Concurrent Reinforcement Learning from Customer Interactions

David Silver D.SILVER@CS.UCL.AC.UK
Department of Computer Science, CSML, University College London, London WC1E 6BT

Leonard Newnham, Dave Barker, Suzanne Weller, Jason McFall LEONARDN@CAUSATA.COM
Causata Ltd., 33 Glasshouse Street, London W1B 5DG

Abstract

In this paper, we explore applications in which a company interacts concurrently with many customers. The company has an objective function, such as maximising revenue, customer satisfaction, or customer loyalty, which depends primarily on the sequence of interactions between company and customer. A key aspect of this setting is that interactions with different customers occur *in parallel*. As a result, it is imperative to learn online from partial interaction sequences, so that information acquired from one customer is efficiently assimilated and applied in subsequent interactions with other customers. We present the first framework for concurrent reinforcement learning, using a variant of temporal-difference learning to learn efficiently from partial interaction sequences. We evaluate our algorithms in two large-scale test-beds for online and email interaction respectively, generated from a database of 300,000 customer records.

1. Introduction

In many commercial applications, a company or organisation interacts concurrently with many customers. For example, a supermarket might offer customers discounts or promotions at point-of-sale; an online store might serve targeted content to its customers; or a bank might email appropriate customers with loan or mortgage offers. In each case, the company seeks to maximise an objective function, such as revenue, customer satisfaction, or customer loyalty. This objective can be represented as the discounted sum of a reward function. A stream of interactions occurs between the company and each customer, including actions from the company (such as promotions, advertisements, or emails) and actions by the customer (such as point-of-sale purchases, or clicks on a website).

Typically, thousands or millions of these interaction streams occur with different customers in parallel. Our goal is to maximise the future rewards for each customer, given their history of interactions with the company. This setting differs from traditional reinforcement learning paradigms, due to the concurrent nature of the customer interactions. This distinction leads to new considerations for reinforcement learning algorithms. In particular, when large numbers of interactions occur simultaneously, it is imperative to learn both *online* and to *bootstrap*, so that feedback from one customer can be assimilated and applied immediately to other customers.

The majority of prior work in customer analytics, data mining and customer relationship management collects data after-the-fact (i.e. once interaction sequences have been completed), and analyses the data offline (for example, Tsiptsis and Chorianopoulos 2010). However, learning offline or from complete interaction sequences has a fundamental inefficiency: it is not possible to perform any learning until interaction sequences have terminated. This is particularly significant in situations with high concurrency and delayed feedback, for example during an email campaign. In these situations it is imperative to learn online from *partial* interaction sequences, so that information acquired from one customer is efficiently assimilated and applied in subsequent interactions with other customers. In these cases the final outcome of the sequence is unknown, and therefore it is necessary to bootstrap from a prediction of the final outcome.

It is often also important to learn from the *absence* of customer interaction. For example, customer attrition (where a customer leaves the company due to an undesirable sequence of interactions with the company) is often only apparent after months of inactivity by that customer. Waiting until a time-out event occurs is again inefficient, because learning only occurs after the time-out event is triggered. However, the lack

Proceedings of the 30^{th} International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

of interaction by the customer provides accumulating evidence that the customer is likely to attrite. Bootstrapping can again be used to address this problem; by learning online from *predicted* attrition, the company can avoid repeating the undesirable sequence of interactions with other customers.

In addition to concurrency, customer-based reinforcement learning raises many challenges. Observations may be received, and decisions requested, asynchronously at different times for each customer. The learning algorithm must scale to very large quantities of data, whilst supporting rapid response times (for example, < 10 ms for online website targeting). In addition, each customer is only partially observed via a sparse series of interactions; as a result it can be very challenging to predict subsequent customer behaviour. Finally, there is usually a significant delay between an action being chosen, and the effects of that action occurring. For example, offering free shipping to a customer may result in several purchases over the following few days. More sophisticated sequential interactions may also occur, for example where customers are channelled through a "sales funnel" by a sequence of progressively encouraging interactions.

In this paper we formalise the customer interaction problem as *concurrent reinforcement learning*. This formalism allows for interactions to occur asynchronously, by incorporating null actions and null observations to represent the absence of interaction. We then develop a concurrent variant of *temporal-difference* (TD) learning which bootstraps online from partial interaction sequences. To increase computational efficiency, we allow decisions to be taken at any time, using an instance of the options framework (Sutton et al., 1999); and we allow updates to be performed at any time, using multi-step TD learning (Sutton and Barto, 1998). We demonstrate the performance of concurrent TD on two large-scale test-beds for online and email interaction respectively, generated from real data about 300,000 customers.

