A Python Script for the Fitted Q Iteration Algorithm

Aivar Sootla

I. FOREWORD

In this note, the code used in the papers [1] and [2] is described. The code realises an extension to the fitted Q iteration algorithm [3], which is concerned with a periodic reference tracking problem. However, the algorithm reduces to the original fitted Q algorithm, if the period is equal to one, that is the reward function is time independent. Hence this code can be seen as an implementation of [3], as well. If you have any questions regarding the implementation, proposals for code modification, suggestions to be implemented please do not hesitate to contact me. The current e-mail is provided below; however, I am postdoc and do not have a permanent affiliation. I hope this code and this note will be helpful to your research, if it is you are welcome to cite:

@INPROCEEDINGS{sootla2013tracking,
    author = {Sootla, A. and Strelkowa, N. and Ernst, D. and Barahona, M. and Stan, G.B.},
    title = {On Reference Tracking Using Reinforcement Learning with Application to Gene Regulatory Networks},
    year = {2013},
    month = {March},
    note = {arXiv:1303.2987},
    booktitle = {submitted to 52nd Conf. Decision Control}
}

This work was performed in collaboration with Dr Guy-Bart Stan and Prof Damien Ernst within the EPSRC project EP/J014214/1, “Data-based optimal control of synthetic biology gene circuits” at the Centre for Synthetic Biology and Innovation, Department of Bioengineering, Imperial College London, UK. I have a feeling that the legal copyright holder of this implementation is Imperial College, therefore in order to protect this fine institution:

The code is provided in good faith, it is free to use, copy, modify, and/or distribute. This code is provided ‘as is’ without warranty of any kind. Any person relying on this code or making any use of this code shall do so at its own risk. The author hereby disclaims any liability and shall not be held liable for any damages. This code may be changed or updated at any time without notice.

A. Introduction to the Note

The code is written in Python 2.7.3, using scikit-learn 0.13.1 [4] used to construct approximators in the form of regression trees, joblib 0.7.0 [5] used to parallelisation, MatPlotLib 1.1.1-1 [6] used for plotting the results, and SciPy 0.10.1 [7] used for scientific computation. It was tested only in Linux operating systems Ubuntu 12.04, 12.10 and 13.04. Note scikit-learn

Aivar Sootla is with the Centre for Synthetic Biology and Innovation and the Department of Bioengineering, Imperial College London, UK a.sootla@imperial.ac.uk.
is provided in Ubuntu repositories as python-sklearn (as well as joblib, which ships with scikit-learn); however if you are using Ubuntu 12.04, you would need to install a new version of scikit-learn.

The structure of the code exploits class inheritance; therefore in order to test it on your problem, you will need to create children to some of the classes (specifically, model and runModel). An example describing how it may be done is present in examples.py (car on hill in Section IV). The base classes of the programme are distributed among the following files: rl.py (contains the classes which realise the fitted Q algorithm), plots.py (contains the classes for plotting the results), models.py (contains the base classes for describing a model of a system), and examples.py (contains car on hill and left and right examples). The biological control examples (toggle switch [1] and generalised repressilator [2]) will be added in the beginning of August.

B. Missing Features and Known Issues

There are few useful feature missing in this implementation, and there are, of course, some issues. Please feel free to expand this list with suggestions and issues by contacting me.

Pruning of the decision trees is missing. It could be a very useful feature for stochastic problems as demonstrated in [3]. Pruning can be implemented by choosing appropriately parameter min_samples_split using n-fold cross validation. There are plans to implement this features.

Fixing the tree structure is impossible at the moment. Fixing the structure of the trees is suggested in [3] to achieve convergence of the algorithm. As far as I understand the structure of the decision trees in scikit-learn prohibits such a manipulation.

Exploration/Exploitation strategies are not implemented, except for the ε-greedy control using a single trajectory. These strategies will be implemented in the near future.

Parallelisation is a very useful feature of the code; however, it should be used with care. For unknown reasons parallelisation in the regulation problem (the reward function does not depend on time) is not efficient. In fact, for small problems, the use of larger number of cores can lead to longer computational times. This issue is illustrated in the example in Section IV. Moreover, in some bits of the code, the data is copied, while being passed to the parallel computation functions, which can lead to memory exhaustion for large problems and parallelisation with a large number of cores.

C. Organisation

This note is organised as follows. In Section II a general idea of functionality of the classes is provided. The main methods for the implementation of particular problems are described in Section III. The example is given in Section IV. A detailed description of the classes and their most important methods is given in Appendix II, while the periodic reference tracking fitted Q Algorithm is described in Appendix I.

II. DESCRIPTION OF THE CLASSES

- Classes in models.py:
  - runModel - The main class constructed to run the simulations. In this class, the instances of all the other classes are created, the optimal policy given this data is computed and the controlled trajectory is plotted. For specific examples, inheritance may be required to tweak the functionality.
- **model** - The class with the description of the model and some problem specific definitions such as reference trajectory, reward function. In order to use for a specific example, the children should be created, which overload some of the class’ methods.

- Classes in rl.py:
  - **dataSet** - The class is used for the creation and storage of the set of one-step transitions $F = \{n_i, u_i, n_i^+\}_{i=1}^F$ (see Appendix I), values of the $Q$ function, the policy function.
  - **baseRL** - The class, with methods computing the $Q$ function, controlling the system using the computed $Q$ function, $\varepsilon$-greedy control.
  - **approximator** - The class is the approximator to the $Q$ and policy functions. It is a list of instances of the `ExtraTreeRegressor` or `ExtraTreeClassifier` classes from `scikit-learn` library. The length of the list is equal to the period of the reference trajectory (or equal to one, if the rewards are time-independent).
  - A useful feature of the code is the ability to parallelise some of the computations, such as the generation of one step transitions, and building the trees. Building the trees is the most expensive procedure. At the moment, the structure of the regression tree cannot be fixed and new trees are built on every iteration of the $Q$ function. All bits of the code, which are run in parallel, are extracted from the classes to stand-alone functions, which are called all through another stand-alone method `paralleliseMethod`. This is done since `joblib` cannot pickle object instances.

