Empirical Evaluation of Adversarial Surprise

Samyak Parajuli

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-203
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-203.html

August 17, 2021

Acknowledgement

Thank you to all my friends and family who have supported me along the way.
This thesis discusses material from an academic paper titled "Explore and Control with Adversarial Surprise" prepared in collaboration with authors Arnaud Fickinger, Natasha Jaques, Michael Chang, Nick Rhinehart, Glen Berseth, and Sergey Levine.

Empirical Evaluation of Adversarial Surprise

by

Samyak Parajuli

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Alexandre Bayen, Chair
Professor Sergey Levine

Spring 2021

The thesis of Samyak Parajuli, titled Empirical Evaluation of Adversarial Surprise, is approved:

Chair Alexandre M. Bayen Date 8/16/2021

Sergey Levine Sergey Levine Date 8/16/2021

Date

University of California, Berkeley

Empirical Evaluation of Adversarial Surprise

Abstract

Empirical Evaluation of Adversarial Surprise

by

Samyak Parajuli

Masters of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Alexandre Bayen, Chair

In this report, we describe experiments supporting a new unsupervised reinforcement learning method, Adversarial Surprise, which has two policies with opposite objectives take turns controlling a single agent [12]. The Explore policy maximizes entropy, putting the agent into surprising or unfamiliar situations. Then, the Control policy takes over and seeks to recover from those situations by minimizing entropy. Through multi-agent competition, this adversarial game between the two policies allows for the agent to both find increasingly surprising parts of the environment as well as learn to gain mastery over them. We show empirically that our method leads to more effective exploration of stochastic, partially-observed environments, is able to perform meaningful control to minimize surprise in these environments, and allows for the emergence of complex skills within these environments. We show that Adversarial Surprise is able to outperform existing intrinsic motivation methods based on active inference (SMiRL), novelty-seeking (Random Network Distillation (RND)), and multi-agent unsupervised RL (Asymmetric Self-Play (ASP)) in MiniGrid, Atari and VizDoom environments.

Contents

List of Figures

Acknowledgments

Thank you to all my friends and family who have supported me along the way.
This thesis discusses material from an academic paper titled ”Explore and Control with Adversarial Surprise” prepared in collaboration with authors Arnaud Fickinger, Natasha Jaques, Michael Chang, Nick Rhinehart, Glen Berseth, and Sergey Levine.

Chapter 1

Introduction

As described in [34], reinforcement learning is learning what actions to perform in order to maximize a numerical reward signal. Generally, this signal is extrinsic, meaning it comes from an external source (reaching a goal state in the environment, killing an enemy, etc.) Agents that are externally motivated have seen widely successful results in an assortment of various tasks such as [25], [5], and [22]. However, designing rewards of this nature can be difficult and inaccurate. For example, providing a dense reward requires not only specifying what to do to but also what not to do - known as the "side effects problem" [1]. A sparser reward is easier to specify, but can cause learning to be slow and difficult [36], [26]. As a result, a class of unsupervised RL algorithms has been developed that do not depend explicitly depend on task reward and instead focus on intrinsic motivation [8]. The purpose of this intrinsic motivation is to develop a sense of "broad competence" through exploration and general novelty seeking behavior as opposed to a more specific external goal [9].

This novelty seeking behavior can be helpful in exploring the underlying state space at a faster rate, but it can also be prone to the "noisy TV problem" coined in [7]. The essence of this problem is that if an agent is rewarded for seeking novel states, it could get stuck at highly entropic elements within the environment (e.g. a "noisy" TV that produces unpredictable noise and static.) The agent would continuously be rewarded for watching this TV, but would not learn meaningful behavior or make any progress within the environment.

Conversely, there exists work based on the "free-energy principle", which proposes that all biological systems are driven to minimize "free energy" - calculated to be the difference between an organism's predictions about its sensory inputs and the sensations it actually encounters [16]. This reasoning guides works such as [6] and [17], leading to their approaches of "surprise minimization" and "active inference", respectively. Although this strategy can encourage the development of complex behavior, it can also suffer from the "dark room problem". By adopting the objective of minimizing surprise, an agent can enter an area that maintains the least amount of sensory information (a "dark room") and never leave.

Building upon both of these ideas, we present an empirical analysis on the recently proposed Adversarial Surprise algorithm - an unsupervised multi-agent reinforcement learning algorithm that balances exploration and control as an adversarial game between two policies. [12]. The Explore policy is novelty-seeking, and attempts to maximize surprise over the course of the episode, putting the agent into a diverse range of novel states. In turn, the Control policy must minimize surprise by learning to manipulate its environment in order to return to safe and predictable states.

This report is structured with the second chapter presenting background on relevant works and topics, the third chapter presenting the novel Adversarial Surprise algorithm and detailing implementation along with experiment analysis, and finally the fourth chapter summarizing the results and postulating future directions for this work.

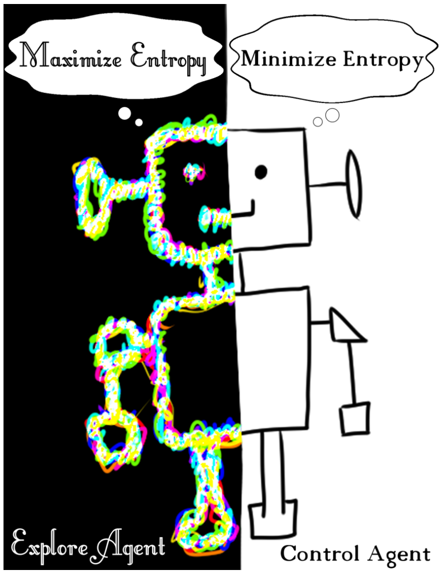

Figure 1.1: Personification of Adversarial Surprise Algorithm

Chapter 2

Background

2.1 Partially Observable Markov Decision Process

Standard Markov decision processes assume that the complete state of the world is visible to the agent. This assumption becomes unrealistic when considering real-world agents. They become limited by the particular sensors they use; for instance, a robot with only a front-facing camera would not have the full information of the environment around it. In these types of scenarios, it makes sense to map observations of states into actions in a framework known as a partially observable Markov decision process (POMDP) [20]. Formally, A POMDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{O}, r, \gamma)$, where $s \in \mathcal{S}$ are states, $a \in \mathcal{A}$ are actions, $r(a, s)$ is the reward function, and $\gamma \in [0, 1)$ is a discount factor. The environment is only partially observable, so the agent cannot observe the true state s, but rather observes $o \sim p(O|s)$. At each timestep t, the agent selects an action a_t according to its policy $\pi(a_t|o_t)$, receives reward $r(a_t, s_t)$, and the environment transitions to the next state according to $\mathcal{T}(s_{t+1}|s_t, a_t)$. We are interested in stochastic environments, in which the emission distribution distribution is inherently entropic for some states, i.e. $\exists s : H(p(O|s)) > 0$.

2.2 Multi-Agent Reinforcement Learning

Although we only employ one embodied agent, we have two policies that are optimized sequentially. This is different from a typical multi-agent learning problem that has multiple agents interacting simultaneously with their environment and potentially with each other, but can utilize similar techniques to learn. One approach is using fully decentralized learners [35] that directly apply single-agent algorithms to the multi-agent domain, where each agent independently learns its own policy, treating other agents as part of the environment. These methods are easier to implement, but can suffer from unstable convergence and may create a non-stationary problem since the future transitions no longer depend on the current state, violating the Markov property. [23]. In this work, we use an independent learning approach with further details explained in section 3.2.

