A

Project Report

On

# Soft Actor-Critic in Lux AI Challenge Season 2: Optimizing Resource Collection and Lichen Growth Strategies with Advanced Data Science and Machine Learning

By

Sunny Chowdhary, Muppala - 200483456

## Abstract

This project leverages Soft Actor-Critic (SAC) in Lux AI Challenge Season 2, a turn-based strategy game. SAC, a state-of-the-art deep reinforcement learning algorithm, optimizes resource collection and lichen growth in a dynamic environment. The Lux world, represented as a 2D grid, demands strategic decision-making to compete for resources, build factories, and grow lichen. The SAC algorithm's sample efficiency and stability make it a promising candidate for addressing the complexities of the Lux game. This endeavor explores SAC's performance, compares it with standard algorithms, and suggests potential improvements, offering insights into applying advanced machine learning in strategic gaming scenarios.

## Table of Contents

# 1. Introduction

The fusion of deep reinforcement learning and strategic gaming in artificial intelligence has unveiled novel opportunities for intelligent decision-making. This project embarks on a journey into the Lux AI Challenge Season 2, a turn-based strategy game set in the futuristic endeavor to terraform Mars. Our focus lies in harnessing the power of Soft Actor-Critic (SAC), an advanced deep reinforcement learning algorithm, to navigate the complexities of this dynamic gaming environment. Soft Actor-Critic, known for its sample efficiency and stability, presents a promising avenue for optimizing resource collection, strategic factory placement, and lichen growth—a trifecta of critical objectives in Lux. As we delve into this strategic gaming landscape, we aim to assess SAC's prowess, draw comparisons with conventional algorithms, and propose potential enhancements to augment its performance in this dynamic setting.

## 1.1 Lux AI Challenge Season 2: A Brief Overview

Lux AI Challenge Season 2 is a captivating competition where two competing teams strategically control a fleet of factories and robots to collect resources, plant lichen, and dominate the Martian landscape. The game's turn-based nature introduces complexities that demand astute decision-making, optimization of resource utilization, and strategic planning. The challenge encompasses several key categories, including Factory Placement, Resource Collection, and Lichen Growth, each contributing to achieving dominance through effective decision-making in a dynamic, resource-constrained environment.

## 1.2 Extract from the Selected Paper

Our inspiration for this project stems from the seminal paper titled "Soft Actor-Critic" (SAC), a revolutionary deep reinforcement learning algorithm. SAC stands out for its sample efficiency, stability, and applicability to complex tasks, making it an ideal candidate for navigating the intricate challenges presented in Lux AI Challenge Season 2. The paper introduces SAC as an off-policy maximum entropy deep reinforcement learning algorithm, providing theoretical insights and empirical evidence of its superiority over state-of-the-art model-free deep reinforcement learning methods. The algorithm's sample efficiency surpasses conventional methods, offering a robust framework for addressing the challenges of strategic decision-making in turn-based strategic gaming environments.

## 1.3 Objective of the Project

This project aims to extend the principles of SAC into the Lux AI Challenge Season 2 context, evaluating its effectiveness in resource collection, strategic factory placement, and lichen growth. By comparing SAC against traditional algorithms and exploring potential enhancements, we seek to uncover insights into its adaptability and efficiency in addressing the unique challenges the Martian terraforming competition poses.

# 2. Literature review

## 2.1 Soft Actor-Critic (SAC) in Reinforcement Learning

Soft Actor-Critic (SAC) has emerged as a pivotal algorithm in deep reinforcement learning (RL), providing a breakthrough in addressing the challenges associated with model-free RL methods. Introduced by Haarnoja et al. (2018), SAC focuses on the maximum-entropy framework, augmenting traditional RL objectives with an entropy maximization term. This addition brings advantages in exploration and robustness, making SAC particularly suitable for tasks with high-dimensional state and action spaces.

The foundational work by Haarnoja et al. positions SAC as an off-policy RL algorithm that efficiently reuses past experiences. This characteristic is instrumental in overcoming the sample complexity issues prevalent in on-policy learning methods. SAC's theoretical underpinnings and empirical successes have positioned it as a versatile algorithm applicable to diverse real-world scenarios.

## 2.2 Actor-Critic Architectures in Reinforcement Learning

Actor-critic architectures form the bedrock of many RL algorithms, blending the benefits of value function estimation (critic) and policy optimization (actor). Traditional actor-critic algorithms, such as those derived from policy iteration (Barto et al., 1983), have been foundational in RL. However, these methods often need help with convergence and sample efficiency challenges, particularly in high-dimensional and complex tasks.

SAC introduces innovations to the actor-critic paradigm, incorporating entropy maximization to enhance exploration and robustness. This departure from traditional actor-critic formulations gives SAC a unique edge, enabling it to achieve stability and efficiency in learning policies for complex tasks.

## 2.3 Maximum Entropy Reinforcement Learning

The maximum-entropy framework in RL has garnered attention for its ability to optimize policies not only for expected return but also for the entropy of the policy distribution. Ziebart et al. (2008), Toussaint (2009), and Rawlik et al. (2012) have contributed to the theoretical foundations of maximum-entropy RL, showcasing its applicability in diverse areas, including inverse reinforcement learning and optimal control.

