Farm-Gym: A modular reinforcement learning platform for solving agronomy-inspired games.

Odalric-Ambrym Maillard
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9198-CRIStAL, F-59000 Lille, France
ODALRIC.MAILLARD@INRIA.FR

Editor:

Abstract

We introduce `Farm-gym` to the reinforcement learning community, a modular gym platform dedicated to providing a range of agronomy games, implemented in various environments of diverse difficulty that are inspired from realistic challenges in agronomy and agroecology. The platform works as a kind of atari platform, providing a single interface to many games, each game being an environment compatible with the openai gym abstract programming interface. A specific game is created in a modular way, by specifying a farm layout and choosing to use some modules each governing the dynamics of some entity (such as the soil, a plant, pollinators, etc.) in its own way, and specifying scoring, constraint and stopping conditions of the game. Each entity comes with a set of available actions, and interaction with other entities, as well as a yaml parameter file specifying different instances (e.g. bean, tomato, corn for the plant entity). By varying the number of modules, instances, size of the farm and scoring rules, the researcher in reinforcement learning can create from simple to highly complex environments. The dynamics of each provided default entity is highly simplified, to focus on interactions and loop-effects rather than accurate biological equations. This contrasts with agronomy accurate decision-support simulators that provide a highly realistic dynamics of a single entity (usually the weather, soil and plant) while neglecting other influential entities (weeds, pests, pollinators, etc.), and are also usually not adapted to step by step interactions. Last, a user can specify a new module, or a new version of an existing module to expand the catalogue of simulated entities. We specify a few games and provide some illustrative experiments using classical reinforcement-learning algorithms.

Keywords: Reinforcement Learning, Open-source, Simulation environment, Agronomy, Gamification.

1. Introduction

Reinforcement Learning (RL) is a popular machine learning paradigm to model the problem of *decision making* under uncertainty when interacting with a dynamical system. The uncertainty may come from the fact the system is only partially known (information uncertainty), or that the dynamics of the system is too complex to be represented accurately by the decision maker (representation uncertainty), or simply due to intrinsic stochastic nature of the system (noise uncertainty). Over the last decades, a series of challenging environments have been made available by the researchers in the field to help identify and overcome bottlenecks hindering the development of reinforcement learning. By fostering coopetition and peer-review in a reproducible-friendly framework, such environments attracted significant focus that eventually yield novel ideas benefiting the field at large. For instance the Atari games Pitfall and Moctezuma from the Arcade Learning Environment Bellemare et al. (2013) revealed the need for safer and better exploration in large state-space environments. Following the initial openai-gym environments (e.g. Atari, Mujoco, Classic control tasks) Brockman et al. (2016), a number of third parties environment have been provided by the community, on video games, board games, robotics, autonomous driving, etc. to highlight a specific difficulty. For instance the high-way environment Leurent (2018) aims at safe planning in the context of

traffic with vehicles whose dynamics is unknown; the NLPgym from Ramamurthy et al. (2020) focuses of reinforcement challenges in the context of Natural Language Processing.

Motivation and challenges Motivated by filling the gap between actual reinforcement learning and real-world societal applications, we introduce to the RL community a novel set of environments inspired by *agronomy*. Decision making in agronomy shares many similarities with reinforcement learning. Unlike more traditional environments (e.g. go, atari), the process of growing a plant in a farm is naturally *stochastic* rather than deterministic, due to the key influence of external factors such as the weather (but also population of insects, spontaneous plants, etc.). Moreover, the process naturally involves a number of entities of various types (such as the weather, the soil, the plants, the insects, etc.) co-interacting, which may generate non-trivial dynamics involving loop effects and equilibrium. Each entity is coded with its own dynamic possibly interacting with other entities. By combining various entities to form an environment, the resulting global dynamics for the environment *couples* the dynamics of each of its entities in a modular and challenging sway. Besides, a key bottleneck in agronomy is the ability to *observe* (measure) some specific entity, which usually comes with a cost that must be traded-off with acting on the field (e.g. watering, harvesting). For instance, observing if a pest-attack or disease occurs may be crucial to decide weather or not to add specific nutrients, but may take significant time preventing the farmer to perform another action. Hence, due to their natural cost-constrained nature, agronomy tasks induce what we call an *observation-action trade-off*. Last, the typical goal in agronomy is not to obtain a good score (e.g. yield) in expectation, but rather to get a good score according to some *risk-aversion* criterion. In practice, the score may further combine various variables (e.g. bio-diversity index, soil-health index) according to each farmer's preferences. We call the specification of a score together with a risk-aversion criterion a score model. Addressing each of these four features (stochastic dynamics, coupled dynamics, cost-constrained actions, score model) represents a non-trivial reinforcement learning challenge of independent interest beyond the application to farming.

Literature overview Development of RL algorithms for different applications often start with developing simulators or gamified platforms that mimic the challenges of the real-world complex problems in computationally controllable manner. For example, Atari games (Bellemare et al., 2013) provided a stimulating benchmark to develop RL algorithms in the context of computer games. Mujoco (Todorov et al., 2012) is proposed as a physics-driven engine for fostering RL algorithms for continuous control and robotics. OpenAI Brockman et al. (2016) proposed a set of lightweight RL environments with a standardized Application Programming Interface (API), called OpenAI `gym`. Presently, `gym` API became a reference in the RL community to create standardized RL environments in order to compare performances of RL algorithms. Following `gym`, several platforms are developed to foster reproducible frameworks and improved algorithms. For example, the highway environment Leurent (2018) aims at safe planning in the context of traffic with multiple vehicles and unknown dynamics, and NLPgym Ramamurthy et al. (2020) focuses of Natural Language Processing challenges. Often these `gym` frameworks have deterministic environments which are unknown and reveal themselves through noisy observations.

Regarding RL attempts at agriculture, Garcia (1999) first proposed an RL agent interacting with a crop simulator to learn to maximize wheat yield while restricting nitrogen fertilization. Recently, Sun et al. (2017); Chen et al. (2021); Yang et al. (2020) use different RL algorithms to maximize the yield when focusing on learning the best irrigation schedules. Trépos et al. (2014) learns a technical itinerary that considers the date of sowing and the potential weather conditions. Unfortunately, none of these works provides an open source and standardized RL environment. In context, a few `gym` environments are proposed that model various decision-making problems while trying to optimize the yield of a given crop. For example, `CropGym` Overweg et al. (2021) focuses on fertilization strategies. Kemanian et al. (2022) introduces `CyclesGym` that simulates multi-year crop strategies (e.g. crop-rotation strategies). But often the models

considered here are deterministic like traditional `gym` environments. Last but not least, Gautron et al. (2022) introduces `gym-DSSAT`, an environment with maize fertilization and irrigation problems. It is backed by the DSSAT crop models, allowing accurate simulation of a wide range of real-world growing conditions. However, apart from the weather, all other processes are deterministic.

The existing works consider specific settings of interest and focus on optimizing simple strategies (e.g. date of sowing or fertilization) for maximizing yield output with accurate models of plants. They do not consider either interactions between different entities (e.g. weeds, insects and plants), or complete stochasticity as seen in real-world. In contrast, `Farm-gym` provides a holistic platform to study different entities and interactions in a farm under stochasticity and user-defined objectives. As such, we are unaware of reinforcement learning gym environments targeting such challenges, especially that of coupled-dynamics in a stochastic environment.

An agronomic system can be seen as a complex system with many parameters to take into account. Thanks to modern technologies the farmer gets more and more data about his crops and fields (humidity, temperature, lack of nutrients, ...). To help him cope with this increasing flow of information and to help him achieve his personal objectives, Decision Support Systems (DSS) have been developed since the 1980s. Sheng and Zhang (2009) defined it as "a human-computer system which is able to collect, process, and provide information based on computers". Nevertheless, it is fundamental to understand that this system does not seek to substitute for human choice, in the end it is the farmer who makes the decision. Hence such systems are decision companions. It is even crucial to note that for the same agronomic system a DSS should produce different recommendation depending on the farmers' preferences, some seeking to maximize e.g. yield, others to minimize the risks taken or the amount of resources spent. Hence DSS should implement personalized-recommendation, adapted to farmer preferences and farming context. There already exists many programs that model agronomic system like DSSAT (Decision Support System for Agrotechnology Transfer) that successfully simulates growth, development and yield of crops as a function of the soil-plant-atmosphere dynamics. Nonetheless, while such models are extremely detailed (thanks to many years of agronomic research) on one aspect (mainly the soil-plant-atmosphere dynamics) the often fail to take into account other important phenomenons like pest insects-weeds-plant dynamics.

Contribution `Farm-gym` takes a complementary gamification approach in contrast to the high-precision simulators whose goal is to accurately model an individual entity. Each farm is constructed with multiple entities in a *modular* way, which can be seen as a game. We can choose which dynamics of the environment to simulate, and also to combine these dynamics to study specific coupling phenomena. In the end, each farm, created by composing several entities in a modular way can be seen as a game, the set of games that `Farm-gym` offers can be seen as a form of the popular arcade environment Atari but for agronomy games. The agronomy-oriented `Farm-gym` platform offers a variety of environments to focus on addressing reinforcement learning in the context of four stimulating challenges: intrinsically *stochastic* dynamics, interaction and *coupled* dynamics, the *observation-action trade-off* and user-defined *score models*. In order to help design novel strategies progressively addressing all these challenges, the `Farm-gym` platform is designed to host a number of environments built by combining different modules, cost-constraint and scores, such that each environment can be considered as a separate game of various and progressive difficulty. While we provide some initial simple and more challenging games, the platform is designed so that new modules can be added easily, hence offering the possibility to build games involving more complex and refined interactions in the future. We believe designing learning strategies able to simultaneously solve many `Farm-gym` games will help pave the way towards real-life applicable reinforcement learning.

Organization and overview We provide some background material and formalism on reinforcement learning in MDPs and PO-MDPs in Section 2. We detail the main architecture of the `Farm-gym` plat-

form in Section 3.1, where we explain the APIs and provide a first use-case in a simple environment for illustration purpose. In Section 4, we detail a few pre-defined modules implementing some key entities that are provided in our implementation, focusing on their dynamics and interaction with other entities. We detail in greater details the Plant module in Section 4.2 and Soil module in Section 4.3 that are the most complex ones. In Section 5, we make use of the generic architecture to explain the specification of a few games highlighting challenges of various difficulty related to the agricultural concerns listed above. We provide a case-study illustrating a few interesting behavior.

Contents

2. Markov Decision Processes and friends

A Markov decision process (MDP) is a classical framework to study sequential decision process in a dynamic environment. It is specified Puterman (2017) Sut (2018) by a tuple $(\mathcal{S}, \mathcal{A}, \mathbf{p}, \mathbf{r})$, where $\mathcal{S}$ denotes the state space and $\mathcal{A}$ the action space. The dynamic of the MDP is defined by a possibly stochastic transition function $\mathbf{p} : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ to which is associated a reward function $\mathbf{r} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{P}(\mathbb{R})$. At decision time $t \in \mathbb{N}$, the system is in state $s_t \in \mathcal{S}$, the learner chooses actions a_t, then the system transits to a new state $s_{t+1} \sim \mathbf{p}(s_t, a_t)$ at the next decision time and produces reward $r_{t+1} \sim \mathbf{r}(s_t, a_t, s_{t+1})$. Note that crucially, the fact that $s_{t+1} \sim \mathbf{p}(s_t, a_t)$, that is, the transition only depends on the description of the state at previous decision step s_t and the chosen action a_t (and not on any state before this) induces a key constraint on the definition of the states: they must describe a Markov process (of order 1). In practice, the MDP framework is flexible enough to model various decision making problems by defining an appropriate state space. However, to be complete one must specify what information is made available to the learner at each time step. Indeed in many situations, the learner cannot directly observe the full state s_t describing the system at time t. Rather there is a limited set of D-many (possibly stochastic) observation functions $\mathbf{o}_d : \mathcal{S} \to \mathcal{P}(\mathbb{R} \cup \{\perp\})$, $d \in [D]$ such that for each d, only observation $o_{t,d} \sim \mathbf{o}_d(s_t)$ is available, where $\perp$ stands for no observation. For example, in a field, a farming agent has only a partial view of the full development status of its plants, nutrients available in the soil and presence of insects, some variables being observed and some not. Hence in practice, the state s_t is often not directly accessible to the learner.

Fully, Partially, and Actively observed MDPs We distinguish three different setups depending on the information made available to the learner: In a Fully-observate Markov Decision Process (FO-MDP), the knowledge of the state s_t of the system at time t is directly given to the learner (with the observations functions defined above, $D = 1$ and $\mathbf{o}_1(s)$ is the Dirac mass at state s). In a Partially-observed Markov decision process (PO-MDP), the knowledge of the state s_t is not available. However an observation o_t is given to the learner, given by a (fixed) set of variables functions of the state s_t. In an Actively-observed Markov decision process (AO-MDP), the knowledge of the state s_t is not available, and the set of variables to form the observation o_t must be actively chosen by the learner (that is the learner chooses a subset $\mathcal{D}_t \subset [D]$). In such a setup, observing variable $\mathcal{D}_t$ further comes with a cost $c_t \sim \mathbf{c}(\mathcal{D}_t)$ specified by a cost function $\mathbf{c} : 2^D \to \mathcal{P}(\mathbb{R})$. Hence a FO-MDP is fully specified by $(\mathcal{S}, \mathcal{A}, \mathbf{p}, \mathbf{r})$, a PO-MDP is fully-specified by $(\mathcal{S}, \mathcal{A}, \mathbf{p}, \mathbf{r}, (\mathbf{o}_d)_{d\in[D]})$, and an AO-MDP is fully specified by $(\mathcal{S}, \mathcal{A}, \mathbf{p}, \mathbf{r}, (\mathbf{o}_d)_{d\in[D]}, \mathbf{c})$. The goal now is not to maximize rewards, but to maximize rewards minus observation costs. We now summarize the interaction between a learner and an MDP in each of these three setups:

FO-MDP interaction: At decision time t the state of the system is s_t. The learner receives observation s_t and reward r_t (resulting from the previous action) and chooses action a_t to be played. Then action a_t is played and the system moves to state s_{t+1}. Then the learner observes s_{t+1} and reward r_{t+1} before playing action a_{t+1}. A typical strategy of the system is $(s_t, r_t, a_t)_{t\in\mathbb{N}}$ with initial state s_1 and initial reward $r_1 = 0$.

PO-MDP interaction: At decision time t, the state of the system is s_t. The learner receives observations $o_t = (o_{t,d})_{d\in[D]}$ (as a function of the s_t) and reward r_t (resulting from the previous action) from the fix set of variables $[D]$ and chooses action a_t to be performed. Then action a_t is played and then the system moves to state s_{t+1}. Then the learner receives observation o_{t+1} from the same variables and reward r_{t+1} before playing action a_{t+1}. A typical trajectory of the system is $(s_t, o_t, r_t, a_t)_{t\in\mathbb{N}}$, with initial state s_1 and initial reward $r_1 = 0$.

AO-MDP interaction: At decision time t, the state of the system is s_t. The learner receives observations $o_t^F = (o_{t,d})_{d\in F}$ (as a function of the s_t) from a (possibly empty) fix set of variables $F \subset [D]$ and reward r_t

(resulting from the previous action), then chooses actively a set of variables $\mathcal{D}_t \subset [D] \setminus F$ to be observed, forming $o_t^A = (o_{t,d})_{d \in \mathcal{D}_t}$ (from variables $\mathcal{D}_t$, as a function of the s_t), and paying cost $c_t \sim \mathbf{c}(D_t)$. Then, having observed $o_t = (o_t^F, o_t^A)$, the learner chooses action a_t to be performed. Then action a_t is played and then the system moves to state s_{t+1}. Then the learner receives observations o_{t+1}^F and reward r_{t+1}, then chooses the next set of variables $\mathcal{D}_{t+1}$ to be observed, receives observation o_{t+1}^A before playing action a_{t+1}. A typical trajectory of the system is $(s_t, o_t^F, r_t, \mathcal{D}_t, o_t^A, c_t, a_t)_{t \in \mathbb{N}}$, with initial state s_1 and initial reward $r_1 = 0$, and the goal is to maximize the sum of $r_{t+1} - c_t$ over decision time steps.

