Deep Q-Learning vs. Actor Critic Model for Optimal V2G Scheduling in an Ideal Microgrid Setting

May 4, 2024

Carla Becker, Baptiste Faugere, Max Kasteel, John Schafer

Contents

1 Introduction and Motivation

Climate scientists agree – decarbonization of power generation and consumption is critical for cutting greenhouse gas emissions [1]. In recent years, the share of energy produced by renewable sources like wind and solar has increased and more infrastructure continues to be built to serve electric vehicles (EVs). The increased penetration of renewable energy sources is not without its challenges, however. The intermittent generation of wind and solar power can stress a grid that depends on both renewable and traditional, carbon-based generation. During hours of peak renewable production, traditional production is ramped down, but when renewable production falls off, traditional production must ramp quickly meet consumer demand. This fast ramp stresses the grid and has its own additional energy cost. The most promising solution to this problem is to store excess energy produced by renewable sources during peak hours for later use, eliminating the need for abrupt ramps and curtailments in traditional production. Batteries are another costly piece of infrastructure, however, which additionally do not have the cleanest footprint due to their mined, heavy-metal constituents. Thus, it is desirable to use as little battery storage as possible in the collective energy ecosystem while still minimizing stress on the grid. A popular hypothesized solution is Vehicle-to-Grid (V2G) charging/discharging, wherein the batteries of electric vehicles are used as flexible grid energy storage resources while their owners are at work or at home. In this report, we attempt to model a 3.5 MW microgrid, estimated to serve 5,000 people, with 800 grid-connected EVs, each with an 11 kW battery. We use two methods to find an optimal scheduling policy: Deep Q-learning (DQL) and the Actor Critic model.

2 Literature Review

Before deciding to use Deep Q-learning to find an optimal scheduling policy, we first conducted a review of the state of the art for modeling and optimizing the V2G problem. We found that most researchers constructed objective functions for their optimizations that maximized grid stability and/or energy efficiency. The researchers then either tried to find a policy that met consumer demand with energy stored in EVs or tried to schedule EVs to charge during non-peak hours. The common test beds used in these papers included microgrids connected to larger grids and IEEE test bus systems, often with additions like inclusion of real-world data and/or treatment of EVs in aggregates. In our review, we found the most popular methods to be robust optimization [2], Newtwork Constrained Economic Dispatch [3], genetic algorithms [4], model predicitive control [5], and Lyapunov optimization with high performance computing [3]. Many researchers, like [6], [7], and [8], seem to have converged on Deep Q-learning, and we chose to use Deep Q-Learning for this reason. We also decided to use real-world data to inform our energy generation and consumption modeling and to treat all EVs in our system as a single aggregate for computational efficiency.

In DQL, deep neural nets are used to approximate the quality values of state-action pairs instead of using the Bellman equation and then these values are used to determine the optimal policy. The Bellman equation is used in DQL to update the Q-values, as in temporal difference learning. DQL can run into issues with high-dimensional or continuous state spaces however, and accordingly we also decided to investigate using an actor-critic model. Instead of computing a value function or quality function, actor-critic models are policy-based methods where one neural net learns the policy for a given state (the actor) and a second neural net evaluates the action taken by the first neural net to adjust the policy (the critic) [9].

3 Problem Statement

Our proposed project seeks to answer in general, given a model representative of a small-medium town/microgrid, can renewable assets paired solely with a V2G-enabled electric vehicle aggregator (EVA) effectively balance demand and generation over a daily time frame? In simpler words, can V2G be the solution for storage shortages in 100% renewable microgrids?

To answer this question, we develop two models: a deep Q-learning model and an actor-critic model to propose the ideal scheduling policy for EVA charging. The "optimal schedule" for V2G utilization will be one that effectively transfers energy from times of low demand to times of high demand and minimizes a) the cost of running an auxiliary diesel generator and b) the curtailment of renewable assets. In other words, the optimal V2G scheduling will be one that effectively reduces the variability of the load curve, rendering it constant at all times. This research topic is of great importance to both car makers and grid operators, as V2G requires special hardware implementations and cooperation with policy makers and grid operators.

4 Methodology

4.1 Formulation of Markov Decision Process

The following is the complete description of the given problem as a Markov Decision Process (MDP) tuple $(\mathcal{S}, \mathcal{A}, P, R)$. As any Reinforcement Learning (RL) process working under MDP, it contains a state, set of actions, a transition model (not needed here, see 4.1.5) and a reward.

4.1.1 State Space

For any state S within the state space, the environment is fully described. For this project, the state space is defined as follows:

$$\mathcal{S} = \begin{bmatrix} \text{Demand(t)} \\ \text{Solar(t)} \\ \text{Wind(t)} \\ \text{SoC}_{\text{EV}_1}(t) \\ \vdots \\ \text{SoC}_{\text{EV}_N}(t) \end{bmatrix}$$

where:

- **Demand**(t) represents the current electricity demand of the grid at time t.
- **Solar**(t) represents the current power generation from solar sources at time t.
- **Wind**(t) represents the current power generation from wind sources at time t.
- $\mathbf{SoC}_{\mathbf{EV}_i}(t)$ represents the State of Charge (SoC) of the i-th electric vehicle, where $i = 1, 2, \ldots, N$ and SoC $\in [0, 100]$ at time t.

4.1.2 Action

The action space of our system is defined by a set of scalars $a \in [-1, -.9, -.8 \cdots 0, .1, \cdots, 1]$ that denotes the fraction of EV vehicles slated to charge or discharge at time t.

$$a = \begin{cases} \text{fraction of charging EVs} & \text{if } a > 0 \\ \text{all EVs disconnected} & \text{if } a = 0 \\ \text{fraction of discharging EVs} & \text{if } a < 0 \end{cases}$$

Consequently, there are 21 distinct actions the agent can take. This formulation of the action ensures that the aggregate V2G activities of the EVs are defined to be synced and reduces the complexity of the action space as compared to deciding the charging status of each EV individually. In other words, charging / discharging / not connected are mutually exclusive events. Furthermore, by specifying the fraction of EVs to engage in V2G, the agent is theoretically enabled to handle medium-to-large fluctuations in the overall grid status. We also picked a 0.1 constant step in order to cover a wide range of actions, meanwhile limiting their numbers, again preventing us to deal with massive Action Space.

