Université de Montréal

Optimizing Vertical Farming: Control and Scheduling Algorithms for Enhanced Plant Growth

par

Cong Vinh Vu

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

October 27, 2023

Résumé

L'agriculture verticale permet de contrôler presque totalement les conditions pour croître des plantes, qu'il s'agisse des conditions météorologiques, des nutriments nécessaires à la croissance des plantes ou même de la lutte contre les parasites. Il est donc possible de trouver et de définir des paramètres susceptibles d'augmenter le rendement et la qualité des récoltes et de minimiser la consommation d'énergie dans la mesure du possible. À cette fin, ce mémoire présente des algorithmes d'optimisation tels qu'une version améliorée du recuit simulé qui peut être utilisée pour trouver et donner des lignes directrices pour les paramètres de l'agriculture verticale. Nous présentons égalementune contribution sur la façon dont les algorithmes de contrôle, p. ex. l'apprentissage par renforcement profond avec les méthodes critiques d'acteurs, peuvent être améliorés grâce à une exploration plus efficace en prenant en compte de l'incertitude épistémique lors de la sélection des actions. cette contribution peut profiter aux systèmes de contrôle conçus pour l'agriculture verticale. Nous montrons que notre travail est capable de surpasser certains algorithmes utilisés pour l'optimisation et le contrôle continu.

Mots-clés : Agriculture verticale, recuit simulé, apprentissage par renforcement profond, méthodes critiques d'acteurs.

Abstract

Vertical farming provides a way to have almost total control over agriculture, whether it be controlling weather conditions, nutrients necessary for plant growth, or even pest control. As such, it is possible to find and set parameters that can increase crop yield, and quality, and minimize energy consumption where possible. To that end, this thesis presents optimization algorithms such as an enhanced version of Simulated Annealing that can be used to find and give guidelines for those parameters. We also present work on how real-time control algorithms such as Actor-Critic methods can be made to perform better through more efficient exploration by taking into account epistemic uncertainty during action selection which can also benefit control systems made for vertical farming. We show that our work is able to outperform some algorithms used for optimization and continuous control.

Keywords: Vertical Farming, Simulated Annealing, Deep Reinforcement Learning, Actor-Critic methods

Contents

List of tables

List of figures

Liste des sigles et des abréviations

RL	**R**einforcement **L**earning
HC	**H**ill **C**limbing
GA	**G**enetic **A**lgorithm
DSE	**D**esign **S**pace **E**xploration
KPI	**K**ey **P**erformance **I**ndicator
TD	**T**emporal **D**ifference
DQN	**D**eep **Q**-**N**etwork
DDPG	**D**eep **D**eterministic **P**olicy **G**radient
TD3	**T**win **D**elayed **D**eterministic **P**olicy **G**radient
SHCP	**S**tochastic **H**ill **C**limbing with **P**runing

SAAC	**S**imulated **A**nnealing-assisted **A**ctor-**C**ritic

Chapter 1

Introduction

1.1. Context

Controlled Environment Agriculture (CEA) [**32**] is an approach where crops are grown in environments where most parameters necessary for plant growth are controlled. Temperature, relative humidity, carbon dioxide levels, lighting, water quality, and nutrient concentration are just some of the many parameters controlled by CEA. Benefits of CEA include climate control, which enables year-round agricultural production. In addition, due to advanced irrigation systems, there is reduced water usage and water recycling, leading to better water efficiency. Examples of CEA are indoor environments such as a greenhouse or a plant factory.

1.2. Problem

The research project in which my work is involved consists of developing a digital twin of a vertical farm which is a virtual representation of the farm that uses real-world data along with data analysis. A key component of the digital twin involves data analysis used for understanding and reasoning to provide the vertical farm's operator with key insights on improving production. In order to be able to get insights on how to improve their production, we generate programs that the farm's operator can use as guidance to enhance specific key performance indices (KPIs). The programs consist of tunable parameters controlled by the farm that can be set to enhance production on a day-to-day basis for the entirety of the production cycle. Since most parameters necessary for the growth of strawberries are controlled by engineered systems, using optimization algorithms, resource management could be done in an efficient way which would reduce energy consumption, and increase fruit yield and quality. By making a simulator that would more or less accurately depict the control systems of the vertical farm while also modeling strawberry plant growth, data can be collected to allow for better reasoning surrounding the entire vertical farm's operation.

In addition, if sufficient accurate data is gathered from the simulator, there is potential for the application of real-time control to take place, allowing a system that would be able to manage resources efficiently while also correcting any error introduced during production.

1.3. Contribution

In order to increase efficiency during production cycles, reasoning must be done on data gathered through simulation and production. The data can then be used to find a general directive on increasing yield while minimizing operations costs. While the general directive shows the path that would lead to an increase in efficiency, the path itself might be rugged, hence the need for algorithms that can delicately allow the user to follow the path without stumbling. As such it is necessary to have a control algorithm that would supplement the first algorithm tasked to find programs that would lead to better production. It would allow the operator to follow the program found and adjust the farm's parameters in the event of unexpected changes in conditions during plant growth.

This thesis introduces the notion of generating operators' programs that can be followed by a vertical farm's operator to increase plant yield while also minimizing energy consumption. To that end, we explore the usage of optimization algorithms that can find a suitable set of parameters that can benefit plant growth during certain growth phases. In addition, we also look at how some control algorithms can be improved by enhancing the manner in which exploration is done. Enhancing exploration would lead to better control algorithms that have the potential to be applied to control systems such as the ones used in a vertical farm. The contributions present in this thesis are as follows:

- An encoding of different parameters controlled by a vertical farm of strawberries that can be used for optimization algorithms shown in Section 3.1
- An algorithm that reduces the parameters' search space for more efficient exploration demonstrated in Section 3.2
- An algorithm that uses past knowledge learned to enhance exploration within a search space of configurations shown in Section 3.3
- An algorithm that modifies exploration done by Actor-Critic methods which allow for enhanced learning present in Section 4.2

1.4. Thesis structure

Chapter 2 gives the background necessary to understand the work done in Chapters 3 and 4. It gives an overview of the basics of Reinforcement Learning, Deep Reinforcement Learning, and Optimization Algorithms. Chapter 3 explains our first contribution, how to generate plant growth programs that increase crop yield and minimize energy consumption for CEA. Chapter 4 details our second contribution, how some control algorithms can be enhanced to perform better in the long run. These control algorithms are used to Finally, we conclude the thesis in Chapter 5, giving a brief summary of our work as well as describing its limitations and possible future work.

Chapter 2

Background and Related Work

In this chapter, we present the necessary literature to understand the work presented in this thesis. We first present a brief overview of reinforcement learning along with its fundamental algorithms. We then present background related to deep reinforcement learning and some of the state-of-the-art algorithms. Lastly, we go over some optimization algorithms which are used for the contributions in this thesis.

2.1. Markov Decision Process

A Markov Decision Process (MDP) is a mathematical formulation used to model and solve decision-making problems where outcomes are stochastic, but also in part under the control of the decision maker called an agent. Key components of a MDP involve states (s), actions (a), a transition function ($P(s', |s, a)$), and a reward (r). States represent the conditions in which the environment is due to the agent's actions and the transition function related to it. Actions represent the set of choices the agent can make in order to change the state of the environment. Each state has its corresponding set of actions which an agent can take. The transition function $P(s', |s, a)$ is a probability function that takes as input the action of the agent as well as the current state of the environment. It outputs the next state with a certain probability, meaning that for the same input, the state returned by the function might be different. Lastly, the reward is received by the agent after taking an action which leads it to a state. It can be positive or negative and quantifies the desirability of taking a certain action a in state s, which leads it to the next state s'. The goal of MDPs is to find the optimal policy $\pi*$, an optimal mapping of states to actions, which leads to the highest expected cumulative reward over time.

2.2. Reinforcement Learning

Reinforcement learning is a type of machine learning that uses MDPs to train an agent to make sequential decisions in an environment to maximize a cumulative reward [**30**].

The agent can sense its environment, has explicit goals, and can influence the environment through its actions. To be able to achieve its goals, the agent has a policy that maps its behavior according to what it senses and a value function from which it is able to tell how valuable a state is in the long term. The policy $\pi(s, a)$ is a mapping between an agent's actions, defined as the variable a, and the current state of the environment, defined as variable s. The value function is a mapping of real values to states s, with the values representing the long-term desirability of states. To achieve its goals, the agent learns by exploring the environment and uses its policy and value function to exploit to be able to maximize the reward signal.

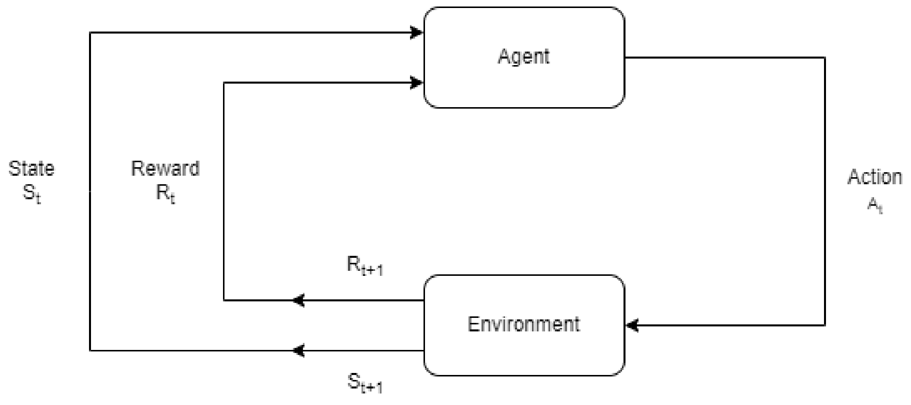

Fig. 2.1. Interaction cycle between an agent and the environment in the context of reinforcement learning

Figure 2.1 shows how a reinforcement learning agent interacts with its environment. The agent outputs an action a which is applied to the environment, which in turn creates a new state S_{t+1} and reward R_{t+1}. The state and reward are then fed as input to the agent to improve its policy $\pi(s, a)$ and retrieve the next action given the current state of the environment.

Reinforcement learning algorithms have many applications, notably for solving decision-making and control problems. By training a model that learns to choose its actions depending on its observations and the rewards it gets from taking those actions, these models be used to guide us to valuable states from a set of configurable actions. They have been known to be able to tackle both deterministic and stochastic problems well enough. In our case, crop growth is a stochastic process where an agent can learn to tune parameters to maximize expected plant growth. Reinforcement learning can also be made to adjust parameters within an acceptable threshold in a similar fashion to a control system.

2.2.1. Temporal Difference Learning

Temporal difference (TD) learning is an algorithm that allows the agent to learn an estimate of a value function directly from experience without needing to wait for a final outcome [**29**]. The algorithm only requires the next state s_{t+1} and reward observed r_{t+1} to update $V(s)$.

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.2.1}$$

TD learning is a bootstrapping method as it uses existing estimates of $V(s)$ to update the estimates of future states with a discount factor of γ. The difference between the predicted value of a state s and the observed reward r is used to adjust the value estimates to better approximate the true value of states.