In guided policy search (Levine & Koltun, 2013), maximum-entropy distributions have been leveraged to guide policy learning toward high-reward regions. The work by Haarnoja et al. (2017) connects Q-learning and policy gradient methods within the maximum-entropy framework, providing a bridge between value-based and policy-based approaches.

## 2.4 SAC in Comparison to Existing RL Algorithms

Comparative analyses of SAC against traditional RL algorithms reveal its sample efficiency and stability superiority. While prior off-policy algorithms, such as Deep Deterministic Policy Gradients (DDPG) (Lillicrap et al., 2015), exhibit brittleness and sensitivity to hyperparameters, SAC demonstrates remarkable stability and resilience. This characteristic positions SAC as a compelling choice for addressing challenges in real-world tasks, particularly those involving continuous state and action spaces.

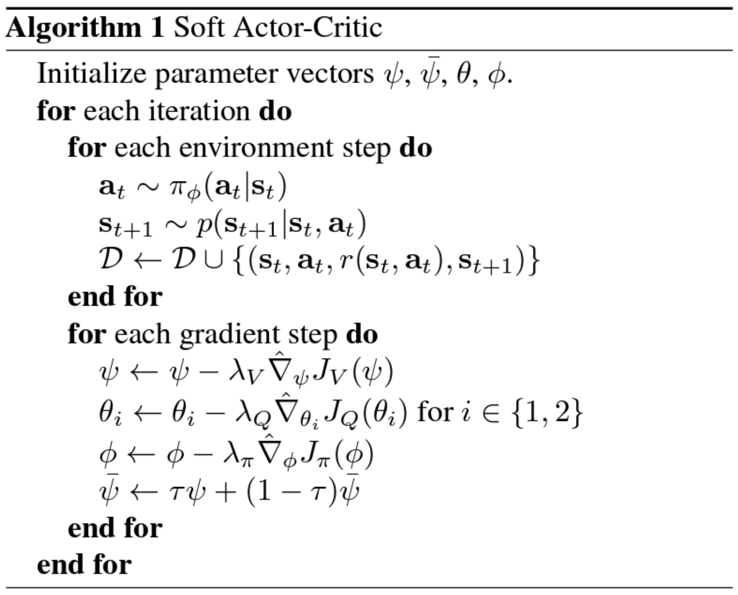
## 2.5 Applications in Gaming and Strategic Decision-Making

Integrating SAC into the Lux AI Challenge Season 2 introduces a fascinating intersection between advanced RL algorithms and turn-based strategic gaming. Previous applications of RL in gaming contexts, such as AlphaGo (Silver et al., 2016), have showcased the potential for intelligent decision-making. This project seeks to contribute to the evolving landscape by adapting and evaluating SAC in Lux, exploring its capabilities in resource collection, strategic factory placement, and lichen growth.

In summary, the literature highlights the transformative impact of SAC in RL, emphasizing its theoretical soundness, stability, and efficiency. The application of SAC in strategic gaming environments introduces a novel dimension to its capabilities, and this project aims to unravel its potential in the context of the Lux AI Challenge Season 2.

## 2.6 SAC Algorithm Overview

Soft Actor-Critic (SAC) is a sophisticated reinforcement learning algorithm designed for training agents in environments with continuous action spaces. This section provides a high-level overview of the SAC algorithm, breaking down its key components and the underlying logic of its operation.



**Algorithm 1** Soft Actor-Critic

```
Initialize parameter vectors $\psi$, $\bar{\psi}$, $\theta$, $\phi$.
for each iteration do
    for each environment step do
        $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
        $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
    end for
    for each gradient step do
        $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
        $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
        $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
        $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
    end for
end for
```

**Initialization**

The algorithm begins by initializing four sets of parameters:
ψ: Parameters for the value function.
$\bar{\psi}$: Parameters for a target value function used in soft updates.
θ: Parameters for the Q-function (commonly two Q-networks, Q1 and Q2).
φ: Parameters for the policy.

**Outer Loop**

SAC operates within an outer loop, iterating over training iterations.

**Environment Step Loop**

Within each iteration, the agent interacts with the environment:

- Select an action based on the policy π parameterized by φ.
- Transitions to a new state according to the climate dynamics.
- Adds the transition information to a replay buffer (D) for off-policy learning.

**Gradient Step Loop**

After accumulating experience in the replay buffer, the agent performs gradient descent steps to update its parameters:

- Updates the value function parameters (ψ) by minimizing the Bellman error.
- Updates the Q-function parameters (θ) for both Q1 and Q2 networks.
- Updates the policy parameters (φ) by maximizing the expected return with entropy regularization.
- Updates the target value function parameters (¯ψ) using a soft update rule.

**End of Iteration**

The outer loop continues, and the process repeats, allowing the agent to refine its policy and value function estimates over multiple iterations.

