# Integrating Anticipatory Classifier Systems with OpenAI Gym

Norbert Kozlowski
Wroclaw University of Science and Technology
Department of Computer Engineering
Wroclaw, Poland
norbert.kozlowski@pwr.edu.pl

Olgierd Unold
Wroclaw University of Science and Technology
Department of Computer Engineering
Wroclaw, Poland
olgierd.unold@pwr.edu.pl

## ABSTRACT

This paper explains the process of integrating ACS2 algorithm with the standardised framework for comparing reinforcement learning tasks - *OpenAI Gym*. The new Python library is introduced alongside with standard environments derived from LCS literature. Typical use cases enabling quick evaluation of different research problems are described.

## CCS CONCEPTS

• **Computing methodologies** → *Rule learning*; • **Software and its engineering**;

## KEYWORDS

Aniticipatory Learning Classifier Systems, OpenAI Gym

**ACM Reference Format:**
Norbert Kozlowski and Olgierd Unold. 2018. Integrating Anticipatory Classifier Systems with OpenAI Gym. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan,* xxx, yyy, and zzz (Eds.). ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205651.3208241

## 1 INTRODUCTION

This paper describes the process of integrating Anticipatory Classifier Systems, which is a learning algorithm based on *Learning Classifier Sytems (LCS)* and psychological learning mechanism of *"Anticipatory Behavioral Control"* with *OpenAI Gym* interface, which is a toolkit used for comparing and evaluating reinforcement learning algorithms. Such integration makes it possible to create and reuse previously declared environments (employing a unified interface) and enables conducting modern reproducible research in LCS realm.

Section 2 describes the *OpenAI* project and its *Gym* framework. Part 3 provides a quick recap of the ACS2 algorithm. Section 4 summarises environments in which ACS2 can operate. Two of them (Maze and Boolean Multiplexer) were created from scratch, and the other two (Go game and FrozenLake) are existing *OpenAI Gym* implementation. Section 5 describes the process of re-implementing ACS2 in Python language (new *PyALCS* open-source package), that is later applied in following chapter 6. Some universal integration use-cases are presented followed by experiments results in section 7. Finally, section 8 summarises the paper and offers some ideas for future work.

## 2 OPENAI

OpenAI[1] is non-profit AI research company with a mission to build safe AGI (*artificial general intelligence*). It aims to collaborate with other institutions and researchers by making its patents and research open to the public.

It was founded in 2015 by Elon Musk and Sam Altman with the intention of mitigating the risk of possible *"intelligence explosion"*, that is believed to happen someday with advanced AI.

### 2.1 OpenAI Gym

*OpenAI Gym* is a toolkit for creating and comparing reinforcement learning algorithms [1]. It provides an open-source interface developed in Python (more languages will be available soon) for defining environments the agents (term used to describe algorithms) will be interacting with.

There are already a vast amount of predefined environments ranging from algorithmic problems, board games to 3D simulations of games such as Doom. The literature describes them as partially observable Markov decision processes (POMDP), which are general enough to model real-world sequential decision processes.

Creation of own environment was also simplified - each one must initialise specific properties and implement required interface methods. Most importantly it should declare the *action space* (agent's possible moves) and the *observation space* (agent's perception in given state) that can be either discrete or continuous.

The agent interacts with the environment using two primary operations - observing current state (quantified information about perceived surroundings) and taking action. Taking action executes given movement inside environment, returning to a user information about new state, reward obtained, information whether the episode was finished, and optionally some debug data.

Algorithms operating within *OpenAI Gym* are trained episodically, which means that their experience is broken down into a series of episodes. In each one agent's state is reinitialised, and the interaction with the environment proceeds until it reaches a terminal state. The overall goal is to maximise the expectation of total reward per episode and to achieve a high level of performance in as few episodes as possible.

Authors of the library paid particular attention to environment versioning. To assure reproducibility and enable evolution or modification of environment a specific version number accompanies each one's name.

[1]https://openai.com/

Alongside the software library, there is a website[2] where users can find scoreboards for each environment, showcasing results obtained by other users and their agent implementations.

## 3 ACS2

In 1993 Hoffman formulated a learning mechanism with an underlying assumption that a decisive factor for purposive behaviour is the anticipation of the behaviour consequences [7]. Behaviour consequences usually depend on the situation in which the action is executed. So it is necessary to learn in which situation **S** which behaviour **R** (reaction) leads to which effects **E** (presented in Figure 1).