2. Prior Work

Reinforcement learning has previously been applied to sequential marketing problems (Pednault et al., 2002; Abe et al., 2002), cross-channel marketing (Abe et al., 2004), and market discount selection (Gomez-Perez et al., 2008). This prior work has found that model-free methods tend to outperform model-based methods (Abe et al., 2002); has applied model-free methods such as batch Sarsa (Abe et al., 2002) and batch Monte-Carlo learning (Gomez-Perez et al., 2008); using regression trees (Pednault et al., 2002) and neural networks (Gomez-Perez et al., 2008) to approximate the value function. However, this prior work ignored the concurrent aspect of the problem setting: learning was applied either in batch (incurring opportunity loss by not learning online), and/or by Monte-Carlo learning (incurring opportunity loss by waiting until episodes complete before learning). Our approach to concurrent reinforcement learning both learns online *and* bootstraps using TD learning, avoiding both forms of opportunity loss. In Section 5 we provide empirical evidence that both components are necessary to learn efficiently in concurrent environments.

Much recent research has focused on a special case of the customer-based reinforcement learning framework, using contextual bandits. In this setting, actions lead directly to immediate rewards, such as the click-through on an advert (Graepel et al., 2010), or news story (Li et al., 2010). A key assumption of this setting is that the company's actions do not affect the customer's future interactions with the company. However, in many cases this assumption is false. For example, advertising too aggressively to a customer in the short-term may irritate or desensitise a customer and make them less likely to respond to subsequent interactions in the long-term. We focus specifically on applications where the sequential nature of the problem is significant and contextual bandits are therefore a poor model of customer behaviour.

To avoid any ambiguity, we note that there has been significant prior work on *distributed* (sometimes also referred to as *parallel*) reinforcement learning. This body of work has focused on how a serial (i.e. continuing or episodic) reinforcement learning environment can be efficiently solved by distributing the algorithm over multiple processors. Perhaps the best known approach is distributed dynamic programming (Bertsekas, 1982; Archibald, 1992), in which Bellman backups can be applied to different states, in parallel and asynchronously; the value function is then communicated between all processors. More recently, a distributed TD learning algorithm was developed (Grounds and Kudenko, 2007). Again, this focused on efficient distributed computation of the solution, in this case applying TD backups in parallel to different states, and then communicating the value function between processors. Other work has investigated multi-agent reinforcement learning, where multiple agents interact together within a single environment instance (Littman, 1994). Our focus is very different to these approaches: we consider a single-agent reinforcement learning problem that is fundamentally concurrent (because the agent is interacting with many instances of the environment), rather than a distributed solution to a serial problem (where the agent interacts with a single instance of the environment at any given time).

We note that algorithms for concurrent reinforcement learning may themselves utilise distributed processing, however that is not the focus of this paper and is not discussed further.

3. Concurrent Reinforcement Learning

Reinforcement learning optimises long-term reward over the sequential interactions between an agent and its environment. Environments are typically divided into two categories: *episodic* and *continuing* environments (Sutton and Barto, 1998). In an episodic environment, the interaction sequence eventually terminates, at which point the environment resets and a new interaction sequence begins; whereas in continuing environments, there is a single interaction sequence that never terminates. In both cases, interactions occur serially: the agent receives an observation, takes an action, and receives a reward.

We introduce a third category for reinforcement learning environments. In a *concurrent* environment, the agent interacts in parallel with many instances of the environment. In our motivating application, the agent is a company and the environment instances are customers. At any given time, the company may be involved in interaction sequences with many different customers; furthermore new customers may arrive, or old customers may leave, at any time. This is quite different from episodic reinforcement learning, in which one customer must complete its sequence before a new customer arrives; or from continuing reinforcement learning, which considers a single customer forever. The distinct nature of concurrent environments leads to new challenges and considerations for reinforcement learning algorithms.

One challenge that arises naturally in concurrent environments is that actions and observations may occur *asynchronously*: actions may be executed, and observations or rewards received, at different times for each customer. We address this asynchronicity by representing customer interaction sequences at a fine timescale, and introducing *null* actions and observations to represent the absence of interaction with a given customer. Specifically, we define a_t^i to be the decision (or decisions) taken by the company with respect to customer i at time t. If the company does not take any action at time t, then that action is defined to be null, $a_t^i = a_\emptyset$. Similarly, we define o_t^i to be the observation of the customer (which might include actions by the customer, or any new information acquired about the customer) i at time t; if we do not observe anything for that customer, then that observation is defined to be null, $o_t^i = o_\emptyset$. Finally, we define r_t^i to be the reward for customer i at time t; in the absence of any response, the reward is defined to be zero, $r_t^i = 0$. Real (non-null) observations and real actions may occur at different time-steps, with many intervening null interactions and zero rewards. Each time-step has a duration of d; this is typically short and real interactions with each customer are typically sparse.[1] We assume that there are a total of N customers; if customers arrive in or exit the system, then their periods of inactivity are represented by null actions.

