Edward Groom

Reinforcement learning for building control with EV charging

Computer Science Tripos – Part II

Pembroke College

June, 2023

Declaration of originality

I, Edward Groom of Pembroke College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed Edward Groom

Date May 9, 2023

Proforma

Candidate Number:	**2258E**
Project Title:	**Reinforcement Learning for building control with a bi-directional EV charger**
Examination:	**Computer Science Tripos – Part II, June, 2023**
Word Count:	**12600**[1]
Code Line Count:	**1000**
Project Originator:	**Prof Srinivasan Keshav**
Supervisor:	**Prof Srinivasan Keshav**

Original Aims of the Project

Work Completed

All that has been completed appears in this dissertation.

Special Difficulties

Learning how to incorporate encapulated postscript into a LaTeX document on both Ubuntu Linux and OS X.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Contents

List of Figures

Acknowledgements

In developing and writing this project, I have had support from many people. Most of all, I am grateful for the support from:

- **Prof Srinivasan Keshav:** for their advice, suggestions and feedback throughout this project, as well as inspiring me with endless possibilities for further exploration.
- **My parents:** for their constant support and care, throughout the difficulties faced.

Chapter 1

Introduction

1.1 Motivation

As the number of electric vehicles (EVs) continues to grow on our roads and in our markets, we expect to see decreasing levels of vehicular greenhouse emissions, more investment in electric infrastructure and a more stable grid from customers' demand-response. At the same time, an influx of EVs in a particular area with undesirable charging patterns could lead to new high peak load levels or wasted PV generation.

The problem of when and how to charge your EV is not an easy system to model, and so it would be well suited by a model-free deep reinforcement learning control algorithm which can learn to make good charging decisions on behalf of the EV owner. There are many objectives that the consumer might care about: overall money spent to the grid, maximum usage of household PV generation, maximising grid cooperation (usage at off-peak times), or perhaps simply minimising the time you have to wait to fully charge the EV.

1.2 Problem overview

The original inspiration of the project came from the list of project suggestions from Srinivasan Keshav, related to his work and leadership in the Energy and Environment Group, in the Computer Laboratory, University of Cambridge. It is included in Appendix A. The goal was to use reinforcement learning to develop a control strategy for charging (and potentially discharging) an EV battery in a household containing solar. This paper [14] was included as an example of an approach to take, but has many features that are not very realistic. For example, it makes the assumption of identical, driving habits, and home load conditions every day. Further, this paper does not include residential PV installation.

The goal of this project is to investigate techniques for reinforcement learning control (RLC) of household energy management, in a single residential household with photovoltaic installation (PV supply), and a electric vehicle (EV). The question for the RLC agent to learn to answer is when to charge the EV, and from what source to charge the EV?

I will be reimplementing a deep RLC approach from existing academic literature in order to investigate and reiterate their findings. The RLC algorithm benefits from being implemented in a deep neural network (DNN), as it can learn from an environment even in a model-free setting, as the neural network can find abstract features and connections between its state and input data.

I will be looking through the state-of-the-art techniques and identifying a paper whose results I will try to replicate. A good choice will have thorough evaluation metrics and benchmarks, a main solution with enough contributions to be interesting, but still within the grasp of a Part II project.

To facilitate training and testing of the RLC approaches, a learning environment will be constructed that implements a simulation of household energy load, PV supply, and of course an EV. The learning agents will be able to interact directly with the simulation environment, provide actions, and receive feedback in rewards.

Bi-directional EV charging is the concept of allowing excess energy in the EV battery to be discharged to provide for building demand, or sold back to the grid. The introduction of such a vehicle-to-grid (V2G) system tends to be a progression from the concept of dynamic energy prices throughout the day, as V2G would allow revenue to be made through arbitrage of storing energy from the grid when it is at low cost and selling it back at a higher price. Dynamic pricing is a major strategy employed by grid operators to dampen the high levels of demand fluctuation according to the time of day and the similarly-distributed habits of the customers.

Chapter 2

Preparation

In this chapter I will conduct a literature review to identify a suitable academic work to replicate and investigate.

I will describe my starting point, detail the requirements analysis for the chosen project, and then present a background overview of the topics and tools that I will need to become familiar with.

Finally I will present my specific implementation plan, and the engineering strategies and tools that I will employ.

2.1 Literature review

In order to make a useful investigation into the open-ended topic of reinforcement learning for building control, a literature review of recent academic works was needed to become familiar with the state-of-the-art techniques, as well as examining their quality and suitability as a use case for my own investigation.

The paper that was given as an example in the project description [14] explores profiting from V2G-capable charging and uses Double Deep Q-learning (DDQN). However, there is no consideration of solar generation, and it has a totally unrealistic environment structure where there are the same conditions and vehicle habits each day.

There are many papers which address the problem of model-free real-time scheduling of control of EV charging, but which do not include a solar supply in the home. For example this paper by Wan et al. [28] who's contribution is using a representation network to extract features from the dynamic electricity price, which is concatenated to the input of a deep Q network. There is also this paper by Jaehyun et al. [16] learns a good charging policy, but does not even have household load, as well as solar, so it is not realistic enough to be useful.

Some papers show large-scale systems. For example this paper by Mocanu et al. [18] uses two well-established RLC algorithms, Deep Q-learning and Deep Policy Gradients, to solve cost-minimisation and peak-reduction problems in a building context with multiple consumption profiles where multiple actions have to be taken at the same moment. There is also this paper by Sunyong et al. [24] which uses Q-learning (not deep QL) to reduce operating costs on a smart building, but the complication comes from the fact that there is a very high number of batteries: one in the ESS, on in the Smart Building, and one for each of the many EVs in the charging station.

Some have a good problem description, but poor investigation and presentation. For example this paper by Rabusseau et al. [20] presents a smart building, with an EV, home solar, a home battery, and varying grid energy costs. They propose two solutions: a Neural Fitted Q Iteration, and a Deep Q Network.

The charging control problem we are looking to address is also being solved with methods other than reinforcement learning. This paper by Dunbar et al. [13] has a similar setup, EV, household load, PV generation, varying grid prices, and a V2H infrastructure (discharge EV battery to home, not grid). They achieve great success in reducing operational costs, however, they do not use reinforcement learning but Model Predictive Control, which we will talk more about later.

2.1.1 Chosen paper

As a generalisation, EVs tend to be more expensive as an initial investment than traditional motor vehicles. However, their benefit for the environment and cheaper mileage costs [1]2 encourage the investment into an EV. Very similar arguments are made for installation of Photovoltaic (PV) infrastructure in a home in terms of the investment and savings, so it seemed appropriate to include PV supply in the learning environment, as many EV owners will have invested in solar panels for their home and their vehicle.

Some papers do include bi-directional charging, which is the V2G for V2H concept. While the ability to discharge the EV does provide cost benefits of being able to store excess PV supply and discharge to fulfill household demand, investigation into this concept would not be very visible. The alternative strategy of buying-to-store as much energy as possible from the grid at low cost (instead of storing from PV) to resell later is more unbounded in terms of quantity. The controller would completely charge the battery at low cost, and then it cannot be charged again so any excess solar will be sold after dealing with household demand. Suddenly it has become an investigation into arbitrage revenue where the PV plays little part, as opposed to learning a control strategy of when to charge the EV according to varying availability of PV supply. Because of these reasons, bi-directional control appeared to be an added complication that in fact diminished the interest in the

[2]Drivers charging at home on an electric vehicle (EV) tariff can save over 56% in UK [2] or 62% in Germany [1] per mile compared to petrol or diesel, depending on local markets.

investigation.

Narrowing down to papers considering PV supply, no V2G architecture, and only a single residential household for representational clarity, the focus was now on papers with apparent professionalism in the investigation, and with enough metrics of evaluation to properly compare their solution to existing alternative strategies.

The paper "Deep reinforcement learning control of electric vehicle charging in the presence of photovoltaic generation" [12] by Dorokhova et al. demonstrates expressing the same charging problem in three different ways, such that a range of different deep RL control algorithms can be applied to the different formulations of the same problem, and hence directly compared. A further contribution is that the objective is not only cost minimisation, which is expressed as maximising PV self-consumption (making use of as much PV generation as possible during each episode), but it is a dual-objective of also maximising the EV battery state of charge (SoC) at departure time. These three algorithms are then compared to Rule Based Control (RBC), Deterministic Optimisation and Model Predictive Control (MPC) for comprehensive benchmarking.

2.2 Starting point

My experience with deep neural networks did not extend further than the last two lectures of the Part IB Artificial Intelligence lecture course. In particular, Reinforcement Learning did not feature in this course.

In terms of tools, I started with lots of experience with `Python` and using different libraries, including `Matplotlib`, but I had roughly only 1.5 hours experience using `pytorch`, and not in an academic setting.

I had no knowledge of `Open AI Gym` (Gym) [10] before conducting the literature review and seeing its usage as a simulation environment and as a useful interface for interacting with learning agents. I began also totally unfamiliar with the `Bauwerk` [23] framework, which is building control simulation framework built upon the Gym backbone, and was recommended to me based upon my chosen use case.

2.3 Requirements analysis

The requirements of this project are to emulate the investigation in the chosen paper by Dorokhova et al. [12].

The aim is to investigate the performance of different deep reinforcement learning control algorithms on the same EV charging control environment. First, the simulation environment will be created with `Open AI Gym`, and then the action space will be formulated differently so that the environment can be applied to different algorithms.

These are the three deep reinforcement learning control algorithms and the corresponding formulation of the environment:

1. **DDPG** Deep deterministic policy gradients, which acts on a continuous action space.
2. **DDQN** Double deep Q-learning network, which acts on a discrete action space.
3. **P-DQN** Parametric deep Q learning, which acts of a hybrid discrete-continuous parameterised action space.

The performance of these algorithms will be evaluated by comparison with some trivial charging strategies, and then with baseline techniques which have proved very successful: Rule based control (RBC), deterministic optimisation, and deterministic model predictive control (MPC).

2.4 Background

2.4.1 Machine Learning with Neural Networks

Artificial neural networks (ANNs) are computing systems based on a biological neural network where computational nodes take the place of neurons. The nodes are organised in layers and there are weights between the nodes of connecting layers which are adjusted so that the network can learn to represent patterns and classifications, amongst many other functions. Deep neural networks (DNNs) are a subset of ANNs, where there is a general focus on having many hidden layers between the input and output layers, which allows the network to represent more numerous and more complex features from the input data.