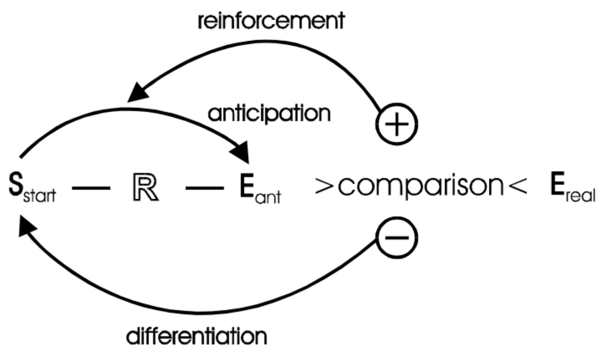

**Figure 1: Anticipatory Behavioral Control. Figure adapted from [11, p. 2].**

In 1997 Stolzmann combined the idea of LCS and Hoffman's theory introducing a new class of systems - ALCS (Anticipatory Learning Classifier Systems). Its first implementation - ACS (Anticipatory Learning Classifier) [11] used the new part of the classifier structure - *"effect"* which holds anticipations for given situation.

ACS2 [2] is very similar to the original Stolzmann's work, but there are also several essential discrepancies:

- each classifier explicitly represents anticipations,
- by better integrated and improved ALP and GA pressures ACS2 can evolve a complete, accurate, and maximally general model [4],
- classifiers updates are made in the whole action set (not just on the executed classifier).

### 3.1 Knowledge representation

In ACS2 the knowledge is represented by a population of classifiers. Each classifier represents a condition-action-effect triple that anticipates the model state after the execution of the given action in the specified conditions. Both condition and effect part consists of the discrete values perceived by the environment and # symbol, which works differently in each part. In condition part, it's called *"don't care"* and denotes that the classifier matches any value in this attribute. On the other hand in effect part, it's called as *"pass-through"* and means that the anticipation for given value will not change after the execution of the specified action.

[2]https://gym.openai.com

A classifier structure also holds other information such as mark (information about situations where given classifier was wrong), quality, reward, intermediate reward, and various counters [2, p. 123].

### 3.2 Interaction with environment

ACS2 interacts *autonomously* with an environment (Figure 2).

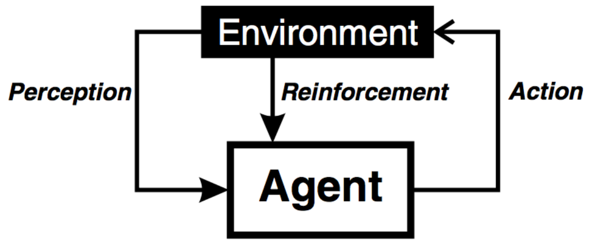

**Figure 2: Interaction between ACS and environment. Figure adapted from [2, p. 14].**

In a behavioural act at a specific time $t$, the agent perceives a situation $\sigma(t) = \{l_1, l_2, \ldots, l_m\}^L$, where:

- $m$ denotes the number of possible values of each environmental attribute (or feature),
- $l_1, l_2, \ldots, l_n$ indicate the different possible values for each attribute,
- $L$ means the string length.

Note that each attribute can only take discrete values.

The system can act upon the environment with an action $\alpha(t) = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, where:

- $n$ specifies the number of different possible actions in the environment,
- $\alpha_1, \alpha_2, \ldots, \alpha_n$ denote the various possible actions After the execution of an action, the environment provides a scalar reinforcement value $\rho(t) \in \mathbb{R}$.

### 3.3 Environmental Model

By interacting with the environment, ACS2 learns about its structure. Usually, the agent starts without any prior knowledge. Initially, new classifiers are mainly generated by a covering mechanism in ALP. Later the ALP creates specialised classifiers while the GG tries to introduce some genetic generalisation.

Figure 3 presents the interaction with greater details.

(1) After the perception of the current situation $\sigma(t)$, ACS2 forms a match set `[M]` comprising all classifiers in the population `[P]` whose conditions are satisfied in $\sigma(t)$,
(2) ACS2 chooses an action $\alpha(t)$ according to selected strategy,
(3) Concerning the selected action, an action set `[A]` is generated that consist of all classifiers in `[M]` that specify the chosen action $\alpha(t)$,
(4) After the execution of $\alpha(t)$ classifier parameters are updated by ALP and RL. New classifiers might be added or deleted due to the ALP and GG.

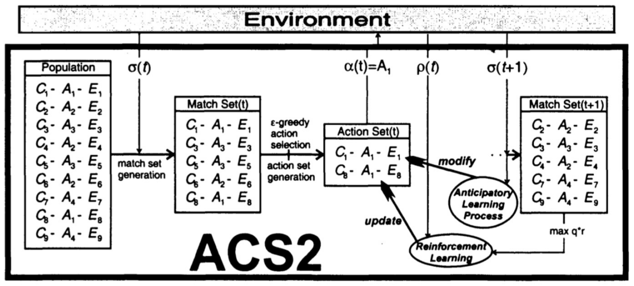

**Figure 3: A behavioral act in ACS2 with reinforcement learning and anticipatory learning process application. Figure adapted from [2, p. 27].**

*3.3.1 Anticipatory Learning Process (ALP).* The ALP was initially derived from the cognitive theory of anticipatory behavioural control [7].