On the other side, there are fully centralized policies that are modeled as standard Markov decision processes consisting of a set of global states S and a set of joint actions A [39]. The advantages are that these methods handle coordination better and avoid non-stationarity, but they become impractical to implement with many agents due to exponential scaling.

In between these two approaches is a paradigm known as centralized training with decentralized execution, in which agents are trained in a centralized fashion (i.e. information is shared between them), but operate in a decentralized manner during test time [24], [13]. These approaches can leverage additional information and are good for simulation settings, but there is no obvious way to extract a policy.

Emergent Behaviors

Interaction between multiple policies can produce emergent behaviors, which are behaviors that are not attributed to the behavior of an individual agent, but rather a global outcome of the system. Previous works in this domain analyze different scenarios from cooperative to competitive to a mix of both; in this work, we focus on the competitive case. The complexity of behavior acquired by the trained agent is usually dependent on the complexity of the environment. However, as pointed out by [2], a multi-agent system trained in a competitive manner with self-play can produce behaviors that are far more complex than the environment itself. Our method showcases a similar phenomenon wherein after each agent adapts, the learning problem for the other agent becomes increasingly difficult, leading to the emergence of an automatic curriculum of challenging learning tasks. This approach is most similar to Asymmetric Self-Play (ASP) [33], [27]; however, our method is applicable in more scenarios because, unlike ASP, it is formulated with general information theoretic quantities.

2.3 Intrinsic Motivation

Intrinsic motivation is based in human psychology, in which an intrinsically motivated behavior is defined as one “engaged for its own sake rather than as a step toward solving a specific problem” [9]. The main way to integrate an intrinsic reward into a RL framework is to consider it as a “bonus” term involving a weighted sum between the external reward and intrinsic reward $r = \alpha r_{\text{int}} + \beta r_{\text{ext}}$ [18], [7], which is what we use for Adversarial Surprise. An example model of this framework is shown in Figure 2.1.

Novelty Seeking

Typically, intrinsic motivation is incorporated into frameworks to encourage novelty-seeking behaviors by maximizing uncertainty. One approach uses count-based exploration, determining the novelty of a particular state based on how many times it has been seen. This approach doesn’t scale well in high-dimensional state space environments, so approaches

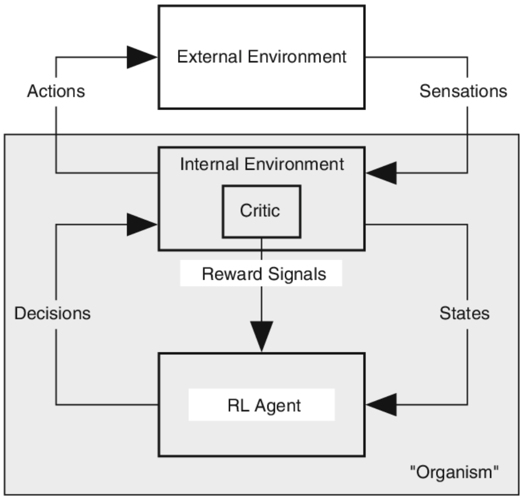

Figure 2.1: Model of an organism with internal and external motivation

such as [4] use a density model to approximate the frequency of state visits and then derive a pseudo-count from this model.

Another approach uses rewards that are based on an agent's knowledge of the environment estimated through a prediction framework. One example is the Intrinsic Curiosity Module [29], which learns a state space encoding with a self-supervised inverse dynamics model. These methods can suffer from the aforementioned "noisy TV problem", in which agents get stuck at highly-entropic elements [30]. However, we show that Adversarial Surprise can break out of this behavior.

Surprise Minimization

Intrinsic motivation can also be used to incentivize an agent to minimize surprise over the distribution of states generated by the policy, as described in SMiRL [6]. Specifically, SMiRL uses a density model $p_\theta(s)$ to keep track of the state history, and finds that the intrinsic reward $r(s_t) = \log p_\theta(s)$ can be used as an upper bound on the entropy of the state marginal density. This approach and others like it are based on the free-energy principle [17]. However, agents trained in this why are vulnerable to the "dark room problem", in which agents in partially-observed or low-entropy environments learn to stay in a highly-predictable area of the environment and never learn meaningful behavior [15]. Adversarial Surprise uses the Explore policy, which is designed to seek out entropic areas of the environment, to avoid this problem.

Chapter 3

Adversarial Surprise

3.1 Algorithm

Adversarial Surprise (AS) [12] aims to have a single embodied agent learn both a meaningful exploration and control strategy. This process is learned through a two-player game between, π^E, an exploration policy and π^C, a control policy. The Control policy uses a learned model p_θ to minimize its own surprise, or observation entropy. The Explore policy's goal is to maximize the surprise that the Control policy experiences. The policies take turns taking actions for the agent, switching back and forth throughout the episode. The policy controlling the RL agent changes every k steps, such that:

$$a_t \sim \begin{cases} \pi^E(a_t|o_t) & \text{if } \exists n, t \in [2nk, (2n+1)k] \\ \pi^C(a_t|o_t) & \text{otherwise} \end{cases} \tag{3.1}$$

Each policy is given several steps to act. This enables it to reach states that will be challenging for the other policy to recover from, thus facilitating learning more complex and long-term exploration and control behaviors.

To estimate surprise, we learn a density model which estimates the agent's likelihood of experiencing observation o, $p_\theta(o)$. Because the Control policy is surprise-minimizing, its reward is $r(s_t) = \log p_\theta(o_t)$, which resembles SMiRL [6], but using the observation in place of the state. The goal of the Explore policy is to maximize the observation surprise of the RL agent when the Control policy is in control. This creates an adversarial game in which the Explore policy attempts to find surprising situations with which to expose the Control policy, and the Control policy's objective is to recover from them. Therefore, the Explore policy's reward is based on the surprise for the observations of the Control policy. Adversarial Surprise thus defines the following adversarial game between the two policies:

Algorithm 1: Adversarial Surprise

```
Randomly initialize $\phi^E$ and $\phi^C$;
for $episode = 0, ..., M$ do
    Initialize $\theta, R^i = 0$, $\beta \leftarrow \{\}$, explore_turn = True, $t^C = k$,
     $s_0 \sim p(s_0), o_0 \sim p(O_0|s_0)$;
    for $t \leftarrow 0$ to $T$ do
        if explore_turn then
            $a_t \sim \pi^E(o_t, h_t^E)$ ;                                           // Explore
        else
            $a_t \sim \pi^C(o_t, h_t^C)$ ;                                           // Control
        end
        $s_{t+1} \sim \mathcal{T}(s_{t+1}|s_t, a_t)$, $o_{t+1} \sim p(O_{t+1}|s_{t+1})$ ;             // Environment step
        $r_t^i = \log p_\theta(o_{t+1})$ ;                               // Compute intrinsic reward
        if not explore_turn and $t - t^C > k/2$ then
            $R^i = R^i + r^i$ ;
        end
        $\beta = \beta \cup \{o_t, a_t, o_{t+1}\}$ ;                                // Update buffer
        if $t == t^C$ then
            explore_turn = not explore_turn ;                              // Switch turns
            $t^C = t^C + 2k$;
        end
        $\theta_{t+1}$ =MLE_update($\beta, \theta_t$) ;                              // Fit density model
    end
    $\phi^E$ =RL_update($\beta, -R^i$) ;    // Train Explore policy $\pi^E$ with reward $-R^i$
    $\phi^C$ =RL_update($\beta, R^i$) ;       // Train Control policy $\pi^C$ with reward $R^i$
end
```

$$\max_{\pi^E} \min_{\pi^C} -E\left[\sum_{t=t^C}^{t^C+k} \log p_\theta(o_t)\right] \tag{3.2}$$

[12] additionally provides a theoretical derivation on how the algorithm fully covers the latent state space of a large family of POMDPs under some restrictions on their structure. We present a high-level overview of the procedure in Algorithm 1 and then give specific implementation details in Section 3.2.