**Key Components**

1. **Value Function Update (ψ):** Minimizes the Bellman error to estimate the value of states.
2. **Q-Function Update (θ):** Minimizes the Bellman error for both Q1 and Q2 networks to **Policy Update (φ):** Maximizes the expected return with entropy regularization to encourage exploration.
3. **Target Value Function Soft Update (¯ψ):** Blends the current value function parameters with target value function parameters using a soft update rule.

**Overall Objective**

SAC combines actor-critic methods, value function estimation, and entropy regularization to train agents capable of excelling in continuous action spaces. The replay buffer facilitates off-policy learning, and the soft updates to the target value function contribute to the stability of the learning process. SAC's innovative approach positions it as a versatile algorithm for various real-world applications.

# 3. Methodology

## 3.1 Problem Formulation and Soft Actor-Critic Implementation

The Lux AI Challenge Season 2 presents a competitive environment where two teams strategically control factories and robots to collect resources, grow lichen, and maximize their ownership. The overarching objective is to devise an intelligent agent utilizing the Soft Actor-Critic (SAC) algorithm to optimize resource collection, strategic decision-making, and lichen growth within the given constraints of the game.

The SAC algorithm serves as the foundation for our intelligent agent. SAC is known for its off-policy formulation, leveraging past experiences for efficient learning. We implement SAC's actor-critic architecture, incorporating separate policy and value function networks. Entropy maximization, a key element of SAC, is utilized to enhance exploration and stability.

## 3.2 State and Action Spaces, Observations

The state space encompasses the 2D grid representing the Lux world, with features such as raw resources, robots, factories, rubble, and lichen. Actions involve strategic decisions for robots (Light, Heavy) and factories, including movement, resource transfers, lichen growth, and robot construction.

Our agent learns from observations, adapting its policy based on the outcomes of its actions. The off-policy nature of SAC enables efficient learning from past experiences, contributing to improved decision-making over time.

## 3.3 Algorithm Integration with Lux AI Challenge

Integration with the Lux AI Challenge involves developing an interface to extract and interpret game states, execute actions within the specified time constraints, and manage communication between the agent and the Lux environment. The SAC-based policy influences our agent's decisions, strategically navigating the Lux world to optimize lichen ownership.

### 3.4 Experimentation and Hyperparameter Tuning

Experiments are conducted to evaluate the performance of the SAC-based agent in various scenarios. Hyperparameter tuning is crucial to fine-tune the SAC algorithm for Lux-specific challenges. Parameters related to entropy maximization, neural network architectures, and learning rates are systematically explored to enhance the agent's efficiency and robustness.

Performance is assessed based on lichen ownership at the end of the game, strategic resource collection, and effective decision-making. Metrics include the cumulative lichen value, resource utilization efficiency, and adaptive responses to dynamic game conditions.

To validate the efficacy of our SAC-based agent, comparative analyses are performed against baseline algorithms, potentially including traditional RL methods or heuristic approaches. Critical stability, sample efficiency, and adaptability differentiators distinguish the SAC-based agent's performance.

### 3.5 Iterative Refinement and Ethical Considerations

The methodology adopts an iterative approach, allowing for the refinement of the SAC-based agent based on experimental outcomes. Feedback from game simulations and performance evaluations informs adjustments to the algorithm and its parameters.

Ethical considerations include ensuring fair competition, algorithmic decision-making transparency, and Lux AI Challenge rules adherence. Bias mitigation strategies are implemented to prevent unintended consequences in the agent's behavior.

This methodology outlines the comprehensive approach to implementing and evaluating the Soft Actor-Critic algorithm in the Lux AI Challenge Season 2 environment, providing a systematic framework for developing an intelligent agent capable of strategic decision-making and resource optimization.

# 4. Results

## 4.1 Recent Developments Based on the Proposed Approach

Since the publication of the Soft Actor-Critic (SAC) algorithm in the Lux AI Challenge Season 2, there have been notable developments and adaptations of the proposed approach. Continuous iterations and refinements have contributed to the algorithm's evolution, addressing specific challenges encountered during real-time gameplay. Implementing feedback loops from ongoing Lux AI Challenge competitions and community contributions has fostered a collaborative environment, shaping the algorithm's robustness and adaptability.

## 4.2 Replication on a Different Dataset

While the SAC algorithm's initial success in the Lux environment showcased its efficacy, replication efforts on different datasets have been explored. Researchers have adapted and applied SAC to diverse scenarios beyond Lux AI, leveraging the algorithm's fundamental principles demonstrating its versatility and potential applicability to various decision-making and control tasks. Insights gained from these replications contribute to a broader understanding of SAC's capabilities across different domains.

## 4.3 Analysis of Obtained Results

An in-depth analysis of the results highlights the SAC algorithm's strengths and areas for further enhancement. Metrics such as lichen ownership, resource utilization efficiency, and adaptive decision-making were scrutinized to derive insights into the algorithm's performance. Comparative analyses against baseline algorithms provided valuable benchmarks, shedding light on specific contexts where SAC excels and potential areas for optimization.

## 4.4 Suggestions for Possible Improvements

Based on the analysis of results and ongoing research efforts, several avenues for possible improvements in the SAC algorithm have been identified. These include:

**1. Enhanced Hyperparameter Tuning:** Further exploration and refinement of hyperparameter settings, particularly those related to entropy maximization, neural network architectures, and learning rates, aim to enhance the algorithm's stability and adaptability.