Improvement from Progress Report: We originally set one action a_i per vehicle i, defining the action for every car individually. Every a_i would take on the value -1, 0, or $+1$, corresponding to discharging, disconnected, or charging. Even though that description was more realistic, the action space had a cardinality of 3^n, where n was the total number of EVs, making the model training impossible to complete on a laptop CPU in a reasonable amount of time.

4.1.3 Power Status of Electric Vehicles

P_{EV_i}, is a critical component of our model. It represents the power that each EV either contributes to, or consumes from, the grid, determined by a function of the EV's state of charge (SoC) and the current action a_{EV}. The action can be to charge, discharge, or disconnect the EVs in agregrate. We still individually determine the power status of each EV P_{EV_i} by:

1. **Eligibility:** Determine which EVs are eligible for the action:

$$\text{Eligible EVs} = \begin{cases} \{i \mid \text{SoC}_i < \text{Max Soc (80)}\} & \text{if } a > 0 \\ \{i \mid \text{SoC}_i > \text{Min Soc (20)}\} & \text{if } a < 0 \\ \{\} & \text{otherwise} \end{cases}$$

2. **Selection:** Compute the number of affected EVs given the action and select randomly:

$$\text{Number of EVs to implement action} = a \times \text{Total number of EVs}$$

$$\text{EVs to Sample} = \min[\text{Number of Eligible EVs}, \text{Number of EVs to implement action}]$$

$$\text{Selected EVs} = \text{Random}(\text{EVs to Sample}, \text{Number of EVs to implement action})$$

3. **Computation:** For each selected EV i, update the SoC and power:

$$\text{Power} = \frac{\text{Max Power} \times \text{Voltage}}{4.2}$$

with: 4.2 = maximum voltage per cell when fully charged

$$\begin{cases} \text{Energy added} = \text{Power} \times \text{timestep} \times \text{Charge efficiency} & \text{if } a > 0 \\ \text{Energy used} = \text{Power} \times \text{timestep} \times \text{Discharge efficiency} & \text{if } a < 0 \end{cases}$$

$$\begin{cases} \text{SoC}_{i,\text{after}} = \text{SoC}_i + \frac{\text{Energy Added}}{\text{Battery Capacity} \times 100} & \text{if } a > 0 \\ \text{SoC}_{i,\text{after}} = \text{SoC}_i - \frac{\text{Energy used}}{\text{Battery Capacity} \times 100} & \text{if } a < 0 \end{cases}$$

$$P_{\text{EV}_i} = \begin{cases} \text{Power} & \text{if } a > 0 \\ -\text{Power} & \text{if } a < 0 \end{cases}$$

4.1.4 Reward

The objective of our model is to minimize the discrepancy between the energy demand and the total energy supply, which includes contributions from solar and wind power sources, as well as the net effect of electric vehicles (EVs) connected to the grid. The reward function rewards actions that result in a supply closely matching the demand.

Mathematically, this can be expressed as:

$$R = -\left| \text{Demand} + \sum_{i=1}^{N} P_{\text{EV}_i} - (\text{Solar} + \text{Wind}) \right|$$

where:

- P_{EV_i} denotes the power contribution of the i-th EV. $P_{\text{EV}_i} > 0$ denotes the EVs charging, or adding to the total demand. If $P_{\text{EV}_i} < 0$, the EVs are discharging, or adding to the RES(they always contribute negative power in terms of reward)

This reward function essentially penalizes any deviation from a perfect balance between demand and supply. The negative sign of absolute value combined with the RL default goal of maximizing the reward would make the discrepancy converge to zero. By doing so, it guides the agent towards actions that optimize the grid's efficiency, leveraging renewable energy sources and the flexible energy capacity of EVs for V2G services.

4.1.5 Model-free Learning – No Need for a Probability Transition Matrix

A full Markov Decision Process would also be characterized by a probability matrix, with $P_{ss'} = \Pr[S_{t+1} = s' | S_t = s, A_t = a]$ defined by the conditional probability of moving from one combined demand / renewable energy storage (RES) state to another, given a specific action taken to charge, discharge or disconnect the vehicles. However, as our state space is continuous and modeled by historical data, we chose not to model the probability matrix and instead use *model-free* learning.

4.2 Deep Q-learning

The goal of this project is to effectively schedule, or control, the vehicle-to-grid activities of an aggregated group of EVs to ensure the balance of the grid's energy resources. To achieve this, we implemented a deep Q-Learning algorithm. By opting for a model-free approach, we circumvented the complexity associated with handling a massive state-action matrix. Deep Q-Learning enables our system to learn optimal policies directly from experience, without requiring explicit knowledge

of the environment's dynamics. This flexibility allows us to efficiently adapt to the evolving conditions of the grid and the diverse behaviors of the EV fleet, ultimately enhancing the grid's stability and sustainability.

4.2.1 Environment Parameters

The state of the grid is represented through the state matrix S (see 4.1.1) such as the EV fleet, represented through their SoC and PEV (see 4.1.3), the set of actions with a (see 4.1.2) and the Reward R (see 4.1.4).
We assumed 800 the number of cars in the EV fleet.
The environment was split into episodes of 24 hours, each one divided in 96 time points (every 15 min). For more convenience, the training was first looped over a constant episode, which we determined averagely over all the data set. Once our model trained, we tried it on a set of different days with specific demand and supply.