3.2 Implementation

All of our implementations are in PyTorch [28], a free and open-source machine learning library which provides an intuitive framework for reinforcement learning. Below, we include relevant implementation details unique to Adversarial Surprise.

Preprocessing

Most of our experiments use the Atari Learning Environment (ALE) [3], with preprocessing wrappers implemented in [11]. We use the Vectorized Environment implementation from [19], which allows for stacking multiple independent environments into a single environment. Notably, the environments are automatically reset at the end of each episode with this implementation, which is reflected in the policy section below. The MiniGrid environments are trained with 16 parallel environments [12]. Although we ended up training with one environment at a time for the non-MiniGrid experiments, the efficiency of this implementation still reduced training time.

For Atari, we use the "NoFrameskip-v4" version of the environments, which has no action repeat stochasticity. This contrasts with v0 which has a "repeat action probability" of 0.25, meaning that 25% of the time, the action that was used in the previous time step will be used instead of the new action. It also does not use any frame skips which contrasts with the deterministic and non-deterministic environments that use a fixed frameskip of 4 and a frameskip uniformly sampled from $(2, 5)$. For the Take Cover scenario in VizDoom, we enable only left and right actions, turn off god mode, and use a time limit of 1000.

Models and Surprise Wrapper

Listing 3.1: Explore Model

```
class ExploreModel(Module):
    def __init__(self, envs, frames=4):
        super(ExploreModel, self).__init__()
        self.network = Sequential(
            Scale(1/255),
            layer_init(Conv2d(frames, 32, 8, stride=4)),
            ReLU(),
            layer_init(Conv2d(32, 64, 4, stride=2)),
            ReLU(),
            layer_init(Conv2d(64, 64, 3, stride=1)),
            ReLU(),
            Flatten(),
            layer_init(Linear(3136, 512)),
            ReLU()
        )
        self.actor = layer_init(Linear(512, envs.action_space.n)
                     ,std=0.01)
        self.critic = layer_init(Linear(512, 1), std=1)

    def forward(self, x):
        x = self.network(x)
        return x
```

Listing 3.2: Control Model

```
class ControlModel(Module):
    def __init__(self, envs, frames=4, control_samps=None):
        super(ControlModel, self).__init__()
        self.network = Sequential(
            Scale(1/255),
            layer_init(Conv2d(frames, 32, 8, stride=4)),
            ReLU(),
            layer_init(Conv2d(32, 64, 2, stride=2)),
            ReLU(),
            layer_init(Conv2d(64, 64, 2, stride=1)),
            ReLU(),
            Flatten())
        self.network2 = Sequential(
            layer_init(Linear(3585, 512)),
            ReLU()
        )
        self.actor = layer_init(Linear(512, envs.action_space.n)
                     ,std=0.01)
        self.critic = layer_init(Linear(512, 1), std=1)

    def forward(self, x, batch=0):
        x = self.network(x)
        num_samples = self.envs.num_samples
        x = cat((x, num_samples), 1)
        x = self.network2(x)
        return x
```

Listing 3.3: Surprise Wrapper

```
class BaseSurpriseWrapper(gym.Wrapper):
    def __init__(self, env, buffer, time_horizon,
    environment_weight=0.7, clip_smirl=0,
    smirl_scale=0.001, obs_space_key=None):
        self.buffer = buffer
        theta = self.buffer.get_params()
        self.env_obs_space = env.observation_space
        obs_dic = {"obs": deepcopy(self.env_obs_space)}
        shape = list(self.env_obs_space.low.shape)
        shape = [4, 20, 60]
        obs_dic.update({'augmented': Box(
               zeros(shape),
                ones(shape))
            })
        self.observation_space = Dict(obs_dic)
    def num_samples(self):
        return self.buffer.buffer_size
    def step(self, action):
        obs, env_rew, envdone, info = self.env.step(action)
        info['external_reward'] = env_rew
        encoded_obs = self.encode_obs(obs)
        self.last_enc_obs = encoded_obs
        self.last_enc_obs = resize(encoded_obs, (4, 20, 20))
        encoded_obs_resized = resize(encoded_obs, (4, 20, 20))
        rew = self.buffer.logprob(encoded_obs_resized)
        if self.clip_smirl:
            rew = clip(rew, -self.clip_smirl, self.clip_smirl)
        rew *= self.smirl_scale
        encoded_obs_resized = resize(encoded_obs, (4, 20, 20))
        self.buffer.add(encoded_obs_resized)
        obs = self.augment_obs(obs, encoded_obs_resized)
        obs = obs if self.dict_obs else obs['augmented']
        return obs, rew, envdone, info

    def augment_obs(self, obs, encoded_obs):
        theta = self.buffer.get_params()
        num_samples = ones(1) * self.buffer.buffer_size
        shape = list(encoded_obs.shape)
        shape = (4, 20, 40)
        theta = theta.reshape(shape)
        encoded_obs = resize(encoded_obs, (4, 20, 20))
        augmented_obs = concatenate((encoded_obs, theta), axis = 2)
        obs = {"obs":self.encode_obs(obs)}
        obs['augmented'] = augmented_obs
        return obs
    def reset_buffer(self):
        if self.last_enc_obs is None:
            assert False
        self.buffer.reset()
        self.buffer.add(self.last_enc_obs)

    def reset(self):
        obs = self.env.reset()
        encoded_obs = self.encode_obs(obs)
        self.encode_obs(obs)
        self.buffer.reset()
        encoded_obs_resized = resize(encoded_obs, (4, 20, 20))
        self.buffer.add(encoded_obs_resized)
        obs = self.augment_obs(obs, encoded_obs_resized)
        obs = obs if self.dict_obs else obs['augmented']
        return obs

    def encode_obs(self, obs):
        return obs.copy()

class GaussianBuffer(BaseBuffer):
    def __init__(self, obs_dim):
        super().__init__()
        self.buffer = zeros((1, obs_dim))
        self.buffer_size = 1
        self.obs_dim = obs_dim
        self.add(ones((1,obs_dim)))
        self.add(-ones((1,obs_dim)))

    def add(self, obs):
        self.buffer = concatenate((self.buffer, obs.flatten()))
        self.buffer_size += 1

    def get_params(self):
        means = mean(self.buffer, axis=0)
        stds = std(self.buffer, axis=0)
        params = concatenate([means, stds])
        return params

    def logprob(self, obs):
        obs = obs.copy().flatten()
        means = mean(self.buffer, axis=0)
        stds = std(self.buffer, axis=0)
        stds = stds + thresh
        logprob = -0.5 * sum(log(2 * pi * stds)) - sum(square(obs - means)
        / (2 * square(stds)))
        return logprob

    def reset(self):
        self.buffer = zeros((1, self.obs_dim))
        self.buffer_size = 1
```

These are Explore and Control models for the Atari and VizDoom environments. As we can compare, both the Explore model and Control model are virtually identical convolutional models. The MiniGrid experiments use 3 convolutional layers with 16, 32 and 64 output channels and a stride of 2 for every layer. The difference between the Explore and Control models lies in the input and the additional parameter representing number of samples that is concatenated to the flattened feature representation.

