Artificial Intelligence: Probabilistic & learning techniques

Chris Pal

Project (INF8225)

Farnoush Farhadi & Juliette Tibayrenc

21 Apr. 2016

POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

Contents

Introduction

Our AI course has allowed us to discover Markov Decision Processes, which we decided to further explore in our project. Looking for an application for them, we stumbled upon Mastronarde & Van der Schaar's article, 'Fast Reinforcement Learning for Energy-Efficient Wireless Communications'[1], and we chose it as the basis for our work.

The article presents the two researchers' work in developing an algorithm that would enable systems to adopt the best policy to quickly send data packets in the most efficient way possible, an important goal to achieve since the majority of today's surfing is done on smartphones whose battery does not last longer than a day (at best).

In this report, we will first introduce the article, with a quick review of the state of the art at the time it was written (2011), then summarize it. We will continue with a presentation of some theoretical elements, highlighting the differences between what we saw in class and the processes on which the article and our implementations are based, and the way the reward matrix used is calculated (a process that does not appear in the article). Finally we will talk about our own implementation of classic Q-learning and one of its variants, speedy Q-learning, and compare the results we obtain with the results obtained with the authors'algorithm, plus the results we get by using value iteration and policy iteration.

1. Reference article

1.1 Literature review

Many studies have been performed on energy consumption of Wireless Sensor Networks (WSN), mainly focused on the transfer and the routing of the data. A low level model for lithium-ion batteries and a high level model for the time-varying load were proposed in [2] that require a predetermined workload profile which is not considered as a priori known. This workload evolves as a random process. In these problems, the battery behavior is treated as a stochastic process whose parameters are extracted from the electrochemical characteristics of the simulated battery. Numerous stochastic models have been proposed in the literature which uses a discrete-time Markovian chain model. In fact, this approach is a Markovian chain of the battery states of charge with forward and backward sweeps regarding to the normal discharge and recovery effect processes, respectively. Some state-of-the-art researches divided the problem of energy efficient wireless communications into two categories: physical (PHY) layer-centric solutions and adaptive modulation and coding (AMC) and system-centric solutions such as dynamic power management (DPM). These techniques can be used to trade-off delay and energy to increase the lifetime of battery-operated mobile devices by proposing a meticulous and unified framework for simultaneously utilizing physical and system level techniques to minimize power consumption subjecting to delay constraints when we have stochastic and unknown traffic and channel conditions. They formulate the power management problem as a Markov decision process and solve it using reinforcement learning method [1].

1.2 Article summary

In this project, we consider the problem of energy-efficient point-to-point transmission of delay-sensitive data in dynamic environments based on Mastronarde & Van der Schaar's paper. In such environments there are time-varying channel conditions (e.g., fading channel) and dynamic traffic loads in which the main concern is the reliable delivery of data to the receiver within a tolerable delay. The research for addressing this problem can be roughly divided into two categories: physical (PHY) layer-centric solutions such as power-control and adaptive modulation and coding (AMC) and system-centric solutions such as dynamic power management [1]. Although these techniques are different significantly, they can be applied to trade-off delay and energy to enhance the lifetime of battery consumption devices. An excessive amount of PHY-centric methods focus on optimal single-user power-control

to minimize the transmission power subject to queuing delay constraints. The main disadvantage of such methods is that they typically require statistical knowledge of the underlying dynamics like the channel state and traffic distributions, which is not always available. PHY-centric methods are effective for minimizing transmission power. But, a considerable amount of power are wasted in such systems for staying the wireless card on to be ready to transmit. To address this problem, System-level solutions are applied that rely on DPM, which enables system components to be put into low-power states when they are not required. In this paper authors consider the structure of the problem to the DPM and exhibit suboptimal learning performance. They adapt the power-control, AMC, and DPM policies according to the current and future traffic and channel conditions. Furthermore, they provide a unified framework for jointly optimizing system-level and PHY-centric power management for delay-sensitive wireless at the first time. They suggest a novel decomposition of the value iteration and Reinforcement Learning algorithms based on dividing the systemâs dynamics into a priori known and unknown elements. Using these partial system information requires to be learned less than when using conventional Reinforcement Learning algorithms and avoids the need for action exploration, which leads to speed up the adaptation and performance of existing Reinforcement Learning methods. The method does not need any priori information about the traffic arrival and channel statistics to determine the jointly optimal physical-layer and system-level power management strategies; It also applies partial information about the system in a way that less information required to be learned than when using conventional reinforcement learning algorithms; and by using this approach, we don't need for action exploration, which severely restricts the adaptation speed and run-time performance of conventional reinforcement learning methods.
We try to implement their proposed framework for simultaneously utilizing physical and system-level techniques to minimize energy consumption, under delay constraints, in the presence of stochastic and unknown traffic and channel conditions. The problem is formulated as a Markov decision process and solve it using value iteration and policy iteration methods. We aim to find the optimal policy (directly or indirectly by optimizing the value function). We also implement another solution based on Q-learning consists of dynamics and reward function initially unknown to learn the best policy from history. We compare the MDP results with that of obtained by Q-learning methods.

