Data-driven supervisory control of microgrids

Mid-term Presentation of Semester Project

Lab : LA – Automatic Control Laboratory

Prof. Giancarlo Ferrari Trecate

Assistant: Mustafa Sahin Turan

Student : Maxime Dimitri GAUTIER

18.11.2019

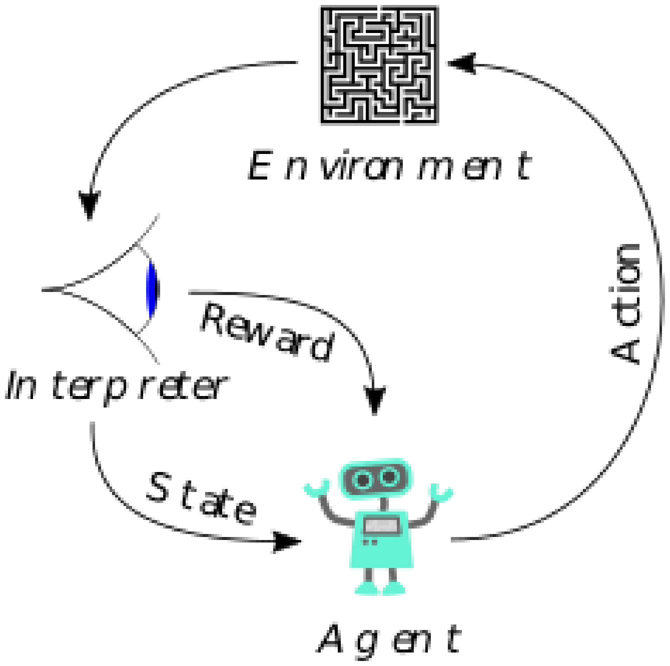
Global Objectives of the Project

- Microgrids require high level centralized control.
- No easy way to design such a controller.
- Apply Reinforcement Learning to build a robust controller for supervisory control of microgrids.

Reinforcement Learning

Principle : An agent takes actions in an environment to maximizes its reward.

Output : Policy that gives best behavior depending on situation.



Why use it?

- Online learning -> doesn’t require huge quantities labeled data like supervised learning
- Data-driven method – perfect mathematical model not necessary
- Find possible new applications for RL

Outline of Project

- Read relevant litterature and familiarise with subject
- Derive a microgrids model and implement simulator
- Fix model to be both realistic and adapted to RL
- Derive and test different RL algorithms with model

Simulator and local control

Local control of Distributed Generation Units (DGUs) based on Davide

Riccardi’s and Giuseppe Tagliaferri’s master theses on Plug-and-Play control.

Control modes :

- MPPT – Maximizes power production of Solar Panel
- Charge – Charges battery with constant power flow
- PnP – Keeps whole system stable

Simulator Model

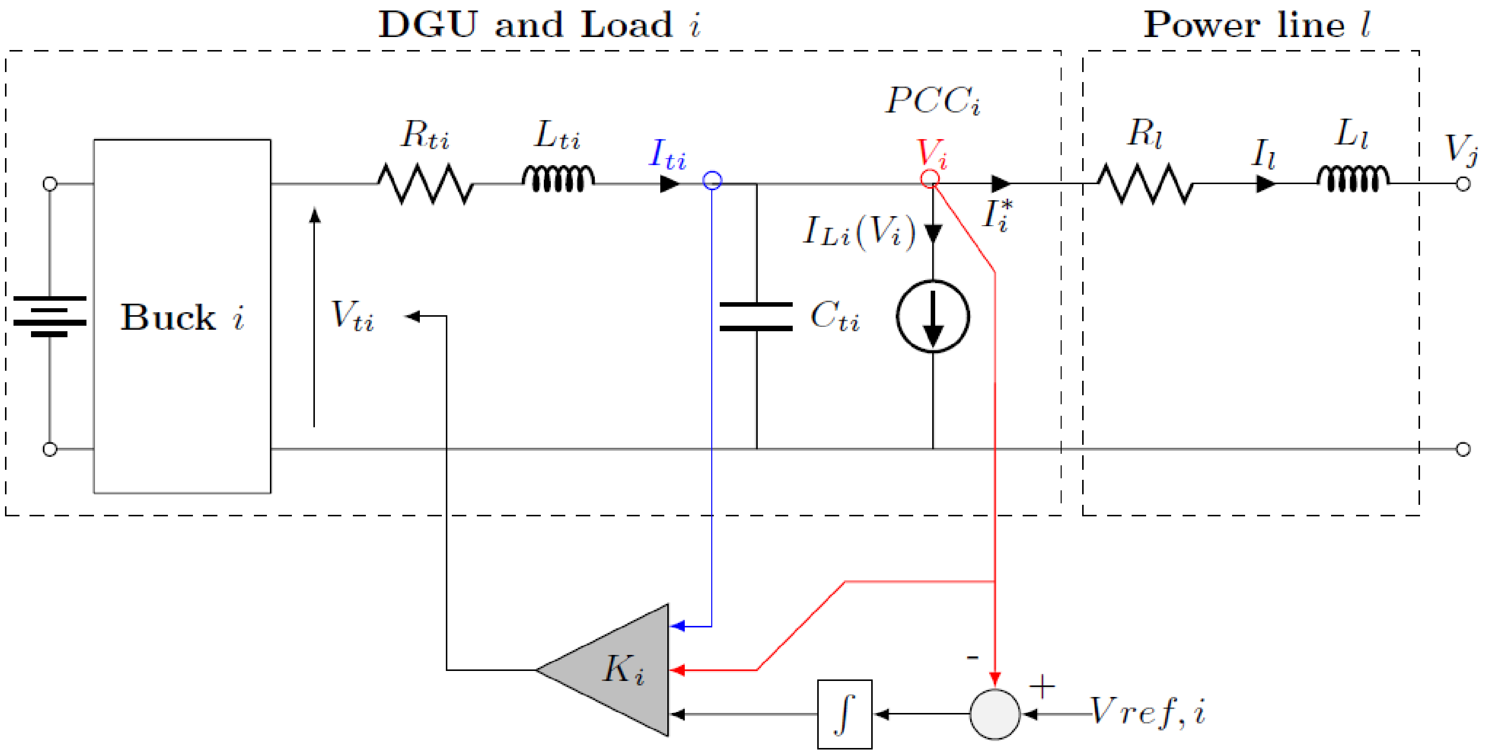
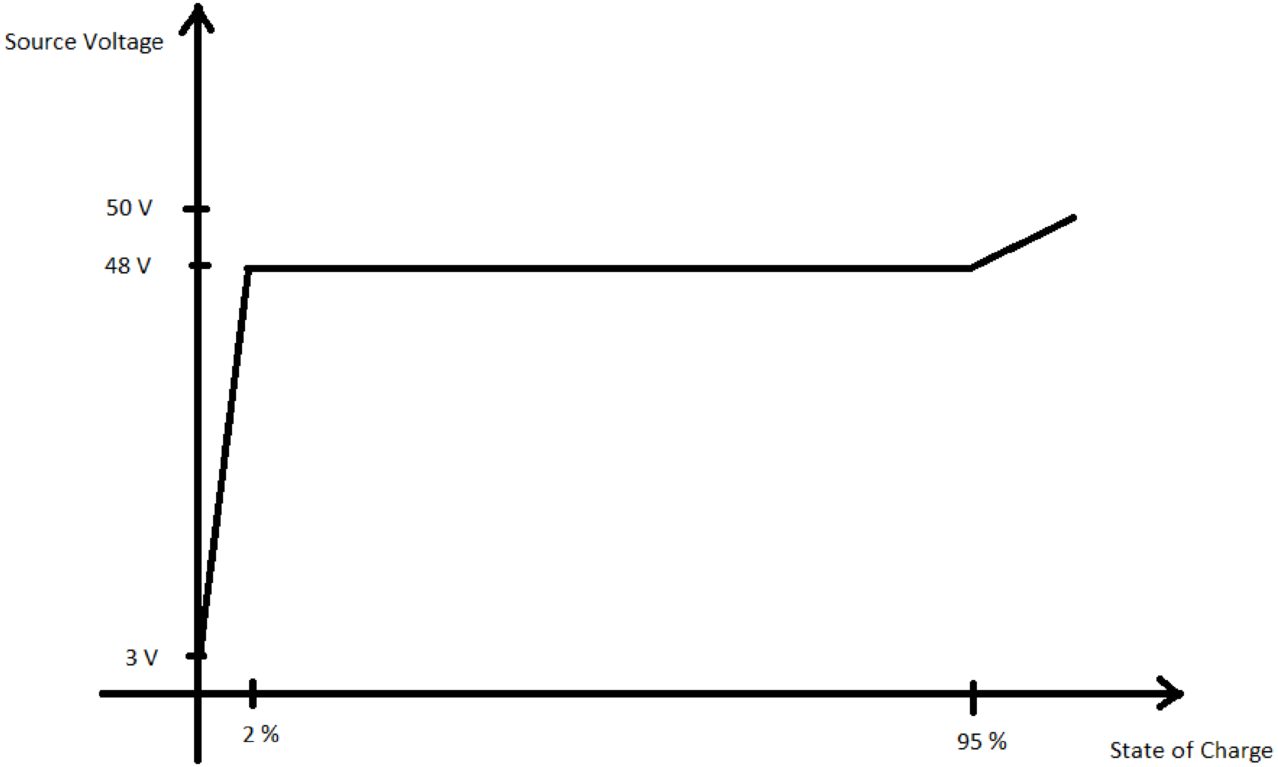


Fig. 5. Electric Scheme of i^{th} DGU along with load, connecting line(s), and local PnP voltage controller.