- Classes in plots.py
  - **oneDimPlots** Plots the results for one dimensional problems (that is, the $V$ function depends on one state)
  - **twoDimPlots(oneDimPlots)** Plots the results for two dimensional problems (that is, the $V$ function depends on one state)
  - Note that it is straightforward to plot the trajectories depending on time given those two classes regardless of the dimension of the model.

### III. Short Description of Some Methods

The methods in the **runModel** and **model** classes.

- **runModel.initModel** - Initialises and returns an instance of the **model** class.
- **runModel.generateData** - Initialises and returns an instance **dataSet** class by computing the set of one-step transitions. Contains the input data to the fitted $Q$ algorithm - a set of one-step transitions
- **runModel.computeBatchMode** - Computes the $Q$ function based on the set of one-step transitions in **dataSet** and **model**, adds the values of the $Q$ and policy functions to the instance of **dataSet**.
- **runModel.batchModeControl** - Given a computed $Q$ or policy functions, controls the **model**.
- **runModel.plotResults** - Plots the results of the **batchModeControl** given a system trajectory `trajectory`, exploratory actions `explore_actions`, $Q$ function `qTree`, the policy function `policyTree`. This method exploits methods of (the children of) **oneDimPlots**.
- **runModel.runBatchMode** - The main method for the batch-mode control. First calls **runModel.initModel**, which returns an instance of the **model** class. After that calls **runModel.generateData** thus creates an instance of **dataSet** and generates data. Then calls **runModel.computeBatchMode**, which computes the approximators for $Q$ and policy functions as variables `qTree` and `policyTree`. Note that `qTree` and `policyTree` are lists of instances of **approximator** classes. The length of each list `qTree` and
policyTree is equal to model.period, every entry of qTree is an extra trees regressor, which approximates the $Q$ function, and policyTree is an extra trees classifier, which approximates the policy function $\arg\max_u Q(x, u)$. Finally computes the trajectories by calling runModel.batchModelControl and plots them using the method runModel.plotResults

• model/newState - Returns the value of the successor state given a state-action pair.
• model/reward - Returns the value of the instantaneous reward provided a state-action pair in one vector learn_sample, current value of the reference trajectory reference_point, and the successor sample next_sample.
• model/terminalSet - Checks if sample is still within the bounds defined by model/terminal_set
• model/computeReferencePoint - Returns the value of the reference trajectory at time time_sample

IV. Car on the Hill Example

In order to test the algorithm, run these lines in the python environment:

```python
import pylab as pl
from examples import runCarHill
runCH = runCarHill(n_jobs = 4, verbose = 1)
all_classes = runCH.runBatchMode(
    n_trajectories = 10000,
    n_samples = 1,
    n_iterations = 50,
    n_estimators = 50,
    max_features = "auto",
    min_samples_split = 2,
    discount_factor = 0.75)
pl.show()
```

Input parameters in the class constructor:

- n_jobs - the number of computer cores used in the computation
- verbose - toggle output verbose = 1, otherwise verbose = 0

Input parameters in the method runBatchMode:

- n_trajectories - the number of generated trajectories
- n_samples - the maximum number of samples in each trajectory
- n_iterations - the number of iterations in the fitted Q algorithm
- discount_factor - the discount factor in the objective function
- n_estimators, max_features, min_samples_split - parameters of the ExtraTree classes for scikit-learn. These are number of trees, the number of features to consider when looking for the best split, the minimum number of samples required to split an internal node, respectively.

In this example, three children classes are created:

- runCarHill - a child of runModel
- carHill - a child of model
- carHillPlots - a child of twoDimPlots; however, only one method is overloaded

Most of the methods in the class runCarHill are overloaded; however, one could overload just initModel and runBatchMode. The method defineBounds, which limits the state-space of the plots, is overloaded in carHillPlots a child of twoDimPlots. In this example the whole state-space of the problem is plotted. The main work is performed in the class carHill, which includes overloading just a few methods: newState, computeReferencePoint,
<table>
<thead>
<tr>
<th>Number of used cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000 one step transitions</td>
<td>80</td>
<td>70</td>
<td>47</td>
<td>62</td>
</tr>
<tr>
<td>5000 one step transitions</td>
<td>223</td>
<td>147</td>
<td>101</td>
<td>95</td>
</tr>
<tr>
<td>10000 one step transitions</td>
<td>440</td>
<td>289</td>
<td>185</td>
<td>156</td>
</tr>
<tr>
<td>20000 one step transitions</td>
<td>864</td>
<td>521</td>
<td>343</td>
<td>277</td>
</tr>
<tr>
<td>100000 one step transitions</td>
<td>4579</td>
<td>2554</td>
<td>1617</td>
<td>1271</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of used cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000 one step transitions</td>
<td>48</td>
<td>54</td>
<td>38</td>
<td>37</td>
</tr>
<tr>
<td>5000 one step transitions</td>
<td>140</td>
<td>105</td>
<td>78</td>
<td>83</td>
</tr>
<tr>
<td>10000 one step transitions</td>
<td>278</td>
<td>202</td>
<td>141</td>
<td>133</td>
</tr>
<tr>
<td>20000 one step transitions</td>
<td>498</td>
<td>330</td>
<td>230</td>
<td>211</td>
</tr>
<tr>
<td>100000 one step transitions</td>
<td>2707</td>
<td>1595</td>
<td>1036</td>
<td>926</td>
</tr>
</tbody>
</table>

**The Computational Time of Data Generation with a Different Number of Cores (in Seconds)**

<table>
<thead>
<tr>
<th>Number of used cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000 one step transitions</td>
<td>32</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>5000 one step transitions</td>
<td>83</td>
<td>42</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>10000 one step transitions</td>
<td>162</td>
<td>87</td>
<td>44</td>
<td>23</td>
</tr>
<tr>
<td>20000 one step transitions</td>
<td>368</td>
<td>191</td>
<td>113</td>
<td>66</td>
</tr>
<tr>
<td>100000 one step transitions</td>
<td>1872</td>
<td>959</td>
<td>580</td>
<td>343</td>
</tr>
</tbody>
</table>

**A. Performance**

The code was tested on an HP Z 820 workstation equipped with two eight-core Intel Xeon E5-2670 processors, 128 GB of RAM, Ubuntu 13.04. The parameters of the approximators are set to default values, the data points are generated randomly, in every example fifty iterations are computed. The memory requirement for the examples presented below is at most 10 GB of RAM (for 100000 one-step transitions and 8 cores). Note that the more cores is used, the more memory is required. Some data, while being dispatched for parallel computation is actually copied, which creates this issue.