A second challenge is partial observability. Reinforcement learning often focuses on the fully observable setting where observations o_t^i satisfy the Markov property $\mathbb{P}\left(o_{t+1}^i \mid o_t^i, a_t^i\right) = \mathbb{P}\left(o_{t+1}^i \mid o_1^i, a_1^i, ..., o_t^i, a_t^i\right)$, in other words the agent is provided with full knowledge of the state of the environment. However, when interacting with a customer, the environmental state includes the latent "mental" state of the customer. Rather than attempting to model beliefs over this unwieldy environmental state – the approach taken by POMDPs (Kaelbling et al., 1995) – we represent the customer's state directly by the history of their interactions. This is, by definition, a Markov state representation: the interaction histories form an infinite, tree-structured Markov decision process with the empty history at the root, and histories of length t at depth t.

We make a simplifying assumption that customer behaviour is identically distributed and conditionally independent given their personal interaction history. In other words, customers are fully described by the observable facts about that customer: interactions with one customer do not affect the behaviour of other customers (for example via communication between customers); and unobservable variations between customers (for example due to personality or mood) are modelled as noise.[2] Informally, our goal is to maximise future rewards for a customer given the interactions with that customer. We therefore focus on predicting future rewards directly from interaction histories, rather than marginalising over latent variables that model human decision-making behaviour.

Formally, we define an interaction history to be a sequence $h_t = \{a_1, o_1, r_1, ..., a_t, o_t, r_t\}$, containing actions $a_k \in \mathcal{A} \cup a_\emptyset$, observations $o_k \in \mathcal{O} \cup o_\emptyset$, and rewards $r_k \in \mathcal{R}$. There are a total of N interaction histories, $\mathbf{h_t} = [h_t^1; ...; h_t^N]$, occurring concurrently (the superscript i is henceforth used to denote the customer, which will be suppressed when discussing a single customer; and bold font to denote the vector

[1] The continuous-time case is derived from the limit $d \to 0$; for simplicity we treat time as discrete.

[2] Note however that fine distinctions between the behaviour of individual customers may still be modelled, for example by including customer attributes such as gender or location, as an initial observation.

of all customers). The observations and rewards are assumed to be i.i.d. given their interaction history, $\mathbb{P}\left(\mathbf{o_{t+1}}, \mathbf{r_{t+1}} \mid \mathbf{h_t}, \mathbf{a_t}\right) = \prod_{i=1}^{N} \mathbb{P}\left(o_{t+1}^i, r_{t+1}^i \mid h_t^i, a_t^i\right)$. We define a *policy* $\pi(a_{t+1} \mid h_t) = \mathbb{P}\left(a_{t+1} \mid h_t\right)$ to be a single strategy that can be applied to any customer; the next action is selected according to the interaction history for that customer only. We define a *concurrent algorithm* $\boldsymbol{\pi}(\mathbf{a_{t+1}} \mid \mathbf{h_t}) = \mathbb{P}\left(\mathbf{a_{t+1}} \mid \mathbf{h_t}\right)$ to be a strategy that depends on interaction histories for *all* customers. In particular, a concurrent algorithm can apply experience gained from interactions with one customer so as to improve the policy for another customer.

The return is the discounted sum of rewards for a single customer, $R_{t:\infty} = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$, where $\gamma \in [0, 1]$ is the discount factor. This represents, for example, the total revenue from a customer. The action-value function Q^π, is the expected return for a customer, given their interaction history followed by action a, $Q^\pi(h_t a) = \mathbb{E}\left(R_{t:\infty} \mid h_t a\right)$. The *optimal value function* is the maximum action-value function achievable by any policy, $Q^*(h_t a) = \max_\pi Q^\pi(h_t a)$; our goal is to approximate the *optimal policy* $\pi^*(a_{t+1} \mid h_t)$ that achieves this maximum, i.e. the best policy given the accumulated information about that customer.

There are an infinite number of possible interaction histories, and therefore it is not feasible to represent the value function for all distinct histories. We utilise linear value function approximation, $Q^\pi(h_t a) \approx Q_\theta(h_t a) = \phi(h_t a)^\top \theta$ to estimate the value function for the current policy π, where $\phi(h_t a)$ is a $n \times 1$ feature vector, θ is an $n \times 1$ parameter vector. The feature vector $\phi(h_t a)$ summarises the relevant information about both the interaction history h_t and also the proposed action a into a compact representation.

4. Concurrent TD Learning

In concurrent reinforcement learning, the importance of bootstrapping is accentuated and it is crucial to use TD learning. However, naive application of TD(λ) is computationally inefficient and does not exploit the asynchronous structure of concurrent reinforcement learning. In this section we develop a simple variant of TD learning that is practical for large-scale concurrent reinforcement learning environments.

If the time interval d is very small, and the number of customers N is large, it is computationally inefficient to update all customers at all time-steps. Typically, little changes between time-steps in which only null actions and null observations are observed. Learning at microscopic time-scales may also be data inefficient: for example when using TD(0), backups must propagate over many time-steps, and the algorithm's behaviour depends on the choice of time interval d.