4.2.2 Agent Parameters

The training hyperparameters are the core of RL, governing the training process with learning rate, batch size, discount factor, exploration strategy, as well as:

The sequence length parameter captures temporal dependencies in the data, enhancing the agent's decision-making by considering previous states.

The memory parameter stores past interactions, enabling the agent to learn from diverse scenarios and promote adaptive behavior. Here it is 2000 (max value).

The discount parameter (gamma) influences the agent's assessment of future rewards, balancing immediate gains against long-term objectives. Its value is 0.

The exploration rate (epsilon) parameter regulates exploration-exploitation trade-off, ensuring the agent explores sufficiently while avoiding premature convergence. Its value is 0.2.

The epsilon min parameter sets the minimum exploration rate, preventing premature exploitation of learned knowledge. Its value is 0.01.

The epsilon decay parameter modulates exploration over time, allowing the agent to adapt its strategy dynamically. Its value is 0.999.

The learning rate parameter governs the rate of model updates, impacting the speed and stability of learning. Its value is 0.1.

Finally, the details of the neural network used to approximate the Q-function, including the number of layers, the number of neurons in each layer, and the activation functions. is described in the following section.

4.2.3 Agent Deep Neural Network Architecture

Throughout the course of the project, many neural network architectures were tested. However, in addition to past academic work, it became clear that a Recurrent Neural Network (RNN) architecture would be the most ideal for our DQL agent because of the inherent time-dependency of the historical data that was powering the environment. A very simple RNN structure was chosen with two hidden RNN layers, a fully connected dense layer and a final output layer. The RNN expects not only the current state but the past n states as well when making a mapping from the state space to the value of the quality functions. The sequence length, or the number of stored past states, is easily changed and many values were tested ranging from a length representing 1 hour to an entire day. The model performed best when 24 hours of past states were fed in.

After an RNN was chosen, no structural changes were made as the performance of the model was much more dependent on other parameters such as the discount rate. The RNN estimated the action-value function according to the following Q-function:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

4.2.4 Simulation with Historical Data

To inform our environment model, we used 2 years of 1-minute resolution wind, solar, and building load power generation data from the California Independent System Operator (CAISO), which we accessed through the Power Systems and Machine Learning (PSML) database [10]. We split the data into daily episodes and normalized by the daily averages, with the hope of training our model on multiple days. Ultimately, we only trained on a single day.

4.3 Actor-Critic Model

In addition to a DQL model, an Actor-Critic Model was also employed to improve general performance. The following equations describe the update rules for both the Actor and Critic for training.

- **Actor Update:**

  $$L(\theta) = -\log(\pi(a|s;\theta)) \cdot \delta, \quad \theta \leftarrow \theta + \alpha \nabla_\theta L(\theta)$$

  where $\delta = r + \gamma V(s') - V(s)$ is the advantage, $\pi(a|s;\theta)$ is the policy, and α is the learning rate.

- **Critic Update:**

  $$\text{target} = r + \gamma V(s') \cdot (1 - \text{done}), \quad L(\phi) = (V(s;\phi) - \text{target})^2$$

  $$\phi \leftarrow \phi - \beta \nabla_\phi L(\phi)$$

  where ϕ are the critic's parameters, $V(s)$ is the state value, and β is the learning rate for the critic.

5 Results and Discussion

5.1 Deep Q-learning Results

We tried numerous parameters combinations in tuning our DQL model, but the final DQL model presented in this report was trained on 2000 "episodes" (all representing the same average day). The training was broken into numerous stages and represents $\sim$ 8-10 hours of continuous computation. Figures 1, 2a, and 2b demonstrate the behavior of this final model.

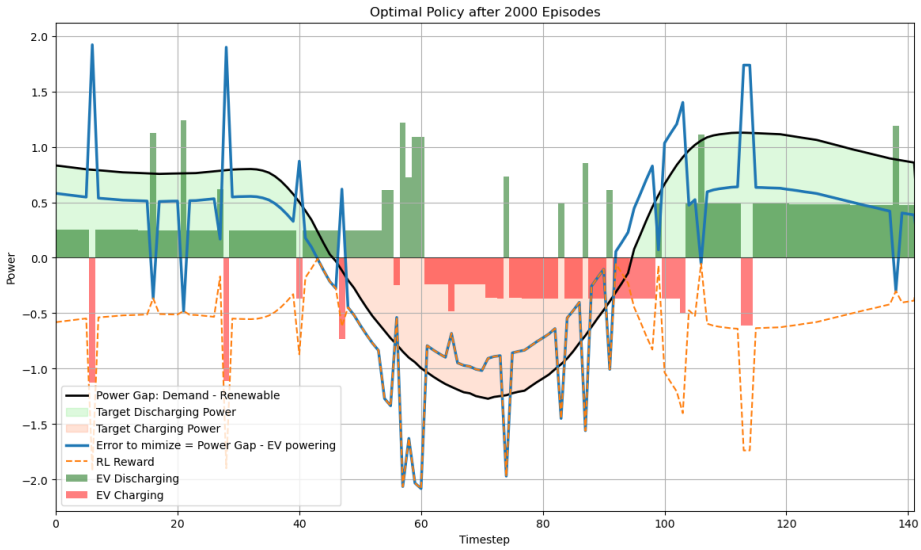
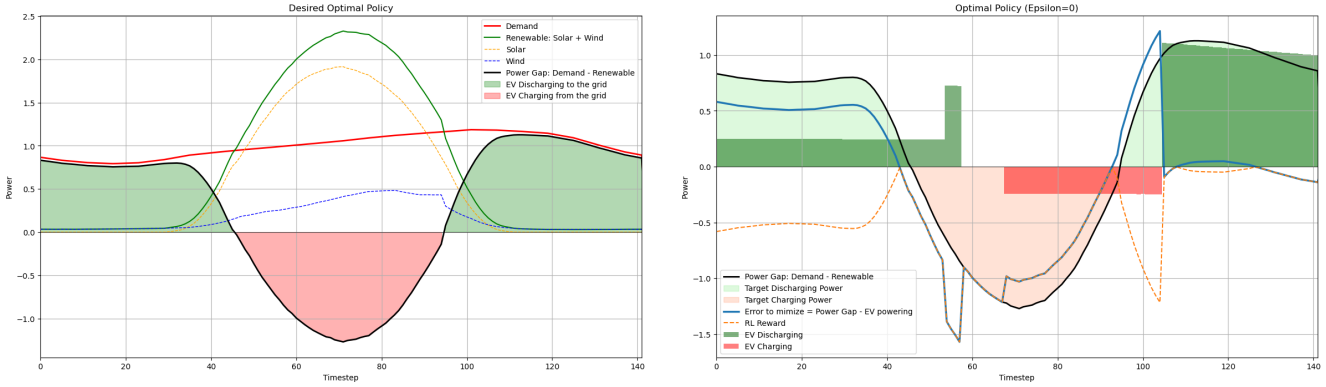