There are some other issues with parallelisation of the fitted Q Algorithm, which can be clearly seen in Table I. For small problems the large number of cores can actually result in longer computational times. There are two suspected reasons for this, firstly, some time is spend on the dispatch of parallel tasks. When the tasks are easy to compute, then more time is spent of dispatching then actually computing. Secondly, I use scikit-learn, joblib libraries, which I cannot control and do not really know; hence some redundancies are introduced into my code. Therefore be advised to use an appropriate number of cores for your computations.

**REFERENCES**


The situation becomes worse if more than 10 cores is used; however, the performance is not affected if the periodic reference tracking problem is considered. In this case, the parallelisation is done within the script without delegating it to scikit-learn.
APPENDIX I. THE REFERENCE TRACKING FITTED Q ITERATION

B. Fitted Q Iteration

1) Problem Formulation: Consider a deterministic discrete-time dynamical system

\[ n_{t+1} = f(n_t, u_t) \]  

(1)

where the bold values \( n \) stand for vectors with elements \( n^i \), \( n_t \) is the state of the system at time \( t \), and \( u_t \) is a control input, which at each time \( t \) belongs to a compact set \( U \). Consider also, associated with this dynamical system, an optimal control problem, which is defined in terms of the minimisation of an infinite sum of discounted costs \( c(n, u) \):

\[ \min_{\mu()} \sum_{i=t}^{\infty} \gamma^{i-t} c(n_i, \mu(n_i)) \]

where \( \mu(\cdot) \) is a mapping from the state-space onto \( U \), which is called the feedback control policy, and \( \gamma \) is a positive constant smaller than one, which is called the discount factor. For the purpose of this paper, it is assumed that \( c(\cdot, \cdot) \) is a given function. The goal is to compute the optimal policy based only on one-step transitions of the system (1). One-step system transitions are given as a set \( F = \{ n_l, u_l, n_l^+ \}_{l=1}^k \# F \), where \( n_l^+ \) denotes a successor state of the system in state \( n_l \) subjected to input \( u_l \) (if the function \( f(\cdot, \cdot) \) is known then \( n_l^+ \) is simply equal to \( f(n_l, u_l) \)).

2) Algorithm: The central object in Fitted Q Iteration is the \( Q \) function, which is defined as follows:

\[ Q(n_l, u_l) = c(n_l, u_l) + \min_{\mu()} \sum_{i=t+1}^{\infty} \gamma^{i-t} c(n_i, \mu(n_i)). \]

The main idea of the approach is to exploit the celebrated Bellman equation

\[ Q(n, u) = c(n, u) + \gamma \min_{u' \in U} Q(f(n, u), u'), \]

(2)

which provides a convenient expression for the computation of the optimal feedback control policy:

\[ \mu^*(n) = \arg\min_{u \in U} Q(n, u) \]  

(3)

The Bellman equation can be theoretically solved using an iterative procedure

\[ Q_k(n, u) = c(n, u) + \gamma \min_{u' \in U} Q_{k-1}(f(n, u), u'), \]

where \( Q_0 = c \) and \( Q_\infty \) is the unique solution to (2) due to the fact that \( T(Q) = c + \gamma \min_{u' \in U} Q \) is a contraction mapping (cf. [8]). However in the continuous state-space case, this iterative procedure is hard to solve, especially when only the one-step system transitions in \( F \) are given. The Fitted Q iteration algorithm computes, from the sole knowledge of \( F \) a sequence of functions \( \hat{Q}_1, \hat{Q}_2, \ldots \) that approximates the sequence \( Q_1, Q_2, \ldots \). Let \( \hat{Q}_0 = c \) and for every \( (n_l, u_l, n_l^+) \) in \( F \) compute:

\[ \hat{Q}_1(n_l, u_l) = c(n_l, u_l) + \gamma \min_{u \in U} \hat{Q}_0(n_l^+, u). \]  

(4)

This expression gives \( \hat{Q}_1 \) only for \( n_l, u_l \) in \( F \), while the entire function \( \hat{Q}_1(\cdot, \cdot) \) is estimated using a regression algorithm (e.g., EXTRA Trees [9]). Now the iterative procedure is derived by generalising (4) as follows:

\[ \hat{Q}_k(n_l, u_l) = c(n_l, u_l) + \gamma \min_{u \in U} \hat{Q}_{k-1}(n_l^+, u), \]

(5)
Algorithm 1 Fitted $Q$ iteration

**Inputs:** Set of one-step system transitions $\mathcal{F} = \{n_i, u_i, n_i^+\}_{i=1}^{\#\mathcal{F}}$, cost $c(\cdot, \cdot)$, stopping criterion

**Outputs:** Policy $\hat{\mu}^*(n)$

```
\begin{align*}
k & \leftarrow 0 \\
Q_k(\cdot, \cdot) & \leftarrow c(\cdot, \cdot) \\
\text{repeat} \\
k & \leftarrow k + 1 \\
\text{Compute (5) to obtain the values of } \hat{Q}_k(\cdot, \cdot) \text{ for all } \{n_i, u_i\} \text{ in } \mathcal{F} \\
\text{Estimate the function } \hat{Q}_k(n, u) \text{ using a regression algorithm with input pairs } (n_i, u_i) \text{ and function values } \hat{Q}_k(n_i, u_i). \\
\text{until stopping criterion is satisfied} \\
\text{Compute the policy according to (6) }
\end{align*}
```

where at each step $\hat{Q}_k(\cdot, \cdot)$ is estimated using a regression algorithm. If the iteration procedure stops at the iteration number $k$, an approximate policy can be computed as follows:

$$
\hat{\mu}^*(n) = \arg\min_{u \in U} \hat{Q}_k(n, u)
$$

(6)

A simple stopping criterion can be, for example, an *a priori* fixed maximum number of iterations $N_{it}$. The value $N_{it}$ is chosen such that $\gamma^{N_{it}}$ is sufficiently small and, hence, the values $\hat{Q}_k(n_i, u_i)$ are not modified significantly for $k$ larger than $N_{it}$. Other, more advanced, stopping criteria are described in [3]. The resulting iterative method is outlined in Algorithm 1. A major property of the fitted $Q$ iteration algorithm is convergence. This is understood as convergence of $\hat{Q}_k$ to a fixed state-action value function $Q^*$ given a fixed set $\mathcal{F}$ as $k$ grows to infinity. It was shown in [3], that under certain conditions Algorithm 1 converges and it is possible to estimate in advance the distance between $\hat{Q}^*$ and the iterate $\hat{Q}_k$.