Model assumptions

- Buck converter and supply modelised as Voltage Source
- No sun variation -> constant voltage from source in PhotoVoltaic modules
- Constant loads for all DGUs
- Current Limiter Mode not implemented
- DGUs can’t disconnect from network

Model Upgrades



- Added Bumpless transfer

-> necessary for RL

- Added State of Charge/Source Voltage dependency

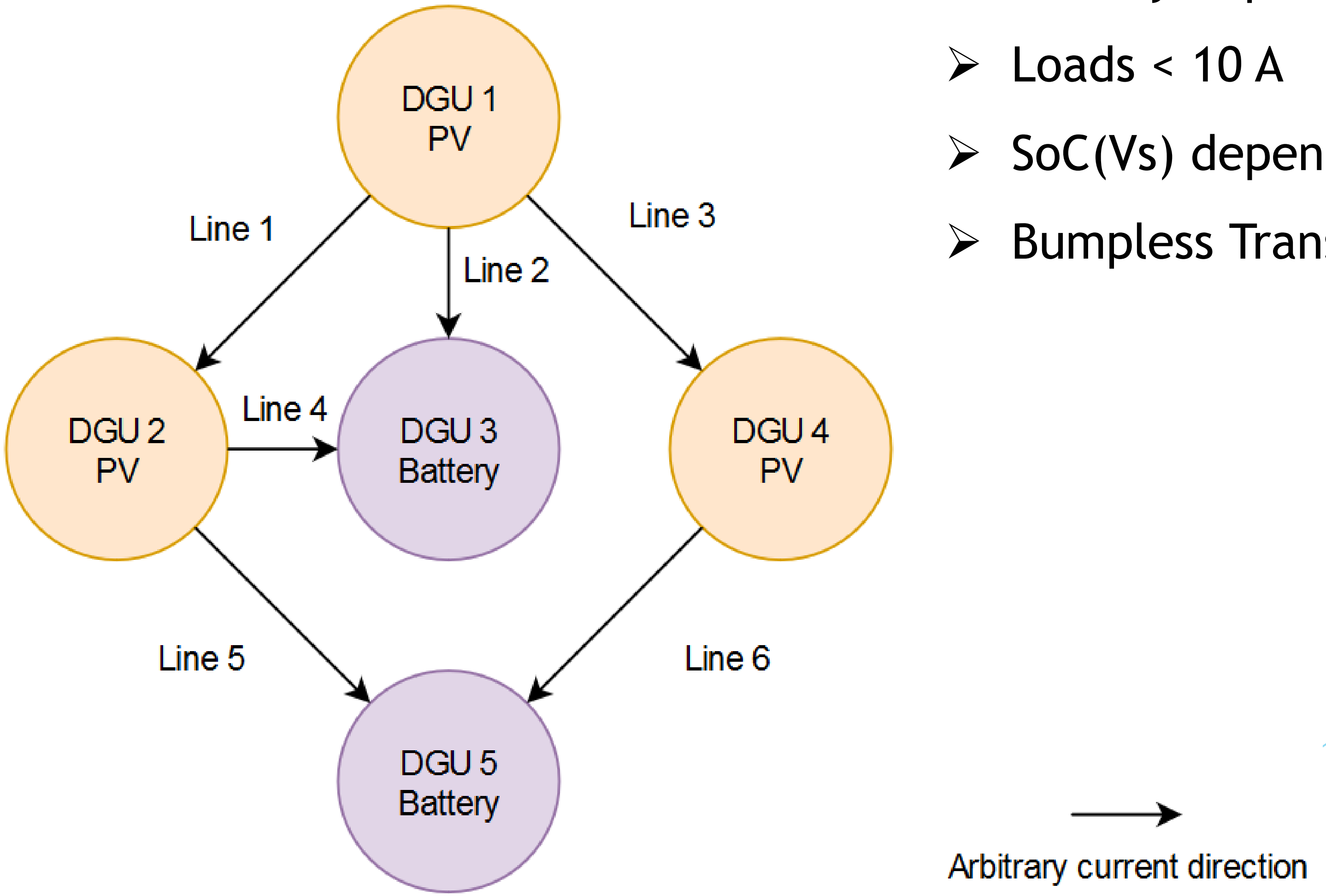
-> More realistic behavior around min and max SoC

-> Mimics Sheperd's model

- Changed battery capacity to be more realistic

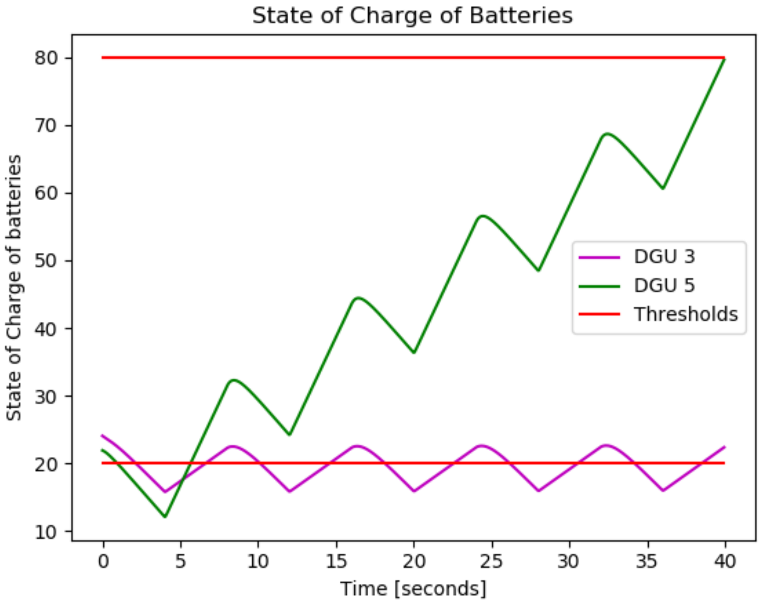
- Converted model to openAI gym environment -> re-usable with several python librairies

