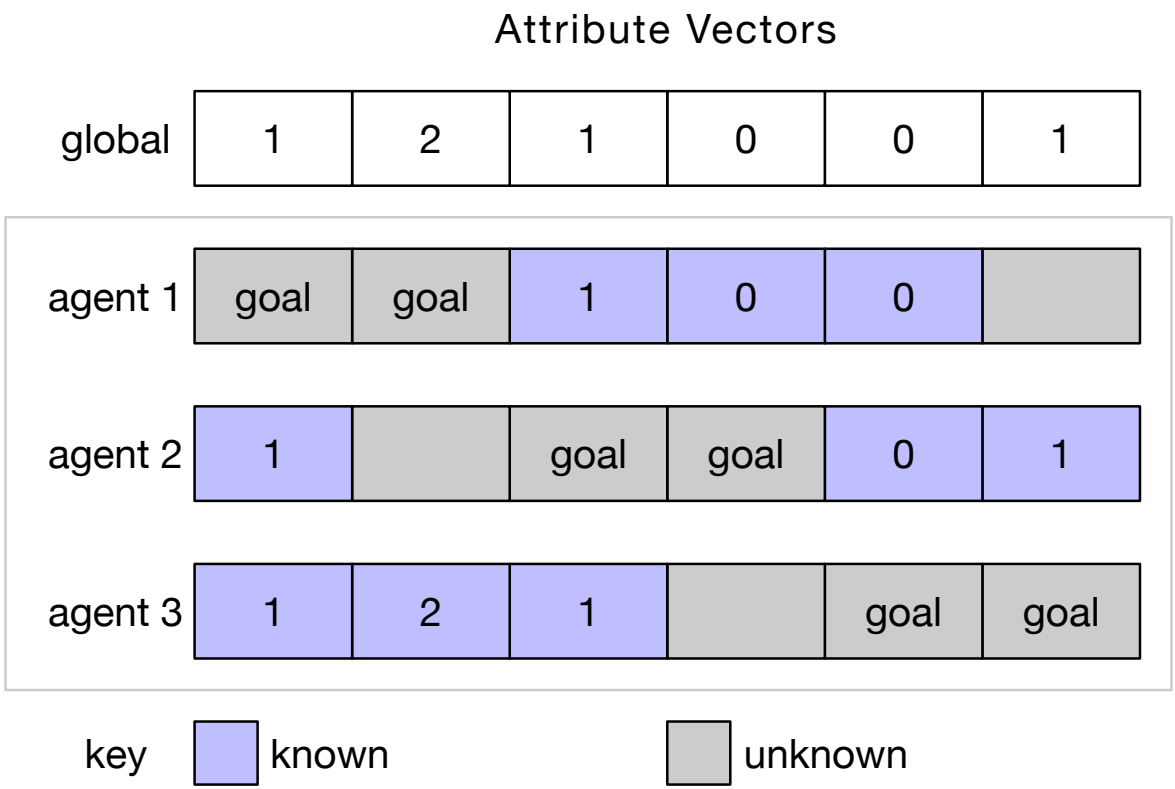
You can think of the CASI simulation as a game:

- There are n_a agents.
- Each episode consists of T steps.
- If an agent correctly guesses an unsolved goal attribute, it gains points.
- If an agent guesses all of its n_g goal attributes, it gains many points and wins.
- Various actions are penalized (more on this later).
- For example, asking a question has a small cost.

Simulation Attributes

Each simulation has a configurable number of attributes.

Here is an example with three agents and six attributes:



Agent's Perspective

Each agent sees an observation and must choose an action.