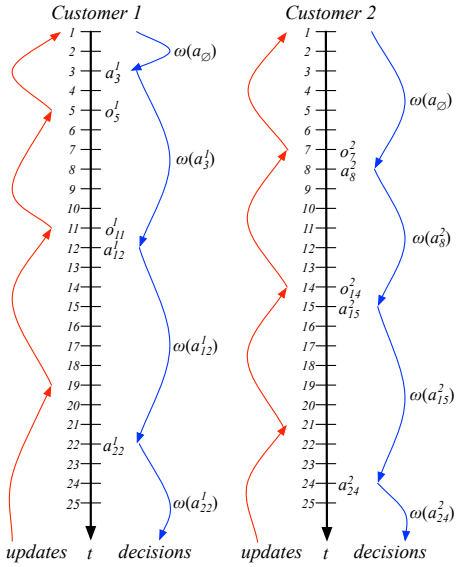


Figure 1. Concurrent TD applied to two example customers. Only real (non-null) actions and observations are shown. Curved red lines indicate concurrent TD backups; curved blue lines indicate concurrent TD options.

In our approach, the system only makes decisions at specific time-steps where a decision has been requested; and "executes" null actions (i.e. does nothing) at all other time-steps, without any further computation. Decisions may be requested for a variety of reasons: for online content serving, a real customer response will typically trigger a decision request; for email serving, a timer event might trigger a decision request (note that the decision may be a null action, i.e. choosing to do nothing). In addition, it is not necessary to perform learning updates at all time-steps, but only at those time-steps where an update has been requested. Again, updates might be triggered by real customer responses, but also by timer events. In general, our approach is fully asynchronous and there is no requirement that decisions and updates occur at the same time-step.

To implement asynchronous decision requests, we define a set of options $\{\omega(a) \mid a \in \mathcal{A}\}$, corresponding to actions $a \in \mathcal{A}$. Each option $\omega(a)$ performs the corresponding real action a exactly once, then performs null actions at all subsequent time-steps until the next decision request or a time-out event. We seek the optimal policy over these options, taking account of the randomness in the decision requests; i.e. the best we can do under the constraint that non-null actions can only be taken on time-steps when decisions are requested.[3]

[3]We note that the history $h_t a$ uniquely identifies the event of starting an option $\omega(a)$ at time-step t, and so the action-value function $Q(h_t a)$ also defines the value of each corresponding option $\omega(a)$. Unlike options in general,

To implement asynchronous update requests, we utilise multi-step TD learning (Sutton and Barto, 1998) to evaluate the current option-selection policy. K-step TD updates provide a consistent learning algorithm for any K; mixing over different K leads to the well-known TD(λ) algorithm (Sutton, 1988). In our algorithm, we apply TD updates between successive update requests at time-steps t and t'. In particular, these time-steps need not correspond to decision requests, since the value of taking an option may legitimately be evaluated by intra-option value learning at any time-steps during its execution (Sutton et al., 1999). Specifically, we perform a $t' - t$ step *Sarsa* update. This updates the action-value function $Q(h_t a_{t+1})$, towards the subsequent action-value function $Q(h_{t'} a_{t'+1})$, discounted $t' - t$ times, plus the discounted reward accumulated along the way, $R_{t:t'} = \sum_{k=0}^{t'-t} \gamma^k r_{t+k+1}$,

$$\Delta\theta = \alpha \left(R_{t:t'} + \gamma^{t'-t} Q(h_{t'} a_{t'+1}) - Q(h_t a_{t+1}) \right) \phi(h_t a_{t+1}) \tag{1}$$

The *concurrent temporal-difference learning* algorithm combines the options framework for asynchronous decision requests, with multi-step TD updates for asynchronous update requests (Figure 1). The implementation of this algorithm is straightforward (Algorithm 1): at every decision request, an action is selected by ϵ-greedy maximisation of the action-value function $Q(h_t a)$; and at every update request, the action-value function is updated according to Equation 1.

As discussed earlier, it is often desirable to learn from the absence of customer feedback. To allow for this possibility, we include updates for time-steps at which no real observation occurred (for example, the final update for each customer in Figure 1). These updates can be scheduled so as to minimise computational expense, while ensuring that learning takes place for any significant change in customer value. In the following experiments these updates are scheduled at exponentially increasing time intervals after the last real observation, eventually resulting in a time-out and a final update upon termination of the interaction sequence.