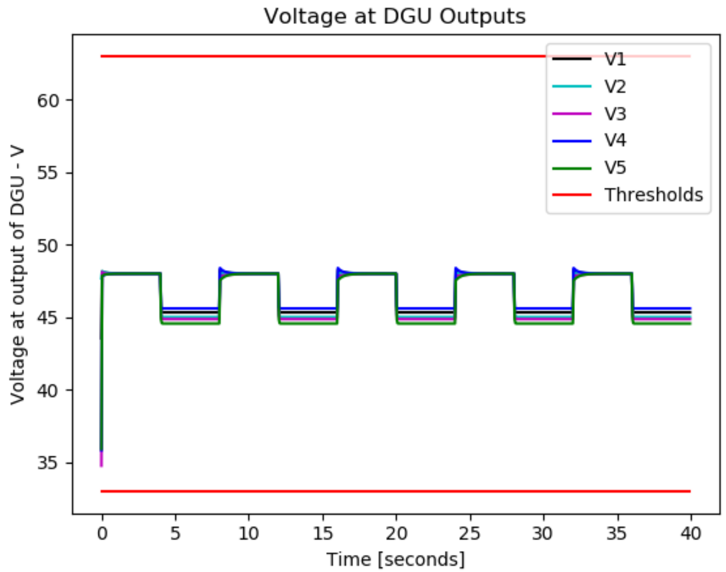
Network Representation

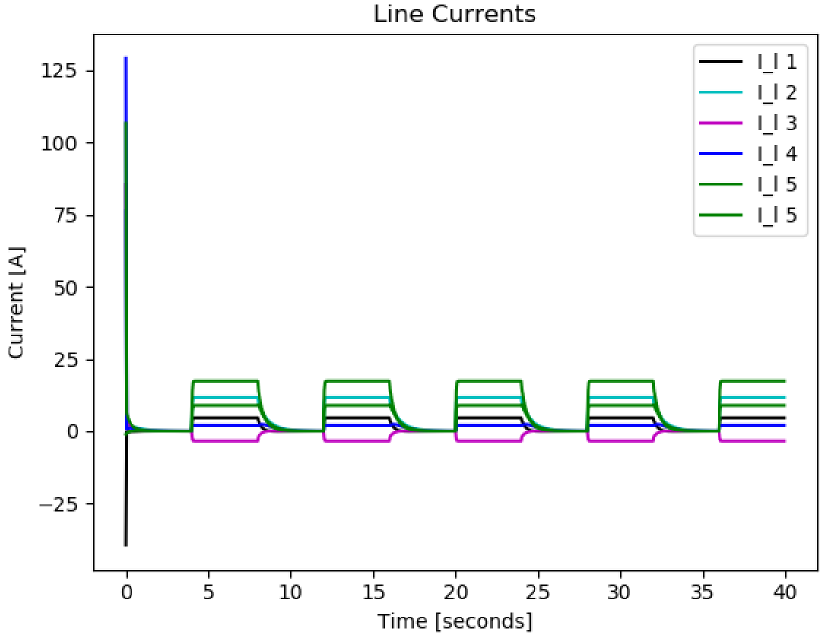


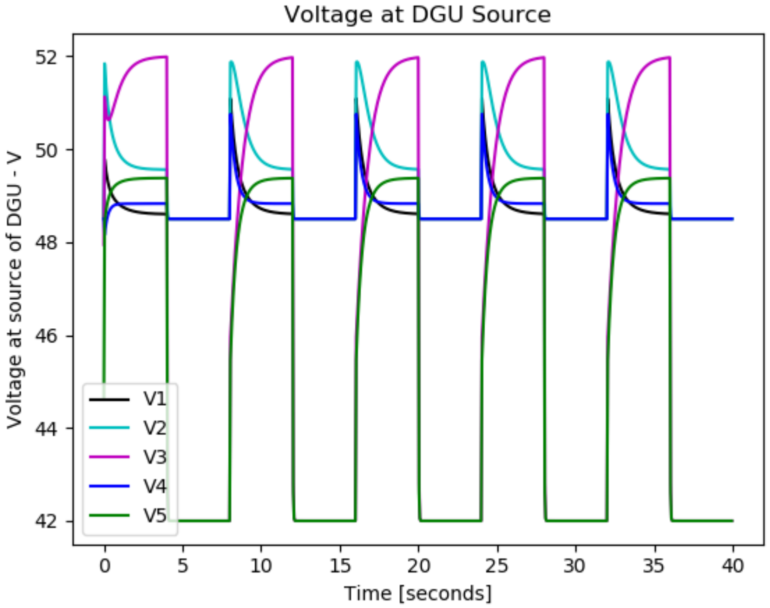
- Timestep = 2.0e-5
- Battery Capacitance = 3.5 Ah
- Loads < 10 A
- SoC(Vs) dependency
- Bumpless Transfer

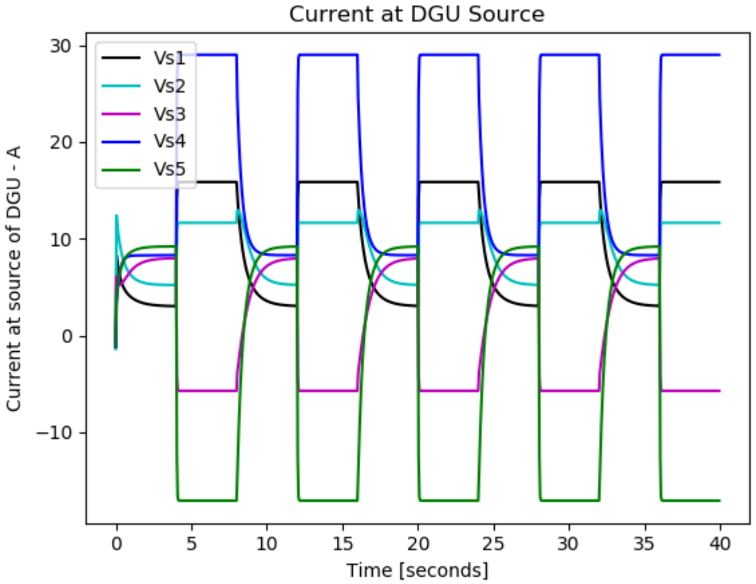
Bumpless demonstration

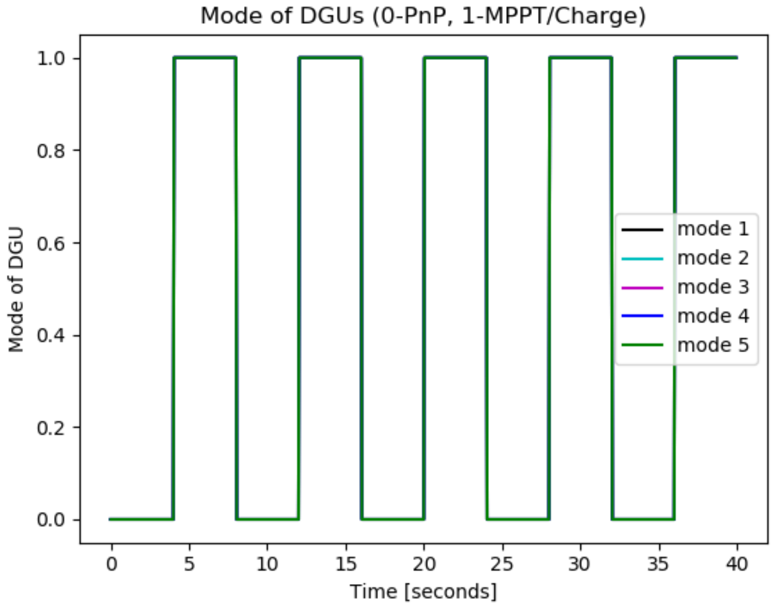












RL controller

- **Purpose** – Maintain output Voltage and State of Charge in healthy zone
- **State Representation** : 7 values -> 5 output voltage and 2 State of Charge Continuous, range : [0; 100]
- **Action Representation** : 5 values -> local control mode for each DGU Discreet, either 0 (PnP) or 1 (MPPT/Charge)
- **Reward** : - negative reward (cost) when state is out of healthy zone

  - small positive reward when state close to ideal state
- **Environment** : Simulator

Value function approximation

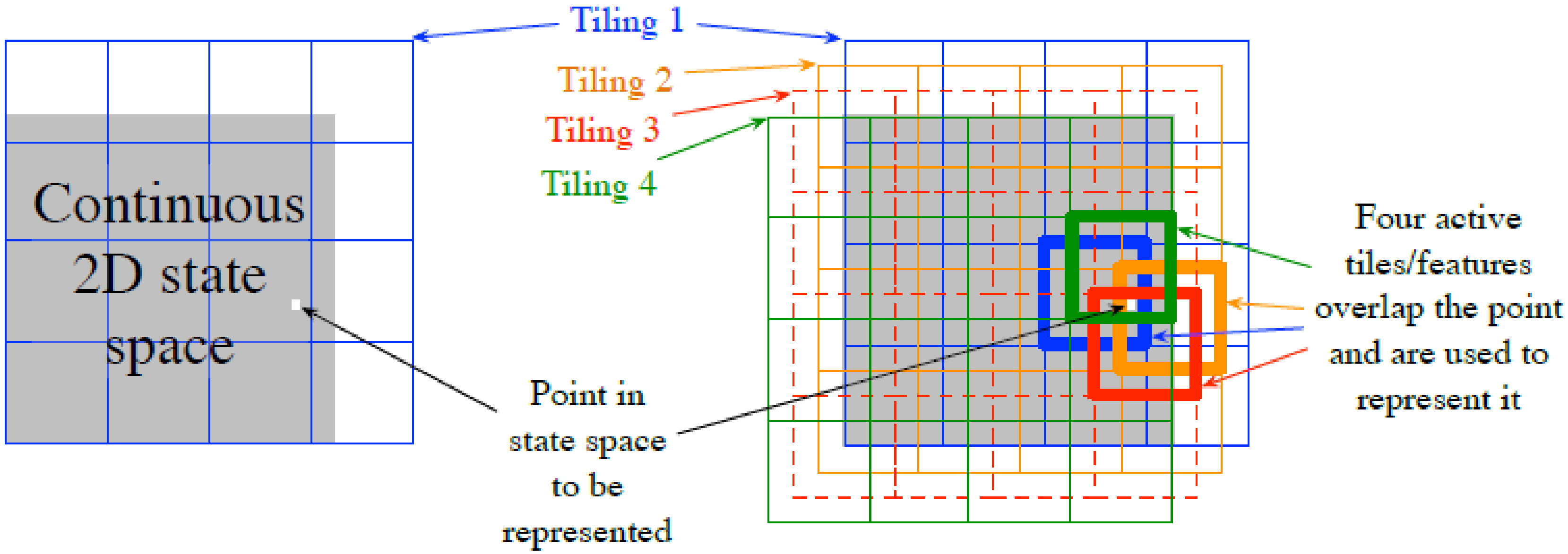
- Continuous state-space –> need to approximate Q functions

Several possible methods for Value approximation :

- Simple state aggregation -> didn’t work, unprecise
- Tile Coding -> successful implementation
- Neural networks -> no time to explore settings but used with good results

Tile Coding

- Method presented and used by R. Sutton
- Coarse coding with several layers, uses hashing for faster computation
- Always represents state with same number of features (number of tilings)
- Chosen resolution : 6V for V and 2.5% for SoC