The three main paradigms for machine learning are supervised learning, unsupervised learning, and reinforcement learning. None of these are required to use neural networks, for example supervised learning can be implemented with Support Vector Machines or decision trees, unsupervised learning can be implemented with clustering or dimensionality-reduction techniques, and reinforcement learning have been approached with methods like

Monte Carlo or temporal-difference learning.

ANNs or DNNs are often chosen instead of these methods, or alternatively are used to enhance parts of their implementation. In particular, ANNs can be used to represent a system state when it has high dimensionality, or is not strictly defined, as these factors will cause tabular methods to scale impractically large or simply be incompatible.

Here is a brief overview of 3 main architectures of neural networks with different purposes. Multilayer perceptrons (MLPs) or Feedforward neural networks (FNNs) are used as universal function approximators and so can create mathematical models by regression analysis, and analyse data that is not linearly separable. CNNs (Convultional Neural Networks) make use of convolutional layers which allow local operations on spatially-correlated data, which is especially useful for image features and classification. Finally we have RNNs (Recurrent Neural Networks) and Transformers, which deal especially with analysing and predicting sequential data: Transformers have had notorious success in language modelling, and LSTM (Long Short Term Memory) models (a subset of RNNs) have uses in predicting weather patterns from cyclical temporal data. An instance of LSTMs is used in the prediction stage for the deterministic MPC baseline control strategy.

2.4.2 Reinforcement Learning Control (RLC) and Deep Reinforcement Learning

Reinforcement learning is where an agent ('the learning agent') learns a good policy on how to make control decisions based on interaction with an environment, and observing the evolution of the environment's state and reward feedback from its actions. The agent tends to maintain either a policy function or a value function, optimising with the aim of maximum expected reward. The policy function maintains a probability distribution of best actions for each state, while the value function maintains a relative value for each action for each state.

The game is defined as a Markov decision process, which is a stochastic control process defined by the evolving four-tuple: (S, A, P_a, R_a), where S is the set of possible states (the state space), A is the set of possible actions (the action space), $P_a(s, s')$ is the probability of action a evolving state s to state s', and $R_a(s, s')$ is the reward of this evolution from this action.

The general reinforcement learning algorithm consists of the following:

1. **Action** taken based on state observation and current best-guess policy.
2. **Observation** of new state of environment, and **reward** from the previous action.
3. **Update policy** based on the learned reward from previous observation and action.

The actions are often chosen randomly with a decaying probability, which allows a trade-off between exploration of unknown areas, and aims to avoid getting stuck in sub-optimal

solutions by only sticking to the actions that have proven fruitful so far. The policy itself can be deterministic and only output the best-choice action for the state, or stochastic where the policy provides a probabilistic distribution from which the action is chosen.

The most interesting collection of RLC algorithms are model-free as they do not require a formally defined model of a complex and perhaps abstract state, but model-based methods include Monte Carlo Tree Search and Model Predictive Control (MPC). I will implement deterministic MPC as a comparison to the model-free algorithms.

Common examples of model-free RLC algorithms include Q-learning, SARSA, and Policy Gradients, and Actor-Critic methods. Q-learning and Policy Gradients are model-free, off-policy algorithms that learns a policy to maximise the expected cumulative reward. Q-learning supports discrete action spaces, and learns an optimal action-value function (Q-function) by iteratively updating its Q-values based on the following Bellman optimality equation[3] which steps the Q-function towards the target or expected Q values, denoted by Q':

$$\underbrace{\text{New}Q(s,a)}_{\text{New Q-Value}} = Q(s,a) + \underbrace{\alpha}_{\text{Learning rate}}[\underbrace{R(s,a)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount rate}} \overbrace{\max_{a'} Q'(s',a')}^{\substack{\text{Maximum predicted reward, given} \\ \text{new state and all possible actions}}} - Q(s,a)] \tag{2.1}$$

The discount rate balances immediate with future rewards, and the learning rate controls the size of the step to take towards to estimated more-optimal Q value for this state-action pair. On the other hand Policy Gradients supports continuous actions spaces and optimises a parameterised policy by gradient ascent on the gradient of its objective function, which is the expected cumulative reward. Actor-Critic methods utilise a separate function approximator for learning the policy (actor) and for learning the value function (critic). They aim to leverage the strengths of both value function-based (Q-learning) and policy-based (Policy Gradients) methods.

DRL (Deep Reinforcement Learning) is a class of algorithms which implement RLC with the help of DNNs. The DNNs are often function approximators for the value function, policy function, and the state representation. They are useful for systems with high levels of dimensionality, continuous state spaces, and where there might be complex unpredicted features and relations between the state and input data.

[3]The LaTeX for this equation came from TeX.StackExchange [27], before some adjustments.

Experience replay buffer

All three of our proposal learning agent solution utilise the concept of an experience replay buffer. Every interaction with the environment is stored in this memory, and then the actual training steps come after an interval of environment steps have been observed. This is when the agent samples mini-batches of experiences from the buffer, which it uses to learn from and update its Q-network, or both its actor and critic networks.

Interactions with an environment will always be a sequential process, which means that consecutive samples will have an inherent level or correlation. Correlated samples can have an destabilising effect, and so randomly sampling from the memory breaks these correlations and helps to stabilise the training.

A replay buffer can also help with exploration, as the network are not updated every time new information comes in, but they are allowed to produce many samples from each iteration of the Q-function (or equivalent).

Deep deterministic policy gradients (DDPG)

The DDPG [17] algorithm combines the concept of a Deep Q-learning Network (DQN) and Policy gradients.

DDPG uses the actor-critic architecture described above. The actor network represents the policy mapping to continuous actions, and the critic network implements the Q function for state-action pairs which must be discrete. This is where DDPG uses deterministic policy gradients to update the actor network, by calculating the gradient of the critic network and finding the actor network's gradient from there.

Inspired by DQN, DDPG has a main pair of actor-critic networks, and also a target actor and a target critic network. This allows loss to be calculated during training, and also facilitates the target networks being updated only gradually using a mixing factor τ.

To facilitate exploration, noise is added to the actions produced by the actor network. Ornstein-Uhlenbeck noise is a common choice: it is well-suited for continuous control tasks as it has temporal correlations (the noise looks very similar to Brownian motion).

Pseudocode for the training loop for DDPG is found in Appendix B, B.1.

Double deep Q-learning (DDQN)

DDQN [15] was made to overcome the problem of overestimation bias of Q-values in the original DQN. It solves this problem with the same idea as is used in actor-critic struc-

tures. Action selection and action evaluation are separated and performed by separate Q networks.

The target Q network provides target Q-values from which you can find a loss (from the predicted Q-values) and use this to stabilise the training and update the weights of the main Q-function in a more reliable direction. The parameters of the target network are updated with those from the main network periodically after a certain interval of episodes.

Instead of using noise DDQN (and DQN) uses epsilon-greedy exploration, which is where a random action is taken with epsilon probability. Depending on implementation, epsilon will decay over time.

Pseudocode for the training loop for DDQN is found in Appendix B, B.2.

Parametrized Deep Q-learning (P-DQN)

There are not many RL algorithms that have been designed specifically for hybrid action spaces, but this P-DQN algorithm [22] presented by Tencent AI Lab has the most intuitive implementation, and fits similar properties of the DDQN, DDPG, and general of-policy algorithms.

The P-DQN algorithm is implemented with 2 main networks and 2 target networks. One pair is for the discrete action choice, and the other produces the continuous parameters that belong to those discrete actions. The main network weights are slowly copied over to the target network in small increments each time after training is done at each timestep of the environment, again with the rate parameter τ.

This algorithm in fact implements exploration through an epsilon-greedy policy, but also secondarily with noise (again Ornstein-Uhlenbeck) to explore the continuous aspect.

Pseudocode for the training loop for P-DQN is found in Appendix B, B.3.

Let us take an example to visualise what is meant by a hybrid discrete-continuous parameterised action space. In environment X there are three possible actions that can be taken, and each of them can take values from 0 up to 1, 2, and 3 respectively. The P-DQN agent observes the environment state, and outputs (1, 0, 0.53, 0). The 1 identifies that the second action (index 1) was chosen, and the continuous scale parameter for that action is 0.53. Some implementations keep all the action parameters even of those not chosen, but most choose to pad the action with zeros at the unchosen locations.

2.4.3 Open AI Gym environments

Open AI Gym (Gym), which after Gym v26 migrated to Gymnasium, is a library that provides an interface for single-agent reinforcement learning environments, and acts as a (non-direct) implementation and simulation environment for the Markov Decision Process. The core of Gymnasium is the `Env` (environment) class with four main functions: `make`, `reset`, `step` and `render`. Custom environments are implemented by inheriting the `Env` class, or alternatively through the `gym.Wrapper` class, which can use all of the parent environment's features, but can change results passed to the user, and change user's inputs before passing to the environment. The `gym.make(env_name)` command creates an environment that is keyed by the environment's name, and uses the `init` function inside of the environment, where any setup occurs, but always includes defining the `action_space` and the `observation_space`. Calling `CustomWrapper(env)` on an initialised `env` will then call the wrapper's `init`, always re-initialising the environment before overriding or adding new features.

The `reset` method is called at the end of every learning episode from the learning agent's perspective, and the `step(action)` method takes an action as its input, checks that it is contained inside of the defined `action_space`, runs some simulation and returns the tuple `(observation, reward, terminated, truncated, info)`. `Observation` is the new environment state, the `reward` is from reaching the new state from the old state with this action, `terminated` is a marker that the episode has finished, `truncated` is a marker that a fixed number of timesteps was reached to end the episode, and `info` is a custom field that can pass general information, perhaps for data collection.

Inside `gym.Spaces` there is a collection of Spaces classes (eg `Box`, `Tuple`, `Dict`, ...), which act as sets (formed by cross-product) of possible values. For example, a parameter declared of type `Box([0.], [3.936], (1,), np.float32)` is a numpy array of shape `(1,)`, with elements able to take any floating point values between 0. and 3.936. Elements of `Tuple` and `Dict` Spaces must also be declared typed by a specific Space when the `Tuple` or `Dict` is defined. A `Discrete(n)` object is a singular integer which can take integer values from `0` to `n-1`.