The input to the Control Model is an “augmented state”, which is the original state in addition to sufficient statistics of the density model, $p_{\theta t}(o)$. To greatly reduce the number of parameters θ for this model, we found that reducing image size to 20 x 20 was helpful. In the above case for Atari, we resize the original input from (4, 192, 160) to (4, 20, 20). We also stack together the latest 4 images in VizDoom, resizing each image from a grayscale (48, 64) resolution to (20, 20). We found that adding a sample of positive and negative 1 after resetting the density model’s buffer helped stabilize learning when taking the log probability.

Above we show the case for $p_{\theta t}(o)$ being a Gaussian distribution, which is used in Atari and VizDoom. In this case, we use the mean of the sample and the standard deviation along with the number of states seen so far representing a sufficient statistic. For the MiniGrid experiments, we use 7×7 independent categorical distributions with 12 classes each for the density model, with the sufficient statistic being the set of $7 \times 7 \times 12$ probabilities. This augmented MDP is based on [6]; however, instead of using states, we use observations. This general technique has shown to be helpful empirically: [14] introduces a similar type of “fingerprint” shown to stabilize experience replay for multi-agent learning.

Policy

Listing 3.4: Policy Update

```
explore_obs = zeros((explore_num_steps))
+ envs.observation_space[”obs”].shape)
explore_actions = zeros((explore_num_steps)
+ envs.action_space.shape)
explore_logprobs = zeros((explore_num_steps))
explore_rewards = zeros((explore_num_steps)
explore_dones = zeros((explore_num_steps)
explore_values = zeros((explore_num_steps)

control_obs = zeros((control_num_steps)
+ envs.observation_space[”augmented”].shape)
control_actions = zeros((control_num_steps)
+ envs.action_space.shape)
control_logprobs = zeros((control_num_steps))
control_rewards = zeros((control_num_steps))
control_dones = zeros((control_num_steps))
control_values = zeros((control_num_steps))
control_samps = zeros((control_num_steps))
while episode_number < total_episodes:
    #Explore
    lives = envs.unwrapped.ale.lives()
    for step in range(0, explore_num_steps):
        global_step += 1
        explore_obs[step] = next_obs[”obs”]
        explore_dones[step] = next_done
        with no_grad():
            explore_values[step] = explore_agent.get_value(explore_obs[step])
                                   .flatten()
            action, logprob, _ = explore_agent.get_action(explore_obs[step])
        explore_actions[step] = action
        explore_logprobs[step] = logproba
        next_obs, rs, ds, infos = envs.step(action)
        rs = zeros()
        explore_rewards[step] = rs
        next_done = ds
        if envs.unwrapped.ale.lives() < lives:
            lives = envs..unwrapped.ale.lives()
            envs.reset_buffer()
        with no_grad():
            last_value = explore_agent.get_value(next_obs[”obs”])
            #GAE for Explore policy
    #Control
    cum_control = 0
    for step in range(0, control_num_steps):
        global_step += 1
        next_obs = next_obs[”augmented”]
        control_obs[step] = next_obs
        control_samps[step] = envs.num_samples
        bob_dones[step] = next_done
        with no_grad():
            control_values[step] = control_agent.get_value(control_obs[step])
                                   .flatten()
            action, logprob, _ = control_agent.get_action(control_obs[step])
        control_actions[step] = action
        control_logprobs[step] = logprob
        next_obs, rs, ds, infos = envs.step(action)
        if second_half:
            if step >= control_num_steps // 2:
                cum_control += rs
        else:
            cum_control += rs
        if step == control_num_steps - 1:
            explore_rewards[explore_num_steps-1] += -cum_control
            cum_control = 0
        if envs.unwrapped.ale.lives() < lives:
            lives = envs.unwrapped.ale.lives()
            envs.reset_buffer()
        #GAE for Control policy
    # Standard PPO Update for policy and value network
    # First for explore policy then control
    ...
```

As explained in chapter 2, there are many different paradigms possible for multi-agent optimization. We utilize recent empirical analysis from works such as [38] and [40] which state that Proximal Policy Optimization (PPO) [32] utilized in an independent learning approach is surprisingly effective in the multi-agent domain. Furthermore, we utilized Generalized Advantage Estimation [31] which offers a good balance between bias and variance for our advantage estimator.

As we can see, the Explore policy's reward is based on the surprise for the observations of the Control policy. If we assume the Control policy's turn begins at timestep t^C, and it receives a total reward of $R = \sum_{t=t^C}^{t^C+k} \gamma^k r(a_t, s_t)$ for that turn, then, the Explore policy's reward is $-R$, and is applied to the last timestep of the Explore policy's turn (i.e. timestep $t^C - 1$). We have also found it helpful to only compute the surprise reward using observations from the second half of the Control policy's turn; this gives the agent greater ability to take actions that may lead to initial surprise, but reduce entropy over the long term.

We also experimented with when to reset the buffer, in the MiniGrid experiments we we find that resetting the buffer after each round (after the Explore policy and Control policy each take one turn) can sometimes improve performance [12]. For the Atari and Vizdoom experiments, we use the above code that resets the buffer whenever a life is lost. Furthermore, we can see that the code allows the Explore and Control policies to act for a different number of timesteps, tuning the emphasis on exploration or control depending on the environment. For Atari and Vizdoom we use 64 steps for the Explore policy and 128 steps for the Control Policy. In the Minigird experiments, both the Explore and Control policies act for 2 rounds with 32 steps per round for each episode.

3.3 Experimental Analysis

To determine how to evaluate our algorithm, we come with specific criterion that we are looking to optimize for:

- **Exploration and state coverage:** One of the primary goals is that AS should be able to thoroughly explore the state space of a stochastic, partially-observed environment.
- **Control:** In addition to exploration, we want to evaluate how well AS exhibits control by taking actions that will decrease surprise in the environment.
- **Emergence of complexity:** Finally, we want to see the complex behaviors that the interaction of these opposite objectives can result in.

Environment Selection

To evaluate these criteria we look for partially-observed environments that present an exploration challenge, have stochastic phenomena, and offer a degree of complexity that can allow for interesting emergent behavior.

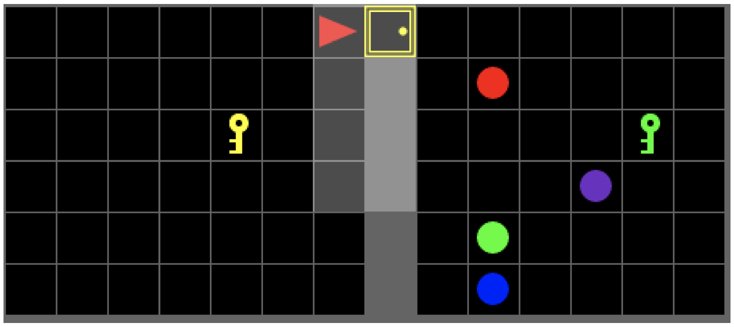

Figure 3.1: **Minigrid:** As shown in [12], we construct a custom family of environments based on MiniGrid [10] that incorporate the above properties. These environments contain rooms that are either empty (dark), or contain stochastic elements such as flashing lights that randomly change color. They also contain elements such as doors that can be opened, and switches that, when flipped, stop the stochastic elements from changing. The agent only sees a 5x5 window of the true state, making the environment partially-observed.