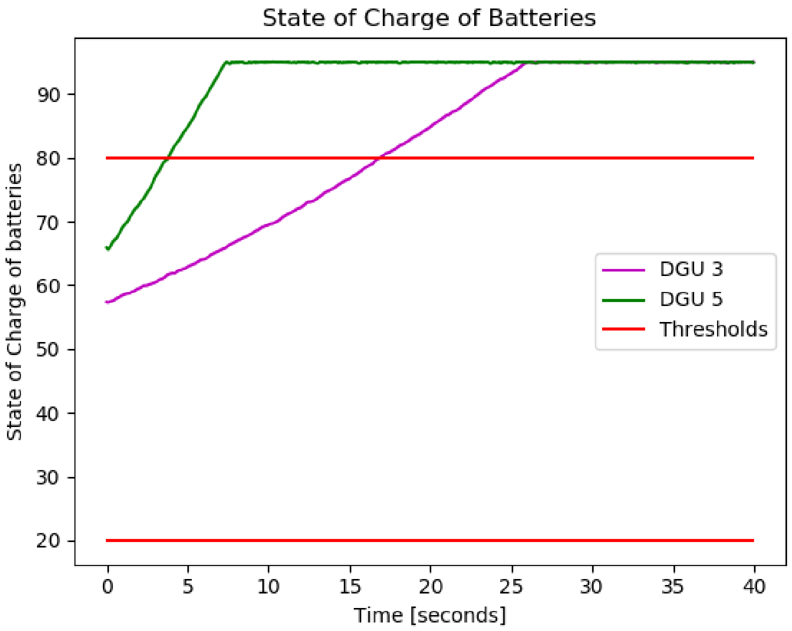
Notes for implementation

Many possible parameters to tweak, finding the 'best' combination is difficult

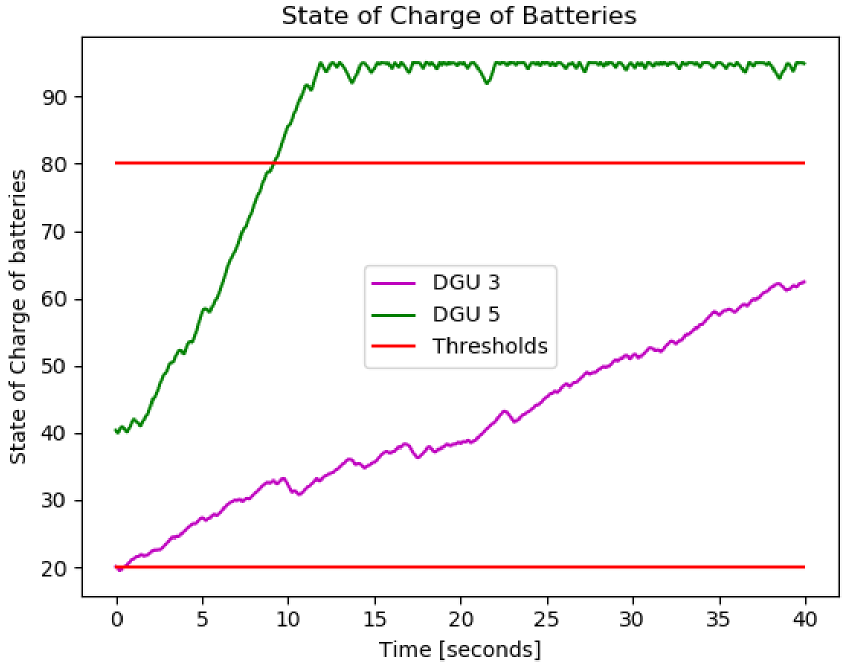
- Length of episodes, number of episodes
- Action Frequency
- RL algo hyperparameters : exploration rate, discount factor, α, β
- State initialization at start of episode
- Reward
- Value Function Approximation parameters