An observation consists of:

- the agent's internal state
- interactions with other agents
  - (answers received, questions asked)

An action consists of:

- intentions on how to respond to questions
- choosing what to do next, one of:
  - (do nothing, guess an attribute, ask a question)

Communication Protocol

- agent i says to agent j:
  "tell me about attribute a"
- agent j chooses an intention
  (i.e. to lie, ignore, or tell the truth)
- the simulation converts the intention to an answer
  and delivers it to agent i

Agent Intentions

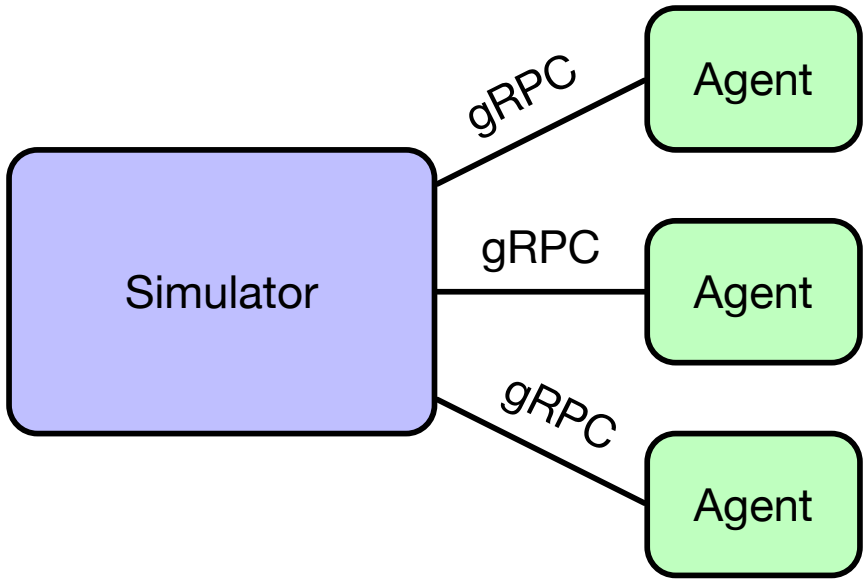
Intention	Response	
	if *unknown* i.e. *None*	if *known* i.e. *Some*(v)
specific lie	*Value*(r)	*Value*(r) $\mid r \neq v$
general lie	*Known*	*Unknown*
ignore	*None*	*None*
incomplete truth	*Unknown*	*Known*
complete truth	*Unknown*	*Value*(v)

where r is a random value

Incentives

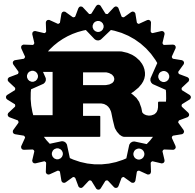
```
let incentives = Incentives { // Experiment #1
    cost_ask_question: 2.0,
    cost_ask_self_question: 20.0,
    cost_excessive_guessing: 20.0,
    cost_incorrect_guess: 20.0,
    cost_known_guess: 20.0,
    cost_non_goal_guess: 20.0,
    cost_solved_guess: 20.0,
    cost_unnecessary_reply: 20.0,
    max_guesses_per_attr: 1,
    reward_correct: 100.0,
    reward_win: 200.0,
};
```

CASI Architecture



- Simulator (written in Rust)
- Learning Agents (written with PyTorch)
- Interprocess communication (uses gRPC)

Why Rust?



Rust helps me write (and refactor) fast and correct software, quickly.

- *Performance*: no runtime, no garbage collector
- *Reliability*: expressive type system gives memory-safety, thread-safety
- *Productivity*: package manager, libraries, community

Why PyTorch?

As you might expect, PyTorch:

- manages a computation graph and auto-differentiation
- includes reusable modules for neural networks
- has CPU and GPU support

In comparison to TensorFlow, I find PyTorch is:

- more natural (has a more intuitive API)
- easier to debug

Why gRPC?

Simple service definition:

- easy refactoring

Works across languages and platforms:

- conveniently bridges Rust and Python
- public and private use cases (RPC, APIs)

Start quickly and scale:

- works on one machine
- viable for large multi-agent systems

Neural Network gRPC Definition

Here is how I use gRPC to define a neural network model for a learning agent:

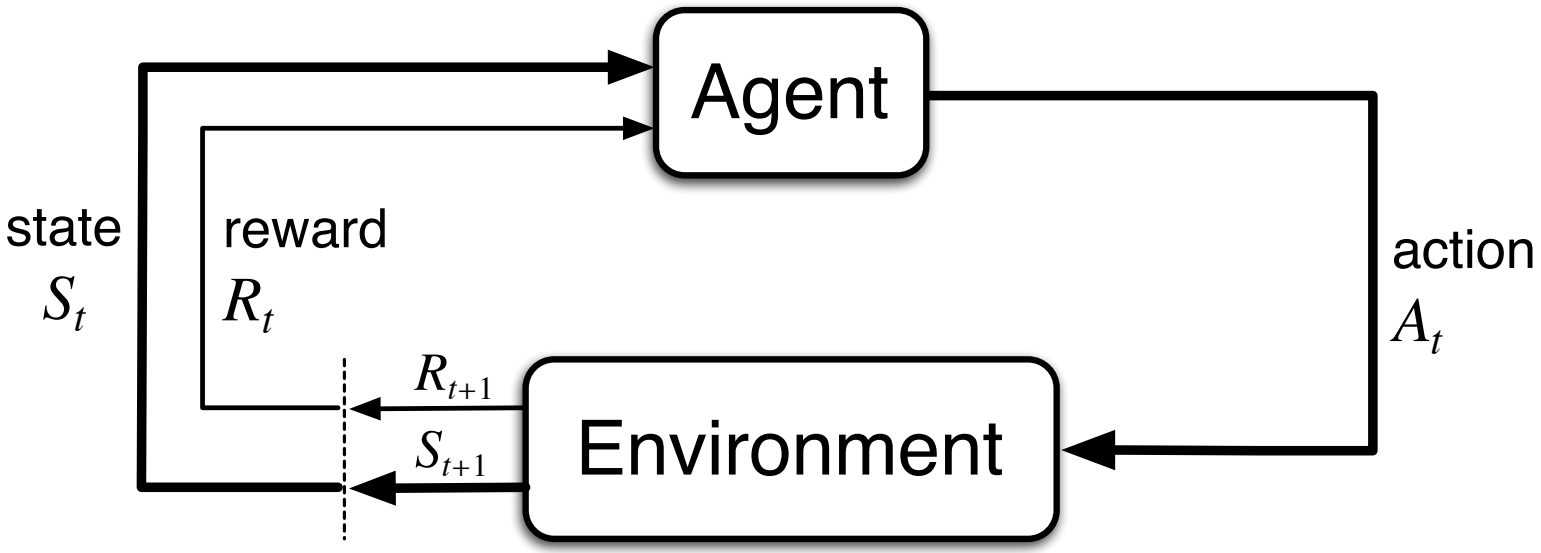
```
// Neural network for action prefs and value function
service Net {
    rpc Init (NetCfg) returns (Empty) {}
    rpc InitHidden (Empty) returns (Empty) {}
    rpc Predict (Observations) returns (Predictions) {}
    rpc ResetOptimizer (OptimCfg) returns (Empty) {}
    rpc Train (Sequences) returns (Loss) {}
    rpc GetParams (Empty) returns (Params) {}
    rpc SetParams (Params) returns (Empty) {}
}
```

This file will generate executable code in a wide variety of languages.