Remark 1 (PO-MDP are FO-MDP on belief state space) *In a PO-MDP, the states s_t are unobserved hence latent. Still, the state space $\mathcal{S}$ is known and one may form a belief state space $\tilde{\mathcal{S}} = \mathcal{P}(\mathcal{S})$. The belief-MDP $(\tilde{\mathcal{S}}, \mathcal{A}, \tilde{\mathbf{p}}, \tilde{\mathbf{r}})$, is then defined with $\tilde{\mathbf{p}}(\tilde{s}_t, a_t)$ denoting the transition from the belief state $\tilde{s}_t$ to the posterior belief state $\tilde{s}'$ after seeing observation o_t, while the belief reward $\mathbf{r}(\tilde{s}, a, \tilde{s}')$ integrates out $\mathbf{r}(s, a, s')$ according to $\tilde{s}(s)$ and $\tilde{s}'(s')$ for $s, s' \in \mathcal{S}$. Of course the resulting state space $\mathcal{P}(\mathcal{S})$ is continuous even when $\mathcal{S}$ is discrete, which makes solving PO-MDPs a non-trivial task.*

Remark 2 (AO-MDP are 2-step PO-MDP) *An AO-MDP can be seen as a PO-MDP $(\tilde{\mathcal{S}}, \tilde{\mathcal{A}}, \tilde{\mathbf{p}}, \tilde{\mathbf{r}}, (\tilde{\mathbf{o}}_d)_{d \in [D]})$ with twice many decision steps. Indexing the decision-steps by half-integers ($1, 1+1/2, 2, 2+1/2$, etc.) the action set is $\tilde{\mathcal{A}}_t = 2^D$, $\tilde{\mathcal{A}}_{t+1/2} = \mathcal{A}$ for $t \in \mathbb{N}$. The extended state space is $\tilde{\mathcal{S}} = 2^D \times \mathcal{S}$, with transition $\tilde{\mathbf{p}}((\mathcal{D}, s), a) = \delta_{\mathcal{D}} \otimes \mathbf{p}(s, a)$, $\tilde{\mathbf{p}}((\mathcal{D}, s), \mathcal{D}') = \delta_{\mathcal{D}'} \otimes \delta_s$, extended reward $\mathbf{r}((\mathcal{D}, s), a, (\mathcal{D}'', s')) = \mathbf{r}(s, a, s')$, $\mathbf{r}((\mathcal{D}, s), \mathcal{D}', (\mathcal{D}'', s')) = -\mathbf{c}(\mathcal{D}')$, and extended observations $\tilde{\mathbf{o}}_d(\mathcal{D}, s) = \mathbf{o}_d(s)$ if $d \in F \cup \mathcal{D}$ else $\tilde{\mathbf{o}}_d(\mathcal{D}, s) = \delta_\perp$. In particular, the action chosen at $t + 1/2$ is allowed to depend on the observation $o_t = (o_t^F, o_t^A)$ received after choosing observation set $\mathcal{D}_t$.*

Contextual and forecast-contextual MDPs While MDPs, and PO-MDPs are popular formulations, in practice it is useful to consider separately the case when exogenous variable not controlled by the learner (that is, it has it own evolution, irrespective of the actions of the learner), influencing the dynamics of the system. A typical example of exogenous variable in an agriculture context is the weather. Another one is the type of soil structure (e.g. sand, clay), considered in first approximation unaffected by the farmers' actions during a crop season, that drastically modify the evaporation rate hence the dynamics of the system. Hence, each value of the exogenous variable induces a different PO-MDP. Instead of considering them separately and learning each PO-MDP separately, a learner may consider learning the set of all PO-MDPs parameterized by the exogenous variable, as they share similar state and actions, and possibly close dynamics and rewards. For instance the dynamics in a 25%-sand, 75%-clay soil may be close to that of a 26%-sand, 74%-clay soil. Contextual MDP (C-MDP), introduced by Assaf Hallak and Mannor (2015), formalize this idea and define a contextual MDP as $(\Theta, \mathcal{S}, \mathcal{A}, (\mathbf{p}_\theta, \mathbf{r}_\theta)_{\theta \in \Theta})$, where Θ denotes the set of context (a.k.a. parameter set). So, essentially, a C-MDP is just a set of models sharing the same state and action space. Of course this can be applied to any FO,PO or AO-MDP model. Note that the context $\theta_t \in \Theta$ at time t is considered known by the learner, and also comes with its own dynamics. The context can be piece-wise constant, only changing after interacting with each corresponding MDP for T decision steps, which is useful when modeling a sequential (one-by-one) learning task on a bunch of MDPs. The context can also be changing at each time step, hence interleaving the dynamics of several MDPs. When considering the rain variable for instance, each specific rain time-series induces a different MDP dynamics. However, since the learner does not know the rain ahead of time, it is actually facing a contextual-MDP with context given by the rain that is possibly changing at each time step. From this perspective, C-MDP are reminiscent of Robust-MDP framework (when at each time step, a dynamics is chosen by Nature among a known set of dynamics), with the difference that the observation of contextual information θ_t tells us which MDP the learner deals with at each time step.

More generally, in practice when facing a C-MDP, the learner may access different information regarding the sequence $(\theta_t)_t$ at each time t. Without observing the sequence, the learner only knows that they are in Θ, which is the robust-MDP framework. In the case when at time t, θ_t is observed perfectly, this is the setup of Assaf Hallak and Mannor (2015). Now it often happens that a learner gets more information at time t about the $\theta_{t'}$ for $t' > t$ (e.g. rain forecast). To completely specify a C-MDP, we hence introduce a forecast function $\mathcal{F} : \mathbb{N} \to \mathcal{P}(\Theta^\infty)$ that gives, for each time step t a distribution over the possible values of $(\theta_t, \theta_{t+1}, \dots)$. We call this a forecast-contextual-MDP (FC-MDP), denoted $(\Theta, \mathcal{S}, \mathcal{A}, (\mathbf{p}_\theta, \mathbf{r}_\theta)_{\theta\in\Theta}, \mathcal{F})$. Intuitively, the forecast models how the agent may access a noisy version of the future evolution of certain components of the latent states. When available, we consider the forecast as one of the observation variables. In agriculture, beyond the weather forecast, one may consider different forecasts such as the forecast of pest-attacks. The case of Robust-MDP is recovered when e.g. the support of $\mathcal{F}(t)$ is simply $\Theta^{\mathbb{N}}$, while in a classical contextual MDP, it would be $\{\theta_t\} \times \Theta^{\mathbb{N}}$. For a rain forecast with 2-day look-head, if $\theta_t \in [0, 1]$ indicates a rain probability, the support of $\mathcal{F}(t)$ could be of the form $\{\theta_t\} \times [\theta^-_{t+1}, \theta^+_{t+1}] \times [\theta^-_{t+2}, \theta^+_{t+2}] \times [0, 1]^{\mathbb{N}}$. Such knowledge can be exploited by the learner to better control the system.

Farm-Gym builds FC-AO-MDPs We instantiate each farm-gym environment as a Forecast-Contextual Actively-observed Markov Decision Process (FC-AO-MDP), where the forecast is simply given by a forecast on the weather that is noiseless for the current day t and noisy for future days $t' > t$ with noise increasing with $t' - t$. The forecast is seen as a specific observation, that can hence be asked actively or not. Further seeing AO-MDPs as PO-MDPs, the FC-AO-MDP structure can be implemented within the OpenAI-gym framework. A typical interaction between a learning agent `learner` and a farm-gym environment `farm` with rewards stored in `rewards` would resemble the following:

```
while (!is_done):
  // First step: observations
  observation_request = learner.choose_observations()
  observations, _, _, info = farm.step(observation_request)
  obs_cost=info['observation cost']
  learner.observation_update(observations,obs_cost)
  // Second-step: interventions
  intervention_request = learner.choose_interventions()
  observations, gain, is_done, info = farm.step(intervention_request)
  int_cost=info['intervention cost']
  learner.update(observations,gain, is_done, int_cost)
  // Storing the rewards:
  rewards.append(gain-obs_cos-int_cost)
```

3. A Reinforcement Learning environment platform for agronomy challenges

We recall that modeling an agronomic system such as a farm, is a challenging task for several reasons. We list (non-exhaustively) that this is an intrinsically *stochastic* system e.g. subject to the variation of weather or pest attacks, with *interaction* of many entities at different spatial (plant, crop, ...) and temporal scales (day/night, week, month, year), creating *coupled dynamics* (predator-pray dynamic, equilibrium, ...), in which *observations* should be *actively* acquired, and are *costly*, in order to optimize a *risk-averse* and *personalized* objective function.

In this section, we present `Farm-gym` a platform to create gym environments that model the agronomic system of a farm implementing all these challenges. It contains a set of entities that can be added in a modular way, hence allowing the apparition of complex and intricate dynamics; the main entities (plant, soil) are modeled with stochastic dynamics; the observations should be chosen actively, and all costs, scores

and objective function can be specified. Our environment uses simplified (but sound) models of growth and development of crops that allows us to easily add more relations between the different variables that describe the system.

3.1 High-level architecture of the platform

A `Farm-gym` environment is the environment with which an agent interacts. It is instantiated as follows

`farm=Farm(fields,farmers,scoring,rules).`

Fields A farm consists of several `fields` that together describe the spatial organization of the system. A field is defined by a location, a shape and a list of entities. The location gather latitude, longitude and altitude information that may typically affect the weather. Each field of shape (`width`,`length`,`scale`) contains `width` × `length` many plots each of size `scale`. A plot is the smallest spatial unit at which interactions between entities take place. Namely, each entity attached to the field will simulate the dynamics of this entity in each plot of the field. In the current version, the location is only used for the weather entity.

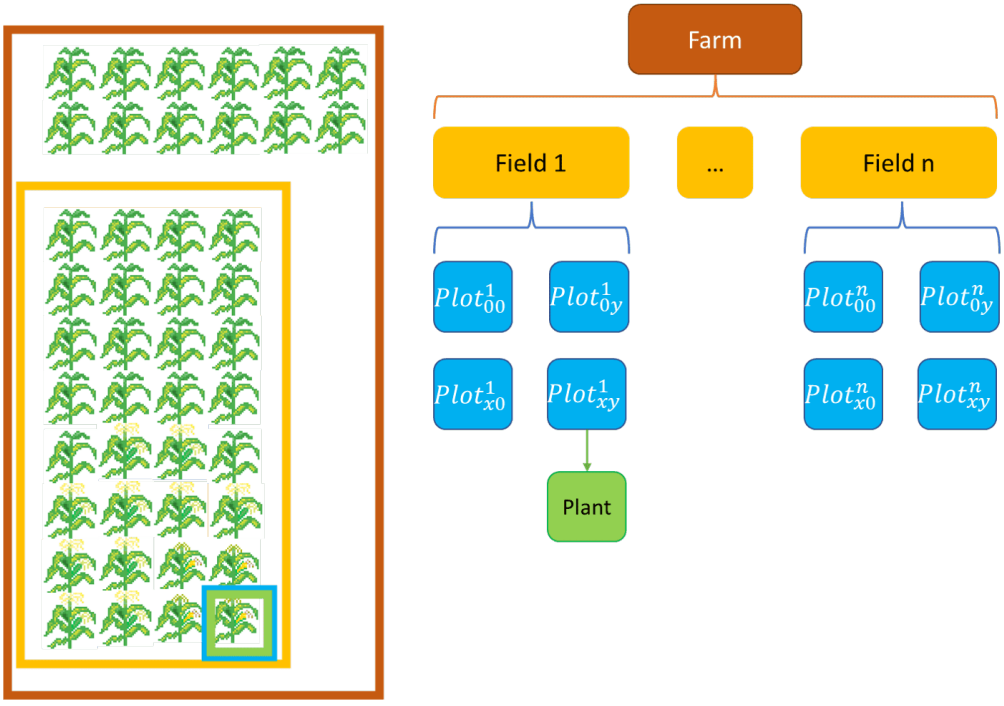

Figure 1: On the left a farm with two fields, on the right the structure of the farm. The entity `plant` simulates the dynamics in each plot of the field.

Instantiating one field specifying one single Plant entity would typically look like[1]:

```
field1=Field(    localization={'latitude#°': 0, 'longitude#°': 0, 'altitude#m': 100},
                 scale={'length#nb': 8, 'width#nb': 4, 'scale#m': 1.},
                 entities=[(Plant,'bean')]   )
```

1. In `Farm-gym` , we adopt the convention that all physical parameters are named in the form *name#unit*, and counting parameters have no units.

Farmers When an action (observation or intervention) is specified by a learner, it is execute by a farmer. A farmer simply allows or not executing an observation and intervention by being available/busy. This enables to implement constraints in the action space. Various farmer modules can be implementing considering various availability models. The default farmer module specifies a farmer who is always available for all observations and interventions, and counts the number of interventions and observations done each day. When the number of observations reaches a maximal value, it stops observing. Likewise, when the number of interventions reaches a maximal value, it stops intervening.

Scoring, rules, policies We discuss the scoring, rules and policies in greater detail later below. At a high-level, the scoring specifies the cost and rewards associated with each observation and intervention. The rules specifies the initial conditions (initial state) of the game, the subset of allowed observations/interventions, the set of observations given for free to the learner at each step, and the conditions under which the game ends. It may also specify a maximum allowed cost for executing an action.

Entities Entities are the physical components of the agrosystem. Example of entities are the weather, the soil, a plant, pollinators, or fertilizers. An entity is attached to a specific field and may have parameters. An entity specifies a dictionary of state variables, actions (observations, interventions), and list of other entities on which its dynamics may depend. A state variable has a name, indicated by the key in the dictionary of state variables, and it must be either of a specific type Range, a numpy array or `Range` values, or a dictionary of sub-variables. The type `Range` codes either a bounded real-number, with minimal and maximal values, or a discrete value in a finite list of values. For instance, the weather entity defines the state variable `wind` as a dictionary with two keys `speed#km.h-1` and `direction` indicating to sub-variables. The wind is in range (0,500) while the direction is in a list `['W','S','E','N']`. A typical initialization is

```
self.variables["wind"]={"speed#km.h-1": Range((0.,500),0.),
                        "direction": Range(['E','S','W','N'],'W')}
```

On the other hand, the soil entity defines a state variable `available_Water#L` as a numpy array of size `field.shape['length']×field.shape['width']`, providing the amount of a water available in each plot of the field. Crucially, each entity specifies its dynamics describing the evolution of the entity in each plot of the field. All entities must implement the same abstract programming interface (API), which enables creating a farm environment in a modular way. Each entity module is associated to a yaml parameter file, hence (Plant,'bean') in the example above instantiates an entity Plant with the parameters specified for the entry 'bean' available in the corresponding plant.yaml file.
In order to represent a sufficiently rich agronomic system, `Farm-gym` provides the following 10 predefined entities. In the future additional entities may be added, such as diseases, mushrooms or slugs.

- The **plant** that uses nutrients from the soil, water and sun to grow. Its development is directly influenced by the weather, soil, pests, pollinators and seed-eating birds. It is typically impeded by pest insects and concurrence with weeds. The full dynamics of this entity is detailed in Section 4.2. We provide three types of plants `corn`, `bean`, `tomato` that have fairly different pollination parameters and growing conditions. The plant entity specifies three possible interventions: **sowing**, **harvesting** and **removing** the plant, and the possibility to observe many state-variables of the plant.

- The **weather**, providing the dynamics of variables such as the daily minimum, maximum and average temperature in Celsius, humidity, occurrence of rain as well as wind and daily sun exposure. The entity specifies no intervention. Observations include current observation as well as a noisy 5-day forecast.

- The **soil**. It contains nutrients and can retain water more or less efficiently. In order to keep the model simple we restrain ourselves to the 4 main nutrients that are `nitrogen (N)`, `potassium (K)`, `phosphorous (P)`, and `carbon (C)`. The soil crucially monitors the amount of available water, due to rain, sun evaporation, plant evapo-transpiration and watering. Four types of soil (that retain more or less the water) are available: `sand`, `loam`, `silt`, and `clay`. The soil also has some microlife activity that favors absorption of nutrients in the soil from fertilizers and prevents nutrients leaching. It is negatively affected by pesticides and herbicides. It comes with one intervention: **watering**, that is adding some specific amount of water.
- The **weeds** that take water and nutrients from the soil faster than the plant, impede it from growing, and spread in nearby plots. The entity specifies one intervention: **removing** the weeds from a plot.
- The **pest insects** that consume the plants and the weeds, and move from plots to plots. The entity specifies no intervention. We can observe their population.
- The **pollinators** that are necessary for insuring the pollination of certain crops. The entity specifies no intervention. We can observe their presence.
- The **birds**, that eat the seeds or insects. They are repulsed by a scarecrow. The entity specifies no intervention. We can observe their population.
- The **fertilizers** that enrich the soil with one type of nutrient (`N`, `K`, `P`, `C`), diffusing at various speed. We specify different types of fertilizers. The entity specifies one intervention: **fertilizing**, that is adding some amount of fertilizer on the soil.
- The **cides** that kill weeds (herbicides) or kill insects (insecticides) (that is both pest insects and pollinator insects indifferently), diffusing at various speed. Each cide also reduces (kills) the microlife activity of the soil. The entity specifies one interventions: **scattering cide** on the soil. We consider two instantiations of cides, one herbicide and one pesticide.
- The **facilities** that represent all structural modification of the environment in the field. This may include a hedge attracting insects and birds, or a scare-crow deterring the birds. The entity comes with one intervention: **adding a scarecrow**.