2.2.2. Q-Learning

A new value function can be obtained by pairing state s with action a to give an action-value function $Q(s, a)$. The action-value function $Q(s, a)$ estimates expected cumulative rewards gained by an agent when choosing action a from a policy π given a state s. Just like TD learning, Q-learning is a bootstrapping method, using current estimates of $Q(s, a)$ to update future ones.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.2.2}$$

Q-Learning is considered as off-policy as the current estimates of Q-values are updated by choosing the best action for the next state s_{t+1}, which differs from the policy which led to the current state s and the choice of the current action a. The Q-values $Q(s, a)$ are stored in a table, and by iteratively updating $Q(s, a)$ for all states and actions, it is guaranteed that the Q-table converges to the optimal Q-values and hence result in the optimal policy [**34**].

2.3. Deep Reinforcement Learning

Issues arise when either the action or state-space is too large. The Q-table used to store the Q-values becomes increasingly large. In addition, the time needed to query the table and the amount of data needed to ensure convergence are substantial. To alleviate these issues, generalization from previous encounters of states and actions is needed. Function approximation allows us to take a desired function and generalize it through the use of an approximation. Both the policy π and action-value function $Q(s, a)$ can be parametrized and approximated through the use of supervised learning methods such as neural networks.

2.3.1. Neural Networks

A neural network or artificial neural network is a mathematical model inspired by how neurons within brains work. Neurons or nodes are organized into layers and interconnected with weights that can be tuned for the purpose of function approximation or classification. The first layer is called the input layer as it takes the raw inputs while the subsequent layers are called hidden layers. The last layer is called the output layer as it outputs a prediction to either classify something or give an approximation of a function. A neuron applies the dot product between weights and an array of numbers, either inputs or outputs of the previous layer. The output of the neuron is then passed through an activation function that introduces non-linearity. The fact that non-linearity is applied throughout each layer is what gives neural networks the ability to learn complex patterns and relationships between data [**14**]. It has been proven that a neural network with a single hidden layer and a finite number of weights can approximate any arbitrary continuous function [**13**].

A neural network learns through the use of a labeled dataset, where the desired output is known for an input. For a given input, it can compare its predictions with the desired output to calculate an error using a loss function to adjust its weights through the use of an optimization algorithm such as gradient descent. The optimization algorithm changes the values of each weight by minimizing the loss function. Neural networks are typically trained using an algorithm called backpropagation. The algorithm uses the chain rule of calculus to compute the gradient with respect to the weights in each layer to tune them appropriately for every iteration. Backpropagation requires a forward pass, where the input is fed to the neural network, with the network computing the values of each neuron in every layer to finally output a prediction. The loss function then calculates the discrepancy between the prediction and desired output to start the backward pass. In the backward pass, the algorithm calculates the gradient of the loss with respect to the weights in each layer to finally update the weights incrementally as to minimize the loss function. These steps are done multiple times with various batches of data from the dataset until the network's error rate reaches a certain acceptable threshold or a criterion is met [**25**].

2.3.2. Deep Q Networks

As previously mentioned, using neural networks for function approximation can alleviate the issue of an environment having a large action or state-space. The action-value function $Q(s, a)$ can be parametrized and approximated using neural networks. However, many issues surround the use of neural networks for reinforcement learning. Most notably, data distribution changes due to an agent learning a new policy, the dependent nature of the sample of experiences collected by the agent, and divergence issues with Q-networks when combining off-policy learning with non-linear function approximation [**6**].

To solve the issues of the data distribution changing as well as breaking the correlation between samples, Deep Q Networks (DQN) introduces experience replay, a replay buffer. During training, the samples of states s, actions a, rewards r, and next states s_{t+1} collected by the agent are stored inside a replay buffer, which is then sampled from randomly to train the neural networks (Q-networks) which approximate the action-value function. In addition, DQN introduces the concept of a target network for the purpose of decoupling the target Q-values from the parameters of the network which gives the Q-network a more stable target. The target network is a copy of the Q-network, however, its parameters are only updated to the current Q-network parameters every few iterations. The loss function minimizes the mean squared error between the Q-networks predictions and the target network predictions, called the target [**22**].

$$target = r + \gamma \max_a Q_{target}(s_{t+1}, a) \tag{2.3.1}$$

The output layer of the Q-networks consists of a Q-value for a given state s and action a for every discrete action possible. The agent then chooses the action with the highest associated Q-value with probability $1 - \epsilon$ as its next action.

2.3.3. Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is a reinforcement learning algorithm that like DQN can handle continuous state-spaces. In addition, DDPG can also handle continuous action spaces. DDPG is an actor-critic algorithm as it has 2 main components. an actor and a critic [**18**]. The actor is a neural network that predicts which action a the agent should take given a state or observation s. The critic meanwhile is trained to evaluate the quality of the actions taken by the actor by approximating an action-value function $Q(s, a)$. DDPG is based on the DPG algorithm (deterministic policy gradient) where the actor directly outputs an action instead of a probability distribution over actions [**28**]. This allows the agent to precisely choose its actions in continuous action spaces. Similarly to DQN, DDPG employs a replay buffer to decorrelate data samples as well as enhance training stability. It also uses a target network to train the critic for the same reasons as DQN but also adds a target network for the actor. Instead of copying the parameters of the actor and critic networks every few iterations, the target networks have their weights slowly changed to the parameters of the actor and critic networks to further enhance stability during training.

$$\max_\theta E[Q_\phi(s, \mu_\theta(s))] \tag{2.3.2}$$

To train the actor, gradient ascent is performed with respect to the policy's parameters θ to maximize the action-value function predicted by the critic $Q_\phi(s, a)$.

$$target = r + \gamma Q_{target}(s_{t+1}, \pi_{\theta_{target}}(s_{t+1})) \tag{2.3.3}$$

The critic is trained similarly as in DQN, except that the target actor $\pi_{\theta_{target}}(s)$ is used to predict the action when computing the target.

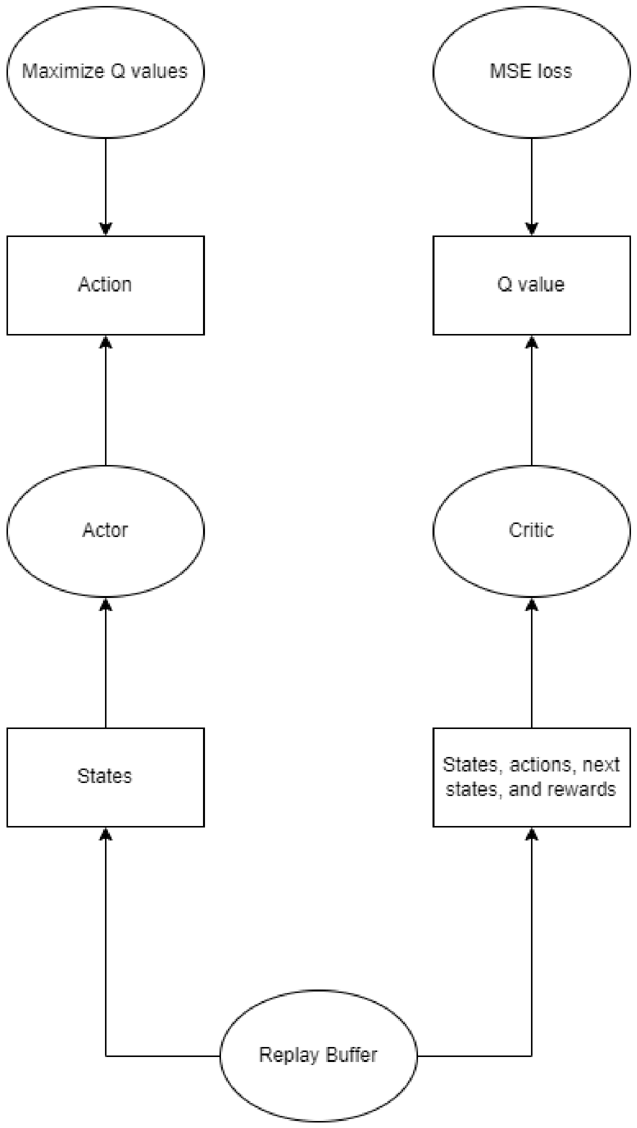

Fig. 2.2. Trainning of the actor and critic

Figure 2.2 demonstrates the process in which the actor and critic are trained. The critic is first trained by sampling tuples from the replay buffer, calculating the target using Equation 2.3.3, and computing the mean-squared error loss between the target and the critic's current predictions. The actor is then trained using the critic and states of the replay buffer, finding an action that maximizes the Q-values for each state.

Exploration is done by adding noise to the actor's actions so that the agent explores its environment. The noise can be decayed over time to ensure that the final policy becomes deterministic and make sure that the agent exploits by taking action a that the actor predicts as the best for a given state s [**20**].

2.3.4. Twin Delayed Deep Deterministic Policy Gradient

Twin Delayed DDPG or TD3, is an extension of the DDPG algorithm. Due to the use of Q-learning, overestimation bias appears within the DDPG algorithm as the maximization of a noisy value estimate leads to overestimation [**31**]. By adding an additional critic network, TD3 tries to mitigate the overestimation bias by choosing the smallest Q-value prediction between the two critic networks as the target from which to compute [**33**]. TD3 also introduces a delay to the policy updates to ensure that the actor is updated with more stable target values while also reducing dependency on the current action-value estimations of the critics. Lastly, noise is added to the target policy when computing the target actions for the critic, as this reduces the overfitting to specific actions when the critic estimates incorrect sharp peaks for those actions [**11**].

2.4. Optimization Algorithms

Optimization algorithms [**17**] are algorithms or computational processes used to either maximize or minimize a set of metrics that are subject to certain constraints. They are used in many disciplines such as physics business, biology, and engineering. Some notable challenges concerning these algorithms are large high-dimensional search spaces, multiple competing objectives, and uncertainty regarding evaluation metrics.

2.4.1. Hill Climbing

Hill Climbing [**26**] is an iterative algorithm that first starts with an initial solution. It then searches around the neighborhood of the solution to pick better a solution by making small modifications to the current one. The algorithm continues until it can no longer find better neighboring solutions. If it compares all the possible neighbors to the current solution before choosing the best solution, it is called the steepest ascent Hill Climbing algorithm instead. Due to the criterion of stopping when no better surrounding solutions are found, it is prone to only finding locally optimal solutions. To alleviate such an issue, multiple variations of the algorithm were proposed such as Stochastic Hill Climbing and Random-Restart Hill Climbing. Stochastic Hill Climbing doesn't always choose the best solution out of all the potential neighbors, instead, it chooses ones at random with a selection probability higher the better the neighboring solution is compared to the current solution. Random-Restart Hill Climbing does Hill Climbing iteratively, with each iteration starting at a new starting solution instead. The algorithm then compares each solution generated by each iteration of Hill Climbing and chooses the best to store.

2.4.2. Metaheuristics

Metaheuristics are optimization algorithms used to find approximate solutions to complex optimization problems when finding exact solutions is either unfeasible or expensive. They explore the search space efficiently and make informed decisions based on the information gathered during the search process. Unlike exact optimization methods or iterative methods, metaheuristics are not guaranteed to find an optimal solution, instead with the amount of time allocated and information given to them, find good or decent solutions. They also have features that avoid the disadvantages of local search by escaping local optima by either finding temporary worst solutions or generating new starting solutions through randomness [**7**].