Table of Contents

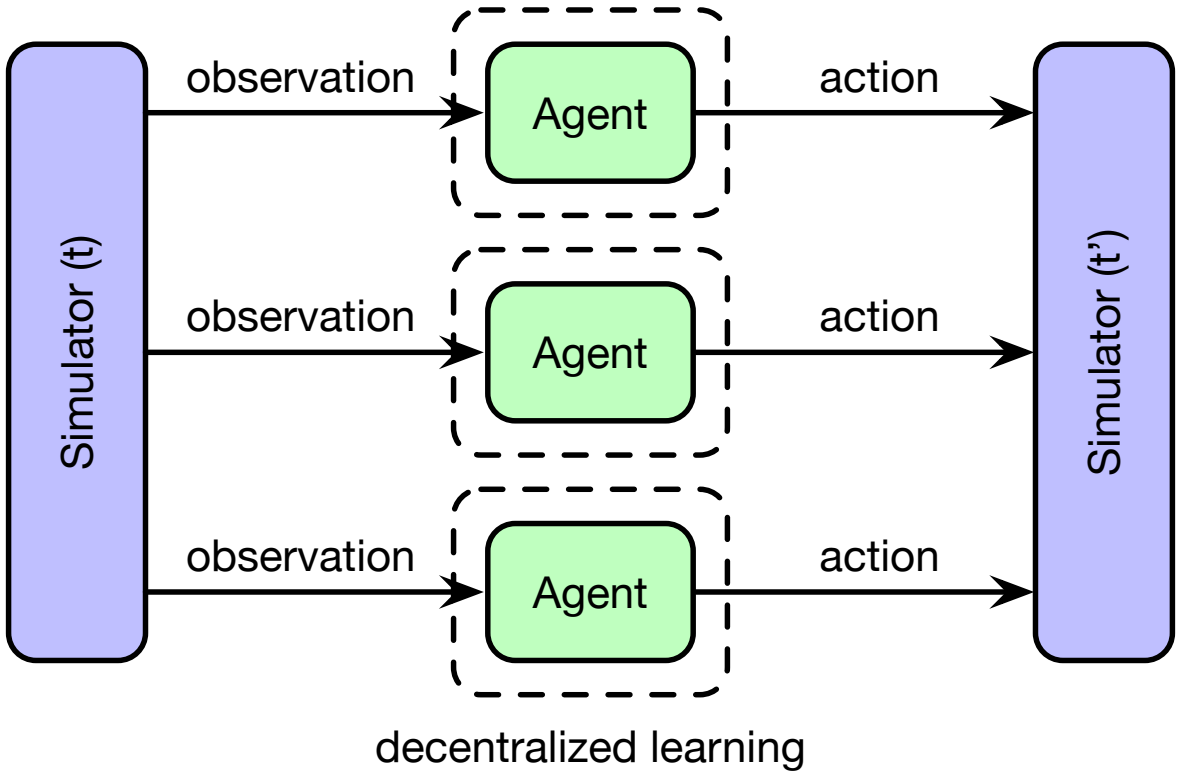
Reinforcement Learning

Reinforcement learning (RL) is learning what to do — how to map situations to actions — so as to maximize a numerical reward signal. - Sutton & Barto



Decentralized RL

You may remember this diagram from before:



Can you think about some pros and cons?

RL Dimensions

Environmental considerations:

- states: fully-observed vs. partially-observed
- actions: discrete vs. continuous

Algorithmic considerations:

- policies: deterministic vs. stochastic
- on-policy vs. off-policy
- model-based vs. model-free *

'Models' in RL

I briefly wanted to mention what 'model' means in the context of RL, since it can be confusing.

Ben Recht says it well in "A Tour of Reinforcement Learning: The View from Continuous Control" (2018):

> The term "model-free" almost always means "no model of the state transition function" when casually claimed in reinforcement learning research. However, this does not mean that modeling is not heavily built into the assumptions of model-free RL algorithms.

Algorithm Experimentation

I've implemented these algorithms from scratch in Rust and PyTorch:

- REINFORCE ('vanilla')
- REINFORCE with baseline
- Proximal Policy Optimization (PPO)
- Advantage Actor Critic (A2C)

I'm currently using A2C + PPO with exploration bonuses.

REINFORCE

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

"Reinforcement Learning: An Introduction" by Sutton & Barto, 2018

PPO Actor-Critic

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

"Proximal Policy Optimization Algorithms" by Schulman et al. 2017

Exploration

There are many kinds of exploration in RL:

- ϵ-greedy exploration *
- a result of stochastic policies
- entropy-based exploration
- seeking novel (state, action) pairs

Count-Based Exploration with Hashing

From *#Exploration* by *Tang et. al, 2017*:

> a simple generalization of the classic count-based approach [. . .]. States are mapped to hash codes, which allows to count their occurrences with a hash table. These counts are then used to compute a reward bonus according to the classic count-based exploration theory.

Locality Sensitive Hashing (LSH)

From Wikipedia:

> LSH hashes input items so that similar items map to the same 'buckets' with high probability [. . .]. LSH differs from conventional and cryptographic hash functions because it aims to maximize the probability of a 'collision' for similar items.

Arguably, the key criteria for choosing an LSH algorithm is the relevant *distance metric*.