To summarize, the nine available categories of interventions made available to the farmer are:

- **sowing**, **harvesting**, **removing a plant**,
- **removing weeds**,
- **watering the soil**,
- **fertilizing**,
- **scattering herbicide**, **scattering insecticide**,
- **adding a scarecrow**.

In practice in a `Farm-gym` environment, some entities may not be present, hence reducing the number of possible interventions. On the other hand, on a field there may be several entities of the same type, for instance different plants in case of a polyculture agrosystem, or different types of fertilizers, scarecrows, etc, which hence increases the total number of available interventions accordingly. For instance, the list

`([Plant, 'bean'), (Plant, 'carrots')]` enables the learner to monitor two species of plants on the same field. Last, each intervention may come with some parameters that must be specified when defining an action. A typical example of parameter is the plot where the intervention should take place. Now some interventions may come with no parameters.

In Figure 2, we illustrate a summary of the entities and their interactions. This help clarifying that due to the coupling dynamics between the different entities, acting on one variable may have gigantic consequences latter on the whole system. For instance, a counter-intuitive example that emerges from these settings and that is seen in real life is that if you kill weeds in your field too quickly at the beginning of the sprouting period of your cultures all the pest insects may concentrate on your crop potentially yielding a smaller harvest than by letting grow a few weeds in your field.

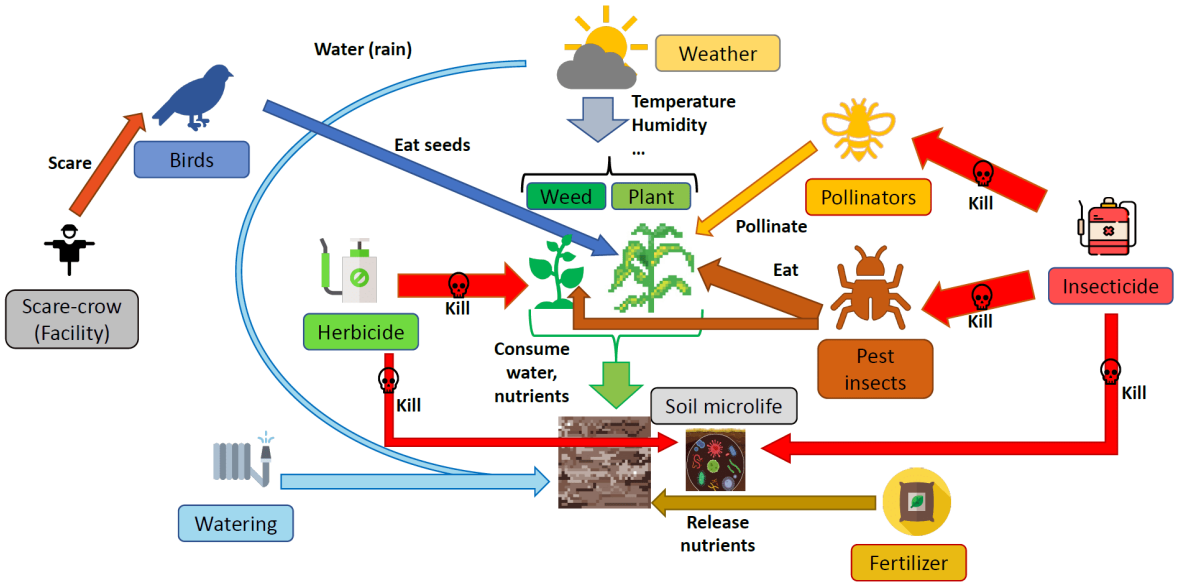

Figure 2: Main interactions between the entities used to describe the system's dynamic.

3.1.1 Actions: Interventions and Observations

In the implementation of `Farm-gym` , the complete specification of an action (either an intervention or observation) follows the following scheme

(<farmer>, <field>, <entity>, <name>, <parameters>).

In this format, each farmer, field and entity is automatically indexed by their name and numbered according to their creation ordering. For instance it two plants are instantiated on a field, say first a bean and then corn, then the bean plant is referred as plant-0, and the corn as plant-1. Here <names> indicates either the name of an entity state variable to be observed, or the name of action (such as watering, scattering-herbicide, etc.) to be performed, while <parameters> defines additional parameters of the action or observation.

Parameters For an observation, <parameters> is a *list* representing a path to a specific component or sub-variable of the state-variable. For instance, requesting to the first farmer in the first field to observe the state-variable `size` (height) of the first plant at plot (2,3) (that is, component (2,3)) would be (`farmer-0`, `field-0`, `plant-0`, `size#cm`, `[(2,3)]`). Note that providing instead (`farmer-0`, `field-0`, `plant-0`, `size#cm`, []) would request to observe the state variable `size#cm` in the entire field, hence the array containing the value of the size of the plant in each plot.

For an intervention, <parameters> is a dictionary of parameters of the action. For instance, the entity fertilizer defines an action scattering taking as parameters the plot where to scatter the fertilizer, and a real-value indicating the amount that is scattered. Specifying the actions would be (farmer-0, field-0, fertilizer-0, scattering, {'plot':(2,3), 'amount#kg':0.5}). Some categories of actions have no parameter in the current implementation, such as harvesting (hence assuming it is always done the same way for simplification). For instance, requesting to harvest the first plant in the first-field (so, entirely, in each plot) would be coded (farmer-0, field-0, plant-0, harvest, {}).

Output When executing an action (intervention or observation request), and output is produced. It is always given as a (possibly empty) list of items in the following format

(<field>, <entity>, <name>, <parameters>, <value>).

Multiple actions In the considered games provided in Farm-gym , we restrict to games when only a single observation and single intervention is allowed per day. However, Farm-gym supports asking a list of observations and interventions each day. Such a list is called a schedule in the code. This considerably increases the complexity of set of actions $\mathcal{A}$, seen from the perspective of a reinforcement learning formalization, moving from a finite set A to A^k if k is the maximal number of (elementary) actions allowed in am action schedule. This enables the interested user to define a challenging environment involving a challenging combinatorial MDP learning problem.

3.1.2 Scoring: rewards, costs, etc.

Specifying a score is a key component of a Farm-gym game, as this is where all rewards are defined. More precisely, any scoring module must implement the same interface, specifying

- a **cost** function mapping any action (observation/intervention) to a real value c called the cost.
- a **reward** function, taking as input a field and returning a real value r computed from the state variables of its entities.
- a **final reward** function, taking as input a field and returning a real value computed from the state variables of its entities.

The first two functions are called at each time step, for each executed action and for each field, and their result is added to form the daily reward. Hence one have e.g. the value $r_1+r_2+r_3-c_1-c_2$ in case there are three fields and two actions are executed by the learner on a certain day. The final reward function is only called at the end of the game, that is when the stopping conditions are triggered according to the rules of the game. It works similarly to the reward function. Note that each game may specify a list of observations given for free to the learner at each time step. The cost of these observations is set to 0, hence the cost function only applies to the additional observations actively asked by the learner.

Predefined score In principle, any scoring module can be used. In practice, since designing a score module may be tedious, Farm-gym provides a basic-score module, that enables to specify the cost of each action and rewards thanks to a simple yaml configuration file.

The *intervention costs* are defined in a yaml file, where a fixed value is assigned to any action that is applied, irrespective of its parameter. Likewise, the *observation costs* are defined in the same yaml file, where a fixed value is assigned to any observation. This value is the unit cost for each entry of the observation. Hence, when a path is specified, the corresponding cost is computed as the number of components of the sub-variable specified by the path, times the unit observation cost.

The *rewards* are defined to be 0, except when a plant moves from one development stage to another, in which case it is given the value 1. This is a very sparse reward signal.

The *final rewards* are defined as a user-defined combination of basic agronomic values, with weights provided in the configuration file. More precisely, the final reward is computed according to $r = \sum_j w_j r_j$ where $j \in \{\text{bio, resource, soil, yield, plant}\}$. The $(w_j)_j$ are the weights specified in the yaml file, assumed in $[0, 1]$. The r_j are computed based on the entities.

- $r_{\texttt{bio}}$ counts the total occurrence of birds, pollinators, pests and weeds observed in during the interaction,
- $r_{\texttt{resource}}$ counts the total amount of water, fertilizer and cides used during the interaction,
- $r_{\texttt{soil}}$ measures the final micro-life activity level of the soil,
- $r_{\texttt{yield}}$ measures the total yield in kg of the harvest of all harvested plants,
- $r_{\texttt{plant}}$ measures some development index of the plant. This includes, for each plot, whether the plant has sprouted or not, the ratio of the size of the plant over its potentially maximal size, the number of pollinated flowers over the number of flowers and the weights of the fruits over the potentially maximal fruit weight.

By adapting the configuration file, one may easily specify combination of these basic rewards, and specify the cost of each action.

3.1.3 Rules: start, end-game conditions, and allowed actions

A rule specification of the game specifies a few things, such as the first day of the game, the set of free observations, the starting conditions and stopping condition of the game.

The *starting conditions* of the game include possible initialization of each entity in each field of the farm (in particular, the starting day, handled by the `Weather` entity). They are summarized in an init yaml file, where each state variable of each entity in each field can be assigned a default value. Not all variables need to be listed in this file, the missing variables are initialized to the default value defined by the corresponding entity class.

The *free observations* is a list of observation actions that are executed for free at the beginning of each stage. Since they are given for free, they are not assumed executed by any farmer. They are coded with format (<field>, <entity>, <name>, <path>), where the farmer field is dropped, as illustrated below

```
free_observations= [ ('Field-0', 'Weather-0', 'rain_amount',[]),
                     (('Field-0', 'Weather-0', 'air_temperature', ['max#°C']),
                     (('Field-0', 'Soil-0', 'available_Water#L', []) ]
```

The *allowed actions* specifies the subset of actions allowed in the game. While a learner may decide to play whatever action provided by the each entity module at each decision step, it may be useful for some games to restrict the user a subset of available actions only. Specifying a restricted set of allowed actions may considerably simplify the learning problem, by focusing the learner on a smaller number of possibilities. Of course this prevents the learner from improving beyond the best policy on this restricted set. This is detailed later in this section.

The *stopping conditions* indicate when to stop the game and set the output `is_done=True` when asking for a step. This condition can be to stop the game after some fixed number of days, or more generally when some specific event is triggered. They are represented with conjunctive-normal-form (CNF) formulas.

To illustrate, a typical instance of rule may look like the following. We detail the starting conditions (`init_configurations`), stopping conditions (`terminal_conditions`) and action space configuration (`actions_configuration`) in the paragraphs below.

```
rules= BasicRule( init_configuration='game0_init.yaml',
                  actions_configuration='game0_actions.yaml',
                  free_observations=free_observations,
                  terminal_conditions=term_conditions)
```

Starting conditions Starting conditions specify the initial value of each variable of each entity. When `farm.reset()` is called on a `Farm-gym` instance `farm`, all entities are reset, then the variable specified in the configuration file are setup to their corresponding values. This enables the user to only specify a few variables and the entities reset all other variables to their default value. For each variable, the user an specify a single value or a range, in which case a random value will be generated in this range at each reset. Now, some variables are vector variables. This is the case of the `stage` variable that indicates for each plot position the corresponding development stage of the plant in this plot. Instead of specifying one value for each, a user can directly specify one value, that will be applied to all entries of the vector. Hence, using such conventions an example init configuration file may look like:

```
Field-0:
    Weather-0:
      day#int365: [50,57,64]
    Soil-0:
      microlife_health_index#%: 80.0
    Plant-0:
      stage: seed
```

In this example, the initial day can be any of the three indicated values. The microlife helath index of the soil is fixed to 80% for each plot and the stage of the plant is set to seed in each plot of the considered field.

Restricted action space Below, we present a typical yaml configuration file displaying all intervention actions generated for a farm that consists of a unique 3×1 field with a single plant, soil, and weather entities will display the possible interventions as presented below. This farm is associated with six types of interventions, each with various parameters, resulting in a total of (60+ 96+ 1+ 3+ 3=)163 discrete and 12 continuous base actions.

```
interventions:
  Field-0:
    Soil-0:
      watering_discrete:
        plot: ['(0, 0)', '(1, 0)', '(2, 0)']
        amount#L: [1.0, 2.0, 3.0, 4.0, 5.0]
        duration#min: [30, 60]
      watering_continuous:
        plot: ['(0, 0)', '(1, 0)', '(2, 0)']
        amount#L: (1, 10)
        duration#min: [30, 60]
    Plant-0:
      sow:
        plot: ['(0, 0)', '(1, 0)', '(2, 0)']
        amount: [1, 3, 5, 10, 15, 20, 25, 30]
        spacing#cm: [5, 10, 15, 20]
      harvest:
      micro_harvest:
        plot: ['(0, 0)', '(1, 0)', '(2, 0)']
      remove:
        plot: ['(0, 0)', '(1, 0)', '(2, 0)']
```

A user may want to restrict only to a subset of possible actions, by specifying subset of allowed actions. In the specification below, there are only two types of interventions, resulting in a total of 1 discrete and 3 continuous actions only.

```
interventions:
  Field-0:
    Soil-0:
      watering_continuous:
        plot: ['(0, 0)', '(1, 0)', '(2, 0)']
        amount#L: (5, 10)
        duration#min: [30]
    Plant-0:
      harvest:
```

Likewise, a user may want to restrict the observations that can be asked by a learner. For instance in a game where some weather observations (such as rain) is given for free to the learner, one may restrict the observations to a few possibilities only. Since observations are indicated using a path, one may stop at any depth of the path and not necessarily at a leaf: we introduce a star character to indicate that we request the whole tree of sub-variables available in case the variable is a dictionary, and the whole vector of values in case it is a value. To illustrate this, the following file for a 3×1 field specifies 10 observation actions with three special `'*'` actions. The request `'*'` for the wind asks all observations available regarding the variable wind (that is `speed#km.h-1` and `direction`), hence a dictionary of size two. The request `'*'` for `available_Water#L` asks all information available for this variable, that is the value in each plot, hence a vector of size three.

```
observations:
  Field-0:
    Weather-0:
      wind:
        '*':
    Soil-0:
      available_Water#L: ['*','(0, 0)', '(1, 0)', '(2, 0)']
    Plant-0:
      global_stage:
    size#cm: ['*','(0, 0)', '(1, 0)', '(2, 0)']
```

Last, since a typical Farm-gym action consists in a list of base intervention or observation actions, a user can specify in the yaml configuration file the maximum length of this list.

```
params:
  max_action_schedule_size: 5
```

Stopping conditions In the code, a the stopping condition event is defined as a generic conjunctive-normal-form formula: The formula ($C1$ and $C2$) or ($C3$) or ($C4$ and $C5$) is represented as the list

$$[[C1, C2], [C3], [C4, C5, C6]].$$

Here each Ci codes an event in the format

$$(\texttt{<observation>}, \texttt{<mapping>}, \texttt{<operator>}, \texttt{<value>}).$$

In this format, <observation> denotes one of the state variables of the entity, in the standard format (<field>,<entity>,<name>,<path>). Now, we allow in <mapping> to specify a custom function mapping the variable <name> to any custom space. <value> is a reference value in this custom space. Finally, the operator <operator> maps pairs of values in this custom space to Boolean. It is used in particular to compare the result of the mapping to the reference value. Typical operators include comparison $==, !=, <=, >=, <, >$ and set operators in, ni. Hence the occurrence of an event is defined by the test

$$\texttt{<operator>}(\texttt{<mapping>}(\texttt{<observation>}), \texttt{<threshold>}).$$

For instance, the event that the current day exceeds day number 250 of the year is represented[2] as follows

```
(('Field-0', 'Weather-0', 'day#int365', []), lambda x: x.value, '>', 250)
```

A stopping condition using such event or the event that the plant is harvested or dead in CNF may look like:

```
term_conditions= [ [(('Field-0', 'Weather-0', 'day#int365', []), id, '>=', 250)],
  [(('Field-0', 'Plant-0', 'global_stage', []), id, 'in', ['harvested','dead'])] ]
```

where id denotes the mapping lambda x:x.value.

3.1.4 Configuration files

Farm-gym provides a few tools to automatically generate configuration files for the score (cost, reward), the initial conditions and the considered action space. More precisely, template files are automatically loaded if specified, or automatically generated when the configuration file does not exist yet or is unspecified. For instance running the command

```
game0 = Farm( fields=[field], farmers=[farmer], rules=rules, scoring=scoring )
```

generates configuration files game0_score.yaml, game0_init.yaml and game0_actions.yaml in the current folder, initialized with all entries written in the prescribed format. These three yaml files detail the configuration respectively for the score, initial conditions and restricted action/observation space. A user may then easily customize the configuration for instance by deleting lines in game0_actions.yaml to specify a subset of available actions only.