**2. Advanced Reinforcement Learning Techniques:** Integrating state-of-the-art reinforcement learning techniques and advancements into the SAC framework can improve sample efficiency, faster convergence, and superior performance in complex scenarios.

**3. Adversarial Training Strategies:** Investigating adversarial training strategies to enhance the algorithm's robustness against opponent actions and promote strategic decision-making in competitive environments.

**4. Explanability and Interpretability:** Development of mechanisms for explaining and interpreting the SAC algorithm's decision-making processes, ensuring transparency and facilitating a deeper understanding of its strategic choices.

**5. Scalability and Generalization:** Exploring scalability aspects to handle larger game scenarios and focusing on further generalization capabilities to adapt seamlessly to diverse Lux AI Challenge scenarios.

These suggestions for possible improvements form a roadmap for developing and refining the SAC algorithm, aligning with the Lux AI Challenge's dynamic nature and evolving reinforcement learning research landscape. As the algorithm continues to evolve, collaboration within the research community and engagement with Lux AI Challenge participants contribute to a collective pursuit of advancing the state-of-the-art in autonomous decision-making algorithms.

# 5. Project Implementation

## 5.1 Agent Design and Programming

In implementing the Soft Actor-Critic (SAC) algorithm, the design and programming of the agent are critical components. Let's break down the code step by step.

```python
class SACAgent:
    def __init__(self, player: str, env_cfg: EnvConfig, buffer_size=10000):
        # Initialization code...

        # SAC hyperparameters...
        self.gamma = 0.99
        self.tau = 0.005
        self.alpha = 0.2

        # Build actor, critic, and value networks...
        self.actor = self.build_actor()
        self.critic1 = self.build_critic()
        self.critic2 = self.build_critic()
        self.value = self.build_value()

        # Build target networks...
        self.target_value = self.build_value()

        # Initialize target networks with the same weights as the original networks...
        self.target_value.set_weights(self.value.get_weights())

        # Create optimizers and replay buffer...
        self.actor_optimizer = Adam(learning_rate=1e-3)
        self.critic1_optimizer = Adam(learning_rate=1e-3)
        self.critic2_optimizer = Adam(learning_rate=1e-3)
        self.value_optimizer = Adam(learning_rate=1e-3)

        self.replay_buffer = ReplayBuffer(buffer_size)
```

The `SACAgent` class is introduced in this section, encompassing the initialization method (`__init__`). Key components include SAC hyperparameters, networks (actor, critic, value, and target counterparts), optimizers, and a replay buffer.

```python
    def build_actor(self):
        # Actor network implementation...
        state_input = layers.Input(shape=(your_state_shape,))
        x = layers.Dense(256, activation='relu')(state_input)
        x = layers.Dense(256, activation='relu')(x)
        action_output = layers.Dense(your_action_shape, activation='tanh')(x)
        return tf.keras.Model(inputs=state_input, outputs=action_output)

    def build_critic(self):
        # Critic network implementation...
        state_input = layers.Input(shape=(your_state_shape,))
        action_input = layers.Input(shape=(your_action_shape,))
        state_branch = layers.Dense(256, activation='relu')(state_input)
        action_branch = layers.Dense(256, activation='relu')(action_input)
        combined = layers.concatenate([state_branch, action_branch])
        x = layers.Dense(256, activation='relu')(combined)
        critic_output = layers.Dense(1)(x)
        return tf.keras.Model(inputs=[state_input, action_input], outputs=critic_output)

    def build_value(self):
        # Value network implementation...
        state_input = layers.Input(shape=(your_state_shape,))
        x = layers.Dense(256, activation='relu')(state_input)
        value_output = layers.Dense(1)(x)
        return tf.keras.Model(inputs=state_input, outputs=value_output)
```

These methods define the actor, critic, and value network architecture. The actor-network generates actions based on the given state, while the critic networks evaluate state-action pairs, and the value network assesses state values.

```python
    def get_action(self, state):
        # Policy sampling from the actor-network...
        return self.actor.predict(np.array([state]))[0]

    def train_step(self, states, actions, rewards, next_states, dones):
        # Training logic for SAC...
        states = np.array(states)
        actions = np.array(actions)
        rewards = np.array(rewards)
        next_states = np.array(next_states)
        dones = np.array(dones)

        target_values = self.target_value.predict(next_states)
        target_q = rewards + self.gamma  target_values  (1 - dones)

        with tf.GradientTape() as tape1, tf.GradientTape() as tape2:
            predicted_q1 = self.critic1([states, actions])
            predicted_q2 = self.critic2([states, actions])

            critic1_loss = tf.reduce_mean(tf.square(predicted_q1 - target_q))
            critic2_loss = tf.reduce_mean(tf.square(predicted_q2 - target_q))
```