LSH for Hamming Distance

Early LSH papers on Hamming distances suggest using single-column sampling techniques.

- Each output bit is created simply by sampling a single column from the input vector.
- However, sampling only a fraction of the input vector seems suboptimal. I would expect benefits from drawing information from all columns.

Hamming Distance LSH with Super-Bits

I blended two approaches from the literature as follows:

- Sample from each column but only once. Otherwise, differences can be accentuated.
- Designate groups of the output bits. This allows trading-off between the number of 'anchor' points for the distance calculation vs. the amount of compression. (Inspired by Super-Bit LSH by Jie et al., 2012.)
- Combine columns using the Hamming distance. Store the result in each super-bit group.

Table of Contents

Attributes

Here is an example of a Rust data structure to configure attributes for an experiment.

```rust
/// Attribute configuration
pub struct AttrCfg {
    /// Total number of attributes in simulation
    pub count: u16,

    /// Attributes known from direct experience (per agent)
    pub direct: u16,

    /// To win, an agent must solve this many attributes
    pub goal: u16,

    /// Number of possible values for each attribute
    pub values: u8,
}
```

Experiment 0 Description

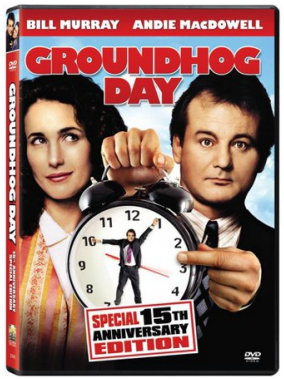
Two Agents:

- One non-learning random agent
- One learning agent
  - a feedforward neural network

Question: Can this agent learn to win?

Answer: Perhaps surprisingly, it can, by exploiting the lack of random initialization.

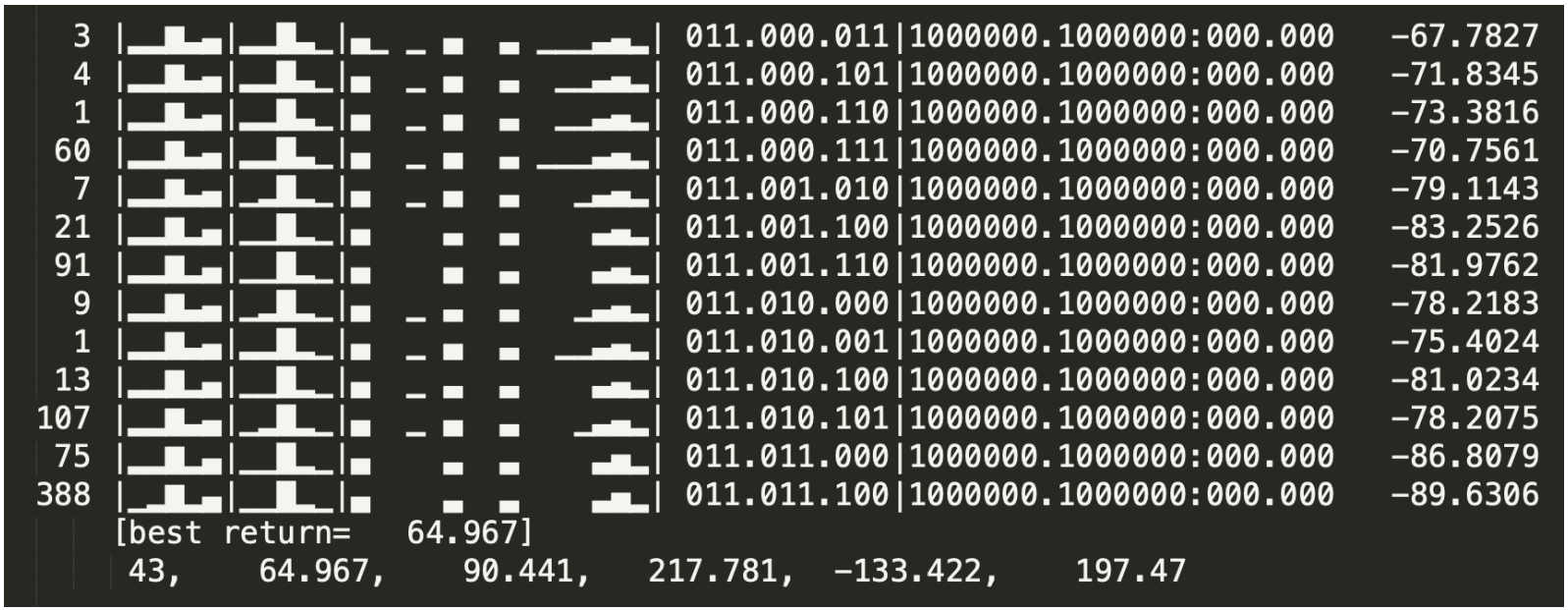
You might call it 'Groundhog Day' learning.