It compares the anticipation of each classifier in an action set with the real next situation $\sigma(t+1)$.

The process results in the evaluation and specialisation of the anticipatory model in ACS2.

The execution of an action is accompanied by the formation of the action set, which represents the anticipations of the real next situation. Thus, ACS2 satisfies the first point of Hoffmann’s theory of anticipatory behavioural control which states that any behavioural act or response (R) is accompanied with anticipation of its effects. Moreover, the comparison of the anticipation of each classifier in `[A]` can be compared to a continuous comparison of anticipations with real next situations as stated in Hoffmann’s second point. The third and fourth point address the consequences of the comparison and are realized in the distinction between an unexpected case and an expected case.

The ALP process is responsible for generating specialised offspring and/or deleting inaccurate classifiers. Classifiers are created during expected or unexpected case or by covering mechanism [2].

*3.3.2 Genetic Generalization.* While the ALP specialises classifiers in a quite competent way, over-specialisations can sometimes occur as studied in [2]. Since various circumstances can cause the over-specialisation cases, a genetic generalisation (GG) mechanism was applied that, interacting with the ALP, results in the evolution of a complete, accurate, and maximally general model.

ACS2 figures if GG should be involved. If so the process takes place after ALP in the whole action set. Roulette wheel selection selects two classifiers (parents). Later on, mutation and crossover takes place resulting in new child classifiers, that might be inserted back to a population under certain circumstances [2].

*3.3.3 Reinforcement Learning.* ACS2 adapts the Q-learning [15] idea for reinforcing chosen actions, which is a step away from the traditional bucket brigade algorithm [8]. To learn an optimal behavioural policy in ACS2, the reward prediction $r$ of each classifier in an action set as well as the immediate reward prediction $ir$ are continuously updated. For the reliability of the maximal Q-value in the progressive state, the quality of the classifier is considered assuming that the reward converges in common with the accuracy of the anticipation [2].

## 4 ENVIRONMENTS

This section describes required steps needed to declare custom (Maze, Boolean Multiplexer)[3] and how to use already prebuilt environments (Go, FrozenLake) [4] that ACS2 (with discrete observation space) is capable of interacting with.

### 4.1 Custom Environments

Custom environments can execute any arbitrary code as requested by the developer. The only requirement is that OpenAI Gym contract needs to be met.

(1) Environment class must extend `gym.Env` class and should be initialized in `def __init__(self)` method,
(2) Method for executing action - `def _step(self, action)` must be implemented. It should evaluate given activity and return four values - current state (perception), reward obtained, information whether the episode is over and optionally debug information,
(3) Method for reinitialising the environment - `def _reset(self)` must be implemented. This method is called after each episode assuring that initial condition are provided. As a result, the current state is returned.
(4) Optionally implement a method for rendering the environment state with `def _render(self)` method. That is useful for inspecting actual behaviour.
(5) Environment class must be registered within the application using `register` function from `gym.envs.registration` package.

*4.1.1 Maze.* Maze environment is a well-known problem for inspecting multi-step capabilities of learning classifier systems. The current implementation provides a universal environmental setting where a maze is represented as a two-dimensional grid. Each field can be occupied either by an obstacle, denoted by number `"1"`, a food item denoted by `"9"` or can be empty - `"0"`.

In each episode, the animat is inserted into a random cell and perceives its immediate surroundings starting with the field to the north and coding clockwise. Thus the observation space can be specified as $\mathcal{I}_{maze} = \{0, 1, 9\}^L$, where $L = 8$, the eight adjacent cells.

It can also perform eight simple actions, the movements to the adjacent fields. $\mathcal{A}_{maze} = \{\mathrm{N}, \mathrm{NE}, \mathrm{E}, \mathrm{SE}, \mathrm{S}, \mathrm{SW}, \mathrm{W}, \mathrm{NW}\}$. When the action is not possible (i.e. leads to a position blocked by an obstacle), it has no effect.

Once the animat finds the cell with food, a constant reward $\rho = 1000$ is returned, and a current episode ends.

The implementation allows versatile maze configurations by providing own Numpy arrays in distinct classes like:

```
from gym_maze.envs import AbstractMaze
import numpy as np

class MazeF1(AbstractMaze):
    def __init__(self):
        super().__init__(np.matrix([
```

[3]https://github.com/ParrotPrediction/openai-maze-envs
[4]https://github.com/openai/gym/tree/master/gym/envs

```
            [1, 1, 1, 1],
            [1, 0, 9, 1],
            [1, 0, 1, 1],
            [1, 0, 0, 1],
            [1, 0, 1, 1],
            [1, 1, 1, 1],
        ]))
```

Once defined the maze needs to be registered

```
from gym.envs.registration import register

register(
    id='MazeF1-v0',
    entry_point='gym_maze.envs:MazeF1',
    max_episode_steps=50,
    nondeterministic=False)
```

Currently classical literature mazes like *MazeF1*, *MazeF2*, *MazeF3*, *MazeF4* [11, p. 181], *Woods1*, *Woods14*, *Maze4*, *Maze5 Maze6* are implemented.