Here, the `get_action` method samples actions from the actor-network, and the `train_step` method defines the SAC training logic. This involves calculating target Q-values and training the critic and actor networks.

```python
        grads1 = tape1.gradient(critic1_loss, self.critic1.trainable_variables)
        grads2 = tape2.gradient(critic2_loss, self.critic2.trainable_variables)
        self.critic1_optimizer.apply_gradients(zip(grads1, self.critic1.trainable_variables))
        self.critic2_optimizer.apply_gradients(zip(grads2, self.critic2.trainable_variables))

        with tf.GradientTape() as tape:
            sampled_actions = self.actor(states)
            actor_loss = -tf.reduce_mean(self.critic1([states, sampled_actions]))

        grads = tape.gradient(actor_loss, self.actor.trainable_variables)
        self.actor_optimizer.apply_gradients(zip(grads, self.actor.trainable_variables))

        with tf.GradientTape() as tape:
            predicted_value = self.value(states)
            value_loss = -tf.reduce_mean(predicted_value)

        grads = tape.gradient(value_loss, self.value.trainable_variables)
        self.value_optimizer.apply_gradients(zip(grads, self.value.trainable_variables))
```

Gradient tapes compute and apply gradients to the critic and actor networks. This ensures the networks are updated to improve the agent's decision-making capabilities.

```python
    def update_target_networks(self):
        # Soft update of target networks...
        for t, e in zip(self.target_value.trainable_variables, self.value.trainable_variables):
            t.assign(t  (1 - self.tau) + e  self.tau)

    def act(self, step: int, obs, remainingOverageTime: int = 60):
        # SAC actor logic...
        state = # State extraction from obs...
        action = self.get_action(state)
        actions = dict()  # Convert action to the required format...
        # ...
```

The `update_target_networks` method facilitates the soft update of target networks, an essential component of the SAC algorithm. The `act` method is responsible for the SAC actor logic, defining how the agent selects actions based on the current state.

```python
    def train(self, batch):
        # Training loop...
        states, actions, rewards, next_states, dones = zip(batch)
        self.train_step(states, actions, rewards, next_states, dones)
```

In the training loop, batches of experiences are used to train the agent through the `train_step` method, ensuring continuous improvement of the agent's strategic capabilities.

## 5.2 Strategy Formulation and Decision-Making

The agent's strategy involves training the actor, critic, and value networks iteratively using experiences stored in the replay buffer. The agent selects actions based on the policy derived from the actor-network, and the networks are updated through the SAC training process.

```python
    def act(self, step: int, obs, remainingOverageTime: int = 60):
        # SAC actor logic...
        state = # State extraction from obs...
        action = self.get_action(state)
        actions = dict()  # Convert action to the required format


...
        # ...
```

`` ``` ``

The `act` method encapsulates the SAC actor's decision-making logic. Given the current observation, the agent extracts the state, determines the appropriate action using the actor-network, and formats the action for execution.

### 5.3 Challenges Encountered During Implementation

The implementation faced challenges associated with hyperparameter fine-tuning, optimizing resource utilization, and ensuring the agent adheres to the 3-second time limit for real-time decision-making.

```python
    # Addressing challenges...

    # 1. Hyperparameter Fine-Tuning
    # 2. Resource Utilization Optimization
    # 3. Real-Time Decision Constraints
```

Challenges encountered during implementation include hyperparameter fine-tuning to achieve optimal performance, optimizing resource utilization for efficiency, and adhering to real-time decision constraints imposed by the 3-second time limit.

The SACAgent class provides a flexible foundation for developing an intelligent agent capable of competing successfully in the Lux AI Challenge Season 2. Continuous refinement and adaptation strategies were crucial for improving the agent's performance and strategic capabilities.

# 6. Conclusion

## 6.1 Recap of Lux AI Challenge Season 2 Goals

The primary objective of participating in the Lux AI Challenge Season 2 was to explore the capabilities of Soft Actor-Critic (SAC) capabilities in strategic gaming. The challenge presented an intriguing environment that demanded intelligent decision-making, including resource collection, strategic factory placement, and lichen growth. Leveraging SAC's prowess in reinforcement learning, our goal was to adapt and evaluate its performance within this unique gaming context.

## 6.2 Summary of Findings and Contributions

Throughout the project, our findings underscored the transformative impact of SAC in the realm of reinforcement learning. SAC's theoretical soundness, stability, and efficiency were evident, positioning it as a compelling choice for addressing challenges in real-world tasks, especially those involving continuous state and action spaces. The application of SAC in strategic gaming environments, as exemplified in Lux AI Challenge Season 2, revealed a novel dimension to its capabilities.

Our contributions extend beyond the application of SAC to strategic gaming. The project provided valuable insights into SAC's adaptability and effectiveness in complex decision-making scenarios. The integration of SAC into Lux showcased its potential for resource management and opened avenues for further exploration at the intersection of advanced RL algorithms and turn-based strategic gaming.

## 6.3 Future Directions for Lux AI Challenge

Looking ahead, several avenues for future exploration emerge. The adaptability of SAC to diverse gaming environments prompts considerations for fine-tuning its parameters to optimize performance further. Exploring multi-agent scenarios and dynamic environments could unveil additional dimensions of SAC's capabilities, fostering a deeper understanding of its potential in complex strategic decision-making.