Figure 1: DQL Model on 2000$^{\text{th}}$ Episode



(a) Desired Optimal Control Policy

(b) DQL Model with $\epsilon = 0$

Figure 2: Performance of the DQL Model

As denoted by the legend, the red bars represent the EV Aggregator (EVA) charging, or adding demand to the grid, while the green denote the EVA discharging and acting like a RES. Initially, the policy the DQL has found after much training is a very poor approximation of what we know would be the "desired" policy in 2a. However there are emergent behaviors in the semi-*correct* direction that are especially visible when ϵ is set to zero in 2b. For one, the agent seems to recognize the microgrid needs additional resources supporting it at the beginning and end of the day. While in the middle of the day the excess RES generation needs to be soaked up. Besides the generally correct pattern of discharge $\rightarrow$ charge $\rightarrow$ discharge, there appears to be no fundamental recognition the magnitude of these different activities needs to change at every new timestep. 2b shows there are only 5 distinct actions taking place (out of 21 potential actions) throughout the entire episode. This is most certainly due to the greedy algorithm within the DQL over exploiting the same actions that seemed to work at one time step, repeatedly.

5.2 Inherent Problems with Deep Q-learning

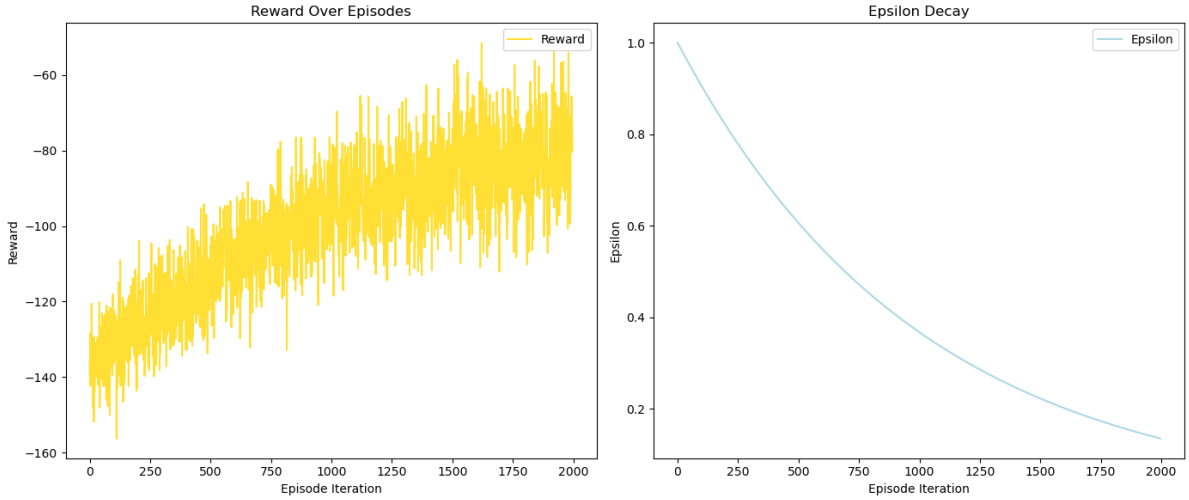


Figure 3: Reward and Epsilon Decay over the 2000 Episodes of Training

Of course it was not a surprise that the final DQL model was a poor controller. Throughout testing the model was continually becoming stuck in "local minima" of control policies that were brazenly sub-optimal. In fact the most common local minima was no action at all throughout the entire episode! The inability of the DQL to perform adequately was primarily due to:

1. Continuous nature of the state space.
2. A highly variable reward function

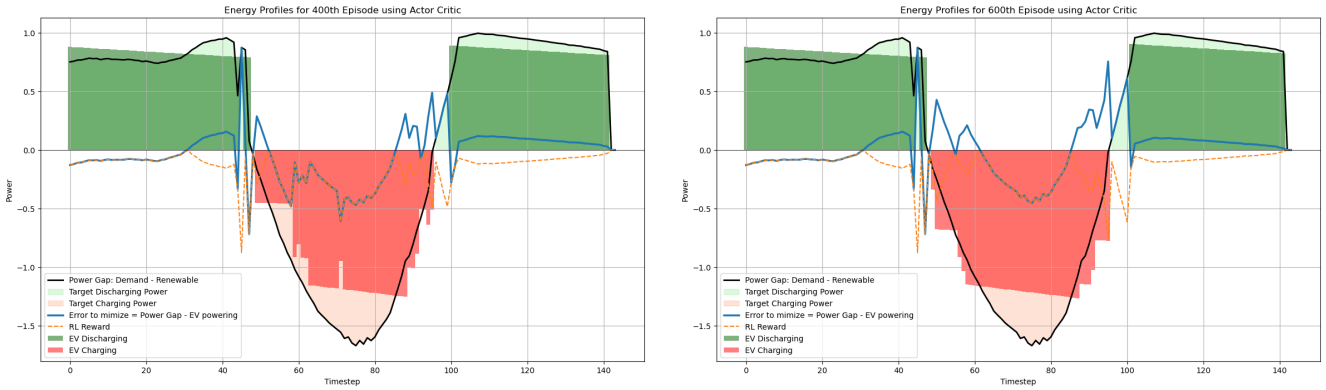
From the beginning of the project, the continuous nature of the state space was a concern. However we hypothesized that retraining on the same day over and over again as well as feeding in 24 hours worth of states into the RNN for every action would provide the model enough information to adequately learn the environment. It obvious this thinking was flawed.