*4.1.2 Boolean Multiplexer.* Multiplexer is typically used to evaluate the generalisation capabilities of LCS. It is described by a function whose complexity increases exponentially with the number of relevant attributes and is defined for lengths $l = k + 2^k (k \in \mathbb{N})$, where the first $k$ bits address one bit in the $2^k$ remaining bits.

Because ACS2 learns from the relation of situation-action-effect triples, it is necessary to add *perceptual causality* to the environment to make it solvable for ACS2 [2, p. 55]. In this case, the string is augmented with extra digit determining the correctness of classification (0 for the wrong answer, 1 for correct one).

In this case either observation space is equal to $\mathcal{I}_{mp} = \{0, 1\}^{l+1}$ and action space $\mathcal{A}_{mp} = \{0, 1\}$.

The implementation allows registering various length multiplexer problem (where the number of control bits is passed as an argument).

```
from gym.envs.registration import register

name = "boolean-multiplexer"
max_episode_steps = 1

register(
    id='{}-11bit-v0'.format(name),
    entry_point='gym_multiplexer:BooleanMultiplexer',
    max_episode_steps=max_episode_steps,
    kwargs={'control_bits': 3})
```

## 4.2 Pre-built Environments

*4.2.1 Go.* OpenAI Gym uses *Pachi Framework* for simulating Go gameplay (can be used with other board games such as Weiqi or Baduk as well). The framework is undergoing active research, implementing all suggestions leading to substantial performance improvements.

By default is uses UCT engine that combines Monte Carlo approach with tree search (UCB1AMAF tree policy using the RAVE method). It plays Go by Chinese rules and says to be about 7d KGS strength on 9x9 board. On 19x19, it can hold a stable KGS 2d rank. [5]

[5]http://repo.or.cz/pachi.git/blob_plain/HEAD:/README

*4.2.2 FrozenLake.* Frozen Lake is a maze-similar environment where the agent controls the movement of a character in a grid world. Some tiles of the grid are walkable; the others lead to the agent falling into the water. Additionally, the movement of the agent is not deterministic and only partially depends on chosen action. [6]

```
SFFF     (S: starting point, safe)
FHFH     (F: frozen surface, safe)
FFFH     (H: hole)
HFFG     (G: goal)
```

The agent always starts at the same location "S" and receives reward $\rho = 1$, when reaching final goal - "G".

## 5 PYALCS FRAMEWORK

The original Butz ACS2 algorithm was created in 2001 in C++ [6]. Apparently, the main advantage of choosing such a language is excellent control of all data stored in memory, thus performance efficiency. On the other hand research work, performing analysis and applying modifications is pretty costly and cumbersome.

Therefore the code was rewritten into Python 3 language which is now considered a top language used by data scientists. Having a huge ecosystem of external libraries (like interactive notebooks for reproducible analysis or various plotting libraries) consecutive research work can be applied much faster. Similar approaches was already taken to create educational purpose LCS systems (eLCS) [13] or ExSTraCS [14].

*PyALCS*[7] library provides an implementation of the ACS2 algorithm (can be easily customised or extended to other LCS) with OpenAI Gym interface. It was designed for rapid testing and evaluating algorithms by exposing two main classes - `ACS2` and `ACS2Configuration`.

### 5.1 ACS2 class

The `ACS2` class is responsible for declaring an agent that will be interacting with the environment. It can be created by passing two arguments:

(1) `ACS2Configuration` configuration object,
(2) an initial list of classifiers (optionally)

```
# Declare ACS2 agent
agent = ACS2(cfg)
```

Such an agent has three ways of interacting with an environment - *explore*, *exploit* (no ALP and GA modules are enabled, best classifier's action for given situation is executed), *explore/exploit* (consequently switch between explore and exploit modes in each trial).