2. Theoretical elements

2.1 Value iterating & policy iterating vs. Q-learning

In our project, we compare the performance of several algorithms against that of the authors. These algorithms have to be evaluated along several criteria, among which the elements that need to be known for them to work, the speed with which they are executed (partly dependent on the efficency of the implementation and the language used) and the number of iterations their internal loop is executed for.

These algorithms are value iteration, policy iteration, classic Q-learning and speedy Q-learning.

2.1.1 Value iteration

One method to find an optimal policy is to find the optimal value function by determining a simple iterative algorithm called value iteration. The value iteration method is referenced by Bellman equation for MDP problems which is obtained simply by turning the Bellman optimality equation into an update rule. If there exist n possible states, then we have n Bellman equations, one per each state. The n equations contain n unknowns which are the utilities of theses states.

We aim to solve these simultaneous equations to find the utility values. These equations are non-linear, since the max operator is not a linear one. Hence, it can be determined by a simple iterative algorithm that can be shown as follows:

$$V_{i+1}(s) = R(s) + \gamma \mathrm{max}_{\mathrm{a}} \sum_{s} P(s, a, s') V_i(s')$$

where i is the iteration number. Value iteration starts at $i = 0$ and then iterates, repeatedly computing V_{i+1} for all states s, until V converges. Value iteration formally requires an infinite number of iteration to converge to V^*. In practice, it stops once the value function changes by a small amount in a sweep which satisfies the Bellman equations. This process is called a value update or Bellman update/backup. The algorithm for value iteration is as follows (from Artificial Intelligence, 3rd edition, French version [3]):

Algorithm 1 Iterative Value Algorithm

```
Data: $P(s, a, s')$ /* a transition matrix */, $a$ /* actions */, $R(s)$ /* reward signal
      */, $\gamma$ /* discount*/, $\epsilon$ /* the maximum error allowed in the utility of any
      state */
Result: $V(s)$ /* Value function */
Initialize:
$V(s) \leftarrow$ arbitrarily, for all $s \in \mathcal{S}^+$
Repeat
$V \leftarrow V'$
foreach state  $s \in \mathcal{S}^+$ do
    $V'(s) \leftarrow R(s) + \gamma \max_{\mathrm{a}} \sum_{s'} P(s, a, s')V(s')$ $\Delta \leftarrow \max(\Delta, V(s) - V'(s)|)$
end
Until $\Delta < \epsilon$
return $V(s)$
```

Note that the value iteration backup is identical to the policy evaluation except that it requires the max operator to be taken over all actions.

2.1.2 Policy iteration

In policy iteration algorithm, the agent manipulates the policy directly, instead of finding it indirectly via the optimal value function. The policy often becomes exactly optimal long before the utility estimation have converged to their true values. This phenomena leads us to another way of finding optimal policies, called policy iteration. Policy iteration picks an initial action arbitrarily by taking rewards on states as their utilities and computing a policy according to the maximum expected utility. Then it iteratively performs two steps: firstly, it determine the values through computing the utility of each state given the current action, and then policy improvement by updating the current policy if possible. To detect when the algorithm has converged, it should only change the policy if the new action for some state improves the expected value; that is, it should stabilize the policy. The algorithm for policy iteration is as follows:

Algorithm 2 Iterative Policy Algorithm

```
Data: $P(s,a,s')$ /* a transition matrix */, $a$ /* actions */, $R(s)$ /* reward signal */, $\gamma$ /* discount*/, $\epsilon$ /* the maximum error allowed in the utility of any state */
Result: $V(s)$ /* Value function */
Initialize
$V(s) \leftarrow$ arbitrarily, for all $s \in \mathcal{S}^+$
$\pi(s) \leftarrow$ arbitrarily, for all $s \in \mathcal{S}^+$
PolicyEvaluation :
$V \leftarrow V'$
foreach state $s \in \mathcal{S}^+$ do
    $V'(s) \leftarrow R(s) + \gamma \mathrm{max_a} \sum_{s'} P(s,a,s')V(s')$
    $\Delta \leftarrow \max(\Delta, V(s) - V'(s)|)$
end
until $\Delta < \epsilon$
PolicyImprovement :
stable == True
foreach state $s \in \mathcal{S}^+$ do
    $a \leftarrow \pi(s)$
    $\pi(s) \leftarrow \mathrm{argmax}_a \sum_{s'} P(s,a,s')V(s')$
    if $a \neq \pi(s)$ then
        stable == False
    end
end
```

At the end of algorithm, we test the stable variable; if it is 'True', algorithm terminates, otherwise, control jumps to the Policy Evaluation and continues in an iterative manner. Policy iteration often converges in few iterations. The policy improvement theorem guarantees that these policies works better than the original random policy and achieves to the goal states in the minimum number of steps.

2.1.3 Classic Q-learning

Q-learning is part of a second category of algorithms that do not presuppose the full knowledge of the system. Contrary to value iteration and policy iteration, the environment does not need to be known for the algorithm to work. The algorithm presented by the authors of our reference article is similar in its requirements, though it goes further and actually uses the knowledge we do have about the environment, which makes it more efficient.

Q-learning helps us find the optimal policy by using the interactions between the system and the environment. In it, we compute an estimate of the cumulative discounted reward for each action executed in each state.

The algorithm for one step is as follows (from Artificial Intelligence, 3rd edition, French version[3]:

Algorithm 3 Q-learning algorithm for one step

```
Data: $s'$ /* current state */, $r'$ /* reward signal */
Result: $Q$ /* action values table indexed by states and actions */, $T$ /* frequency
        of state-action couples */, $s, a, r$ /*previous state, action and reward */
if $s$ final then
    $Q(s, Empty) = r'$
end
if $s \neq 0$ then
    $T(s, a) + +$
    $Q(s, a) + = \alpha \cdot T(s, a)(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
end
$s, a, r = s',_{a'} f(Q(s', a'), T(s', a')), r'$
return $a$
```

2.1.4 Speedy Q-learning

Speedy Q-learning is developed in the article ‘Speedy Q-learning’ (Ghavamzadeh et al. [4]). It is, as its name indicates, a variant of Q-learning that converges faster than the form we presented in the previous subsection. We use the synchronous version, as presented in the paper, since we do have a generative model. This is, however, a restriction on the actual effectiveness of applying this algorithm that we will have to keep in mind later when comparing results.

The algorithm is presented in [4] as follows:

Algorithm 4 Speedy Q-learning

```
Data: $Q_0$ /* Initial value for Q */, $\gamma$ /* discount factor */, $T$ number of iterations,$R$
      reward matrix
Result: $Q_T$
$Q_{-1} = Q_0$;
for $k = 0 : 1 : T - 1$ do
    $\alpha_k = \frac{1}{k+1}$;
    $(x, a)$ Generate next state sample $y_k$
    ${}_kQ_{k-1}(x, a) = R(x, a) + \gamma \max_{a \in A} Q_{k-1}(y_k, a)$;
    ${}_kQ_k(x, a) = R(x, a) + \gamma \max_{a \in A} Q_k(y_k, a)$;// Empirical Bellman operator
    $Q_{k+1}(x, a) = Q_k(x, a) + \alpha_k({}_kQ_{k-1}(x, a) - Q_k(x, a)) + (1 - \alpha_k)({}_kQ_k(x, a) -_k$
    $Q_{k-1}(x, a))$;//SQL update rule
end
return $Q_T$
```

We notice that in the update rule, the second term is made more important by its learning step $1-\alpha_k$. The gist of why the speedy Q-learning algorithm is faster, or rather has a lower bound on its performance than the classic Q-learning algorithm, is shown is section 3.2 of the article: what interests us in particular, here, is that ‘Q_T converges almost surely to $Q*$ with the rate $\sqrt{1/T}$’.