Algorithm 1 Concurrent TD

```
Initialise θ
Initialise u^i ← 0, R^i ← 0, ∀i ∈ [1, N]
for each time-step t do
  for each customer i do
    if decision requested for customer i then
      a^i_{t+1} ← ε-greedy(Q)
    else
      a^i_{t+1} ← a_∅
    end if
    if update requested for customer i then
      δ ← R^i + γ^{t−u^i} Q(h^i_t a^i_{t+1}) − Q(h^i_{u^i} a^i_{u^i+1})
      θ ← θ + αδφ(h^i_{u^i} a^i_{u^i+1})
      u^i ← t, R^i ← 0
    end if
    if reward observed for customer i then
      R^i ← R^i + γ^{t−u^i} r^i_{t+1}
    end if
  end for
end for
```

5. Experiments

To evaluate our concurrent reinforcement learning algorithms, we used a commercial simulator for internet targeting and email campaign scenarios.[4] The simulator was based on a nearest neighbour model, constructed from a database containing 300,000 anonymised customer records from an online bank. Our experiments were designed to evaluate two questions. First, how important is it to learn online, and to bootstrap, when operating at different levels of concurrency (i.e. when interacting with different numbers of customers in parallel)? Second, in a concurrent setting, how important is it to learn from delayed rewards (i.e. the full reinforcement learning problem), compared to learning from immediate rewards (i.e. the popular contextual bandit setting)? To answer these questions, we compared TD learning (which learns online from partial interaction sequences) to Monte-Carlo learning (which learns after-the-fact from complete interaction sequences) and to a contextual bandit algorithm (which learns online from immediate rewards). The simulator included eight scenarios for internet targeting and email campaigns, which varied considerably in their "sequentiality" (i.e. whether actions have long-term consequences). We investigated how the relative performance of the three algorithms changed across different levels of sequentiality. We now give details for both the simulator and algorithms.

Each customer was represented by 30 variables such as session count, page view counters, and maximum credit. In addition, time-based variables have a special importance, due to the time-dependence of many customer interactions. For example, the probability of a positive customer response is typically dependent on the recency and frequency with which the company has been interacting with that customer: too little interaction and the customer is unlikely to respond; too

which require a semi-Markov representation (Sutton et al., 1999), these simple options do not violate the Markov property since the event of starting the macro-action $\omega(a)$ at time-step t is represented in the interaction history $h_t a$. As a result, whenever a decision request is received, an option is selected simply by maximising (with some exploration) over the action-value function $a_t = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(h_t a)$.

[4] Simulator developer anonymised for blind review.

much interaction and the customer is likely to be desensitised or annoyed; similarly, the current enthusiasm of the customer is often captured by the recency and frequency with which the customer has been interacting with the company. To represent these factors, we also included two time-based variables that depend directly on the recent interaction history. One of these variables, τ_o, measured the time since the last real observation; the second variable, τ_a, measured the time since the last real action.

Each simulated customer was initialised with a real behavioural history taken from the customer database. Subsequently, the reward and transition probabilities were generated from a commercial simulator. The simulator included four internet targeting and four email campaign scenarios. In all scenarios, the agent was able to select amongst 7 real actions plus the null (do nothing) action $a_\emptyset$. Customers were limited to a binary response at each time-step (i.e. they either respond or fail to respond). The reward was $r = 1$ for a positive response and $r = 0$ for no response. The probability of response is a Bernoulli random variable with a mean that depends on all 30 customer variables, based on a nearest-neighbour representation, multiplied by a function of the two time-based variables. Customers returned after a mean time interval of 13 time-steps, but with a probability of attrition of 0.05. Once a customer attrites, she never returns (this is therefore equivalent to a discount factor of $\gamma = 0.996$). Action decisions were requested for each customer: once initially, and subsequently once each time they return.

In high-dimensional problems based on real customer data, it is common for some variables to be more significant than others; for some variables to be collinear; and for other variables to be predominantly noisy. This effect was modelled in the simulator by weighting the variables appropriately in the nearest neighbour response model. We combined two techniques to deal with the high-dimensional nature of the input.

First, each of the 30 customer variables and 2 time-based variables was discretised into twelve bins with dynamically adapted bin boundaries, using a variant of online k-means in each dimension. The feature vector $\phi(h_t a)$ contained one feature for each combination of bin and action (for customer variables), and one feature for each combination of bin and real or null action (for time-based variables) to give a total of $n = 2916$ features, each of which captures an aspect of the relationship between one variable and one action.

Second, we automatically adapted the step-size for each feature, by stochastic gradient meta-descent (Sutton, 1992; Schraudolph, 1999). The key idea of this algorithm is to adapt a *gain vector* of step-sizes, by gradient descent, so as to minimise the mean-squared prediction error. Using this algorithm, the step-size of noise variables is gradually reduced, allowing more credit to be assigned to the significant variables.

We compared three reinforcement learning algorithms, all of which used the adaptive discretisation and adaptive step-size algorithm described above. The computation required by each of these algorithms is minimal, and they are therefore suitable for large-scale implementation with rapid response times. In each algorithm we used a naive exploration policy, based on ϵ-greedy with $\epsilon = 0.1$. We note that, although more sophisticated exploration strategies have been considered in the literature (especially in the contextual bandit setting), these do not always perform significantly better in real applications, and often less robustly, than a naive ϵ-greedy algorithm (Li et al., 2010).