Action frequency too low

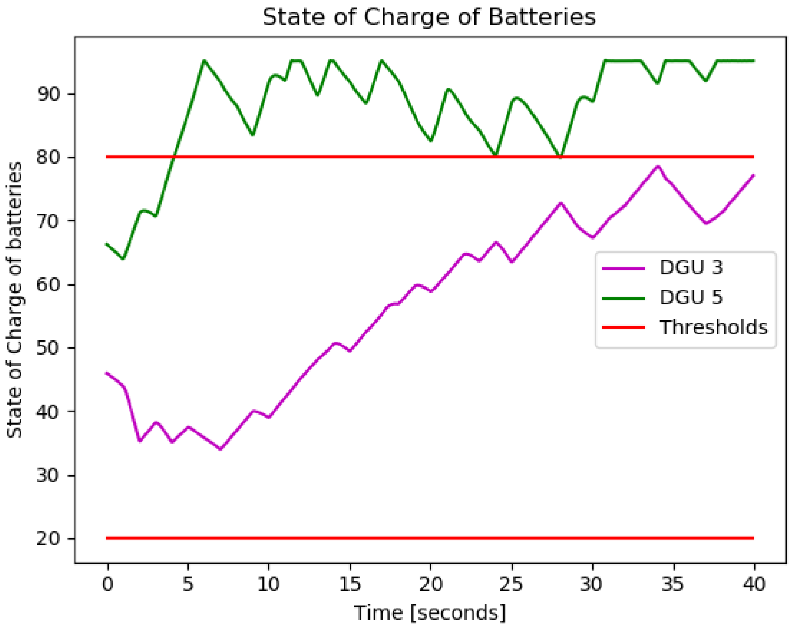
Action Frequency 10 ms



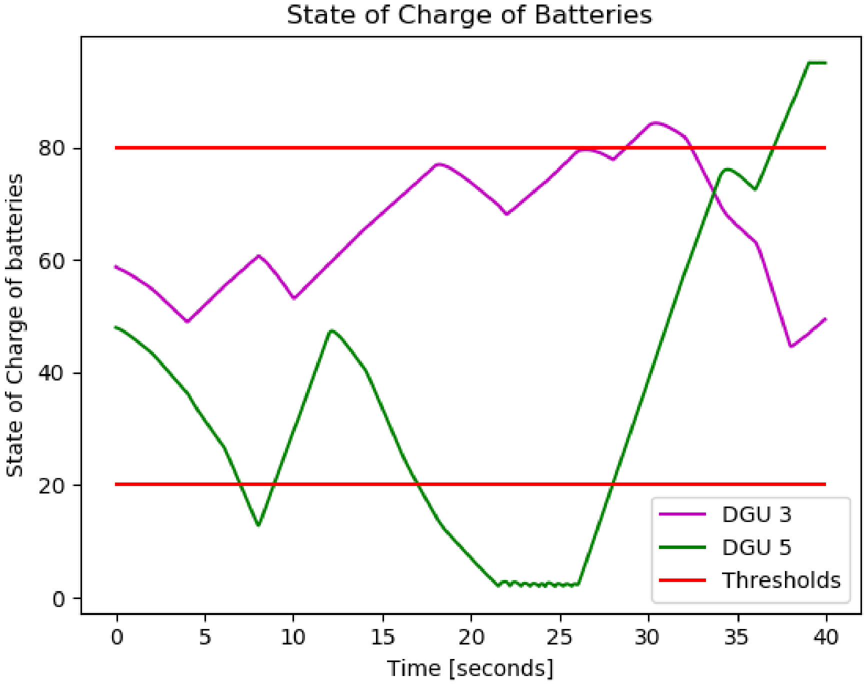
Action Frequency 100 ms



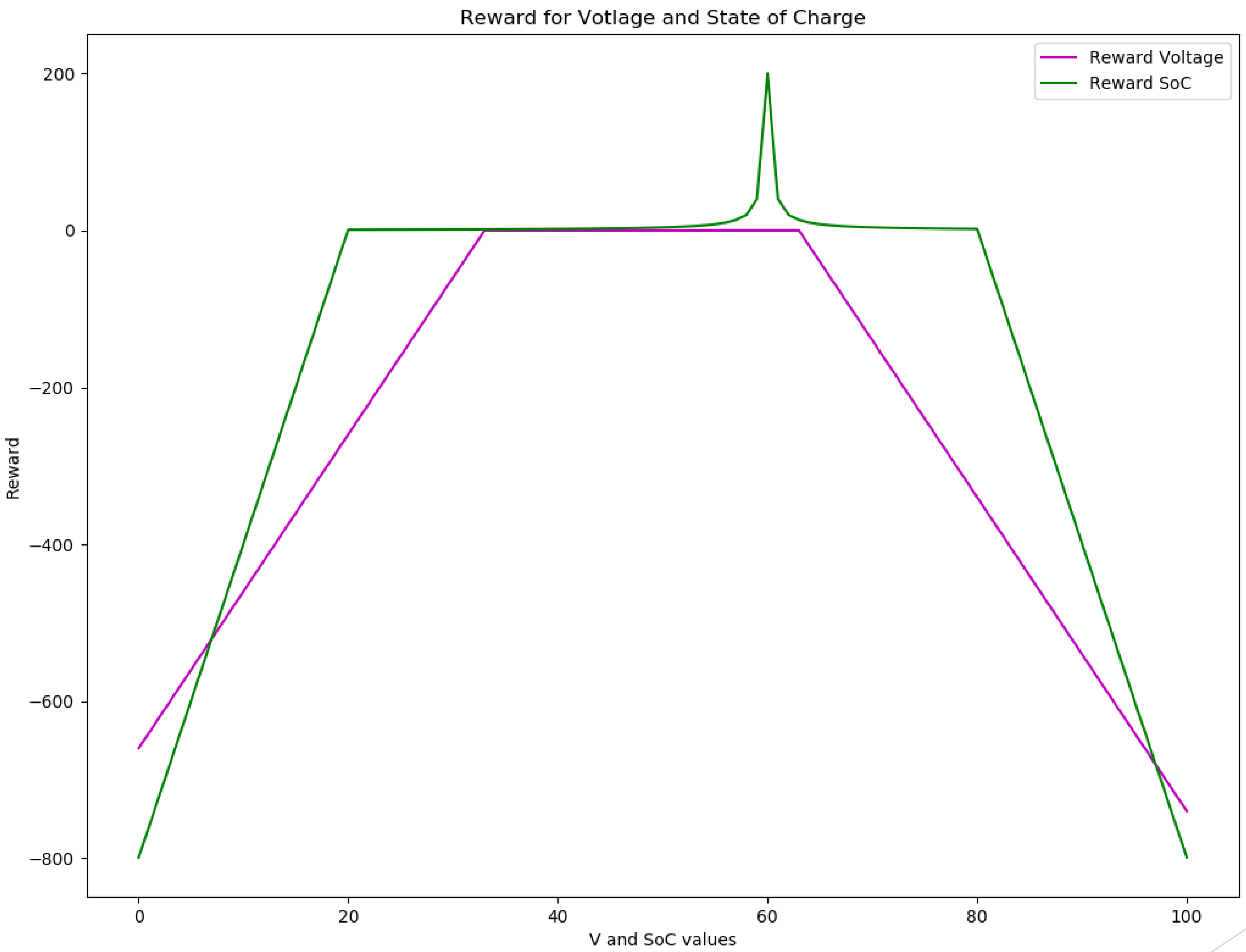
Action Frequency 1 s



Action Frequency 2 s



Reward representation



- SoC reward more important than V
- Guides agent toward reference

First controller - SARSA

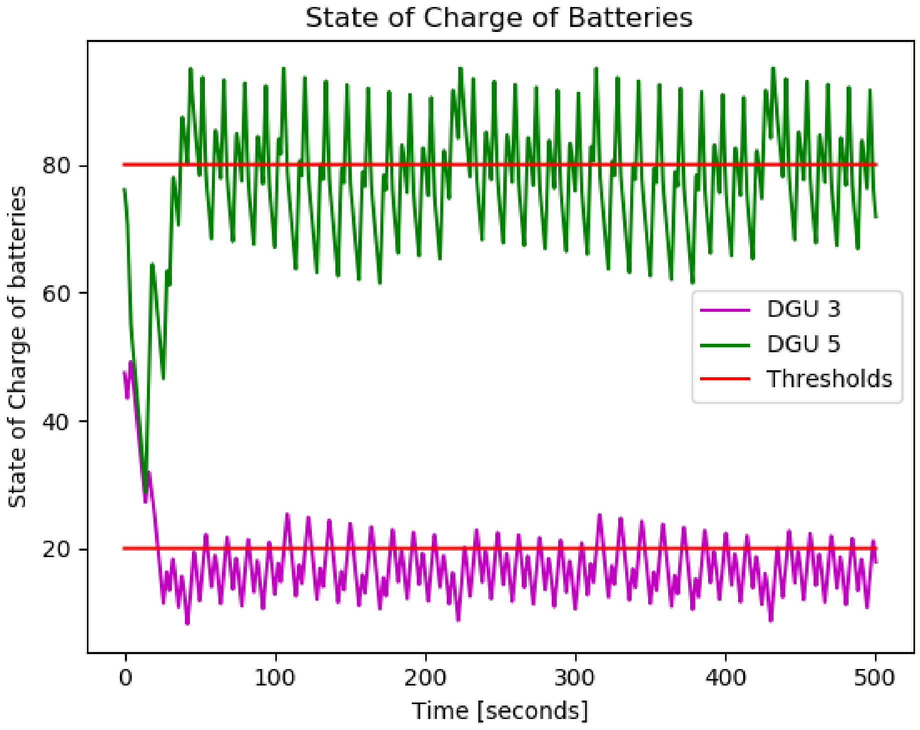
Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

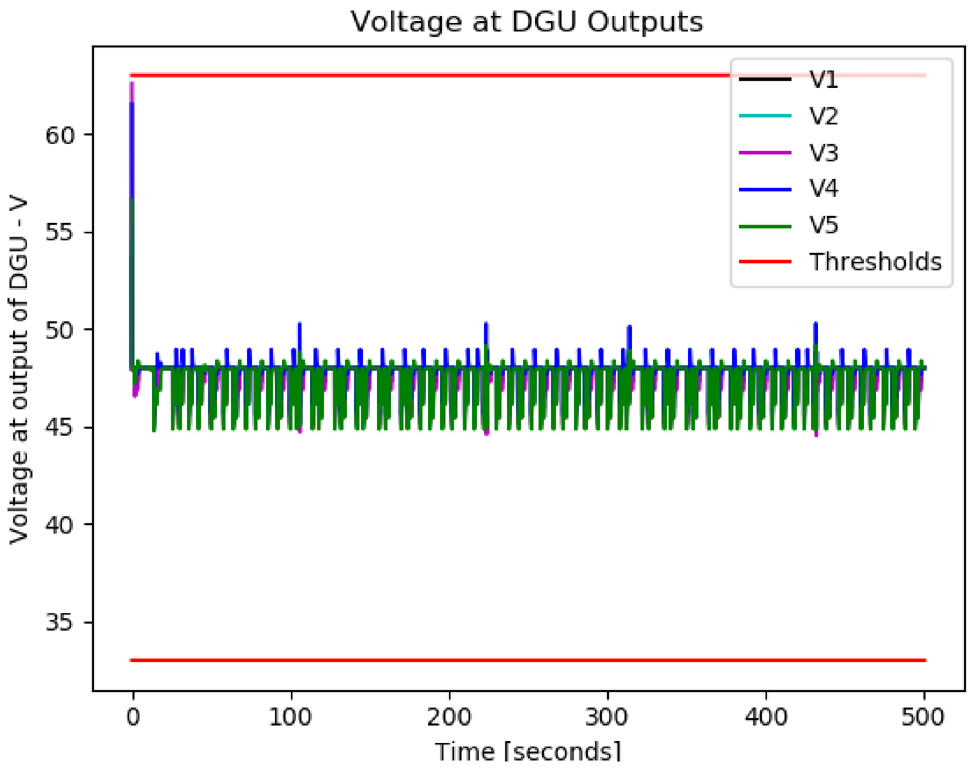
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S, and action A
Loop for each step:
    Take action A, observe R, S'
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
    $\bar{R} \leftarrow \bar{R} + \beta\delta$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