2.4.4 Bauwerk

Bauwerk [23] is described in its documentations as "a meta reinforcement-learning benchmark with building control environments; an attempt to help scale greener building controllers to more buildings". Bauwerk contains a main SolarBatteryHouse environment, that after installation is made by calling `gym.make("bauwerk/SolarBatteryHouse-v0")`. It has the main methods described above for general Gym environments, with the addition of a collection of helper functions, for example `getObservationFromState` and `getPowerFromAction`. The former allows the environment to maintain a complex list of state parameters while the learning agent only sees a subset (the observation), and the

later translates relative power actions to absolute charging power.

The environment contains four electrical components which it simulates: a stationary household battery, a PV installation, a residential electrical load, and an on-demand grid energy supply.

The default action space is `Box([-1.], [1.], (1,), float32)`, which is equivalent to the continuous action space described in **2.5.1 Markov decision process**. The observation space is a `Dict` object, with keys `battery_cont`, `load`, `pv_gen`, `time_of_day`. Furthermore, the `time_of_day` field has shape `(2,)`, as it uses a circular representation of time.

2.5 Planning

2.5.1 Markov decision process

Recap: a Markov decision process is defined by the tuple (S, A, P_a, R_a).

State space

The state space S is prescribed in the paper by Dorokhova et al. [12] to be a continuous 4D space. This is to say that the state is represented by the four-tuple $(PV_t, L_t, D_t, SoC_t$. PV_t and L_t are the PV generation and household energy demand at time t, and D_t and SoC_t are the time until departure and battery relative state of charge at time t.

Action space

The action space A is different for each of the learning agents.

- For DDPG it is a continuous space ranging from -1 to 1, where negative values represent grid-charging, and positive values represent solar-charging. These values are relative to the maximum charging power (P_{max}), and will be scaled up too add to current SoC_t.
- For DDQN the discrete action space is the set of integers from 0 to 21, and they go through the following transformation to become relative again: $(p - 10)/10$.
- For P-DQN, the parametric action space, actions are accepted of the form (choice, p_g, p_s) where $choice \in \{0, 1\}$, and p_g and p_s are grid and solar charging parameters that are chosen to act with by the choice parameter.

Some constraints include requiring the EV to only be charged simultaneously by one power source ($EV_t^G EV_t^{PV} = 0$), whereas the load at time t can be supplied by both ($L_t = L_t^G + L_t^{PV}$).

Further, the priority is to allow household load to use the PV generation, and so $L_t^{PV} = \min(PV_t, L_t)$ is always calculated before the EV charge powers, where we see that $EV_t^{PV} \leq PV_t - L_t^{PV}$.

State transitions

The state evolutions, instead of being a probability distribution mapped from (s_t, a) to s_{t+1}, are evaluated by simulating the environment evolution inside the step function of the environment.

Rewards

Since this is a model-free environment, there is no notion of an immediate calculable reward, and so the reward is both sparse and binary. This means that the reward is always either 0 or 1, and it can only be non-zero at the end of the episode.

The reward is formulated as a multi-objective reward, so the reward is 1 if the EV has both **1.** reached the desired SoC (which is 100%) within a tolerance range epsilon (ϵ), and **2.** achieved a PV self-consumption score (SC_t) again within ϵ tolerance of the maximum possible PV self-consumption (SC_T^{max}).

SC_T is defined as the total amount of PV generation that was used

$$\text{PV Self-consumption}, SC_T = \frac{\sum_{t=t_s}^{T}(L_t^{PV} + EV_t^{PV})}{\sum_{t=t_s}^{T} PV_t} \tag{2.2}$$

while SC_T^{max} is the total amount that could have been used if the EV charge action was always chosen to be solar:

$$SC_T^{max} = \frac{L_{max}^{PV} + EV_{max}^{PV}}{\sum_{t=t_s}^{T} PV_t} \tag{2.3}$$

$$L_{max}^{PV} = \sum_{t=t_s}^{T} \min(PV_t, L_t) \quad , \quad EV_{max}^{PV} = \min\left(\sum_{t=t_s}^{T} \max(PV_t - L_t, 0), [1 - SOC_{t_{arr}}]C_{max}\right) \tag{2.4}$$

EV_{max}^{PV} cannot be larger than $(1 - SoC_{t_{arr}})C_{max}$ as we cannot expect EV charging power from PV generation to exceed the total power needed to charge the battery.

2.5.2 Environment setup

In this section is the plan of the changes that need to be made to the `SolarBatteryHouse` environment, in order to create the new environment (named `newBaseEnvWrapper`) to reflect the specific use case from the paper by Dorokhova et al. [12].

Action space wrappers

Wrappers will be used to change the action space in a different way in different parts of the developmental workflow. For the final new implemented environment, the action space change needs to be implemented as a wrapper around the `newBaseEnvWrapper`. However, action space changes will also be implemented just on the original `SolarBatteryHouse` environment so that the imported learning agents can be checked to be inter-operable with the new action spaces, without having to wait until the full environment is completed.

As indicated above in the Markov decision process definition, the continuous action space will remain `Box(-1,1,(1,),float32)`; the discrete action space will be `Discrete(21)`; and the parametric action space will be `Tuple(Discrete(2),Box(0,1,float32),Box(0,1,float32))`

Setup changes

As the observation space is meant to be a 4D space, we need to convert the `Dict` into a `Box` of shape `(4,)`. Otherwise, the observation space is the same, except that instead of the `time_of_day` parameter, we want a `time_until_departure` field instead. By swapping out the `time_of_day` parameter, we do not have to deal with translating it from a circular 2D representation of time back to 1D.

My wrapper's `init` method is also where configuation changes need to be made, for example the battery capacity and max charge power. The `grid_selling_allowed` config parameter is set to false also. The reward tolerance value is passed in now as a required argument for initialising the wrapper.

Step changes

The step method needs to be re-implemented in my wrapper as the logic for distribution of energy from its sources is different from the original Bauwerk concept, the new reward structure requires a check of episode termination before returning the reward, and also cumulative calculations for PV consumption need to be saved inside of the state.

The battery's chemistry needs to be bypassed, as there is a complex implementation of this component, but Dorokhova et al. dictate a 100% charging efficiency on the EV battery.

Episode changes

Episode length in Bauwerk is 365 days, however for our sparse rewards we want an episode to last one day. In fact the episode starts when the EV returns to the car, at which point the environment determines its time of departure (uniformly between an 7am and 5pm), and hence an initial `time_to_departure`. The episode only ends once the `time_to_departure` reaches 0, at which point the environment is reset and the time is set to the new `time_on_arrival`, which is uniformly between 1 hour after it left and 11pm. It also randomly samples at this point what will be the state of charge on return of the EV to the household.

Additionally, on reset of the wrapper at the end of each daily episode, any features or fields that should remain consistent to the system to maintain continuity should be stored and reloaded after the reset. For example, before resetting we determine the new time upon resetting. However, the required call to `super().reset()` will set the PV and household load data to start back from the initial data index, so after resetting we need to reload the old data index, and set the time of day to be equivalent such that the PV and load data is realistic.

Training and test dataset

To implement the held-out test dataset, a new unique seed can be passed into the environment (and its resets). This would randomise the order of the daily/episodic batches of data to a new sample which has not been seen. These are described as batches as the data modules restart from a new point in the data whenever they reset, depending on the seed (adjusted to the correct time-of-day).

An alternative would be to make the data index jump to a new point whenever it finds itself in the last 60 episodes of the data. This would ensure that this data is not seen at all during training, as opposed to the previous method of simply not seeing the specific order of the days. Then when we are training, there would have to be a similar parameter that ensures that the data index is always inside of the last 60 episodes of data.

The third option would be to go into the Bauwerk source files, find the data and split it manually. Then the wrapper can create its own solar and grid module on being initialised according to an initial argument of it the wrapper will be for training or testing.

Workflow

2.5.3 Learning agent implementations

The learning agents to be used were to be found as pre-constructed implementations that I am able to import and connect to my environments. They could come from respected libraries of RLC algorithms, from a codebase attached to an academic paper, or even as a standalone project on GitHub.

Priorities in terms of identifying a good source for each are as follows: professional well-written code, well-documented, clear demonstrations of example performance, and popular consensus from a large group of users who approve of the implementation. Also important is the ability to extract and store data from the training, and also data from the `info` output during and after episodes, so that training data and example episodes can be plotted to track performance and examine local operations.

It can be verified that the learning agents work by applying them to smaller games that will be easier to learn from.

2.5.4 Evaluation metrics

The purpose of constructing the Gym simulation environment is to be able to evaluate the functionality and performance of the solution.

Specific metrics that are interesting to monitor and visualise graphically include the performance of the agent while it is learning (average episode reward), the rate of improvement of the policy while learning, and the time it takes to converge upon a good policy. After training the model's peak performance can then be evaluated compared to other models and other baseline algorithms.

Training

The initial method of evaluating the performance of the algorithms compared to each other is to train the agents for 2000 episodes, 5 times each with a new random seed. This will allow a mean and standard deviation of the objective episode reward function to be plotted for each algorithm throughout the training process. The complexity of the algorithms is represented by reporting the time taken to conduct the training, for various episode lengths to show the rate of increase and relationship between training time and number of training episodes.

Testing

The second stage involves evaluating the now-pretrained models on a held-out test dataset of 60 episodes. The same test data is passed through the baseline algorithms so that we can compare the performance of the trained models with the more standard techniques. The data can be most informative in the form of a box-plot so that we can visualise the minimum, maximum, sample median, and 1st and 3rd quartiles. These features will be plotted for PV self-consumption, the achieved state of charge, and the total energy bought from the grid (equivalent to the total cost, for constant energy pricing). The execution time for running the algorithms on the test data will also be very informative in terms of how suitable it is for implementation as an online algorithm.

We will also want to visualise sample episodes from the trained models and the baselines, so episodes will be chosen that start with the same initial state of charge and the same episode length so that we can directly compare how the learned policies are behaving.

Maximal performance will be found if I keep a running saved state of the model for the instance when it is achieving maximum performance, as opposed to simple the final state of the model after the 2000 episodes. Early stopping is a common technique to avoid the problem of overfitting a model on training data. It is where the model stops training whenever the validation loss reaches a minimum. Here it is not directly applicable as we do not have a concept of training or validation loss, but we are able to save the state of the model at certain points and reload it later.