3.1.5 Gym compatibility

A Farm-gym environment finally implement the gym environment interface. Before detailing them, we first introduce a specific Space structure enriching the list of existing gym spaces (that is Box, Discrete, Dict, Tuple, etc).

2. Indeed "day#int365" is a Range variable in (0,365), with value accessible by .value

Union and MultiUnion spaces A Union of spaces implements the set-theoretic union of several spaces. A sample from a Union gym space is a pair, with first value indicating the index of the space, and the second value the sample in that generated in space. A MultiUnion implements the power set of a union of spaces. We further introduce a parameter m, so that `MultiUnion[m]` denotes the set of all subsets of maximal cardinality m instead of the full power-set.

State space In `Farm-gym` , each environment generates a gym *state space* provided in `state_space` and automatically built from the specification of a farm. This is simply a gym `Tuple` with components corresponding to each variable in each entity for each entity in each field. Each variable can either be a gym `Box`, `Discrete` or `Dict` depending on the considered variable to be observed.

Action space The gym *action space* is built using a specific structure called a `MultiUnion` from the restricted list of actions specified by the configuration file. The spaces that are considered are `Dict(1)` for each observation action, and `Dict` spaces for each intervention action to specify the corresponding parameters. To illustrate, a typical action space generated from a `Farm-gym` farm with 3 observation actions, 2 intervention actions and restricting to $m = 2$ may look like:
`MultiUnion[2](Discrete(1),Discrete(1),Discrete(1),`
`Dict(amount#L:Discrete(5), duration#min:Discrete(2), plot:Discrete(3)),`
`Dict(plot:Discrete(3)))`
Sampling from this space generates lists of actions of size at most 2 randomly chosen between the 5 base actions. A method `gymaction_to_farmgymaction` further enables to map any gym action to its corresponding `Farm-gym` action. For instance the gym action `(1,0)` may be mapped to the `Farm-gym` action `('BasicFarmer-0', 'Field-0', 'Soil-0', 'available_Water#L', [])`

Step `Farm-gym` specifies a `farmgym_step` method that takes as input a list of actions in `Farm-gym` format, as specified in the previous sections, and `gym_step` method that takes as input an action from the `MultiUnion` gym action space generated for this farm. The `step` method is equivalent to `gym_step`.

Rendering `Farm-gym` enables to produce a rendering of the farm in several ways. First printing the farm with `print(farm)` displays in text mode an informative summary of the farm, including its name, fields, entities and state variables, as well as available actions. On top of that, the `farm.render()` method generates images in `.png` format of the farm at each time step (day). Finally, two methods are provided, `generate_video` and `generate_gif`, to generate a video and a gif from the images.

Monitoring A FarmGym environment typically comes with many states. FarmGym provides the possibility to monitor each state variable. That is, to add a way to visualize the evolution of the state variable over time. To do so, one should just add run the method `farm.add_monitoring(var)`, where `var` is the list of considered variables.

The syntax for a variable to be monitored is the following one:

```
(field, entity, variable, function, title, range)
```

where the first three elements indicate the state variable that one wants to be monitored, `function` ia a user-defined function mapping the state variable either to a real-value, a matrix or an image. Finally, range indicates a predefined range of values, and can be set fo "`range_auto`" by default. Below is an example to monitoring the sum of the available quantity of nitrogen in the whole field, using the predefined function `sum_value`

```
var.append(("Field-0", "Soil-0", "available_N#g", lambda x:sum_value(x)
                , "Available Nitrogen (g)", 'range_auto'))
```

3.2 Example of a simple game

In order to better understand the `Farm-gym` environment, we detail below an example of a simple game in which the agent only has to control the amount of water poured each day and when to harvest. We consider a 4×3-plot field with only three entities, a simple weather from Montpellier, a clay soil, and one plant chosen to be corn. `Farm-gym` enables to generate a game in a modular way. In this simple illustrative game, there is no pest insect, pollinator, nor birds. There are however weeds, with the possibility to remove them manually, or to add herbicide. Removing weeds has no side effect, while adding herbicide kills both the weeds and part of the soil microlife.

```
field = Field(  localization={'latitude#°': 0, 'longitude#°': 0, 'altitude#m': 100},
                shape={'length#nb': 4, 'width#nb': 3, 'scale#m': 1.},
                entities=[(Weather, 'montpellier'), (Soil, 'clay'), (Plant, 'bean')] )
```

We define the rules of the game by providing the learner with free weather observations define in the previous section with same terminal conditions. The init configuration files specifies that all plots start in stage `'seed'`, with random water level in the soil.

```
rules = BasicRule(  init_configuration='game0_init.yaml',
                    actions_configuration='game0_actions.yaml',
                    free_observations=free_observations,
                    terminal_conditions=term_conditions)

scoring = BasicScore( cost_configuration='game0_cost.yaml',
                      reward_configuration='game0_reward.yaml')
```

We finally instantiate the farm, considering one farmer that may perform arbitrarily many observations per day but only a single intervention per day, and a scoring considering...

```
farm = Farm( fields=[field], farmers=[BasicFarmer(max_daily_interventions=1)],
             rules=rules, scoring=scoring )
```

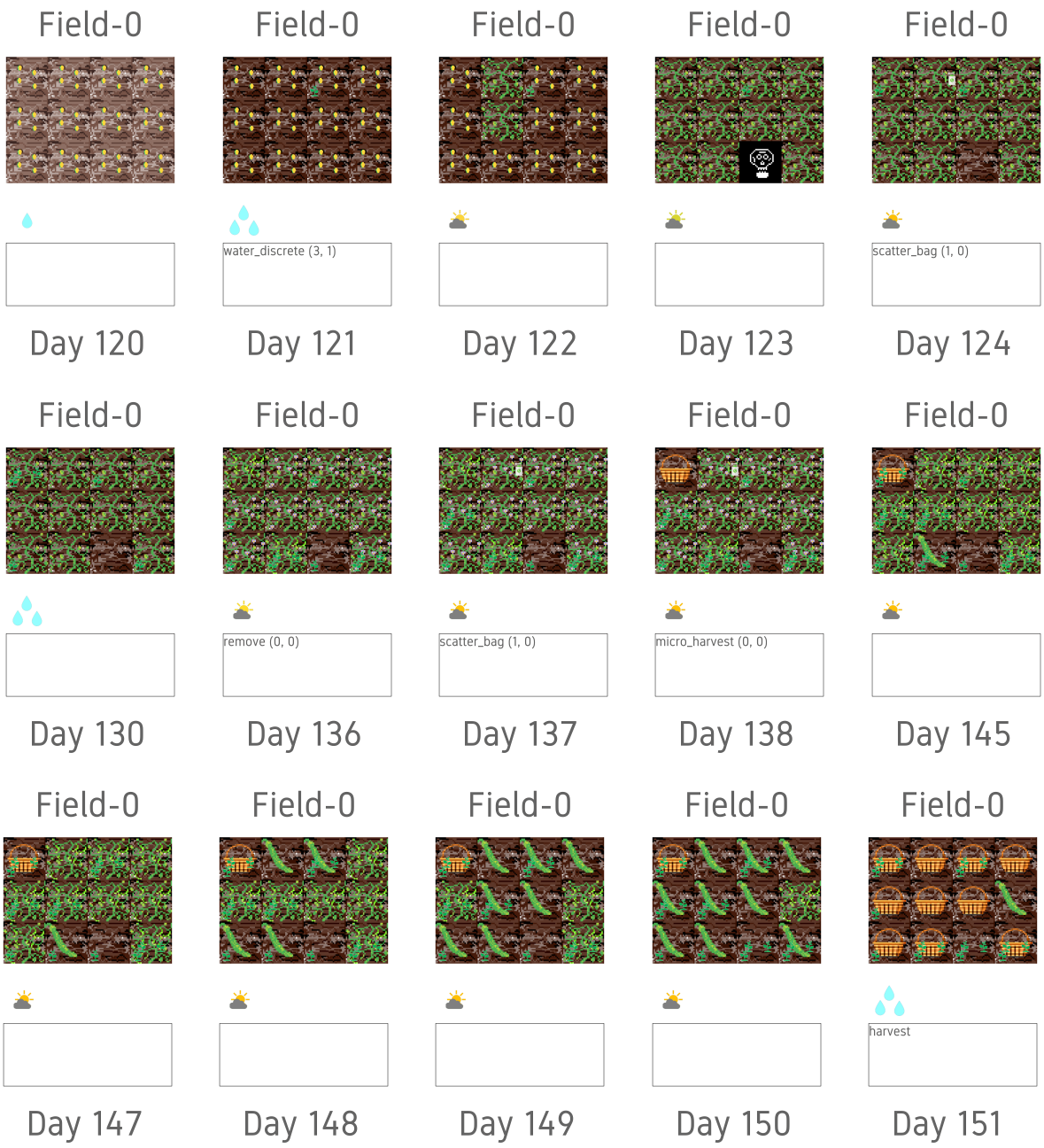

Figure 3: Example illustrating the stages of the crop at a few days (120 to 151) when using a simple random policy to decide when and where to water, remove weeds, scatter herbicide or harvest. The plant used in this example is bean and the soil is clay.

Figure 3 shows the evolution of the field at different stages of development until the harvest. At the beginning (day 120) there are only seeds in the field, then some seeds sprout, together with weeds (121, 122), one dies (123). On day 124, some herbicide is scattered, it is later washed out by rain (130), not without killing some of the soil microlife. On day 136, weeds is removed at $(0, 0)$ while beans start blooming. Some early harvest

is done at $(0, 0)$ on day 138. On day 145, the first plot of beans is ripe, other follows until day 150. On day 151, a full harvest is done, ending the game. Here harvested is done at the right time, waiting too long would potentially make the beans die.

Monitoring The `Farm-gym` platform allows the developer to monitor any set of variables of an environment. This can be done using `farm.add_monitoring(var)` where `var` denotes a list of observations to be monitored. Since by default, monitoring only enables to visualize real-values, matrices and images, a variable to be monitored must be associated with a function mapping its values to the desired output. Further, the variable to be monitored can be given a custom name, for convenience. Last, one may specify a specific range of value or ask the monitoring to find an automatic scale. Hence all in all, the syntax of a monitored variable is:

```
(field,entity, variable, function, name,range)
```

For instance, the size of the plant at position (0,0) can be monitored in the following way

```
('Field-0', 'Plant-0', 'size#cm', lambda x: x[0,0].value, "Size (cm)", 'range_auto')
```

Likewise, the population of pests on the entire field of size `X` $\times$ `Y` can be monitored as a matrix in the following way, using the conversion function `mat2d_value`

```
('Field-0','Pests-0','plot_population#nb',lambda x:mat2d_value(X,Y,x),"Plot population
                                                    (nb)",'range_auto')
```

We provide in Figure 4 below a snapshot showing the evolution of a few monitored variables as a function of the day, for a simple 1×1 farm. In this case the soil is sand and the plant is tomato, and only real-valued variables are monitored. We observe for instance here that the plant is experiencing nutrient stress (this might be due to the fact that, despite the presence of Nitrogen in the soil, the microlife health index stays low, making it little available to the plant). We can also observe the evolution of the plant.

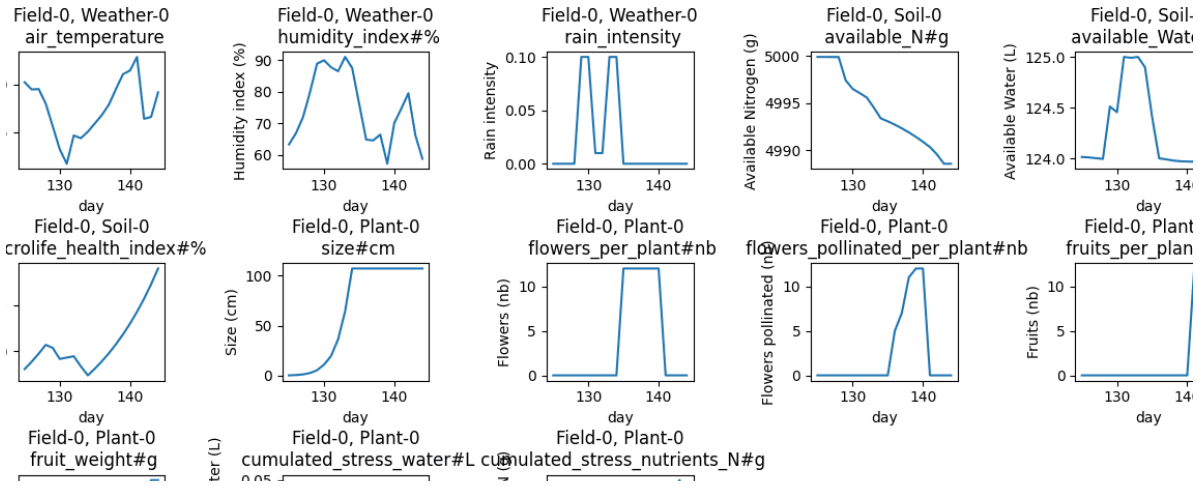

Figure 4: Monitoring of a few variables on a farm.

3.3 Custom specification of farmers, entities, rules and scores

Each of the farm component of a `Farm-gym` environment can be reimplemented, as long as it fits the provided abstract programming interface. This includes the farmers, the rules, the scores as well as each

entities. It is possible to create novel entities to implement additional traits (e.g. fungus, disease, etc.) or build novel version of existing entities, to code an entity with more accurate dynamics.

4. Predefined entities

In this section, we detail the implementation of a some predefined entities, especially focusing on their dynamics. The entities described below allow the emergence of `Farm-gym` environments with a rich dynamic, by composing certain elements (water management, nutrients, insect pests, etc.) that can be activated in a modular way to vary the complexity of an environment. When we look at a real agricultural system, there are two types of actions that can be undertaken by the agent, *interventions* such as watering or applying fertiliser and *observations* that allow the agent to acquire more information such as measuring the quantity of water present in the field or the presence of insects. Hence observations model how an agent can go and get additional information on the environment from time to time. For instance the farmer may want to predict the level of water in the field using the quantity of water added to the system, the rain, the temperature, type of soil and the growth of crops. Nonetheless, it will inevitably make some mistakes, under or overestimating the quantity of water in the field. To avoid that, the farmer may take the time to measure (observe) the water level before some crucial point in the crop development.

On top of these observations, the user has access to free observations, and also some contextual information, such as shape of each field of the farm. Unlike free observations, *observations actions* can be associated with a cost to retrieve them. We can specify a price on these actions in the score configuration file, in which case the agent must learn to make parsimonious use of the extra information. It can for instance use them to decide if it is the time to harvest or if the level of pest insects is a threat for the crops.

Interventions We list below the possible parameterized interventions allowed by our pre-defined entities.

Entities	Intervention actions	Intervention parameters
Bird		
Cide	`scattering` `scattering_bag`	`plot, amount#kg` `plot, amount`
Facility	`put_scarecrow` `remove_scarecrow`	`type`
Fertilizer	`scattering` `scattering_bag`	`plot, amount#kg` `plot, amount`
Pest		

Plant	`sow`
	`harvest`
	`micro_harvest`
	`remove`
		`plot, amount, spacing#cm`
		
		`plot`
		`plot`

Pollinator		
Soil	`watering_discrete` `watering_continuous`	`plot, amount#L ,duration#min` `plot, amount#L ,duration#min`
Weather		
Weeds	`remove`	`plot`

Each entity may come with a few parameters. For instance, watering must specify the plot to which it is applied, the amount of water that is poured and the duration of the watering during the day (due to water evaporation on the ground, and water absorption speed, the duration of watering affects the amount of water

available in the soil). These parameterized interventions allows for wide range of strategies, especially if we constrains the agent with different costs. For instance, one agent can use Nitrogen in high quantity with slow diffusion speed to account for the fact that this element is continuously use in high quantity by the plant and only add Potassium in small quantity with high diffusion speed for the fruiting stage when it could be required the most. Further, one may consider resorting to several entities of the same type, with different parameters. For instance, one may consider one `Cide` entity that acts on weeds and has a slow diffusion speed, and a second entity acting on insects with a high diffusion speed. Likewise one may consider a `Fertilizer` bringing nitrogen and another one bringing potassium.

Observations Each state variable of each entity can be measured, after incurring the corresponding cost defined in the score configuration file. We de not list them all as they follow the generic format for observation actions explained earlier. Interestingly, some entities such as the Weather entity offer the possibility to observe a (possibly noisy) forecast of some variables (e.g. air temperature). The forecast is made with noise linearly increasing with the look-ahead.