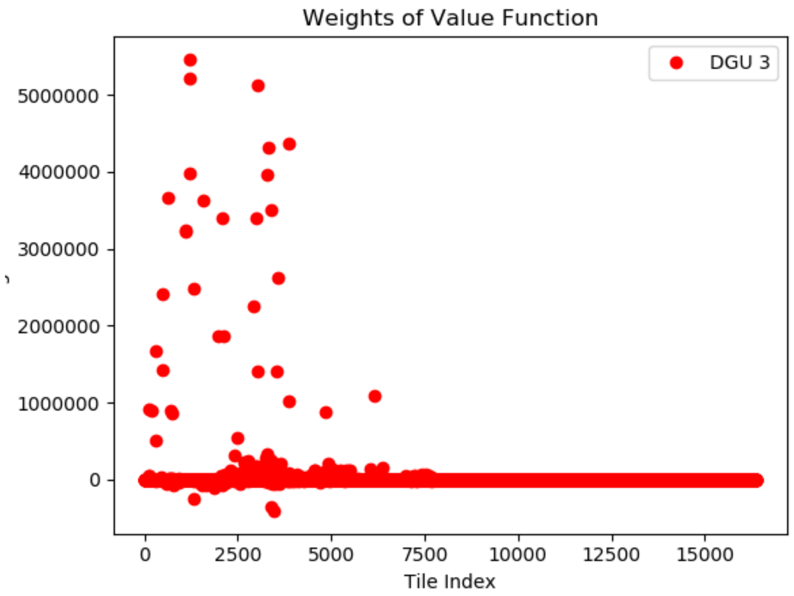
SARSA v13

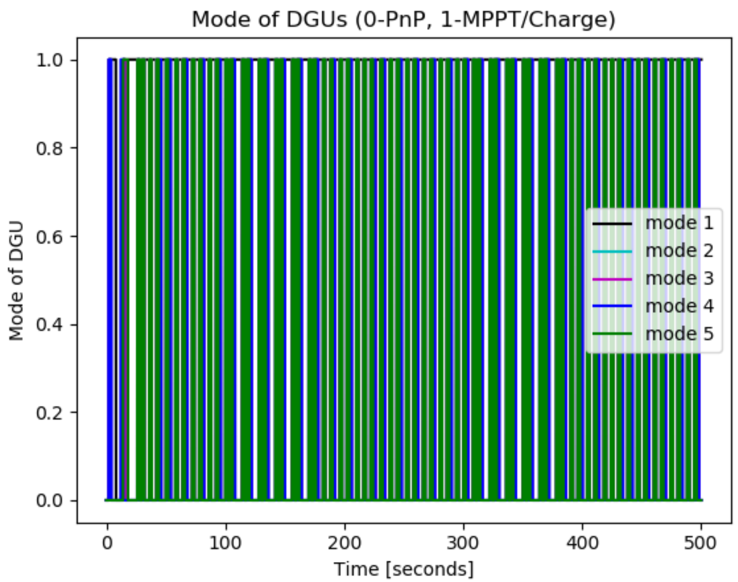
Testing





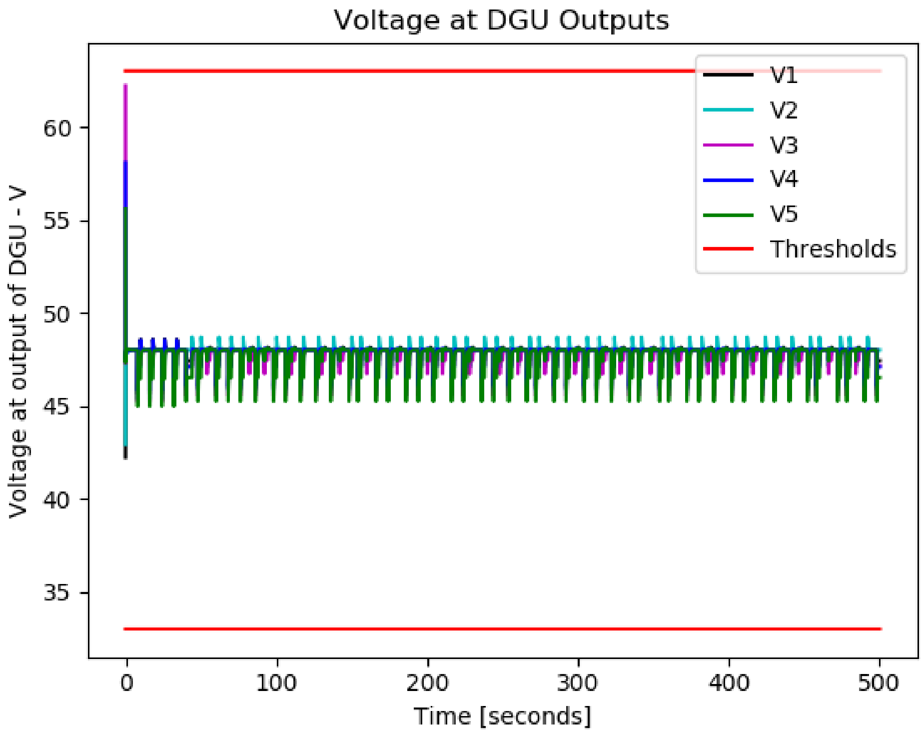
- 1h of training
- 2 episodes
- 30 minutes per episode
- 2s action frequency



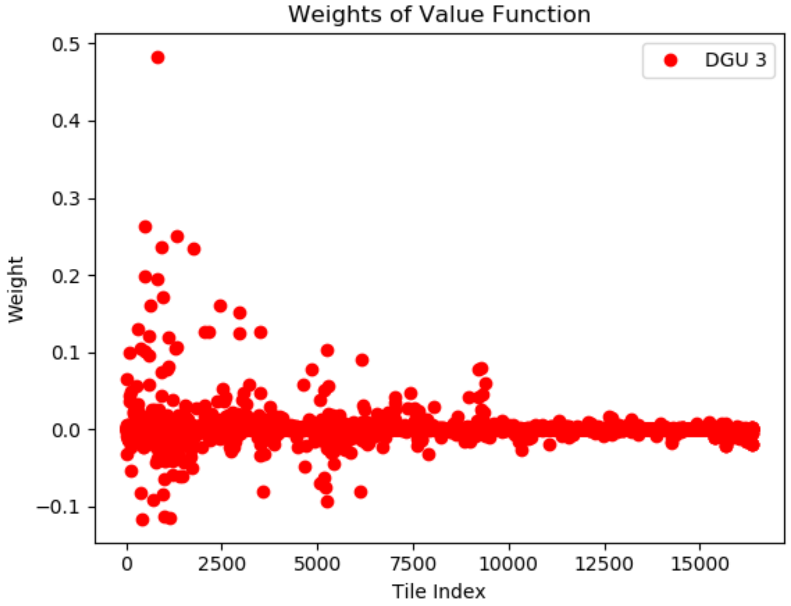


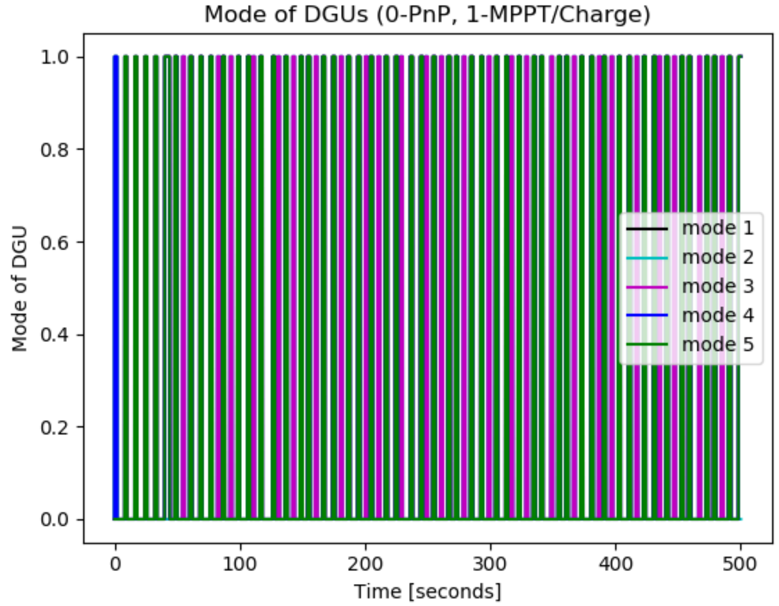
SARSA v15

Testing

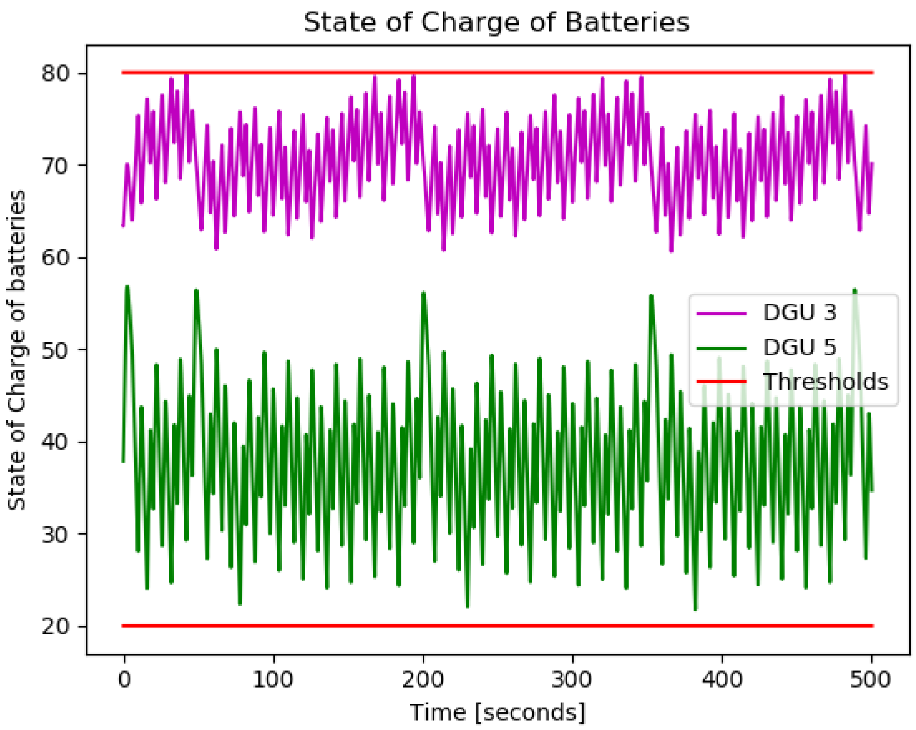


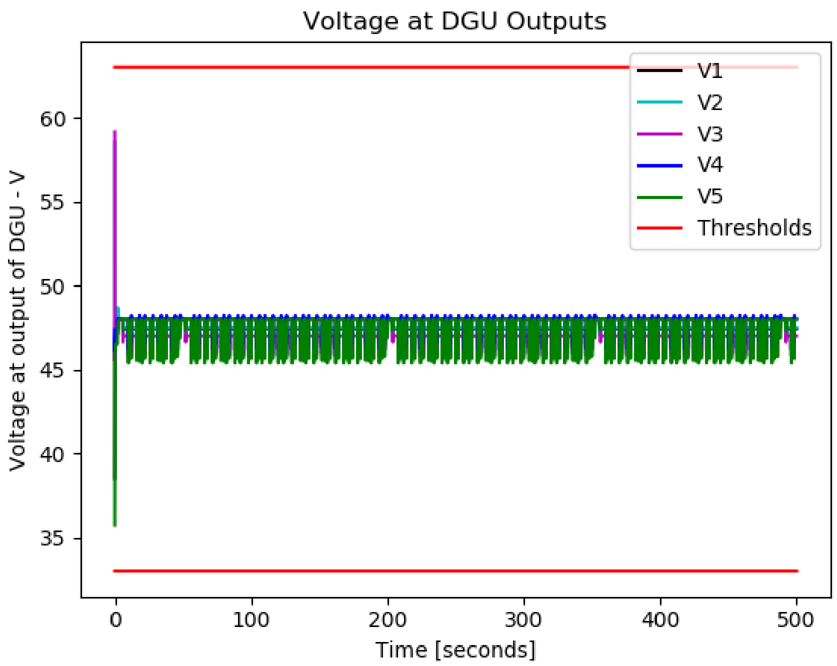
- Trained for 2 hours
- 20 episodes
- 3 minutes per episode
- 2s action frequency