Performance benchmarks

Random charging and naive charging are the two most clear choices to use as a baseline to see if an algorithm is doing anything clever. If an agent is perhaps not learning about interacting with the environment at all, then we might expect similar performance to random actions. Further, the agent could be considered useful in real terms specifically if it performs better than naive charging, which is defined by a simple instinctive practicality-based strategy that many users would use if not thinking about the system as a whole.

Rule based control (RBC) is an extension of naive charging by adding some more clever logic which directly addresses attempting to fulfill both of the environment's objectives. Here the rules correspond to delaying charging from the grid for as long as possible, and until it is close to the time of departure try to use solar as much as possible.

Deterministic optimisation is a more rigorous technique that aims to find strictly optimal solutions. A problem is defined using constraints (assuming future system behaviour is known) and an objective function which it wants to maximise, and searches for

the optimal solution by simply considering all possible scenarios and identifying the most profitable set of system variables.

Deterministic model predictive control (MPC) , however, is an iterative strategy which does not know the future state of the system. Instead it maintains predictions of future behaviour, and at each timestep it conducts an optimisation to discover the optimal control actions based on the predicted future state, but then only takes the first action from that optimal set of actions. At each next timestep, it computes a more accurate prediction of the future system state and then discovers optimal actions from solving an optimisation, but again only to carry out the first action from that result.

Strategy for optimisations

Since the optimisations use a constraint-based solver to search the solution-space, instead of interacting directly with the environment I will instead prepare the data generation in advance so that it can be formatted in a way to be easily accepted as optimisation constants.

In addition, start and end timesteps for the episodes will be pregenerated. For example, SOC_{arr} which is a optimisation constant will be held inside an array with one value for each episode that the optimisation will be run for. Inner workings of the step methods have been replaced by the constraints.

2.6 Engineering tools and techniques

2.6.1 Software engineering methodology

Throughout my project I employed an iterative development model. Instead of aiming to construct full solutions and then undergo long debugging processes, small milestones were identified which allow incremental updates to be iteratively tested and evaluated.

This makes it easier to adapt to newly discovered problems and introduce new solution strategies into the workflow. A Waterfall model can be more suitable for project where all requirements are known beforehand, but since most components of the system were new to me, an iterative development model was more sustainable.

2.6.2 Version control and backups

Version control of my project was managed using Git. Changes to my working files were pushed to my remote GitHub repository, with detailed commit messages so that I would be able to recover any particular state of my repository if I needed to backtrack. It also allowed me to work on multiple devices as described in my proposal: my personal desktop and laptop. A secondary backup strategy was that my most recent local repository was consistently backed up on the cloud using Microsoft OneDrive.

2.6.3 Main tools

Open AI Gym / Gymnasium. Gym was chosen as it is very popular as an API for RLC agents to interact with, it has extensive documentation, and it allows the use of Bauwerk.

Bauwerk. Bauwerk is a building control simulation environment, built on top of the basic Open AI Gym environments. It was developed by Arduin Findeis as part of xxx. Many aspects of the environment that I need to implement are present in Bauwerk's SolarBatteryHouse environment, so I am using this as a starting point which I will extend and modify to suit my use case.

Visual Studio IDE. The python code development for this project was done mostly with the Visual Studio integrated development environment. It was chosen because of my familiarity with it, because of the high availability of extensions, and because of its easy integration with Git version control.

2.6.4 Libraries and licences

Software	Licence
Python 3.9 NumPy Matplotlib	PSFL
Gymnasium Bauwerk pytorch-DDQN pytorch-PDQN tqdm	MIT
Pytorch	BSD
pytorch-DDPG cvxpy	Apache 2.0
gurobipy	Academic

Table 2.1: Licences of third party dependencies.

Chapter 3

Implementation

3.1 Code structure

Note: the final repository overview is presented at the end of this chapter. This section presents how I initially planned to organise my files based on how I had planned the development pathways to be carried out.

3.2 Agents on new action space

3.2.1 Changing action space

Changing the environment action space is done easiest with wrappers, and more specifically with an ActionWrapper, which is a subclass of `gym.Wrapper`. Wrappers allow new functionality to be added to an environment without directly changing the environment. For example, observations and rewards can be modified before being returned to the learning agent, and actions can be modified before being passed into the environment's `step` method. More fundamentally, a wrapper can override the `init`, `reset`, and `step` methods to either extend or re-implement them entirely.

Here, large changes to the environment are not being attempted, we simply want the `step` / action-input interface to be altered by defining the action space differently. The simpler extension is making it accept and expect discrete actions, and in other extension it should accept the more complex parametric form of the action (both as defined in **2.5.2**). Basic initial tests of these wrappers are performed by running a loop passing in samples from the environment's action space into the step method.

My attempts at implementing the update started by calling

```
self.action_space = Discrete(21)
```

inside the `init` method for my DiscreteActions wrapper, after calling `super().__init__(env)`. Initial tests were returning assertion errors, because the SolarBatteryHouse environment was checking that the action was included in the action space, but it was still checking the old action space of Box(-1, 1, (1,), float32) instead of the new Discrete(21) action space.

It later became clear that the new action space did not change the underlying environment, but only the visible action space from the outside such that learning agents can sample actions from this new action space. This means that another addition is needed to process the new actions being passed into the wrapper, to translate them back into the old action type so that the step function of the unmodified underlying environment can recognise the valid actions. This translation is easily done inside an `action` method of an ActionWrapper which overrides the direct passing of the action.

Unfortunately, I took a different path. The other solution is to completely re-implement the `step` function inside of the wrapper, in which case the assert to check valid actions now is referring to the new action space. I used the logic that, because the `step` function was going to be completely reconstructed anyway for creating the new environment structure, there was no harm in implementing the action space change by copying over the entire step function into the ActionWrapper.

It worked for the purposes of checking compatibility of the learning agents, but it laid a baseline where I now had three environment definitions (one for each action space) which all needed to be upgraded to the new structure. If the action space changes could have been implemented without overriding the `step` function at all, then there would only need to be one workflow of upgrading the environment to the new system of the household battery in fact being an EV battery with completely different episode structure.

3.2.2 Importing agent implementations

In this section I describe how each agent implementation was chosen and imported and, after familiarising, how they were connected to the ActionWrappers to check compatibility. Many changes needed to the wrappers or agents were anticipated during inspection, but many required debugging and careful planning.

Where there is an error in operating an agent on a wrapped environment, either the details of the agent or the wrapper can be changed. On the one hand adjusting the wrappers meant I had confidence in not breaking the agents, while instead adjusting the agents to fit the wrappers meant that the wrappers could stay consistent with one another, and would not cause divergent development streams of needing to implement the same environment changes on the three different wrappers.

Since I appreciated that I was not currently trying to work towards a final solution, with the low-level goal of seeing if the chosen agents would in principle be compatible with being used to control these environments of different action space formulations, and since (as described above) the action space changes had already initiated these divergent development steams, I initially solved the problem by instituting the changes on the wrappers.

An example of a requirement to make changes to the agents' implementations is the following: the initialisation methods for each of the agents receive the environment's action space and observation space to setup the memory sizes and the network input and output shapes, in order to process the observations and produce actions in the correct form. For example, the DDPG setup uses simply the number of state elements and action options (`env.observation_space.shape[0]`, `env.action_space.shape[0]`) as integers, DDQN calls instead `env.action_space.n` as it is a Discrete Gym space object, and yet MP-DQN calls `env.observation_space.spaces[0]` which requires observation space to an iterable Tuple of Gym spaces.

The initial observation space from Bauwerk is a Dict object, which doesn't have a shape or spaces attribute, so the observation space was changed from a Dict of Boxes to a Tuple of Boxes. Alternatively, if I didn't want to change the wrappers, the calls to .shape, .n, and .spaces above could be replaced by declaring a one-element tuple `( len(observation_space), )`. This shape representation could in fact also be used to represent the alternative Tuple observation space.

Any results from these tests were scrapped as they have no meaning relevant to the project: the intended environment implementation is not yet made. However, it was successfully verified that each of the methods from the imported learning agents will work without errors, even if they do not learn. This process also provided the opportunity to familiarise myself with the structure and specific code of the algorithms being used, before the time scheduled for debugging, making changes, or searching for why it may or may not be learning properly.

DDPG

The implementation chosen can be found here [25] and was written by Guan-Horng Liu from the Georgia Institute of Technology. I chose this implementation because it seems very popular, the files are organised concisely, and sample test performance plots were provided.

The structure of the solution is as follows: The main training and testing loops are in `main.py`. They define Actor and Critic networks, then combine to define the pdqn agent, which also uses a SequentialMemory (made up of RingBuffers), and random process file which defines the OrnsteinUhlenbeckProcess for the noise. They have also provided a normalised-environment wrapper which scales the actions, and an evaluator object for

training evaluation.

Deep Deterministic Policy Gradients is the only RCL algorithm of the three we are investigating that is provided in a respected package for RL algorithms. `StableBaselines3` does provide clean usage, but the API is very abstracted, so tracking the learning parameters and plotting sample episodes would become much more complex than the alternative of having direct access to the training loop where custom data collection can be inserted. Of course this implementation could be implemented, including with hooks, but it seemed best to have increased visibility to the agents and training loops. Another factor is the benefit of keeping the approach consistent with the other agents, who do provide the direct training loop to use.

DDQN

The implementation chosen can be found here [26] and was written by Guan-Felix Schur from ETH Zürich. I chose this implementation because it seems very popular, the files are organised concisely, and sample test performance plots were provided.

DDQN only needs a simple Q-network one, which is simply initiated twice. The memory is instead defined by a Deque (double-ended queue) for fast updates and sampling. This implementation doesn't have save-state methods for the agent, which I implemented by retrieving the main network's parameters at certain intervals.

P-DQN

The Parametric Deep Q-learning implementations are harder to find, and hence there is less freedom of choice between options. Even more, there are no standardised examples of environments with parametric actions that the agent could be tested on, and the few available (and their recommended P-DQN agents) had very different styles of syntax for how to define the action.

This (unchosen) implementation [5] comes from DI Engine, a fairly extensive and well-used generalized decision intelligence engine for PyTorch, but it has no documentation or examples of this algorithm being run.