Notations Each entity defines a set of local variables, that together form the state of the system. In the sequel, we denote by $\mathcal{X}$ the state space, and by $x_t \in \mathcal{X}$ the state at time t. The state is thus made of several components, and we denote component j by $x_{j,t}$, where j could be a variable name for a specific entity, e.g. `plant-0.stage` for the stage of the first plant entity `plant-0` in the farm. We further denote by $\mathcal{X}_j$ the range of variable j. For convenience, we sometimes denote in the sequel `p['name']` a parameter of an entity with name `name`, `self.parameters['name']`, and `v['name']` a variable of an entity with name `name`. `self.variables['name']`.

Dynamics As many entities depend on some "favorable conditions" to transit from one state to another one, we find it convenient to introduce a generic function in order to model this principle easily in many entities. Formally, we introduce the following exponential general linear function

$$\mathcal{E}(\beta, y, \mathcal{Y}) = \exp(-\beta_0 - \sum_{j \in \mathcal{J}} \beta_j d(y_j, \mathcal{Y}_j)) \,, \tag{1}$$

where $\beta = (\beta_0, (\beta_j)_j)$ is a vector of weight parameters, $y = (y_j)_j$ is a vector of real-values, $\mathcal{Y} = (\mathcal{Y}_j)_j$ a set of intervals in $\mathbb{R}$, and $d(y, \mathcal{Y})$ denotes the distance from y to the set $\mathcal{Y}$. In the sequel, we make use of this model for to model most of stochastic transitions of the entities. While this choice is questionable from an accurate modeling perspective, it enables to model influences from external factors in a simple and modular way. In the entities we present below, we consider somewhat arbitrary sets of factors $\mathcal{J}$ for illustrative purpose. They can be easily changed, to add or remove other external factors.

4.1 The Weather

The Weather entity generates weather observations from actual weather data collected over one year, in a CSV format with six variables `Tmin`,`Tmax`,`T`,`RH`,`U`,`Rain`. These variables denote respectively the daily minimum, maximum and average temperature in Celsius, the relative humidity in percent, the average speed of the wind during the day, a discrete variable in $\{0;1\}$ indicating the occurrence of rain. They are recorded for a full year (365 days).

Variables and parameters The weather entity maintains several variables including the actual day of the year (`day#int365`), the air temperature, wind, sun, humidity index and rain information, as well as the number of consecutive most recent dry or frost days. Further, it provides also a noisy forecast of the weather variables for the next days. The actual weather at a given day is generated from the provided series

in the following way: the air temperature (`air_temperature`) is generated by adding some Gaussian noise to the CSV values. The wind speed and humidity index are generated similarly. The wind direction is generated randomly as well as the sun exposure. The module maintains these variables in its dictionary `variables`, in a way parameterized in the file `weather_specifications.yaml`.

Rain When rain occurs, the amount (light or heavy) of the rain is defined randomly with probability given by some parameter `rain_lightheavy_proba` $\in [0, 1]$. A heavy rains saturates the soil with water, putting it at maximum capacity, while a light rain fills it with random amount not more than half its capacity (see the soil entity). The intensity of the rain, between 0 and 1 is specified depending on whether the rain is light or heavy, by the parameters. The intensity has an effect on a possible leakage of nutrients in the soil, the value one creating maximal leakage (Leakage is also reduced by microlife soil activity, see the soil entity).

Forecast To obtain the forecasts for the next `forecast_lookahead` days we add a Gaussian noise to the daily variables, with a variance linearly increasing with the distance to the current day. More precisely, the variance of the noise at i^{th} day in the future for the air temperature is given by `parameters['air_temperature_noise']+parameters['forecast_noise']*i`. This creates a noisier and noisier forecast as the look-ahead increases.

Actions There is no intervention regarding the weather.

4.2 The Plant

In this section, we explain how is modeled the plant entity. We first provide some high-level descriptions of the variables, parameters and actions, we then detail the dynamics in the sub-sections below.

Our model account for a generic annual plant growing flowers once, producing fruit then dying. The growth of our model plant is divided in several stages: seed stage, growing stage, blooming stage, fruiting stage, ripe stage, and death. More formally, in the sequel, we consider a plant can either be in one of several development stages or be dead. The set of development stages is an ordered set

$$\mathbb{S} = \{\texttt{none}, \texttt{seed}, \texttt{grow}, \texttt{bloom}, \texttt{fruit}, \texttt{ripe}, \texttt{dead}\},$$

such that `grow` is the successor state of `seed` and so on. On top of that, from each state except `none`, it is possible to reach the state `dead`, according to some transition probability. In the subsections below, we detail each time both the transitions to the next state and to the `death` state in each stage.

There are plenty of ways to model the development of plants. We focus here on creating a simplified approach using *similar* modeling equations for most stages.

At each stage the plant evolves under the influence of the other variable of the system (e.g temperature, presence of pest insects, ...) according to a stochastic model. We model all these transition and stochastic evolutions using an exponential general linear model (1), specifying each tome the appropriate variables and ranges . Besides, within the stage `grow` and the stage `fruit`, one variable evolves according to some simple flow equations. This is the size of the plant in stage `grow` and the weight of the fruit in stage `fruit`, respectively. We make then evolve using a similar model (6).

Parameters The plant comes with a few parameters, `'initial_stage'` indicating the initial stage of a plant, the maximal size `'size_max#cm'`, number of flowers `flowers_max#nb`, and weight of a fruit `fruit_weight_max#g`. They also specify a shadow `'shadow_coeff#\%'` that is useful to compute soil evaporation, and a `pest_repulsive_effect#float` the affects the population of pests coming on the plant. Then, it specifies a number of parameters associated to stage specifying the conditions of development of the plant in this stage.

Variables The plant entity defines a large number of state variables. First, the variables related to the stage development of the plant:

Variable name	Type	Range
"stage"	Range	$\mathbb{S}$
"global_stage"	$X \times Y$ list	$\mathbb{S} \cup \{\texttt{undefined}\}$
"population#nb"	$X \times Y$ array	$(0, 100)$
"size#cm"	$X \times Y$ array	$(0, 100)$
"grow_size_threshold#cm"	$X \times Y$ array	$(0, 100)$
"flowers_per_plant#nb"	$X \times Y$ array	$(0, 100)$
"pollinator_visits#nb"	$X \times Y$ array	$(0, 100)$
"flowers_pollinated_per_plant#nb"	$X \times Y$ array	$(0, 100)$
"fruits_per_plant#nb"	$X \times Y$ array	$(0, 100)$
"fruit_weight#g"	$X \times Y$ array	$(0, 100)$
"fruit_weight_threshold#g"	$X \times Y$ array	$(0, 100)$
"harvest_weight#kg"	$X \times Y$ array	$(0, 100)$

The global_stage is defined, when 75% of plots are in the same stage, to be the corresponding stage, and otherwise is defined as undefined.

Then, it maintain variables regarding the amount of nutrients and water it received or not (stress).

Variable name	Type	Range
"cumulated_nutrients_N#g"	$X \times Y$ array	$(0, 100)$
"cumulated_stress_nutrients_N#g"	$X \times Y$ array	$(0, 100)$
"cumulated_nutrients_P#g"	$X \times Y$ array	$(0, 100)$
"cumulated_stress_nutrients_P#g"	$X \times Y$ array	$(0, 100)$
"cumulated_nutrients_K#g"	$X \times Y$ array	$(0, 100)$
"cumulated_stress_nutrients_K#g"	$X \times Y$ array	$(0, 100)$
"cumulated_nutrients_C#g"	$X \times Y$ array	$(0, 100)$
"cumulated_stress_nutrients_C#g"	$X \times Y$ array	$(0, 100)$
"cumulated_water#Lg"	$X \times Y$ array	$(0, 100)$
"cumulated_stress_water#L"	$X \times Y$ array	$(0, 100)$

Finally, it defines a set of counters. They count the amount of days spent in each stage, or from the last change in this stage.

Variable name	Type	Range
"age_seed#day"	$X \times Y$ array	$(0, 100)$
"consecutive_nogrow#day"	$X \times Y$ array	$(0, 100)$
"age_bloom#day"	$X \times Y$ array	$(0, 100)$
"consecutive_noweight#day"	$X \times Y$ array	$(0, 100)$
"age_ripe#day"	$X \times Y$ array	$(0, 100)$

Actions There are several actions attached to a plant entity. These are sow, harvest, micro-harvest, and remove. Action sow is parameterized by the plot position, the amount of seeds and spacings between them. Action micro-harvest, and remove are parameterized by the plot position only. Action harvest has no parameters and harvests the whole field.

4.2.1 SEED

This stage goes from the seed in the soil to the apparition of the very first leaves and roots.
Death probability The seed stays alive according to a Bernoulli distribution with parameter with probability $p_{\text{stay}}(t)$ given by the exponential linear model:

$$p_{\text{stay}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{appear}} \theta_j d(y_{j,t}, \mathcal{Y}_j))]\,, \tag{2}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
`Birds`	number of birds eating seeds	$[0, B_{\max}]$
`A`	`v['age_seed#day']`	`[p[sprout_age_min#day]`, $\infty)$

That is the plant stays in the same stage `seed` or moves to the stage `dead` according to $\mathcal{B}(p_{\text{stay}}(t))$.
Sprouting The seed then turns into a sprout (grow stage) according to a Bernoulli distribution with probability $p_{\text{stay}}(t)$ given by the exponential linear model:

$$p_{\text{sprout}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{appear}} \theta_j d(y_{j,t}, \mathcal{Y}_j))]\,, \tag{3}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
`T`	`weather.v['air_temperature']['mean#C']`	$[T_{\min}, T_{\max}]$
`RH`	`weather.v['humidity_index#%']`	$[RH_{\min}, RH_{\max}]$
`A`	`v['age_seed#day']`	`[p[sprout_age_min#day]`, $\infty)$

That is the plant stays in the same stage `seed` or moves to the stage `grow` according to $\mathcal{B}(p_{\text{stay}}(t))$.

4.2.2 GROWING

This stage goes from the first leaves (sprout) to the apparition of the first flowers. The plant grows by some rate that is governed by probability $p_{\text{rate}}(t)$ given by the exponential linear model:

$$p_{\text{rate}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{rate}} \theta_j d(y_{j,t}, \mathcal{Y}_j))\,, \tag{4}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
`T`	`weather.v['air_temperature']['mean#C']`	$[T_{\min}, T_{\max}]$
`N`	`soil.v['available_N#g']`	`[p['N_grow_consumption#g.mm-1']`, $\infty]$
`K`	`soil.v['available_K#g']`	`[p['K_grow_consumption#g.mm-1']`, $\infty]$
`P`	`soil.v['available_P#g']`	`[p['P_grow_consumption#g.mm-1']`, $\infty]$
`C`	`soil.v['available_C#g']`	`[p['C_grow_consumption#g.mm-1']`, $\infty]$
`W`	`soil.v['available_W#L']`	$[w, \infty]$

Here, w is the minimum amount of water the plant can obtain from the soil. This quantity depends on the weed sensibility to draught `draught` and wilting point of the soil, given by

$$w = ((1. - \texttt{draught}) \times \texttt{soil.parameters['wilting\_point\#L.m-3']} + \texttt{draught} \times \texttt{soil.parameters['max\_water\_capacity\#L.m-3']}) \times v \quad (5)$$

where $w = \texttt{soil.parameters['depth\#m']} \times \texttt{surface}$ is the volume of the plot soil.

Rate of growth Then, in case $r_t = \max(p_{\text{rate}}(t) + \xi, 0)$, where ξ is a Gaussian noise $\xi \sim \mathcal{N}(0, \sigma^2)$ is larger than some minimal growth rate `p['grow_rate_min#']` $\in [0, 1]$, then the plant grows, otherwise it does not grow. If the plant grows, then its size s_t =`variables['size#cm']` increases according to a simple flow equation

$$s_{t+1} = s_t + r_t \times \left(1 - \frac{s_t}{\texttt{size}_{\max}}\right)\sqrt{s_t} \quad (6)$$

In that case, since the sizes of the plant has increased, it also evaporates more water, which makes diminish its stored water by the corresponding amount.

When at least $x_{threshold} \in [0, 1]$ percentage of the maximum size is reached, the plant goes from stage `grow` to `bloom`, according to a Bernoulli with probability $p_{\text{bloom}}(t)$ given by the exponential linear model:

$$p_{\text{bloom}}(t) = \exp[-d(y_{j,t}, \mathcal{Y}_j)], \quad (7)$$

with variable given by

j	$y_{j,t}$	$\mathcal{Y}_j$
`size`	`variables['size#cm']`	$[x_{threshold} \times$ `p['size_max#cm']`$, \infty)$

That is, a Bernoulli $\mathcal{B}(p_{\text{bloom}}(t))$ is sampled to decide if the plant moves to stage `bloom`, or not (stays in state `grow`). The threshold $x_{threshold}$ depends of the stress received by the plant. If it has received a lot of stress it will stop its growth earlier than expected. Denoting by `stress` the total amount of nutrient and water stress the plant has received (the stress is the difference between required quantity and actual quantity provided by the environment)

$$x_{\texttt{threshold}} = (1 + \exp(-\texttt{stress}))/2. \quad (8)$$

Death probability The plant may stay alive according to a Bernoulli distribution with probability $p_{\text{stay}}(t)$ given by the exponential linear model:

$$p_{\text{stay}}(t) = \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{stay}} \theta_j d(y_{j,t}, \mathcal{Y}_j))], \quad (9)$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
`No-grow`	`variables['consecutive_nogrow#day']`	$[0, d_{\max}]$
`Pests`	number of pests on plant	$[0, P_{\max}]$

That is the plant stays in the same stage `grow` or moves to the stage `dead` according to $\mathcal{B}(p_{\text{stay}}(t))$.

4.2.3 BLOOMING

This stage goes from the first flowers to the fall of the flowers. Modelling this stage is difficult due to the vast amount of type of flowers and ways of blooming that we can find in nature. To simplify we approximate all the type of flowers to a simple stamen-pistil flowers (hermaphroditic type by opposition to the monoecious type where there are male and female flowers) assuming one flower gives one fruit. We also consider that the flower continuously blooms during this stage.

To determine the number of flowers that bloom per plant we use as a proxy the ratio between the final height of the plant and the maximum possible height. Indeed a plant that grows continuously without times of stress (due to temperatures, lack of water,.) that can hinder its growth will produce more flowers. To take that into account we propose the following formula for the number of flowers $n_{flowers}$:

$$n_{\texttt{flowers}} \sim \mathcal{B}(\texttt{maximum\_number\_of\_flowers}, \frac{\text{height of the plant}}{\text{maximum height}}) \tag{10}$$

Where `maximum_number_of_flowers` is a parameter that depends of the type of plant.

Pollination To produce fruits, flowers need to be pollinated. We keep track of the number of pollinated flowers with a variable `flowers_pollinated_per_plant#nb`. There are two types of pollination: self-pollination when the plant reproduces with itself and cross pollination when the plant reproduces with another plant. The last one can be carried by insects or wind. In all models presented hereafter the duration of blooming is not directly taken into account to model the success of pollination (its effect is unclear for the wide range of species considered)

For the number of flowers successfully pollinated at time step t, n_t^{po}, we resort to the following formula using binomial law:

$$\left\{ n_t^{po} = \quad w_{\text{auto}}\mathcal{B}(n_t^{npf}\ p_{\text{auto}}) + w_{\text{wind}}\ \mathcal{B}(n_t^{npf}, p_{\text{wind}}) + w_{\text{insect}}\ \mathcal{B}(n_t^{npf}, p_{\text{insect}}) \right. \tag{11}$$

that is, we increase `flowers_pollinated_per_plant#nb` by n_t^{po}, where n_t^{npf} is the number of remaining non pollinated flowers at time step t, $w_{...}$ denotes the proportion of the pollination of the plant [auto, wind, insect] such that these proportions sum up to 1 and $p_{...}$ denotes the probability of success of this type of pollination. We now detail how the different pollination probabilities are computed.