Furthermore as seen in 3, the reward was highly variable from episode to episode. This variability is incredibly hard for the ϵ-greedy DQL to adapt to as it is constantly overly optimistic or overly pessimistic of the true quality of the actions it is taking for every state. Promising results illustrated in 1 were only achieved due to the implementation of a very slow epsilon decay rate of 0.999. Thus after even 2000 episodes, the DQL was still exploring over 10% of the time. The only way to combat the DQL becoming trapped in sub-optimal policies was to keep the agent exploring as much as possible. However, after 10 hours of training it was clear it would take much more exploration for our model to even attempt to get close to the"desired" optimal policy.

5.2.1 Actor Critic Model

Dismayed by the performance of the DQL approach to our problem after much tweaking and fine tuning, we decided to implement an Actor Critic Model. In an Actor Critic Model, two "dueling" neural networks (actor and critic) work together to find an optimal policy. Inherently this formulation is much better at coping with continuous state spaces which we believed to be the biggest impediment to the original DQL built.

5.2.2 Actor Critic Model Results

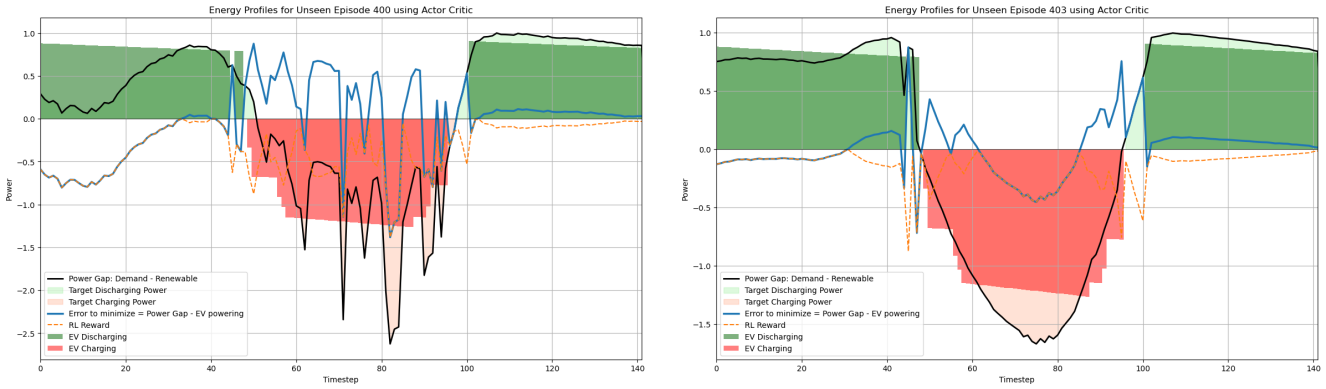


(a) AC Model 400 Episodes of Training

(b) AC Model 600 Episodes of Training

Figure 4: Progression of the AC Model over episodes

Above is the control policy after 400 and 600 episodes of training respectively representing 50-100 min of compute time. The Actor-Critic (AC) formulation vastly outperforms the DQL with little tweaking of parameters, NN architecture etc. In fact, the AC model outperformed the final DQL model after only about 20 episodes of training, or 3 min of compute time. Finally we can see in 4a and 4b the more nuanced control behavior of ramping up and ramping back down charging to soak up the excess RES generation.



(a) AC Model on Unseen Day 1

(b) AC Model on Unseen Day 2

Figure 5: AC Model Performance on Unseen Days

And finally 5a and 5b showcase the performance of the AC model on two completely unseen days or episodes. Unsurprisingly, the model does not do well with the much more variable RES generation, yet there seems to be glimmers of adaptability within the model. At this point we have arrived to where we originally thought we would pursue the project to, which is train and run a RL model on a sequence of many different days worth of day. With more time we think this AC model would do well training an many different days and developing a control policy robust to the variability of RES.

We are confident with more dedicated time to tweak this AC model as well as increase the training time, this model would begin to fully answer the problem of optimal V2G scheduling.

6 Conclusion

In this study, we experienced how unbelievably complicated managing the renewable grid will be. We made a lot of assumptions and simplifications from the energy supply to the human charging behavior. From there we explored the applicability of Deep Q-Learning (DQL) and Actor Critic models for optimizing Vehicle-to-Grid (V2G) scheduling within a microgrid environment. Our investigations revealed that while DQL struggled with the continuous nature of the state space and exhibited sensitivity to reward variability, leading to suboptimal performance and frequent convergence to local minima, the Actor Critic model demonstrated superior capability in managing the complexities of the task. The Actor Critic model adapted more effectively to the continuous state space and showed promising results even with minimal parameter tuning and shorter training periods.

Despite some initial promising outcomes, the Actor Critic model's performance on unseen days highlighted challenges with variability in renewable energy source (RES) generation, suggesting that further model refinement and extended training over diverse conditions are necessary. Future work should focus on enhancing the robustness of the Actor Critic model, potentially incorporating ensemble techniques or advanced exploration strategies to handle a wider range of operating scenarios more effectively.

Ultimately, this research underscores the potential of reinforcement learning approaches to significantly impact future energy systems, particularly in optimizing the integration of electric vehicles as dynamic storage solutions within renewable-powered microgrids.