C. Periodic Reference Tracking Using the Fitted $Q$ Iteration Algorithm

1) Problem Formulation: Consider, a system:

$$
\begin{align*}
n_{t+1} & = f(n_t, u_t) \\
r_{t+1} & = g(r_t)
\end{align*}
$$

(7)

where the function $f(\cdot, \cdot)$ is unknown. The function $g(\cdot)$, however, is a known periodic function of period $T$, and the variable $r$ takes only a finite number of values $\{v_i\}_{i=1}^T$. Hence, $g(v_i)$ is equal to $v_{i+1}$ for all $i$ smaller than $T$, and $g(v_T)$ is equal to $v_1$. The reference tracking problem is defined as follows:

$$
\begin{align*}
\min_{\mu(\cdot, \cdot)} & \sum_{i=t}^{\infty} \gamma^{i-t} c(d(n_i, r_i), n_i, u_i) \\
\text{subject to } & \text{system dynamics (7), and} \\
& \mu(n_i, r_i) = u_t \in U
\end{align*}
$$

(8)

where $c$ is a given instantaneous cost function, $d(\cdot, \cdot)$ is a function defining the distance between the current state $n$, and reference $r$, and $\mu(\cdot, \cdot)$ is a feedback control policy. In order to track the reference $r$, reducing the value of the cost $c$ must reduce the distance $d$. The instantaneous cost can optionally depend on the control signal $u$ and the states $n$ in order to provide additional constraints in the state-space and/or control action space. As for the Fitted $Q$ Iteration, the control policy is inferred based solely on the trajectories given in the form of one-step system transitions $\mathcal{F} = \{n_i, u_i, n_i^+\}_{i=1}^{\#\mathcal{F}}$, where $n_i^+$ denotes a successor state of the system in state $n_i$ subjected to input $u_i$. 

2) Algorithm: The variable \( r \) can be seen as an additional state in the extended state-space \( \{n, r\} \). Based on this extended state-space we can derive the Bellman equation for the tracking problem and the following iterative procedure for the computation of the \( Q \) function:

\[
Q_k(n, r, u) = c(d(n, r), n, u) + \min_{u' \in U} Q_{k-1}(n^+, g(r), u') \quad \forall n, r, u
\]  

(9)

where \( Q_0 \) is equal to \( c \). It can be shown that this iterative procedure has a unique solution \( Q^* \) based on a similar contraction mapping argument as in the previous section. Hence the optimal policy in the extended state-space is computed as:

\[
\mu(n, r) = \min_{u \in U} Q^*(n, r, u)
\]

The input set to the algorithm normally consists of the one-step system transitions \( F = \{n_i, u_i, n_i^+\}_{i=1}^\#F \). However, since our state-space has been extended to include \( r \), \( F \) should now include \( r \) as well. Since the time evolution of \( r \) is known a priori we can simply modify the set as follows:

\[
T F = \{n_i, v_i, u_i, n_i^+, g(v_i)\}_{i,t}
\]

where \( i = 1, \ldots, T \) and \( l = 1, \ldots, \#F \). This will effectively copy \( T \) times the training set \( F \). Now given this modification, we can proceed to the computational procedure. As before \( Q_0 \) is equal to \( c \) and the next iterates can be obtained according to:

\[
\hat{Q}_k(n_i, v_i, u_i) = c(d(n_i, v_i), n_i, u_i) + \min_{u' \in U} \hat{Q}_{k-1}(n_i^+, g(v_i), u') \quad \forall l, v_i
\]

(10)

According to the Fitted Q Iteration framework, every function \( \hat{Q}_k(\cdot, \cdot, \cdot) \) must be estimated by a regression algorithm, which uses the input set \( \{n_i, v_i, u_i\}_{i,l} \) and the corresponding values of the approximated function \( \hat{Q}_k(n_i, v_i, u_i) \). This input set can grow significantly, if the period \( T \) of the reference trajectory is large, which can render a regression algorithm computationally intractable. For example, with only a thousand one-step system transitions \( \{n_i, u_i, n_i^+\}_{i=1}^\#F \) and period \( T \) equal to 200, the total number of samples \( \{n_i, u_i, u_i\}_{i,l} \) is equal to 200,000. Therefore, it is proposed to break up the regression of \( \hat{Q}_k(\cdot, \cdot, \cdot) \) into \( T \) independent regression problems, one for every function \( \hat{Q}_k(\cdot, v_i, \cdot) \). This can be done, because the evolution of the variable \( r \) is known in advance and it takes a finite number of values. To make these ideas more transparent, (10) is rewritten using a different notation as follows:

\[
\hat{Q}_k^{v_i}(n_i, u_i) = c(d(n_i, v_i), n_i, u_i) + \min_{u' \in U} \hat{Q}_{k-1}^{g(v_i)}(n_i^+, u) \quad \forall l, v_i
\]

(11)

where \( \hat{Q}_k^{v_i}(\cdot, \cdot) \) stands for the function \( \hat{Q}_k(\cdot, v_i, \cdot) \). For every value \( v_i \), the regression algorithm will approximate the function \( \hat{Q}_k^{v_i}(\cdot, \cdot) \) by using the input set \( \{n_i, u_i\}_{i,l} \) and the corresponding values \( \hat{Q}_k^{v_i}(n_i, u_i) \). Thus the initial regression problem is indeed separated into \( T \) independent problems.