In the end, the most popular in terms of feedback, usage, and citations was MP-DQN [3] from Craig Bester at the University of the Witwatersrand, Johannesburg, which is related to his paper [9]. This has investigates multipass implementations splitting the action-parameter inputs to the Q-network using several passes, but the extension locations were clear and I was able to strip it down to single pass for my usage.

The PDQNAgent class is built out of defined Actor and ParamActor classes, and the same RingBuffer-based memory as with DDPG. The most interesting thing with this implemen-

tation is the use of very many self-defined wrappers around the environment. For their example with the parametric Goal environment, they use: GoalFlattenedActionWrapper, ScaledParameterisedActionWrapper, ScaledStateWrapper, and a (Gym) Monitor. The flattened actions is just for preference for how what format output the actions, and the scaled actions is unnecessary as my action parameters are already in the range 0 to 1. The scaled state wrapper is most interesting, and in fact does work on my environment after a few changes.

3.3 Construct full wrappers

New observation space keys

This code was written in the file called `wrapperPartial_contObs.py`.

The new observation space has the same fields except that it includes a `time_til_departure` field instead of the `time_of_day` state entry. Before changing the type of the observation space, we must ensure that it contains the right items. At the end of the `step` and `reset` methods, a function called `getObsFromState` is called which constructs the observation object that is returned. It selects fields from the environments state according to a list `obs_keys` in the environment's `cfg` (config) object. When the wrapper is initialised, the `time_of_day` key is removed from this list and the new `time_til_departure` key is added. For now we add a temporary value to the `time_til_departure` entry in `self.state`.

This makes the `getObsFromState` function in the wrapper's methods construct the correct observation. However, on wrapper initialisation it must call a preparatory `reset`, which in turn calls the SolarBatteryHouse (SBH) `reset` method. Inside the parent environment's `reset` method the state is redefined (without the new entry), but `cfg.obs_keys` is not redefined, so an error is found when it tries to retrieve the initial observation in the environment's required preparatory reset. The only clear solution to this is not pretty, and so this issue was logged for later analysis. Inside the wrapper's `reset` method (which is whenever `super().reset()` is called) the `obs_keys` are changed back to the SBH default right before calling the parent `reset` method, and then after is restored back to how the new wrapper expects it.

Continuous observation space

The observation space needs to be a continuous space, so was changed from a `gym.spaces.Dict` type to a `gym.spaces.Box` of shape `(4,)`. Observations of this form will now be 4-element Numpy float arrays, with upper and lower bounds defined by max

and min parameters passed to the observation space's initialisation, namely `statehigh` and `statelow`.

New reset architecture

Now the `time_til_departure` is used to change the episode length. SBH episodes are singular year-long episodes, while my episode is daily. The episode ends when the EV departs from the house at t_{dep} (uniformly sampled from 7am to 5pm), and then the episode starts when the EV returns to the home at t_{arr} (uniformly sampled between 1 hour after departure and 11pm).

At every `reset`, the environment determines when the EV will return home (which becomes the new time), the new t_{dep}, the SoC on return (uniform between 0.2 and 0.5), and it will set the `time_til_departure` field. The `time_til_departure` will then decrement at every environment step until it reaches zero, when it triggers the `terminated` flag which will cause the agent to reset the environment.

The amount of time skipped at each reset is also tracked carefully so that the `solar` and `load` data modules can be reset back to the correct time-of-day in their sequencing.

New action logic and sparse reward

The chosen charge action, along with the PV generation and household load combine together to determine the `RealChargeAction`, which is then added to the EV battery's running capacity.

Then at each step all the data is used to accumulate more information in the following fields: `cum_pv_gen`, `cum_pv_used`, `cum_EVmaxPV`, `cum_LmaxPV`, `my_pv_consumption`, `max_pv_consumption`. The final two, along with the ϵ tolerance parameter passed on wrapper initialisation, determine the value of the boolean `enough_pv_consumption`. Then the reward, only if at the end of an episode, can be calculated to be 1 if `enough_pv_consumption` $\wedge$ `enough_SOC`.

Consolidate wrappers

Up until this point [will discuss if to be included]

Action wrappers

It is at this point where I managed to properly implement the `gym.ActionWrapper` wrapper subclass so that the action space can be changed with a separate wrapper around the

main newly-implemented wrapper, instead of needing to include the `step` functions so that I can directly manipulate the `action_space.contains(action)` assert.

I had previously thought that changing the action space would change the parent environment's action space also, but this became clear not to be the case. I had also previously tried to implement this ActionWrapper in the same way as my now working solution by overriding the `step` method, and preprocessing the action here before calling `super().step(action)`. This was closer to the correct idea.

In fact it was the `action()` method which needed to be overridden. No call to another method is needed here, as the `action` method is directly used in an ActionWrapper for preprocessing of the action before being used in the body of the `step` method.

Training learning agents on new wrappers

The learning agents were connected to the wrappers, and again needed only minor adjustments for compatibility, mostly in formatting of shapes, dimensions, and lengths.

The agents were trained for various lengths, ranging from 100 to 3000 episodes, and plots were made with points at intervals of up to 50 episodes, plotting the mean and standard deviation of the total episode reward over the last interval.

At almost all episodes, the SoC would reach its maximum value of 1 very quickly, but the PV consumption would stay pretty steady at values between 0.52 and 0.68. Since the tolerance value was set to 0.3, the average reward was steady at around 0.2. Varying the tolerance value did not seem to change the lack of learning occurring from the agents.

Plots for testing wrappers

There were several methods employed to verify that the wrappers had been implemented correctly and were not exhibiting clearly undesirable behaviour:

- Most simply, at the end of each episode a report was printed which was a single line in the console output with many details, including the data index before and after reset, the episode start/stop timesteps, PV consumption, final SoC, total cost, and resultant sparse reward.
- Data was collected from the training loop and the `info` return object of a small collection of specific sample episodes for which plots were made to track PV generation, load, SoC, running cost, and real charge actions.
- Plots over 5 days were generated also, which vertical grey slices at night-time to track PV generation and household load to verify that the data remains in step with the episodes and the sun isn't appearing in the middle of the night.

The plots of sample episodes prompted me to look into different scaling factors for the data generation, both PV and load. Initially were at a peak value of 3W, which would charge 3Wh over an hour. These values should be scaled up by 500 - 1500 times to be more realistic. It was considered that to encourage PV self-consumption to be increased, then the load could be scaled up more than the PV.

Plots of sample data stretching multiple days quickly showed me that the time-of-day was not staying consistent. Even though I was moving the index for the time skipped at the end of the episodes, I had not accounted for sample data outputs that are called once at each reset, so the index simply needed to be decremented once after this reset sample output is called.

There was also a set of unit tests done specifically to check the action space changes, which are detailed below in **3.6.1 Unit tests**.

3.4 Improving and testing agents

All three agents not learning would possibly indicate that the environment is too difficult to learn from, as opposed to the agents being broken. Nonetheless, to troubleshoot why the agents do not learn, it is necessary to check that they can learn in the first place and that I have not broken the implementation during the course of importing and editing them.

3.4.1 Testing agents on easy environments

Simple games were chosen, with fewer moving parts, that the agents should not have trouble learning from. The learning agent folders were copied into a new section called `Bc-testingAgentsOn--BasicEnvs`, so that they can be kept separate and compared if needed.

Each of the examples required a small level of setup alterations, but further changes needed are described in their sections below.

DDPG on Pendulum

NormalisedEnv is a wrapper which is recommended to be around this Pendulum environment before loading the agent and training. It is an ActionWrapper which scales actions such that they are in the range 0 to 1.

Unfortunately there is not much to learn from this as the DDPG algorithm learns very quickly on the Pendulum environment with or without the wrapper. Furthermore, the

continuous action space for my newly wrapped continuous environment is already normalised from -1 to 1.

There is also a `_reverse_action` method in the ActionWrapper, which would not be called in my environment, so it would be very difficult to replicate this wrapper without knowing more about its specific purpose.

P-DQN on parametric Pendulum

There is not a large supply of Gym environments that use the hybrid parametric action space, and even fewer that define it in the same format. In the end it made most sense, instead of using a complex game by a less-trustworthy source, to use this agent on Pendulum, but preprocessing inside of the training loop such that P-DQN can pretend that Pendulum comes with a parametric action space.

P-DQN working on parametric Pendulum took the longest to get to learn out of the three of these "easy environments". It did not look like anything useful was happening, until I paid attention to the `initial_params` array that starts off the guessing for values for each of the two parameters. After three educated guesses, the algorithm suddenly was learning very quickly and maintaining high stability.

DDQN on Cartpole

The console was outputting a warning `opt.step must be called before lr.step`.

The DDQN algorithm is learning well for the first 300 episodes, after which there is very dramatic fluctuations, before it in the end forgets its high performance policy. Interestingly, online I have found many people getting the exact same type of behaviour with DDQN, and also on this environment.

There are several approaches on how to avoid this instability problem, including early stopping and saving high performance state, or alternatively a rigorous search over the hyper-parameter space. Either way, the parameters would not correspond to my learning environment, and it has been shown that the agent does learn, and also shows string similarities in performance with other DDQN implementations.

3.4.2 Takeaways

There was only a small number of concrete changes that could be made to the agent implementations connected to the building control environment. However, we now have confidence that the algorithms themselves do work.

From here, other than minor tweaking, improvements in algorithm performance is expected to come only from the extension implementation of Hindsight Experience Replay, and also of course from conducting a thorough search of the hyperparameter-spaces in order to search for optimal selections.

3.5 Baseline evaluation methods for comparison

Naive and Random control

Random control is implemented by at each hour taking a completely random decisions, both about what source to charge from, and also the power level of the charging attempt.

Naive control represents an instinctive charging habit of many users with a PV supply. The charging starts as soon as the EV returns to the house, with the supply coming from any excess PV generation after household usage and otherwise from the grid at the maximum charging power.

Rule based control

In this protocol a small set of more intelligent rules determine when to charge the EV, based on how much time there is left to charge from the grid and trying to delay this option to give greatest opportunity to charge from PV even if it is not available in this timestep.

Primarily, if there is excess PV generation available, charge with this renewable supply.

Else, the agent uses the current proportion left to charge $(1 - SoC_t)$, the full battery capacity (C_{max}), and time left to departure (D_t) to charge the EV from the grid only if the following condition is true, else there is no charge.

$$(1 - SoC_t)C_{max} >= (D_t - t_{lag})P_{max}$$

C_{max} is the battery's total capacity, and the t_{lag} parameter can be tuned to control shortsightedness, but is not needed if there is exact confidence in the time-to-departure value.