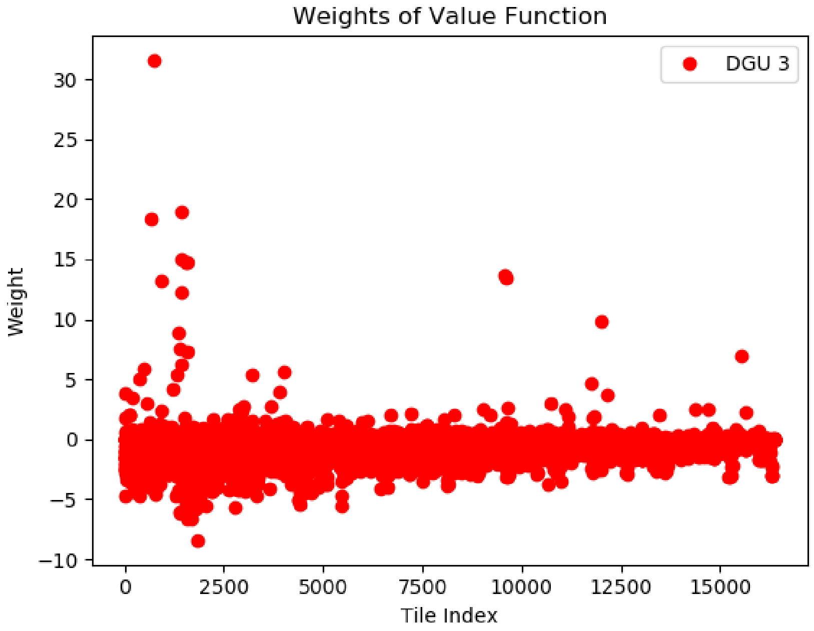


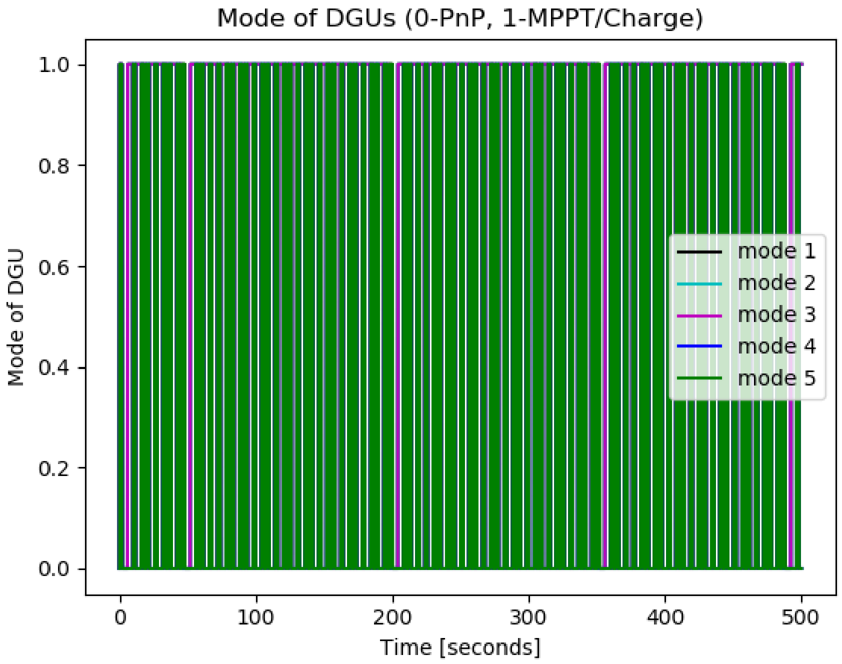
SARSA v17 Testing





- 2h of training
- 20 episodes
- 3 mins per episode
- 2s action frequency





Second Controller - Qfit

ALGORITHM 3.1 Least-squares approximate Q-iteration for deterministic MDPs.

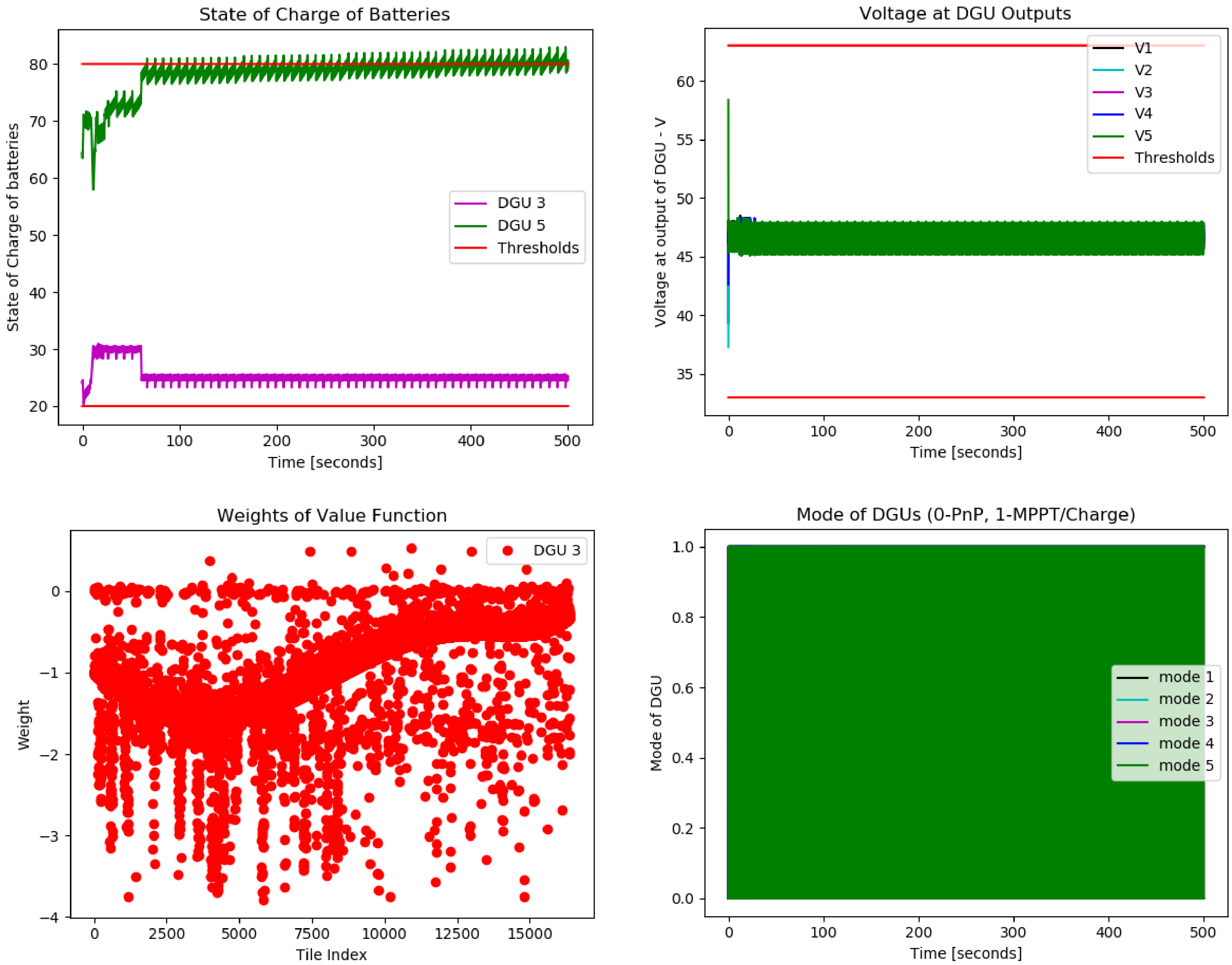
Input: dynamics f, reward function ρ, discount factor γ,
approximation mapping F, samples $\{(x_{l_s}, u_{l_s}) \mid l_s = 1, \dots, n_s\}$

initialize parameter vector, e.g., $\theta_0 \leftarrow 0$
repeat at every iteration $\ell = 0, 1, 2, \dots$
  for $l_s = 1, \dots, n_s$ **do**
    $Q^{\ddagger}_{\ell+1}(x_{l_s}, u_{l_s}) \leftarrow \rho(x_{l_s}, u_{l_s}) + \gamma \max_{u'} [F(\theta_\ell)](f(x_{l_s}, u_{l_s}), u')$
  end for
  $\theta_{\ell+1} \leftarrow \theta^{\ddagger}$, where $\theta^{\ddagger} \in \arg\min_\theta \sum_{l_s=1}^{n_s} \left(Q^{\ddagger}_{\ell+1}(x_{l_s}, u_{l_s}) - [F(\theta)](x_{l_s}, u_{l_s}) \right)^2$
until $\theta_{\ell+1}$ is satisfactory

Output: $\widehat{\theta}^* = \theta_{\ell+1}$

Qfit v2.2

Testing



- 1h40 of training
- 15 episodes
- Training sample : 1000
- Action frequency : 0,2 s

Python librairies

- Keras-rl -> Implementation of several Deep RL algorithms :
  - Deep Q learning (DQN), Double DQN, Deep Deterministic Policy Gradient (DDPG), Continuous DQN (CDQN or NAF), Cross-Entropy Method (CEM) , Dueling network DQN (Dueling DQN) , Deep SARSA
  - Use processor for environement to facilitates adaptation of action, state and reward from environement to RL algorithm
- Garage -> toolkit for developping and evaluating RL algorithms
  - CEM, CMA-ES, REINFORCE, DDPG, DQN, DDQN, ERWR, NPO, PPO, REPS, TD3, TNPG, TRPO
- Several smaller open-source repositories on Github with various RL implementations