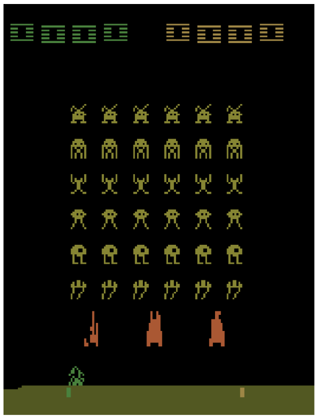

Figure 3.2: **Space Invaders:** The player controls a laser cannon by moving it horizontally across the bottom of the screen and firing at descending aliens. The goal is to defeat all of these aliens before they arrive at the bottom of the screen while avoiding their shots. They start to speed up after every wave has been cleared. The laser cannon is protected by several stationary defense bunkers, but these can get damaged by both the player and aliens. [37]

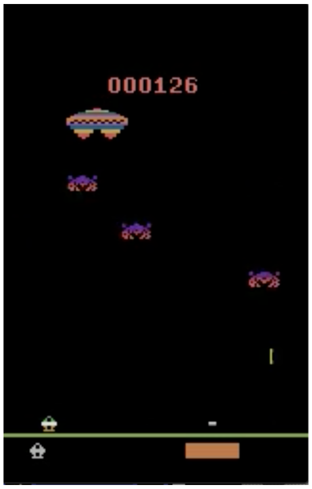

Figure 3.3: **Assault**: The player controls a weapon and faces off with an alien mother ship, which continually deploys three smaller ships during play. The mother ship and the smaller vessels shoot at the player and the player's aim is to eliminate them while preventing getting destroyed. [37]

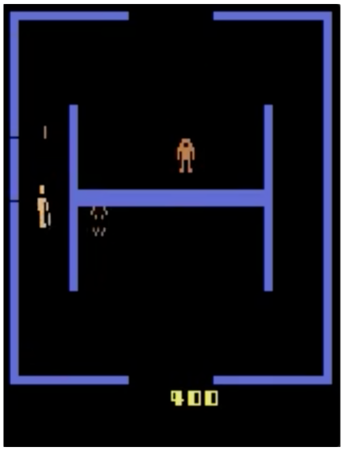

Figure 3.4: **Berzerk**: The player controls a stick man with a weapon within a maze filled with robots that fire lasers at the player. The player is killed by being shot, by running into a robot, or coming into contact with the electrified walls of the maze, The player advances by escaping from the maze through an opening in the wall. If the player destroys all the robots in the current maze before escaping, they receive a per-maze bonus. [37]

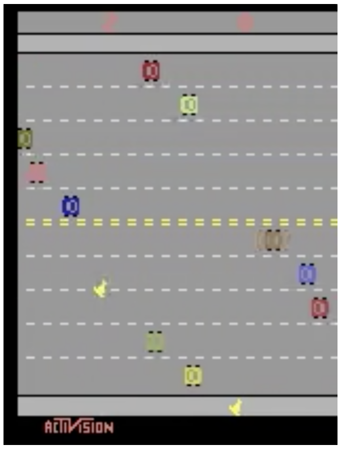

Figure 3.5: **Freeway**: The player controls a chicken with the goal of running across a ten lane highway with incoming traffic. The player is reward every time they get the chicken all the way to the other side, if they are hit by. a car, they are forced back slightly. [37]

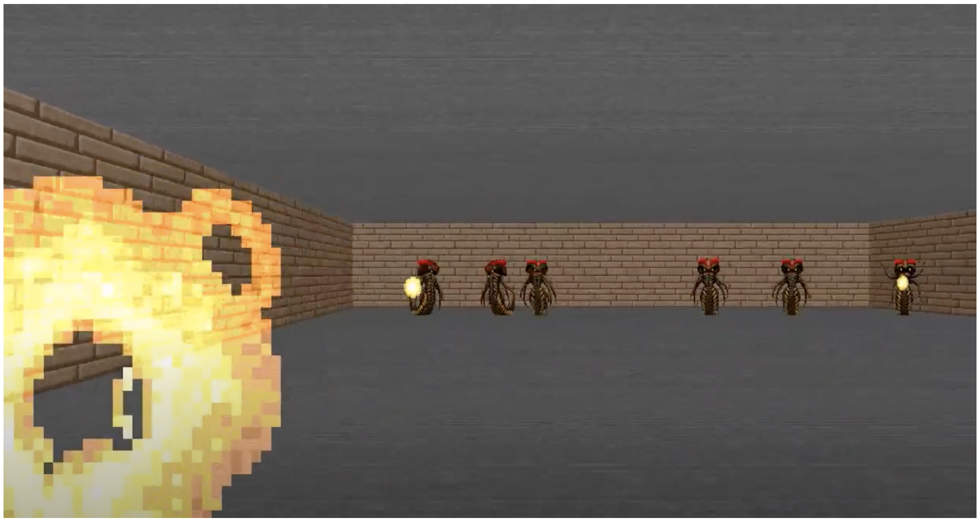

Figure 3.6: **VizDoom**: We consider one VizDoom environment from [21]- Take Cover. The player controls an agent that must learn to avoid fireballs shot by monsters from the other side of the room. The Take Cover environment was also used in [6] because of the evolving nature of new enemies that appear.

Baselines

We compare AS to three competitive unsupervised RL baselines: Random Network Distillation (RND) [7] (a state-of-the-art exploration method), SMiRL [6](a recently proposed method based on surprise minimization), and Asymmetric Self-Play [33] (a state-of-the-art multi-agent curriculum method).

Results

Exploration

To evaluate thorough state-space exploration, we see how many rooms the agent learns to visit within the MiniGrid environments [12]. Figure 3.7 shows the results across the cumulative training procedure for the agent.

- **RND** becomes distracted by the stochasticity within the environment, leading to higher prediction error, although it is tied with AS for most amount of rooms reached.
- **SMiRL** is not able to sufficiently explore, due to the partially observed nature of the environment, it suffers from the dark room problem and does not venture into rooms with stochastic elements.
- **ASP** is shown to handle the MiniGrid environments poorly, because their stochasticity makes it difficult for Bob to replicate the goals that Alice produces.
- **AS** not only visits the most rooms compared to the other baselines, it also does so the most quickly.

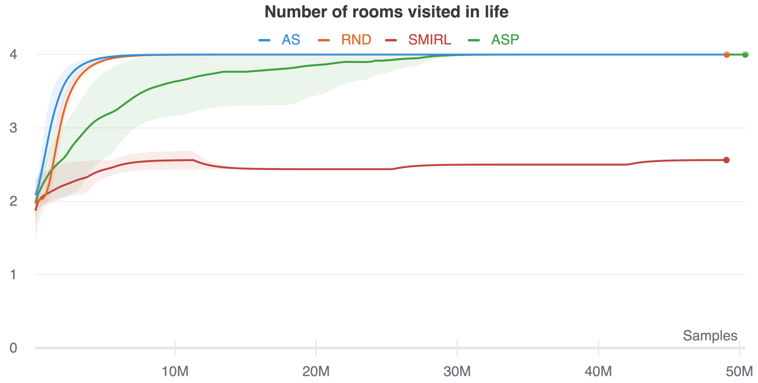

Figure 3.7: Rooms Visited Plot

Control

To see the effects of surprise-minimizing control, the average number of times the agent presses switches that stop the stochastic elements in the environment from changing color is measured [12].