Our approach can be also seen as a modification of the \( \hat{Q}_k \) function approximator. The first layer of our approximator is a deterministic branching according to the values \( v_i \). After that a regression algorithm is performed to approximate the functions \( \hat{Q}_k^{v_i}(\cdot, \cdot) \) as prescribed. Finally, if the iterative procedure has stopped at iteration \( k \), a near-optimal policy can be computed as follows:

\[
\hat{\mu}^*(n, r) = \min_{u \in U} \hat{Q}_k^*(n, u)
\]

(12)

Our approach is outlined in Algorithm 2. Periodicity is a crucial assumption, since the period of the reference trajectory corresponds to the number of different \( \hat{Q}_k^* \) functions built in this algorithm. Convergence of Algorithm 2 can be established using the following lemma.

**Lemma 1:** Algorithm 2 converges under similar conditions and considerations as the fitted \( Q \) iteration algorithm in [3]. Moreover, the stopping criteria from [3] can be directly applied to Algorithm 2.
Algorithm 2 Reference Tracking Fitted $Q$ Iteration

Inputs: Sets of one-step system transitions $\mathcal{F} = \{n_i, u_i, n_i^+\}_{i=1}^{\#\mathcal{F}}$, function $g(\cdot)$ and reference values $\{v_i\}_{i=1}^{\mathcal{T}}$, cost $c(d(\cdot, \cdot), \cdot, \cdot)$, stopping criterion

Outputs: Policy $\hat{\mu}^*(n, r)$

1. $k \leftarrow 0$
2. $\hat{Q}_0(\cdot, \cdot) \leftarrow c(d(\cdot, \cdot), \cdot, \cdot)$
3. repeat
   1. $k \leftarrow k + 1$
   2. Compute (11) to obtain the values of $\hat{Q}_k^{v_i}(\cdot, \cdot)$ for all $\{n_i, u_i\}$ in $\mathcal{F}$ and $v_i$
   3. Estimate the functions $\hat{Q}_k^{v_i}(n, u)$ for every $v_i$ using a regression algorithm with input pairs $(n_i, u_i)$ and function values $\hat{Q}_k^{v_i}(n_i, u_i)$.
4. until stopping criterion is satisfied
5. Compute the policy using (12)

To prove this lemma, we have to make sure that the approximator of the $\hat{Q}_k$ functions will not break the convergence proof in [3]. Only one modification in this approximator is made, which is the deterministic branching in its first layer according to $v_i$. It can be shown that this modification does not violate the convergence arguments in [3].

Note, if we assume that the cost function is time-independent, i.e., $r$ is constant and equal to $v$, and $g(v)$ is equal to $v$, equation (11) becomes:

$$\hat{Q}_k^{v}(n_i, u_i) = c(d(n_i, v), n_i, u_i) + \min_{u' \in U} \hat{Q}_{k-1}^{v}(n_i^+, u') \quad \forall l$$

which reduces Algorithm 2 to Algorithm 1 without any artefacts.

APPENDIX II. List of Classes and Description of the Methods

D. Class runModel

Constructor:

def __init__(self, 
    n_jobs = 1, 
    verbose = 1)

- n_jobs - the number of computer cores used for the computation
- verbose - toggling/untoggling output. If verbose is equal to one, the internal output will be printed.

Major methods in the class:

def runBatchMode(self, 
    n_trajectories = 1, 
    n_samples = 1, 
    n_iterations = 1, 
    n_estimators = 50 
    max_features = "auto", 
    min_samples_split = 2, 
    period = 1, 
    max_time = 100, 
    initial_set = np.array([0, 1]), 
    initial_state = None, 
    discount_factor = 0.75, 
    reward_metric = 'euclidean')
Inputs:

- `n_trjectories` - an input to `dataSet`. The number of trajectories computed for generation of one step transitions.
- `n_samples` - an input to `dataSet`. The number of samples in each generated trajectory
- `initial_set` - An input to `dataSet`. Initial points of every trajectory are randomly generated in within the `initial_set` set
- `initial_state` - the initial point, which has the dimension of the full state of the model
- `max_time` - control horizon
- `discount_factor` - An input to `baseRL`. The discount factor, larger than zero, smaller than one.
- `n_iterations` - An input to `baseRL`. the number of iterations in the fitted Q algorithm
- `n_estimators`, `max_features`, `min_samples_split` - parameters of the ExtraTree classes for scikit-learn, number of trees, the number of features to consider when looking for the best split, the minimum number of samples required to split an internal node, respectively.
- `reward_metric` - a metric for computing the distance between the reference trajectory and current state.
- `period` - the period of the reference trajectory. If `period` is equal to one, it is simply the regulation problem.

Outputs:

- `my_data` - an instance of the class `dataSet`
- `my_model` - an instance of the class `model`
- `base_class` - an instance of the class `baseRL`
- `qTree` - an instance of the class `approximator`. The regressor, which approximates the Q function
- `policyTree` - an instance of the class `approximator`. The classifier, which approximates the policy function

The main method for batch-mode control. First calls `initModel`, which returns an instance of the `model` class. After that calls `generateData` thus creates an instance of `dataSet`. Passes these instances to an instances `baseRL`, which computes the approximator for Q and policy functions as variables `qTree` and `policyTree`. Note that `qTree` and `policyTree` are lists of instances of `approximator` classes. The length of each list `qTree` and `policyTree` is equal to `_model.period`, every entry of `qTree` is an extra trees regressor, which approximates the Q function, and `policyTree` is an extra trees classifier, which approximates the policy function \( \text{argmax}_u Q(x,u) \). Finally computes the trajectories and plots them using instances of (the children of) `oneDimPlots`.

```python
def initModel(self,
               reward_metric = 'euclidean',
               period = 1):
```

Inputs:

- `period` - the period of the reference trajectory. If `period` is equal to one, it is simply the regulation problem.
- `reward_metric` - a metric for computing the distance between the reference trajectory and current state.

Outputs:

- `my_model` - an instance of the `model` class.