Each mode takes two arguments - environment to interact with (OpenAI Gym compliant) and a number of trials. As a result, a list of classifier population and evaluation metrics are returned.

```
# Perform exploration for 10 trials
population, metrics = agent.explore(env, 10)
```

[6]https://gym.openai.com/envs/FrozenLake-v0/
[7]https://github.com/ParrotPrediction/pyalcs

**Table 1: OpenAI environment interaction configuration arguments.**

| Argument | Description |
|---|---|
| `perception_mapper_fcn` | A function for mapping perception obtained from OpenAI into PyALCS compliant vector form. |
| `action_mapping_dict` | Dictionary mapping internal *PyALCS* consecutive action numbers into specific OpenAI action representation. |

**Table 2: Metrics configuration arguments.**

| Argument | Description |
|---|---|
| `environment_metrics_fcn` | A function involving environment for calculating metrics based on its state (i.e. position of tokens on a board game). Might return a dictionary of various metrics. |
| `performance_fcn` | A function involving the environment, actual population of classifiers for calculating performance (i.e. estimating knowledge learned). Might return a dictionary of various metrics. |
| `performance_fcn_params` | Optional parameters passed as a dictionary to `performance_fcn` argument above. |

## 5.2 `ACS2Configuration` class

Configuration object serves as a single point of truth when declaring assumptions on how the algorithm will work (taking default parameter values from literature). It also allows configuring how observations and actions are mapped inside agents realm and collect custom metrics.

```
# Declare common ACS2 configuration object
cfg = ACS2Configuration(
    classifier_length=8,  # required
    number_of_possible_actions=8,  # required
    epsilon=0.7,  # overriding default
    do_ga=True  # overriding default
    )
```

The obligatory arguments are `classifier_length` specifying the length of perceptual string used in the representation of condition and effect string, and `number_of_possible_actions` specifying how many actions are possible for the agent to execute.

There is also a set of optional parameters related to interaction with environment described in Table 1 and for collecting custom metrics in Table 2.

## 6 INTEGRATION

This section shows how *PyALCS* framework can be integrated with any environment with discrete observation space. It will demonstrate a workflow when performing various integrations.

### 6.1 Explore/Exploit workflow

A typical example is to train the ACS2 agent for some set of trials, and later reuse created set of classifiers for exploitation.

Here the assumption is that the environment:

- returns a vector of strings representing current perception,
- all actions are expressed as consecutive numbers.

```
# Load desired environment
env = ...

# Prepare common configuration
cfg = ACS2Configuration(...)

# Explore the environment
agent = ACS2(cfg)
population, explore_metrics = agent.explore(env, 50)

# Exploit the environment
agent = ACS2(cfg, population)
population, exploit_metric = agent.exploit(env, 10)
```

**Listing 1: Explore/Exploit workflow.**

### 6.2 Standardising perception

Some environments might return state representation in a format not suitable for *PyALCS* (2D matrix, a vector of numbers, etc.). In this case, a special mapper function passed as `perception_mapper_fcn` configuration argument needs to be used parsing it into a vector.

```
# Load desired environment
env = ...

# Define a function taking as an argument
# current state obtained from environment.
# Process it returning a list of strings
def process_state(raw_state):
    pass

# Prepare configuration
cfg = ACS2Configuration(
        ...
        perception_mapper_fcn=process_state,
        ...)

# Explore the environment for 50 trials
agent = ACS2(cfg)
population, metrics = agent.explore(env, 50)
```

**Listing 2: Standardising perception.**

### 6.3 Standardising available actions

Often actions representations are not uniformed and can be assigned various values such as *"Down"* or *"0x22"*.

Some pre-processing by mapping all possible moves into a dictionary passed by `action_mapping_dict` argument needs to be made beforehand.

```
# Load desired environment
env = ...

# Prepare a dictionary mapping
# all possible actions.
moves = {
  0: '0x11',
  1: '0x24',
  ...
}

# Prepare configuration
cfg = ACS2Configuration(
        ...
        action_mapping_dict=moves,
        ...)

# Explore the environment
agent = ACS2(cfg)
population, metrics = agent.explore(env, 50)
```

**Listing 3: Standardising available actions.**

## 7 RESULTS

This section described the performance of *PyALCS* evaluation on previously described environments. All results are available as fully reproducible available as Jupyter notebooks or Python scripts[8].

All experiments were performed with default parameters (unless stated differently): $\beta = 0.05$, $\gamma = 0.95$, $\theta_i = 0.1$, $\theta_r = 0.9$, $\epsilon = 0.5$, $u_{max} = 100000$, $\theta_{exp} = 20$, $\theta_{ga} = 100$, $\theta_{as} = 20$, $\mu = 0.3$, $\chi = 0.8$, subsumption enabled and genetic generalization disabled.

### 7.1 Custom Environments

*7.1.1 Maze.* Plots 5, 6 describe the performance of the algorithm on the `Maze5` (3000 explore trials, and 400 exploit trails) and `Woods14` (1000 explore trials and 200 exploit trails) environments (visualised in Figure 4).

In both cases:

- *learned policy* showing best action in given situation (higher colour saturation means greater classifier's fitness value),
- *"achieved knowledge"* - dissecting every possible position on board and sees if there is a reliable classifier capturing the transition into neighbour states,
- number of overall classifiers (numerosity) and reliable ones during all trials,
- number of steps needed to find the reward.