- **RND** has no incentive to learn to control the environment, and thus never learns to press the switch.
- **SMiRL** learns to take entropy-reducing actions, and performs comparably to **AS** as-is.
- **ASP** yields similar results, since reducing the entropy would make it easier for the Bob agent to replicate the Alice agent's final state. Thus, ASP will not always lead the agent to learn all possible behaviors relevant to controlling the environment.
- When we train **AS** by resetting the buffer β used to fit the density model after each *round* rather than each episode (i.e. after both policies have taken one turn), we see that the number of entropy-reducing actions it takes increases to exceed SMiRL's. This is likely because resetting the buffer eliminates the incentive to return to previously-seen states, therefore encouraging the agent to reduce entropy immediately.

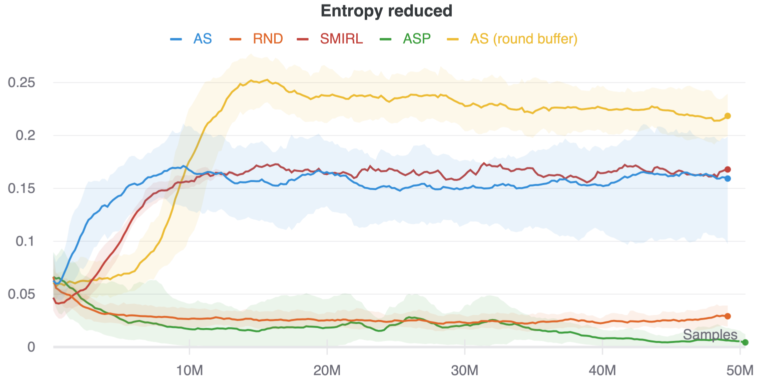

Figure 3.8: Control Plot

Emergence

In the modified MiniGrid environment [12], we have one dark room and one noisy room separated by a door whose position changes for every episode, with the agent beginning in the noisy room. We see an emergence of behavior of increasing complexity through phases, wherein the Control policy first learns to get to the dark room. Then, the Explore policy learns to go back to the noisy room and reach a point where the Control policy can not

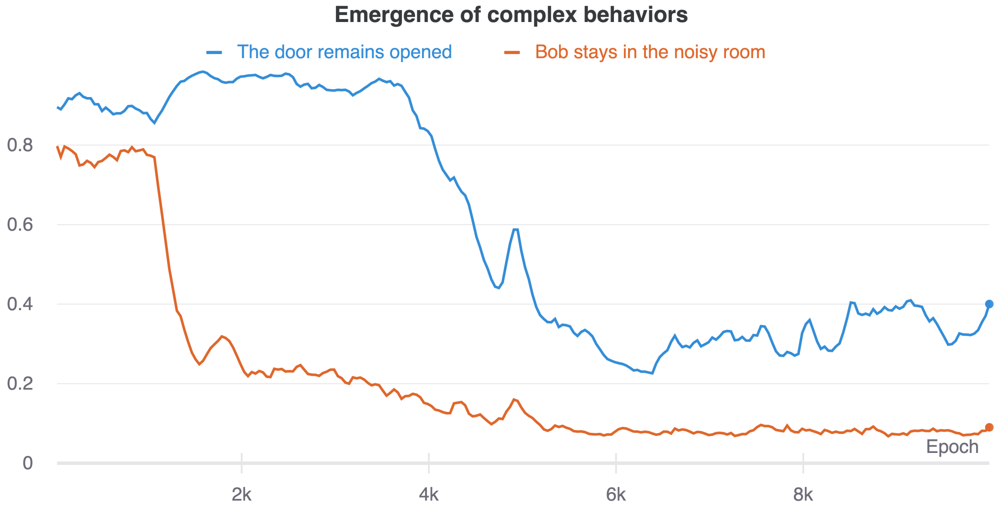

Figure 3.9: Emergence Plot

take the agent back to the dark room within its allocated amount of steps. This leads the Control policy to take the agent into the dark room and close the door to make it harder for the Explore policy to reach the noisy room in subsequent rounds. These actions show how Adversarial Surprise can learn an auto-curricula of increasing difficulty through multi-agent competition.

For the next set of experiments, we use only intrinsic reward, then assess the amount of task reward obtained in the standard Atari benchmark [3] and Vizdoom environments [21]. Many Atari games involve the concepts of both progress, which is tied to coverage or portion completed, and 'death', an event where all progress is lost. Notably, in existing literature, there are claims that both novelty-seeking exploration methods [7] and surprise-minimization methods [6] achieve high scores in these settings. Thus, we hypothesize that AS, a method designed to simultaneously maximize coverage and maintain control, may adapt to and perform well in a variety of these games.

The plots below show game reward across several Atari environments and the VizDoom Take Cover environment for Adversarial Surprise compared with RND, SmiRL, and ASP. These environments reward a range of behaviors including avoiding obstacles, collecting items, and attacking enemies.

- **RND**'s novelty-maximization strategy performs well in certain environments, but often gradually degrades due to a shrinking bonus as it sees states more often. It also tends to die frequently in high-risk environments like Doom, Freeway, and Space Invaders.

- **SMiRL**'s entropy minimization allows it to survive for longer in environments like Freeway, but translates to hiding from enemies in ones like Assault and Space Invaders, blocking it from achieving high rewards.

- In **ASP**, Alice can reach particular states quickly which Bob cannot replicate, which hinders learning.
- Finally, **AS** achieves high rewards in each of these environments, demonstrating the efficacy of optimizing for both exploration and control when task reward is unknown.