Deterministic optimisation

Deterministic optimisation provides an optimal solution, based on an objective function and a set of constraints. I have had to modify the constraints presented in the paper by Dorokhova et al. as they describe a different problem, namely one where selling energy to the grid is allowed.

The objective function aims to maximise both the total PV self-consumption and the SoC at departure:

$$\max \left(\frac{\sum_{t=t_s}^{t_{dep}} (L_t^{PV} + EV_t^{PV})}{\sum_{t=t_s}^{t_{dep}} PV_t}, (1 - SOC_{t_{dep}}) \right) \tag{3.1}$$

The following constraints are valid for all $t \in \{t_s, ..., t_e\}$:

- $PV_t + P_t^B - EV_t - L_t = 0$, where P_t^B is power bought from the grid.
- $PV_t \geq EV_t^{PV} + L_t^{PV}$
- $P_t^B = EV_t^G + L_t^G$
- $EV_t = y_t^{EV} EV_t^G + (1 - y_t^{EV}) EV_t^{PV}$, where $y_t^{EV} \in \{0, 1\}$ is a binary variable that forbids simultaneous charging from grid and PV power supply.
- $SOC_{t_{arr}} = SOC_{arr}$, where SOC_{arr} is a system constant for each episode.
- Next we include the constraints presented above in the **Action space** subheading in **2.5.1 Markov decision process**.
- Finally there remain only the trivial constraints that the SoC must remain inside $[0, 1]$; EV_t^G and EV_t^{PV} must be inside $[0, P_{max}]$; and that $SOC_{t+1} = SOC_t + \frac{EV_t^G + EV_t^{PV}}{C_{bat}}$ (timestep always 1 hour, and charging efficiency of 100% may be omitted from the maths).

Insert

implementation

details

Deterministic MPC

Deterministic MPS is similar to deterministic optimisation in that it contains mostly the same constraints. However, with MPC the optimisation problem is solved once at each timestep, instead of all in one go. At each timestep the current environment state is known, but neural networks are used to predict the future PV generation and household load values for the rest of the episode in order to optimise on the predictions to take an optimal action in the right direction.

Specifically, a Long Short-Term Memory (LSTM) network is used for predicting PV_t values, and an Auto Regressive Integrated Moving Average (ARIMA) model is used for

the L_t load values, for each timestep after the current timestep ($t = t_d$) until the end-of-episode timestep ($t = \{t_f, ..., t_e\}$).

The constraints are identical, except that they are duplicated so that they can apply to decision and prediction stages separately. The objective function needs to be altered to reflect the separation of the data being optimised on:

$$\max\left(\frac{E^{PV}_{t_d} + L^{PV}_{t_d}}{P^{V}_{t_d}} + \frac{\sum_{t=t_f}^{t_e}(E^{PV}_{t} + L^{PV}_{t})}{\sum_{t=t_f}^{t_e} P^{V}_{t}} - \left(1 - SOC_{t_{dep}}\right)\right) \tag{3.2}$$

Insert

implementation

details

Strategy for optimisations

Since the optimisations use a constraint-based solver to search the solution-space, instead of interacting directly with the environment I instead prepared the data in advance so that it can be formatted in a way to be easily accepted as optimisation constants.

In addition, start and end timesteps for the episodes are be pregenerated. For example, SOC_{arr} which is an optimisation constant will be held inside an array with one value for each episode that the optimisation will be run for. Inner workings of the step methods have been replaced by the constraints.

3.6 Summary of unit tests and utils

3.6.1 Unit tests

This section provides a summary of the unit test that I included in my work. Unit tests, or component tests, are exist to validate that individual components of source code perform as they were designed and intended. In my code they were introduced for two reasons: either to verify the semantics of a new function that was created, or

disc_to_contTest.py

This file included tests for different styles of data processing, which was useful during the process of deciding on good formats for the action spaces, and on the different places

where I can process that data. For example the Discrete could start at 0 or at -10, and for each there is a different function that I would need to insert into the `action` overriding method so that the `step` function received the action in the correct format.

Moreover, at the start of the project I was implementing the step function individually for each wrapper, so I did not need to process these actions to the standardised continuous format. Instead I could choose to have them represented in a more meaningful way. In hindsight, it would have been easier to keep every `step` function expecting continuous actions, but I had mistakenly thought that this wasn't possible based on the action space change.

Regardless, the tests in this file were useful to verify that my initial ideas would in principle work, and which ones of them required less processing.

Overriding parent methods

I met some confusion here, which was cleared up by these tests. Since function definitions with the same name as functions on a lower wrapper level override those functions, I initially thought that meant that the initial function definitions would never be used. This was proven quickly to be false, but being systematic with tests allow me to easily see that overriding function definitions would only be used if called from that wrapper layer or above, or if accessed from outside of the wrapped environment. In hindsight this is a clear demonstration of scoping, but it wasn't visible to me because I didn't have a true understanding of the wrapper structure.

Data indexing

An outline of the difficulties with consistency in data generation and correct daily phase synchronisation is details below in **Problems encountered**. The solutions are presented below, test data plots provided visual insights, but small in-place tests were essential to track and verify expected changes in data timestep values before and after environment resets and more specific adjustments. These include comparing current timestep value to a saved value from a few lines earlier, or verifying they are equal to the time of day (modulo 20).

Config object changes

These tests we not very complex, but were very valuable in terms of examining the specific effects of changing the `config` object of the environment that is inside the wrapper.

Final ActionWrapper tests

The outline of these tests was to collect the final wrappers (`NewBaseEnvWrapper` and the ActionWrappers of form `DiscreteOverBase`) and check two things.

First we verify that the action space change has been implemented properly by extracting the action space with similar code to `result1 = (env.action_space == gym.spaces.Box([-1.], [1.], (1,), dtype=np.float32)`.

Then we check that it persists and works as intended: it should accept any valid action input into `step` function, it should reject any invalid action, and calls to reset the environment should not undo this change. This involves are more complex loop.

At the start of the loop we pass in the actions which are on the bounds of the action space [1], then we pass in samples of the action space [2], and at the end we repeat the initial action-bounds test.

The normal environment reset procedure is included, resetting from the loop whenever the `terminated` flag is activated, but at the end the environment is reset manually in advance of this point to check it is no different.

Finally we input actions that are just outside of each bound of the action space to check that they are rejected. This is done with a nested `try, except` structure, as the test should only pass if all of these final tests throw an `AssertionError` from the `step` function.

General use of asserts in debugging

Some of these unit tests were implemented on their own away from working code, and so it was easy to simply return values of True or False from specific tests. This style is always preferred as all edge cases can be manually added to consideration.

Other tests had to be more localised inside of the environment and so there were two approaches here:

1. The use of `assert` expressions that would clearly flag if undesired behaviour was occurring.
2. The use of `assert False` statements so that I can have specific control of the program endpoint, and examine local behaviour by extracting and printing values, or temporarily adding in new edge cases to the local section under criticism.

[1] `(-1,1)` for continuous, `(0,20)` for discrete, and (`(0,1,0)`, `(1,0,1)`, `(0,0,0)`, `(1,0,0)`) for parametric

[2] Gym Space classes provide a `sample` function to randomly choose a value from the valid actions.

3.6.2 Utils

check_versions.py

This is a very simple file which was used a lot in the early days of the project when I was switching between Gym v21 and Gym v26, and also between different python versions as required for different packages that I was exploring during the initial experimenting and familiarising phase.

ShowRawData

There were here a collection of files which prepared and plotted data from the datamodules of the different versions of the environment, with grey columns depicting night-time. These had three main uses:

1. To check that the data does not lose sync with the time of day upon environment reset.
2. To check how the environment seed affects the data stream.
3. To examine the relationship between PV generation and household load after different scaling factors are applied.

CreateTestData

This file contains a few different methods of separating the input data into a training set and a test set.

General plotting functions

This folder was just a location where I stored different working plotting functions, so that I could carry forward code structures I had written for storage and later use. This was useful when I wanted to rewrite or extend a plotting function for a new purpose, but I did not want to lose my debugged solution.

3.7 Extensions

3.7.1 Hindsight Experience Replay (HER)

Environments with sparse binary rewards have a very low supply of positive training samples, and so training is very slow. The HER presented by Andrychowicz et al. [8] attempts to address this issue. Every episode is replayed (stored in the buffer) for a second time, but with a reward function that assumes the episode was successful. This allows the agent to learn from unsucessful episodes by turning them into successful episodes, just with a different objective. Instead of learning nothing, the agent learns positively that certain actions will lead to a different objective, which is equivalent to saying that they are bad actions for leading to the true objective.

The objective is defined as a point or region in the observation space which must be reached for a reward. The agent needs to be observing the environment moving through the objective-space, so we concatenate the objective to the end of the state observation.

During each episode's first iteration, the true objective `(pvconsum = 1, SOC = 1)` is concatenated to the state when added to the agent's replay memory, and when inputting to the agent for action selection. On the episode's second iteration, the same (state, action, reward, done) experiences are added to the memory, but this time concatenating the achieved objective, which is the values of `(pvconsum, SOC)` at the end of the timestep. The rewards in these experiences are also related to the achieved objective, but since they are still sparse the rewards are 0 for all but the last step, and from the definition of the achieved objective the reward from the last step is always 1.

[Wait for fuller results] It does not seem to be better. To test HER improvements on my agents I could implement it on basic environments to see if they get any better.

3.7.2 Further extensions

Overcharging penalty

In a very large number of the episodes, the EV is reaching its max SOC very quickly. Only a few grid charge actions are needed to fill the battery. We want the agents to learn that they can afford to delay charging the EV from the grid, in order to make the most of the PV generation for as long as they have the time.

One proposition is the idea of penalising the agent for overcharging the battery. The agent continues to make action decisions even after SOC=1 has been reached, and we do not want it learning from these actions as we know them to have no meaning. Applying a negative reward to overcharging would certainly allow the agent to learn not to continue

trying to overcharge, but it is not obvious that it would teach it to approach max SOC more slowly.

Selling to grid (V2G) / battery discharge (V2H)

With or without dynamic pricing, V2G would mean that PV consumption would be maximal, because any unused generation can be sold to the grid. There would also be no drawback to instantly charging the EV to full from the grid, as the alternative of getting the power from PV generation would reduce the amount of PV that can be sold in equivalent amount.