Initialises and returns an instance of the `model` class.

```python
def generateData(self,
                  my_model,
```
Inputs:
- `my_model` - an instance of the `model` class.
- `initial_set` - an input to `dataSet`. Initial points of every trajectory are randomly generate in within the `initial_set` set.
- `n_trajectories` - an input to `dataSet`. The number of trajectories computed for generation of one step transitions.
- `n_samples` - an input to `dataSet`. The number of samples in each generated trajectory.

Outputs:
- `my_data` - an instance of the class `dataSet`

Initialises and returns an instance `dataSet` class.

```python
def computeBatchMode(self, my_data, my_model, n_iterations = 1, period = 1, discount_factor = 0.75, qTree = None, policyTree = None, n_jobs = 1):
```

Inputs:
- `my_data` - an instance of the class `dataSet`
- `my_model` - an instance of the `model` class.
- `n_iterations` - An input to `baseRL`. the number of iterations in the fitted Q algorithm
- `period` - the period of the reference trajectory. If period is equal to one, it is simply the regulation problem.
- `discount_factor` - An input to `baseRL`. The discount factor, larger than zero, smaller than one.
- `qTree` - an instance of the class `approximator`. The regressor, which approximates the Q function.
- `policyTree` - an instance of the class `approximator`. The classifier, which approximates the policy functions.
- `n_jobs` - the number of computer cores used for the computation

Outputs:
- `base_class` - an instance of the class `baseRL`
- `qTree` - an instance of the class `approximator`. The regressor, which approximates the Q function
- `policyTree` - an instance of the class `approximator`. The classifier, which approximates the policy function.

Computes the Q function using the fitted Q iteration algorithm.

```python
def batchModeControl(self, my_model, base_class, qTree, policyTree, initial_state = None,
```
max_time = 400,
discount_factor = 0.75,
policy_computation = 'Qfun')

Inputs:
- my_model - an instance of the model class.
- base_class - an instance of the class baseRL
- qTree - an instance of the class approximator. The regressor, which approximates the Q function
- policyTree - an instance of the class approximator. The classifier, which approximates the policy functions
- initial_point - the initial point, which has the dimension of the full state of the model
- max_time - the control horizon
- discount_factor - the discount factor in objective function
- policy_computation - defines which approximator to use for action computations qTree (policy_computation= Qfun’) or policyTree (policy_computation =’policy’). The policy computed based on qTree and policyTree can be slightly different, by default use qTree.

Outputs:
- trajectory - trajectory of the full states and actions of the model
- explore_actions - exploratory actions taken during the control phase. If only batch-mode control was used, then explore_actions is an empty list.

Controls my_model using a policy computed in computeBatchMode

```python
def plotResults(self,
    my_model,
    trajectory = None,
    explore_actions = None,
    plot_type = 'D',
    action_space = None,
    policyTree = None,
    qTree = None
nSam = 100):
```

Inputs:
- trajectory - trajectory of the full states and actions of the model
- explore_actions - exploratory actions taken during the control phase. If only batch-mode control was used, then explore_actions is an empty list.
- plot_type - set the type of the plot
  - ’T’ - the plot of system’s trajectory and reference trajectory
  - ’D’ - the plot of system’s trajectory and distance to the reference trajectory
  - ’V’ - the policy computed based on the function qTree
  - ’P’ - the policy computed based on the function policyTree
  - ’Q’ - the Q function given a specific action (provided in the variable action_space
If the model has two states, then there additional options:
  - ’TV’ - the plot of system’s trajectory overlapped with the policy computed based on the function qTree
  - ’TP’ - the plot of system’s trajectory overlapped with the policy computed based on the function qTree
- action_space - values of actions used for plotting the results
- policyTree - an instance of the class approximator. The classifier, which approximates the policy functions
- qTree - an instance of the class **approximator**. The regressor, which approximates the $Q$ function
- nSam - the number of data point in each axis of the plot

Plots the results of the `batchModeControl` given the system trajectory `trajectory`, exploratory actions `explore_actions`, $Q$ function `qTree`, policy function `policyTree`. The variable `polt_type` defines what to plot. If the model has only one state, then:

```python
def runEpsGreedy(self,  
    max_time = 100,  
    n_iterations = 1,  
    discount_factor = 0.75,  
    update_time = 1,  
    reward_metric = 'euclidean',  
    period = 1,  
    initial_state = None)  
```

**Inputs:**
- `discount_factor` - the discount factor in the objective function, larger than zero, smaller than one.
- `max_time` - control horizon
- `update_time` - the $Q$ function is updated every `update_time` time samples
- `n_iterations` - An input to `baseRL`. the number of iterations in the fitted $Q$ algorithm
- `reward_metric` - a metric for computing the distance between the reference trajectory and current state.
- `period` - the period of the reference trajectory. If `period` is equal to one, it is simply the regulation problem.
- `initial_state` - the initial point, which has the dimension of the full state of the model

**Outputs:**
- `my_data` - an instance of the class `dataSet`
- `my_model` - an instance of the class `model`
- `base_class` - an instance of the class `baseRL`
- `qTree` - an instance of the class `approximator`. The regressor, which approximates the $Q$ function
- `policyTree` - an instance of the class `approximator`. The classifier, which approximates the policy function

**E. Class `model`:**

**Variables:**
- `terminal_set` - the measured state is to be contained within the set `terminal_set`. If it leaves it, the control algorithm stops.
- `n_actions` - the number of actions

**Constructor:**
```python
def __init__(self,  
    action_set = None,  
    n_states = 1,  
    n_measurements = 1,  
    history_window = 0,  
    reward_metric = 'euclidean',  
    control_delay = 0,  
    period = 1):```
Inputs:
- **action_set** - The set of all possible admissible actions. If it is a list of arrays, then the number of actions is the length of the list.
- **n_states** - the number of states in the model.
- **n_measurements** - the number of measured states in the model.
- **history_window** - the length of the window of the past measurements. If zero, then only current measurements are used in the control algorithm, if one, the measurements at time $t$ and $t - 1$ are used in the control algorithm, etc.
- **reward_metric** - a metric for computing the distance between the reference trajectory and current state.
- **control_delay** - control delay of the action.
- **period** - the period of the reference trajectory. If **period** is equal to one, it is simply the regulation problem.

Major methods:

```python
def newMeasurement(self, state):
    Inputs:
    • **state** - the full state of the system
    Outputs:
    • **measured_state** - the observed part of the state of the system
    Returns the measurement of the model seen by the algorithm. By default returns the full state of the system.
```

```python
def newState(self, state, action, final_time = 0.1):
    Inputs:
    • **state** - current full state of the system
    • **action** - current action
    • **final_time** - is the sampling time, if a continuous-time model is being considered.
    Outputs:
    • **next_state** - the successor state
    Returns the value of the successor state given state-action pair.
```

```python
def reward(self, learn_sample, reference_point, next_sample):
    Inputs:
    • **learn_sample** - current sample-action pair in one vector
    • **reference_point** - current value of the reference trajectory
    • **next_sample** - the successor sample
    Outputs:
    • **reward** - instantaneous reward
    returns the value of the instantaneous reward provided measured state-action pair in one vector **learn_sample**, current value of the reference trajectory **reference_point**, and the successor sample **next_sample**.
```
def terminalSet(self, sample):

Inputs:
• sample - current sample

Outputs:
• one if the sample belongs to the model.terminal_set, zero if the sample does not belong to the model.terminal_set.

Checks is sample is still within the bound defined by model.terminal_set

def computeReferencePoint(self, time_sample):

Inputs:
• time_sample - current time

Outputs:
• referencePoint - current value of the periodic reference trajectory

returns the value of the reference trajectory at time time_sample.

def metric(self, valX, valY):

Inputs:
• valX - vector X
• valY - vector Y

Outputs:
• distance between X and Y.

Computes the distance between valX and valY vectors.

Other methods:

def newStateSim(self, state, action, final_time = 0.1, time_steps = 0.1):

Is used if the model is continuous. Computes a successor state and all states in between them with time step time_steps. Creates a smoother trajectory for plots. Not explicitly used in any of the examples. Not really tested in a while!

def newSample(self, state, last_sample):

Creates a 'sample' for the algorithm based on the full state of the system and last sample. By default equal to measured state, if history_window is equal to k, contains measured_state at times from 0 to k.

F. Class dataSet

Variables:
• n_samples - the total of one step transitions
• learning_set - list of input data to approximators. Every entry of the list is a vector, which contains current state and current action
• next_samples - list of successor states. Every entry corresponds to the entry with the same number in learning_set. That is next_samples[ii] is a successor state of state action pair in learning_set[ii]
• current_samples - list of current states.
• exploded_samples - list of all successor states, each with extended by every possible action.
• non_terminal_picks -

Constructor:
```python
def __init__(self, 
    my_model, 
    verbose = 1, 
    n_jobs = 1)
```

Methods:
```python
def generate(self, 
    my_model, 
    initial_set, 
    n_trajectories, 
    n_samples)
```
• my_model - an instance of the model class or its child
• initial_set - an input to dataSet. Initial points of every trajectory are randomly generate in within the initial_set set
• n_trajectories - an input to dataSet. The number of trajectories computed for generation of one step transitions.
• n_samples - an input to dataSet. The number of samples in each generated trajectory

This method is called to generate one step transitions. The input my_model is a class, which describes the model of the controlled system, the initial point of each trajectory are randomly generate in within the initial_set set. The inputs n_trajectories, n_samples define the number of trajectories to be generate and maximum number of samples generate in each trajectory. The trajectory can have less the n_samples, if the current state goes beyond my_model.terminal_set set. These two methods are called from generate method.
```python
def processData(self, 
    my_model, 
    _raw_data)

def genInitialPoints(self, 
    my_model)
```
• model - an instance of the model class or its child
• _raw_data - data set created in the parallel computation loop, unstructured.

Since the trajectories are generated in parallel, processData simply processes the generated data and creates the following dataSet variables, which will be used in the fitted Q algorithm: non_terminal_picks, exploded_samples, n_samples, learning_set current_samples next_samples.

G. Class baseRL

Variables:
• n_jobs - the number of computer cores used for the computation
• verbose - toggling/untoggling output. If verbose is equal to one, the internal output will be printed.
• discount_factor - An input to baseRL. The discount factor, larger than zero, smaller than one.
• n_iterations - An input to baseRL, the number of iterations in the fitted Q algorithm
• reward_metric - A metric for computing the distance between the reference trajectory and current state.

Constructor:
```
def __init__(self, 
n_jobs = 1, 
verbose = 1):
```

Main methods:
```
def batchModeFittedQ(self, 
my_data, 
discount_factor = 0.75, 
qTree = None, 
policyTree = None, 
n_iterations = 1, 
n_jobs = 1):
```

Inputs:
• my_data - an instance of the dataSet class
• qTree - approximator for the Q function
• policyTree - approximator for the policy function
• n_iterations - the number of iterations in the fitted Q algorithm
• n_jobs - number of computed cores used for the computation

Outputs:
• qTree - an instance of the class approximator. The regressor, which approximates the Q function
• policyTree - an instance of the class approximator. The classifier, which approximates the policy function

Computes the Q function in the batch-mode reinforcement learning setting according to the fitted Q algorithm. Modifies _data.
```
def batchModeControl(self, 
my_model, 
qTree, 
policyTree, 
initial_state, 
max_time = 5000, 
discount_factor = 0.75, 
policy_computation = 'Qfun')
```

Controls the system with a policy computed in batchModeFittedQ method. Inputs:
• my_model - an instance of the model class or its child
• qTree - approximator for the Q function
• policyTree - approximator for the policy function
• initial_state - the initial point, which has the dimension of the full state of the model
• max_time - the control horizon
• discount_factor - the discount factor in the objective function
• policy_computation - defines which approximator to use for action computations qTree (policy_computation= 'Qfun') or policyTree (policy_computation = 'policy'). The policy computed based on qTree and policyTree can be slightly different, by default use qTree.
Outputs:

- **score** - sum of accumulated rewards
- **trajectory** - trajectory of the full states and actions of the model
- **explore_actions** - exploratory actions taken during the control phase. If only batch-mode control was used, then `explore_actions` is an empty list.