References

[1] M. Z. Jacobson, *No Miracles Needed: How Today's Technology Can Save Our Climate and Clean Our Air*. Cambridge University Press, 2023.

[2] R. Shi, S. Li, P. Zhang, and K. Y. Lee, "Integration of renewable energy sources and electric vehicles in v2g network with adjustable robust optimization," *Renewable Energy*, vol. 153, pp. 1067–1080, 2020.

[3] F. Safdarian, L. Lamonte, A. Kargarian, and M. Farasat, "Distributed optimization-based hourly coordination for v2g and g2v," in *2019 IEEE Texas Power and Energy Conference (TPEC)*, College Station, TX, USA, 2019, pp. 1–6.

[4] K. M. Tan, V. K. Ramachandaramurthy, J. Y. Yong, S. Padmanaban, L. Mihet-Popa, and F. Blaabjerg, "Minimization of load variance in power grids—investigation on optimal vehicle-to-grid scheduling," *Energies*, vol. 10, p. 1880, 2017.

[5] Y. Shi, H. D. Tuan, A. V. Savkin, T. Q. Duong, and H. V. Poor, "Model predictive control for smart grids with multiple electric-vehicle charging stations," *IEEE Transactions on Smart Grid*, vol. 10, no. 2, pp. 2127–2136, 2019.

[6] J. Dong, A. Yassine, A. Armitage, and M. S. Hossain, "Multi-agent reinforcement learning for intelligent v2g integration in future transportation systems," *IEEE Transactions on Intelligent Transportation Systems*, 2023.

[7] X. Hao, Y. Chen, H. Wang, H. Wang, Y. Meng, and Q. Gu, "A v2g-oriented reinforcement learning framework and empirical study for heterogeneous electric vehicle charging management," *Sustainable Cities and Society*, vol. 89, p. 104345, 2023.

[8] F. Alfaverh, M. Denaï, and Y. Sun, "Optimal vehicle-to-grid control for supplementary frequency regulation using deep reinforcement learning," *Electric Power Systems Research*, vol. 214, p. 108949, 2023.

[9] Y. Jia and X. Y. Zhou, "Policy gradient and actor-critic learning in continuous time and space: Theory and algorithms," *Journal of Machine Learning Research*, vol. 23, no. 275, pp. 1–50, 2022. [Online]. Available: http://jmlr.org/papers/v23/21-1387.html

[10] X. Zheng, N. Xu, L. Trinh, D. Wu, T. Huang, S. Sivaranjani, Y. Liu, and L. Xie, "A multi-scale time-series dataset with benchmark for machine learning in decarbonized energy grids," *Scientific Data*, vol. 9, no. 1, p. 359, 2022. [Online]. Available: https://doi.org/10.1038/s41597-022-01455-7