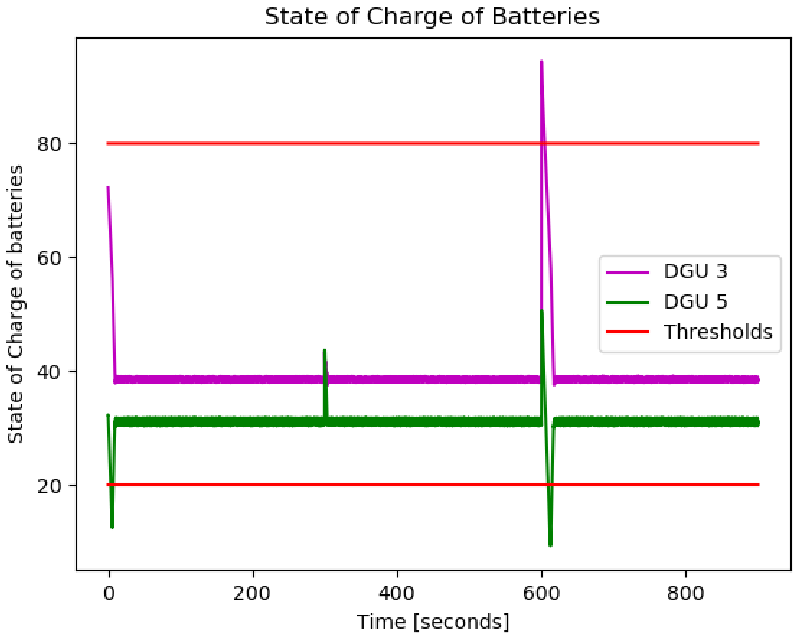
Normalized Advantage Functions

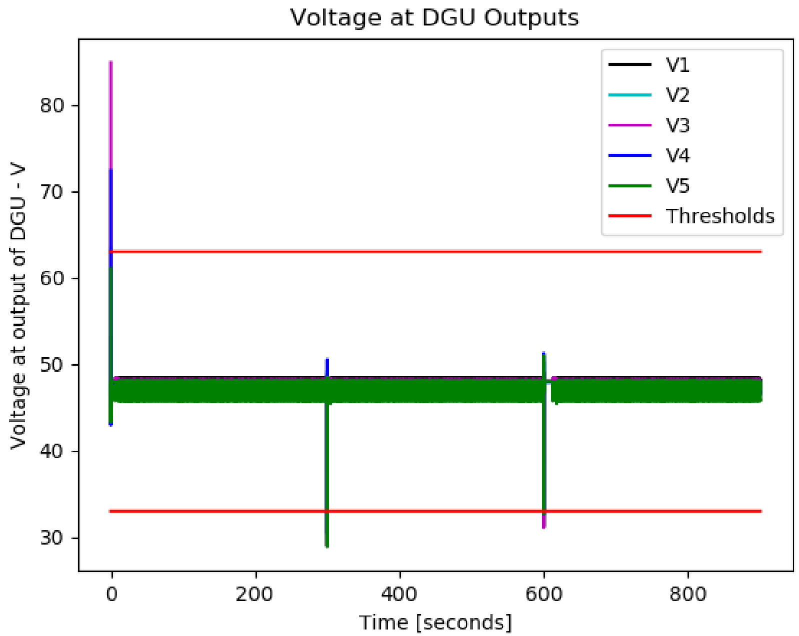
Algorithm 1 Continuous Q-Learning with NAF

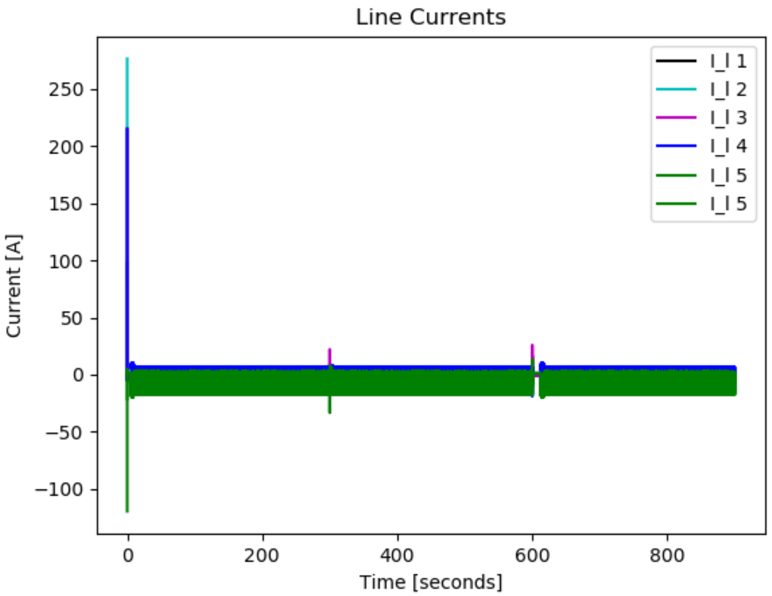
Randomly initialize normalized Q network $Q(\boldsymbol{x}, \boldsymbol{u}|\theta^Q)$.
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$.
Initialize replay buffer $R \leftarrow \emptyset$.
for episode=1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $\boldsymbol{x}_1 \sim p(\boldsymbol{x}_1)$
  for t=1, T **do**
    Select action $\boldsymbol{u}_t = \mu(\boldsymbol{x}_t|\theta^\mu) + \mathcal{N}_t$
    Execute $\boldsymbol{u}_t$ and observe r_t and $\boldsymbol{x}_{t+1}$
    Store transition $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$ in R
    for iteration=1, I **do**
      Sample a random minibatch of m transitions from R
      Set $y_i = r_i + \gamma V'(\boldsymbol{x}_{i+1}|\theta^{Q'})$
      Update θ^Q by minimizing the loss: $L = \frac{1}{N}\sum_i(y_i - Q(\boldsymbol{x}_i, \boldsymbol{u}_i|\theta^Q))^2$
      Update the target network: $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    end for
  end for
end for

Should work best for continuous agent in continuous state-space

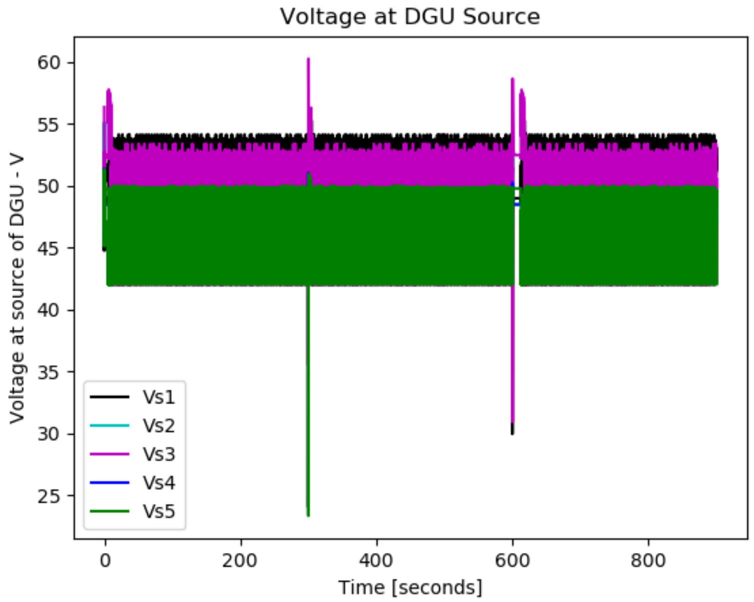
NAF results

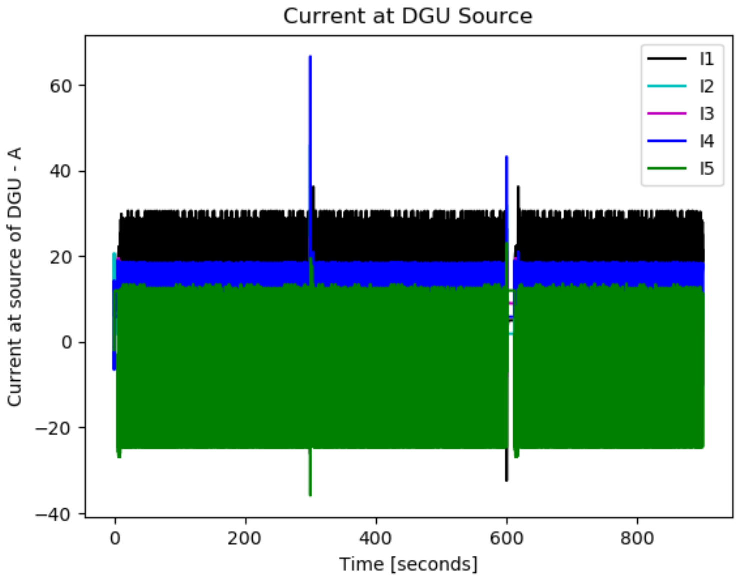






- 25 minutes of training
- 1 episode
- Action frequency : 100 ms





Algorithms Comparaison

- Behavior of agent
  - SARSA – Many oscillations but stays well withthin range
  - Qfit – Many oscillations, stays close to range border and SoC5 tends to go up
  - NAF – Many oscillations but SoC stays constant

- Training time
  - SARSA – 2h
  - Qfit – 1h40
  - NAF – 25 minutes

- Action Frequency
  - SARSA – 2 seconds / 0.5 Hz
  - Qfit – 200 ms / 5 Hz
  - NAF – 100 ms / 10Hz

Potential future work

- Explore Neural Network approximation and adapt it to model
- Deep SARSA and Deep Q learning exploration
- Upgrade model - make it more realistic/complex
  - Add Solar variation to PV
  - Test different, bigger networks
  - Increase Battery Capacitance
  - Enable DGU disconnection from Network

Thank you !

Any questions ?