The auto-pollination success, p_{auto}, is a constant parameter that depends on the type of plant. We model p_{wind} the success of wind pollination assuming that the plant in a field big enough to ensure the position of the plant does not affect pollination. Moreover, we suppose that the direction of the wind is uniformly random for all directions and its speed does not influence the pollination because the density of plant is high enough. Thus, only the temperature that condition the opening of flowers can influence pollination. We model p_{wind} as an exponential linear model

$$p_{\text{wind}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{wind}} \theta_j d(y_{j,t}, \mathcal{Y}_j))]\,, \tag{12}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
T	`weather.v['air_temperature']['mean#C']`	$[T_{\min}, T_{\max}]$

The success of pollination by the insects is directly related to the number of visits, so we model p_{insect} as an exponential linear model

$$p_{\text{insects}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{insects}} \theta_j d(y_{j,t}, \mathcal{Y}_j))]\,, \tag{13}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
visits	number of pollinator insects occurrence	[p['pollinator_visits_min'], ∞)

Fruit probability The plant may move from bloom to fruit stage according to a Bernoulli distribution with probability $p_{\text{fruit}}(t)$ given by the exponential linear model:

$$p_{\text{fruit}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{stay}} \theta_j d(y_{j,t}, \mathcal{Y}_j))] \,, \tag{14}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
Flower-age	weather.v['age_bloom#day']]	[p['bloom_duration#day'], ∞)

That is the plant stays in the same stage bloom or moves to the stage dead according to $\mathcal{B}(p_{\text{stay}}(t))$.

Death probability The plant may stay alive according to a Bernoulli distribution with probability $p_{\text{stay}}(t)$ given by the exponential linear model:

$$p_{\text{stay}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{stay}} \theta_j d(y_{j,t}, \mathcal{Y}_j))] \,, \tag{15}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
Frost	weather.v['consecutive_frost#day']]	$[0, d_{\max}]$

That is the plant stays in the same stage bloom or moves to the stage dead according to $\mathcal{B}(p_{\text{stay}}(t))$.

4.2.4 FRUITING

This stage goes from the fall of flowers to the full grown fruits. Here, our model considers that all flowers pollinated at the end of the previous stage will become fruits with same final weight. Hence, we only compute the dynamic of growth of one fruit. The number of fruits per plant fruits_per_plant#nb is initialized with the number of flower pollinated flowers_pollinated_per_plant. The weight of a fruit grows at some rate that is governed by probability $p_{\text{rate}}(t)$ given by the exponential linear model:

$$p_{\text{rate}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{rate}} \theta_j d(y_{j,t}, \mathcal{Y}_j)) \,, \tag{16}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
T	weather.v['air_temperature']['mean#C']	$[T_{\min}, T_{\max}]$
N	soil.v['available_N#g']	[p['N_grow_consumption#g.mm-1'], ∞]
K	soil.v['available_K#g']	[p['K_grow_consumption#g.mm-1'], ∞]
P	soil.v['available_P#g']	[p['P_grow_consumption#g.mm-1'], ∞]
C	soil.v['available_C#g']	[p['C_grow_consumption#g.mm-1'], ∞]
W	soil.v['available_W#L']	$[w, \infty]$

where w is the minimum amount of water the plant requires, see (5).

Rate of growth Now, in case $r_t = \max(p_{\text{rate}}(t) + \xi, 0)$, where ξ is a Gaussian noise $\xi \sim \mathcal{N}(0, \sigma^2)$ is larger than some minimal growth rate `p['weight_rate_min#']` $\in [0, 1]$, then the fruit gains weight, otherwise it does not. If the fruit gains weight, then its weight w_t =`variables['fruit_weight#g']` increases according to a simple flow, similar to that governing the size evolution of the plant (6)

$$w_{t+1} = w_t + r_t \times \left(1 - \frac{w_t}{\text{weight}_{\max}}\right)\sqrt{w_t} \tag{17}$$

When at least $x_{threshold\_w}$ percentage of the maximum weight is reached, the plant goes from stage `fruit` to `ripe`, according to a Bernoulli with probability $p_{\text{ripe}}(t)$ given by the exponential linear model:

$$p_{\text{ripe}}(t) \quad = \quad \exp[-d(y_{j,t}, \mathcal{Y}_j)]\,, \tag{18}$$

with variable given by

j	$y_{j,t}$	$\mathcal{Y}_j$
`weight`	`variables['fruit_weight#g']`	$[x_{threshold\_w} \times$ `p['fruit_weight_max#g']`$, \infty)$

That is, a Bernoulli $\mathcal{B}(p_{\text{ripe}}(t))$ is sampled to decide if the plant moves to stage `ripe`, or not (stays in state `fruit`). Similarly to the dynamics for the `size`, the threshold $x_{threshold}$ depends of the stress received by the plant. If it has received a lot of stress it will stop its growth earlier than expected. Likewise, the quality of the pollination may affect this value, especially for plants that depend a lot on insect pollination. Denoting by `stress` the total amount of nutrient and water stress the plant has received (the stress is the difference between the required quantity and actual quantity provided by the environment), it comes

$$x_{\texttt{threshold}} = \exp[-\sum_{j \in \mathcal{J}_{\texttt{threshold}}} \theta_j d(y_{j,t}, \mathcal{Y}_j)] \tag{19}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
`stress`	`stress`	$[0, \infty)$
`pollinators`	`v['pollinator_visits#nb']`	[`p['fruit_pollinators_min#nb']`$, \infty)$

Death probability The plant may stay alive according to a Bernoulli distribution with probability $p_{\text{stay}}(t)$ given by the exponential linear model:

$$p_{\text{stay}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{stay}} \theta_j d(y_{j,t}, \mathcal{Y}_j))]\,, \tag{20}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
`No-grow`	`v['consecutive_noweight#day']`	$[0, d_{\max}]$
`RH`	`weather.v['humidity_index#%']`	$[RH_{\min}, RH_{\max}]$
`Pests`	number of pests on plant	$[0, P_{\max}]$

That is the plant stays in the same stage `fruit` or moves to the stage `dead` according to $\mathcal{B}(p_{\text{stay}}(t))$.

4.2.5 RIPE

This stage goes from the fall of the full grown fruits to either the harvest by the farmer or the death of the plant. The idea behind the different models is that the longer the farmer wait to harvest the most probable a fruit is lost (due to weather conditions, pest, ...).

The number of fruits per plant `variables['fruits_per_plant#nb']` decreases by a multiplicative factor at each time step. This factor is given by $r_t^{\text{decay}} = \max(p_{\text{decay}}(t) + \xi, 0)$, where ξ is a Gaussian noise $\xi \sim \mathcal{N}(0, \sigma^2)$. The probability $p_{\text{decay}}(t)$ given by the exponential linear model:

$$p_{\text{decay}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{\text{decay}}} \theta_j d(y_{j,t}, \mathcal{Y}_j))]\,, \tag{21}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
`Rain`	$\mathbb{I}\{$`weather.v['rain_amount']`$\}$	$(-\infty), 0]$
`Frost`	`weather.v['consecutive_frost#day']`	$[0, F_{\max}]$
`Pests`	number of pests on plant	$[0, P_{\max}]$
`Age ripe`	`v['age_ripe#day']`	$[0, d_{\max}]$

Hence, the number n_t^f of fruits per plant becomes $n_f^t = \lfloor r_t^{\text{decay}} \times n_f^{t-1} \rfloor$.

Death Then the number of fruits per plant reaches 0, the plant moves from stage `ripe` to stage `dead`.

4.2.6 DEAD

When the plant is dead, we do not consider it disappears from the plot. Instead, it stops eating nutrients and starts releasing them in the soil (see soil entity). When the total amount of nutrients in the plant reaches 0, then it is considered gone, hence transits to state `none`.

4.2.7 WATER CONSUMPTION

Finally, in each stage of the plant, due to evapotranspiration, the plant looses some water. The amount of water that is evaporated depends on the reference evaporation constant $\texttt{ET}_0$ at this latitude (in $mm.m^{-2}.day^{-1}$), that is the amount of water in mm that evaporates per square meter of surface under the sun per day. We detail its formula in (23) in the soil Section below. Building on suggestions from the FAO, see Valiantzas (2018), we consider the amount of water evaporated by a plant of size h (in cm) on day t, takes the form $\texttt{ET}_0 \times \texttt{K}_t/100$, with

$$\texttt{K}_t = (\alpha_0 + h \times \alpha_1 + (0.04 \times (u-2) - 0.004 \times (rh - 45)) \times ((h/300)^{0.3}))$$

where u denotes the wind speed `weather.variables["wind"]["speed#km.h-1"]`, rh the relative humidity index `weather.variables['humidity_index#\%']` and α_0, α_1 are parameters of the plant. This is a simplification of the model considered by Valiantzas (2018), that still conveys some key features, such the influence of the wind and height of the plant. This amount is then removed from the plant water storage `variables['cumulated_water#L']`. It is also used to compute the amount of water that the plant requires from the ground at each time step.

4.3 The Soil

The plot is the scale at which the water and nutrient cycle are managed, plus the possible presence of (herbi/pesti) cides. A subdivision of the field that should be seen as a small area surrounding the plant. Our

model assumes that each plot is independent of the others, so there is no exchange of water or nutrients by diffusion between plots.

Parameters The parameters of a soil are its depth `'depth#m'`, several parameter related to the water: the maximal storage capacity of the soil `'max_water_capacity#L.m-3'`, the minimal proportion of water below which no plant can absorb water `'wilting_point#L.m-3'`, and the speed at which water put on the surface moves inside the soil `'water_surface_absorption_speed#m2.day-1'`. The soil also releases some nutrients at a small speed, due to interaction between the soil microlife and the bedrock. They are gathered in parameters `'bedrocks_release_N#mg.day-1`, and similar ones for P,K and C.

Variables

Variable name	Type	Range
`"available_N#g"`	$X \times Y$ array	$(0, 10^5)$
`"available_P#g"`	$X \times Y$ array	$(0, 10^5)$
`"available_K#g"`	$X \times Y$ array	$(0, 10^5)$
`"available_C#g"`	$X \times Y$ array	$(0, 10^5)$
`"available_Water#L"`	$X \times Y$ array	$(0, 10^5)$
`"wet_surface#m2.day-1"`	$X \times Y$ array	$(0, 10^5)$
`"microlife_health_index#%"`	$X \times Y$ array	$(0, 10^6)$
`"amount_cide#g"`	Dict of $X \times Y$ array	$(0, 10^6)$

Dynamics The dynamics models the inputs and outputs for the water, the nutrients and cides. Crucially, it also models the microlife health dynamics of the soil.

WATER INPUTS The water receives water from the rain, or by a watering action of the learner. Rains can be either light or heavy. A light rain is considered to bring an amount of water at most half the storage capacity of the soil C_{water}, uniformly randomly. That is a random amount $\mathcal{U}([0, 0.5 \times C_{\text{water}}]$, where $\mathcal{U}$ denotes the uniform distribution is brought by the rain. A heavy rain is considered to bring an amount equal to C_{water}. Now, if the soil already contains water, the water input may exceeds the capacity of the soil. This creates a water surplus. This surplus may in turn create some nutrients leeching and damage microlife health of the soil (see below). In both cases, the entire surface of the plot is considered wet.

NUTRIENTS INPUTS The bedrocks releases nutrients N,K,P,C everyday proportionally to the microlife health index. If N_t denotes the amount of nutrient N available at time t in grams, we thus have

$$N_{t+1} = N_t + \frac{\texttt{v['microlife\_health\_index\#\%']}}{100} \frac{\texttt{p['bedrocks\_release\_N\#mg.day-1']}}{1000}.$$

This quantity is further increased by the presence of fertilizers, depending on their release speed for each nutrient. This is obtained by the method `release_nutrients` from the Fertilizer entity.

Last, a plant may further release nutrients. This typically happens when the plant is dead and not removed from a plot. The amount released by a plant is given by `release_nutrients`.

MICROLIFE As fertilizers may release continuously some nutrients, likewise, cides release cides continuously overtime. The amount of cides in the soil may hence increase. This is obtained by the method `release` from the Cide entity. The amount of cides directly affects the microlife health index negatively. Further, an excess of water may also affects microlife health (by removing oxygen and killing part of the microlife). We model the probability that soil stays alive introducing $p_{\text{stay}}(t)$ at time t, given by the exponential linear model:

$$p_{\text{stay}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{\text{stay}}} \theta_j d(y_{j,t}, \mathcal{Y}_j))], \tag{22}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
Cide	variables['amount_cide#g']['soil']	$(-\infty, 0]$
Water	water_surplus/max_capacity	$(-\infty, 0]$

where water_surplus is the amount of water that the soil received above its max-capacity. This is typically due to rain, or too much watering. We then define the flow governing the dynamics of the microlife health μ as

$$\mu_{t+1} = \left(p_{\text{stay}}(t) \times (1 + 0.1 p_{\text{stay}}(t)) + (1 - p_{\text{stay}}(t)) p_{\text{stay}}(t) \right) \mu_t$$

that is, with probability $1 - p_{\text{stay}}(t)$, the microlife dies, and is reduced by a fraction to $p_{\text{stay}}(t)\mu_t$, while with probability $p_{\text{stay}}(t)$, it stays alive, hence slowly increases by an amount $0.1 p_{\text{stay}}(t)\mu_t$. This models the fact the soil can recover provided there is some microlife and no aggression.

NUTRIENTS OUTPUT Then, the exchange with the plants and weeds is considered. Each weed and plant is asked what amount of each nutrient it requires using method requirement_nutrients from both Plant and Weeds entities. However, to model the fact nutrients are made available to the plant by the microlife, we consider that when the plant asks for some quantity of nutrients, it requests to the soil only a fraction variables['microlife_health_index#\%'] of this amount. Hence, unless with a fully healthy soil, the plant will always receive a smaller amount of nutrients that what it requires, causing some stress. Now, if the requested amount is below the stored amount in the soil, it is depleted from the soil. If it exceeds, then a stress appears, as only part of the requested nutrients can be sent to the plant. The amount of nutrients and stress is then sent back to the plant dynamics, using method receive_nutrients.

A water surplus may cause some leeching, as well as rain. This removes some nutrients. Introducing the quantity $q = \texttt{rain}_\lambda \times \texttt{surface} + \frac{\texttt{water\_surplus}}{\texttt{max\_capacity}}$, where $\texttt{rain}_\lambda \in [0, 1]$ is the intensity of the rain and surface the area of the surface of the plot. The amount of each nutrients N, P, K, C removed is given $q \times (1 - \mu_t)$ and is proportionally to $1 - \mu_t$. Hence, the more microlife health, the less nutrients leaks due to water excess. A similar effect happens for cides, with however stronger intensity, as the amount of cide is multiplied by $\exp(-q)$.

WATER OUTPUT Like nutrients, water is absorbed by plants and weeds. However, water absorption can only be done above the wilting point, hence the effective amount of water in the soil is the amount of water minus the wilting point times the volume of soil of the plot.

Now, unlike nutrients, we model the fact that water evaporates. The precise evaporation dynamics depends on weather conditions, shadows on the ground, and how wet is the surface of the soil.

The ground evaporation in a plot is computed based on the reference evaporation per m^2 per day $\texttt{ET}_0$ (in $mm.m^{-2}.day^{-1}$) at that latitude, times the surface of the plot, times the proportion of the surface that is concerned by evaporation. To evaporate water, a part of the plot should be wet, and also under the sun, not shadow. The wet fraction of the plot is given by $\texttt{wet} = \texttt{v['wet\_surface\#m2.day-1']/surface}$. The fraction of plot that is under the sun is 1 minus the fraction under shadow. The shadow depends on the plants and weeds on the plot. Each plant creates a shadow proportional to its size and depending on its shadow coverage parameter. Finally, the amount of water evaporated at day t is given by

$$\texttt{ET} = \texttt{ET}_0 \times \min(1. - \texttt{shadow}, \texttt{wet}) \times \texttt{surface}$$

On top of this water evaporation, we model the fact that different soil may retain the water differently. To do so, we remove some water everyday, given by $(1.1 - \mu) \times p[\texttt{'water\_leakage\_max\#L.m-3.day-1'}] \times V$ where $\mu \in [0, 1]$ denotes the microlife health index of the soil and V the volume of the soil.

We detail now the calculation of $\texttt{ET}_0$ using the model presented in the FAO book Crop evapotranspiration (in particular chapters 4 and 7). To evaluate ET_0 at time t, several models exist. We do not use the Penman-Monteith equation suggested by the FAO (chapter 3) as it requires too many parameters. Instead, we use the simplified approximation formula given in Valiantzas (2018)

$$ET_{0_t} \approx 0.018(1-\frac{RH_t}{100})^{0.2}(T_{max_t}-T_{min_t})^{0.3}(R_{A_t}\sqrt{T_t+10}-40)+0.1(T_t+20)(1-\frac{RH_t}{100})(\frac{U_t}{2})^{0.6} \quad (23)$$

Where RH is relative humidity measure in the atmosphere in percent (around 1 meter above the field), $T = \frac{T_{max}+T_{min}}{2}$, R_A is the extraterrestrial radiation in $Mj.m^{-2}.day^{-1}$ (depending of the latitude and month, but there exist some tables of it anyway) and U is the wind speed.