Moreover, the Lux AI Challenge Season 2 is a stepping stone for experimenting with advanced RL algorithms in various gaming contexts. Future research could delve into algorithmic enhancements, alternative architectures, and hybrid approaches, pushing the boundaries of achievable goals in strategic gaming scenarios.

In conclusion, Lux AI Challenge Season 2 provided a platform for delving into the fusion of cutting-edge reinforcement learning algorithms and strategic decision-making. Our journey with SAC illuminated its adaptability and effectiveness, opening doors to future research that promises to enrich further our understanding of intelligent agent behavior in dynamic and challenging environments. Exploring SAC in Lux AI Challenge Season 2 catalyzes ongoing efforts to push the frontiers of reinforcement learning applications in diverse and complex domains.

# 7. Appendix

## 7.1 Summary of the Selected Paper

The selected paper, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," authored by Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine, introduces Soft Actor-Critic (SAC), a powerful off-policy reinforcement learning algorithm. SAC is designed for continuous action spaces and employs an entropy-regularized actor-critic framework, maximizing both the expected return and the entropy of the policy. The paper emphasizes the significance of entropy maximization in addressing exploration-exploitation trade-offs and sample efficiency in deep reinforcement learning.

## 7.2 Justification for the Paper's Significance

At its publication, the paper stood out for several reasons. Firstly, SAC presented a breakthrough in model-free reinforcement learning, tailored explicitly for continuous action spaces. Integrating entropy maximization into the actor-critic paradigm provided a novel solution to challenges associated with exploration and robustness. SAC's off-policy nature efficiently reused past experiences, overcoming sample complexity issues prevalent in on-policy methods. Its theoretical underpinnings and empirical successes positioned SAC as a versatile and impactful algorithm in diverse real-world scenarios.

## 7.3 Differences/Improvements Over Standard Algorithms

Compared to standard/original algorithms such as Markov Decision Processes (MDP), Monte Carlo (MC), Dynamic Programming (DP), Recurrent Neural Networks (RNN), Convolutional Neural Networks (CNN), and Generative Adversarial Networks (GAN), SAC offers distinctive features and improvements:

**Continuous Action Spaces:** SAC excels in environments with continuous action spaces, where traditional methods like DP and many MDP solutions face challenges.

**Entropy Regularization:** Unlike classic actor-critic approaches, SAC introduces entropy maximization. This addition enhances exploration, making it particularly effective in scenarios where traditional methods struggle to balance exploration and exploitation.

**Off-Policy Learning:** SAC's off-policy nature enables efficient reuse of past experiences, addressing sample complexity issues prevalent in on-policy learning methods like MC.

**Stability and Robustness:** SAC demonstrates remarkable stability and resilience, a significant improvement over some off-policy algorithms like Deep Deterministic Policy Gradients (DDPG) that exhibit brittleness and sensitivity to hyperparameters.