```python
def epsGreedyControl(self, my_data, my_model, initial_state = None, qTree = None, policyTree = None, max_time = 1000, update_time = 200, vareps = 0, policy_computation = 'Qfun', n_iterations = 30, discount_factor = 0.75):
```

- **my_data** - an instance of the class `dataSet`
- **my_model** - an instance of the `model` class or its child
- **qTree** - approximator for the Q function
- **policyTree** - approximator for the policy function
- **initial_state** - the initial point, which has the dimension of the full state of the model
- **max_time** - the control horizon
- **n_iterations** - the number of iterations in the fitted Q algorithm
- **update_time** - frequency of running the fitted Q iteration algorithm. That is the algorithm is run every `update_time` time samples
- **discount_factor** - An input to `baseRL`. The discount factor, larger than zero, smaller than one.

Outputs:

- **score** - the sum of discounted rewards
- **trajectory** - the trajectory of the full states and actions of the model
- **explore_actions** - the exploratory actions taken during the control phase. If only batch-mode control was used, then `explore_actions` is an empty list.

Methods called internally:

```python
def computeGreedyAction(self, my_model, qTree, policyTree, current_sample, policy_computation):
```

- **my_model** - an instance of the `model` class.
- **qTree** - an instance of the class `approximator`. The regressor, which approximates the Q function
- **policyTree** - an instance of the class `approximator`. The classifier, which approximates the policy function
- **current_sample** - the current sample

Computes a greedy action based on `qTree` or `policyTree`, which chosen according to `policy_computation`. 
```python
def qIteration(self, n_iterations, n_jobs, discount_factor, my_data, qTree, policyTree):
    Inputs:
    • my_data - an instance of the class dataSet
    • my_model - an instance of the model class.
    • n_iterations - An input to baseRL. the number of iterations in the fitted Q algorithm
    • discount_factor - An input to baseRL. The discount factor, larger than zero, smaller than one.
    • qTree - an instance of the class approximator. The regressor, which approximates the Q function
    • policyTree - an instance of the class approximator. The classifier, which approximates the policy function
    Outputs:
    • qTree - an instance of the class approximator. The regressor, which approximates the Q function
    • policyTree - an instance of the class approximator. The classifier, which approximates the policy function
    Computes n_iterations of Q iteration algorithm
def simulateOneStep(self, my_model, current_state, current_action, trajectory, cur_time):
    Inputs:
    • model - an instance of the class model or its child
    • current_state - current full state
    • current_action - current action
    • trajectory - trajectory of the full states and actions of the model
    • cur_time - current time
    Outputs:
    • trajectory of the full states and actions of the model
    • successor state
    • next time sample
    Based on the model model, current state current_state, current action current_action, and current time cur_time returns appends the successor state to the trajectory trajectory, increments time cur_time
def generateRandomAction(self, my_model):
    Inputs:
    • my_model - an instance of the class model or its child
    Outputs:
    • - random action
    Generates and returns a random action for the model.
```
H. Methods Used for Parallelisation

```python
def paralleliseMethod(info_class, 
    method_parallel = None, 
    total_n_jobs = 1, 
    batch = 1, 
    n_jobs = 1, 
    data_set = None):
```

Inputs:
- `info_class` - a class with auxiliary information required for parallelisation, passed to the `method_parallel`
- `method_parallel` - a method which will be run in parallel
- `total_n_jobs` - the total number of jobs
- `batch` - the number of jobs dispatched at once
- `n_jobs` - the maximum number of jobs to be run in parallel
- `data_set` - the input data to the `parallelise_method`. Should be a list with the length `total_n_jobs`

Outputs:
- list of outputs of the methods `method_parallel`

```python
def qFunUpdate(my_data, 
    seq, 
    tree_class):
```
Updates the Q function and rebuilds the tree in `tree_class` based on `_data`, which is an instance of `dataSet`. The variable `seq` is the number of the job, in this case the tree number in the list `qTree`

```python
def fitPTree(my_data, 
    seq, 
    tree_class):
```
Builds the policy function classifier in `tree_class` based on `_data`, which is an instance of `dataSet`. The variable `seq` is the number of the job, in this case the tree number in the list `policyTree`

```python
def computeSeriesRewards(my_model, 
    seq, 
    data_set):
```
Computes the instantaneous rewards for a set of one-step transitions in the instance of `dataSet`. Note the variable `data_set` in this case is `None`.

```python
def computeTrajectories(my_model, 
    seq, 
    next_state):
```
Computes trajectories and thus generates one-step transitions.

I. Class `approximator`

A shell around scikit-learn classes, has two methods `fit` and `predict`, which are calling the corresponding methods in scikit-learn classes.

Variables:
- `tree` - an instance of `ExtraTreeRegressor` or `ExtraTreeClassifier`

```python
def __init__(self, 
    n_estimators = 50, 
)
max_features = "auto",
min_samples_split = 2,
n_jobs = 1,
approximator_type = 'reg'):

The variables n_estimators, max_features, min_samples_split, n_jobs are passed directly to the constructor ExtraTreeRegressor or ExtraTreeClassifier.
If approximator_type is equal to 'reg', then ExtraTreeRegressor is constructed, 'class'
- ExtraTreeClassifier

def fit(self, input_set, output_set):
def predict(self, input_set):
Call the corresponding methods in ExtraTree classes

J. Class oneDimPlot and its child twoDimPlot

Main method:
def plotResults(self,
    my_model = None,
    trajectory = None,
    explore_actions = list([]),
    qTree = None,
    policyTree = None,
    state_space = None,
    action_space = None,
    nSam = 50,
    plot_type = 'T',
    ref_value = 0):

- my_model - an instance of the model class or its child
- trajectory - system's trajectory, including the taken actions
- explore_actions - list of exploratory actions
- qTree - list of approximators of the Q function
- policyTree - list of approximators of the policy function
- state_space - state space bounds on the policy function plots
- action_space - action set for computing the policies
- nSam - number of points in one dimension of the grid
- plot_type - plot type. The options are the same as for the input plot_type in runModel.plotResults
- ref_value - the number of the tree approximator in the list qTree (or policyTree) to plot