Monte-Carlo This algorithm learns after-the-fact from complete interaction sequences. A time-out event is generated if a customer i does not respond for 100 time-steps. No updates are performed until the time-out event occurs; at which point the value function is retrospectively updated towards the observed return: for each time-step at which a real action was selected, $\{t \mid a_t^i \neq a_\emptyset\}$, the parameter vectors are updated by linear regression, $\Delta\theta = \alpha(R^i_{t:\infty} - Q(ha^i_t))\phi(ha^i_t)$.

Contextual bandit This was a simple instance of a contextual bandit algorithm, under the simplifying assumption that actions do not affect the customer's state. This was implemented in the same way as the Monte-Carlo algorithm, but a time-out was generated before the next real action.

Concurrent TD A $(t' - t)$ step TD update was applied between successive time-steps (t, t') at which decisions were requested $\{t \mid a_t^i \neq a_\emptyset\}$, or where a time-out event has occurred. The update is identical to that described in Algorithm 1, $\Delta\theta = \alpha\left(R^i_{t:t'} + \gamma^{t'-t}Q(ha^i_{t'}) - Q(ha^i_t)\right)\phi(ha^i_t)$.

In all cases we compared a variety of constant step-size parameters α, and also an automatically adapted step-size, using a variety of meta-step-size parameters β. For each algorithm we report the results for the best-performing parameters.

We compared the three algorithms on all eight scenarios of the simulator. To evaluate the performance of the algorithms, we measured their *efficiency*. We define the efficiency $E^{\boldsymbol{\pi}}(t_1, t_2)$ of an algorithm $\boldsymbol{\pi}$ to be the mean total reward over time-period $[t_1, t_2]$, normalised such that the uniform random policy has efficiency 0

Internet				Email			
A	B	C	D	A	B	C	D
100	99	66	52	40	30	22	15

Table 1. Efficiency of the greedy oracle in each scenario. In a true bandit, the greedy oracle has efficiency 100.

and the optimal policy[5] has efficiency 100. To estimate the sequentiality of each scenario, we used an oracle that greedily maximises immediate reward, when provided with the true reward function. We measured the efficiency of this "greedy oracle", $E^{greedy}(0, T)$, for large T (Table 1). We note that Internet A scenario is a true bandit scenario (i.e. actions have no effect on customers beyond their immediate response), whereas the other seven scenarios are increasingly sequential.

Each algorithm was run approximately 100 times for each setting. In our first experiment, we measured the efficiency of the algorithms over time, using 100 concurrent customers and online updates. We measured the efficiency over windows of 1,000 time-steps, $E(t, t + 1000)$, and plotted the learning curves (Figure 2a). For the true bandit scenario, Internet A, the Contextual Bandit outperformed Concurrent TD. This is not surprising, since the contextual bandit exploits prior knowledge that the immediate response by the customer is independent of subsequent visits.[6] However, whenever there was any significant degree of sequentiality, such as the Internet C, D and email scenarios, Concurrent TD outperformed the Contextual Bandit, by learning the sequential behaviour of customers during repeated visits. The advantage of Concurrent TD over the Contextual Bandit became more pronounced with increasing levels of sequentiality. Monte-Carlo learning was more effective in the email scenario; however, it was slower to learn than Concurrent TD, due to the opportunity-loss incurred by waiting until a time-out before learning.[7]

In our second experiment, we measured the efficiency of the algorithms at different levels of concurrency. For this experiment we continued until customers had returned 100,000 times in total (requiring 100,000 decisions to be made), but we varied the number of concurrently interacting customers at any given time. In other words we maintained a constant size pool of active customers; if one customer attrited then a new customer would join the pool. A concurrency level of 1 is just like an episodic environment: the company interacts with a single customer at a time, until 100,000 decisions have been completed. At the other extreme, at a concurrency level of 10,000, there is a pool of 10,000 customers interacting in parallel, each of which returns just 10 times. We started each run with a "warm-up" period (selecting actions randomly, without any learning updates) to ensure that the customer pool included customers at a variety of different stages of interaction. The efficiency was measured over the second half of each run (Figure 2c). The results show a general degradation of performance with larger concurrency, illustrating the increased opportunity loss. In all cases, the Monte-Carlo algorithm was most sensitive to concurrency; at high concurrency levels Monte-Carlo barely outperformed the uniform random baseline. This demonstrates that after-the-fact learning incurs a prohibitively large opportunity loss at high concurrency. In contrast, the Concurrent TD algorithm is more robust at high concurrency, thanks to bootstrapping. In non-sequential scenarios such as Internet A, the Contextual Bandit was the most robust to high concurrency. However, for the non-sequential Email scenarios, the Contextual Bandit performed poorly, only occasionally stumbling on important sequences of null actions through random exploration. For all algorithms, the degradation of performance with concurrency was most apparent in the strongly sequential scenarios. In these scenarios customers must return several times before positive responses are observed, increasing the potential for opportunity loss.