2.4.2.1. Genetic Algorithm. Genetic Algorithm (GA) is a metaheuristic approach that is inspired by the process of natural selection [**9**]. It operates on a population of candidate solutions which are then modified through genetic operators to yield new neighboring solutions which then replace the original population. Using a fitness function, each candidate is then evaluated and selected to generate new candidates. GA has 5 steps, the first initialization, generates an initial population of solutions randomly. The next 4 steps are done iteratively to refine candidate solutions. The candidate solutions are then evaluated using the fitness function that quantifies the quality of each candidate solution. Selection is then done where the fittest candidates have a higher probability of getting chosen. Using those candidates, the crossover is then applied where selected pairs of solutions have their genetic material exchanged and recombined to create new offspring. A mutation is also applied to these offspring by modifying random components of the new solutions to introduce new changes and variations to the pool of candidates. These new candidates then replace the original population. These steps are done iteratively until a termination criterion is met. Unfortunately, genetic algorithms have certain limitations. GA scales poorly in highly high-dimensional solution spaces and they have higher computational complexity due to the need to evaluate every single individual in the population [**24**].

2.4.2.2. Simulated Annealing. Simulated Annealing (SA) is a metaheuristic approach that is inspired by the annealing process of metallurgy [**16**]. It is a search process where at the start, a high temperature allows for more the selection of worst neighboring solutions, but as time progresses and temperature decreases, only better solutions are selected to ensure convergence towards an optimal solution. The algorithm starts by taking an initial solution which is then evaluated. A neighbor to the current solution is then generated by making small modifications to the current solution. The neighboring solution is then compared to the current solution and depending on the temperature has a certain probability of being accepted as the new solution. These steps are repeated until a stopping criterion is reached such as reaching the maximum number of iterations.

By being a fairly simple algorithm yet fairly efficient at finding good solutions without getting stuck in local optimas, simulated annealing is well suited for finding good configurations for problems where the search space is extremely large and searching is costly. By adjusting the number of neighbors visited, the search time can be decreased while not degrading the configurations by a substantial amount as only better solutions are selected to ensure convergence toward an optimal solution, promising a solution that at the very least is close to the optimal solution.

2.4.2.3. Neural Simulated Annealing. Neural Simulated Annealing [**9**] introduces the idea of using a reinforcement learning algorithm to model the function that generates the neighbor for Simulated Annealing. They use Proximal Policy Optimization (PPO) as the reinforcement learning algorithm used to learn a proposal distribution used to generate neighboring solutions.

PPO is an on-policy actor-critic algorithm that clips the policy updates in order to force the newly learned policy to be close to the old policy so as to not have drastic policy changes that could collapse the performance of the agent from one iteration to another. It is an on-policy algorithm that uses the samples from the rollout of its current policy in order to learn a better policy. Contrary to DDPG and TD3, PPO's actor outputs a probability distribution over actions which is then sampled to get the action the agent performs in the current state of the environment [**27**].

Neural Simulated Annealing outperforms vanilla SA while also finding solutions near global optima. It is also able to reduce the amount of time necessary to find good solutions for problems where the solution space is large. SA can be used for finding good solutions for deterministic problems with a large search space, as the simulator for this project is deterministic, SA is well suited to find nearly optimal parameters for crop growth.

2.4.3. Design Space Exploration

Design space exploration (DSE) [**15**] is the process of finding design solutions that best meet the desired requirements from a space of possible designs. The size of the search space is usually large and therefore necessitates the usage of optimization algorithms in order to find suitable designs for specific requirements. Exhaustive search, metaheuristics, and dynamic programming have been successful in finding optimal design solutions. In most cases, exhaustive search is infeasible as the space of designs is far too large, instead, most approaches to DSE use either Stochastic Hill Climbing, genetic algorithms, or even machine learning. There are two types of design space exploration problems, single-objective optimization problems, and multiobjective optimization problems.

For single-objective optimization problems, the goal is to find a solution that minimizes a cost or maximizes a key performance indicator (KPI). Multiple costs or KPIs can be combined together using a weighted sum to turn a multiobjective optimization problem into a single-objective optimization problem. However, when the goal of DSE is to minimize multiple independent costs, it is better to consider it as a multiobjective optimization problem.

A multiobjective optimization problem finds a set of optimal solutions with each respecting different and often conflicting objectives. To accomplish that, one must find Pareto-optimal design solutions. It is based on the concept of the Pareto dominance principle which stipulates that solution A is said to dominate another solution B if it is at least as good as solution B for every objective and better in at least one objective. A Pareto-optimal design solution is not dominated by any other solution in the space of design solutions. The set of all Pareto-optimal design solutions is called the Pareto frontier, a set where improving one objective necessitates the sacrifice of another objective. Identifying the set of all Pareto optimal design solutions results solves the multiobjective optimization problem [**8**].

An example of such a problem involving a multiobjective optimization problem is generating models for model-driven engineering (MDE) [**4**]. MDE is an approach that emphasizes the use of models during the development process for software engineering. The models capture the requirements behavior, and issues of systems under development, and are typically expressed using Unified Modeling Language (UML) or domain-specific-modeling languages (DSMLs). DSE is used to find models that are reachable from an initial model by applying a sequence of exploration rules, while also being constrained to include complex structure and numerical restrictions. An exhaustive search for MDE is impossible as there is no upper bound on the number of elements one can add to a model. In addition, most models necessitate the exploration of designs in an incremental manner as elements are added to previous solutions, meaning that starting the search from scratch becomes difficult. To that end, global search techniques such as genetic algorithms can be used for solving DSE problems.

Chapter 3

Design Space Exploration for Vertical Farming

The purpose of this chapter is to present and describe solutions thought of to aid human decisions in controlled vertical farms. We first give a brief introduction of the problem and the necessary background to understand it. We then present possible solutions to the problem and the choice we take with the reasoning behind it. Lastly, we will detail the results of our experiments as well as the limitations encountered while evaluating our solutions.

3.1. Digital twin of a vertical farm

To adhere to the definition of a digital twin [**10**], the farm is represented by a simulator which would enable data analysis, reasoning, and answer what-if questions related to the farm's production cycle [**23**]. To accomplish the task at hand, the primary focus of the research was to aid the operators of the controlled vertical farms by generating sets of configurations that would either enhance production and/or reduce energy consumption. The digital twin of the vertical farm consists of a simulator that models the environment in which the plants grow as well as the growth of the plants [**5**]. As shown in Figure 3.1, the simulator would take as input a configuration, a set of controlled parameters that affect the plants' growth and output key performance indicators (KPIs) which would be used as a heuristic for our algorithms in order to generate a recipe, an ordered set of configurations, which would help enhance production, help reduce energy consumption, or both. Our work also stores the states of the simulator after each step as well as the values of each KPI. The reasoning behind it is that it would allow the farm's operator to retrace the history of the simulation, adjust configurations to their liking, and view the effects of various configurations on certain KPIs.

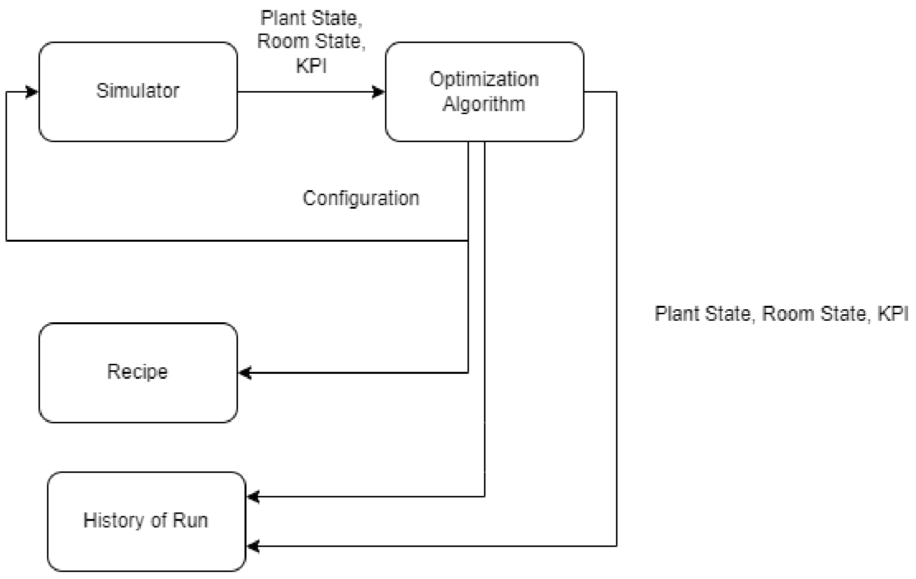

Fig. 3.1. Digital twin cycle for the vertical farm

As previously indicated, configurations are a set of parameters that can be controlled by the vertical farm's operators themselves. The controlled parameters in our case are the room temperature, the intensity of the lights as well as their spectrum, and the humidity level within the room. The temperature is measured in $°C$, the light intensity is measured in $\frac{\mu mol}{m^2 \dot{s}}$, the light's wavelength is measured in nm, and the room's humidity is measured as relative humidity RH (%) as shown in Figure 3.2 and Table 3.1. After discussing with the farm's operator, it was decided that a single configuration would contain the duration of a day, the room's day and night temperature, the intensity and spectrum of lights of the room, and the room's day and night relative humidity. The day's duration is a parameter used to apply two different temperatures and RHs, one during daytime and one during nighttime. The simulator interpolates between the daytime and nighttime values to increase or decrease the values according to the day's duration parameter. While for nighttime, they are decreased for the rest of the day as demonstrated in Figure 3.3. A recipe would contain in total three configurations, one for each phase of growth. The first phase of growth is called rooting and lasts for three weeks. The second phase is called flowering which lasts for five weeks and finally, the last phase which is flowering lasts for eight weeks. A single configuration is used for each phase, where every day for that phase, the same configuration is used to induce growth within the plants. Each phase has an associated KPI used to estimate how well a configuration performs during that particular growth phase. We use the number of leaves as well as the leaves dimensions for the first phase, while for the second one, we look at the number of leaves to truss ratio to determine the potential of a configuration. For the last phase, we look at the total energy consumption as well as the number of kilograms of fruits produced by the recipe to judge how well the entire recipe performed.