2.2 Reward matrix

Our reference article skips over presenting the way the reward matrix is calculated, while it is an interesting problem. Hence, we present here the algorithm used to compute it:

```
**Algorithm 5** Computing the reward matrix
**Data**: $n \in \{0, 1, ...\}$
$l^n$ : Data arrival, i.i.d
$\text{states} = \begin{cases} b^n \in \{0, \cdots, \mathrm{B}\} : \texttt{Buffer State} \\ h^n : \texttt{Channel State} \\ x^n \in \{\text{on}, \text{off}\} : \texttt{Power Management State} \end{cases}$
$\text{actions} = \begin{cases} z^n, 0 \le z^n \le b^n : \texttt{Packet Throughput} \\ \mathrm{BEP}^n : \texttt{Bit Error Probability} \\ y^n \in \{s_{\text{on}}, s_{\text{off}}\} : \texttt{Power Management Action} \\ f^n, 0 \le f^n \le z^n : \texttt{Goodput} \end{cases}$
Initialize $b^0 \leftarrow b_{init}$

$b^n \leftarrow \min(b^n - f^n(\mathrm{BEP}^n, z^n) + l^n, \mathrm{B})$
$\mathrm{P}^x = \mathrm{P}^x(y) = [\mathrm{P}^x(x'|x, y)]_{x,x'}$
$\mathrm{P}^h = \mathrm{P}^h(h'|h)$
$\mathrm{P}^b = \mathrm{P}^b([b, h, x], \mathrm{BEP}, y, x) = \begin{cases} \sum_{f=0}^{z} \mathrm{P}^l(b' - [b - f])\mathrm{P}^f(f|\mathrm{BEP}, z), & \text{if } b' \le \mathrm{B} \\ \sum_{f=0}^{z} \sum_{l=\mathrm{B}-[b-f]}^{\infty} \mathrm{P}^l(l)\mathrm{P}^f(f|\mathrm{BEP}, z), & \text{if } b' = \mathrm{B} \end{cases}$
$s \leftarrow (b, h, x)$
$a \leftarrow (\mathrm{BEP}, y, z)$
$\mathrm{P}(s'|s, a) = \mathrm{P}^b \times \mathrm{P}^h \times \mathrm{P}^x$
```

As we can see, the reward matrix is computed from indicators of the state we are in, which depends on the buffer state, the channel state and the power management state, as well as from indicators of the actions we "take" (this is simulated data).

We find, with the reward matrix, the underlying real-life significance of the Markov process we work with in all the previous algorithms (value iteration, policy iteration, two versions of Q-learning).

3. Comparing algorithms

To compare the efficiency of the four algorithms we talked about against the efficiency of the authors' algorithm, we use Matlab to implement them and obtain graphics of the most interesting measurements.

3.1 Implementation notes

Our first two algorithms, value iteration and policy iteration, are already implemented (initially as a benchmark against which to compare the performance of the article's algorithm, constrained policy iteration). They are not part of the same category of algorithms as the two others, since the problem is not reinforcement learning. Since we use simulated data, though, we can use them as well as the reinforcement learning algorithms. Their final implementation is a bit different from the algorithms we wrote above, since we need to use more return variables than indicated here to be able to compare the algorithms.

The returned variables in the classic Q-learning and speedy Q-learning algorithms are similarly modified as well.

3.2 Results

We use several measurements to compute our results.

3.2.1 Convergence speed

We first observe the convergence speed, that we can observe easily by using the diminution of δ (maximum difference between Q_k and Q_{k-1} for Q-learning, and the Q used in this implementation of policy learning, and maximum difference between V_k and V_{k-1} for value iteration; difference between λ_k and λ_{k-1} for the authors' algorithm, all of this in absolute value).

We display it against the number of iterations it takes each algorithm to take it to near 0 (it is our convergence criterion):