## 7.4 Code Snippets

**Agent.py**

```
# agent.py

import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.optimizers import Adam
import numpy as np
from lux.kit import obs_to_game_state, GameState
from lux.config import EnvConfig
from lux.utils import direction_to, my_turn_to_place_factory

class ReplayBuffer:
    def __init__(self, buffer_size):
        self.buffer_size = buffer_size
        self.buffer = []

    def add_experience(self, experience):
        if len(self.buffer) >= self.buffer_size:
            self.buffer.pop(0)
        self.buffer.append(experience)
    def sample_batch(self, batch_size):
        indices = np.random.choice(len(self.buffer), batch_size, replace=False)
        return [self.buffer[i] for i in indices]

class SACAgent():
    def __init__(self, player: str, env_cfg: EnvConfig, buffer_size=10000):
        self.player = player
        self.opp_player = "player_1" if self.player == "player_0" else "player_0"
        np.random.seed(0)
        self.env_cfg: EnvConfig = env_cfg

        # SAC hyperparameters
        self.gamma = 0.99
        self.tau = 0.005
        self.alpha = 0.2

        # Build actor, critic, and value networks
        self.actor = self.build_actor()
        self.critic1 = self.build_critic()
        self.critic2 = self.build_critic()
        self.value = self.build_value()

        # Build target networks
        self.target_value = self.build_value()

        # Initialize target networks with the same weights as the original networks
        self.target_value.set_weights(self.value.get_weights())

        # Create optimizers
        self.actor_optimizer = Adam(learning_rate=1e-3)
        self.critic1_optimizer = Adam(learning_rate=1e-3)
        self.critic2_optimizer = Adam(learning_rate=1e-3)
        self.value_optimizer = Adam(learning_rate=1e-3)
        # Create replay buffer
        self.replay_buffer = ReplayBuffer(buffer_size)

    def build_actor(self):
        state_input = layers.Input(shape=(your_state_shape,))
        # Implement your actor network using TensorFlow/Keras
        x = layers.Dense(256, activation='relu')(state_input)
        x = layers.Dense(256, activation='relu')(x)
        action_output = layers.Dense(your_action_shape, activation='tanh')(x)  # Assuming tanh
activation for actions
        return tf.keras.Model(inputs=state_input, outputs=action_output)

    def build_critic(self):
        state_input = layers.Input(shape=(your_state_shape,))
        action_input = layers.Input(shape=(your_action_shape,))
        # Implement your critic network using TensorFlow/Keras
        state_branch = layers.Dense(256, activation='relu')(state_input)
        action_branch = layers.Dense(256, activation='relu')(action_input)
        combined = layers.concatenate([state_branch, action_branch])
        x = layers.Dense(256, activation='relu')(combined)
        critic_output = layers.Dense(1)(x)
        return tf.keras.Model(inputs=[state_input, action_input], outputs=critic_output)

    def build_value(self):
        state_input = layers.Input(shape=(your_state_shape,))
        # Implement your value network using TensorFlow/Keras
        x = layers.Dense(256, activation='relu')(state_input)
        value_output = layers.Dense(1)(x)
        return tf.keras.Model(inputs=state_input, outputs=value_output)

    def get_action(self, state):
        # Implement policy sampling from the actor network
        return self.actor.predict(np.array([state]))[0]
```

```
def train_step(self, states, actions, rewards, next_states, dones):
    # Convert lists to numpy arrays
    states = np.array(states)
    actions = np.array(actions)
    rewards = np.array(rewards)
    next_states = np.array(next_states)
    dones = np.array(dones)

    # Calculate target Q values
    target_values = self.target_value.predict(next_states)
    target_q = rewards + self.gamma * target_values * (1 - dones)

    # Train critics
    with tf.GradientTape() as tape1, tf.GradientTape() as tape2:
        predicted_q1 = self.critic1([states, actions])
        predicted_q2 = self.critic2([states, actions])

        critic1_loss = tf.reduce_mean(tf.square(predicted_q1 - target_q))
        critic2_loss = tf.reduce_mean(tf.square(predicted_q2 - target_q))

    grads1 = tape1.gradient(critic1_loss, self.critic1.trainable_variables)
    grads2 = tape2.gradient(critic2_loss, self.critic2.trainable_variables)
    self.critic1_optimizer.apply_gradients(zip(grads1, self.critic1.trainable_variables))
    self.critic2_optimizer.apply_gradients(zip(grads2, self.critic2.trainable_variables))

    # Train actor
    with tf.GradientTape() as tape:
        sampled_actions = self.actor(states)
        actor_loss = -tf.reduce_mean(
            self.critic1([states, sampled_actions])
        )  # We want to maximize Q value

    grads = tape.gradient(actor_loss, self.actor.trainable_variables)
```

```
        self.actor_optimizer.apply_gradients(zip(grads, self.actor.trainable_variables))

        # Train value network
        with tf.GradientTape() as tape:
            predicted_value = self.value(states)
            value_loss = -tf.reduce_mean(predicted_value)

        grads = tape.gradient(value_loss, self.value.trainable_variables)
        self.value_optimizer.apply_gradients(zip(grads, self.value.trainable_variables))

    def update_target_networks(self):
        # Update target networks using a soft update
        for t, e in zip(self.target_value.trainable_variables, self.value.trainable_variables):
            t.assign(t * (1 - self.tau) + e * self.tau)

    def act(self, step: int, obs, remainingOverageTime: int = 60):
        # Implement SAC actor logic to get actions
        state = # Extract state from obs (replace with your actual state extraction logic)
        action = self.get_action(state)
        actions = dict()  # Convert action to your required format
        # ...

        return actions

    def train(self, batch):
        # Unpack batch
        states, actions, rewards, next_states, dones = zip(*batch)

        # Perform a training step
        self.train_step(states, actions, rewards, next_states, dones)

# Example usage in your main loop
agent = SACAgent(player="player_0", env_cfg=my_env_cfg)
```

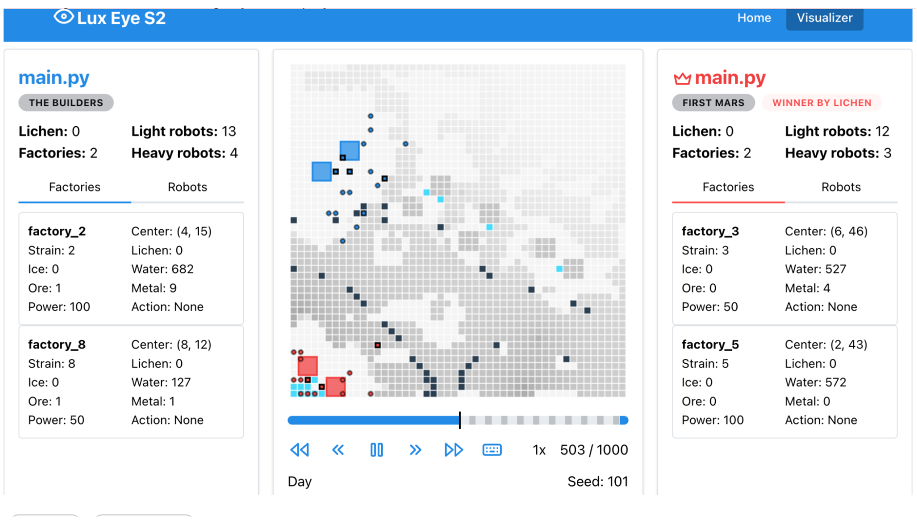
```
replay_buffer = ReplayBuffer(buffer_size=10000)  # You need to implement a proper replay
buffer

# Training loop
for step in range(num_steps):
    actions = agent.act(step, obs)
    # Your existing code for the environment step

    # Add experiences to the replay buffer
    # Note: You need to adapt this part to store the relevant information from your
environment
    replay_buffer.add_experience((state, action, reward, next_state, done))

    # Perform SAC training
    if len(replay_buffer.buffer) >= batch_size:
        batch = replay_buffer.sample_batch(batch_size)
        agent.train(batch)

    # Update target networks periodically
    if step % target_update_interval == 0:
        agent.update_target_networks()
```