Appendix 1 (Environment)

```
#Environment

import numpy as np
from collections import deque
import random

class GridEnvironment:
    def __init__(self, N, demand_data, solar_data, wind_data, day_index, timestep_length):
        self.current_timestep = 0
        self.total_timesteps = int(24 / timestep_length) #Assuming episode is 24 hours
        self.timestep_length = timestep_length  # Length of each timestep in hours

        # Need to think about what start / stop time we do. 12am-12am? 4am-4am etc <-- Carla
        self.N = N  # Number of EVs
        self.state_size = 3 # + N + 1 +N # State Size, includes time, and SoC
        self.action_size =  21 #3 **N # State Size

        self.demand_data = demand_data
        self.solar_data = solar_data
        self.wind_data = wind_data
        self.day_index = day_index

        # Initialize with day 0 data (96 points = 24 hours of 15 min data)
        self.demand = demand_data[day_index,0]
        self.solar = solar_data[day_index,0]
        self.wind = wind_data[day_index,0]


        self.P_EV = [0] * N  # Power status of each EV (non are connected to grid)
        # TODO Answer: If each episode is finite, how does the SoC status roll over to next e
        self.SoC = [50] * N  # SoC status of each EV (non are connected to grid), used by env


    def reset(self, day):
        self.current_timestep = 0
        self.demand = 0
        self.solar = 0
        self.wind = 0
        self.P_EV = [0] * self.N
        #CHANGE WHEN GOING THROUGH MORE THAN ONE DAY OF DATA
        return self.get_state()

    def decode_action(self, action_index):
        """
        Decode a single integer action into actions for each EV.

        Args:
        - action: The single integer action to decode that comes from the NN model.
        - N: The number of EVs.

        Returns:
        - A list of actions for each EV.
```

```
    """
    #action from NN is index,
    #set action_list=[-1,0,1]
    #return action_list[nnoutput_index]
    #actions_list=[-1, 0, 1]
    #action=action_list[action]
    #actions = []
    #for _ in range(self.N):
     #  actions.append(action % 3 - 1)  # Decoding to get -1, 0, 1
      #  action //= 3
    #return actions[::-1]  # Reverse to match the original order
    action = (action_index - 10) / 10.0  # Converts 0 to 200 into -1.0 to 1.0
    return action



def battery_voltage(self, soc):
    min_voltage = 3.0  # Minimum voltage at 0% SoC
    max_voltage = 4.2  # Maximum voltage at 100% SoC
    soc_array = np.array(soc)
    return 4.2 * (soc_array / 100)

def get_PEV(self, actions):
    #MAX's CODE
    #ACTION IS A VECTOR OF 0s 1s, -1s
    #return power output of each EV (P_EV) & the SOC for the next state

    #Based on rouh calculations, need roughly 405 EVs
    #10 groups of 41 EVs?
    #Does just multiplying work?
    timestep = (1/60)  # 1 minute
    max_power = (11)/3500  # Maximum power in kW, attempting to scale 100 EVs?
    battery_capacity = (50)/3500 # Battery capacity, scaled by entire system, each repres
    charge_efficiency = 0.90 #changed to .95 from .9
    discharge_efficiency = 0.90
    min_soc = 20
    max_soc = 80
    self.P_EV = [0] * self.N  # Reset power for each EV
    """
    voltage = self.battery_voltage(soc)  # Calculate voltage for each SoC
    power = max_power * voltage / 4.2  # Calculate power for each SoC based on its voltag

    # Ensure new_soc is of a floating point type to accommodate fractional changes
    current_soc=np.array(soc, dtype=float)
    new_soc = np.copy(current_soc) # Cast to float to prevent UFuncTypeError

    powerEV = np.zeros_like(soc, dtype=float)  # Initialize powerEV array with zeros


    # Charging
    charge_indices = (actions == 1) & (current_soc < max_soc)
    added_energy = np.minimum(power[charge_indices] * timestep, (max_soc - current_soc[ch
    powerEV[charge_indices] = -(added_energy / timestep)  # Negative with respect to grid
    new_soc[charge_indices] += added_energy / battery_capacity * 100
    new_soc[charge_indices] = np.minimum(new_soc[charge_indices], max_soc)
```

```
        # Discharging
        discharge_indices = (actions == -1) & (current_soc > min_soc)
        used_energy = np.minimum(power[discharge_indices] * timestep, (current_soc[discharge_
        powerEV[discharge_indices] = used_energy / timestep  # Positive with respect to grid
        new_soc[discharge_indices] -= used_energy / battery_capacity * 100
        new_soc[discharge_indices] = np.maximum(new_soc[discharge_indices], min_soc)

        # Idle
        idle_indices = (actions == 0)
        powerEV[idle_indices] = 0  # No power exchange for idle

        return new_soc, powerEV
        """
        if actions > 0:  # Charging
            eligible_evs = [i for i in range(self.N) if self.SoC[i] < max_soc]
        elif actions < 0:  # Discharging
            eligible_evs = [i for i in range(self.N) if self.SoC[i] > min_soc]
        else:
            eligible_evs = []
        #print('action test', actions)

        # Determine the number of EVs to affect based on the action percentage
        num_evs_affected = int(abs(actions) * self.N)
        # Ensure we do not sample more EVs than are eligible
        num_evs_affected = min(num_evs_affected, len(eligible_evs))
        selected_evs = random.sample(eligible_evs, num_evs_affected)

        voltages= self.battery_voltage(self.SoC)
        SoC_after_action=self.SoC
        PEV_after_action=self.P_EV
        for i in selected_evs:
            if actions > 0:  # Charging
                power = max_power * voltages[i] / 4.2
                energy_added = power * timestep * charge_efficiency
                SoC_after_action[i] = SoC_after_action[i] + energy_added / battery_capacity *
                PEV_after_action[i] = power # Charging ADDs to Demand (should be negative her
            elif actions < 0:  # Discharging
                power = max_power * voltages[i] /4.2
                energy_used = power * timestep * discharge_efficiency #energy used will be ne
                SoC_after_action[i] = SoC_after_action[i] - energy_used / battery_capacity *
                PEV_after_action[i] = -1*power  ## Discharging subtracts from Demand (should
        PEV_after_action=np.array(PEV_after_action)
        return SoC_after_action, PEV_after_action

    def step(self, action):
        #Apply action-> calculate reward -> update state of environment

        #Apply action, update P_EV states
        actions=self.decode_action(action)
        actions = np.array(actions)
        #print('action decoded', actions)

        next_SoC, next_P_EV = self.get_PEV(actions) #Returns updated SoC & Power levels of ea
        self.SoC = next_SoC.copy()
        #why is this not highlighted green
        self.P_EV = next_P_EV.copy()
```

```
        #Calculate Reward based upon action within same timestep
        reward = self.calculate_reward(next_P_EV, actions)

        #Move env forward one timestep
        self.current_timestep += 1
        done = self.current_timestep >= self.total_timesteps-1

        if not done:
            next_demand, next_solar, next_wind, next_SoC = self.get_state()
        else:
        # Handle the terminal state here, could be resetting or providing terminal values
        # Make sure to handle the case where you don't have a next state to provide
            next_demand, next_solar, next_wind, next_P_EV, next_SoC= 0, 0, 0, [0] * self.N, [


        return reward, done, next_demand, next_solar, next_wind , next_P_EV, next_SoC

#NEED TO FOCUS ON THE SEQUENCE OF, OBSERVE STATE, CALCULATE ACTION, CALCULATE REWARD etc

    def calculate_reward(self, next_P_EV, action):
        current_demand, current_solar, current_wind, current_SoC = self.get_state()

        # Calculate Reward
        reward= -np.abs(current_demand + np.sum(next_P_EV)- (current_solar + current_wind))


        return reward

    def get_state(self):
        # Access the current timestep's data correctly
        current_demand = self.demand_data[self.day_index, self.current_timestep]
        current_solar = self.solar_data[self.day_index, self.current_timestep]
        current_wind = self.wind_data[self.day_index, self.current_timestep]
        current_SoC=self.SoC


        # Depending on your needs, return these values directly or along with other state inf
        return current_demand, current_solar, current_wind, current_SoC


    def render(self):
        # Is this where we get our animations?!
        pass
```