V2H is more interesting but still not very meaningful. Whenever there is more load than PV generation, the EV would decide to supply free energy from discharging its battery. It would hope it can recharge with excess PV generation later, but otherwise it would have to refill that amount from the grid, the same as if the grid had supplied the excess energy needed for the load. iThese two options would give the agent many more options in terms of strategy to develop,

Dynamic electricity prices

V2G and V2H extend the environment to a more complex problem. As described in **2.1.1 Chosen paper**, dynamic prices and a bi-directional charger add greater dimensionality. The agent can learn to maximise a revenue, which goes beyond simply an optimal charging policy. It would be an interesting extension, but rather than an improvement on the current investigation, it would be investigating a different concept entirely.

Getting rid of the dual-objective

Another option which could be valuable in increasing performance by simplifying the game, would be to reduce the dual objective back to the standard cost-minimisation objective that most EV / building control environments employ.

3.8 Problems encountered / difficulties faced and changes explored

3.8.1 Data seeds

Data index timesteps were off, which was caught by plotting the data and adding visualisation of cyclical daily time (see graph **??** in Appendix C). This was a simple fix, but it lead me to see that there was a problem with the seeding of Bauwerk's original implementation of the DataModules for the PV generation and household load data.

The data for PV generation and load come from the DataLoad and DataPV classes, which both inherit the DataComponent class. They have the same functionality other than that they load their data from different paths.

If I do data_start_index is None, only then will the fix_start which have no effect, which means that a call to reset(start) will now reload the data module with the new specific start point. Otherwise the reset should have always started the datastream from index 0. This approach requires always starting the data from 0, but indexing from a different point each time. Alternatively I can load the data at a different point at each reset, and start from just the correct time of day.

The main issue boiled down largely to not seeing that I needed to include `this.env = env` inside of the wrapper's `init`, if I wanted to keep the `step` function of the wrapped environment (not re-implement it) while having it refer to the new action space. Again, the better option was still to keep the wrapped environment, but keep it referring to the original continuous action space, and simply preprocess the action which came from the wrapper's new action space so that the wrapped environment received it in continuous form.

I did reason at the beginning that this was best, but I tried to implement this preprocessing by overriding the `step` function, translating it to continuous, and then calling `super().step(continuousAction)`. This did not work, and so I thought that the strategy of only implementing the more extensive step function once was not going to work in my use case.

In the end there were one or two benefits, as I was looking over the code many times and indeed coming out with differently structured solutions and increased understanding. When I discovered how to implement them as one and reduce the ActionWrappers to much smaller implementations, examining all the differences in the code to collate them together made me confident that I had considered the best ways to implement the EV charging simulation environment.

3.8.2 Full implementation but no learning

A major problem was encountered when the agents would not learn.

Parameter search

[results of parameter search]

Data scaling search, tolerance

[results of datascaling, tolerance search]

Excess charging penalty

Simpler dense reward

[Results from getting rid of dual reward, and making it dense.]

3.9 Final repository overview

Chapter 4

Evaluation

4.1 Baselines

4.2 Model parameters, metrics, statistics

4.3 Performance

4.4 HER

4.5 Boundary conditions, persistence of problems

4.6 Success criteria

1. Paper identified, with a RL approach whose scheme most aligns with the requirements of my proposed scenario.
2. Control algorithms to be used for comparison and evaluation of proposed methodology to be properly formalised.
3. Plan and completion of informative, rigorous, and runnable simulation and testing environment in a suitable platform.
4. Proposed algorithm and comparison algorithms implemented in sensible, testable modules.
5. Rigorous training and testing of all agents, allowing evaluation and comparisons of performance.

6. Proper review of lessons learned, open questions, improvements and extensions.

4.7 Discussion

Chapter 5

Conclusions

5.1 Achievements, summary of work completed

5.2 Lessons learned and personal reflections

5.3 Future work

Bibliography

[1] Cost comparison electric car vs petrol which car costs more annually.

[2] Cost of running electric car vs petrol.

[3] Craig Bester. Source code for the dissertation: "multi-pass deep q-networks for reinforcement learning with parameterised action spaces".

[4] Emma Brunskill. Stanford cs234: Reinforcement learning, 2019.

[5] DI Engine. Di engine: pdqn.

[6] Abadi M. et al. Large-scale machine learning on heterogeneous systems, 2015. software available from tensorflow.org.

[7] Agarwal R. et al. Deep reinforcement learning at the edge of the statistical precipice, 2021. Google Research Berkeley EECS, arXiv:2108.13264.

[8] Andrychowicz et al. Hindsight experience replay, 2017.

[9] Bester et al. Multi-pass q-networks for deep reinforcement learning with parameterised action spaces, 2019.

[10] Brockman G. et al. Openai gym, 2016. arXiv preprint arXiv:1606.01540.

[11] Chan S. et al. Measuring the reliability of reinforcement learning algorithms, 2019. arXiv:1912.05663.

[12] Dorokhova et al. Deep reinforcement learning control of electric vehicle charging in the presence of photovoltaic generation. *Applied Energy*, 301:117504, 2021.

[13] Dunbar et al. Machine learning approach for electric vehicle availability forecast to provide vehicle-to-home services.

[14] Farkhondeh et al. Integration of electric vehicles in smart grid using deep reinforcement learning. In *2020 11th International Conference on Information and Knowledge Technology (IKT)*, pages 40–44, 2020.

[15] Hasselt et al. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[16] Jaehyun et al. Electric vehicle charging and discharging algorithm based on reinforcement learning with data-driven approach in dynamic pricing scheme. *Energies*, 13(8), 2020.

[17] Lillicrap et al. Continuous control with deep reinforcement learning, 2019.

[18] Mocanu et al. On-line building energy optimization using deep reinforcement learning. *IEEE Transactions on Smart Grid*, 10(4):3698–3708, 2019.

[19] Paszke A. et al. Pytorch: An imperative style, high-performance deep learning library, 2019. In Advances in Neural Information Processing Systems 32, Curran Associates, Inc., pp. 8024–8035.

[20] Rabusseau et al. Optimizing home energy management and electric vehicle charging with reinforcement learning. 2018.

[21] Raffin et al. Stable baselines3, 2019. published to GitHub.

[22] Xiong et al. Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space, 2018.

[23] Arduin Findeis. Bauwerk: Meta reinforcement learning benchmark with building control environments.

[24] Sunyong Kim and Hyuk Lim. Reinforcement learning based energy management algorithm for smart energy buildings. *Energies*, 11(8), 2018.

[25] Guan-Horng Liu. Implementation of the deep deterministic policy gradient (ddpg) using pytorch.

[26] Felix Schur. Implementation of double dqn reinforcement learning for openai gym environments with pytorch.

[27] Sebastiano. How can i properly write this equation in latex?

[28] Zhiqiang Wan, Hepeng Li, Haibo He, and Danil Prokhorov. Model-free real-time ev charging scheduling based on deep reinforcement learning. *IEEE Transactions on Smart Grid*, 10(5):5246–5257, 2019.

Appendix A

Project description

A.1 Project suggestion text

Reinforcement Learning for Bi-directional EV charging

Suggested by Prof Srinivasan Keshav

link: `http://svr-sk818-web.cl.cam.ac.uk/keshav/wiki/index.php/Projectlist`

Today's EVs are mostly one-way, that is, they charge, but they cannot supply energy back to the grid. However, vehicle-to-grid charging is far more rewarding for home owners, especially those with their own solar panels. But when exactly should the EV be charged and when should it be discharged? This is a complex problem that is determined by when the EV is present at home, the next day's travel plans, grid requirements, and so on. The goal of this project is to use reinforcement learning to come up with EV charge/discharge control, similar to the approach proposed here, but extending the use cases to make it more realistic.

Appendix B

Algorithms pseudocode

B.1 Initialisation and trainging loop

These imags have come from the Dorokhova et al. paper, and can be found here: https://www.sciencedirect.com/science/article/pii/S0306261921008874#fd7.

Algorithm 2 Deep Deterministic Policy Gradient (DDPG)

Initialize: critic network $Q(s, a|\theta^Q)$ and actor network $\mu(s|\theta^\mu)$, target critic network Q' and target actor network μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$, replay buffer $\mathcal{R}$
for each episode **do**
  observe current state s_t
  initialize random process $\mathcal{N}$ for action exploration
  for each step in the environment **do**
    select action $a_t \sim \mu(s_t|\theta^\mu) + \mathcal{N}_t$
    execute action a_t
    observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    store (s_t, a_t, s_{t+1}, r_t) in replay buffer $\mathcal{R}$
    update current state $s_t \leftarrow s_{t+1}$
    sample minibatch N of experiences:
    $e_i = (s_i, a_i, s_{i+1}, r_i)$ from replay buffer $\mathcal{R}$
    Train critic:
    compute $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    compute loss $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    perform stochastic gradient descent step on L
    Train actor:
    $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s_i, \mu(s_i)|\theta^Q)\nabla_{\theta^\mu}\mu(s_i|\theta^\mu)$
    update target actor and critic networks parameters:
    $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for

Figure B.1: DDPG training loop

Algorithm 1 Double Deep Q-Networks Learning (DDQN)

Initialize: online network Q_θ and replay buffer $\mathcal{R}$,
target network $Q_{\theta'}$ with weights $\theta' \leftarrow \theta$
for each episode **do**
  observe current state s_t
  for each step in the environment **do**
    select action $a_t \sim \pi(Q_\theta(s_t))$ according to policy π
    execute action a_t
    observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    store (s_t, a_t, s_{t+1}, r_t) in replay buffer $\mathcal{R}$
    update current state $s_t \leftarrow s_{t+1}$
  end for
  for each update step **do**
    sample minibatch N of experiences:
    $e_i = (s_i, a_i, s_{i+1}, r_i)$ from replay buffer $\mathcal{R}$
    compute expected Q-values:
    $Q^*(s_i, a_i) \approx r_i + \gamma Q_{\theta'}(s_{i+1}, \arg\max_{a'} Q_\theta(s_{i+1}, a'))$
    compute loss $L = \frac{1}{N}\sum_i (Q^*(s_i, a_i) - Q_\theta(s_i, a_i))^2$
    perform stochastic gradient descent step on L
    update target network parameters:
    $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$
  end for
end for