We can obtain R_A with tabulated value or by calculating it form the day of the year ($J \in [0, 365]$) and the latitude φ in degree.

$$R_A = \frac{24 \times 60}{\pi} G_{sc} d_r [\omega_s \sin(\frac{\pi}{180}\varphi)\sin(\delta) + \cos(\frac{\pi}{180}\varphi)\cos(\delta)\sin(\omega_s)] \quad (24)$$

$$\begin{cases} G_{sc} = 0.0820\ Mj.m^{-2}.minute^{-1} & \text{solar constant} \\ d_r = 1 + 0.033\cos(\frac{2\pi}{365}J) & \text{inverse relative distance Earth-Sun} \\ \delta = 0.409\sin(\frac{2\pi}{365}J - 1.39) & \text{solar decimation} \\ \omega_s = \arccos(-\tan(\frac{\pi}{180}\varphi)\tan(\delta)) & \text{sunset hour angle} \end{cases} \quad (25)$$

Note that each plant is also subject to evapotranspiration. The quantity $\texttt{ET}_0$ is directly used to compute the amount of water that a plant evapotranspirate, see the plant entity.

Actions The actions possible is to water the soil. There is one discretized version with a discretized amount, and a continuous version with a continuous amount. In each case, one must also specify the plot position `'plot'` and duration `'duration#min'` of the watering action in minutes. When watering is done on a plot, we consider the entire soil surface of the soil is wet for the duration of the watering.

4.4 Other entities

It is at the field level that we manage phenomena that involve move between plots, such as the move of pest insect, pollinator swarms and the spread of weeds.

4.4.1 Birds

We briefly explain below how a Bird entity is working.

Parameters A bird instance is specified by four parameters. First integer `max_population` indicating the maximum of birds possible on a field, then three Boolean parameters defining whether the Bird entity is a `seed_eater`, a `pest_eater` and/or a `pollinator_eater`.

Variables The Bird entity maintains the population of birds in a variable `population#nb`.

Variable name	Type	Range
`"population#nb"`	Range	$(0, 10^5)$
`"total_cumulated_birds#nb"`	Range	$(0, 10^6)$

Dynamics The population of birds is simply randomly generated each day uniformly within its range. It is then reduced by the possible presence of a scarecrow (see entity Facility), in a way that depends on the strength of the scarecrow.

Actions There is no intervention regarding the birds.

4.4.2 WEEDS

We consider the model of weeds to be similar to that of a plant, with some key differences. On the one hand, we consider a simplified model, where weeds starts from seeds, grow, make flowers, and produce new seeds. On the other hand, we maintain a population of weeds on a plot, some of which could be in stage seeds, while others are making flowers. Hence, we consider the dynamics of the number of weeds in each stage. Weeds are more resilient, mostly grow due to available water and nutrients, and are not sensitive to stress. They can die due to herbicide, being removed or bad weather/nutrients conditions. In terms of dynamics, weeds comes from the edge of the field, and produce seeds. When a weed reach flowering stage, it then turns into seeds and scatter seeds around in near-by plots. The seeds turn into weeds when proper temperature and nutrient conditions, disregarding water.

Parameters The parameter of a Weed entity are grouped into appear conditions and grow conditions, plus the number of flowers per plant and the max number of (effective) seeds it can produce.

Variables

Variable name	Type	Range
`grow#nb`	$X \times Y$ Array	$(0, 1000)$
`seeds#nb`	$X \times Y$ Array	$(0, 1000)$
`flowers#nb`	$X \times Y$ Array	$(0, 1000)$

Dynamics We detail each stage transition below.

Appearance of seeds from the edge follow a binomial distribution with parameter $n =$ `max_new_seeds#nb`, and probability $p_{\text{appear}}(t)$ at time t, given by the exponential linear model:

$$p_{\text{appear}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{appear}} \theta_j d(y_{j,t}, \mathcal{Y}_j))] \,, \tag{26}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
D	`distance_to_edge`	$(-\infty, 0]$

That is, a random variable is sampled from $\mathcal{B}(n, p_{\text{appear}}(t))$, that is added to `variables["seeds#nb"]`.

The population of weeds in `seed` stage turn into `grow` stage according to a distribution with parameter $n =$ `variables["seeds#nb"][x, y]` and probability $p_{\text{grow}}(t)$ given by the exponential linear model:

$$p_{\text{grow}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{\text{grow}}} \theta_j d(y_{j,t}, \mathcal{Y}_j))] \,, \tag{27}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
T	`weather.v['air_temperature']['mean#C']`	$[T_{\min}, T_{\max}]$
RH	`weather.v['humidity_index#%']`	$[RH_{\min}, RH_{\max}]$
H	`soil.v['amount_cide#g']['weeds']`	$(-\infty,$ `p[grow_herbicide_max#g]`$]$

That is, a random variable is sampled from $\mathcal{B}(n, p_{\text{grow}}(t))$, that is added to `variables["grow#nb"]` and subtracted from `variables["seeds#nb"]`.

The population of weeds in `grow` stage turn into `bloom` stage according to a binomial with $n =$ `variables["grow#nb"][x, y]` and probability $p_{\text{bloom}}(t)$ given by the exponential linear model:

$$p_{\text{bloom}}(t) \quad = \quad \exp[-\theta_0 - \theta_{\text{H}} d(y_{\text{H},t}, \mathcal{Y}_{\text{H}}]\,. \tag{28}$$

That is, a random variable is sampled from $\mathcal{B}(n, p_{\text{bloom}}(t))$, that is added to `variables["flowers#nb"]` and subtracted from `variables["grow#nb"]`. Further, the population that stays alive follows a binomial distribution with parameter $n =$ `variables["grow#nb"][x, y]` and probability $p_{\text{stay}}(t)$ given by the exponential linear model:

$$p_{\text{stay}}(t) \quad = \quad \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{appear}} \theta_j d(y_{j,t}, \mathcal{Y}_j))]\,, \tag{29}$$

with variables given by

j	$y_{j,t}$	$\mathcal{Y}_j$
T	`weather.v['air_temperature']['mean#C']`	$[T_{\min}, T_{\max}]$
H	`soil.v['amount_cide#g']['weeds']`	$(-\infty$, `p[death_herbicide_max#g]`$]$
N	`soil.v['available_N#g']`	[`p['N_grow_consumption#g.mm-1']`, ∞]
K	`soil.v['available_K#g']`	[`p['K_grow_consumption#g.mm-1']`, ∞]
P	`soil.v['available_P#g']`	[`p['P_grow_consumption#g.mm-1']`, ∞]
C	`soil.v['available_C#g']`	[`p['C_grow_consumption#g.mm-1']`, ∞]
W	`soil.v['available_W#L']`	$[w, \infty]$

where `p[death_herbicide_max#g]` denotes the maximum amount of pesticide the weed can tolerate and w is the minimum amount of water the plant requires. This quantity depends on the weed sensibility to draught `draught` and wilting point of the soil, given by (5) exactly as for the plants. That is, a random variable is sampled from $\mathcal{B}(n, p_{\text{stay}}(t))$, that replaces `variables["grow#nb"]`.

Finally, the population of weeds in `bloom` stage turns into `seeds` according to a binomial distribution, with parameter $n =$ `variables["flowers#nb"][x, y]` of flowers, and probability `parameters['flower_to_seed#\%']` that indicates the percentage of flowers turned into seeds. Then, each one produces `p['seed_per_flower#nb']` may seeds, The total population of seeds on the plot is then scattered uniformly randomly on the neighborhood of the plot (including the plot).

That is, a random variable is sampled from $\mathcal{B}(n,$ `parameters['flower_to_seed#%']`$)$ that is and subtracted from `variables["flowers#nb"]`, and then split in random parts added to `variables["seeds#nb"]` at current and neighbouring positions.

Actions There is one possible action, that is `remove`, parameterized by the considered plot location. This action removes all weeds in `grow` or `bloom` stage on the plot, but not the ones in seed stage.

4.4.3 Pest Insects

We briefly explain below how a Pests entity is working.

Pest insects arrived by successive waves in the field from the wild edge and days after days can progress from a plot to another one or die.

The effective appearance of pests on plot is influenced by several factors, including distance to the edge (the further, the less probable), weather conditions, and the presence of pesticides. Likewise, pests may leave a plot, depending on weather condition, presence of pest-eaters (e.g. birds), and pesticides.

Further, the pests move into the field from plots to neighbor plots, influenced by the wind direction.

Last, in each plot, the population of pests splits between the plants and potential weeds, proportionally to their number, and weighted by the repulsive/attractive effect of each plant or weed.

Parameters Arrival frequency, minimum and maximal number of pests arrival each day, appear and leave conditions parameters.

Variables

Variable name	Type	Range
`plot_population#nb`	$X \times Y$ Array	$(0, 1000)$
`onplant_population#nb`	Dict of $X \times Y$ Array	$(0, 1000)$ per plant

Dynamics The population of pests at the edge arrives in bursts ranging in between a minimal and maximal value, at the given frequency f. At day t, it is specified as

$$n_t^{\text{edge}} \sim U([0, p_{max} - p_{min}])\mathbb{I}\{o_t\} \text{where} o_t \sim \mathcal{B}(f), \tag{30}$$

with parameters

- $f =$`parameters['arrival_frequency#day-1']`,
- $p_{min} =$`parameters['min\_population\#nb']`,
- $p_{max} =$`parameters['max\_population\#nb']`.

The dynamics of pest insects between a plot and the edge is based on on binomial laws, and define the number of pests appearing and leaving on each plot as follows

$$\begin{cases} n_t^{appear} \sim \mathcal{B}(n_t^{\text{edge}}, p_{appear}(t)) \\ n_t^{pest} \sim \mathcal{B}(n_{t-1}^{pest}, p_{stay}(t)) + n_t^{appear} \end{cases} \tag{31}$$

where $n_{t-1}^{pest} \in \mathbb{N}$ is the total number of pests, that is `plot_population#nb` at previous day. The probabilities $p_{appear}(t)$, $p_{stay}(t)$ follow an exponential linear model. More precisely, appear conditions in a specific plot depend on the distance to the edge of the field, the mean air temperature, and the amount of pesticide on a plot. Likewise, stay conditions depend on the distance to the edge of the field, the mean air temperature, the number of pest-eaters (birds) and the amount of pesticide on a plot.

$$p_{appear}(t) = \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{appear}} \theta_j d(y_{j,t}, \mathcal{Y}_j))]\,. \tag{32}$$

$$p_{stay}(t) = \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{leave}} \theta_j d(y_{j,t}, \mathcal{Y}_j))]\,. \tag{33}$$

The quantities appearing for each probability are summarized below.

j	$y_{j,t}$	$\mathcal{Y}_j$
D	distance_to_edge	$(-\infty, 0]$
T	weather.v['air_temperature']['mean#C']	$[T_{\min}, T_{\max}]$
Pesticide	soil.v['amount_cide#g']['pests']	$(0, P_{\max}]$
D	distance_to_edge	$[\max(X, Y)/2, \infty]$
T	weather.v['air_temperature']['mean#C']	$[T_{\min}, T_{\max}]$
Pesticide	soil.v['amount_cide#g']['pests']	$(0, P_{\max}]$
Birds	number of birds eating pests	$(0, B_{\max}]$

Now, part of the population of pest insects in a plot can randomly move to the four cardinal directions $\{\texttt{north}, \texttt{south}, \texttt{east}, \texttt{west}\}$ or `stay` on the plot. This random walk is weighted uniformly in all directions except in the direction of wind, proportionally to its speed. More precisely, the number of insects moving to each direction `i` is proportional to $p^{\texttt{i}}/P$ where $p^{\texttt{i}} \sim \mathcal{U}([0,1])$ if i is not in the direction of the wind, and $p^{\texttt{i}} \sim \mathcal{U}([0,1]) + \texttt{weather.v["wind"]['speed\#km.h-1']}/10$ otherwise, and $P = p^{\texttt{north}} + p^{\texttt{south}} + p^{\texttt{east}} + p^{\texttt{west}} + p^{\texttt{stay}}$ is the normalization factor.

Last, within a plot, the population of pests distributes between the different plants and weeds on this plot, weighted by their repulsive affect. This specifies `onplant_population#nb`. Given a list L of (alive) plants and weeds on plot (x,y), each one ℓ is associated with a weight that is equal to $w_\ell = \exp(-\texttt{pest\_repulsive\_effect}_\ell)/W$, where W is a normalization factor. The population on each plant/weed follows a multinomial distribution parameterized by $n_t^{\texttt{pest}} = \texttt{plot\_population\#nb}$, the total population on the plot, and the L weights. That is, $\texttt{onplant\_population\#nb} \sim \text{Mult}(n_t^{\texttt{pest}}, (w_\ell)_{\ell \in L})$.

Actions There is no action associated with the pests.

4.4.4 POLLINATORS

We briefly explain below how a Pollinator entity is working. The pollination insects (bees are in mind but other insects can be describe like that), move every day from their resting place (out of the field), collect some nectar and pollen doing several round trips. We model the occurrence of a visit rather than the actual number of pollinators. They penetrate in each field depending on their preference about the plant cultivated (for instance bees love rapeseed flowers, thus their density is more or less constant even in the centre of the field). All fields are surrounded by a wild edge where the density of pollinating insect is constant. Moreover pollinating insects are sensitive to temperature, wind, rain and insecticides (to which they may be more or less sensitive). Thus we model the probability of presence of a pollinators in a plot follow a Bernoulli's law $\mathcal{B}(p_{visit})$ where p_{visit} is described using a GLM with weights.

Parameters To specify a pollinator entity, one must define some quantities that affect the visiting probability of a plot by a pollinator. These quantities are grouped under the parameter `'visit_conditions'`.

Variables

Variable name	Type	Range
`occurence#bin`	$X \times Y$ Array	{True, False}

Dynamics The dynamics consists, in each plot (x, y) of the field to draw a Boolean random variable that defines the occurrence of a visit from a pollinator. The probability of occurrence is modeled using an exponential linear model, and computed at day t according to

$$p_{occur}(t) = \exp[-(\theta_0 + \sum_{j \in \mathcal{J}_{occur}} \theta_j d(y_{j,t}, \mathcal{Y}_j))] \,. \tag{34}$$

The variables at hand in $\mathcal{J}_{occur} = \{\texttt{D}, \texttt{T}, \texttt{Wind}, \texttt{Birds}, \texttt{Rain}, \texttt{Pesticide}\}$ are the distance `D` to the edge of the field, the mean air temperature `T`, the wind speed `Wind`, the occurrence of rain `Rain` and the amount of pesticide `Pesticide` present on the plot. The parameter `'visit_conditions'` specifies the value of each θ_j, and the corresponding favorable range of values $\mathcal{Y}_j$. They are given as $\mathcal{Y}_{\texttt{Wind}} = [0, W_{\max}]$, $\mathcal{Y}_{\texttt{Pesticide}} = [0, P_{\max}]$, $\mathcal{Y}_{\texttt{Birds}} = [0, B_{\max}]$, $\mathcal{Y}_{\texttt{T}} = [T_{\min}, T_{\max}]$. In particular, the user must specify the minimal and maximal values $T_{\min}, T_{\max}$ of the mean air temperature corresponding to optimal visit conditions, and the maximal number of pollinator-eating bird $B_{\max}$, maximal wind speed $W_{\max}$ and maximal amount of pesticide $P_{\max}$ it can tolerate. This is summarized in the table below

j	$y_{j,t}$	$\mathcal{Y}_j$
`D`	`distance_to_edge`	$(-\infty, 0]$
`T`	`weather.v['air_temperature']['mean#C']`	$[T_{\min}, T_{\max}]$
`Wind`	`weather.v['wind']['speed#km.h-1']`	$(0, W_{\max}]$
`Birds`	number of birds eating pollinator	$(0, B_{\max}]$
`Rain`	$\mathbb{I}\{$`weather.v['rain_amount']`$\}$	$(-\infty, 0]$
`Pesticide`	`soil.v['amount_cide#g']['pollinators']`	$(0, P_{\max}]$

Actions There is no action associated with pollinators.

4.4.5 Cides

We briefly explain below how a Cides entity is working. Cides model chemicals that kill life. This includes mainly Herbicides and Pesticides, with negative consequences on the soil microlife as well.

Parameters To specify a cide entity, one should indicate the degree to which it affects `'weeds'`, `'pollinators'`, `'pests'`, and `'soil'`. Additionally, a cide entity, when released, has a base absorption speed specified by `'base_absorption_speed#kg.week-1'`, and it is released in bags of a certain amount `'bag#kg'`.

Variables

Variable name	Type	Range
`'amount#kg'`	$X \times Y$ Array	$(0, 10^3)$
`'total_cumulated_scattered_amount#kg'`	$X \times Y$ Array	$(0, 10^4)$

Dynamics The dynamics consists in releasing the amount of cide in the soil progressively over time, at the speed specified by `'base_absorption_speed#kg.week-1'`.