Experiment 0 Output 1

```
Running experiment e1
iteration, ret_mean,  ret_std,  ret_max,  ret_min, loss_mean
  201 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.000.000|1000000.1000000:000.000    -0.4073
   14 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.000.001|1000000.1000000:000.000    -0.4081
   70 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.000.010|1000000.1000000:000.000    -0.4056
   43 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.000.011|1000000.1000000:000.000    -0.4045
   31 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.000.100|1000000.1000000:000.000    -0.4105
   15 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.000.101|1000000.1000000:000.000    -0.4115
   16 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.000.110|1000000.1000000:000.000    -0.4048
   45 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.000.111|1000000.1000000:000.000    -0.4050
   71 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.001.000|1000000.1000000:000.000    -0.4068
   10 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.001.010|1000000.1000000:000.000    -0.3940
    9 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.001.110|1000000.1000000:000.000    -0.3880
   77 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.010.000|1000000.1000000:000.000    -0.4115
   26 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.010.001|1000000.1000000:000.000    -0.4117
   14 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.010.101|1000000.1000000:000.000    -0.4149
  121 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.011.000|1000000.1000000:000.000    -0.3929
   17 |▂▂▂▂|▂▂▂▂|▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁| 011.011.100|1000000.1000000:000.000    -0.3963
      [best return= -337.672]
        0,  -337.672,   126.714,     3.619,  -474.629,   1742.59
```

Experiment 0 Output 2



Experiment 0 Output 3

```
 30 |  ▉  |  ▉  |     ▉          | 011.000.111|1000000.1000000:000.000   162.8091
 30 |  ▉  |  ▉  |        ▉       | 011.010.101|1000000.1000000:000.000    98.6273
720 |  ▉  |  ▉  |▉               | 011.011.100|1000000.1000000:000.000    15.7926
[perfect return=  370.000]
[three perfect returns]
[stop early]  best_return:  370.000 at iteration 120 (after 123 iterations)
```

Experiment 1 Description

Same as Experiment 0 but with random experiment initialization:

Two Agents:

- One non-learning random agent
- One learning agent
  - a feedforward neural network

Question: How can this agent learn to win?

Answer: It can't; communication would be required.

Experiment 2 Description

Two Agents:

- One non-learning completely honest agent
- One learning agent, with both:
  - a feedforward neural network
  - an LSTM

Question: How can this agent learn to win?

Answer: By asking questions of the honest agent.

Discuss: Is the LSTM necessary in this case?

Table of Contents

Problem: Diverse Purposes for Runs

Since the simulation engine is a platform for experimentation, it has different purposes:

- Verifying new algorithms
- Inspecting policy changes over time
- Exploring hyperparameters
- Monitoring optimizer performance

Getting the verbosity level is tricky: **too little** misses out, but **too much** creates clutter...

Solution: Granular Logging

One solution is to use fine-grained logging:

```
let log_cfg = LogCfg { // subset
    log_action_probs: false,
    log_actions_rewards: false,
    log_attrs: false,
    log_encodings: false,
    log_experiment_start: true,
    log_interactions: false,
    log_iteration_csv: true,
    log_learning: false,
    log_obs_summary: true,
    log_trajectories: false,
};
```

Problem: When to Stop Optimization?

Policy gradient (PG) methods aim to improve a policy over many iterations.