Results show that the algorithm can thoroughly learn given both environments.

*7.1.2 Multiplexer.* The experiments were performed on 6bit, 11bit, 20bit and 37bit Boolean Multiplexer run during 1 000 000 trials in *"explore-exploit"* fashion. All parameters were left as default, with the exception that genetic generalisation mechanism was turned on. Figure 7 shows that the number of reliable classifiers converges to near optimal solution (which can be fine-tuned by adjusting the configuration parameters). Better results might be obtained trying to represent the condition and effect part with state-machine based encoding scheme [9], but this was performed on XCS so far, and further research for anticipatory classifiers is needed.

[8]https://github.com/ParrotPrediction/pyalcs/tree/master/notebooks

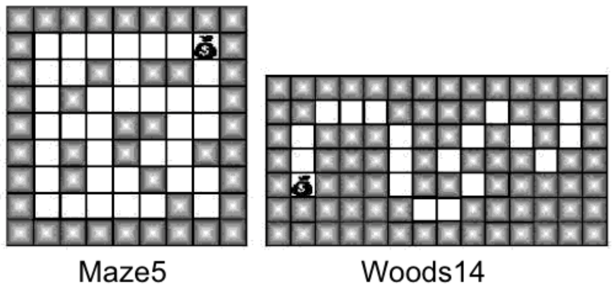

**Figure 4: Evaluated multi-step, deterministic maze environments. Figure adapted from [3, p. 187].**

### 7.2 Prebuild environments

Some effort was undertaken to obtain sensible results when evaluating the performance in pre-built OpenAI Gym environments.

*7.2.1 Go.* It's relatively easy to adjust ACS2 into the Go board realm. In all attempts, the more accessible version of the game played on 9x9 board was chosen. Agent's effect and condition part were represented by vectors holding a complete representation of the board, where each cell was described using three different states - black, white, empty. Both players could also execute 81 possible actions.

In this case, a simple evaluation metric, calculating the ratio between white and black tokens was measured. It's rising value in the long term could potentially mean that the algorithm understands the rules of the game trying to capture the majority of the board.

There were couple difficulties with this approach:

- putting the token onto a previously taken position on board results in an illegal action and immediately aborts the game,
- an agent was not possible to ever obtain a reward for winning a match (related to the previous point),
- a population of classifiers increased very fast which makes consecutive trials significantly slower (time needed to operate on all classifiers in match and action sets during learning phases was getting longer).

An attempt similar to one described in [10] increasing the speed of learning by incorporating two competing ACS2 agents (sharing a collective population of classifiers) was also examined but without luck.

*7.2.2 FrozenLake.* The original version of the FrozenLake environment was not suited for the basic version of ACS2 due to its stochastic nature. The agent was unable to capture the states changing in an indeterministic way, although implementing extensions described in the literature as PEEs [5] might help.

However, it's possible to reconfigure environment always to execute the desired action (`is_slippery` modifier). After this modification, it behaves like a typical maze environment and is solvable by the algorithm. Figure 8 depicts the number of macroclassifiers in both cases. In basic version ACS2, it is not able to converge.

