Navigator's Reward Function

Ensure that all future implementations of reward functions are version controlled and tracked through whatever machine learning experiment tracking approach that we used

Reinforcement learning (RL) consists of the iterative evaluation and improvement of a policy. A policy model approximates the mapping of each state in the state space to its contextual value. We use an actor critic policy gradient model. The actor samples an action based off of observations of previous states. The action is invoked against an environment and the return of that interaction is used to compute a greedy reward. Subsequently, the critic estimates the value (cumulative discounted reward) of the action. After however many episodes configured for a given epoch, buffers of accumulated and estimated reward are used to update the actor and critic models. The policy is coerced to make better choices of action and the critic to make better estimates of the value function. The inspiration of the adversarial-flavored architecture is the dopamine circuit of the brain which operates by constantly making contextual predictions of how rewarding something is in the environment. The motivation for learning is then minimizing this prediction error.

Exploration Based Reward

The baseline reward of **NAV** functions as a bootstrapping mechanism and as a hedge. In classic Rumsfeldian fashion, we want to create an "automatic" mechanism [1] for the "unknown unknowns" of the problem we are trying to solve. This is critical as we can run with this reward alone as a pre-training strategy, learning the core system dynamics for "free", maximizing the value of each label in the fine-tuning process. One could think of the work we do in pre-training as freeing up the "budget" of the finite amount of time we have to incorporate expert judgement into the model.

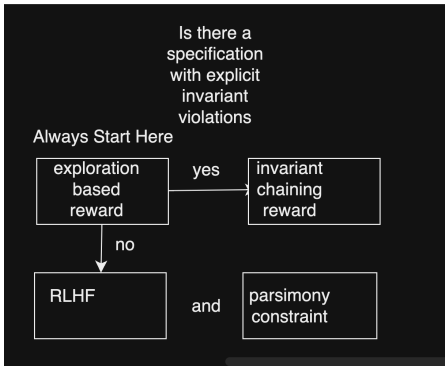

Figure 1: Caption for the figure.

This reward is taken directly from the paper "Curiosity Exploration for Self-Supervised Prediction" which introduces a dense reward for tasks that are difficult to learn due to their sparse reward. Intuitively, curiosity means that the agent will be motivated in some way by its own uncertainity. The rub that makes this reward construction so novel is that it trains the agent to be curious about states if and only if they are learnable (alterable via the invocation of an

[1] invocations of an executable are cheap and with an exploration motivating reward the agent can learn automatically without ground truth and without human intervention

action by an agent). This works quite well with our framework as we take a strong stance of partial observability - instead of simply exploring any new state our agent will be motivated to use inputs that change the states of watchpoints (directly or indirectly).

To review, this reward works by first executing the first steps of the policy gradient algorithm

1. the policy samples an action conditioned on its observations of previous states
2. the action is executed in the environment. In our case, the input is passed to - for example - the Linux kernel and the resultant trace from Cannoli is propagated.
3. this trace is s_{t+1} and represents (like s_t the time step before) the program partially observed through its watch points

From this sequence of interactions we have the tuple (s_t, s_{t+1}, a) but haven't yet computed the reward. We do this by first using s_t and s_{t+1} to predict $\hat{a_t}$ minimizing the distance between ground truth a_t and $\hat{a_t}$ each training step [2].This represents how well the model can predicts its own dynamics [3] and also gives us - via the middle layers of the network- an embedding layer ϕ [4]. The embedding layer ϕ allows us to deal with high dimensional and continuous state spaces and transforms the current and previous state to $\phi(s)$ and $\phi(s_t)$. The forward dynamics model uses $\phi(s), a$ to predict $\phi(s_{t+1})$. When the loss is high, the system is uncertain about the next state, i.e. it is not easily predicted using knowledge of prior states explored by the policy.