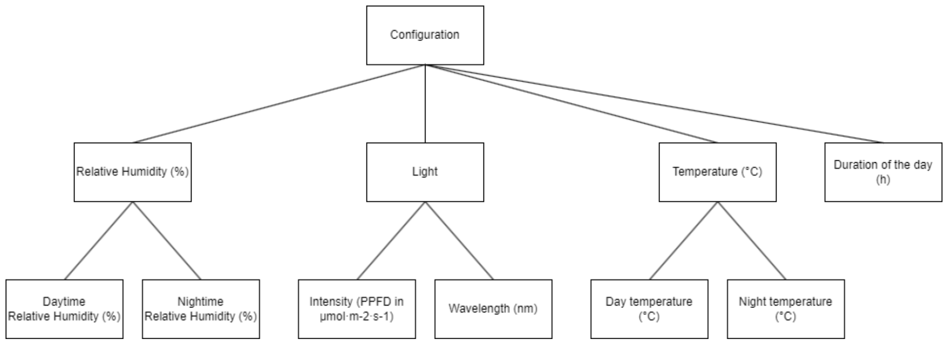

Fig. 3.2. Elements of a single configuration used as input for the simulator

Parameters of a configuration	Units
Daytime Relative Humidity	%
Light Intensity	$\frac{\mu mol}{m^2 \dot{s}}$
Wavelength	nm
Day temperature	$^{\circ}C$
Night temperature	$^{\circ}C$
Daytime duration	hours

Table 3.1. Adjustable parameters found within a configuration that is sent to the simulator for simulation of a full day of crop growth

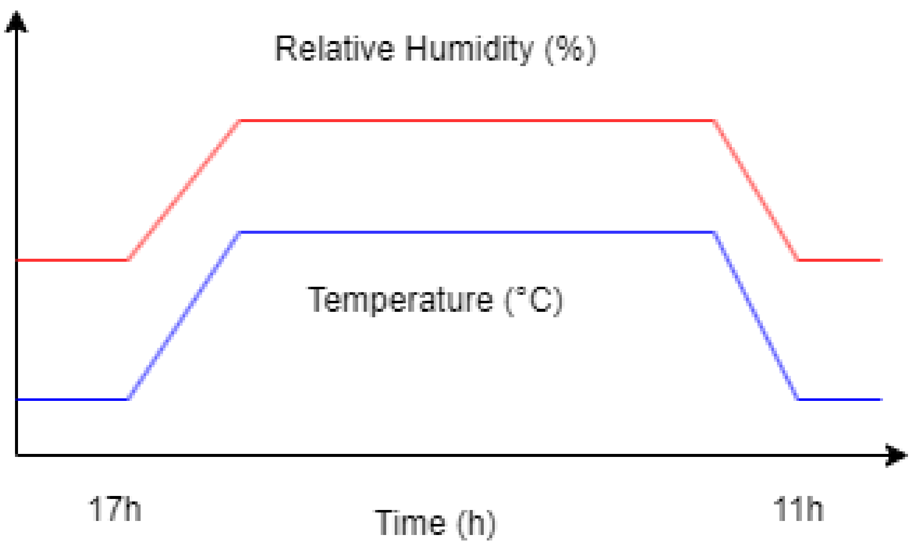

Fig. 3.3. Duration of one day within a configuration and how RH and temperature increase or decrease depending on the time of the day

3.2. Hill Climbing with Pruning of Neighbors

We first tried using a variant of the hill climbing algorithm shown in 3.2.1. For each configuration x, represented as a vector of continuous values, a target function $f(x)$ gives a corresponding value for the said configuration. The simulator in our case acts as the target function f, which we feed our configurations x to as input, to get the value of the configuration. Unlike vanilla hill climbing, we generate a subset of neighbors, with each neighbor having one or multiple of its elements changed. We then compute the associated KPI, represented as a value, of each configuration by asking the simulator to evaluate what would happen if it took a step using the configuration. We calculate the advantage of each neighboring configuration by subtracting the neighboring configuration values from the current configuration value. The advantages are passed to a softmax function which generates a probability distribution and assigns a probability for each configuration [**12**]. The softmax function itself is regulated by a variable that decreases the likelihood of choosing the worst configurations over time. The next configuration to iterate on is the one chosen depending on the probability distribution generated and if it is found to be better than previous configurations, it is assigned as the best configuration. The entire process is repeated k times and the configuration with the highest associated value is returned.

We generate neighboring configurations by either increasing or decreasing each element by a small step. The granularity of the step for each element is configurable and can be changed according to the user's preference. the neighborhood function generates all the combinations of increments and decrements possible for a single configuration and applies each of those changes to the original configuration to create a set of neighboring configurations. As such, if a configuration has p parameters, the neighboring function generates up to $3^p - 1$ neighbors.

Algorithm 3.2.1. Steepest Ascent Stochastic Hill Climbing

```
Input: A configuration $x$ and heuristic $f(x)$
$current\_config = x$; $current\_config\_value = f(x)$;
$best\_config = x$; $best\_config\_value = f(x)$;
for $i$ in $K$;
    $N := generate\_neighbors(x)$;
    $V := f(N)$;
    for $v$ in $V$
        $a := v - current\_config\_value$;
        Add $a$ into $A$;
    $P := softmax(A)$;
    Assign $current\_config$ and $current\_config\_value$ according to probabilities $P$;
    if $current\_config\_value > best\_config\_value$;
        $best\_config := current\_config$;
        $best\_config\_value := current\_config\_value$;
end;
return $best\_config$, $best\_config\_value$;
```

A potential issue with the method above is the running time of evaluating all neighbors n in N. As the number of elements in a configuration x grows, so does the number of neighbors generated by the neighborhood function. For a configuration with two elements, it would generate eight neighbors in total, while a configuration with three parameters would have 26 neighbors. If calling the target function $f(x)$ is time-consuming, evaluating each neighbor can become prohibitively expensive. To alleviate this problem, a potential solution was to create an approximation of an action-value function ($Q(s, a)$) which would take as input a state s (simulator state) and action a (configuration) while outputting the value of choosing a configuration in the current state of the simulator. The assumption behind this solution is that the computational cost of $f(x)$ is much larger than $Q(s, a)$. Using $Q(s, a)$ as a heuristic instead, we then choose the top k neighbors before passing them to the simulator to get the heuristic value of each configuration, resulting in Algorithm 3.2.2.

Algorithm 3.2.2. Stochastic Hill Climbing with Pruning (SHCP)

```
Input: $x$, $f(x)$, $Q(s,a)$, $k$ and simulator state $s$
$current\_config = x$; $current\_config\_value = f(x)$;
$best\_config = x$; $best\_config\_value = f(x)$;
**for** $i$ **in** $K$**;**
    $N := generate\_neighbors(x)$;
    $Q := Q(s,N)$;
    Choose the top k neighbors out of N using Q and assign it to N;
    $V := f(N)$;
    **for** $v$ **in** $V$
        $a := v - current\_config\_value$;
        Add $a$ into $A$;
    $P := softmax(A)$;
    Assign $current\_config$ and $current\_config\_value$ according to probabilities $P$;
    **if** $current\_config\_value > best\_config\_value$**;**
        $best\_config := current\_config$;
        $best\_config\_value := current\_config\_value$;
**end;**
**return** $best\_config$**,** $best\_config\_value$**;**
```

The action-value function ($Q(s,a)$) is created by training a neural network that takes as input past configurations and simulator states which were previously visited. To update the neural network, we use a second neural network which is called a target network ($Q_{target}(s,a)$) to compute the state-action values for the next state encountered. The parameters of the target network are not trained, but instead periodically updated from the trained network. We then compute the target by taking the sum of the reward gained visiting state s, taking action a, and landing in the next state s_{t+1} and adding to the discounted value of the previously computed state-action values from the target network.

$$target = R(s,a,s_{t+1}) + \gamma Q_{target}(s_{t+1},a') \tag{3.2.1}$$

Finally, we compute the mean squared error between our neural network predictions and the target to update the parameters of our neural network.

$$loss = (Q(s,a) - target)^2 \tag{3.2.2}$$

To collect data, we use SHCP as the policy with which we collect our (s,a,s_{t+1},r) tuples which are stored in a replay buffer to be used to train our $Q(s,a)$ network. Once our neural network is trained we then use the same network to evaluate which configurations given the state of the simulator warrant simulation time to get the heuristic value of said configuration in the current simulator state.

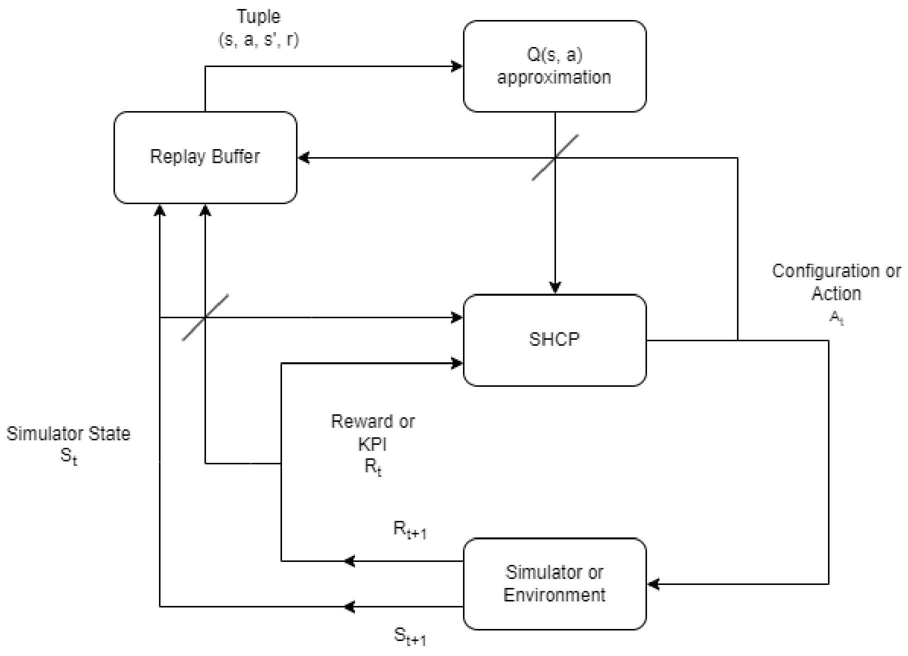

Fig. 3.4. Training and usage cycle of SHCP

Figure 3.4 shows how different components of the algorithm interact with the simulator. The simulator takes as input a configuration while it outputs its next state and the KPI associated with the state. They are sent to the replay buffer for storage to train the action-value function approximator. The next state is used by the action-value function approximator to find configurations that potentially have a high value. Those configurations are then simulated to get a potential next state and the actual KPI associated with it. The algorithm then chooses a configuration that is then sent to the simulator as input.

3.3. Simulated Annealing aided by an Actor-Critic algorithm

One of the big downsides of the algorithm presented above is the computation cost related to it. Even with the pruning of multiple neighbors, we end up simulating $K \times k$ neighbors as the algorithm reduces the number of neighbors sent to the simulator to k while also iterating K times to find one configuration for a given simulator state and step. While reducing both parameters is feasible, reducing K reduces the depth and range of the search of configurations while reducing k impacts heavily on the quality of the configurations explored. To alleviate such an issue, we decided to try using Simulated Annealing instead as the number of configurations evaluated by the simulator can be easily adjusted while keeping the odds of finding good configurations high.

One of the important components of Simulated Annealing is the distribution with which neighboring configurations are generated. In our case, we decided to use a multivariate normal distribution where the mean is computed by an actor-critic algorithm as sen in algorithm 3.3.1. By computing the mean of the multivariate distribution, it would enhance the search for better configurations as the mean configuration should already be of decent quality. We train an actor-critic algorithm with Simulated Annealing performed on the output of the actor to collect the necessary experience to train the actor. Once the actor is trained, we its action as a starting point and mean for the Simulated Annealing algorithm. Simulated Annealing then searches in the neighborhood for potentially better actions/configurations according to its heuristic.