Figure B.2: DDQN training loop

Algorithm 3 Parametrized Deep Q-Networks Learning (P-DQN)

Initialize: action value network $Q(s, x_k|\theta^Q)$ and action parameter network $\mu(s|\theta^\mu)$, target action value network Q' and target action parameter network μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$, replay buffer $\mathcal{R}$
for each episode **do**
  observe current state s_t
  initialize random process $\mathcal{N}$ for action parameter exploration
  for each step in the environment **do**
    compute action parameters $x_k \leftarrow \mu(s_t|\theta^\mu) + \mathcal{N}_k$
    compute action values $Q_k \leftarrow Q(s_t, x_k|\theta^Q)$
    select action $a_t = (k_t, x_{k_t})$ according to the ϵ-greedy policy
    execute action a_t
    observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    store (s_t, a_t, s_{t+1}, r_t) in replay buffer $\mathcal{R}$
    update current state $s_t \leftarrow s_{t+1}$
    sample minibatch N of experiences:
    $e_i = (s_i, a_i, s_{i+1}, r_i)$ from replay buffer $\mathcal{R}$
    decompose a_i into k_i and x_{k_i}
    Train Q:
    compute $y_i = r_i + \gamma \max_{k_i \in [K]} Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    compute loss $L = \frac{1}{N}\sum_i (y_i - Q(s_i, x_{k_i}|\theta^Q))^2$
    perform stochastic gradient descent step on L
    Train μ:
    compute loss $L = -\frac{1}{N}\sum_i Q(s_i, \mu(s_i|\theta^\mu)|\theta^Q)$
    perform stochastic gradient descent step on L
    update target action value and target action parameter networks parameters:
    $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for

Figure B.3: P-DQN training loop

Appendix C

Project Proposal

Reinforcement Learning for building control with a bi-directional EV charger

***

14 October 2022

Project Originator: ***

Project Supervisor: ***

Director of Studies: ***

Overseers: ***

Introduction

More are more, the charging apparatus for electric vehicles (EVs) come with support for roughly symmetrical supply of power from the EV's battery to the power grid (vehicle-to-grid, V2G), in addition to the traditional grid-to-vehicle (G2V) direction for charging the EV battery. Because it is a complex problem for grid operators to vary the energy output from its power sources to directly match the total current energy demand from the grid and its customers, variation in demand can be dealt with in other ways.

One strategy is variable pricing throughout the day, encouraging customers with lower prices to schedule their energy requirements during off-peak times. In addition, where a building has excess energy, they can sell this back to their energy supplier by discharging a battery or other power source into the grid, in exchange for fiscal rewards again at variable pricing throughout the day.

A building controller which can learn an optimal control policy would provide revenue, while also contributing to dampening the burden on the grid from highs or lows in demand.

Description of the Work

In this project I will be considering a single building with one EV (with bidirectional charging), photovoltaic supply (solar panels with an associated storage battery), and a single (for simplicity) controller which determines what flows of energy can meet the demands of building's energy usage and those of the EV utility, while pursuing the goal of maximising customer satisfaction by offloading power back to the grid.

I will be reimplementing a Reinforcement Learning algorithm with a model-free approach from existing academic literature, which will assume the role of the controller to learn an optimal control strategy to maximise overall revenue. I will consider several contemporary solutions, and conclude primarily on one that best aligns with the problem I am trying to investigate.

In order to evaluate the functionality and performance of the solution, I will construct a simulation environment which be able to run the algorithm implementations on the data.

Most of the data (building energy demand, supply from solar installation, grid energy price fluctuation) will likely come from existing sources of collected data, potentially with some noise added for generality. The simulation environment will also be able to generate some data if necessary (eg for utility requirements of the EV), but implications of this generation will need to be evaluated. This environment will likely be using a Python framework, possible options including PyTorch [19], TensorFlow [6], and OpenAI Gym [10]. Approprite separations of training and testing data will be planned to ensure correctness of training approaches.

As I will be reimplementing a state-of-the-art approach, it will likely be evaluated in the paper by comparison with more established RL approaches. I will implement these alternative approaches, supplementing where necessary to have a more representative set of approaches, in order that the evaluation metrics can be directly compared. This should give insight into any pitfalls or limitations, as well as both a qualitative and quantitative analysis, of the proposed approach. Consideration will be made as to if I should implement these alternative approaches myself, or utilise existing algorithm codebases, for example from Stable Baselines 3 [21].

Evaluation

The initial criteria for evaluation is the extent to which the implementations matched their desired functionality. Thorough testing and analysis will be done on each implementation to obtain a high level of confidence.

The performance and stability can be measured by analysing a graph of accumulated reward against time, looking at the extent to which it converges to some optimal policy, and its performance while learning the weights for this policy. The simulation can also be run N times in order to obtain an average episodic reward, and approximate 95% confidence intervals of the mean score from the N runs. This is one example of addressing the potential pitfalls of relying on only a single run-through.

Beyond this, I will research and utilise some more sophisticated Reinforcement Learning evaluation techniques. Included in metrics to be considered will be some techniques proposed in this paper by Google Research Berkeley EECS [11] and components from the rliable library proposed here [7].

Evaluation of all approaches will be done both individually, and in direct comparison with each other.

After evaluating the proposed approaches, there will finally be evaluations of my own academic and practical approach during the planning and implementation stages of the project.

Starting point

Before starting any preparation for the project, I had no experience with Reinforcement Learning. Before the start of term, I became familiar with some of the principles of RL, and several different classes of approaching RL. This was through reading several papers in the broad academic space to gain an understanding of different goals and considerations, as well as looknig at the historical evolution of the academic space with publically available lectures on RL from Stanford University [4].

I have gained some familiarity on some surrounding topics, including the how this intelligent controller would fit into a generic infrastructure, and the potential benefits for a single user and a larger grid-based ecosystem if the potential proposed product would come into a commercial market.

Success criteria

1. Paper identified, with a RL approach whose scheme most aligns with the requirements of my proposed scenario.
2. Control algorithms to be used for comparison and evaluation of proposed methodology to be properly formalised.
3. Plan and completion of informative, rigorous, and runnable simulation and testing environment in a suitable platform.
4. Proposed algorithm and comparison algorithms implemented in sensible, testable modules.
5. Rigorous training and testing of all agents, allowing evaluation and comparisons of performace.
6. Proper review of lessons learned, open questions, improvements and extensions.

Work plan

I have agreed biweekly meetings with my supervisor. In addition, to keep track of my working progress, I will create weekly reports detailing achievements and problems encountered. I am also maintaining a logbook, which will be invaluable for recording more detailed insights and planning strategy throughout the project.

1. **20th - 26th Oct (1 week): Planning 1**

- Extensive review of requirements analysis and approaches for many existing solutions to the control problem.
- Identify an academic paper which attempts to most closely address the conditions I am concerned with, and which contains approaches which I can reimplement to conduct my own version of the invesetigation.

2. **27th Oct - 9th Nov (2 weeks): Planning 2**

   - Complete the formalising of the proposed algorithm, and initial formalising of algorithms chosen for comparison.
   - Thorough planning of the structure of my code for each module, and considerations about different possible flow diagrams to visualise the topology of the whole system.
   - Demonstration of proficiency with the chosen framework and any new language with some small testing blocks.

3. **10th - 16th Nov (1 week): Implementation 1**

   - Construction started of the simulation, training and testing skeletal framework, with minimal functionality further than demonstration of runnability with some diagnostics.
   - Finalising of process flow diagrams identifying the specific breakup and interaction of all necessary modules.

4. **17th Nov - 7th Dec (3 weeks): Implementation 2**

   - Significant construction of the main implementation, including dynamic review of code choice and style.
   - Finalising implementation of simulation environment for data collection and direct interaction.
   - Unit testing of the simulated data collection environment.

5. **8th Dec - 4th Jan (4 weeks): Implementation 3 (Christmas break)**

   - Algorithms for comparison are implemented, shown to be workable in simulation environment, and shown to be true reflections of their conventional ideas.
   - Time in the Christmas Break is allocated for catch-up contingency, revision, and refreshing.

6. **5th - 25th Jan (3 weeks): Review and extensions analysis**

   - Majority of deliverable codebase has been completed and is now reviewed for mistakes, inconsistencies, bad design, and ability to yield meaningful results.
   - Exploration of further questions that can be explored, assumptions that can be minimised, or extensions that could add to the value and interest of the project.

7. **26th Jan - 1st Feb (1 week): Final testing preparation**

   - Collation of final metrics for evaluation of the systems, considering all questions I am trying to resolve.
   - Preparation for the presentation of evaluation data to be collected, including all

diagnostics and results, and graphs and necessary visual aids.
Milestone: Progress report submittion.

8. **2nd - 8th Feb (1 week): Main testing, gathering of results**

   - Training and similations are run for several cases in order to obtain close to the full set of results and output data needed to evaluate the performance of each system.

9. **9th - 15th Feb (1 week): Dissertation 1**

   - Introduction and Preparation chapters should be written, using Logbook as main source for explanation of approach and reasoning.

10. **16th Feb - 1st Mar (2 weeks): Dissertation 2**

    - Implementation chapter should be written.

11. **2nd - 8th Mar (1 week): Evaluation 1**

    - Evaluation of recorded results for each approach, identifying where they meet and break expectations based on results obtained in previous research.

12. **9th - 15th Mar (1 week): Evaluation 2**

    - Presentation of results to present compelling arguments for all conclusions that might be drawn from the data seen produced, and what has been simulted.
    - Evaluation chapter written.

13. **16th Mar - 5th Apr (3 weeks): Conclusion**

    - Conclusion chapter written.
    - Any lessons learned will be reflected upon; any questions posed in the introduction (or throughout) should be fully answered and concluded.
    - Snapshot draft chapters should be sent for review.

14. **6th - 19th Apr (2 weeks): Dissertation 3**

    - Final construction of the dissertation, and dissertation draft sent for review.
    - Discussion of potential next steps or possible extensions if project taken further than Part II.

15. **3 weeks until dissertation deadline to account for any delays, or problems along the way.**

Resource declaration

1. I will be using my personal laptop and a personal desktop computer for working on my project. All files are consistently stored on Microsoft Onedrive.

2. I will be using GitHub for version control and backups for my code filing system.