Actions There are two related actions. First `'scatter'` parameterized by the plot position in the field `plot` and the (continuous) amount scattered `'amount#kg'`. Then a `'scatter_bag'` parameterized by the plot position in the field `plot` and the (discrete) amount of bags scattered `'amount#bag'`.

4.4.6 Facilities

We briefly explain below how a Facility entity is working.

Parameters To instantiate a facility, one must specify one parameter, `scarecrow_strength` that specifies how many birds a scarecrow may deter.

Variables

Variable name	Type	Range
`"scarecrow"`	Range	['none','basic','advanced']

Dynamics There is no dynamics. However, the presence of a scarecrow affects the dynamics of birds.

Actions There are two actions. First, an action `put_scarecrow`, parameterized by the type of scarecrow, either `'basic'` or `'advanced'`. Then, an action `remove_scarecrow`.

5. The atari of farming

In developing `Farm-gym` , we aim to create an environment that can mimic the dynamics of an agricultural system while remaining simple enough. One of the strengths of our environment its modularity. We can choose indeed which dynamics of the environment we want to simulate, we can combine these dynamics to make appear coupling phenomena. Hence `Farm-gym` proposes many variants of various difficulties which can serve to model particular challenges (non-stationarity, coupling, long-term planning, active observation). Another advantage of this environment is its great versatility with the possibility to choose many parameters. In the end, each farm, created by composing several entities in a modular way can be seen as a game, the set of games that `Farm-gym` offers can be seen as a form of the popular arcade environment Atari for agronomy games. In this section, we first provide a high-level description of a a few types of games, showing the versatility of the platform. Then, we provide a case study, highlighting some non-trivial phenomena even in simple games.

5.1 Challenges of the Farm-Gym environments

The `Farm-gym` learning environment we present in this paper presents several challenges that make it both rich and relevant for seeking to create an agent capable of understanding the difficulties of interaction with a real environment.

- **Highly stochastic environment**. The use of stochastic processes in the dynamics of all entities (e.g. stochastic growth models) makes it challenging for many learning algorithms. Indeed, today's algorithms (such as GO-explore) designed to explore and solve complex environments such as Montezuma are based on function approximation using deep learning and are not designed to handle a stochastic dynamic. In a stochastic environment, it is very difficult to obtain the same trajectory twice even by applying similar policies twice in a row. As a result, they struggle exploring and successfully solve a far-gym environment. Strategies inspired from multi-armed bandit are designed to handle noise, however the struggle with large state-action space dynamics. Hence, we expect `Farm-gym` to foster interesting crossbreeding between these deep reinforcement learning and model-based, multi-armed bandit communities.

- **Observation-action trade-off**, in many games, the agent must perform a specific action with an associated cost in order to retrieve relevant information (the amount of water in the soil, the average number of insects in the field, etc.). The agent must learn to identify when these additional observations are crucial to improve the management of the field, that it, it must learn when to act. To some extent this requires the agent to evaluate the state of its knowledge about the system. If it feels that its representation of the farming system is no longer adequate it can obtain the updated values by going and looking as a farmer would. This however creates a non-trivial challenge for traditional reinforcement learning agents.

- **A large, structured action space**. As each entity comes with its own set of possibly parameterize actions, a farm game can rapidly feature a large number of actions, of mixed nature, either discrete or continuous. When adding the challenge of observation, that is not giving observation for free to

the learner, so that observing a state variable becomes an action, the number of action increases even more. Then, several of these actions act on coupled dynamics, all of which tends to increase the difficulty of exploration. On the other hand, the action space is very structured, and some actions (e.g. watering) directly affect only part of the system. We believe this fosters research on studying reinforcement learning in the context of a large structured action space.

- **Reward-scarce, cost-constrained learning**. Similar to some board game environments or reward-scare aracade environments (Moctezuma), the rewards is mostly obtained at the end of the game, following a long sequence of actions. This makes learning particularly difficult for the agent. This challenge is acknowledged in the literature, as several works studied it. However, extending the solutions to the stochastic `Farm-gym` environments seem not trivial. When actions are considered costly, then along the trajectory the agent receives negative rewards that model the costs of the actions. This enables to model cost-constrained reinforcement learning, a challenging variant of reinforcement learning.

- **Forecasts of external variables** Last, in the observations made available by `Farm-gym` , there is the possibility of getting a forecast on the future evolution of some uncontrollable variables (e.g. weather). While correctly handling such information can be extremely beneficial to the learner, it also represents a challenge for most learning algorithms. Further, such observations are inaccurate, for instance the forecast for the next day may contain little noise while the forecast for the following days are increasingly noisy.

The great modularity of `Farm-gym` makes it easy to created games to study combination of these multiple difficulties having a real-world flavor. Hence, this makes this platform particularly interesting to foster the development of agents able to interact with a real environment. It is possible to create your own set and select which modules and information types the agent should handle. These should be considered with some care. For instance it should be noted that having all observations permanently available, hence making a environment an (fully-observed) MDP, does not necessarily make the game easier. Indeed, the large number of observations that are not necessarily useful for solving a particular game, and the fact that some of them are noisy (e.g. the forecasts), can greatly slow down the agent's learning process.

5.2 Plants

We consider three types of plants: beans, tomatoes and corn. These plants cover the main families of cultivated plants and have distinct growth and pollination characteristics, soil preferences and sensitivity to stress (water, heat, etc.). For instance, regarding pollination, beans best grow when pollinated by insects, while tomatoes auto-pollinate, and Corn best grow when pollinated with wind. Regarding temperature, beans is best adapted to cold and wet whether, while tomato and corn to hot weather. Most plants can be seen as interpolating between these three archetypes. It should be noted that these plant models, despite being a bit crude, are inspired by characteristics of real plants for better interpretability purpose. The goal in `Farm-gym` is however not to reproduce accurate dynamics of actual plants.

Figure 5: Illustration of the development stages for three plant models, Bean, Corn and Tomato.

5.3 Illustrative study of some dynamics

In this section, we illustrate basic results on a few farms featuring various entities (Soil, Plant, Pollinators, Weeds, Pests). They highlight the complex dynamics that can emerge by considering multiple entities interacting with each other in `Farm-gym` . In this section we will investigate a few games, for illustrative purpose. To understand the different mechanics at work in farm-gym, we propose to observe the effect of deterministic policies in simple environments. By setting up simple deterministic policies in different games, we highlight the different levels of complexity of the environment.

Watering and soil types In the first game, the farm is 1×1 and the player only has to choose each day how much to water the plant, and when to harvest. The agent has access to all observations. Here we explore a simple deterministic policy that waters the same amount each day and harvests when the plant reaches ripe stage. Each plant (bean, corn, tomato) has its own characteristics, and their development depends on the type of soil, so we compare the three plants proposed in the two most different soils: sand, that does not retain water and clay that has high water retention. For the reward, we simply consider the yield. All boxplots are built using 100 repetitions. In this experiment, we fixed a weather to a dry weather, so that the crop is not rainfed and the only water input comes from watering. Figure 6 illustrates a clear difference between the type of soils, as with less that 2L of water inputs in the sand soil, the plant does not grow, unlike in clay.

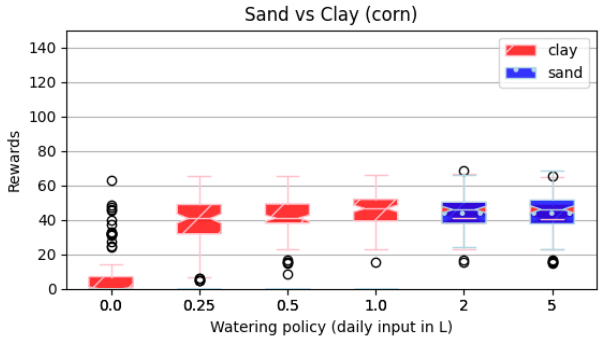

Figure 6: Effect of watering in different soils (clay and sand).

Pollinators In Figure 7, considering a clay soil, we then illustrate the effect of having pollinators in the field. Different plant react differently to pollinators. For instance corn does not depend much on insect pollination, while beans depend a lot on it. We observe that the presence of pollinators has a major positive effect on the yielf for beans, and has limited effect on corn.

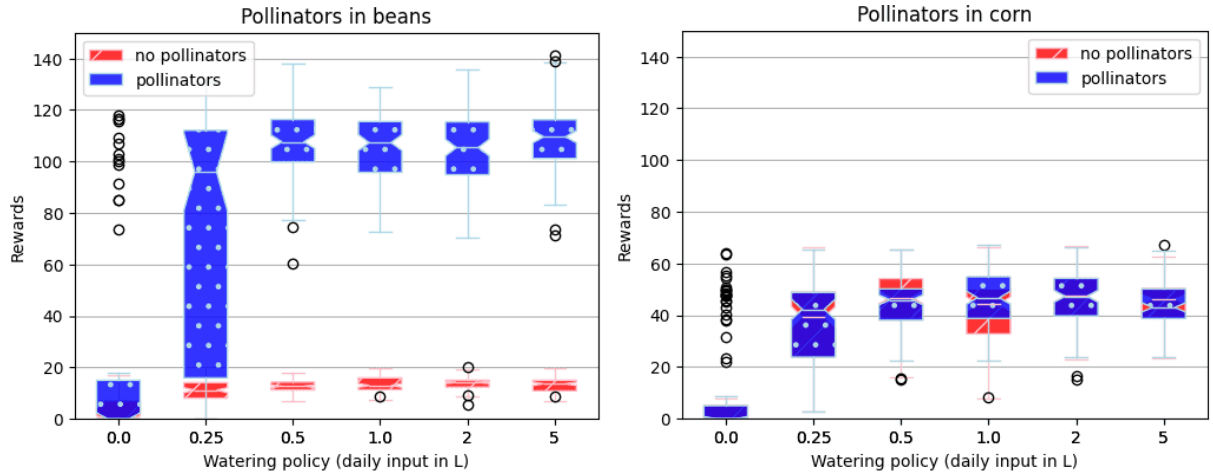

Figure 7: Effect of watering with or without pollinators.

Pest-Weed coupling game In the second game, the farm is 1×1 and the player now has to choose each day between removing weeds, doing nothing or harvesting. In this game, the farm receives good watering conditions, however there are weeds and pests. Since pests spread between plants and weeds in a plot, the more weeds the less pest on the main plant. However weeds compete with the plant for nutrients. This creates a trade-off. Indeed, while weeds compete with the plant in terms of nutrients and water, some can also protect the main plant from pests. In that case, removing the weeds may not be the best option. We consider here the learner spreads a tiny mount of herbicide at different fixed frequencies. At low frequency, the effect is to slow-down the developments of weeds, while at high frequency, this kills the weeds, causing the pests to attack the plant only. In Figure 8, we illustrate this interesting coupling phenomenon.

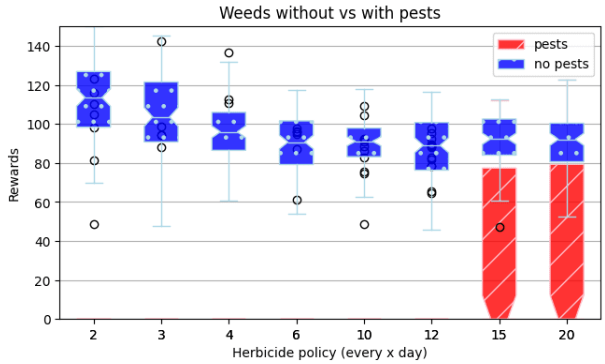

Figure 8: Effect of spreading low herbicide on weeds, with or without pest insects.

6. Conclusion and perspective

We have presented `Farm-gym` [3] with the aim to create a modular environment that mimics the dynamics of an agricultural farm but still is simple enough to control and diagnose. `Farm-gym` takes a gamification approach in contrast to the high-precision simulators whose goal is to accurately model an individual entity. Each farm is constructed with multiple entities in a *modular* way, which can be seen as a game. We can choose which dynamics of the environment to simulate, and also to combine these dynamics to study specific coupling phenomena. Hence, `Farm-gym` proposes many variants of the farming problem, which can serve to model particular challenges, such as non-stationarity, coupling, long-term planning, active observation, etc. Specially, the `Farm-gym` platform is designed to host a number of environments built by combining different modules, cost-constraint and scores, such that each environment can be considered as a separate game of various and progressive difficulty. This provides even a non-expert an opportunity to play with and study parts of the complex farming problem with simple games.

For RL researchers, the agronomy-inspired `Farm-gym` offers a bouquet of stimulating challenges to study and experiment with, such as intrinsically *stochastic* dynamics, interaction and *coupled* dynamics, the *observation-action trade-off*, and user-defined *score function*, defined as the reward of the user minus the cost due to observation and intervention. We experimentally demonstrate the impact of the coupled dynamics, and the challenges encountered by classical deep RL algorithms (Section 5). Experimental results indicate that designing learning strategies capable of simultaneously solving many `Farm-gym` games will help pave the way towards real-life applicable RL.

While we provide a few initial games and entity modules, the platform allows easy addition of new games and modules (e.g. fungus, slugs). Hence, `Farm-gym` also offers the possibility to study games involving more real-like complex and refined interactions in the future.

Acknowlegements

We thank the contribution of Thomas Carta, who started working on an earlier version of this project during his internship at Scool in summer 2020. Especially, even though the entire code and document has been redesigned since then, he brought the connection to the water model and plant development models to which he contributed greatly, as well as the cultivar choices and their initial parameter tuning.

3. Code is available at: https://github.com/farm-gym/farm-gym

This work has been supported by the French Ministry of Higher Education and Research, Inria, Scool, the Hauts-de-France region, the Inria A.Ex SR4SG, the MEL and the I-Site ULNE regarding project R-PILOTE-19-004- APPRENF.

References

Reinforcement Learning, second edition: An Introduction. MIT Press, 2018.

Dotan Di Castro Assaf Hallak and Shie Mannor. Contextual markov decision processes. *arXiv preprint arXiv:1502.02259v1*, 2015.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Mengting Chen, Yuanlai Cui, Xiaonan Wang, Hengwang Xie, Fangping Liu, Tongyuan Luo, Shizong Zheng, and Yufeng Luo. A reinforcement learning approach to irrigation decision-making for rice using weather forecasts. *Agricultural Water Management*, 250:106838, 2021.

Frédérick Garcia. Use of reinforcement learning and simulation to optimize wheat crop technical management. In *Proceedings of the International Congress on Modelling and Simulation (MODSIM'99) Hamilton, New-Zealand*, pages 801–806, 1999.

Romain Gautron, Emilio J. Padrón, Philippe Preux, Julien Bigot, Odalric-Ambrym Maillard, and David Emukpere. gym-DSSAT: a crop model turned into a Reinforcement Learning environment. Research Report RR-9460, Inria Lille, July 2022. URL https://hal.inria.fr/hal-03711132.

Armen R Kemanian, Yuning Shi, Charles M White, Felipe Montes, Claudio O Stöckle, David R Huggins, Maria Laura Cangiano, Giovani Stefani-Faé, and Rachel K Nydegger Rozum. The cycles agroecosystem model: Fundamentals, testing, and applications. *SSRN Electronic Journal*, 2022.

Edouard Leurent. An environment for autonomous driving decision-making. https://github.com/eleurent/highway-env, 2018.

Hiske Overweg, Herman NC Berghuijs, and Ioannis N Athanasiadis. Cropgym: a reinforcement learning environment for crop management. *arXiv preprint arXiv:2104.04326*, 2021.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley and Sons,, 2017.

Rajkumar Ramamurthy, Rafet Sifa, and Christian Bauckhage. Nlpgym – a toolkit for evaluating rl agents on natural language processing tasks, 2020.

Y.K. Sheng and S. Zhang. Analysis of problems and trends of decision support systems development. *International Conference on E-Business and Information System Security*, 2009.

Lijia Sun, Yanxiang Yang, Jiang Hu, Dana Porter, Thomas Marek, and Charles Hillyer. Reinforcement learning control for water-efficient agricultural irrigation. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 1334–1341. IEEE, 2017.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109.

Ronan Trépos, Stéphane Lemarié, Hélène Raynal, Muriel Morison, Stéphane Couture, and Frédérick Garcia. Apprentissage par renforcement pour l'optimisation de la conduite de culture du colza. In *14e Journées Francophones Planification, Décision, Apprentissage pour la conduite de système-JFPDA*, pages 1–13, 2014.

Jhon D Valiantzas. Temperature-and humidity-based simplified penman's et0 formulae. comparisons with temperature-based hargreaves-samani and other methodologies. *Agricultural Water Management*, 2018.

Yanxiang Yang, Jiang Hu, Dana Porter, Thomas Marek, Kevin Heflin, and Hongxin Kong. Deep reinforcement learning-based irrigation scheduling. *Transactions of the ASABE*, 63(3):549–556, 2020.