Algorithm 3.3.1. Simulated Annealing aided by an Actor-Critic algorithm

```
Input: $f(x)$, $\sigma$ simulator state $s$, and policy $\pi$
$current\_config = \pi(s)$; $current\_config\_value = f(\pi(s))$;
$best\_config = \pi(s)$; $best\_config\_value = f(\pi(s))$;
for $i$ in $K$
    $config \sim \mathcal{N}(current\_config, \sigma^2)$;
    $config\_value := f(config)$;
    $T := temperature(i, T)$;
    if $P(config\_value, current\_config\_value, T) > random(0, 1)$
        $current\_config := config$;
        $current\_config\_value := config\_value$;
    if $current\_config\_value > best\_config\_value$
        $best\_config := current\_config$;
        $best\_config\_value := current\_config\_value$;
end;
return $best\_config$, $best\_config\_value$;
```

As demonstrated in Figure 3.5, the actor and critic are trained by a training cycle where the actor outputs an action which is then passed to Simulated Annealing. Simulated Annealing then uses that action to find more valuable configurations. These configurations are then passed to the simulator which outputs its next state as well as the associated reward or KPI for that state. All of that data is collected and sent to a replay buffer from which the actor-critic algorithm can sample to train both actor and critic.

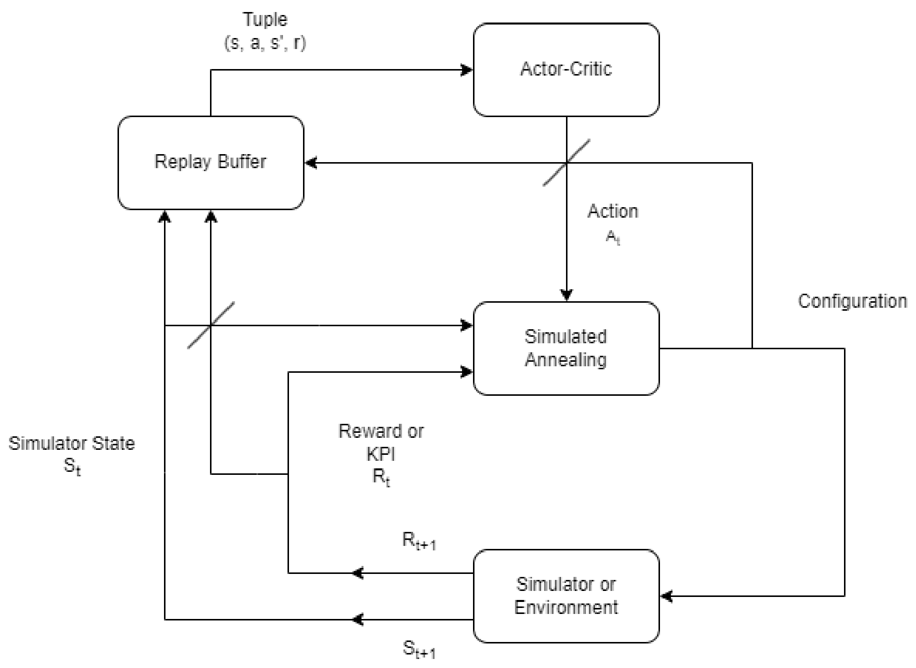

Fig. 3.5. Training and usage cycle for simulated annealing aided by an actor-critic algorithm

3.4. Evaluation

In order to evaluate the algorithms described above, we used our own setup composed of hand-made mock environments. These are simple environments that are made to test out the performance and viability of an algorithm. The details of the mocks are shown later in this section.

We tested the validity and performance of the algorithm described above in two simple synthetic environments. In both of them, we start at a random state s and the goal is to get to a goal state g. For both mock environments, the metric to evaluate how an algorithm performs in the environment uses the Euclidean distance. To make a result of 1.0 the best possible result for any algorithm, we changed the metric as seen below:

$$\begin{aligned} value &= 1.0 - d(s, g) \\ &= 1.0 - \sqrt{\sum_{i=0}^{n}(s_i - g_i)^2} \end{aligned}$$

The value represents how valuable a certain state or observation is for the algorithm. The higher the value, the better it is to reach a certain state s. In our evaluations, most states near the goal state g tend to have higher values than states further from g as the larger the distance between the current state and goal state, the lesser value the algorithm gets.

The first mock environment is an environment set in a three-dimensional space with the goal state being set in advance. The expectation is that as the algorithm gets closer to the goal state, the metric used to evaluate the algorithm will come closer to 1.0. The environment is simple and every algorithm tested on the environment achieved good results as expected. The main utility of this environment is to get quick feedback on whether the algorithms were implemented correctly and if they have some potential.

Algorithms	Hill Climbing	Simulated Annealing	Simulated Annealing with TD3
Configuration Values	0.9998	0.7560	0.9703

Table 3.2. Average configuration values found by each algorithm within mock environment

As seen in Table 3.2, the Simulated Annealing with TD3 performs really well in the mock environment. With 1.0 being the maximum attainable value, it reaches close with a 0.9703 score on average. Hill climbing also performs great while Simulated Annealing performs more poorly. These results are to be expected as our version of Hill Climbing iterates through many neighbors allowing for a gradual increase in its score and hence it gets a near-perfect score. Meanwhile, Simulated Annealing randomly samples configurations, meaning that for some tryouts, it is unable to attain neighboring configurations that are near the goal state. With an actor-critic algorithm to aid it, however, the actor already pushes the Simulated Annealing algorithm in the direction of the goal state which results in a better performance.

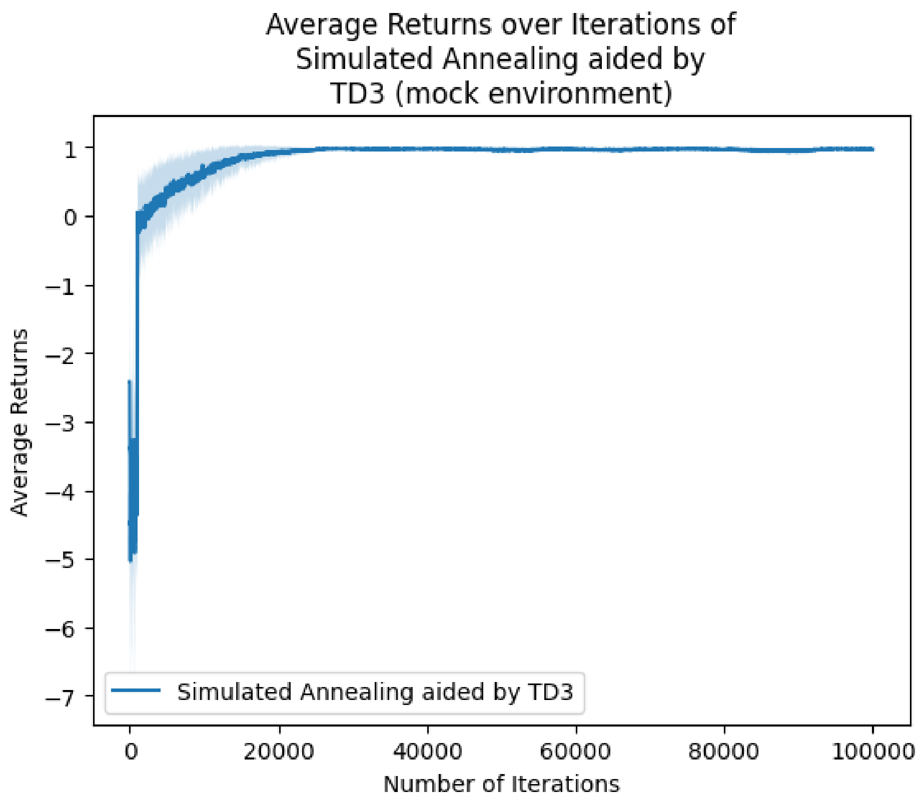

Fig. 3.6. TD3 training curve with Simulated Annealing in mock environment

Figure 3.6 shows the training curve of the Simulated Annealing with TD3 algorithm. It learns within 25,000 iteration how to get to the goal state with a consistent performance for the rest of the training cycle.

The second environment is very similar to the first except that it is set in a two-dimensional space, with the goal state being surrounded by a red zone which returns a value of -0.1 when the algorithm enters it. The goal of such an environment is to see if the algorithms can solve a more complicated environment where the path to the goal state is not straightforward.

Algorithms	Hill Climbing	Simulated Annealing	Simulated Annealing with TD3
Configuration Values	0.7504	0.7590	0.7790

Table 3.3. Average configuration values found by each algorithm within mock_v2 environment

As seen in Table 3.3, the performance of each algorithm is milder. Due to the goal state being surrounded by a zone that gives a negative score, Hill Climbing is unable to perform as well as in the first mock environment where getting to the goal state is fairly straightforward. Simulated Annealing does not suffer from the same issues as the hill climbing algorithm, but nonetheless performs just as worst as it in the first mock environment. Simulated Annealing with TD3 also performs more poorly than in the first mock environment, but due to it being guided by an actor, performs better than vanilla Simulated Annealing, by sometimes reaching the goal state.

Algorithms	Hill Climbing	Simulated Annealing	Simulated Annealing with TD3
Average Run Time (seconds)	0.2477	0.14507	0.3357

Table 3.4. Average run time for each algorithm within mock_v2 environment

The Table 3.4 shown above contains the average run time of each algorithm. Vanilla Simulated Annealing is the fastest, however as indicated in the previous tables, performs less well than its competitors. SA is able to find solutions in a quicker manner as its search is constrained to the number of neighbors it visits. Meanwhile, hill climbing performs a more exhaustive search resulting in poor run-time performance. SA with TD3 is the slowest algorithm when running in the mock environments, as additional time is needed for the actor-critic to find a suitable starting configuration from which to explore neighbors. However, in a situation where the simulation time in an environment is much larger, this run-time overhead is negligible as the cost of a forward pass of an actor-critic algorithm is relatively small compared to the computation time required to simulate the environment when choosing a certain neighbor.

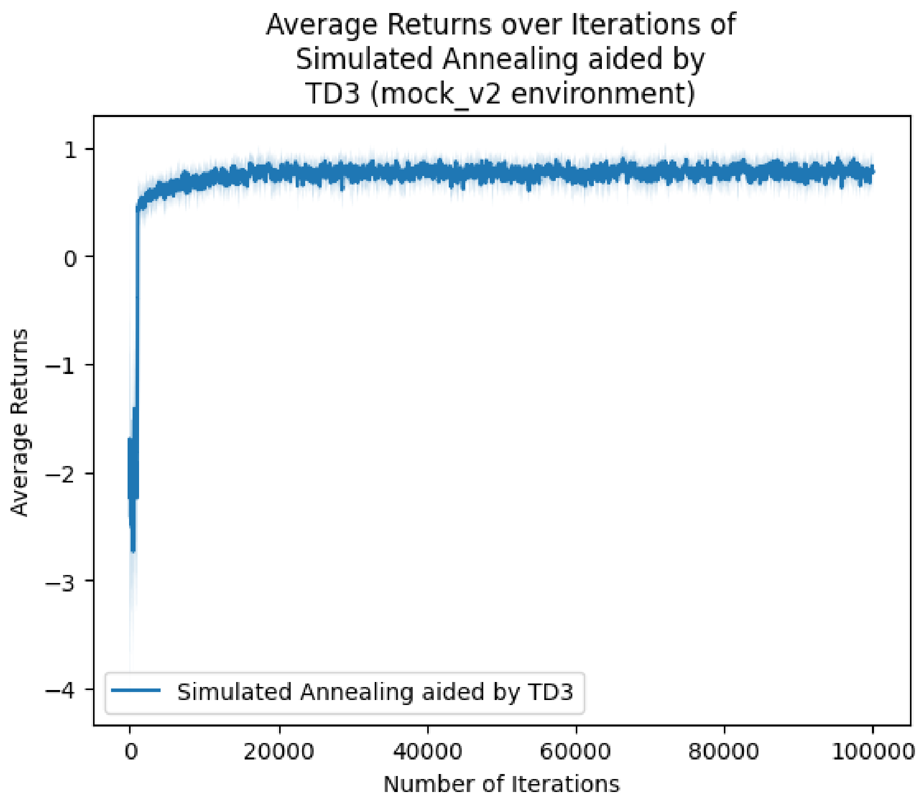

Fig. 3.7. TD3 training curve with Simulated Annealing in mock_2 environment

Figure 3.7 shows the training curve of the Simulated Annealing with TD3 algorithm. It learns within 20,000 iteration how to get to the goal state but oscillates between being within the boundaries of the goal state and outside it, due to the goal state being surrounded by a zone that gives a negative score.