Curiosity Based Reward

$$1 - L(\phi(\hat{s_{t+1}}), \phi(s_{t+1}))$$

Invariant-Chaining Reward

For DemoOne we used a reward strategy intended to encourage the discovery and compositional reuse of gadgets. It assumes sub-sequences that precede states which trigger invariant violations represent gadgets. This is an un-opinionated way of tokenizing the sequence and deserves revision once we come back to a problem where we use a specification and discover explicit invariant violations. Ideally, there would be a deterministic algorithm that would recursively subtract and recombine components of the sequence to discover the

[2] the inverse and forward dynamics model of the curiosity based reward are trained entirely online or can be trained in batch at the end of an epoch - either way, no additional effort is required

[3] it is actually a calibration mechanism and a modeling of the model's uncertainty which might be useful later for interpretibility reasons

[4] every layer of a neural network is a non-linear function -typically some linear function transformed by a non-linear activation function. This embedding transforms the concatenated states to a lower dimensional representation

minimal sub-sequence that results in the invariant violation. We then would precede as we have already, by caching and rewarding the reuse of a gadget highly and giving a medium reward for discovery of novel gadgets. These terms would represent a coefficient that multiples the curiosity based reward. Similar to my thoughts in the next section, longer is always better given that the sequence contains gadgets.

Specification Based Reward

$$\lambda * (1 - L(\phi(\hat{s_{t+1}}), \phi(s_{t+1})))$$

Below, the code taken from **CannoliEnv** describes how we derive the coefficient λ

```
if invariant:

    if ''.join(map(str, self.action_sequence)) in self.memos:
        incremental_reward = 5
        if self.invariants_previously_seen_in_episode:
            incremental_reward *= 15 *
                self.invariants_previously_seen_in_episode
    else:
        incremental_reward = current_step * 10
        if self.invariants_previously_seen_in_episode:
            incremental_reward *= 5 *
                self.invariants_previously_seen_in_episode
    self.invariants_previously_seen_in_episode += 1
```

Listing 1: Invariant Reward

Information Gain Regularization

There are a number of methods for that use information gain to regularize the policy [5]. One method we won't be covering is explicitly randomizing (increasing entropy) of the weights of the policy. What we will cover here is the strategy of rewarding information gain as we step through an environment. The simplest method for doing this is to take a streaming, exponential moving average and (like in the invariant reward) explicitly rewarding long episodes. While we don't have to do early stopping, it might make sense in the future as a configuration option [6]. Luckily, we have an easy information gain metric in the curiosity reward so we can do something like this:

[5] regularize refers to the overall strategy of not just minimizing or maximizing a loss function but maximizing *generalizability* of a function approximation. A good example of regularization terms in a loss function are explicitly punishing model complexity in linear regression. We use it similarly here - we want to derive both an accurate model (in the eliminating false positives and false negatives sense) and a parsimonious model (which is far more likely to generalize)

[6] the trade-off is that adversaries could imagine vulnerabilities that would be missed by us adding this regularization constraint. This is something that deserves more attention and experimentation

$\mathrm{x}_t = 1 - L(\phi(\hat{s_{t+1}}), \phi(s_{t+1}))$
$EMA_t = \alpha \cdot x_t + (1 - \alpha) \cdot EMA_{t-1}$
$EMA_t * surprising_n$

Here $surprising_n$ tracks every state where the (smoothed) information gain is non-zero.

Reinforcement Learning with Human Feedback We currently have a dense reward, with or without a specification. One of our innovative claims is the elimination of false positives via learning user preferences. One of the nice properties of collecting these preferences is that it doesn't just make the results more interesting to the user, it also guides search in more productive directions [7].

We will likely use RLHF with at least one other form of inverse reinforcement learning [8]. One obvious source of these trajectories is a script scraping from the **CVE** database and the reproduction of relevant vulnerabilities through interaction with our instrumented executable.

I propose a reward model $R(\theta)$ trained to predict a number between $1...k$ with k = number of traces the user has configured to be returned iteration over iteration. Note that in a sense the reward model learns a surrogate task by learning the expert's *preferences* instead of the criteria they use to form them. Now that we have a function approximation, we can return a reward for every trace [9]. All of this ultimately results in a "virtuous cycle" where the policy gets targeted feedback on how to correct its value estimations based off of the preferences of the user.

To start, instead of predicting rankings we should simply output 0 or 1 using $R(\theta)$ to represent how likely a trace is to be retained and have instructions to the end user not mention ranking but instead instruct them to simply remove any trace that isn't relevant.

RLHF reward

$$\alpha * \sum_{t=1}^{n} EMA_t * surprising_n$$

[7] this sounds a bit tautological but given that human experts know things that can't easily or efficiently be put into a loss function they aren't just giving their business preferences but also critical information lost in the mathematical representation of the reward function

[8] it is called inverse because instead of generating examples in the form of sequences you are learning from sequences of expert actions

[9] this is an episode level reward although we can do standard reinforcement learning with human feedback and $R(\theta)$ can take a subsequence as input instead