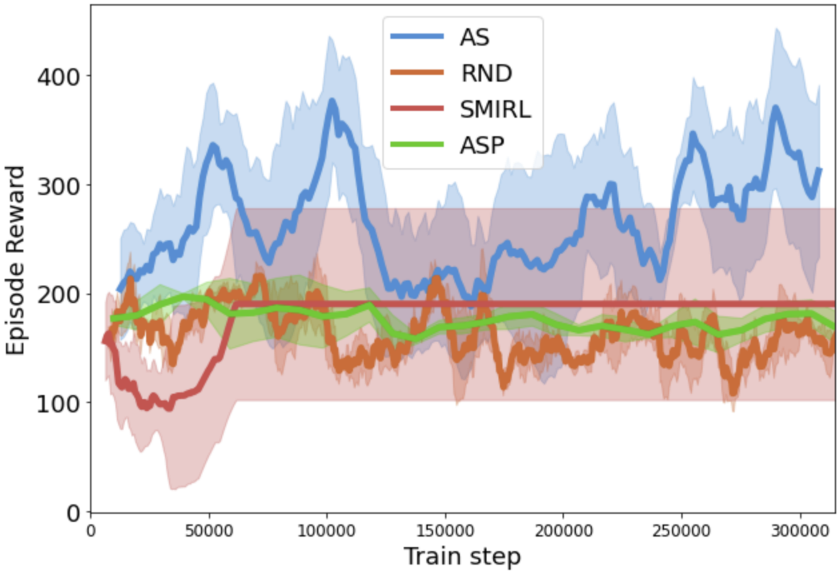

Figure 3.10: Space Invaders Plot

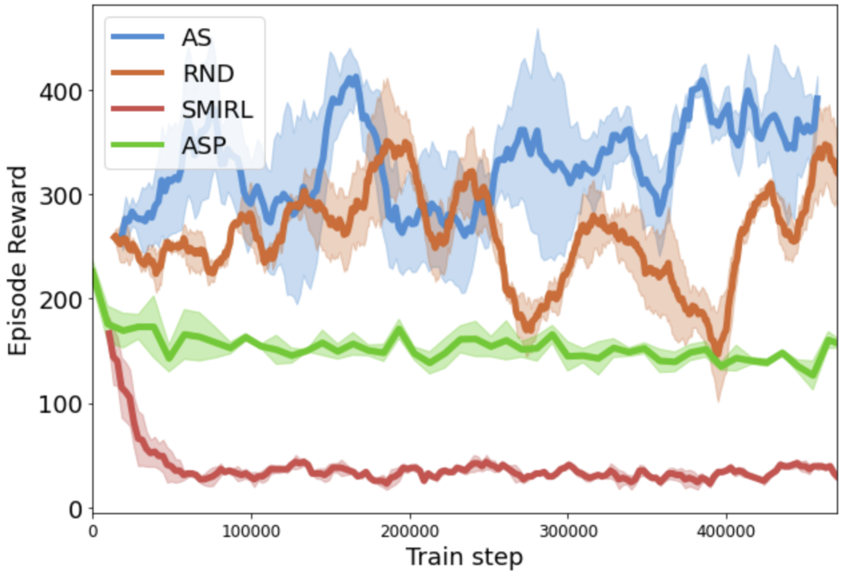

Figure 3.11: Assault Plot

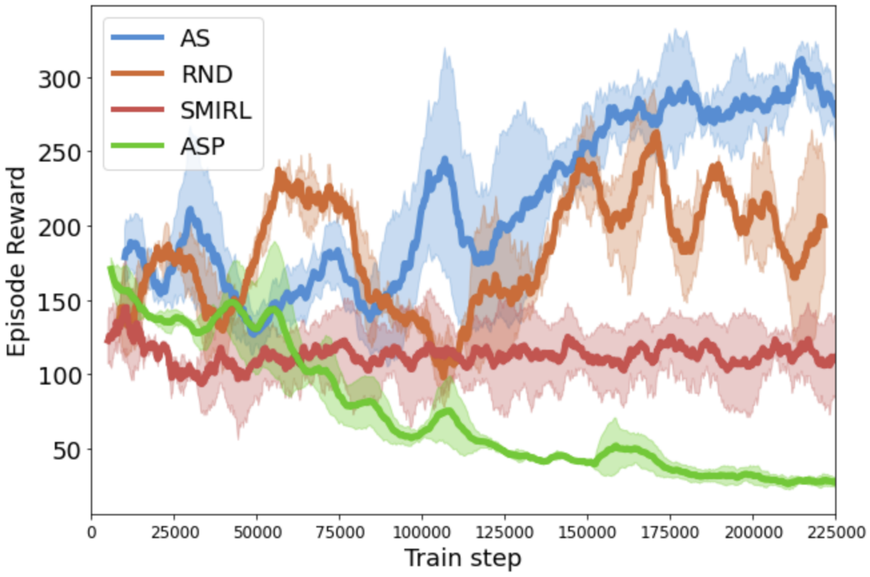

Figure 3.12: Berzerk Plot

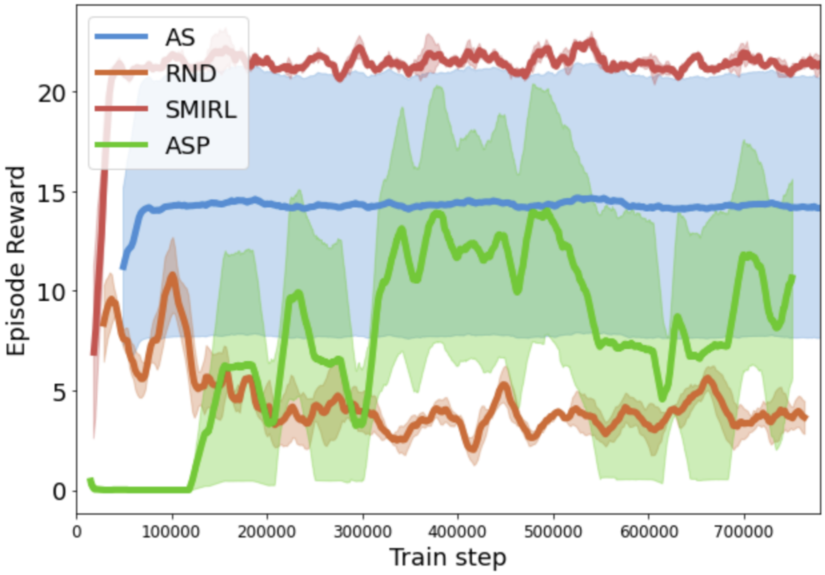

Figure 3.13: Freeway Plot

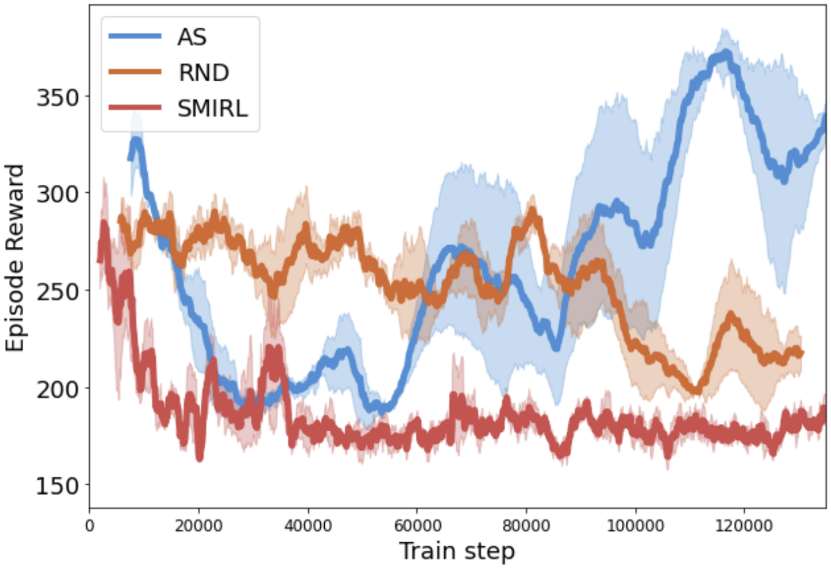

Figure 3.14: Vizdoom Plot

Chapter 4

Conclusion and Future Work

We present an empirical analysis of Adversarial Surprise, a novel unsupervised multi-agent strategy based on the simultaneous maximization and minimization of surprise. In this report we show that AS is robust against issues faced by state-of-the-art prior works like RND, ASP, and SMiRL and is able to explore stochastic, partially observed environments more thoroughly and control them more effectively.

Our experiments on Atari and VizDoom focused on the "no external reward" setting to see the emergence of behavior that could arise from just using intrinsic reward. For future work, it would be interesting to compare to methods that use both intrinsic and extrinsic reward to see the performance gain. In addition, further ablation on the effect of the step size should be conducted as it was seen to make an impact on performance. We can additionally explore ways of making this parameter learnable or even introduce a meta-controller trained with a hierarchical policy that can automatically switch between Explore and Control policy.