Through the usage of those mock environments, the goal was to test each algorithm's implementation and see whether they could potentially solve larger and more complicated problems by first being able to solve smaller and simpler ones. Regarding the actor-critic algorithm of choice for Simulated Annealing, we chose TD3. We made the choice of TD3 due to it being one of the state-of-the-art actor-critic algorithms while also not being too difficult to implement.

As previously mentioned, each configuration controls the room's day and night temperature, the intensity and spectrum of lights in the room, as well as the room's day and night relative humidity. These values are passed to the simulator which does the necessary interpolation between the current values and the requested values to achieve the desired result within the simulator's room. Once a configuration is sent to the simulator, it uses the same configuration for the entirety of the growth phase it currently is in. The simulator then returns a value that indicates how well the configuration performed during the growth phase. To move on to the next growth phase, we call the function "run" with the desired configuration, the state of the plant, and the state of the room passed in as a parameter. To backtrack to the desired phase, we simply need to pass the previous state of the simulator as input. Unfortunately, we weren't able to produce results with the simulator, as it is currently incomplete, has a fairly long run time, and is not validated. The simulator still currently lacks the computation needed to compute the KPI for each given phase.

Generating a recipe is only part of the work needed to increase yield in a suitable manner. Control must also be done in order to ensure that the recipe is followed by the letter. In the subsequent chapter, we will discuss reinforcement learning algorithms, how substantial improvements can be made during training, and lastly how their application can benefit the task of control.

Chapter 4

Enhancing Exploration with Simulated Annealing

In addition to generating recipes from which the farm's operator can follow, we also explored the possibility of implementing a real-time control algorithm for the farm's control system. Once the recipe is generated, a control algorithm will be required to keep the parameters within the values specified by the recipe. For that purpose, we will be presenting an improvement of control-oriented reinforcement learning algorithms to act as control algorithms. As such, the purpose of this chapter is to present a solution for the improvement of reinforcement learning methods such as Actor-Critics methods by giving them a better and more humanlike exploration strategy for the eventual purpose of using these methods as real-time control algorithms. We first detail the background necessary to understand our work, followed by the contribution itself, and the results of our experiments.

4.1. Uncertainty in Deep Reinforcement Learning

There are two sources of uncertainty in deep reinforcement learning: epistemic aleatoric and epistemic uncertainty [**21**]. Aleatoric uncertainty stems from the stochasticity of the environment. This kind of uncertainty can be caused by stochastic observations. Stochastic observations can be found in environments where the agent sees imperfect information, such as when there is a fog of war. Randomness within transition dynamics also causes observations to become stochastic. On the other hand, epistemic uncertainty stems mainly from the side of the agent. It is tied to the exploration versus exploitation trade-off, to find a better policy than its current one, the agent must decide to explore instead of exploiting using its current policy. High epistemic uncertainty is caused by the lack of knowledge of the agent regarding its present observations. To reduce this uncertainty, the agent must therefore explore and gather more data concerning those observations to be able to know which actions would benefit it.

One way of estimating epistemic uncertainty is through the use of dropout layers within a neural network architecture. Dropout reduces overfitting in a neural network by randomly dropping certain neurons to 0 during training. However, for deep reinforcement learning, the usage of dropout layers usually results in poor performance due to the lack of a stationary target during training. Another way to estimate epistemic uncertainty is through the use of ensembles. Ensembles use multiple neural networks trained on the same data but initialized with different weights [**19**]. By computing the variance of the predictions of each network given the same input, we get an estimate of the epistemic uncertainty. Since the neural networks are initialized with different sets of weights, the variance between predictions should be relatively small for a given input that they have already seen. For inputs on which the networks have not been trained, the variance between predictions should be significant.

4.2. Simulated Annealing-assisted Actor-Critic

Our algorithm Simulated Annealing-assisted Actor-Critic (SAAC) is an algorithm that performs enhancements to the way the agent explores its environment. To help the agent better explore the environment we first add a component to the agent to estimate the epistemic uncertainty surrounding its actions. Simulated Annealing is then used in conjunction with uncertainty estimation to guide the agent in choosing better actions.

4.2.1. Estimating Epistemic Uncertainty

We chose to use ensembles to estimate epistemic uncertainty for deep reinforcement learning instead of using dropout as it would result in poor results. We could either choose to make an ensemble of actor networks or an ensemble of critic networks. We chose the latter as the output of the critics represents how good is it to take action a when in state s, the critics are an estimation of the action-value function $Q(s, a)$. In addition, unlike a critic network, actor networks can output multiple values depending on the dimensionality of the action space. As such calculation, the variance of the critics is simpler and less expensive computationally. As previously mentioned, we create multiple critics, each initialized with a random set of weights. Each critic is trained using the same batch of data (s, a, s_{t+1}, r tuples) fetched from a replay buffer. To estimate the epistemic uncertainty of taking action a in state s, we compute the variance between the $Q(s, a)$ prediction of the critics.

4.2.2. Modifications of Actions taken by the Agent

No changes are made to the actor. Similarly to most actor-critic methods, noise is added to the actions predicted by the actor. However, those actions are then passed to our Simulated Annealing algorithm to enhance the exploration done by our agent as shown in Algorithm 4.2.1. We calculate the variance of taking action a given the current observation

s of the agent and add it to the agent's estimate of its state-action value as the value of the current action. Simulated Annealing is then used to find actions a' that have a higher state-action value than the previously computed value. The idea behind it is that if the agent has never tried action a with its current observation, adding the variance to the state-action value would persuade it to explore and execute action a instead of choosing to exploit. However, if the state-action value is low and there is little epistemic uncertainty regarding a, the agent should instead proceed with executing an action that would benefit it better in the long run, with the possibility of exploring a better action along the way. Therefore every iteration, the actor outputs an action, noise is added to it, and it is passed to Simulated Annealing as demonstrated in Figure 4.1. Simulated Annealing is used to choose whether it is worth executing the action for exploration purposes or if is it better to exploit to be able to make further progress within the environment.

Algorithm 4.2.1. Simulated Annealing-assisted Actor-Critic

```
Input: state $s$, action $a$, action-value function $Q(x, y)$, and variance function $Var(x, y)$
$best\_action = a$;
$best\_action\_value = Q(s, a) + Var(s, a)$;
$action\_value = Q(s, a) + Var(s, a)$;
for $i$ in $K$;
    $a' \sim \mathcal{N}(a, \sigma^2)$;
    $T := temperature(i, T)$;
    $advantage := Q(s, a') - action\_value$;
    if $P(advantage, T) > random(0, 1)$;
        $a := a'$;
        $action\_value := Q(s, a')$;
    if $Q(s, a') > best\_action\_value$;
        $best\_action := a'$;
        $best\_action\_value := Q(s, a')$;
end;
return $best\_config$;
```

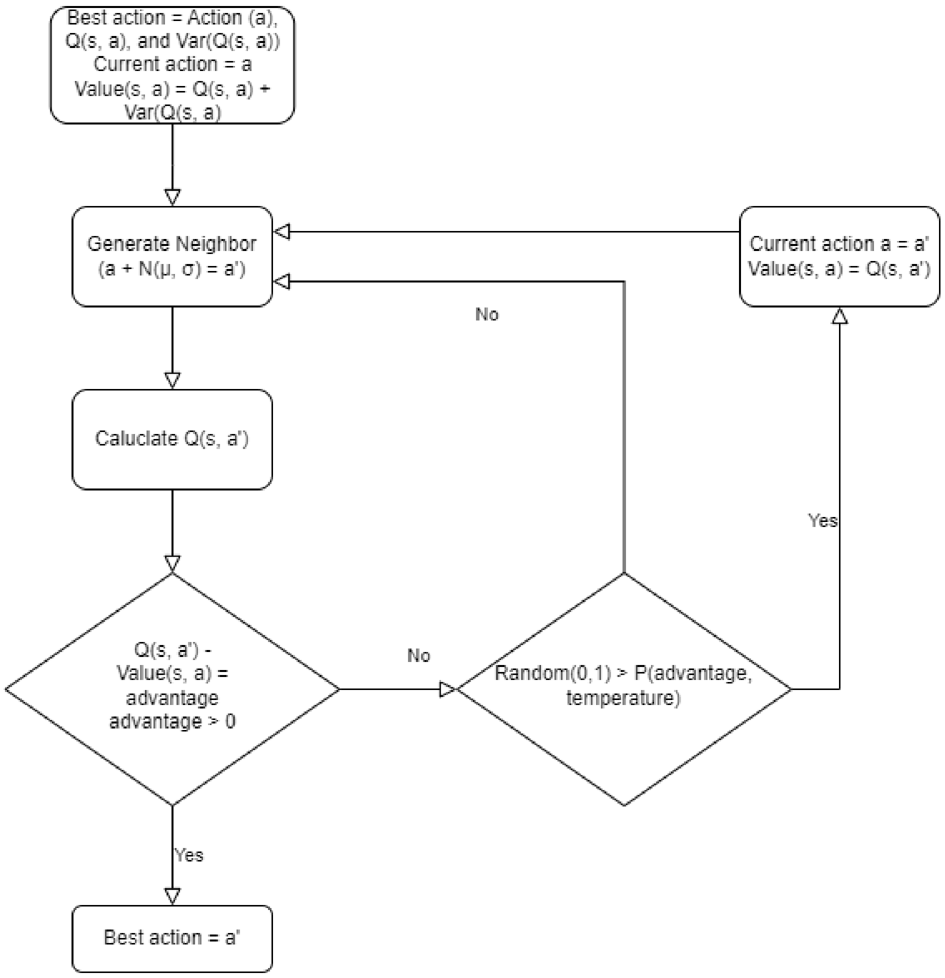
Best action = Action (a), Q(s, a), and Var(Q(s, a))
Current action = a
Value(s, a) = Q(s, a) + Var(Q(s, a)
Generate Neighbor (a + N(μ, σ) = a')
Current action a = a'
Value(s, a) = Q(s, a')
No
Caluclate Q(s, a')
Yes
Q(s, a') - Value(s, a) = advantage
advantage > 0
No
Random(0,1) > P(advantage, temperature)
Yes
Best action = a'

Fig. 4.1. Flowhchart of the action selection process done by our Simulated Annealing algorithm

Figure 4.2 demonstrates how SAAC operates. It acts very similarly to the actor-critic algorithms explained such as DDPG and TD3. The difference is that action selection is not only done by the actor but also by a Simulated Annealing component which decides whether to explore or exploit given the current state of the environment and the agent's familiarity with the said state.