Appendix 2 (DQL Agent)

```
from collections import deque
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import load_model
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Reshape
import random
import numpy as np


class DQNAgent:
    def __init__(self, state_size, action_size, sequence_length, model=None):
        self.state_size = state_size
        self.action_size = action_size
        self.sequence_length= sequence_length
        self.memory = deque(maxlen=2000)  # Experience replay buffer
        self.gamma = 0.0 # discount rate tweak
        self.epsilon = .2  # exploration rate Tweak, no decay in this version
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.999 # tweak
        self.learning_rate = 0.1

        if model is None:
            self.model = self._build_model()
        else:
            self.model = model
    """
    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        #LSTM for temporal dependicies of data
        model = tf.keras.Sequential()
        model.add(layers.LSTM(50, input_shape=(self.sequence_length, self.state_size), return
        model.add(layers.LSTM(50, return_sequences=False))
        model.add(layers.Dense(24, activation='relu'))
        model.add(layers.Dense(self.action_size, activation='linear'))
        model.compile(loss='mse', optimizer=tf.keras.optimizers.legacy.Adam(learning_rate=sel

        return model
        """
    def _build_model(self):
        model = tf.keras.Sequential()
        # First RNN layer with return_sequences=True to pass sequences to the next RNN layer
        model.add(layers.SimpleRNN(50, input_shape=(self.sequence_length, self.state_size), r
        # Second RNN layer with return_sequences=False to flatten the output
        model.add(layers.SimpleRNN(50, return_sequences=False))
        model.add(layers.Dense(24, activation='relu'))
        # Output layer for action predictions
        model.add(layers.Dense(self.action_size, activation='linear'))
        model.compile(loss='mse', optimizer=tf.keras.optimizers.legacy.Adam(learning_rate=sel

        return model



    def act(self, state_history):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        lstm_input = np.expand_dims(state_history, axis=0)
        act_values = self.model.predict(lstm_input, verbose=0)
        return np.argmax(act_values[0])  # returns action
```

```
    def learn(self, state_history, action, reward, future_state, done):
        next_state_history = np.append(state_history[1:], [future_state], axis=0)  # make seq

        lstm_input = np.expand_dims(state_history, axis=0)
        next_lstm_input = np.expand_dims(next_state_history, axis=0)

        target = reward
        if not done:
            target = (reward + self.gamma * np.amax(self.model.predict(next_lstm_input, verbo

        target_f = self.model.predict(lstm_input, verbose=0)
        target_f[0][action] = target
        self.model.fit(lstm_input, target_f, epochs=1, verbose=0)
```

Appendix 2 (Actor Critic)

```
    import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, SimpleRNN, Dense, Layer
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import legacy as lg
from tensorflow.keras.optimizers import Adam
import random

class AGENT_CRITIC:
    def __init__(self, state_size, action_size, sequence_length, actormodel=None, criticmodel
        self.state_size = state_size
        self.action_size = action_size
        self.sequence_length = sequence_length
        self.gamma = .99  # Discount rate
        self.learning_rate = 0.001
        self.epsilon = 0  # Exploration factor
        self.epsilon_decay=.90
        self.optimizer = Adam(learning_rate=self.learning_rate)

        #WORKS WITH GAMMA=.99, lr=.001, epsilon=.1

        if criticmodel is None:
            self.critic = self.build_critic()
        else:
            self.critic = criticmodel

        if actormodel is None:
            self.actor = self.build_actor()
        else:
            self.actor = actormodel
    def build_actor(self):
        inputs = Input(shape=(self.sequence_length, self.state_size))
        x = SimpleRNN(50, return_sequences=True)(inputs)
        x = SimpleRNN(50)(x)
        x = Dense(24, activation='relu')(x)
        outputs = Dense(self.action_size, activation='softmax')(x)
```

```
    model = Model(inputs=inputs, outputs=outputs)
    return model

def build_critic(self):
    inputs = Input(shape=(self.sequence_length, self.state_size))
    x = SimpleRNN(50, return_sequences=True)(inputs)
    x = SimpleRNN(50)(x)
    x = Dense(24, activation='relu')(x)
    outputs = Dense(1)(x)
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='mse')
    return model

def custom_loss(self, delta):
    def loss(y_true, y_pred):
        epsilon = 1e-8
        log_probs = y_true * tf.math.log(tf.clip_by_value(y_pred, epsilon, 1.0))
        return -tf.reduce_mean(log_probs * delta)
    return loss


def act(self, state):
    # Check if we should explore randomly
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)

    # Reshape the state to match the expected input format of the model
    state = state[np.newaxis, :]  # Adds an extra dimension to the state, making it (1, s

    # Predict action probabilities using the actor model
    probabilities = self.actor.predict(state, verbose=0)[0]

    # Choose an action based on the probability distribution
    return np.random.choice(self.action_size, p=probabilities)

def train_step(self, states, actions, rewards, next_states, dones):
    with tf.GradientTape() as tape:
        y_pred = self.actor(states, training=True)
        delta = self.calculate_delta(states, rewards, next_states, dones)
        action_masks = tf.one_hot(actions, self.action_size)
        loss = self.custom_loss(delta)(action_masks, y_pred)

    gradients = tape.gradient(loss, self.actor.trainable_variables)
    self.optimizer.apply_gradients(zip(gradients, self.actor.trainable_variables))
    return loss



def learn(self, state_history, action, reward, next_state, done):
    if len(state_history) < self.sequence_length:
        return

    # Prepare the inputs for the actor and critic
    lstm_input = np.expand_dims(state_history, axis=0)
    next_state_history = np.append(state_history[1:], [next_state], axis=0)
    next_lstm_input = np.expand_dims(next_state_history, axis=0)
```

```
# Compute values for current and next states using the critic
with tf.GradientTape() as tape:
    critic_value = self.critic(lstm_input, training=True)
    critic_value_next = self.critic(next_lstm_input, training=True)

    # Calculate target and delta
    target = reward + (0 if done else self.gamma * tf.squeeze(critic_value_next))
    delta = target - tf.squeeze(critic_value)

    # Run the actor model and calculate the custom loss
    y_pred = self.actor(lstm_input, training=True)
    action_masks = tf.one_hot([action], self.action_size)
    loss = self.custom_loss(delta)(action_masks, y_pred)

# Compute gradients and apply them for both actor and critic
gradients = tape.gradient(loss, self.actor.trainable_variables + self.critic.trainabl
self.optimizer.apply_gradients(zip(gradients, self.actor.trainable_variables + self.c

# Optionally, update the critic separately if needed
self.critic.fit(lstm_input, np.array([[target]]), verbose=0, batch_size=1)
```