4.1 Statement of Contributions

My contributions to the project involved adapting and implementing a version of the algorithm for the Atari and VizDoom environments. Arnaud, Natasha, and Sergey designed the core algorithm. Arnaud implemented the algorithm and ran the Minigrid experiments, the ASP baseline, and along with Natasha worked on the primary theoretical and conceptual arguments. Michael, Nick, Glen, and Sergey provided valuable insights to solidify the conceptual points and helped with directions to take for the experiments.

Bibliography

[1] Dario Amodei et al. *Concrete Problems in AI Safety.* 2016. arXiv: `1606.06565 [cs.AI]`.

[2] Trapit Bansal et al. *Emergent Complexity via Multi-Agent Competition.* 2018. arXiv: `1710.03748 [cs.AI]`.

[3] Marc G Bellemare et al. "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[4] Marc G. Bellemare et al. "Unifying Count-Based Exploration and Intrinsic Motivation". In: *CoRR* abs/1606.01868 (2016). arXiv: `1606.01868`. URL: `http://arxiv.org/abs/1606.01868`.

[5] Christopher Berner et al. "Dota 2 with Large Scale Deep Reinforcement Learning". In: *CoRR* abs/1912.06680 (2019). arXiv: `1912.06680`. URL: `http://arxiv.org/abs/1912.06680`.

[6] Glen Berseth et al. "SMiRL: Surprise Minimizing RL in Dynamic Environments". In: *CoRR* abs/1912.05510 (2019). arXiv: `1912.05510`. URL: `http://arxiv.org/abs/1912.05510`.

[7] Yuri Burda et al. "Exploration by random network distillation". In: *arXiv preprint arXiv:1810.12894* (2018).

[8] Yuri Burda et al. *Large-Scale Study of Curiosity-Driven Learning.* 2018. arXiv: `1808.04355 [cs.LG]`.

[9] Nuttapong Chentanez, Andrew Barto, and Satinder Singh. "Intrinsically Motivated Reinforcement Learning". In: *Advances in Neural Information Processing Systems.* Ed. by L. Saul, Y. Weiss, and L. Bottou. Vol. 17. MIT Press, 2005. URL: `https://proceedings.neurips.cc/paper/2004/file/4be5a36cbaca8ab9d2066debfe4e65c1-Paper.pdf`.

[10] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. *Minimalistic Gridworld Environment for OpenAI Gym.* `https://github.com/maximecb/gym-minigrid`. 2018.

[11] Prafulla Dhariwal et al. *OpenAI Baselines.* `https://github.com/openai/baselines`. 2017.

[12] Arnaud Fickinger et al. *Explore and Control with Adversarial Surprise*. 2021. arXiv: 2107.07394 [cs.LG].

[13] Jakob Foerster et al. *Counterfactual Multi-Agent Policy Gradients*. 2017. arXiv: 1705.08926 [cs.AI].

[14] Jakob N. Foerster et al. “Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning”. In: *CoRR* abs/1702.08887 (2017). arXiv: 1702.08887. URL: http://arxiv.org/abs/1702.08887.

[15] Karl Friston, Christopher Thornton, and Andy Clark. “Free-energy minimization and the dark-room problem”. In: *Frontiers in psychology* 3 (2012), p. 130.

[16] Karl Friston et al. “Action and behavior: A free-energy formulation”. In: *Biological cybernetics* 102 (Feb. 2010), pp. 227–60. DOI: 10.1007/s00422-010-0364-z.

[17] Karl Friston et al. “Active inference and learning”. In: *Neuroscience & Biobehavioral Reviews* 68 (2016), pp. 862–879.

[18] Karol Gregor, Danilo Jimenez Rezende, and Daan Wierstra. *Variational Intrinsic Control*. 2016. arXiv: 1611.07507 [cs.LG].

[19] Ashley Hill et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2018.

[20] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. “Planning and acting in partially observable stochastic domains”. In: *Artificial Intelligence* 101.1 (1998), pp. 99–134. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(98)00023-X. URL: https://www.sciencedirect.com/science/article/pii/S000437029800023X.

[21] Michal Kempka et al. “ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning”. In: *CoRR* abs/1605.02097 (2016). arXiv: 1605.02097. URL: http://arxiv.org/abs/1605.02097.

[22] Guillaume Lample and Devendra Singh Chaplot. “Playing FPS Games with Deep Reinforcement Learning”. In: *CoRR* abs/1609.05521 (2016). arXiv: 1609.05521. URL: http://arxiv.org/abs/1609.05521.

[23] Guillaume J. Laurent, Laëtitia Matignon, and N. Le Fort-Piat. “The World of Independent Learners is Not Markovian”. In: 15.1 (2011). ISSN: 1327-2314.

[24] Ryan Lowe et al. *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments*. 2020. arXiv: 1706.02275 [cs.LG].

[25] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[26] Ashvin Nair et al. *Overcoming Exploration in Reinforcement Learning with Demonstrations*. 2018. arXiv: 1709.10089 [cs.LG].

[27] OpenAI OpenAI et al. “Asymmetric self-play for automatic goal discovery in robotic manipulation”. In: *arXiv preprint arXiv:2101.04882* (2021).

[28] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: (2017).

[29] Deepak Pathak et al. *Curiosity-driven Exploration by Self-supervised Prediction.* 2017. arXiv: `1705.05363 [cs.LG]`.

[30] Jürgen Schmidhuber. “Formal theory of creativity, fun, and intrinsic motivation (1990–2010)”. In: *IEEE Transactions on Autonomous Mental Development* 2.3 (2010), pp. 230–247.

[31] John Schulman et al. “High-Dimensional Continuous Control Using Generalized Advantage Estimation”. In: *Proceedings of the International Conference on Learning Representations (ICLR).* 2016.

[32] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: `1707.06347`. URL: `http://arxiv.org/abs/1707.06347`.

[33] Sainbayar Sukhbaatar et al. “Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play”. In: *CoRR* abs/1703.05407 (2017). arXiv: `1703.05407`. URL: `http://arxiv.org/abs/1703.05407`.

[34] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[35] Ming Tan. “Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents”. In: *In Proceedings of the Tenth International Conference on Machine Learning.* Morgan Kaufmann, 1993, pp. 330–337.

[36] Mel Vecerik et al. “Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards”. In: *arXiv preprint arXiv:1707.08817* (2017).

[37] B. Weiss. *Classic Home Video Games, 1972-1984: A Complete Reference Guide.* McFarland, Incorporated, Publishers, 2011. ISBN: 9780786487554. URL: `https://books.google.com/books?id=BzxTtml8Jq4C`.

[38] Christian Schröder de Witt et al. “Is Independent Learning All You Need in the StarCraft Multi-Agent Challenge?” In: *CoRR* abs/2011.09533 (2020). arXiv: `2011.09533`. URL: `https://arxiv.org/abs/2011.09533`.

[39] Ping Xuan and Victor Lesser. “Multi-Agent Policies: From Centralized Ones to Decentralized Ones”. In: *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 3.* AAMAS ’02. Bologna, Italy: Association for Computing Machinery, 2002, pp. 1098–1105. ISBN: 1581134800. DOI: `10.1145/545056.545078`. URL: `https://doi.org/10.1145/545056.545078`.

[40] Chao Yu et al. “The Surprising Effectiveness of MAPPO in Cooperative, Multi-Agent Games”. In: *CoRR* abs/2103.01955 (2021). arXiv: `2103.01955`. URL: `https://arxiv.org/abs/2103.01955`.