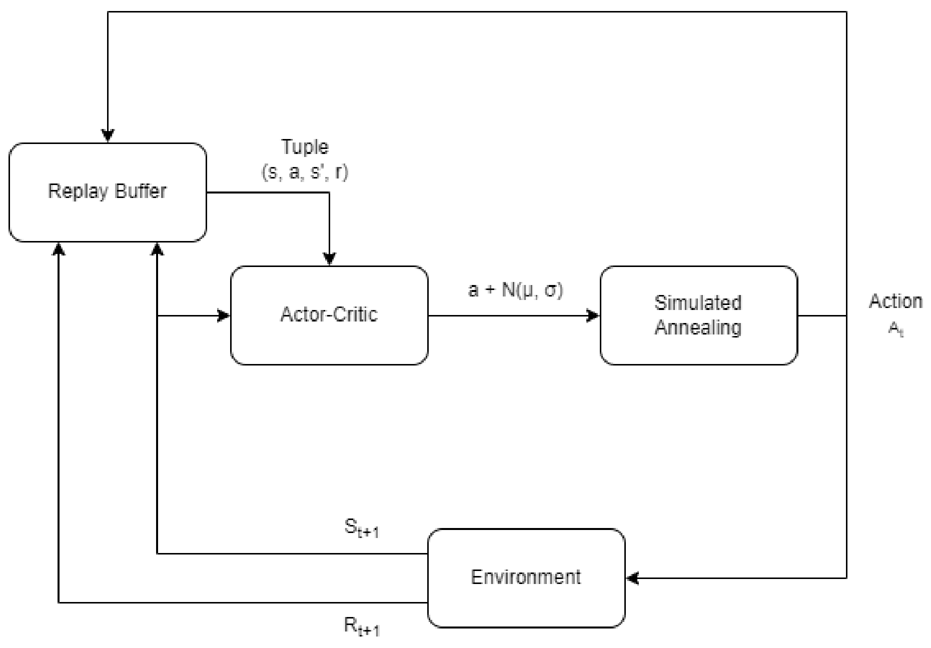

Fig. 4.2. Cycle representing the training and interaction process between the agent which is composed of a replay buffer, an actor-critic algorithm, and a Simulated Annealing component with the environment

While Figure 4.1 demonstrates how the algorithm explores and chooses which action it should take in order to either enhance its exploration of the environment or increase the amount of reward it accumulates, Figure 4.2 shows how SAAC is trained and deployed to solve control oriented problems.

4.3. Evaluation

We tested our algorithm using OpenAI's Gym environments [**3**]. These environments use MuJoCo (Multi-Joint Dynamics with Contact) as a physics engine to help research done in many domains notably robotics. These environments are usually used as benchmarks for the DRL community to test and compare the performance of various algorithms. Most environments are considered difficult to solve as the state and action space are continuous.

4.3.1. Results in the Inverted Pendulum environment

The inverted pendulum is an OpenAI's Gym environment where the agent is a fixed pole. As actions, it can either lean left or right (-3.0 to 3.0). As observations, the agent is shown the positional values of its body parts and the velocity of those parts. The goal of the agent is to be able to balance the pole. The length of an episode in the environment is 1,000 steps. The episode ends and the environment resets when either the 1,000 steps are completed or the pole falls down. The agent is rewarded 1.0 as a reward for every iteration it is able to balance itself. As such, the maximum amount of reward an RL agent can get for a single episode is 1,000 [**2**].

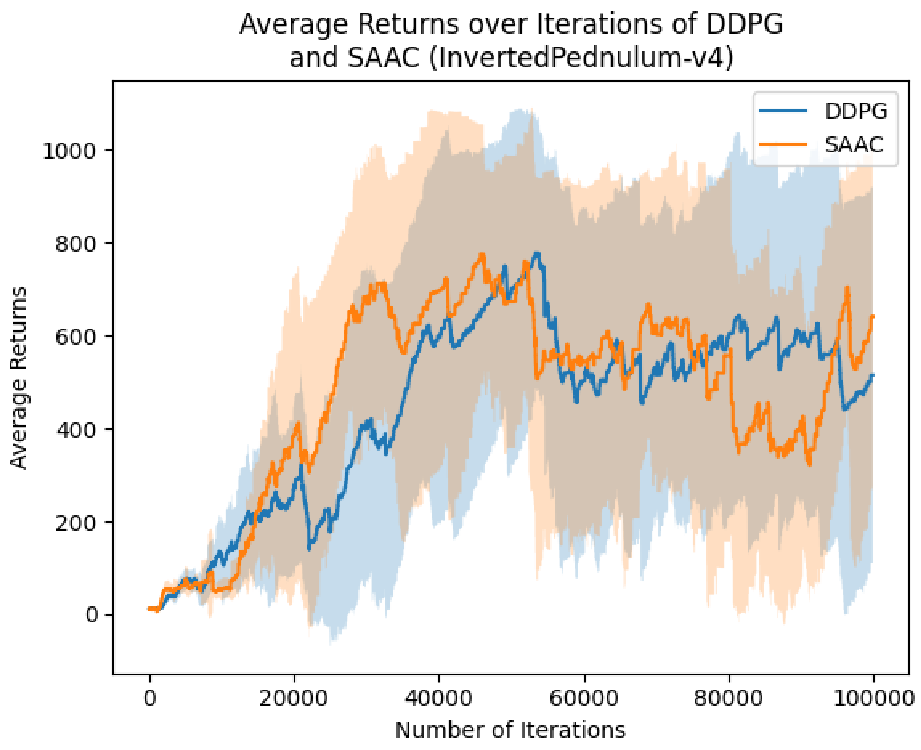

Fig. 4.3. Comparaison between the average performance of DDPG and SAAC in the Inverted Pendulum environment

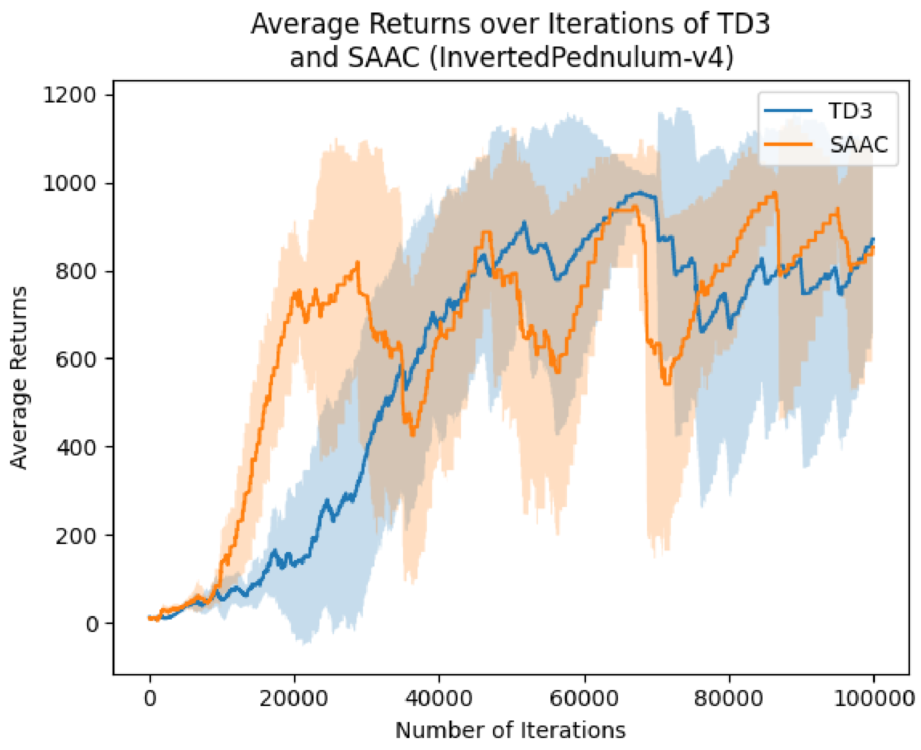

Fig. 4.4. Comparaison between the average performance of TD3 and SAAC in the Inverted Pendulum environment

Figure 4.3 and Figure 4.4 show how SAAC perform with respect to each vanilla actor-critic and their enhanced versions in the Inverted Pendulum environment. We can see that our algorithm performs marginally better than vanilla DDPG and TD3. All of the algorithms shown are able to fetch the maximum amount of reward within a single episode. However, with SAAC performing better exploration, it is able to achieve the threshold of reward faster than the other algorithms. Overall, we cannot conclusively say that SAAC outperforms either DDPG or TD3 by running them within the Inverted pendulum environment.

4.3.2. Results in the Half Cheetah environment

We decided to choose a second environment with which to test SAAC and compare the results with vanilla DDPG and TD3 again. A more challenging OpenAI Gym environment is the Half Cheetah. The environment consists of a 2-dimensional cat-like robot with the agent controlling its joints. The goal of the agent is to move forward as fast as possible, it receives positive rewards for moving forward and negative ones for moving backward. Unlike the previous environment, the action space has six dimensions, and the observation space has 17 dimensions. The agent can apply torques to the joints of its body and it sees the positional values of its body parts as well as their velocities. The episode ends when 1,000 steps were achieved by the agent [**1**].

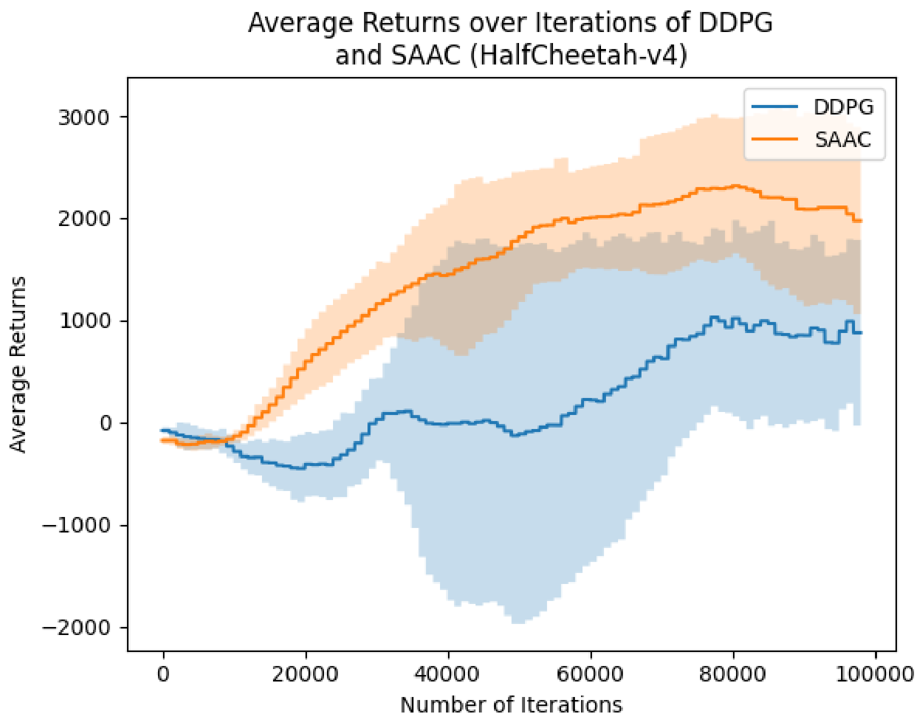

Fig. 4.5. Comparaison between the average performance of DDPG and SAAC in the Half Cheetah environment