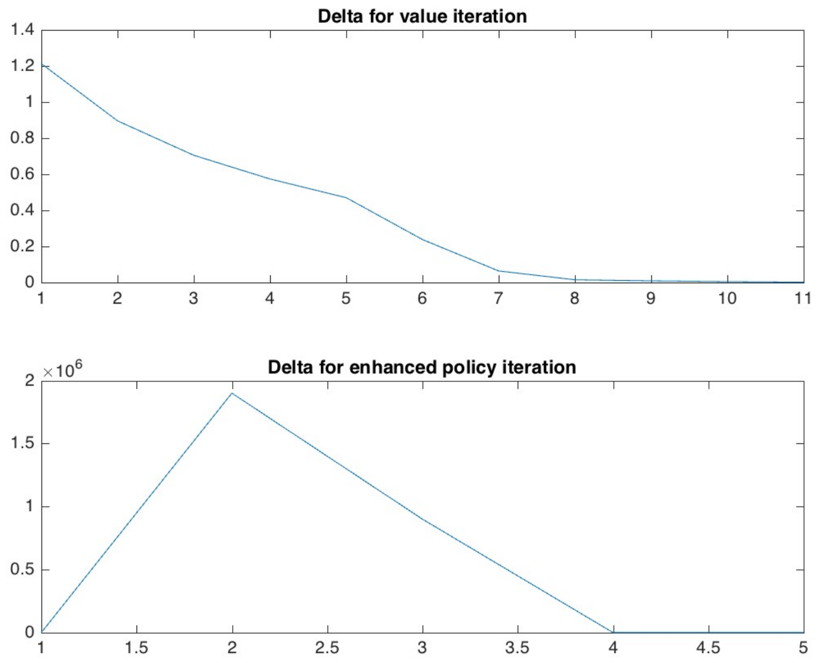

Figure 1: Deltas for value iteration and policy iteration

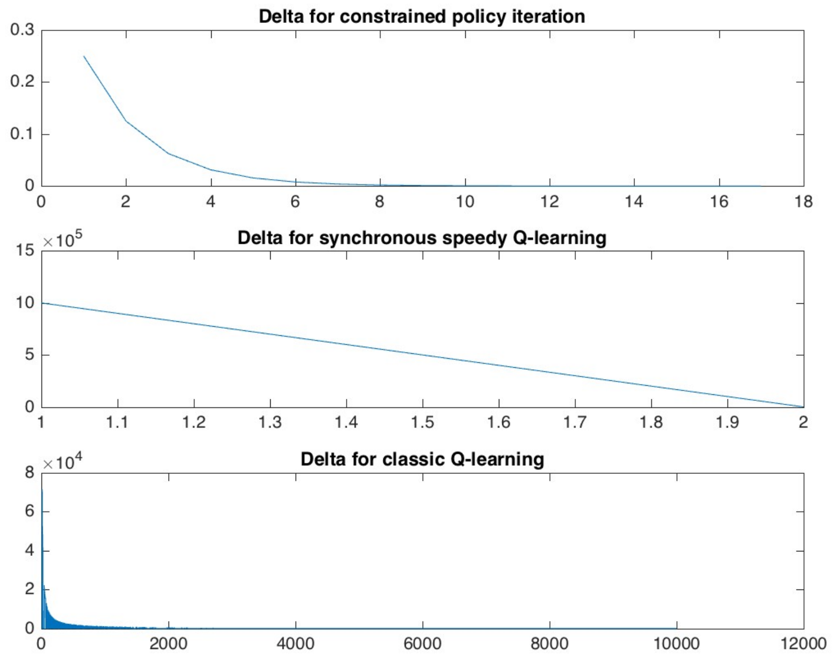

Figure 2: Deltas for constrained policy iteration, synchronous speedy Q-learning and classic Q-learning

Note: We achieve similar results when changing our parameters (in `parameters.m`; we randomized some of them).

The figures show us that, while the convergence speed is faster, in number of iterations, for synchronous speedy Q-learning, value iteration and enhanced policy iteration, they take much smaller values from the first step for constrained policy iteration, for which the curve is much smoother. Classic Q-learning performs the worst by far on both points.

3.2.2 Real time used

Measuring the time taken by each algorithm, we get the following consistent ranking:

1. Enhanced policy iteration
2. Value iteration
3. Classic Q-learning
4. Speedy synchronous Q-learning
5. Constrained policy iteration

Now, this measurement would be more significant if implemented on the platform on which these algorithms would be used (end user case), and of course tightly optimized. As it stands, it does not appear to us of high significance, except perhaps to point out that algorithms that don't have to involve full matrices tend to be faster.

3.2.3

3.3 Additional remarks

Further work & conclusion

Bibliography

[1] N. Mastronarde and M. Van der Schaar, “Fast reinforcement learning for energy-efficient wireless communication,” *IEEE Transactions on Signal Processing*, vol. 59, 12 2011.

[2] N. Salodkar, A. Bhorkar, A. Karandikar, and V. S. Borkar, “An online learning algorithm for energy efficient delay constrained scheduling over a fading channe,” *IEEE J. Sel. Areas Commun.*, vol. 26, 2008.

[3] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall Series in Artificial Intelligence, Prentice Hall, 3rd ed., 2010.

[4] M. Ghavamzadeh, H. J. Kappen, M. G. Azar, and R. Munos, “Speedy q-learning,” in *Advances in neural information processing systems*, pp. 2411–2419, 2011.