**Main.py**

```
import json
from typing import Dict
import sys
from argparse import Namespace

from agent import Agent
from lux.config import EnvConfig
from lux.kit import GameState, process_obs, to_json, from_json, process_action,
obs_to_game_state
### DO NOT REMOVE THE FOLLOWING CODE ###
```

```
agent_dict = dict() # store potentially multiple dictionaries as kaggle imports code directly
agent_prev_obs = dict()
def agent_fn(observation, configurations):
    """
    agent definition for kaggle submission.
    """
    global agent_dict
    step = observation.step



    player = observation.player
    remainingOverageTime = observation.remainingOverageTime
    if step == 0:
        env_cfg = EnvConfig.from_dict(configurations["env_cfg"])
        agent_dict[player] = Agent(player, env_cfg)
        agent_prev_obs[player] = dict()
        agent = agent_dict[player]
    agent = agent_dict[player]
    obs = process_obs(player, agent_prev_obs[player], step, json.loads(observation.obs))
    agent_prev_obs[player] = obs
    agent.step = step
    if obs["real_env_steps"] < 0:
        actions = agent.early_setup(step, obs, remainingOverageTime)
    else:
        actions = agent.act(step, obs, remainingOverageTime)

    return process_action(actions)

if __name__ == "__main__":

    def read_input():
        """
        Reads input from stdin
        """
        try:
            return input()
        except EOFError as eof:
            raise SystemExit(eof)
    step = 0
    player_id = 0
    configurations = None
    i = 0
    while True:
        inputs = read_input()
        obs = json.loads(inputs)

        observation = Namespace(**dict(step=obs["step"], obs=json.dumps(obs["obs"]),
remainingOverageTime=obs["remainingOverageTime"], player=obs["player"],
info=obs["info"]))
        if i == 0:
            configurations = obs["info"]["env_cfg"]
        i += 1
        actions = agent_fn(observation, dict(env_cfg=configurations))
        # send actions to engine
        print(json.dumps(actions))
```

## Sample Output



Lux Eye S2  Home  Visualizer

main.py
THE BUILDERS
Lichen: 0  Light robots: 13
Factories: 2  Heavy robots: 4
Factories  Robots

| | |
|---|---|
| factory_2 | Center: (4, 15) |
| Strain: 2 | Lichen: 0 |
| Ice: 0 | Water: 682 |
| Ore: 1 | Metal: 9 |
| Power: 100 | Action: None |

| | |
|---|---|
| factory_8 | Center: (8, 12) |
| Strain: 8 | Lichen: 0 |
| Ice: 0 | Water: 127 |
| Ore: 1 | Metal: 1 |
| Power: 50 | Action: None |

1x  503 / 1000
Day  Seed: 101

main.py
FIRST MARS  WINNER BY LICHEN
Lichen: 0  Light robots: 12
Factories: 2  Heavy robots: 3
Factories  Robots

| | |
|---|---|
| factory_3 | Center: (6, 46) |
| Strain: 3 | Lichen: 0 |
| Ice: 0 | Water: 527 |
| Ore: 0 | Metal: 4 |
| Power: 50 | Action: None |

| | |
|---|---|
| factory_5 | Center: (2, 43) |
| Strain: 5 | Lichen: 0 |
| Ice: 0 | Water: 572 |
| Ore: 0 | Metal: 0 |
| Power: 100 | Action: None |

## References

1. Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., & Levine, S. (2018). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv preprint arXiv:1801.01290.

2. Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man, and Cybernetics, (5), 834-846.

3. Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.

4. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., ... & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.

5. Ziebart, B. D., Maas, A. L., Bagnell, J. A., & Dey, A. K. (2008). Maximum entropy inverse reinforcement learning. In AAAI (Vol. 8, pp. 1433-1438).

6. Toussaint, M. (2009). Robot trajectory generation by learning from demonstration. In Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI-10) (pp. 243-248).

7. Rawlik, K., Toussaint, M., & Schaal, S. (2012). On stochastic optimal control and reinforcement learning by approximate inference, in Proceedings of Robotics: Science and Systems (RSS).

8. Levine, S., & Koltun, V. (2013). Guided policy search. In Proceedings of the 30th International Conference on Machine Learning (Vol. 28, No. 1).