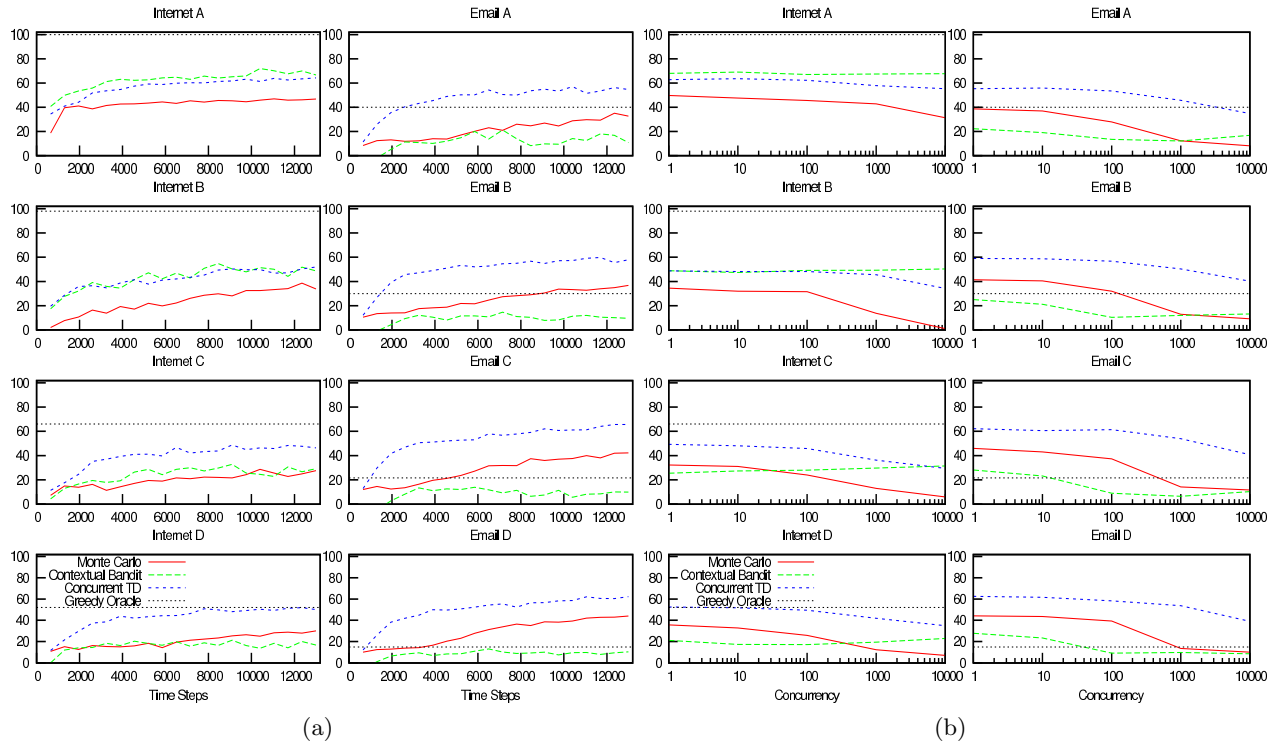
Finally, we compared the efficiency of Concurrent TD between online and batch learning. Rather than learning online at each time-step, updates were batched together and executed every m time-steps, where m is the *batch length*; $m = 1$ is closest to online learning. Efficiency was measured over the full duration of each run. The results clearly show a very significant drop in performance for higher batch lengths. Furthermore, the greater the concurrency level, the faster this drop occurred. At low concurrency, batch length was not too important; for example when the concurrency level is 1 (corresponding to an episodic environment in classic reinforcement learning) performance was maintained up to $m = 100000$. However, at high concurrency batching becomes much more problematic; for example at a concurrency level of 10,000 the performance started dropping significantly at $m = 50$. The natural conclusion is that, when interacting concurrently with large numbers of customers, it is imperative to use online updates rather than the offline updates or large batch lengths used in prior work, e.g. (Pednault et al., 2002; Abe et al., 2002; 2004).

Our experiments show the importance of bootstrapping online. We would expect similar performance for a well-tuned TD(λ) algorithm. Unfortunately, TD(λ)

[5]The simulator was designed such that the optimal policy and optimal greedy policy could both be computed.

[6]This could be implemented in Concurrent TD by $\gamma = 0$

[7]If time-out is too small, Monte-Carlo truncates sequences too early; if it's too big, opportunity loss is exaggerated; results are only reported for optimal time-out.



(a) (b)

Figure 2. a) Efficiency of algorithms over time (on a scale from random=0 to optimal=100), when applied to four internet targeting and four email campaign scenarios of increasing sequentiality, at concurrency 100. Each point represents the average result over a window of 1,000 time-steps; results were averaged over approximately 100 runs. b) Efficiency of algorithms at various levels of concurrency (number of customers interacting with the system at any given time).

requires orders of magnitude more computation, as it updates all weights at every microscopic time-step.

Finally, we note that the performance of all algorithms is significantly below the optimal policy, due largely to limitations of the linear function approximator; this could perhaps be addressed by a richer set of features.