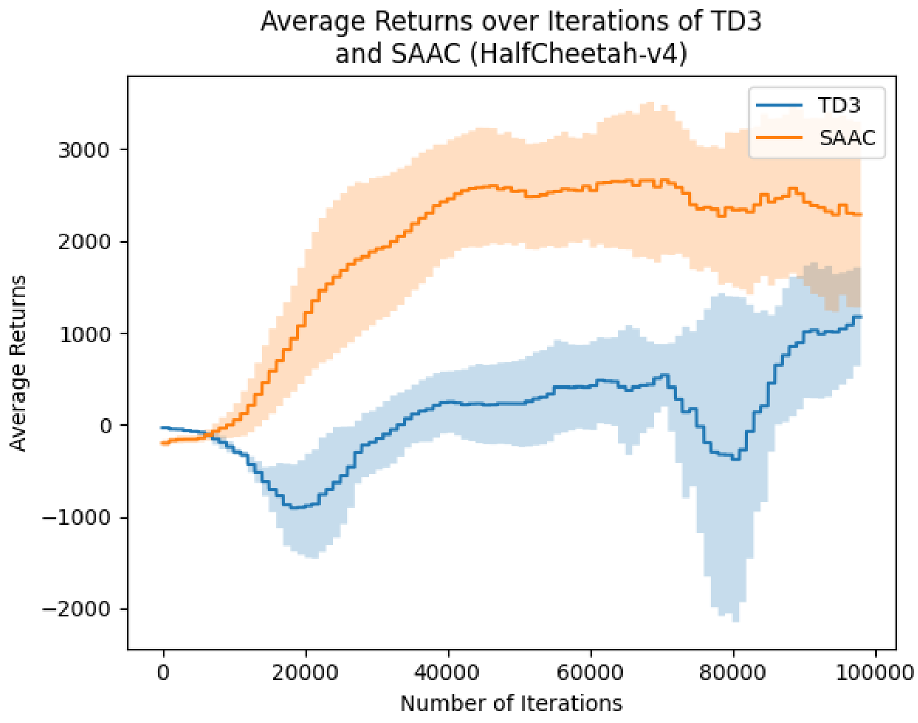

Fig. 4.6. Comparaison between the average performance of TD3 and SAAC in the Half Cheetah environment

Figure 4.3 and Figure 4.4 show how SAAC perform with respect to each vanilla actor-critic and their enhanced versions in the Half Cheetah environment. Unlike in the previous environment, the complexity of learning good policy in the Half Cheetah environment is greater. As such good exploration plays a huge role in finding a good policy with which the agent can perform well. Similarly, as with the Inverted Pendulum, our agent learns a good policy substantially quicker than vanilla DDPG and TD3. In addition, with better exploration SAAC is able to outperform both algorithms substantially, achieving an average reward of over 2000 when coupled with both DDPG and TD3. Meanwhile, vanilla DDPG and TD3 are, on average barely, able to achieve over 1000 rewards.

SAAC is an algorithm that improves actor-critic methods by allowing agents to explore more efficiently. Agents are incentivized to explore when they are unsure about the consequences of taking a certain action when observing the current state of the environment but are pushed to exploit when they have nothing to gain by trying actions that they know don't benefit them. We have shown that it has some potential by comparing the performance of the algorithm against state-of-the-art algorithms such as DDPG and TD3. The SAAC's actor is able to learn a good policy faster, and in the case of the Half Cheetah environment, is able to outperform both baselines. Unfortunately, we were not able to test it with the simulator commissioned by Ferme d'Hiver as it is neither fully operational nor ready for the application of real-time control.

Chapter 5

Conclusion

Our first proposal concerns the usage of Stochastic Hill Climbing with Pruning (SHCP) to find multiple good configurations for each plant growth phase and generate recipes out of them. However, due to the large number of neighbors generated by SHCP even after pruning is done, with each neighbor requiring simulation and evaluation from the simulator, and the fact that the simulator took a large amount of time to simulate growth phases, it was decided that another algorithm was required to find decent configurations. We then proposed using Simulated Annealing assisted with an Actor-Critic algorithm for the means described above. We found that our new algorithm could perform almost as well as SHCP while needing to explore fewer neighboring configurations than SHCP. Unfortunately due to the state of the simulator, our work could neither be applied nor validated for the controlled vertical farm project.

We also attempted to explore using real-time control algorithms for usage in the setting of a controlled vertical farm. We presented our second contribution, Simulated Annealing-assisted Actor-Critic (SAAC), an enhancement that can be used for Actor-Critic algorithms during training. We added the ability for the agent to estimate epistemic uncertainty related to its actions through the usage of deep ensembling of its critics. We then also extended its policy by adding Simulated Annealing to the actions predicted by the actor. By using Simulated Annealing to take into consideration the epistemic uncertainty and value of its actions, the agent could better explore its environment by choosing to explore actions that are deemed to have high epistemic uncertainty while discarding actions that either had little value or did not permit it to explore its environment efficiently. From the results, it was seen that SAAC's better exploration resulted in an increase in the performance of the agent when compared to vanilla DDPG and TD3. Unfortunately, due to the simulator's progress, we are also unable to test our results for the application of real-time control in the scenario of vertical farming.

One of the major limitations of our work is the evaluation of our contributions due to the inoperable state of the vertical farm's simulator. In addition, the KPIs used to evaluate the configurations are at the moment quite simple. The current KPI used to evaluate configurations is the strawberry yield (kg) per amount of energy consumed (kWh). As previously discussed this KPI is used for the final phase of the plant's growth, but due to a lack of data needed to validate the KPI of previous phases, it is also used for rooting and flowering phases. Fortunately, most of our contributions can be used to solve other problems. As such, it would be interesting to see how SAAC performs in more challenging control problems such as robotics. In the future, work can be done to integrate the ideas present in this thesis to a functionning simulator in orer to produce a digital of a vertical farm by recomending new recipes to try out or even help manage daily disturbances in the plant environment through the use of the control algorithm.

References

[1] OpenAi's Gym Documentation Website for Half Cheetah. `https://www.gymlibrary.dev/environments/mujoco/half_cheetah/`. Retrieved: June 20, 2023.

[2] OpenAi's Gym Documentation Website for Inverted Pendulum. `https://www.gymlibrary.dev/environments/mujoco/inverted_pendulum/`. Retrieved: June 20, 2023.

[3] OpenAi's Gym Documentation Website for MuJoCo. `https://www.gymlibrary.dev/environments/mujoco/`. Retrieved: June 20, 2023.

[4] Hani Abdeen, Dániel Varró, Houari Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs et Ákos Horváth : Multi-objective optimization in rule-based design space exploration. *In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 289–300, 2014.

[5] Pascal Archambault, Istvan David, Eugene Syriani et Houari Sahraoui : Co-Simulation For Controlled Environment Agriculture. *In 2023 Annual Modeling and Simulation Conference (ANNSIM)*. SCS, 2023. DOI: TBA.

[6] Leemon Baird : Residual algorithms: Reinforcement learning with function approximation. *In Machine Learning Proceedings 1995*, pages 30–37. Elsevier, 1995.

[7] Christian Blum et Andrea Roli : Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308, 2003.

[8] João M.P. Cardoso, José Gabriel F. Coutinho et Pedro C. Diniz : Chapter 8 - additional topics. *In* João M.P. Cardoso, José Gabriel F. Coutinho et Pedro C. Diniz, éditeurs : *Embedded Computing for High Performance*, pages 255–280. Morgan Kaufmann, Boston, 2017.

[9] Alvaro H. C. Correia, Daniel E. Worrall et Roberto Bondesan : Neural simulated annealing, 2022.

[10] Istvan David, Pascal Archambault, Quentin Wolak, Vinh Vu, Timoth'e Lalonde, Kashif Riaz, Eugene Syriani et Houari Sahraoui : Digital Twins for Cyber-Biophysical Systems: Challenges and Lessons Learned. *In ACM/IEEE 26th International Conference on Model-Driven Engineering Languages and Systems (MODELS)*. IEEE, 2023.

[11] Scott Fujimoto, Herke van Hoof et David Meger : Addressing function approximation error in actor-critic methods, 2018.

[12] Ian Goodfellow, Yoshua Bengio et Aaron Courville : *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[13] Eric J Hartman, James D Keeler et Jacek M Kowalski : Layered neural networks with gaussian hidden units as universal approximations. *Neural computation*, 2(2):210–215, 1990.

[14] Anil K Jain, Jianchang Mao et K Moidin Mohiuddin : Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.

[15] Eunsuk Kang, Ethan Jackson et Wolfram Schulte : An approach for effective design space exploration. *In Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems: 16th Monterey Workshop 2010, Redmond, WA, USA, March 31-April 2, 2010, Revised Selected Papers 16*, pages 33–54. Springer, 2011.

[16] Scott Kirkpatrick, C Daniel Gelatt Jr et Mario P Vecchi : Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[17] Mykel J Kochenderfer et Tim A Wheeler : *Algorithms for optimization*. Mit Press, 2019.

[18] Vijay Konda et John Tsitsiklis : Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[19] Balaji Lakshminarayanan, Alexander Pritzel et Charles Blundell : Simple and scalable predictive uncertainty estimation using deep ensembles, 2017.

[20] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver et Daan Wierstra : Continuous control with deep reinforcement learning, 2019.

[21] Owen Lockwood et Mei Si : A review of uncertainty for deep reinforcement learning, 2022.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra et Martin Riedmiller : Playing atari with deep reinforcement learning, 2013.

[23] Christos Pylianidis, Sjoukje Osinga et Ioannis N. Athanasiadis : Introducing digital twins to agriculture. *Computers and Electronics in Agriculture*, 184, 2021.

[24] A VENKATESWARA RAO^1, GAV RAMACHANDRA RAO^1 et MANDAVA V BASAVESWARA RAO : Coping and limitations of genetic algorithms. 2008.

[25] Raul Rojas et Raúl Rojas : The backpropagation algorithm. *Neural networks: a systematic introduction*, pages 149–182, 1996.

[26] Stuart J Russell : *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford et Oleg Klimov : Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[28] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra et Martin Riedmiller : Deterministic policy gradient algorithms. *In* Eric P. Xing et Tony Jebara, éditeurs : *Proceedings of the 31st International Conference on Machine Learning*, Proceedings of Machine Learning Research, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[29] Richard S Sutton : Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

[30] Richard S Sutton et Andrew G Barto : *Reinforcement learning: An introduction*. MIT press, 2018.

[31] Sebastian Thrun et Anton Schwartz : Issues in using function approximation for reinforcement learning. *In Proceedings of the Fourth Connectionist Models Summer School*, volume 255, page 263. Hillsdale, NJ, 1993.

[32] Kuan Chong Ting, Tao Lin et Paul C Davidson : Integrated urban controlled environment agriculture systems. *LED lighting for urban agriculture*, pages 19–36, 2016.

[33] Harm Van Seijen, Hado Van Hasselt, Shimon Whiteson et Marco Wiering : A theoretical and empirical analysis of expected sarsa. *In 2009 ieee symposium on adaptive dynamic programming and reinforcement learning*, pages 177–184. IEEE, 2009.

[34] Christopher JCH Watkins et Peter Dayan : Q-learning. *Machine learning*, 8:279–292, 1992.