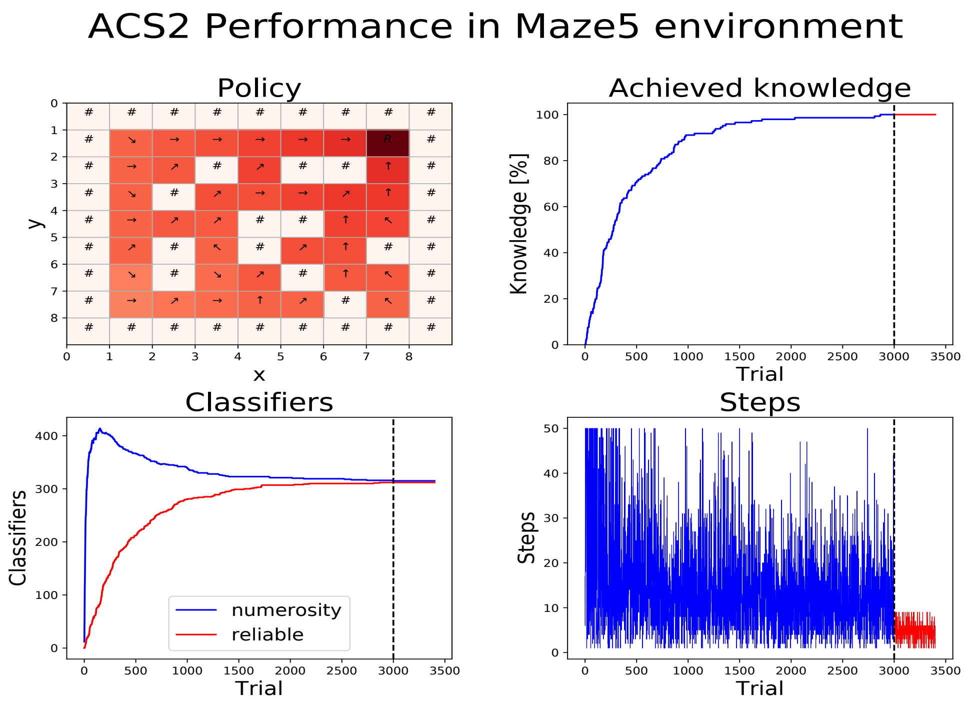

**Figure 5: Performance obtained in Maze5 environment. By performing 3000 explore trials and 500 exploit trials agent was able to perfectly learn the maze. As visualized in policy each state was visited enough times to build strong confidence (greater saturation). The algorithm starts to reach nearly 100% knowledge near 1500 trial slowly stabilising the number of classifiers needed to represent it.**

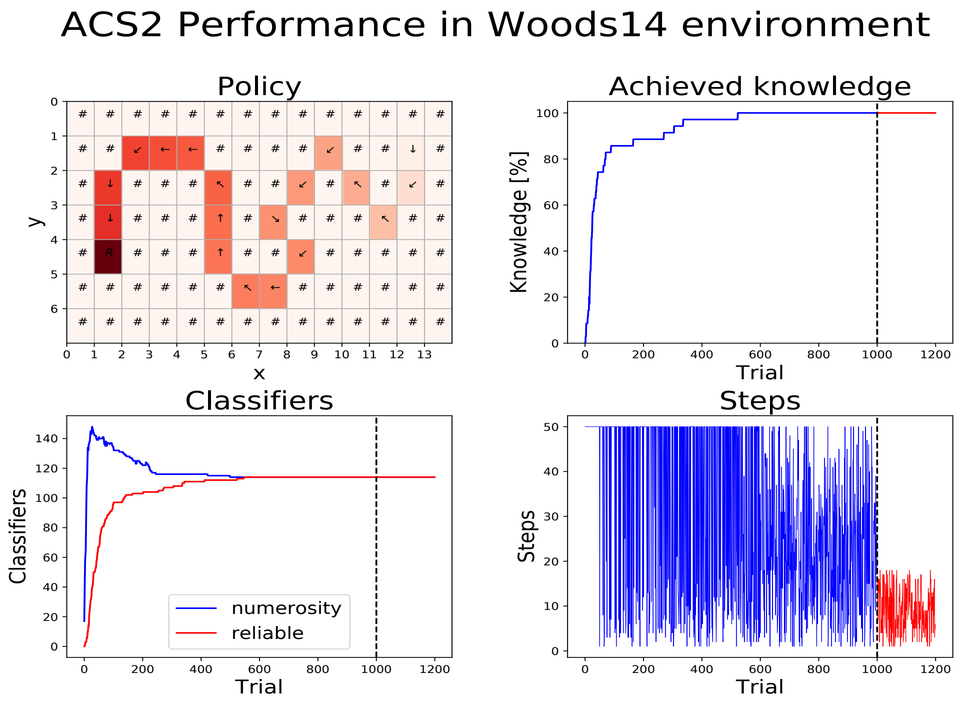

**Figure 6: Performance obtained in Woods14 environment. The algorithm performed 1000 explore and 200 exploit trials. In contrast to figure 5 we see that further states are not so confident as those closer to the reward (need more trials). Although also perfect knowledge is obtained.**

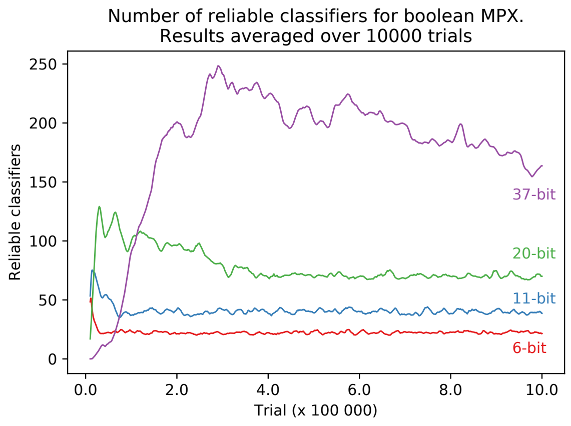

**Figure 7: The number of reliable classifiers in 6bit, 11bit, 20bit and 37bit Boolean Multiplexer tends to converge. Results are averaged on 10 000 trials.**