6. Conclusion

We have introduced a framework for reinforcement learning from customer interaction histories. Its crucial component, compared to traditional reinforcement learning approaches, is that the agent interacts with many customers *concurrently*. Our solution is a temporal-difference learning algorithm that bootstraps online from concurrent interactions. Our algorithm is computationally efficient and suitable for large-scale online learning. We evaluated Concurrent TD in a high dimensional commercial simulator against non-bootstrapping (Monte-Carlo), non-online (batch TD), and non-sequential (Contextual Bandit) algorithms respectively. Our results clearly demonstrate that, in highly concurrent and sequential scenarios, it is vitally important to bootstrap from partial interaction sequences, to learn online, and to use sequential reinforcement learning algorithms. Unlike prior work, our algorithm combines all three of these key properties.

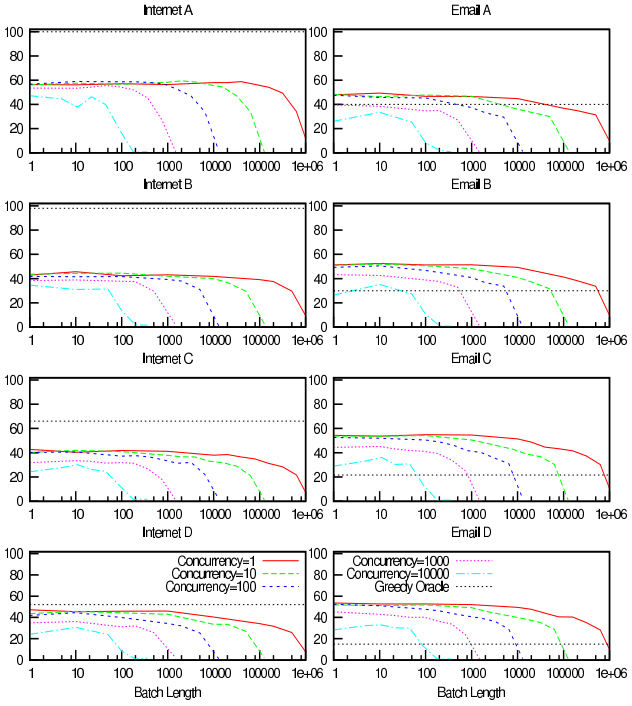


Figure 3. Efficiency of Concurrent TD at various levels of concurrency (number of customers interacting with the system at any given time) and batch lengths (number of time-steps between batch updates).

References

Abe, N., Pednault, E., Wang, H., Zadrozny, B., Fan, W., and Apte, C. (2002). Empirical comparison of various reinforcement learning strategies for sequential targeted marketing. In *International Conference on Data Mining*, pages 3–10.

Abe, N., Verma, N., Schroko, R., and Apte, C. (2004). Cross channel optimized marketing by reinforcement learning. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 767–772.

Archibald, T. (1992). Parallel dynamic programming. In Kronsjö, L. and Shumsheruddin, D., editors, *Advances in parallel algorithms*, pages 343–367. John Wiley & Sons, Inc.

Bertsekas, D. (1982). Distributed dynamic programming. *Automatic Control, IEEE Transactions on*, 27(3):610–616.

Gomez-Perez, G., Martin-Guerrero, J. D., Soria-Olivas, E., Balaguer-Ballester, E., Palomares, A., and Casariego, N. (2008). Assigning discounts in a marketing campaign by using reinforcement learning and neural networks. *Expert Systems with Applications*, (doi: 10.1016/j.eswa.2008.10.064).

Graepel, T., Candela, J. Q., Borchert, T., and Herbrich, R. (2010). Web-scale Bayesian click-through rate prediction for sponsored search advertising in microsoft's bing search engine. In *27th International Conference on Machine Learning*, pages 13–20.

Grounds, M. and Kudenko, D. (2007). Parallel reinforcement learning with linear function approximation. In *Adaptive Agents and Multi-Agent Systems*, pages 60–74.

Kaelbling, L., Littman, M., and Cassandra, A. (1995). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134.

Li, L., Chu, W., Langford, J., and Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. *CoRR*, abs/1003.0146.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *11th International Conference on Machine Learning*, pages 157–163.

Pednault, E., Abe, N., Zadrozny, B., Wang, H., Fan, W., and Apte, C. (2002). Sequential cost-sensitive decision making with reinforcement learning. In *International Conference on Knowledge Discovery and Data Mining (KDD)*.

Schraudolph, N. N. (1999). Local gain adaptation in stochastic gradient descent. In *International Conference on Artificial Neural Networks*, pages 569–574.

Sutton, R. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3(9):9–44.

Sutton, R. (1992). Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *10th National Conference on Artificial Intelligence*, pages 171–176.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: an Introduction*. MIT Press.

Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211.

Tsiptsis, K. and Chorianopoulos, A. (2010). *Data Mining Techniques in CRM: Inside Customer Segmentation*. Wiley.