However, the policy performance does not necessarily monotonically improve.

Empirically, PG methods, even with improvements such as PPO, can sometimes drift away from good performance and never recover.

See: "Where Did My Optimum Go?: An Empirical Analysis of Gradient Descent Optimization in Policy Gradient Methods" by Henderson et al. 2018.

Why do Policy Gradient Methods Degrade?

The Policy Gradient Theorem gives the correct gradient, so what could go wrong in practice?

- Gradient ascent takes many iterations. How many? What does research say about the bounds as applied to PG?
- The underlying function approximator* may lack sufficient representational power.
- The network parameters may not be able to efficiently respond to the latest distributional shift.

Solution: Backtracking Algorithm

I implemented a three-level backtracking algorithm that requires three hyper-parameters:

```rust
/// Backtracking configuration
pub struct BacktrackCfg {
    /// Snapshot pops before stopping early
    pub pop_limit: u32,

    /// Consecutive resets before snapshot pop
    pub restart_limit: u32,

    /// Consecutive unimproved iterations before reset
    pub unimproved_limit: u32,
}
```

The algorithm is shown on the next slide.

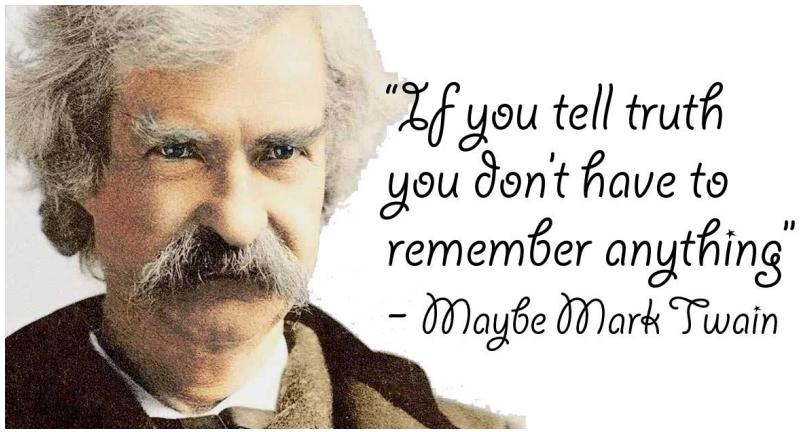
Backtracking Algorithm

Algorithm 1: Three-level backtracking algorithm

```
snapshots ← []; pops ← 0; restarts ← 0; unimproved ← 0
for iteration ← 0 to max_iterations do
    if improved_performance then
        snapshots.push(model); restarts ← 0; unimproved ← 0
    else unimproved ← unimproved + 1
    if unimproved > max_unimproved then
        restarts ← restarts + 1; unimproved ← 0
        if restarts > max_restarts then
            pops ← pops + 1
            if pops > max_pops then stop_early()
            else snapshots.pop(); restarts ← 0
            model ← snapshots.last()
    else
        model.optimization_step()
```

Problem: Consistency of Responses

An agent reports an intention to the simulation; the simulation converts it to a response that another agent observes.



source: http://factmyth.com

Claim: inconsistent lies are easier to discover.

How can the simulation achieve consistent lying?

Consistent Lying

```rust
/// Return a deterministic lie that depends only on:
/// 1. the function arguments
/// 2. the agent's private salt
/// Note: 'q' / 'a' = agent id of questioner / answerer
fn specific_lie(
    &self, q: AgentId, attr_idx: u16, attr_val: Option<u8>
) -> Answer {
    let m = self.attr_cfg.values as u64;
    let hash = self.calc_hash(q, attr_idx);
    let lie = match attr_val {
        Some(true_val) =>
          (true_val as u64 + (hash % (m - 1))) % m,
        None => hash % m
    };
    Answer::Value(lie as u8)
}
```

Table of Contents

Next Steps

- LSTM architecture & parameter tuning
- Create experiments with multiple learning agents
- Improve visualizations
- Try various multi-agent reward structures
- Use unsupervised learning to find patterns in policies
- Improve efficiency by using parallelism in Rust

Discussion

I look forward to your questions and comments.