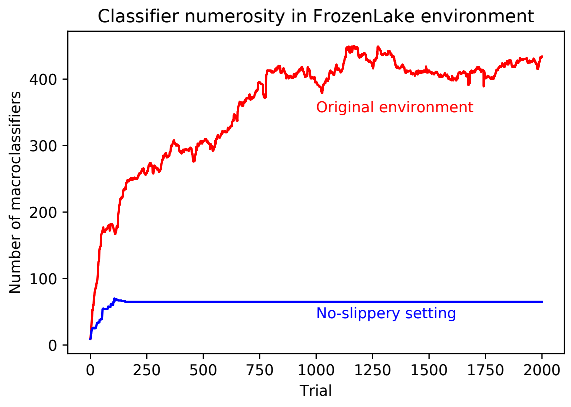

**Figure 8: Number of macro-classifiers in both original (non-deterministic) and modified (deterministic) version of the FrozenLake environment.**

## 8 CONCLUSIONS AND FUTURE WORK

This paper demonstrates full synthesis between ACS2 and *OpenAI Gym* interface. Such integration enables access to a vast majority of environments (at the moment of writing there are dozens to choose from) used in latest reinforcement learning approaches and encourages scientists to perform fully reproducible researches.

By switching to Python language code is much more concise. Understanding it and extending takes significantly less time. But the biggest benefit is the access to huge eco-system of tools facilitating modern data processing. Things like aggregating results, visualizing them in form of interactive reports (like Jupyter Notebook or Zeppelin), reconfiguring experiments are now a breeze. These aspects encourage researchers to investigate deeper the approach or might even contribute to bugfixes or own improvements.

The major drawback, however, is that ACS2 at this point is only capable of interacting with environments with discrete observation space. Work regarding creating an extension enabling communicating with continuous observation space [12] or dealing with situations where not all effects are predictable (non-determinism) will allow new opportunities.

## ACKNOWLEDGMENTS

The work was supported by statutory grant of the Wroclaw University of Science and Technology, Poland.

## REFERENCES

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. 2016. OpenAI Gym. *ArXiv e-prints* (June 2016). arXiv:cs.LG/1606.01540
[2] Martin V Butz. 2002. *Anticipatory learning classifier systems*. Vol. 4. Springer Science & Business Media.
[3] Martin V. Butz. 2006. *XCS in Reinforcement Learning Problems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 181–195. https://doi.org/10.1007/3-540-31231-5_10
[4] Martin V. Butz, David E. Goldberg, and Wolfgang Stolzmann. 2000. Investigating Generalization in the Anticipatory Classifier System. In *Parallel Problem Solving from Nature PPSN VI*, Marc Schoenauer, Kalyanmoy Deb, Günther Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 735–744.
[5] Martin V. Butz, David E. Goldberg, and Wolfgang Stolzmann. 2001. Probability-Enhanced Predictions in the Anticipatory Classifier System. In *Advances in Learning Classifier Systems*, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–51.
[6] Martin V. Butz and Wolfgang Stolzmann. 2002. An Algorithmic Description of ACS2. In *Advances in Learning Classifier Systems*, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–229.
[7] J. Hoffmann. 2016. *Vorhersage und Erkenntnis*. Universität Würzburg. https://books.google.pl/books?id=C2iYAQAACAAJ
[8] John H. Holland. 1985. Properties of the bucket brigade algorithm. *Proceedings of an International Conference on Genetic Algorithms and their Applications* (1985), 1–7.
[9] Muhammad Iqbal, Will N Browne, and Mengjie Zhang. 2013. Extending learning classifier system with cyclic graphs for scalability on complex, large-scale boolean problems. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 1045–1052.
[10] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* 550 (Oct. 2017), 354–. http://dx.doi.org/10.1038/nature24270
[11] Wolfgang Stolzmann. 2000. An Introduction to Anticipatory Classifier Systems. In *Learning Classifier Systems*, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–194.
[12] Olgierd Unold and Marcin Mianowski. 2016. Real-Valued ACS Classifier System: A Preliminary Study. In *Proceedings of the 9th International Conference on Computer Recognition Systems CORES 2015*, Robert Burduk, Konrad Jackowski, Marek Kurzyński, Michał Woźniak, and Andrzej Żołnierek (Eds.). Springer International Publishing, Cham, 203–211.
[13] Ryan J. Urbanowicz and Will N. Browne. 2017. *Introduction to Learning Classifier Systems* (1st ed.). Springer Publishing Company, Incorporated.
[14] Ryan J. Urbanowicz and Jason H. Moore. 2015. ExSTraCS 2.0: description and evaluation of a scalable learning classifier system. *Evolutionary Intelligence* 8, 2 (01 Sep 2015), 89–116. https://doi.org/10.1007/s12065-015-0128-8
[15] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3 (01 May 1992), 279–292. https://doi.org/10.1007/BF00992698